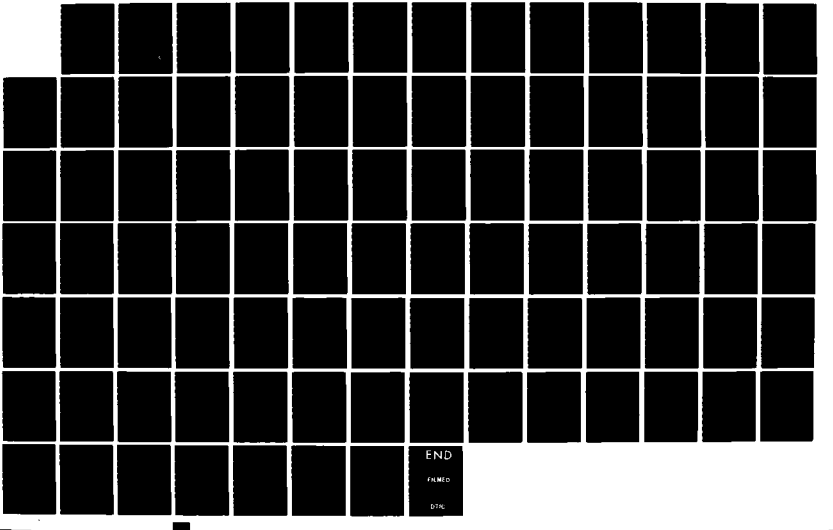
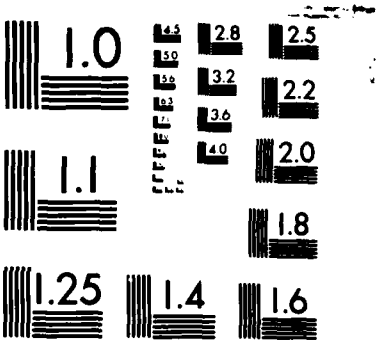
AD-A157 624 COMPUTATION OF MINIMAL ISOVIST SETS(U) MARYLAND UNIV 1/1
COLLEGE PARK CENTER FOR AUTOMATION RESEARCH
M F DOHERTY SEP 84 CAR-TR-87 DAAK70-83-K-0018

UNCLASSIFIED F/G 12/1 NL





CAR-TR-87
CSC-TR-1436

DAAK-70-83-K-0018
September, 1984

COMPUTATION OF MINIMAL ISOVIST SETS

Mark F. Doherty
Center for Automation Research
University of Maryland
College Park, MD 20742

COMPUTER VISION LABORATORY

CENTER FOR AUTOMATION RESEARCH

UNIVERSITY OF MARYLAND
COLLEGE PARK, MARYLAND
20742

2

CAR-TR-87
CSC-TR-1436

DAAK-70-83-K-0018
September, 1984

COMPUTATION OF MINIMAL ISOVIST SETS

Mark F. Doherty
Center for Automation Research
University of Maryland
College Park, MD 20742

ABSTRACT

A minimal isovist set (MIS) of a simple polygonal region P is a smallest set of points in P whose union of isovists equals P (where the isovist of x is the set of all points visible from x). This thesis presents an algorithm to search for an MIS for an arbitrary P. An MIS is shown to be equivalent to a minimal covering of P with star-shaped polygons. A (non-complete) algorithm to find a minimal covering is proposed which uses the vertices of the kernels of the star-shaped polygons. The complexity of finding an MIS is reduced to a worst-case consideration of no more than n^4 points in P. A comparison of the proposed algorithm with two previously published algorithms is made. Extension of this method to exterior views and interior holes is discussed, and areas for future research are mentioned.

The support of the U.S. Army Night Vision and Electro-Optics Laboratory under contract DAAK-70-83-K-0018, and of the Defense Advanced Projects Agency under DARPA Order 3206, is gratefully acknowledged.

1. Introduction.

This thesis addresses a general problem in computational geometry first posed by Chvatal [CHVA 75]. Suppose one is given the task of designing a computer system to guard a museum. The goal is to place video cameras at strategic points in the museum such that the entire area of the museum is within view at all times. It is very desirable to minimize expense and computer image processing requirements by placing only the minimal number of cameras in the museum. How many cameras are required, and what are the strategic points at which they must be placed?

A second example of this general problem falls under the category of Materials Inspection. Suppose XYZ Corporation wishes to automate the quality assurance inspection of some product. As the finished product rolls off the assembly line, video cameras view the product's surface, and the computer analyzes the scenes for scratches, defects, etc. It is clearly desirable to determine the minimal number of cameras necessary to do the job.

In the context of robot navigation, one can imagine giving a robot a model of the environment through which it must navigate, and requesting that it uses its sensing capabilities as little as possible in performing some reconnaissance mission in that environment.

The goal of this thesis is to find a solution to this class of problems which can then be translated into a computationally feasible computer algorithm.

2. Statement of the Problem.

The mathematical model used in this paper was developed by Davis and Benedikt [DAVI 79]. Davis and Benedikt formulated a geometric model for visibility in the two dimensional case called the "isovist", and posed this paper's problem in terms of that model.

Let us consider a simply-connected closed subset P of a plane, and also consider two points, x and y, within the subset P.

Definition 2.1. x is visible from y iff $\overline{xy} \epsilon P$; that is, the line segment $\overline{xy}$ is entirely contained within P. See Figure 2-1.

Definition 2.2. The "isovist of x", called $V_{x,P}$, or V_x if P is understood, is defined to be

$$V_{x,P} = \left\{ y \middle| y \epsilon P \ \& \ \overline{xy} \epsilon P \right\}.$$

In other words, the isovist of some point x in P is the set of all points visible from x. See Figure 2-2.

Definition 2.3. A "sufficient isovist set", or more simply a "SIS", is a set of points $S = \left\{ x_1, x_2, \cdots, x_r \right\}$ in P such that

$$\bigcup_{(k=1,r)} V_{x_k,P} = P.$$

That is, a set of points is sufficient iff all the points in P are visible from some point within the set S. See Figure 2-3.

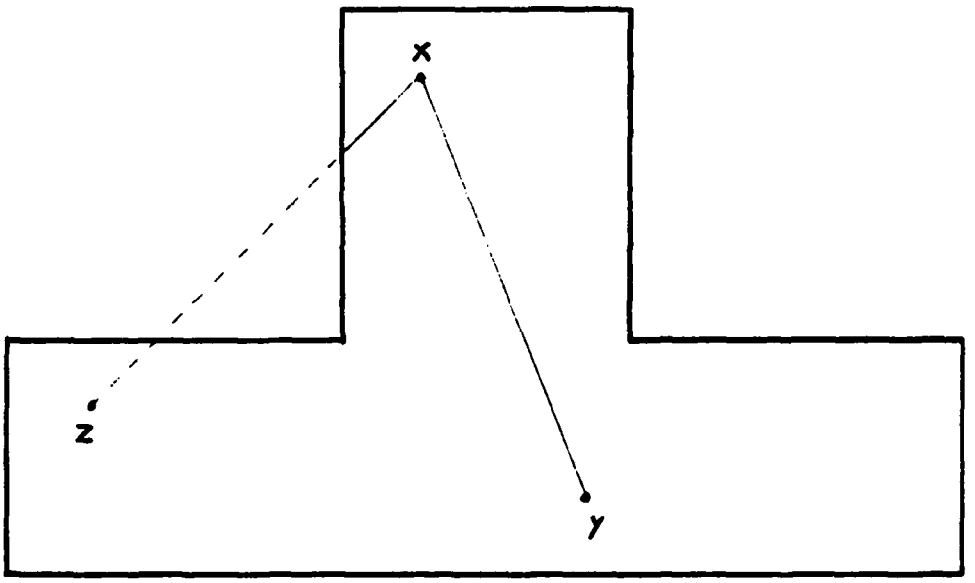


Figure 2-1.

Point y is visible from x; point z is not visible from x.

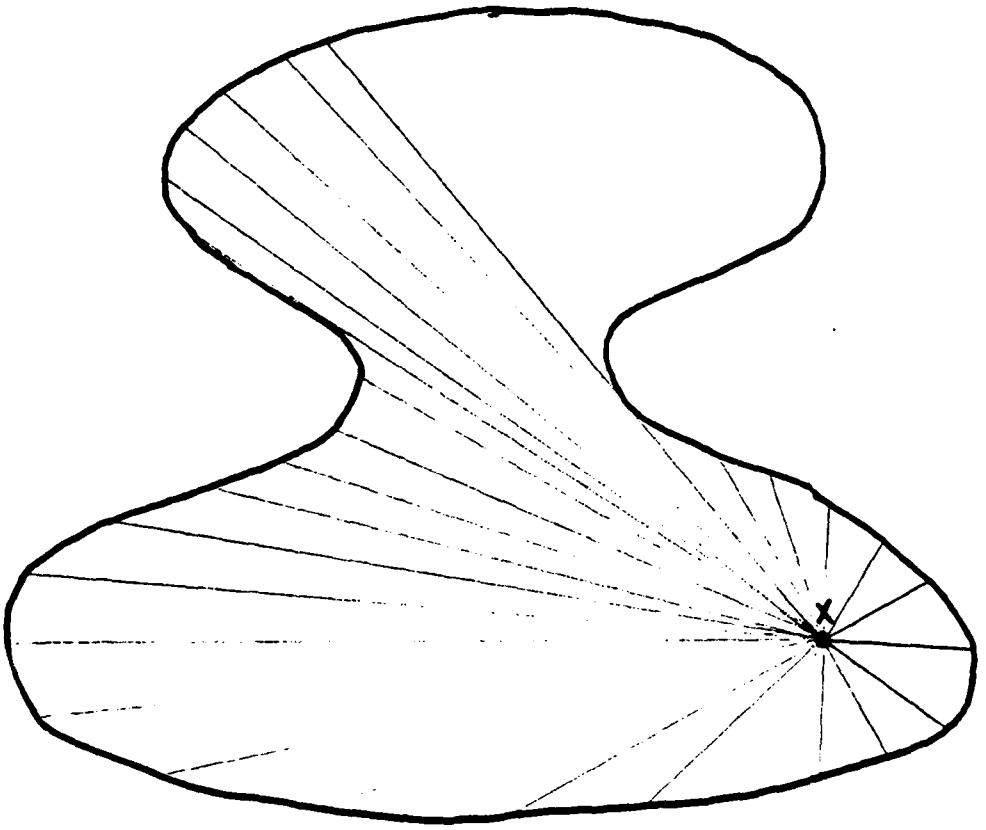


Figure 2-2.

The Isovist of x.

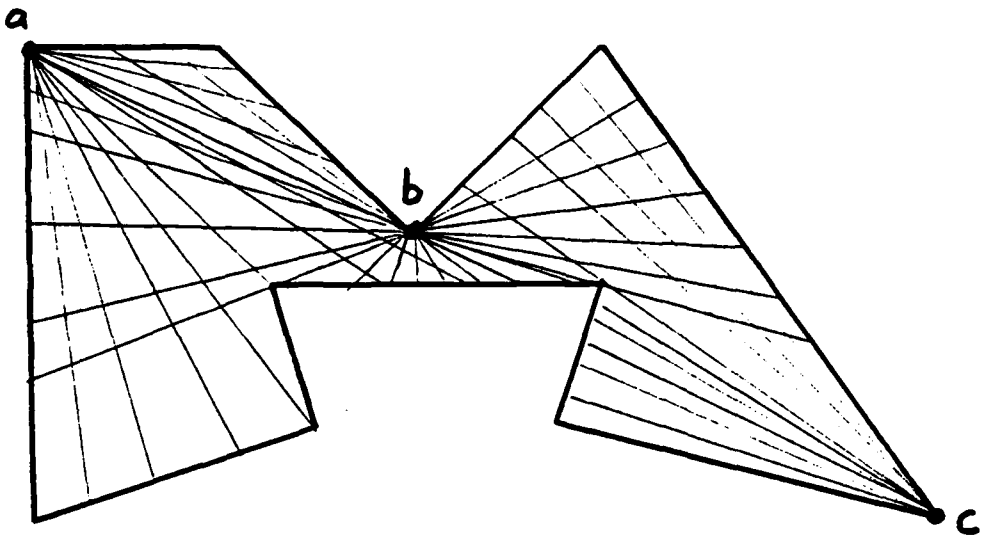


Figure 2-3.

An Example of a Sufficient Isovist Set {a,b,c}.

Definition 2.4. A "minimal isovist set", or "MIS", is a set of points $M = \{x_1, x_2, \cdots, x_i\}$ in P such that M is sufficient and $|M| \leq |Y|$, for all other sufficient sets Y in P. See Figure 2-4.

An MIS conforms precisely with the intuitive description of the paper's problem: an MIS is the smallest number of points in a region from which the entire region is visible.

What are the known facts concerning minimal isovist sets? It is known that for a finite polygonal area, at least one MIS does exist [DAVI 79]. It is not obvious how big the set will be (for an arbitrary figure), nor is it obvious which points will be members of a particular MIS. Fortunately, some partial answers to these questions have been found. V. Chvatal [CHVA 75], in his work on the "Watchman Problem", found that for arbitrary n-gons* there exists an upper bound on the MIS's cardinality, namely, $\left\lfloor \frac{N}{3} \right\rfloor$.**

It should be clear that, unlike Chvatal's upper bound, there is no non-trivial fixed lower bound for MISs. Given any n greater than 5, one can construct two n-gons (see Figure 2-5): the first n-gon is convex and thus any one point constitutes an MIS; the second has at least six of its sides structured as in Figure 2-5 so as to require a cardinality greater than 1 for its MIS.

Davis and Benedikt were unable to find an algorithm which produced a MIS for an arbitrary closed connected subset of the plane. They did, however, provide a method by which a non-trivial lower bound on the cardinality of M for polygonal regions could be obtained.

---

*For precise definition of this and other terms please consult the glossary in Appendix C.

**Historical Note: Chvatal characterized the problem in terms of graph theory. Davis and Benedikt published their geometric model some years later.

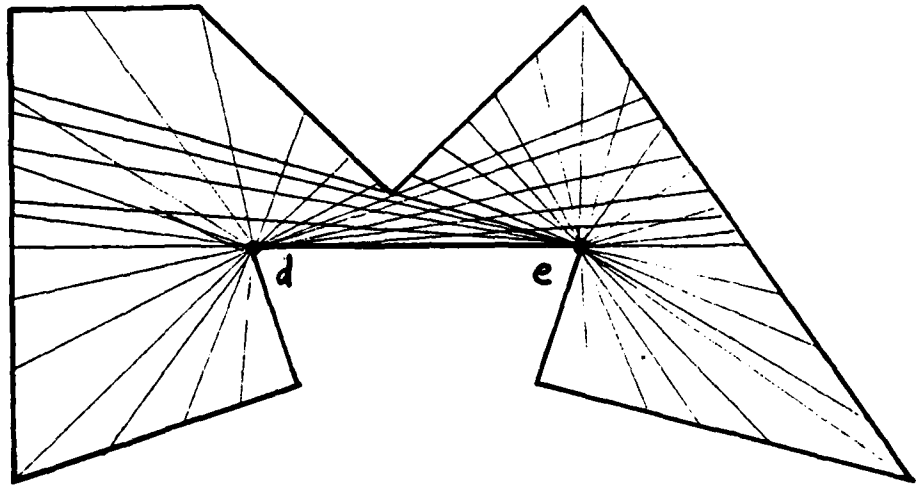


Figure 2-4.

An Example of a Minimal Isovist Set {d,e}.

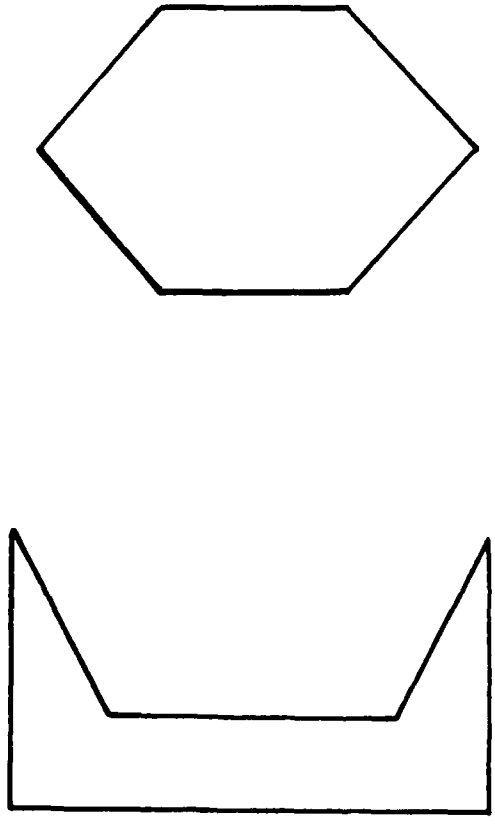


Figure 2-5.

Two 6-gons with different degrees of complexity.

3. Theorems.

Before presenting various theorems which will allow us to search for an MIS, a brief review of some of the definitions and results from computational geometry is in order. This paper will limit itself to computing an MIS for a closed subset of the Cartesian plane bounded by a sequence of n line segments which intersect at only n vertices, where $n \geq 3$. For simplicity, we shall refer to such a region as a polygon.

The kernel of a polygon (denoted K(P)) is the locus of points z such that the line segment $\overline{zp}$ is a subset of P for all points p in P [SHAM 77] (see Figure 3-1). It is a theorem that K(P) is the intersection of the interior half-planes of the edges of P [YAGL 61]. Also, K(P) can be computed in O(n) [LEE 79].

A star-shaped polygon (or star polygon) is a polygon whose kernel is non-empty; alternatively, it is a polygon such that there exists some point x in P that for all y in P, the line segment $\overline{xy}$ is a subset of P.

It is obvious that $V_{z,P}$, the isovist of z, is a star-shaped region, and that when P is a polygon, $V_{z,P}$ is a star-shaped polygon. It is also clear that any star-shaped polygon can be considered as the isovist of some point z, where z is a member of K(P). This similarity brings us to our first attempt at finding an MIS.

NON-Theorem. An MIS for a polygon determines a paritioning of P into a minimal number of star-shaped polygons, and a minimal star-shaped partition yields an MIS.

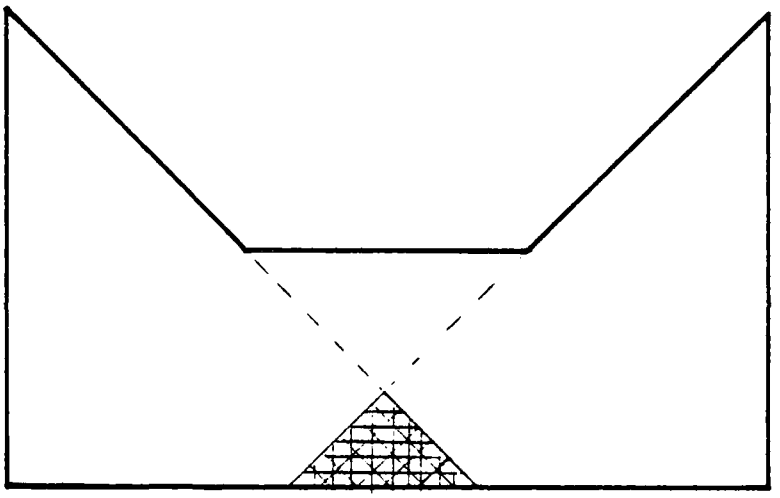


Figure 3-1.

The Kernel of a Polygon.

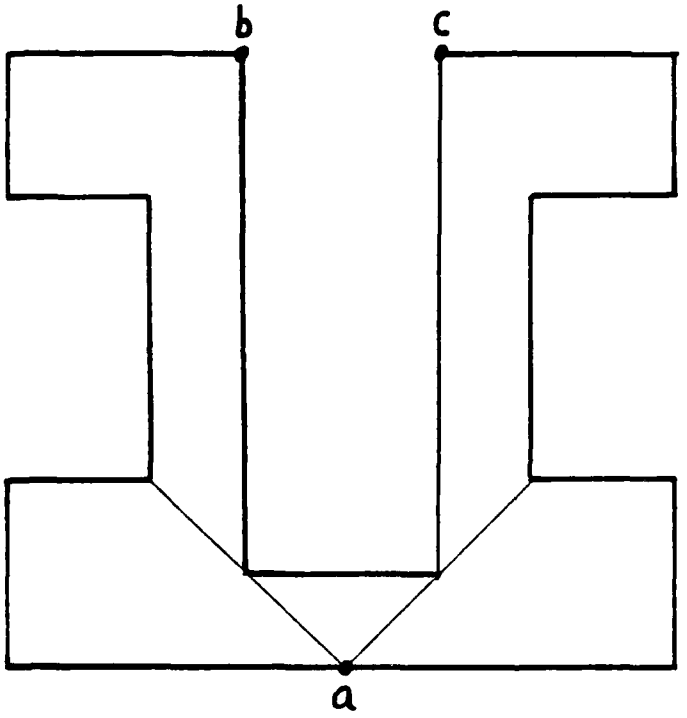


Figure 3-10.

A proof for Theorem 3.3.

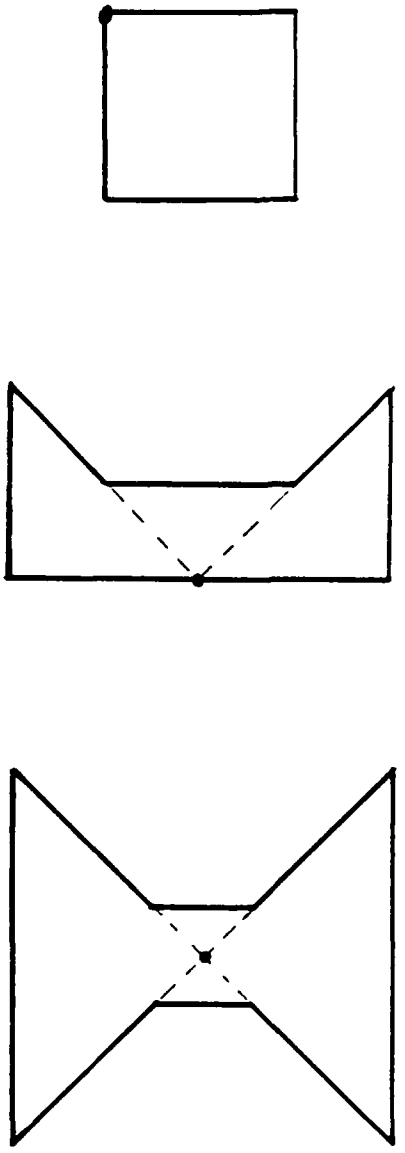


Figure 3-9.

Kernels of polygons which illustrate set C of Theorem 3.2.

q.e.d.

Examples of star polygons whose kernels consist of points from each of the three possible subsets in candidate set C are shown in Figure 3-9.

If it is the case that at least one MSC of an arbitrary (not necessarily star-shaped) polygon P has regions whose kernels all contain points in C, then one can use the set C in constructing an MIS for P. However, this is not always true.

Theorem 3.3. There exists a polygon such that, for all possible MSCs, at least one kernel in any MSC does not contain a point found in set C.

Proof: (By construction) Consider Figure 3-10. There is only one choice for the kernel of the "middle" star-shaped polygon, and that is the point a. Point a is clearly not an element of set C. q.e.d.

It is interesting to note that point a of Figure 3-10* is located at the intersection of two diagonals of P (extended into P). This observation leads to the next theorem.

Theorem 3.4. The kernel of a star-shaped polygon P always includes some point z belonging to the candidate set of points $S = \{ y \mid y$ is the intersection of two diagonals of $P, \overline{v_i v_j}$ and $\overline{v_k v_l}$, which are both wholly contained within $P \}$.

Proof: It is obvious that the set C of Theorem 3.2 is a subset of the set S. q.e.d.

*My thanks to David Harwood for suggesting this example.

The following theorem is now shown to be true.

Theorem 3.2. The kernel of a star-shaped polygon P always includes some point x belonging to the "candidate" set of points $C = \{y \mid y =$ a vertex or $y =$ the intersection of an edge of P with the interior extension of an edge of P or $y =$ the intersection of two (interior) extensions of edges of $P\}$.

Proof: By [YAGL 61], one constructively forms the kernel by intersecting the half-planes formed by the edges of P. Consider the vertices of the kernel of P.

Case 1. The kernel is a single point. Since the boundary of this kernel is the kernel itself, it is clear by Lemma 3.2 that the point is a member of the set C.

Case 2. The kernel is a line segment. Consider the two vertices of the kernel. Clearly, each vertex is by Lemma 3.2 a member of an edge or an edge extension. Now consider the edge of P which is not collinear with the kernel of P and whose half-plane boundary is closest to the vertex. Consider the shortest line segment from the vertex to the half-plane boundary, call it X. Since this boundary is closest to the vertex, there is no other half-plane boundary of P which intersects line segment X. Since the vertex is in the kernel, segment X must also be in the kernel. But the vertex is the terminating point of the kernel. Therefore, the line segment X must actually be a single point; that is, the distance from the vertex to the closest non-collinear half-plane of an edge of P is zero; that is, the vertex lies along the boundary of some half-plane of P other than the edge (or edge extension) indicated by the kernel itself. Clearly, the vertex is the intersection of one edge, or edge extension, with some other edge, or edge extension; thus, the vertex is a member of set C.

Case 3. The kernel is a convex k-gon. Consider the k vertices of the kernel. Each vertex is shown to be a member of the set C by the same reasoning as found in Case 2.

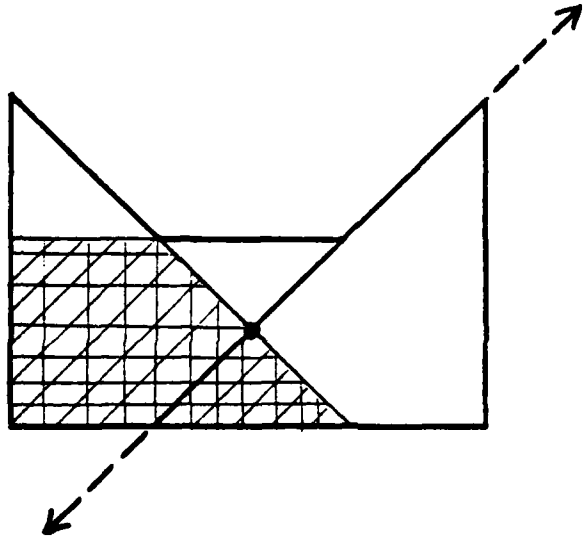


Figure 3-8.

Intersecting at an edge extension.

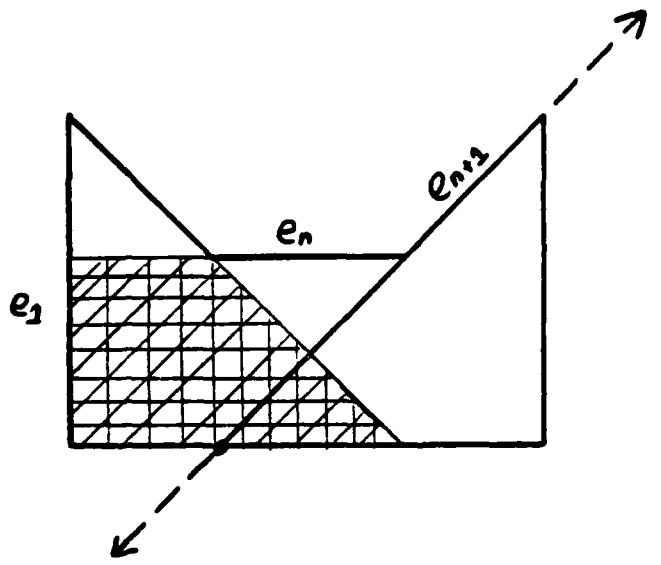


Figure 3-7.

Intersecting along an edge.

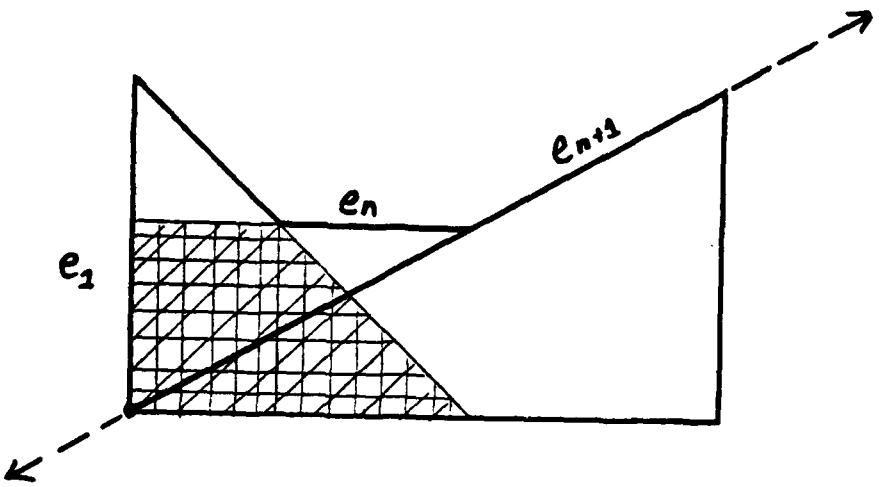


Figure 3-6.

Intersecting at a vertex (Case 3).

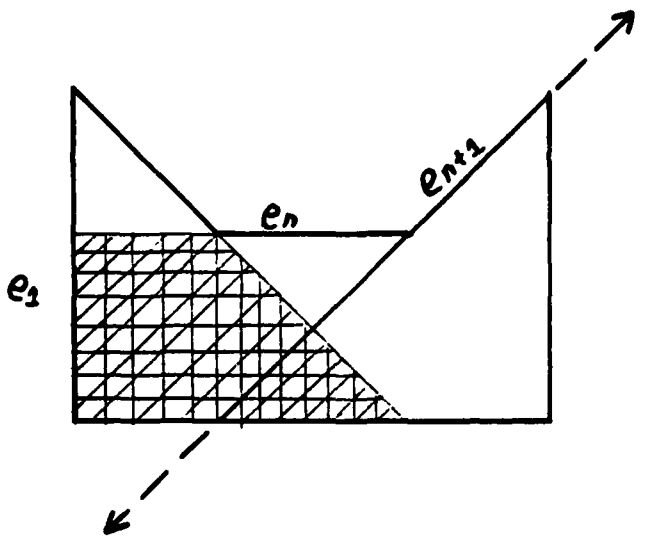


Figure 3-5.

Illustration of Case 3.

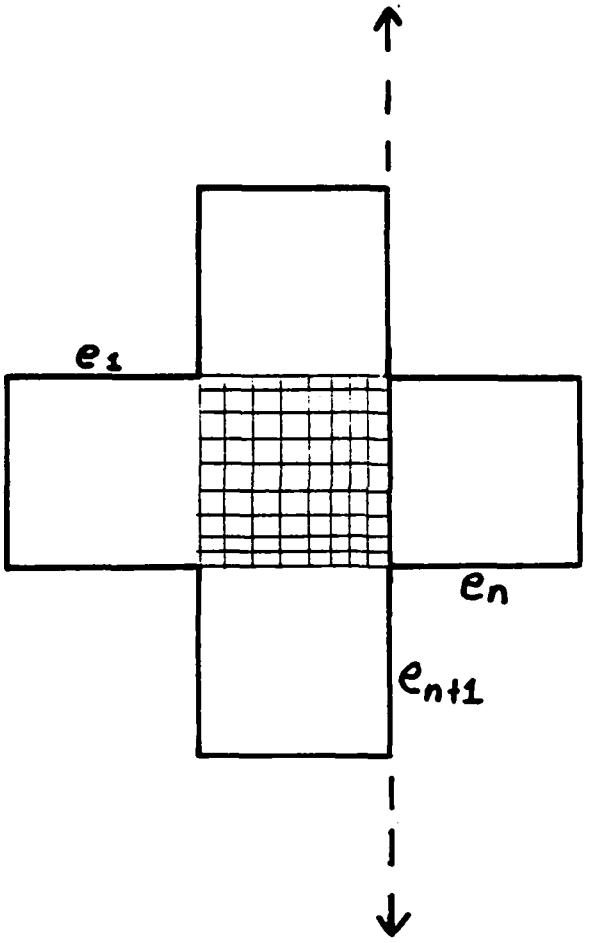


Figure 3-4.

Illustration of Case 2.

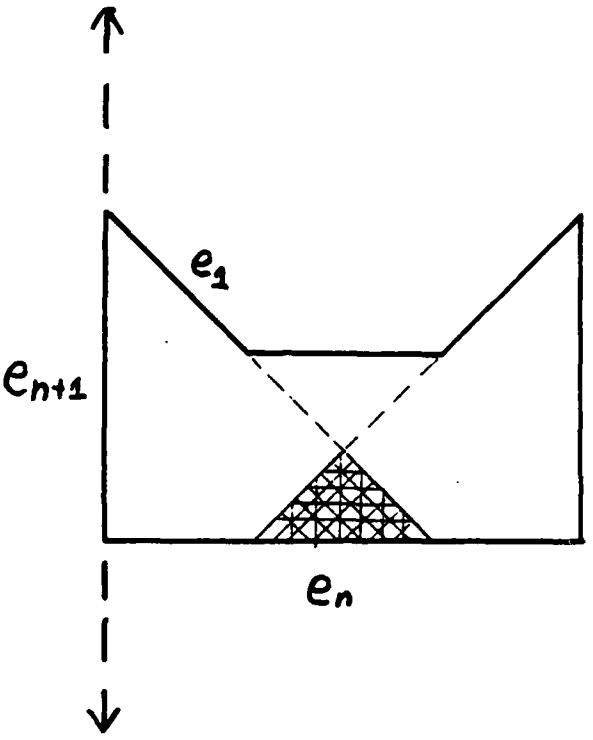


Figure 3-3.

Illustration of Case 1 (Lemma 3.2).

Case 1. The extended edge eN+1 does not intersect K(N) (see Figure 3-3). Since the interior half-plane of eN+1 must intersect K(N) (by definition of star-shaped, and by [YAGL 61]), the entire K(N) lies totally within the N+1 interior half-plane; thus, K(N+1) = K(N) and the boundary qualifies.

Case 2. The extended edge eN+1 intersects K(N) only along part of K(N)'s boundary (see Figure 3-4). Again, K(N+1) = K(N) and the boundary qualifies.

Case 3. The extended edge eN+1 intersects K(N) at non-boundary points of K(N). (See Figure 3-5).Since K(N) is convex (see [SHAM 75], [LEE 79]), eN+1 will (also) intersect two boundary points of K(N). Clearly, those edges of K(N) which are not cut by eN+1 and which lie within the eN+1 interior half-plane are boundary edges of K(N+1) and consist of qualifying points, by the induction hypopaper. Consider now one of the points along the perimeter of K(N) which intersect eN+1. If the point is a vertex (Figure 3-6), then eN+1 (as extended) becomes K(N+1)'s new edge adjacent to that vertex; both of the edges adjacent to the vertex consist of qualifying points. If the intersection point is not a vertex, then it must be on an edge or at an edge intersection. If it is on an edge (Figure 3-7), one may consider the new edge of K(N+1) as a truncated version of K(N)'s edge; by considering Lemma 3.1 it is clear that the new edge of K(N+1) still consists of qualifying points (since it lies along an edge or an extension of an edge of P). Finally, the K(N+1) edge may have its new vertex formed by the intersection of eN+1 (or its extension) with an interior extension of some edge of *P* (Figure 3-8). Both edges bounding K(N+1) at this vertex have qualifying points. q.e.d.

Since m-k is less than m, the original assumption is contradicted; therefore, the covering is also minimal. q.e.d.

The approach of this paper is to determine a well-defined subset of points in any polygon from which an MIS can always be found. When considering the simple case of a star-shaped polygon, it is obvious that any point in the kernel serves as an MIS, and that the polygon itself is an MSC. Can one describe a finite set of points which will always have at least one member in the kernel of any star-shaped polygon? If so, a consideration of this set (for any star-shaped polygon) will always yield an MIS. In order to prove this, a few lemmas must first be considered.

Lemma 3.1. A point interior to a polygon can view all points on an edge of *P* if and only if it can view the determining vertices of the edge [FREE 67].

Lemma 3.2. The boundary of the kernel of a star-shaped polygon *P* is composed of points which lie along the edges of *P* or the extensions of edges through the interior of *P*.

Proof: The proof is by induction on the half-plane intersections of *P*.

Base: The intersection within *P* of the interior half-planes of any two edges, say e1 and e2, is bounded by points belonging to e1 or e2, or by the extensions of e1 or e2 interior to *P*.

Inductive Step: Given a region within *P* formed by the intersection of n half-planes of edges of *P*, such that its boundary is composed of points which lie along the edges of *P* or their extensions interior to *P*: the intersection of the interior half-plane of some arbitrary edge eN+1 with this region results in a new region whose boundary satisfies the stated conditions. Let the original region be named K(N), and the new region K(N+1).

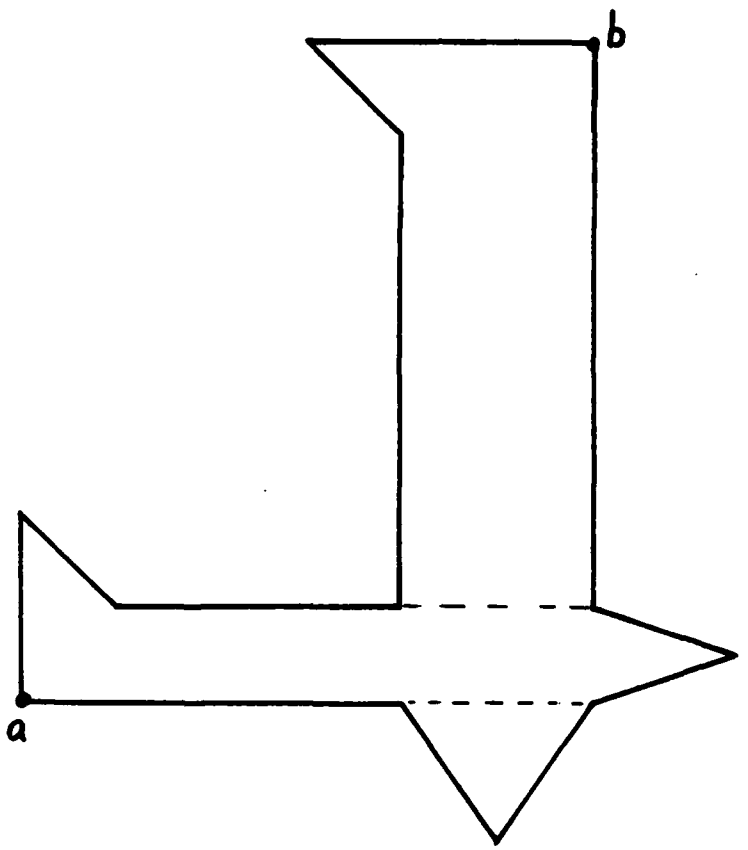


Figure 3-2.

A polygon whose minimal covering is a smaller set of pieces

than its minimal decomposition.

Counter-example: Consider Figure 3-2. Two points, a and b, are clearly an MIS; and yet it is impossible to partition P into less than three star-shaped regions.

One might instead define a "virtual decomposition", or "covering", which allows for overlap, or intersection, of the various pieces of the decomposition; this would be analogous to laying pieces of tiles on top of each other, and then viewing the resulting shape as the region formed by the (non-occluded) outlines of the tiles.

Theorem 3.1. A covering of a polygon with a minimal number of star-shaped polygons yields an MIS, and an MIS determines a minimal star-shaped covering (hereafter called "MSC").

Proof: By choosing one point from each kernel of an MSC of the polygon, one determines an SIS. This SIS is also minimal (i.e. it is an MIS); if it were not, there would be some element of SIS, say u, whose isovist V_u would be a subset of the union of the other elements' isovists. One could then cover the unnecessary element's isovist into pieces which are each totally visible from one particular element of the set SIS' = SIS - $\{u\}$. (For those points in V_u visible from more than one element of SIS', the points are assigned to all those elements which see the points.) This process constructs a new covering of the polygon consisting of the isovists of the points in SIS'. Since the cardinality of SIS' is one less than the number of pieces in our original covering, the new covering is minimal. This contradicts the original assumption. Therefore, the SIS is an MIS. Going the other way, given an MIS with m elements, we can construct m star-shaped regions (i.e. the isovists of the elements of the MIS) whose union is P. These regions form an MSC of P; if there was some other covering with fewer regions, say m-k regions, we could construct (as done in the preceding paragraph) an MIS with m-k elements.

O'Rourke [OROU 82a] has constructed a polygon which shows that S cannot be used as a basis for constructing an MIS. O'Rourke also conjectures [OROU 82b] that no bounded finite set of points can be used as a basis for MIS construction for arbitrary polygons (i.e., one can always produce some special polygon with some MSC, a kernel of which contains no points from the chosen finite set).

4fB. The Algorithm..

In order to use the preceding chapter's theorems, a small modification to the definition of "isovist" is needed.

Definition 4.1. A "restricted isovist" $V_{z,S}$ where $S = \{s_1, s_2, \cdots, s_r\}$ for $s_i \in P$, $1 \leq i \leq r$, is the subset of points (elements) of S visible from z. (Note that this differs from Definition 2.2 in that the isovist of z is taken with respect to a finite set of points in the plane, rather than with respect to a connected subset of the plane.)

The reduction of the search space from the entire polygon P to a finite set of points S is desirable (although the particular set under consideration does not guarantee minimal results in all cases). It is also desirable to reduce the extent of search computation from a consideration of regions to a consideration of points. The above definitions indicate one possible reduction which guarantees a sufficient (and sometimes minimal) result.

The following algorithm uses the candidate set S of Theorem 3.4 in determining an MIS (or SIS) for an arbitrary polygon P.

Algorithm MINIVIST.

Input: A set of vertices $\{v_1, v_2, \cdots, v_N\}$ which represent P.

Output: A set M which constitutes an MIS (or SIS) for P.

Step 0. [INITIALIZATION] Initialize S := {candidate points described in Theorem 3.4}. Initialize M := {the null set}.

Step 1. [CONSTRUCT ISOVISTS] Construct the visibility matrix V, with row headings the restricted isovists $V_{s_i,S}$ and column headings elements of S.

Fill in each element of the matrix V such that $V_{0\le i\le|S|}(V_{s_i,S},s_j) = 1$ if $s_j \epsilon V_{s_i,S}$, 0 otherwise.

Step 2. [REMOVE UNNECESSARY POINTS] Construct a new matrix L, the lattice of the visibility matrix V, by comparing rows of V. If $V_{s_i,S} \subset V_{s_j,S}$ for $i \neq j$, then remove $V_{s_j,S}$ from L. (For $V_{s_i,S} = V_{s_j,S}$, arbitrarily keep one and remove the other.)

Step 3. [DETERMINE NECESSARY POINTS] Compute the column sums for L. For all columns whose sum equals 1, find the row entry in L which equals 1, say $V_{s_i,S}$, and include s_i in the set M. Remove row $V_{s_i,S}$ from L. Remove all columns s_j from L which have $V(V_{s_i,S},s_j) = 1$.

Step 4. [COMBINATORIAL STEP] If L is empty, halt. If L has only one column, arbitrarily include one s_i from the remaining $V_{s_i,S}$ of L in M and halt. If the number of columns remaining in L is less than some threshold t, invoke a combinatorial algorithm to see what subset of remaining elements of S minimally cover the remaining columns of L. (If accuracy is desired, set t very high; if efficiency is critical, t can be low.)

Step 5. [COMPLETENESS] Determine if M covers the region P by considering triads of mutually inter-visible vertices of P. For every triad not a subset of any isovist of points in M, include one vertex from the triad in M.

Step 6. [TERMINATION] Halt.

In order to determine the merits of this algorithm, some lemmas must first be considered.

Lemma 4.1. For a star-shaped polygon, the points in set S not in K(P) do not see all points in set S.

Proof: Assume there exists some candidate point c in S such that c is not in K(P) and for all points x in S, x is visible from c. A point which views all vertices of P is a

member of K(P) [SHAM 77, pg. 118]. Since all vertices of P are in S, if c sees all points in S, it necessarily sees all vertices of P. This means that c is a member of K(P), thus contradicting our original assumption. q.e.d.

Lemma 4.2. A set of star-shaped polygons M = $\{s_1, s_2, ..., s_r\}$ (where $s_i \epsilon P$) covers P if for each triad of mutually inter-visible vertices of P there exists at least one s_i in M which contains the triad.

Proof: A triangulation of P exists [GARE 78]. Thus, every point in P is contained within some triad of mutually inter-visible vertices (by definition of triangulation). If the three vertices of a triangle are contained within (covered by) a region then all points interior to the triangle are also covered [FREE 67]. Since we posit the covering of all possible triads from some element of M, the particular triangulation developed in [GARE 78] is also covered; therefore, every point in P is covered. q.e.d.

The next two theorems will show the merits of MINIVIST from two perspectives: accuracy and efficiency. The first theorem analyzes the accuracy of MINIVIST.

Theorem 4.1. [Accuracy of MINIVIST]. The algorithm MINIVIST always produces an SIS, and sometimes produces an MIS.

Proof: The proof is a step-by-step analysis of the algorithm.

Step 0. MINIVIST initializes S to be the candidate set of points from Theorem 3.4. It is known that some subset M of S is an MIS for many polygons, and that at least one subset of S (i.e. the vertices of P) is an SIS.

Step 1. If one were to directly compute an MIS, one would need to calculate $V_{s,P}$, the isovists with respect to the polygon. However, MINIVIST attempts to calculate a minimal star-shaped cover instead. Lemma 4.1 shows that consideration of the isovists with respect to S enables one to locate the kernels of a star-shaped cover, which is

sometimes an MSC.

Step 2. This step eliminates points in S which are not kernel points of an MSC (or, in the special case, are vertices of a kernel whose representative point has already been chosen).

Step 3. If there exists a point uniquely visible from one candidate point, the candidate point must be included in the MIS being constructed.

Step 4. Combinatorial processing is invoked if there is no obvious choice of minimal isovist points.

Step 5. The preceding steps have found a covering for the finite set of points S. Step 5 ensures that the entire region of the polygon P is properly covered by the isovists of the points in M, as proven in Lemma 4.2. It is noted that Lemma 4.2 states a condition which is far stronger than that which is necessary to ensure a proper covering; that is, there exists a proper cover of some P wherein a particular triad of vertices is subsumed by none of the covering star-shaped polygons.

Step 6. Termination is guaranteed since the number of points considered is finite. q.e.d.

The next analysis considers the algorithm in terms of computational complexity. It considers only the worst-case polygon using the most optimal computing algorithms.

Analysis 4.4. Complexity of MINIVIST.

Step 0. Calculating the set S involves the determination of all diagonals of P and their intersections (within the polygon). A straightforward calculation of these intersections would be $O(n^4)$.

Step 1. The isovist matrix information was gathered in Step 0. If we assume that the compiler pre-processes the matrices to be filled with zeroes (in $O(n^4)$ preprocessing time), then the matrix V can be set during the computation of isovists in Step 0.

Step 2. Constructing a lattice matrix requires $O(k^2)$ time for a k-by-k matrix; therefore, this step requires $O(n^4)$ processing.

Step 3. Matrix sums can be calculated in $O(k^2)$; this step therefore requires $O(n^4)$ processing.

Step 4. This step results in either immediate termination (if a polynomial-time solution is found), or in a combinatorial search which will be exponential in worst case.*

Step 5. Although the isovist information has already been calculated, a worst case estimate assumes that the number of possible triads is C(n,3) or $O(n^3)$. Assuming O(n) elements of M, one arrives at $O(n^4)$ processing time in the extreme case.

Step 6. The total computation time of MINIVIST is $O(n^4)$ whenever a polynomial-time solution is available. If combinatorial processing is necessary, the algorithm is exponential.

At this point some comments are appropriate. First of all, MINIVIST uses matrix computations, which can be done very quickly on various special-purpose processors.

Secondly, the algorithm as presented can easily be modified to return, in lieu of or in addition to an MIS, the corresponding MSC. Thirdly, the kernels of the computed MSC are easily obtained during the computation**. Since any point from each of the kernels is an appropriate element for an MIS, the algorithm can be modified to return the kernels of the MSC (with the kernel represented as the vertices of its convex

*It has been reported to the author that D.T. Lee has shown the paper problem to be NP-complete

**This is done by considering those points not in the chosen MIS which have equivalent restricted isovists to some point in the MIS

polygonal border). This would afford a robot planner additional flexibility in chosing the points most desirable for viewing an entire region. This flexibility may be important for a solution to Davis and Benedikt's minimal path problem [DAVI 79].

Finally, the algorithm's efficiency may be improved by stipulating only necessary (and sufficient) conditions in Step 5 to ensure a proper covering of P.

5. Two-Dimensional Extensions.

§5.1. Polygons with Holes

We now consider the problem of finding an MIS for a broader subset of Cartesian plane regions. Recall the definition of a polygon (Section 3); there were no provisions for discontinuities within the region. The notion of a polygon is now expanded to allow the existence of "holes".

Definition 5.1. A "non-simple polygon" is a (closed) subset of the Cartesian plane bounded by one *or more* sequences of n line segments which intersect (respectively) at only n vertices, where $n \geq 3$.

Definition 5.2. A "hole", with respect to some simple polygon, is an open simple polygon whose boundary coincides with a subset of the boundary of the non-simple polygon.

The stipulation that holes are open regions fits in nicely with the standard digital image processing techniques of considering the exterior boundary of a binary image to be computationally identical with holes inside the image.

Observation 5.1. The degree of complexity of a simple polygon with one or more holes is always $m > 1$ (that is, there are no star-shaped simple polygons with holes).

Rationale: Introduce one triangular hole into a simple polygon *P*. (See Figure 5-1.)

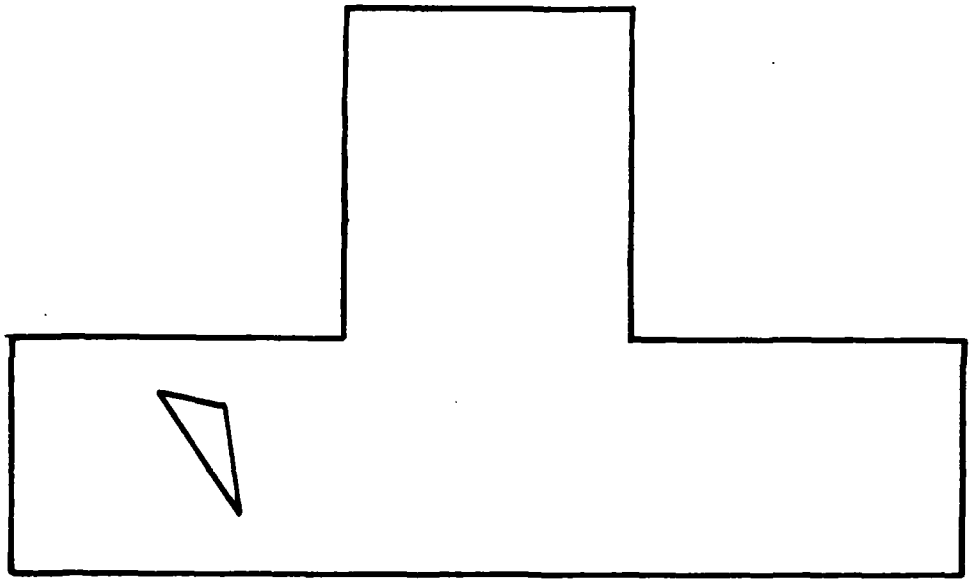


Figure 5-1.

A polygon with a triangular hole.

Regardless of which point x is chosen in P, its view $V_{x,P}$ of the rest of P is always obstructed by the hole; that is, there is always one side of the triangle whose view is occluded from x by the other two sides of the triangle. Clearly, adding more holes or enlarging the size or number of sides of the hole will not eliminate the occlusion. q.e.d.

Observation 5.1 states that the MSC of a polygon with holes will always require at least two "tiles"; alternately, there will always be at least two points within an MIS for a polygon with holes. This fact indicates that the degree of complexity has been raised by expanding the subset of regions being considered. In essence, by introducing a hole into a polygon, one increases the number of edges to be considered, and thus the candidate set S of Theorem 3.4 expands accordingly. However, the computation of an MIS and its processing order of magnitude remain the same as before, assuming one increments n to include the sides of the hole.

By expanding the definition of a polygon, the paper can now be said to have accommodated the "Watchman" real-life example in a two-dimensional setting: the simple polygon corresponds to the floor plan of the art gallery, and the holes correspond to statues and other works of art which remove small amounts of floor space from consideration.

§5.2. Exterior Views

The second real-life example is different from the first in that inspection of an object involves selecting a viewpoint exterior to the object, as opposed to within the interior of the object. In order to accommodate this distinction (again, in a two-dimensional setting), the notion of a "horizon" is introduced.

Definition 5.3. The "horizon H of radius r" for a polygon P is a sequence of four line segments meeting at four vertices

$\left\{h_1=(c_x+r,c_y),h_2=(c_x,c_y+r),h_3=(c_x-r,c_y),h_4=(c_x,c_y-r)\right\}$, where (c_x,c_y) is the centroid of P, and r is a scalar unit of distance such that $RintersectP=P$, where R is a circular region of radius r centered at (c_x,c_y).

This definition is based largely upon Davis and Benedikt's treatment of horizons [DAVI 79, pp. 54-55]. Note that the horizon can be considered as the defining boundary for a simple polygon. By treating it as such, and by letting P be considered as a "hole" of H, one can directly apply the previous discussion concerning polygons with holes to the problem of exterior views. In other words, finding MIS's for "exterior views" and "polygons with holes" are identical problems, from the perspective of the isovist model.

Computationally, there is a slight difference between the two problems. In the first real-life example (that of the art gallery), all data needed to apply MINIVIST are given a priori; in the second example, an appropriate r must be chosen (somehow) and H computed before applying MINIVIST.

This computational difference gives rise to an interesting question: does the number of points in an MIS for an exterior view depend upon the size of r (and thus indirectly upon the method used to select r)?

6. Experimental Results.

In order to determine the relative merits of the algorithm proposed by this paper, one must first establish appropriate criteria for comparing algorithms. We will use two measures of "goodness", accuracy (i.e. does the algorithm compute a minimal set?) and efficiency (i.e. how fast is it?).

The first measure is determined by comparing the subsets of possible polygons for which any two algorithms successfully find an MIS. If one algorithm can find an MIS for a larger subset of polygons than does the other, the first is considered a "better" algorithm. If each finds solutions to polygons for which the other finds no solution, then the two algorithms will be considered "incomparable".

The second measure is determined by an analysis of the order of magnitude of processing, given a polygon with N vertices. An algorithm with a smaller order of magnitude of processing than a second algorithm is considered to be "better"; two algorithms with equivalent orders of magnitude are considered "incomparable".

The first comparison is with an algorithm by Avis and Toussaint [AVIS 80],

in order to decompose a polygon by triangulation. It is first noted that the problem of finding a minimal star-shaped decomposition is slightly different from that of finding an MIS (see Figure 3-2). Chvatal's original formulation of the problem [CHVA 75] does not assume that a strict partitioning of the "art gallery" is necessary, although that is the approach he chose when solving for the upper bound $\left\lfloor \frac{N}{3} \right\rfloor$.

It is clear that MINIVIST is more accurate than Algorithm A. MINIVIST considers a larger subset of diagonals of the vertices of the polygon (viz., all of them) and will thus find an MIS whenever Algorithm A does. However, Algorithm A will never find an MIS

for the polygon in Figure 3-2, since Algorithm A partitions rather than covers the region.

It is also clear that Algorithm A is much faster than MINIVIST (even when one excludes pathological exponential cases), based on the analysis of MINIVIST in Section 4. Algorithm A is O(N log N).

The second algorithm, Algorithm B, was proposed by O'Rourke [OROU 82c] to find an MIS for rectilinear polygons. Algorithm B recursively "cuts" the polygon by extending edge diagonals of reflex vertices in a careful fashion. Since MINIVIST considers all such "cuts", it will always find an MIS whenever Algorithm B does. However, MINIVIST will also find an MIS for the rectilinear polygon in Figure 3-10, whereas Algorithm B will not. Therefore, MINIVIST is more accurate than Algorithm B.

It is difficult to compare MINIVIST's efficiency with that of Algorithm B, since no mention is made of how Algorithm B is to be computed. Although Algorithm B essentially considers only reflex vertices, it is not clear how long the "consideration" of these vertices should be. However, since the number of points considered by Algorithm B is clearly several orders of magnitude smaller than the set considered by MINIVIST, one may assume that a proper implementation of Algorithm B would most likely be more efficient than MINIVIST.

One must conclude that where accuracy is of great importance, MINIVIST is superior to other algorithms; where efficiency is critical, a different algorithm should be considered.

A test series of polygons was analyzed for minimal isovist sets using the algorithm MINIVIST. The series is a set of polygons developed by the author which have some interesting features, and which hopefully will afford some assurance that the implementation is correct.

The algorithm MINIVIST was implemented on a VAX 11/780 computer under the NIX operating system, using the "C" language. The results of the series of test lygons are tabulated below. The source code for MINIVIST is listed in Appendix B.

... means you can have up to 30 vertices
"xn" and "yn" are the coordinates of the nth or last vertex.
c) a blank line follows the last argument (i.e. there is an
extra line after the "yn" argument).

```
include files */
nclude <stdio.h>
nclude <math.h>


global data declarations */

lefine          MAXV 30
lefine MAXV2    600
lefine MAXV4    2000    /* if this isn't enough I'm  surprised! */
lefine MAXLINE 80       /* just like a punch card image, eh? */
lefine PI       3.1415927
lefine NEITHER 0        /* special constants for vertical lines */
lefine FIRST    1
lefine SECOND   2
lefine BOTH     3
lefine UPPER    0
lefine LOWER    1
lefine POSITIVE         0
lefine NEGATIVE         1
lefine REFLEX   1
lefine CONVEX 0
lefine TRACEMODE        0
lefine DEBUG            0


                static int VEXITY [2] [2] [2] = {
                {       {CONVEX, CONVEX,},      /* upper, x + y+- */
                        {REFLEX, REFLEX,},      /* upper, x - y+- */
                },
                {       {REFLEX, REFLEX,},      /* lower, x + x +- */
                        {CONVEX, CONVEX}        /* lower, x - x +- */
                }
                };


                int n;          /* number of edges in test polygon */
                int m;          /* cardinality of the minimal isovist set */
                int c;          /* cardinality of the candidate set CSET */
                int cc;         /* index into current candidate */
                int cx;         /* index into candidate visible from cur cand. */
                int un;         /* index into subsumed isovists array */
```

Appendix B. Program listing for the MINIVIST algorithm.

The algorithm MINIVIST was implemented on a VAX 11/780 under UNIX using the "C" language. The following is the source code listing for the "C" program implementing the MINIVIST algorithm.

```
    /*
File Name:      minivist.c
Author:             Mark F. Doherty
Date Created:   1/23/84
Directory:      /a/mfd (on the CVL VAX)
Version Number: 1.0
Edit Date:      4/20/84

Purpose:        This program calculates a minimal isovist set for
                    a given polygon.

Input:              Polygon vertices {v1,v2,...,vn} are manually input (in order).
Output:             1) An n-gon with its minimal isovist set displayed.
                    2) Statistical information concerning processing times
                    needed to calculate the minimal isovist set.

Remarks:        Refer to master's paper for further information concerning
                    the subject in general.

Compilation:    To compile, enter "cc /a/mfd/minivist.c -lm"
Execution:      "a.out <inputfile"
                    where "inputfile" is a text file which describes the polygon
                    to be tested, in the following format:
                    "testid
                     x1
                     y1
                     x2
                     y2
                     ...
                     xn
                     yn
                     "
                    where:
                    a) each argument is placed on a separate line.
                    b) "testid" is a 5-character or less mnemonic name for the
                    polygon being tested;
                    "x1" is the x-coordinate value (real number) of the
                    first vertex;
                    "y1" is the y-coordinate value (real number) of the
                    first vertex;
```

Appendix A. Calculation of the Candidate Set S.

Input: An arbitrary polygon P in the Cartesian plane represented by the *syntax error file techreport, between lines 1219 and 1220* real-valued coordinates for its n vertices, P = v 1 =.

Given the vertex coordinates, some simple algebra allows us to calculate the remaining points in the set S. To find the intersection of two lines which are represented as v_1v_2 and v_3v_4, one first calculates the equations which signify the two lines, $y=mx+b$. "m" is the slope and can be calculated by using the x and x coordinates of the two points given which determine the *syntax error file techreport, between lines 1229 and 1229* line: m =. Once m is found, one can substitute the x and y coordinates of either determining point to solve for b. After finding m and b for both lines, two linear equations with two unknowns are left. With some algebra, it is easy to see that the x coordinate of the point of intersection of the two lines is: $x = (b_i - b_j) / (m_j - m_i)$, where i and j are the two intersecting lines. After having solved for x, it is a simple task to find the corresponding y by using either linear equation.

It should be noted here that this representation does not work perfectly (on a digital computer) for two reason: a line might be vertical (so that its slope is infinite), and the rounding-off process used for floating point computations sometimes causes two different calculations for the same point to arrive at two slightly different points. The use of an "epsilon" resolution when comparing two computations for identity solves the latter problem. The former problem is resolved by special case processing; if a line is vertical, the x axis is fixed, and intersection and identity are easily resolved due to this additional information.

point set complexity [OROU '82b]) a finite set of degree 1 is sufficient to always find a minimal solution. Could it be the case that for any polygon of degree m, a finite set of Steiner points of degree m is sufficient for a minimal solution? This or some other relationship may or may not exist.

2. One area not well developed, both in this paper and in the literature, is the measurement of the expected or average performance of algorithms given a "random" sampling of polygons. The question of how one defines a random polygon is non-trivial. It is important, however, since the NP-completeness of this problem indicates that worst-case polygons are bound to cause poor performance. If two algorithms had similar accuracy and computational complexity (as measured by the worst case), a measurement of expected performance is greatly desired in order to judge which algorithm is better.

In addition, the question of whether a particular polygon can be solved (by MINIVIST) in polynomial or combinatorial time is of interest. If it could be shown that certain classes of polygons, perhaps detectable by a small amount of preprocessing, always yield combinatorial solutions within the context of a certain algorithm, one could then compare the expected amounts of combinatorial processing.

3. Extension of MINIVIST to a three-dimensional realm would be quite helpful in applying the results to some real-life application problems.

7. Summary and Conclusions.

7.1. Proving Minimality

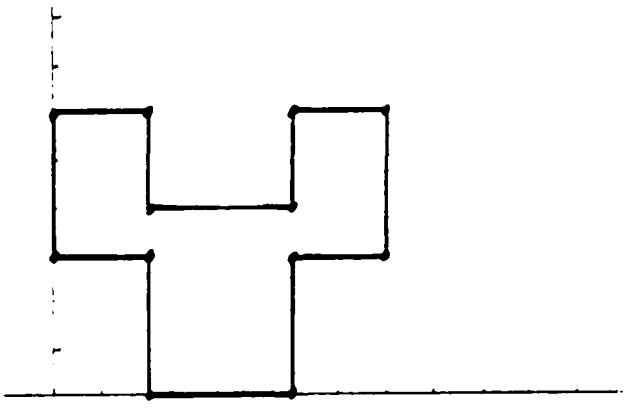
This paper has proposed an algorithm to find a minimal isovist set for a large number of polygons, and at the same time do so in as efficient a manner as possible. A complete algorithm (one which guarantees a minimal solution for an arbitrary polygon), which this paper unsuccessfully attempted to construct, seems to be achievable (albeit elusive). An approach was suggested by Maruyama [MARU 72] which would exhaustively consider convex maximal decompositions, and then use the property of disjointness of kernels of the MSC in order to construct a minimal covering. Although the efficiency of this type of algorithm may be less than that of algorithms using the approach taken in this paper, it would still be desirable to show that the minimal result can be found whenever desired.

It seems important to the author that O'Rourke's conjecture be proven (or shown false). If a trade-off must be made between efficiency and accuracy, one wishes to give away as little of each as is necessary; showing that a finite set of points can or can not be used to correctly reduce the search space would automatically place MINIVIST in a proper context.

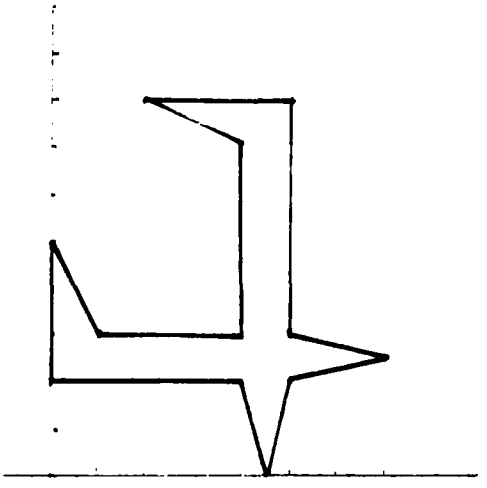
7.2. Future Research

1. The first area for future research, apart from finding a complete algorithm, is clearly the investigation of O'Rourke's conjecture. It may be the case that even though the conjecture is correct, there still exists some relationship between the degree of complexity of the polygon and the degree of complexity of the finite set of points which satisfies that polygon. In the case where the degree of complexity is 1 (i.e. star-shaped polygons), Theorem 3.2 of this paper shows that (as per O'Rourke's definition of Steiner

TEST NUMBER: A7
POLYGON: (2.00,0.00), (2.00,3.00), (0.00,3.00), (0.00,6.00), (2.00,6.00), (2.00,4.00),
(5.00,4.00), (5.00,6.00), (7.00,6.00), (7.00,3.00), (5.00,3.00), (5.00,0.00).
REFLEX VERTICES:
(2.00,3.00), (2.00,4.00), (5.00,4.00),
(5.00,3.00),.
NON-VERTEX POINTS: (6.50,6.00), (7.00,5.00), (4.00,0.00), (7.00,4.67), (0.00,5.00), (7.00,4.40),
(5.00,1.00), (7.00,4.00), (0.00,4.00), (0.00,4.40), (0.00,4.67), (0.50,6.00),
(2.00,1.00), (3.00,0.00), (2.00,3.40), (2.00,3.60), (3.06,1.41), (4.25,3.00),
(3.29,1.71), (3.71,2.29), (3.42,1.89), (4.61,3.48), (4.40,3.20), (3.50,2.00),
(5.71,4.94), (3.11,1.33), (5.46,4.15), (4.50,3.00), (3.36,1.64), (5.40,4.08),
(3.82,2.18), (3.50,1.80), (5.33,4.00), (4.86,3.43), (4.61,3.13), (3.58,1.89),
(5.24,3.88), (5.00,3.60), (5.89,4.67), (3.20,1.20), (6.50,4.50), (3.50,1.50),
(6.25,4.25), (4.00,2.00), (3.64,1.64), (6.00,4.00), (5.33,3.33), (3.71,1.71),
(5.67,3.67), (6.20,4.20), (1.00,4.50), (1.50,3.75), (1.76,3.35), (1.33,4.00),
(1.23,4.15), (1.14,4.29), (2.80,1.80), (3.50,0.75), (2.75,3.25), (2.39,3.13),
(3.87,3.63), (3.50,3.50), (2.60,3.20), (6.09,4.36), (5.86,4.29), (3.00,3.00),
(2.50,3.00), (2.75,3.00), (4.00,3.00), (0.80,4.20), (1.33,3.67), (1.67,3.33),
(1.00,4.00), (0.75,4.25), (0.50,4.50), (3.00,2.00), (3.80,1.20), (1.20,4.80),
(1.11,4.67), (0.67,4.00), (0.82,4.24), (0.91,4.36), (1.29,4.94), (1.76,3.88),
(2.50,3.50), (2.14,3.43), (3.50,3.70), (3.12,3.63), (2.39,3.48), (6.18,4.24),
(5.77,4.15), (3.50,2.50), (4.20,1.80), (1.67,4.00), (1.60,4.08), (1.54,4.15),
(3.18,2.18), (3.89,1.33), (6.33,4.00), (5.67,4.00), (4.50,3.50), (5.00,3.40),
(5.24,3.35), (4.25,3.25), (3.29,2.29), (3.94,1.41), (5.80,4.80), (5.50,3.75),
(6.00,4.50).
NUMBER OF VERTICES (N): 12
NUMBER OF POINTS IN CANDIDATE SET S: 121
MAGNITUDE OF S COMPARED WITH N: CLOSEST TO O(n squared) = 144
SOLUTION IS combinatorial. 12 combinations were evaluated to find a MIS.
NUMBER OF POINTS IN M.I.S.: 2
MINIMAL ISOVIST SET: (2.00,3.00), (5.00,3.00).

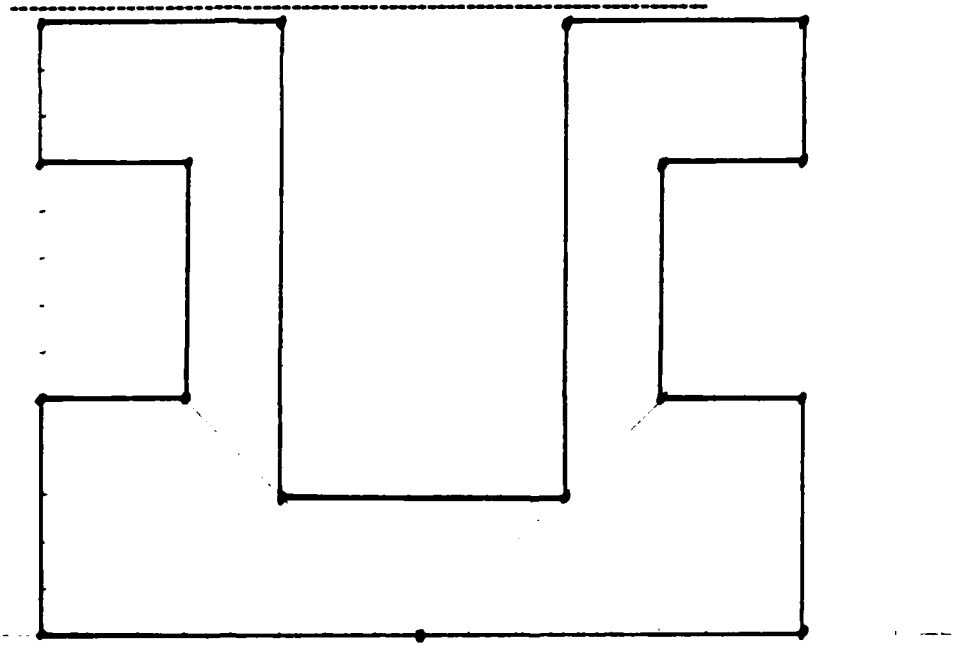


TEST NUMBER: A6
POLYGON: (0.00,2.00), (0.00,5.00), (1.00,3.00), (4.00,3.00), (4.00,7.00), (2.00,8.00),
(5.00,8.00), (5.00,3.00), (7.00,2.50), (5.00,2.00), (4.50,0.00), (4.00,2.00).
REFLEX VERTICES:
(1.00,3.00), (4.00,3.00), (4.00,7.00),
(5.00,3.00), (5.00,2.00), (4.00,2.00).
NON-VERTEX POINTS: (4.00,6.00), (5.00,3.25), (1.50,2.00), (0.00,3.00), (0.00,3.08), (0.00,3.25),
(4.92,1.69), (0.00,3.33), (4.00,8.00), (3.80,2.00), (5.00,6.50), (5.24,2.06),
(3.75,8.00), (3.80,8.00), (3.93,8.00), (4.00,3.25), (4.06,1.76), (5.24,2.94),
(3.75,3.00), (1.33,2.33), (3.25,2.81), (2.50,2.63), (2.29,2.57), (4.94,3.24),
(4.76,3.19), (4.28,3.07), (4.70,3.17), (4.17,3.04), (4.50,3.13), (3.76,2.94),
(1.36,2.27), (3.82,2.76), (2.78,2.56), (2.50,2.50), (3.96,2.79), (4.55,2.91),
(4.17,2.83), (4.03,2.81), (4.00,2.80), (4.81,2.96), (4.30,2.86), (4.68,2.94),
(4.14,2.83), (3.81,2.76), (1.45,2.10), (3.89,2.28), (3.29,2.24), (3.86,2.28),
(4.67,2.33), (4.12,2.29), (4.00,2.29), (5.16,2.37), (4.93,2.35), (4.34,2.31),
(5.00,2.36), (4.65,2.33), (4.05,2.29), (4.89,2.35), (4.31,2.31), (5.09,2.36),
(3.93,2.28), (4.17,2.00), (4.36,2.00), (4.63,2.00), (4.83,2.00), (4.80,3.00),
(4.29,3.00), (4.69,3.00), (4.17,3.00), (3.95,2.75), (4.27,2.73), (4.04,2.75),
(4.00,2.75), (5.09,2.66), (4.86,2.68), (4.31,2.72), (5.00,2.67), (4.67,2.69),
(4.12,2.74), (4.95,2.67), (4.69,2.69), (5.16,2.65), (3.81,2.77), (3.86,2.29),
(4.13,2.22), (4.00,2.25), (4.35,2.16), (4.63,2.09), (4.04,2.24), (4.84,2.04),
(4.20,2.20), (4.60,2.10), (3.93,2.27), (3.81,2.06), (4.18,1.94), (4.37,1.88),
(4.61,1.80), (4.79,1.74), (4.43,1.86), (4.73,6.64), (4.44,5.22), (4.40,5.00),
(4.21,4.05), (4.05,3.24), (3.89,2.44), (5.04,2.83), (4.83,2.86), (4.29,2.95),
(5.00,2.83), (4.68,2.89), (4.16,2.97), (4.97,2.84), (4.86,2.86), (5.20,2.80),
(4.31,2.69), (4.65,2.35), (4.14,2.86), (4.86,2.14), (4.50,2.50), (4.08,2.50),
(4.14,2.14), (4.20,1.80), (4.16,2.03), (4.91,6.55), (4.77,6.62), (4.75,4.00),
(4.50,5.00), (5.20,2.20), (5.13,2.50), (4.71,3.43), (4.45,4.73), (4.79,3.05),
(4.91,2.45), (4.83,2.83), (4.97,2.16), (4.25,3.50), (4.27,3.18), (4.33,2.33),
(4.37,1.84), (4.35,2.06), (5.00,2.17), (4.69,3.08), (4.67,2.67), (4.62,1.90),
(4.63,2.11), (4.20,3.20), (4.83,1.96), (5.04,2.17).
NUMBER OF VERTICES (N): 12
NUMBER OF POINTS IN CANDIDATE SET S: 160
MAGNITUDE OF S COMPARED WITH N: CLOSEST TO O(n squared) = 144
SOLUTION IS combinatorial. 1 combinations were evaluated to find a MIS.
NUMBER OF POINTS IN M.I.S.: 2
MINIMAL ISOVIST SET: (1.00,3.00), (4.00,7.00).



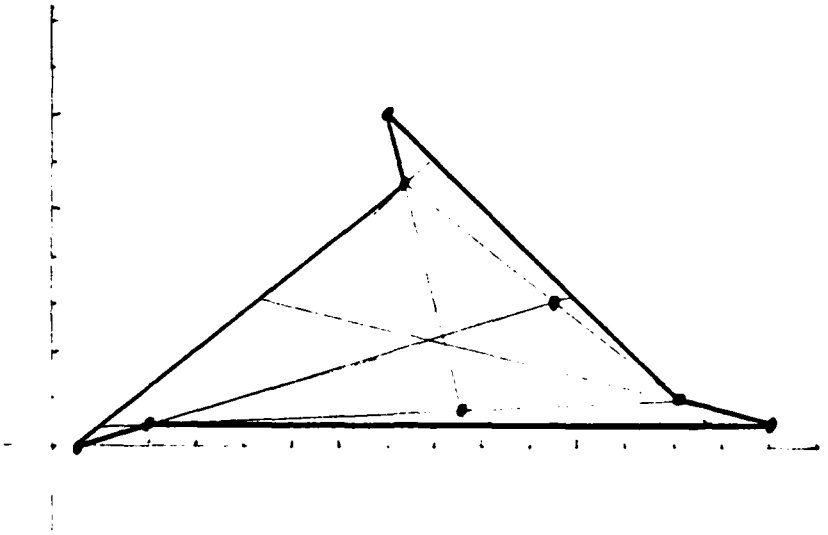
TEST NUMBER: A5
POLYGON: (0.00,0.00), (0.00,5.00), (3.00,5.00), (3.00,10.00), (0.00,10.00), (0.00,13.00),
(5.00,13.00), (5.00,3.00), (11.00,3.00), (11.00,13.00), (16.00,13.00), (16.00,10.00),
(13.00,10.00), (13.00,5.00), (16.00,5.00), (16.00,0.00).
REFLEX VERTICES: (3.00,5.00),
(3.00,10.00), (5.00,3.00), (11.00,3.00),
(13.00,10.00), (13.00,5.00),

NON-VERTEX POINTS: (5.00,8.33), (16.00,4.36), (5.00,5.00), (12.50,0.00), (3.00,13.00), (3.00,0.00),
(1.75,0.00), (8.00,0.00), (5.00,10.00), (5.00,8.00), (5.86,0.00), (2.14,13.00),
(5.00,0.00), (16.00,3.00), (0.00,3.00), (0.00,4.36), (11.00,0.00), (13.86,13.00),
(10.14,0.00), (3.50,0.00), (14.25,0.00), (11.00,8.00), (11.00,10.00), (13.00,0.00),
(13.00,13.00), (11.00,5.00), (11.00,8.33), (2.42,4.03), (4.88,8.13), (3.97,6.61),
(1.80,3.00), (2.25,3.75), (3.00,1.80), (2.06,1.24), (7.43,2.03), (3.00,0.82),
(1.88,0.51), (6.29,1.71), (5.43,1.48), (5.00,1.36), (8.00,2.18), (13.34,3.64),
(13.00,3.55), (13.75,3.75), (4.43,5.00), (3.00,3.80), (2.73,3.91), (11.00,0.60),
(10.38,0.85), (9.29,1.29), (8.00,1.80), (3.00,11.80), (3.00,3.00), (3.00,3.55),
(4.25,10.00), (4.00,9.00), (3.67,7.67), (2.50,3.00), (2.66,3.64), (6.71,1.29),
(1.88,11.13), (2.56,11.54), (5.62,0.85), (5.00,0.60), (13.50,3.00), (13.00,3.00),
(14.20,3.00), (11.00,1.36), (10.57,1.48), (9.71,1.71), (8.57,2.03), (14.12,0.51),
(13.00,0.82), (13.44,11.54), (12.33,7.67), (11.57,5.00), (12.03,6.61), (13.27,3.91),
(13.00,3.80), (13.58,4.03), (13.94,1.24), (13.00,1.80), (14.13,11.13), (13.00,11.80),
(12.00,9.00), (11.75,10.00), (11.13,8.13).
NUMBER OF VERTICES (N): 16
NUMBER OF POINTS IN CANDIDATE SET S: 103
MAGNITUDE OF S COMPARED WITH N: CLOSEST TO O(n log (n squared)) = 88
SOLUTION IS combinatorial. 308 combinations were evaluated to find a MIS.
NUMBER OF POINTS IN M.I.S.: 3
MINIMAL ISOVIST SET: (3.00,10.00), (13.00,10.00), (8.00,0.00).



0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 1 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1

SOLUTION IS combinatorial. 1 combinations were evaluated to find a MIS.
NUMBER OF POINTS IN M.I.S.: 2
MINIMAL ISOVIST SET: (10.27,3.26), (8.43,0.79).



TEST NUMBER: A4

POLYGON: (0.50,0.00), (7.30,5.70), (7.00,7.00), (13.00,1.00), (15.00,0.50), (2.00,0.50).

REFLEX VERTICES:

(7.30,5.70), (13.00,1.00), (2.00,0.50).

NON-VERTEX POINTS: (7.84,6.16), (10.63,3.38), (8.50,0.50), (13.61,0.50), (7.80,6.20), (13.50,0.50), (4.29,3.18), (1.04,0.46), (1.10,0.50), (8.04,2.51), (10.27,3.26), (7.57,2.36), (8.10,2.22), (8.43,0.79), (4.64,3.09).

NUMBER OF VERTICES (N): 6

NUMBER OF POINTS IN CANDIDATE SET S: 21

MAGNITUDE OF S COMPARED WITH N: CLOSEST TO O(n log (n squared)) = 21

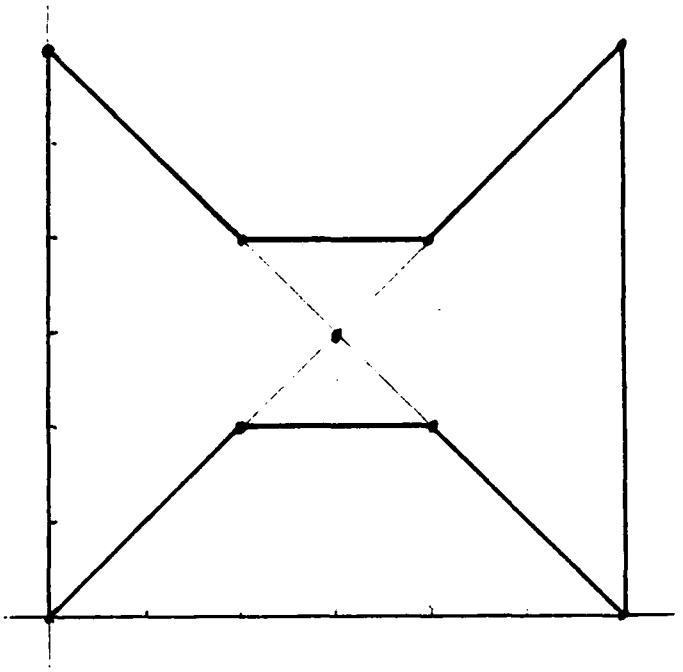
Visibility Matrix V =

1	1	0	0	0	1	1	1	0	0	0	0	1	1	1	1	1	1	0	0	1
1	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
0	1	1	1	0	0	1	1	1	0	1	1	0	0	0	1	1	0	1	1	0
0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
0	0	0	1	1	1	0	0	1	1	0	1	1	0	1	0	0	1	1	1	1
1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	0	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	0	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1
0	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1
0	1	0	1	1	1	0	0	1	1	0	1	1	0	1	1	1	1	1	1	1
0	1	1	1	0	1	1	1	1	0	1	1	0	0	0	1	1	1	1	1	1
0	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1
1	1	0	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1
1	1	0	1	0	1	1	1	0	0	0	0	1	1	1	1	1	1	1	1	1
1	1	0	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1
1	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

Lattice Matrix L of V:

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
SOLUTION IS polynomial.
NUMBER OF POINTS IN M.I.S.: 1
MINIMAL ISOVIST SET: (3.00,3.00).



TEST NUMBER: A3

POLYGON: (0.00,0.00), (0.00,6.00), (2.00,4.00), (4.00,4.00), (6.00,6.00), (6.00,0.00), (4.00,2.00), (2.00,2.00).

REFLEX VERTICES: (2.00,4.00), (4.00,4.00), (4.00,2.00), (2.00,2.00).

NON-VERTEX POINTS: (6.00,4.00), (0.00,4.00), (0.00,2.00), (6.00,2.00), (1.50,3.00), (1.00,2.00), (3.00,3.00), (1.00,4.00), (5.00,4.00), (4.50,3.00), (5.00,2.00).

NUMBER OF VERTICES (N): 8

NUMBER OF POINTS IN CANDIDATE SET S: 19

MAGNITUDE OF S COMPARED WITH N: CLOSEST TO O(n log n) = 16

Visibility Matrix V =

1	1	1	1	1	0	0	1	0	1	1	0	1	1	1	1	0	0	0
1	1	1	0	0	1	1	1	0	1	1	0	1	1	1	1	0	0	0
1	1	1	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1
1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	0	0	1	1	1	1	1	1	0	0	1	0	0	1	0	1	1	1
0	1	0	1	1	1	1	0	1	0	0	1	0	0	1	0	1	1	1
0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1
0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1
0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1
0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

Lattice Matrix L of V:

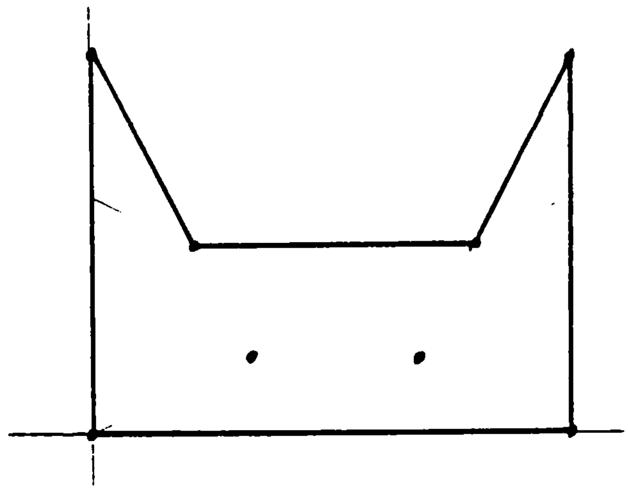
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

TEST NUMBER: A2
POLYGON: (0.00,0.00), (0.00,4.00), (1.00,2.00), (4.00,2.00), (5.00,4.00), (5.00,0.00).
REFLEX VERTICES:
(1.00,2.00), (4.00,2.00),.
NON-VERTEX POINTS: (5.00,2.50), (2.00,0.00), (5.00,2.00), (0.00,2.00), (0.00,2.50), (3.00,0.00), (1.60,0.80), (2.50,1.25), (3.40,0.80).
NUMBER OF VERTICES (N): 6
NUMBER OF POINTS IN CANDIDATE SET S: 15
MAGNITUDE OF S COMPARED WITH N: CLOSEST TO O(n log n) = 10
Visibility Matrix V =

1	1	1	1	0	1	1	1	1	1	1	1	1	1	1
1	1	1	0	0	0	0	1	0	1	1	0	1	0	0
1	1	1	1	0	1	0	1	1	1	1	1	1	1	1
1	0	1	1	1	1	1	1	1	1	0	1	1	1	1
0	0	0	1	1	1	1	0	1	0	0	1	0	0	1
1	0	1	1	1	1	1	1	1	1	1	1	1	1	1
1	0	0	1	1	1	1	1	1	0	0	1	1	1	1
1	1	1	1	0	1	1	1	1	1	1	1	1	1	1
1	0	1	1	1	1	1	1	1	1	0	1	1	1	1
1	1	1	1	0	1	0	1	1	1	1	1	1	1	1
1	1	1	0	0	1	0	1	0	1	1	1	1	1	1
1	0	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	0	1	1	1	1	1	1	1	1	1	1
1	0	1	1	0	1	1	1	1	1	1	1	1	1	1
1	0	1	1	1	1	1	1	1	1	1	1	1	1	1

Lattice Matrix L of V:

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	1	1	1	0	1	1	1	1	1	1	1	1	1	1
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	1	1	1	1	1	1	1	1	1	1	1	1	1



SOLUTION IS polynomial.
NUMBER OF POINTS IN M.I.S.: 2
MINIMAL ISOVIST SET: (1.60,0.80), (3.40,0.80).

TEST NUMBER: A1
POLYGON: (0.00,0.00), (0.00,2.00), (2.00,2.00), (2.00,0.00).
REFLEX VERTICES:

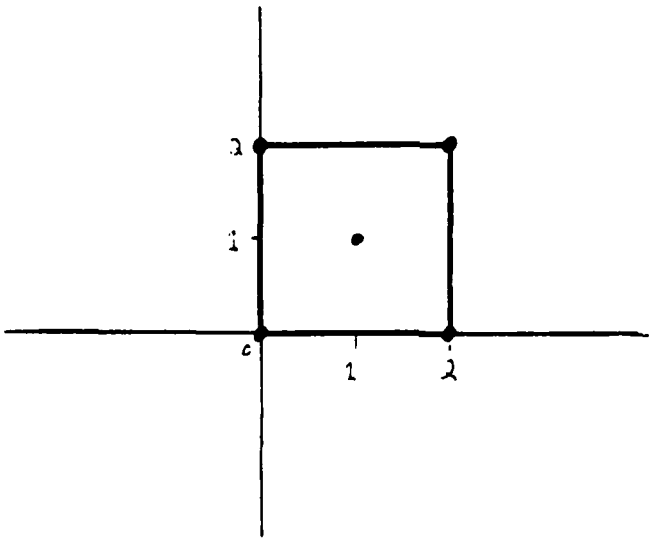
NON-VERTEX POINTS: (1.00,1.00).
NUMBER OF VERTICES (N): 4
NUMBER OF POINTS IN CANDIDATE SET S: 5
MAGNITUDE OF S COMPARED WITH N: CLOSEST TO O(n log n) = 5
Visibility Matrix V =

```
1 1 1 1 1
1 1 1 1 1
1 1 1 1 1
1 1 1 1 1
1 1 1 1 1
```

Lattice Matrix L of V:

```
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
1 1 1 1 1
```

SOLUTION IS polynomial.
NUMBER OF POINTS IN M.I.S.: 1
MINIMAL ISOVIST SET: (1.00,1.00).



```
int temp;       /* temporary storage for any procedure */
int nzr;/* number of non-zero rows in L */
int nzc;        /* number of non-zero columns in L */
int nc;         /* number of combinations tried */
long magnitude;       /* orders of magnitude of size of candidate set */
float ftemp;    /* temporary floating point storage */
float epsilon;  /* precision of resolution allowed in this system */
char test_id [6], order [30], complexity [16];

struct point {
        float x; /* x-coordinate of point */
        float y; /* y-coordinate of point */
        int is_reflex;   /* flag to indicate convexity or reflexivity*/
        }
         P[MAXV],    /* Polygon represented by vertices */
         CSET [MAXV2],      /* Candidate set of points */
         MIS [MAXV/3],      /* Minimal Isovist Set */
         pt;            /* test point */

int V [MAXV2] [MAXV2] = {{0,0,0},};/* init to zeroes */
int L [MAXV2] [MAXV2] = {{0,0,0},};/* init to zeroes */
int LSAVE [MAXV2] [MAXV2] = {{0,0,0},}; /* ditto */
int C [MAXV2] [MAXV2] = {{0,0,0},};/* Compressed L matrix */

int UNNECESSARY[MAXV2];       /* array of subsumed isovist(indexes)*/
int LINDEX [MAXV2];           /* indexes of nzr's per row of C */
int COMBO [MAXV2];            /* current combination to try */


/* declaration of non-integer functions */
float getslope ();
float getintercept ();
float getangle ();
float unitlength ();
float distance ();
double calc_possible_combos ();
double factorial ();




/* main program */
main ()
{


                /* get vertices of polygon from user */
#if TRACEMODE
```

```
                printf ("\nstarting minivist...");
#endif
                initialize ();          /* make all sets empty */
                getngon ();             /* ask operator to enter test polygon */
                minivist ();            /* calculate the minimal isovist set */

                /* now print out the results */
                print_results ();

}

initialize ()
{
                /* set all cardinality index variables to zero, thus
#if TRACEMODE
                printf ("\nhere at init ");
#endif
                setting all sets = the null set. */
                m = 0;          /* for the MIS */
                n = 0; /* for the polygon P */
                c = 0; /* for the candidate set CSET */
                un = 0; /* for the set of subsumed isovists */
                epsilon = 0.00001;      /* resolution for floating point calcs */
}


getngon ()
{
                /* local noun declarations */
                char inbuf [MAXLINE];

                /* get the test identification string */
#if TRACEMODE
                printf ("\ngetting the polygon vertices now...");
#endif
                getline (inbuf, MAXLINE);
                sscanf (inbuf, "%5s", &test_id[0]);
                /* input up to MAXV vertices */
                while (n < MAXV && getline (inbuf, MAXLINE) > 0) {
                        sscanf (inbuf, "%4f", &P[n].x);
                        CSET[n].x = P[n].x;
                        if (getline (inbuf, MAXLINE) > 0) {
                                sscanf (inbuf, "%4f", &P[n].y);
                                CSET[n].y = P[n].y;
                                }
                        else break;
                        ++n;
                        ++c;
                        }       /* end of while loop */
```

```
                    calc_reflex_vertices ();
#if TRACEMODE -
                    printf ("\n %d vertices grabbed",n);
#endif

}


calc_reflex_vertices ()
{
                    /* this routine determines if the vertices of the polygon are
                    reflex or convex */

                    /* local data definitions */
                    int testv, basev, nextv;
                    struct point test, next;
                    float deltax, deltay, slope, xintercept, yintercept;
                    int half, xsign, ysign;


                    for (testv = 0; testv < n; testv++) {
                            basev = (testv + n - 1) % n;
                            nextv = (testv + 1) % n;
                            /* translate base vertex to origin */
                            deltax = 0.0 - P[basev].x;
                            deltay = 0.0 - P[basev].y;
                            test.x = P[testv].x + deltax;
                            test.y = P[testv].y + deltay;
                            next.x = P[nextv].x + deltax;
                            next.y = P[nextv].y + deltay;
                            /* check for vertical edge */
                            if (fabs (test.x) <= epsilon) {
                                    /* edge is vertical - special case */
                                    if (test.y > 0.0) {
                                            /* in upper half of plane */
                                            P[testv].is_reflex =
                                                    (next.x>0.0)? CONVEX : REFLEX;
                                            continue;
                                    }
                                    else {
                                            /* lower half of plane */
                                            P[testv].is_reflex =
                                                    (next.x>0.0)? REFLEX : CONVEX;
                                            continue;
                                    }
                            }
                            /* not vertical - calculate slope */
                            slope = test.y / test.x;
                            yintercept = next.y - (slope*next.x);
```

```
                        /* check for horizontal edge */
                        if (fabs (test.y) <= epsilon) {
                                /* edge is horizontal - special case */
                                if (test.x > 0.0) {
                                        P[testv].is_reflex =
                                                (yintercept>0.0)? REFLEX : CONVEX;
                                        continue;
                                }
                                else {
                                        P[testv].is_reflex =
                                                (yintercept>0.0)? CONVEX : REFLEX;
                                        continue;
                                }
                        }
                        /* can now proceed for "normal" cases */
                        xintercept = (-yintercept) / slope;
                        half = (test.y>0.0)? UPPER : LOWER;
                        xsign = (xintercept>0.0)? POSITIVE : NEGATIVE;
                        ysign = (yintercept>0.0)? POSITIVE : NEGATIVE;
                        P[testv].is_reflex = VEXITY [half] [xsign] [ysign];
                }
}


minivist ()
{
                /* step 0: initialize the minimal set and candidate set */
                /*  calculate the (non-vertex) candidate set of points */
#if TRACEMODE
                printf ("\nhere at minivist");
#endif
                calc_set_S ();          /* step 0 */
                calc_isovists ();       /* step 1 */
                calc_lattice (); /* step 2 */
                find_minimals ();       /* steps 3 and 4 */

                /* step 5 : return to main and print results */
}



calc_set_S()
/* this function computes the points in set S */
{
                /* For each diagonal of P, determine if it lies wholly within P.
                (i.e. it is not occluded, nor is it exterior to P). If so, try
                to extend the diagonal into P. If it can be extended, include
                the point of intersection of the extended diagonal with the boundary
                of P. Also, try intersecting all diagonals which lie interior to P.
```

```
                    If an intersecting point also lies interior to P, include it also. */

                    /* local data definitions */
                    int v1, v2;     /* bounding vertices of diagonal being considered */
                    int d1, d2;     /* current diagonals being intersected */
                    int d;          /* index into array of saved diagonals */
                    struct point DIAG [2] [MAXV4];     /* diagonals interior to P */

                    /* start of code for calc_set_S */
#if TRACEMODE
                    printf ("\ncalc_set_s");
#endif
                    d = 0; /* initialize index into array */
                    for (v1 = 0; v1 < n; v1++) {
                            /* ignore next adjacent vertex - cannot extend an edge */
                            for (v2 = v1+1; v2 < n; v2++) {
#if TRACEMODE
                                    printf ("\nv1=%d,v2=%d",v1,v2);
#endif
                                    /* if this diagonal is occluded or lies exterior to P,
                                    don't use it */
                                    if (is_occluded (P[v1],P[v2]) != NULL) continue;
                                    if (is_exterior (P[v1],P[v2]) != NULL) continue;
                                    /* okay - this is an interior diag - save it */
                                  if (d >= MAXV4) printf ("\nFATAL ERROR diag no room");
                                    DIAG[0][d].x = P[v1].x;
                                    DIAG[0][d].y = P[v1].y;
                                    DIAG[1][d].x = P[v2].x;
                                    DIAG[1][d].y = P[v2].y;
                                    /* now - can we extend this diag inside of P? */
                                    if (P[v2].is_reflex == REFLEX) {
                                            if (is_extensible (P[v1],P[v2]) != NULL) {
                                                    DIAG[1][d].x = pt.x;
                                                    DIAG[1][d].y = pt.y;
                                                    /* yes - include bounding point */
                                                    insert_point_into_cset (pt);
                                            }
                                    }
                                    if (P[v1].is_reflex == REFLEX) {
                                            if (is_extensible (P[v2],P[v1]) != NULL) {
                                                    DIAG[0][d].x = pt.x;
                                                    DIAG[0][d].y = pt.y;
                                                    insert_point_into_cset (pt);
                                            }
                                    }
                                    ++d;
                            }
                    }
                    /* all diag's have been analyzed; now look for intersection points
```

```
                of interior diagonals */
#if TRACEMODE
                printf ("\nintersect diags with d=%d", d);
#endif
                for (d1 = 0; d1 < d; d1++) {
                        for (d2 = d1+1; d2 < d; d2++) {
#if TRACEMODE
                                printf ("\nd1=%d,d2=%d",d1,d2);
#endif
                                if ((line_intersect(DIAG[0][d1],DIAG[1][d1],
                                        DIAG[0][d2],DIAG[1][d2]) != NULL)
                                && (is_interior (DIAG[0][d1],DIAG[1][d1],pt)!=NULL)
                                && (is_interior(DIAG[0][d2],DIAG[1][d2],pt)!=NULL))
                                        insert_point_into_cset (pt);
                        }
                }
}


calc_isovists ()
{
                /* this routine calculates the isovists of the current members of
                CSET, the candidate set of minimal points. */

                /* local data definitions */
                int occluded;  /* flag for occlusion test */

                /* start of code for calc_isovists */
#if TRACEMODE
                printf ("\ncalc_isovists");
#endif
                /* check out all possible candidate points */
                for (cc = 0; cc < c; cc++) {
                        /* consider line segment from cc to all other candidates */
                        /* must determine visibility i.e. member of isovist of cc */
                        for (cx = cc; cx < c; cx++) {
                                if (cx == cc) {
                                        /* obviously visible from self */
                                        insert_isovists ();
                                        continue;
                                        }
                        /* check to see first if the line segment from cc to cx is
                        exterior to the polygon.  If so, return the NULL+1 value which
                        indicates "occlusion", since we do not consider the points
                        visible to each other. */
                                if (is_exterior (CSET[cc],CSET[cx]) != NULL) continue;
                                /* must see if line seg cc-cx intersects an edge
                                  other than edges attached to cc and cx (if any) */
                                occluded = is_occluded (CSET[cc],CSET[cx]);
```

```
                        if (occluded == NULL) insert_isovists ();
                        }
                }
}


calc_lattice ()
{
        /* this routine determines which isovists are subsumed by
        other isovists, and declares them to be "unnecessary" as far
        as the minimal isovist set goes */

        /* local data definitions */
        int i, j;

        /* start of code for calc_lattice */
#if TRACEMODE
        printf ("\ncalc_lattice");
#endif
        for (i = 0; i < c; i++) {
                for (j = i+1; j < c; j++) {
                        if (is_subset (i,j) != NULL) {
                                /* i is a subset of j */
                                insert_unnecessary (i);
                                break;
                        }
                        if (is_subset (j,i) != NULL) insert_unnecessary(j);
                }
        }
        /* now construct the lattice L by copying only rows of V
        which are considered necessary at this point */
        for (i = 0; i < c; i++) {
                if (is_unnecessary (i) == NULL)
                        /* it IS necessary - copy this row */
                        for (j = 0; j < c; j++) L[i] [j] = V[i] [j];
        }
        /* save lattice for results printout in spare buffer */
        for (i = 0; i < c; i++) {
                for (j = 0; j < c; j++) {
                        LSAVE [i] [j] = L [i] [j];
                }
        }
}

find_minimals ()
{
        /* this routine determines which candidate points are absolutely
        necessary members of the Minimal Isovist set. It also does
        "garbage collection" by determining what remaining candidate points
```

```
                exist following this iteration of MIS calculation, and putting only
                those points in the (new) CSET. */

                /* local data definitions */
                int col,row, sum;

                /* start of code for find_minimals */
#if TRACEMODE
                printf ("\nfind_min;c=%d",c);
#endif

                for (col = 0; col < c; col++) {
                        /* sum this column */
                        sum = 0;
                        for (row = 0; row < c; row++) sum += L[row] [col];
                        if (sum == 1) {
                                /* find guilty row */
                                row = 0;
                                while (L [row] [col] == 0) ++row;
                                insert_MIS (row);
                        }
                }
                /* are we all done? If the matrix sums to zero, wow! */
                sum = 0;
                for (row = 0; row < c; row++) {
                        for (col = 0; col < c; col++) {
                                sum += L [row] [col];
                        }
                }
                if (sum == 0)
                        strcpy (complexity, "polynomial ");
                else {
                        strcpy (complexity, "combinatorial ");
                        combinatorics ();
                }
}



                /* subroutines */

is_occluded (p1, p2)
struct point p1, p2;        /* bounding points for a line segment */
{

                /* this routine determines if a particular diagonal or segment of P
                is occluded (i.e. cut by some edge of P). */
```

```
                /* local data definitions */
                int edge;

                /* start of code */
#if TRACEMODE
                printf ("\nis_occluded");
#endif
                for (edge = 1; edge <= n; edge++) {
                        if (line_intersect (P[edge%n], P[edge-1], p1, p2) == NULL)
                                continue;       /* not occluded by this edge */
                        /* since the lines formed by edge and p1p2 intersect,
                        is it a bifurcation of the diag by the edge??? */
                        if ((is_interior (p1, p2, pt) != NULL)
                        &&  (is_interior (P[edge%n], P[edge-1], pt) != NULL))
                                /* interior to both implies bifurcation */
                                return (NULL+1);
                }
                /* since we've gotten to this point, no occlusion has taken place */
                return (NULL);
}

line_intersect (e1, e2, p1, p2)
struct point e1, e2; /* first segment - generally an edge of P */
struct point p1, p2;        /* second segment - prob. a diagonal of P */
{
                /* this routine determines if the two line segments e1e2 and p1p2
                intersect. If they do not, NULL is returned. If they do intersect,
                NULL+1 is returned AND the intersection point is stored in the
                global point "pt". */

                /* local data definitions */
                float eslope, lslope;
                float eintercept, lintercept;

                /* start of code for line_intersect */
                /* first do special case processing for infinite sloping lines*/
#if TRACEMODE
                printf ("\nline_intersect");
#endif
                switch (check_verticals (e1, e2, p1, p2)) {
                        case NEITHER: break;        /*continue with regular calc. */
                        case FIRST:             /* edge is vertical */
                                lslope = getslope (p1,p2);
                                lintercept = getintercept (p1, lslope);
                                /* since e is vertical, z is fixed - so use it! */
                                pt.x = e1.x;
                                pt.y = (lslope*(pt.x))+lintercept;
                                return (NULL+1);
                        case SECOND:                /* second segment is vertical */
```

```
                        eslope = getslope (e1,e2);
                        eintercept = getintercept (e1, eslope);
                        pt.x = p1.x;
                        pt.y = (eslope*(pt.x))+eintercept;
                        return (NULL+1);
                case BOTH:              /* both lines vertical */
                        return (NULL);
        }
        eslope = getslope (e1, e2);
        lslope = getslope (p1, p2);
        /* are they parallel? */
        if (fabs(eslope-lslope) <= epsilon)
                return (NULL);
        /* now get intercepts and calculate point of intersection */
        eintercept = getintercept (e1, eslope);
        lintercept = getintercept (p1, lslope);
        pt.x = (eintercept - lintercept) / (lslope - eslope);
        pt.y = (eslope * (pt.x)) + eintercept;
        return (NULL+1);
}
check_verticals (d1, d2, pt1, pt2)
struct point d1, d2;        /* diagonal of the polygon P */
struct point pt1, pt2;      /* some other line segment */
{

        /* see if either line segment is vertical by checking
        to see if the x coordinates are identical */
#if TRACEMODE
        printf ("\ncheck_verticals");
#endif
        if (fabs(d1.x-d2.x)<=epsilon) {
                if (fabs(pt1.x-pt2.x)<=epsilon)
                        return (BOTH);
                else
                        return (FIRST);
        }
        if (fabs(pt1.x-pt2.x)<=epsilon) return (SECOND);
        return (NEITHER);
}


point_inclusion (pt)
struct point pt;
{
        /* local data definitions */
        float sum;
        struct point cur;
        struct point next;
        float theta;
```

```
                /* N.B.
                This routine uses a test for point inclusion in a polygon found in
W.K.Giloi's book "Interactive Computer Graphics" (Prentice-Hall,1978),
page 159, called "The Containment Test". I have not seen a proof for this
test; however, Giloi claims that it is a well-known procedure in elementary
geometry. It seems reasonable, and is similar to a method used by Freeman
and Loutrel in their solution to the Hidden-Line problem (1967). */
                /* Special NOTE: This test DOES NOT WORK if the test point
is on the boundary of P. This is because sin and cosine are in-
distinguishable at 0 and 180 degrees (ie pi and -pi); the test will
erroneously treat a negative "sweep" of 180 as a positive sweep. */
#if TRACEMODE
                printf ("\npoint_inclusion");
#endif
                sum = 0;
                /* check for special case - is point on perimeter of P? */
                if (is_on_boundary (pt) != NULL) return (NULL+1);
                /* see if test point is at vertex 0 */
                if ((fabs(P[0].x-pt.x)<=epsilon) && (fabs(P[0].y-pt.y)<=epsilon))
                        return (NULL);
                cur.x = P[0].x - pt.x;
                cur.y = P[0].y - pt.y; /* translate for test point at origin */

                for (temp = 0; temp < n; temp++) {
                        /* translate next point */
                        /* first check to see if test point is at vertex next */
                        if ((fabs(P[(temp+1)%n].x-pt.x)<=epsilon)
                        &&  (fabs(P[(temp+1)%n].y-pt.y)<=epsilon))
                                /* pretend no intersection - already in CSET */
                                return (NULL);
                        next.x = P[(temp+1)%n].x - pt.x;
                        next.y = P[(temp+1)%n].y - pt.y;
                        /* determine angle between these two normalized vectors */
                        theta = getangle (cur, next);
                        /* add the angles up */
                        sum += theta;
                        /* cycle to next vertex in polygon */
                        cur.x = next.x; cur.y = next.y;
                }
                /* now return NULL if sum is zero, ~ NULL if = 2pi */
                if (fabs (sum) > 3.0)
                        return (NULL+1);
                else
                        return (NULL);
}


is_extensible (v1,v2)
```

```
truct point v1, v2;

                /* this routine determines if a segment is extensible from point
                v1 through point v2 to a boundary of P. */

                /* local */
                int edge;

                /* start of code */
#if TRACEMODE
                printf ("\nis_extensible");
#endif
                for (edge = 1; edge <= n; edge++) {
                        if (line_intersect (P[edge%n], P[edge-1], v1, v2) == NULL)
                                continue;
                        /* do they intersect along the edge selected? */
                        if (is_interior (P[edge%n],P[edge-1],pt) == NULL) continue;
                        /* it does intersect - but in the correct direction? */
                        if (fabs (v1.x-v2.x) <= epsilon) {
                                /* vertical line - direction depends upon x coor */
                                if ((v1.y>v2.y) && ((pt.y-v2.y)>=(-epsilon))) continue;
                                if ((v1.y<=v2.y) && ((pt.y-v2.y)<=epsilon)) continue;
                                /* possible point - is it inside P? */
                                if (is_exterior(v2,pt) == NULL)
                                        return (NULL+1);
                                else
                                        continue;
                        }
                        /* not a vertical line - direction depends upon x coor */
                        if ((v1.x>v2.x) && ((pt.x-v2.x)>=(-epsilon))) continue;
                        if ((v1.x<v2.x) && ((pt.x-v2.x)<=epsilon)) continue;
                        /* possible */
                        if (is_exterior (v2,pt) == NULL) return (NULL+1);
                }
                return (NULL);          /* no extensible point found in this direction */




s_on_boundary (pt)
truct point pt;

                /*This routine determines if the test point is on the
erimeter of the polygon P. It does so by checking the distance
f the point from the edge segment. The code for this routine was
rovided by Gyorge Fekete. The routine then determines if
he point is interior to the (collinear) edge. */
```

```
                /* local data definitions */
                int edge;

                /* check each edge */
#if TRACEMODE
                printf ("\nis_on_boundary");
#endif
                for (edge = 1; edge <= n; edge++) {
                        if (fabs (distance (pt.x, pt.y, P[edge%n].x, P[edge%n].y,
                           P[edge-1].x, P[edge-1].y)) > epsilon)
                                continue;       /* not collinear */
                        /* pt is collinear - is it inside edge segment? */
                        if (is_interior (P[edge%n], P[edge-1], pt) != NULL)
                                return (NULL+1);
                }
                /* not on any edge in this polygon - therefore not on boundary*/
                return (NULL);
}



float
distance(px, py,  gx1,gy1, gx2,gy2)
float px, py;           /* the point */
float gx1,gy1;          /* base line */
float gx2,gy2;          /* end line */
{  /* func begin */
                float r;
                float gamma1, gamma2, gamma3;
#if TRACEMODE
                printf ("\ndistance");
#endif
                gamma1 = gy1 - gy2;
                gamma2 = gx2 - gx1;
                gamma3 = (gx1 * gy2) - (gx2 * gy1);
                r = gamma1 * px + gamma2 * py + gamma3;
                r /= sqrt(gamma1 * gamma1 + gamma2 * gamma2);
                return(r);
}


insert_point_into_cset (pt)
struct point pt;
{
                /* N.B.
                This routine must check to make sure that the intersection point
of the two extended edges is not already in the candidate set.  This routine
is thus responsible for keeping track of the cardinality of CSET. */
```

```
        /* local data definitions */
        int already_in;

        /* start of code for insert_point_into_cset */
'RACEMODE
        printf ("\ninsert_point_into_cset");
if
        already_in = NULL;
        for (temp = 0; temp < c; temp++) {
                /* compare test point pt against current cset entry */
                if ((fabs(CSET[temp].x-pt.x)<=epsilon)
                &&  (fabs(CSET[temp].y-pt.y)<=epsilon))
                        /* ahah! it's already inside CSET! */
                        already_in = NULL+1;
                }
        if (already_in == NULL) {
                /* it isn't in CSET - so put it in */
                if (c > MAXV2) printf ("\nFATAL ERROR: no room in CSET");
                else {
                CSET[c].x = pt.x;
                CSET[c].y = pt.y;
                ++c;
                }
        }



)pe (d1,d2)
t point d1, d2;         /* bounding vertices of diagonals */

        /* this routine calculates the slope of an edge of P */
    /* WARNING: THIS ROUTINE ASSUMES A NON-VERTICAL LINE IS INPUT */

        /* local data definitions */
        float slope;

        /* start of code for getslope */
'RACEMODE
        printf ("\ngetslope");
lif
        ftemp = d1.x - d2.x;
        slope = (d1.y - d2.y) / ftemp;
        return (slope);



tercept (e, slope)
t point e;
slope;
```

```
it, level;

        /* local */
        int last_entry;

        /* note: this is a recursive routine which determines the
        next combination of non-zero-rows taken "level" at a time. */

        last_entry = digit + nzr - level;
        if (COMBO [digit] == last_entry) {
                next_combo (digit-1, level);
                COMBO [digit] = COMBO [digit-1] + 1;
        }
        else {
                ++COMBO [digit];
        }


_covers (level)
el;

        /* this routine determines if a particular combination of
        non-zero rows in the lattice L covers the remaining set of
        non-zero columns. */

        /* local data definitions */
        int sum, col;

        /* this combo covers the remainder of the polygon iff for every
        remaining element, one of the combo's isovists sees the element. */
        sum = 0;
        for (col = 0; col < nzc; col++) {
                sum += coversum (0, col, level);
        }
EBUG
        if (level == 3) {
                printf ("\ncombo= (%f,%f),(%f,%f),(%f,%f),",
                CSET[LINDEX[COMBO[0]]].x,CSET[LINDEX[COMBO[0]]].y,
                CSET[LINDEX[COMBO[1]]].x,CSET[LINDEX[COMBO[1]]].y,
                CSET[LINDEX[COMBO[2]]].x,CSET[LINDEX[COMBO[2]]].y);
                printf (" and sum = %d", sum);
        }
f
        if (sum == nzc)
                return (NULL+1);
        else
                return (NULL);
```

```
                nzc = Ccol;
}



double
calc_possible_combos (n, r)
int n,r;
{
                /* local definitions */
                double rfac, result;
                int rx;

                /* calculate number of possible combinations of n things taken
                r at a time, which equals n! / [(r!)((n-r)!)]. */

                result = 1;
                for (rx = n; rx > r; --rx) result *= rx;/*n!/r!*/
                rfac = factorial (n-r);
                result /= rfac;
                return (result);
}

double
factorial (n)
int n;
{
                double answer;

                answer = n;
                for (temp = 2; temp < n; temp++) {
                        answer *= temp;
                }
                return (answer);
}


init_combos (level)
int level;
{
                for (temp = 0; temp < level-1; temp++) {
                        COMBO [temp] = temp;
                }
                COMBO [level-1] = level-2;
}

next_combo (digit, level)
```

```
                }
            }
            if (found_combo != NULL) break;
        }
        if (found_combo == NULL) {
            /* uh oh - must use entire lattice!!!!! */
            init_combos (nzr);
            next_combo (nzr, nzr);
            insert_combo (nzr);
        }
```

_C_matrix ()

```
        /* this routine compresses the L lattice matrix by eliminating
        rows and columns which are composed entirely of zeroes. The
        array LINDEX remembers to which row in L a given row in C corresponds.
        By doing this once, we do not have to scan for the next significant
        row (or column) every time we try a new combination. */

        /* local data definitions */
        int Crow, Ccol, row, col;

        /* start by eliminating rows full of zeroes */
        Crow = 0;
        for (row = 0; row < c; row++) {
            if (rowsum (row) == 0) continue;/* ignore zeroed row */
            /* if non-zero, compress by moving into C */
            LINDEX [Crow] = row;
            for (col = 0; col < c; col++) {
                C [Crow] [col] = L [row] [col];
            }
            ++Crow;
        }
        nzr = Crow;  /* number of non-zero rows */

        /* now compress zeroed columns */
        Ccol = 0;
        for (col = 0; col < c; col++) {
            if (colsum (col) == 0) continue;
            /* compress this non-zero column */
            for (row = 0; row < nzr; row++) {
                C [row] [Ccol] = C [row] [col];
            }
            ++Ccol;
        }
```

```
of isovists in order to determine a truly minimal isovist set.
This routine is called only when the algorithm finds no obvious
solution in polynomial time. It only searches the reduced lattice
(i.e. only those members of the lattice which do not see a
point uniquely - some other member of the lattice always sees
a point that it sees). It is possible (Version 2.0) to introduce
a heuristic approach in determining which combination of lattice
elements should be selected for testing (i.e. the order of priority
for testing combinations at some level). The current version of
this algorithm simply tries all combinations in order of appearance
within the lattice L. */

/* local data definitions */
int level, curcombo, found_combo;
double nr_combos;

/* start of code for combinatorics */
nc = 0;                          /* initialize number of combinations tried */
found_combo = NULL;
init_C_matrix ();
#if DEBUG
printf ("\nhere at combinatorics");
printf ("\n nzr = %d, nzc = %d", nzr, nzc);
#endif


for (level = 2; level < nzr; level++) {

        /* consider all combinations of non_zero_row elements
        taken "level" at a time. Since we start with level == 1
        and work upwards, we are guaranteed in this fashion
        to find the minimal combination first. */

        nr_combos = calc_possible_combos (nzr, level);
        init_combos (level);
#if DEBUG
printf ("\nnr_combos = either %d or %ld",nr_combos, nr_combos);
printf (" and found-combo is %d", found_combo);
#endif

        for (curcombo = 1; curcombo <= nr_combos; curcombo++) {
                ++nc;
                next_combo (level - 1, level);/* recursive */
                if ((combo_covers (level) != NULL)
                &&  (check_covering (level) != NULL)) {
                        /* we've found it! */
                        found_combo = NULL+1;
                        insert_combo (level);
                        break;
```

```
        int closest;
        static struct orders {
                char name [30];
                double value;
                } OMTAB [] = {
"n ",  0,
"n log n ",     0,
"n log (n squared) ",  0,
"n squared ",  0,
"n squared log n ",    0,
"n squared log (n squared) ", 0,
"n cubed ",    0,
"n cubed log n ",      0,
"n cubed log (n squared) ",   0,
"n to the fourth ",    0
                        };


        /* start of code */
        /* first initialize all values properly based on n */
        fc = c;
        closest = 0;
        OMTAB[0].value = n;
        logn = log (OMTAB[0].value);
        OMTAB[3].value = n * n;
        logn2 = log (OMTAB[3].value);
        OMTAB[1].value = n * logn;
        OMTAB[2].value = n * logn2;
        OMTAB[4].value = OMTAB[3].value * logn;
        OMTAB[5].value = OMTAB[3].value * logn2;
        OMTAB[6].value = OMTAB[3].value * n;
        OMTAB[7].value = OMTAB[6].value * logn;
        OMTAB[8].value = OMTAB[6].value * logn2;
        OMTAB[9].value = OMTAB[6].value * n;

        /* now calculate closest value to actual size of set S */
        for (temp = 1; temp < 10; temp++) {
                if (fabs (fc - OMTAB[temp].value) <
                   fabs (fc - OMTAB[closest].value))
                        closest = temp;
        }
        strcpy (order, OMTAB[closest].name);
        magnitude = OMTAB[closest].value;
}


combinatorics ()
{
        /* This is the algorithm which exhaustively searches the lattice
```

```
        printf (".");
    printf ("\nNON-VERTEX POINTS:");
    if (n == c) printf (" (none).");
    else {
        pointcount = 0;
        for (temp = n; temp < c-1; temp++) {
            printf (" (%3.2f,%3.2f),", CSET[temp].x,CSET[temp].y);
            if (++pointcount == MAX_ON_1_LINE) {
                pointcount = 0;
                printf ("\n");
            }
        }
        printf (" (%3.2f,%3.2f).", CSET[c-1].x,CSET[c-1].y);
    }
    printf ("\nNUMBER OF VERTICES (N):  %d", n);
    printf ("\nNUMBER OF POINTS IN CANDIDATE SET S:  %d", c);
    calc_order ();
    printf ("\nMAGNITUDE OF S COMPARED WITH N: CLOSEST TO O(%
        order, magnitude);
    print_visibility ();
    print_lattice ();
    printf ("\nSOLUTION IS %s.", complexity);
    if (complexity [0] == 'c')
        printf ("  %ld combinations were evaluated to find a MIS.",
                nc);
    printf ("\nNUMBER OF POINTS IN M.I.S.:  %d", m);
    printf ("\nMINIMAL ISOVIST SET:");
    pointcount = 0;
    for (temp = 0; temp < m-1; temp++) {
        printf (" (%3.2f,%3.2f),", MIS[temp].x,MIS[temp].y);
        if (++pointcount == MAX_ON_1_LINE) {
            pointcount = 0;
            printf ("\n");
        }
    }
    printf (" (%3.2f,%3.2f).", MIS[m-1].x,MIS[m-1].y);
    printf ("\n ");
    for (temp = 0; temp < 72; temp++) printf ("-");
    print_polygon ();
}


calc_order ()
{
    /* this routine calculates the order of magnitude of the
    size of the search space used in finding the MIS */

    /* local definitions */
    double logn, logn2, fc;
```

```
}

print_visibility ()
{
                int e;
                /* print the isovist visibility matrix V */
                printf ("\nVisibility Matrix V = ");
                for (temp = 0; temp < c; temp++) {
                        printf ("\n");
                        for (e = 0; e < c; e++) {
                                printf (" %d ", V [temp] [e]);
                        }
                }
}



print_results ()
{
                /* this routine prints out a table of results */
                /* local data definitions */
                int pointcount;

#define         MAX_ON_1_LINE   6       /* number of cartesian points per printline */

                /* start of printout */
                printf ("\n0EST NUMBER: %s", test_id);
                printf ("\nPOLYGON:");
                pointcount = 0;
                for (temp = 0; temp < n-1; temp++) {
                        printf (" (%3.2f,%3.2f),", P[temp].x,P[temp].y);
                        if (++pointcount == MAX_ON_1_LINE) {
                                pointcount = 0;
                                printf ("\n");
                        }
                }
                printf (" (%3.2f,%3.2f).",P[n-1].x,P[n-1].y);/* last point */
                printf ("\nREFLEX VERTICES:");
                for (temp = 0; temp < n-1; temp++) {
                        if (P[temp].is_reflex == REFLEX)
                                printf (" (%3.2f,%3.2f),", P[temp].x,P[temp].y);
                        if (++pointcount == MAX_ON_1_LINE) {
                                pointcount = 0;
                                printf ("\n");
                        }
                }
                if (P[n-1].is_reflex == REFLEX)
                        printf (" (%3.2f,%3.2f).", P[n-1].x, P[n-1].y);
                else
```

```
#endif
                for (temp = 0; temp < m; temp++) {
                        if ((MIS[temp].x === CSET[cindex].x)
                        &&  (MIS[temp].y === CSET[cindex].y))
                                return (NULL+1);
                }
                return (NULL);
}


set_col_to_zeroes (col)
int col;
{
                /* this routine sets a particular column of the lattice matrix
                        to zeroes, indicating that this point is now visible
                        (i.e. covered) from some point in the MIS */
#if TRACEMODE
                printf ("\nset_col_to_zeroes");
#endif
                for (temp = 0; temp < c; temp++) L [temp] [col] = 0;
}


set_row_to_zeroes (row)
int row;
{
                /* this routine sets a particular row of the lattice matrix
                        to zeroes, indicating that this point is part of the
                        MIS */
#if TRACEMODE
                printf ("\nset_row_to_zeroes");
#endif
                for (temp = 0; temp < c; temp++) L [row] [temp] = 0;
}

print_lattice ()
{
                /* local declarations */
                int row, col;

                /* print row by row */
                printf ("\nLattice Matrix L of V:");
                for (row = 0; row < c; row++) {
                        printf ("\n");
                        for (col = 0; col < c; col++) {
                                printf (" %d ", LSAVE [row] [col]);
                        }
                }
```

```
is_unnecessary (cindex)
int cindex;
{
                /* this routine determines if a candidate point is a member
                of the set of unnecessary (ie subsumed) candidate points */
#if TRACEMODE
                printf ("\nis_unnecessary");
#endif
                for (temp = 0; temp < un; temp++) {
                        if (cindex == UNNECESSARY[temp]) return (NULL+1);
                }
                return (NULL);
}



insert_MIS (cindex)
int cindex;
{
                /* local data declarations */
                int col;

                /* this routine inserts a candidate point into the minimal
                isovist set. */
                /* first make sure it's not already inside MIS, okay? */
#if TRACEMODE
                printf ("\ninsert_MIS");
#endif
                if (is_minimal (cindex) != NULL) return;
                /* okay - insert it */
                MIS[m].x = CSET[cindex].x;
                MIS[m].y = CSET[cindex].y;
                /* now clear out columns seen by this point */
                for (col = 0; col < c; col++) {
                        if (L [cindex] [col] == 1) set_col_to_zeroes (col);
                }
                set_row_to_zeroes (cindex);
                insert_unnecessary (cindex);
                ++m;
}


is_minimal (cindex)
int cindex;
{
                /* this routine checks to see if a candidate point is a member
                of the (current) MIS */
#if TRACEMODE
                printf ("\nis_minimal");
```

```
#endif
                V [key] [iso] = 1;
}


is_subset (i,j)
int i;
int j;
{
                /* Burning Question:
                  Is the isovist of candidate point i a subset of the isovist of
                candidate point j??? This routine answers that question. */

                /* local data definitions */
                int ii;         /* current point in i's isovist being looked for */

                /* start of code for is_subset */
                /* i is a subset of j iff for all ii from 0 to c, V[i][ii]=1 implies
                that V[j][ii]=1 also. That is, if V[i][ii]=1 and V[j][ii]=0, then
                i is NOT a subset of j. */
#if TRACEMODE
                printf ("\nis_subset");
#endif
                for (ii = 0; ii < c; ii++) {
                        if ((V [i] [ii] == 1) && (V [j] [ii] == 0))
                                /* i is NOT a subset of j */
                                return (NULL);
                }
                return (NULL+1);    /* fell through means i is subset */
}



insert_unnecessary (index)
int index;
{
                /* make sure that it isn't already subsumed */
#if TRACEMODE
                printf ("\ninsert_unnecessary");
#endif
                for (temp = 0; temp < un; temp++)
                        if (UNNECESSARY[temp] == index) return;
                UNNECESSARY[un] = index;
                ++un;
}
```

```
                /* select a midway point which is not a vertex */
                div *= 2;       /* try an inbetween point */
        ptemp.x =
                (fabs (p1.x) > fabs (p2.x)) ?
                        ((p1.x - p2.x) / div) + p2.x :
                        ((p2.x - p1.x) / div) + p1.x ;
        ptemp.y =
                (fabs (p1.y) > fabs (p2.y)) ?
                        ((p1.y - p2.y) / div) + p2.y :
                        ((p2.y - p1.y) / div) + p1.y ;
        }
        while (is_vertex (ptemp) != NULL);
        /* okay - our selected mid-way point is not a vertex */
        if (point_inclusion (ptemp) != NULL)
                return (NULL);
        else
                return (NULL+1);
}


is_vertex (pt)
struct point pt;
{
        /* this routine determines if the test point is a vertex of
        the given polygon P */
#if TRACEMODE
        printf ("\nis_vertex");
#endif
        for (temp = 0; temp < n; temp++) {
                if ((fabs(pt.x-P[temp].x)<=epsilon)
                &&  (fabs(pt.y-P[temp].y)<=epsilon))
                        return (NULL+1);
        }
        return (NULL);          /* not a vertex */
}




insert_one_isovist (key, iso)
int key;
int iso;
{
#if TRACEMODE
        printf ("\ninsert_one_isovist");
```

```
                /* vertical - use z axis */
                x1 = p1.y;
                x2 = p2.y;
                coor = pt.y;
        }
        else {
                /* not vertical - use the z axis */
                x1 = p1.x;
                x2 = p2.x;
                coor = pt.x;
        }

        /* now determine if "coor" lies between x1 and x2 */
        if ((x1-x2) < epsilon) {
                if (((coor-x1)>epsilon)&&((coor-x2)<-epsilon)) return (NULL+1);
        }
        else    {
                if (((coor-x1)<-epsilon)&&((coor-x2)>epsilon)) return (NULL+1);
        }
        /* no go */
        return (NULL);
}


is_exterior (p1,p2)
struct point p1, p2;
{

        /* local data definitions */
        int div;
        struct point ptemp;

        /* this routine determines if the line segment p1p2 lies
        exterior to the polygon. It does this by considering some
        point, NOT A VERTEX OF P, which lies between the points cc
        and cx along the line segment cc-cx. If this point is
        exterior to the polygon, the entire line segment is, for
        our purposes, considered "exterior" (i.e. not visible). */

        /* first check to see if point cc = point cx */
#if TRACEMODE
        printf ("\nis_exterior");
#endif
        if ((fabs(p1.x-p2.x)<=epsilon)
        &&  (fabs(p1.y-p2.y)<=epsilon)) return (NULL);

        div = 1;
        do {
```

```
                dirsin2 = p2.y / len2;
        }
        /* now calculate sine and cosine of angle between vectors */
        sin_theta = dirsin1 * dircos2 - dircos1 * dirsin2;
        cos_theta = dircos1 * dircos2 + dirsin1 * dirsin2;
        /* now compute the actual angle using sin and cosine */
        theta = (asin (sin_theta) < 0.0) ? acos(cos_theta) : -acos(cos_theta);
        return (theta);
}

float
unitlength (p1)
struct point p1;
{
        /* calculate the length from origin to the point */
#if TRACEMODE
        printf ("\nunitlength");
#endif
        return (sqrt ((p1.x * p1.x) + (p1.y * p1.y)));
}


insert_isovists ()
{
        /* since cx is visible from cc, cc is visible from cx. Therefore,
        put cc into cx's isovist and cx into cc's isovist. */
#if TRACEMODE
        printf ("\ninsert_isovists");
#endif
        insert_one_isovist (cc,cx);
        insert_one_isovist (cx,cc);
}


is_interior (p1,p2,pt)
struct point p1, p2, pt;
{
        /* this routine determines if point pt is interior to the
        line segment bounded on the x axis by the two parameters */

        /* local data */
        float x1, x2, coor;

        /* check to see if line segment bounded by p1 and p2 is vertical */
        /* if it is, we must use the z axis in lieu of x axis */
#if TRACEMODE
        printf ("\nis_interior");
#endif
        if (fabs (p1.x-p2.x) <= epsilon) {
```

```
{
                /* local data definitions */
                float intercept;

                /* start of code - see "getslope" for comments */
#if TRACEMODE
                printf ("\ngetintercept");
#endif
                intercept = e.y - (slope * e.x);
                return (intercept);
}

float
getangle (p1, p2)
struct point p1;
struct point p2;
{
                /* calculate the angle between two vectors (from origin) */
                /* local data definitions */
                float len1, len2;
                float dircos1, dircos2, dirsin1, dirsin2;
                float sin_theta, cos_theta, theta;

                /* get length of vectors */
#if TRACEMODE
                printf ("\ngetangle");
#endif
                len1 = unitlength (p1);
                len2 = unitlength (p2);

                /* calculate direction angle sines and cosines */
                if (len1 <= epsilon) {
                        /* zero-valued length -problem! */
                        /* pick one coordinate as more important (non-zero)*/
                        dircos1 = (p1.x<=epsilon) ? 0 : 1;
                        dirsin1 = (p1.x<=epsilon) ? 1 : 0;
                }
                else {  /* okay to divide by len1 */
                        dircos1 = p1.x / len1;
                        dirsin1 = p1.y / len1;
                }
                /* now calculate second vector's directional angles */
                if (len2 <= epsilon) {
                        /* don't divide by zero */
                        dircos2 = (p2.x<=epsilon) ? 0 : 1;
                        dirsin2 = (p2.x<=epsilon) ? 1 : 0;
                }
                else {  /* okay to divide by len2 */
                        dircos2 = p2.x / len2;
```

```
coversum (digit, col, level)
int digit, col, level;
{
                if (digit == level-1)
                        return (C [COMBO[digit]] [col]);
                else
                        return (C [COMBO[digit]] [col] | coversum (digit+1,col,level));
}


insert_combo (level)
int level;
{
                int mtemp;

                for (mtemp = 0; mtemp < level; mtemp++) {
                        insert_MIS (LINDEX [COMBO[mtemp]]);
                }
}


rowsum (row)
int row;
{
                /* this routine calculates the sum of the entries in the Lattice
                matrix for a particular row. */

                /* local data definitions */
                int col, sum;

                sum = 0;
                for (col = 0; col < c; col++) {
                        sum += L [row] [col];
                }
                return (sum);
}


colsum (col)
int col;
{
                /* this routine calculates the sum of the entries in the
                Lattice matrix for a particular column. */

                /* local data definitions */
                int row, sum;

                sum = 0;
                for (row = 0; row < nzr; row++) {
```

```
                    sum += C [row] [col];
                }
                return (sum);
}

print_polygon ()
{
                /* dummy routine for now */
}
getline(s, lim)     /* get line into s, return length */
char s[];
int lim;
{
                int c, i;

                i = 0;
                while (--lim > 0 && (c=getchar()) != EOF && c != '0')
                        s[i++] = c;
                s[i] = ' ';
                return(i);
}



match_MIS (index)
int index;
{
                /* this routine takes an MIS array index and returns the
                corresponding visibility matrix index. */

                /* local definitions */
                int vindex;

                /* start of routine */
                for (vindex = 0; vindex < c; vindex++) {
                        if ((fabs (CSET[vindex].x-MIS[index].x) < epsilon)
                        &&  (fabs (CSET[vindex].y-MIS[index].y) < epsilon))
                                return (vindex);
                }
                return (index);
}



check_covering (level)
int level;
{
```

```
/* local declarations */
int t1,t2,t3,cover,covered;

/* use Visibility matrix V to determine if all triads are
covered by the chosen "minimal" set */
for (t1=0; t1<n-2; t1++) {
        for (t2 = t1+1; t2<n; t2++) {
                if (V[t1][t2] != 1) continue;/* not a triad*/
                /* t1 sees t2- now find all t3's */
                for (t3=t2+1; t3<n; t3++) {
                        if ((V[t1][t3] != 1)
                          || (V[t2][t3] != 1)) continue;
                        /* legal triad found - is it covered? */
                        covered = NULL;
                        for (cover=0; cover<m; cover++) {
                                temp = match_MIS (cover);
                                if ((V[temp][t1] == 1)
                                && (V[temp][t2] == 1)
                                && (V[temp][t3] == 1)) {
                                        covered = NULL+1;
                                        break;
                                }
                        }
                        if (covered == NULL) {
                                /* see if covered by new combo */
                                for (cover=0; cover<level; cover++) {
                                        temp=LINDEX[COMBO[cover]];
                                        if ((V[temp][t1]==1)
                                        && (V[temp][t2]==1)
                                        && (V[temp][t3]==1)) {
                                        covered=NULL+1;
                                        break;
                                        }
                                }
                                if (covered==NULL) return (NULL);
                        }
                }
        }
}
return (NULL+1);
}
```

Appendix C. Glossary of Important Terms.

Degree of complexity: A measurement which indicates the amount of processing needed for the measured object. In this paper, a polygon's degree of complexity is the cardinality of its MIS.

Diagonal: A line segment connecting two vertices of a polygon.

Isovist (of $x \epsilon P$): The set of all points visible from x in P.

Kernel: The locus of points x such that $\overline{xp} \subset P$ for all points p in P.

Minimal cover: A set of regions each a subset of P, such that the union of the regions equals P, and there is no smaller set of regions whose union equals P.

Minimal isovist set: The smallest set of points in P such that the union of their isovists equals P.

Minimal star-shaped cover: A minimal cover using star-shaped polygonal regions.

MIS: Short for "minimal isovist set".

MSC: Short for "minimal star-shaped cover".

N-gon: A planar figure composed of n straight line segments intersecting at n vertices.

Polygon: A planar region bounded by an n-gon.

Rectilinear polygon: A polygon whose intersecting edges form right angles.

Star-shaped polygon: A polygon whose kernel is non-empty.

Sufficient isovist set: A set of points in P such that the union of the isovists of the points equals P.

Visible, visibility: Two points x, x in P are visible if the line segment $\overline{xy} \cap P = \overline{xy}$.

References.

[APOS 57] Apostol, T.M., Mathematical Analysis, Addison-Wesley Publishing Co., 1957.

[AVIS 80] Avis, D. and Toussaint, G.T., "An efficient algorithm for decomposing a polygon into star-shaped polygons", Tech. Rept. No. SOCS 80.8, School of Computer Science, McGill University, May 1980.

[CHVA 75] Chvatal, V., "A Combinatorial Theorem in Plane Geometry", Journal of Combinatorial Theory (B) 18 (1975), 39-41.

[DAVI 79] Davis, L.S. and Benedikt, M.L., "Computational Models of Space: Isovists and Isovist Fields", Computer Graphics and Image Processing 11 (1979), 49-72.

[FISK 78] Fisk, S., "A Short Proof of Chvatal's Watchman Theorem", Journal of Combinatorial Theory (B) 24 (1978), 374.

[FREE 67] Freeman, H. and Loutrel, P.P., "An Algorithm for the Solution of the Two-Dimensional 'Hidden-Line' Problem", IEEE Transactions on Electronic Computers EC-16 (1967), 784-790.

[GARE 78] Garey, M.R., Johnson, D.S., Preparata, F.P., and Tarjan, R.E., "Triangulating a Simple Polygon", Information Processing Letters 7 (1978), 175-179.

[LEE 76] Lee, D.T. and Preparata, F.P., "Location of a Point in a Planar Subdivision and its Applications", Proc. of the 8th Annual ACM Symposium on Theory of Computing (May 1976), pp. 231-235.

[LEE 79] Lee, D.T. and Preparata, F.P., "An Optimal Algorithm for Finding the Kernal of a Polygon", Journal of the ACM 26 (1979), 415-421.

[MARU 72] Maruyama, K., "A Study of Visual Shape Perception", Tech. Rept. No. UIUCDCS-R-72-533, Department of Computer Science, U. of Illinois, Urbana, Oct. 1972.

[PLAN 72] Plane, D.R. and Kochenberger, G.A., Operations Research for Managerial Decisions, Richard D. Irwin, Inc. (1972), p. 127.

[OROU 82a] O'Rourke, J., "A Note on Minimum Convex Covers for Polygons", Tech. Rept. No. JHU-EECS 82-1, Department of Electrical Engineering and Computer Science, The Johns Hopkins University, Baltimore, Maryland, Jan. 1982.

[OROU 82b] O'Rourke, J., "The Complexity of Computing Minimum Convex Covers For Polygons", Proceedings of the Twentieth Allerton Conference, Oct. 1982, 75-84.

[OROU 82c] O'Rourke, J., "An Alternate Proof of the Rectilinear Art Gallery Theorem", Tech. Rept. No. JHU/EECS-82/15, Department of Electrical Engineering and Computer Science, The Johns Hopkins University, Baltimore, Maryland, Dec. 1982.

[SHAM 75] Shamos, M.I., "Geometric Complexity", Proc. 7th ACM Symposium on the Theory of Computing, May 1975, 224-233.

[SHAM 77] Shamos, M.I., "Computational Geometry", Ph.D. thesis, Yale University, May 1978.

[TOUS 80] Toussaint, G.T., "Pattern Recognition and Geometrical Complexity", Proceedings of the 5th International Conference on Pattern Recognition (1980), pp. 1324-1347.

[YAGL 61] Yaglom, I.M. and Boltyanskii, V.G., Convex Figures, Holt, Rinehart and Winston (1961), p. 103.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO. AD-A157 624	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) COMPUTATION OF MINIMAL ISOVIST SETS		5. TYPE OF REPORT & PERIOD COVERED Technical
		6. PERFORMING ORG. REPORT NUMBER CAR-TR-87; CSC-TR-1430
7. AUTHOR(s) Mark F. Doherty		8. CONTRACT OR GRANT NUMBER(s) DAAK-70-83-K-0018
9. PERFORMING ORGANIZATION NAME AND ADDRESS Center for Automation Research University of Maryland College Park, MD 20742		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS U.S. Army Engineer Topographic Labs Ft. Belvoir, VA 22060		12. REPORT DATE September 1984
		13. NUMBER OF PAGES 87
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Shape analysis
Isovist sets
Visibility
Star-shaped regions

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

A minimal isovist set (MIS) of a simple polygonal region P is a smallest set of points in P whose union of isovists equals P (where the isovist of x is the set of all points visible from x). This thesis presents an algorithm to search for an MIS for an arbitrary P. An MIS is shown to be equivalent to a minimal covering of P with star-shaped polygons. A (non-complete) algorithm to find a minimal covering is proposed which uses the vertices of the kernels of the

DD FORM 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

star-shaped polygons. The complexity of finding an MIS is reduced to a worst-case consideration of no more than n^4 points in P. A comparison of the proposed algorithm with two previously published algorithms is made. Extension of this method to exterior views and interior holes is discussed, and areas for future research are mentioned.

END

FILMED

9-85

DTIC