Generation of Random Simple Polygons and its Visibility Graph

A dissertation submitted in partial fulfillment
of the requirements of M.Tech. (Computer Science)
degree of Indian Statistical Institute, Kolkata
by

Kumar Swamy H V

under the supervision of

Dr.S.C.Nandy

Indian Statistical Institute
203, Barrackpore Trunk Road
Kolkata-700 108.

22^{th} July 2002

Indian Statistical Institute

203, Barrackpore Trunk Road,

Kolkata-700 108.

Certificate of Approval

This is to certify that this thesis titled **"Generation Random Simple Polygons and its Visibility Graph"** submitted by **Kumar Swamy H V** towards partial fulfillment of requirements for the degree of M.Tech in Computer Science at Indian Statistical Institute, Kolkata embodies the work done under my supervision.

Nandy 23/7/02

Dr.S.C.Nandy
Advanced Computing and Microelectronic Unit,
Indian Statistical Institute,
Kolkata-700 108.

S. Chattopadhyay 23/7/02
Examiner

Acknowledgments

I take pleasure in thanking Dr.S.C.Nandy for his friendly guidance throughout the dissertation period. His pleasant and encouraging words have always kept my spirits up.

I would also like to express my sincere gratitude to Dr. Sandeep Das for agreeing to share a few words with me. A few days of conversation with him gave me a new confidence and enabled me to complete this work successfully.

I take the opportunity to thank my family, my friends and my classmates for their encouragement to finish this work.

Kumar Swamy H V

Abstract :*The problem of randomly generating simple polygons is of considerable importance in the practical evaluation of algorithms that operate on polygons by generating test instances, where it is necessary to check the correctness and to determine the CPU-consumption of an algorithm experimentally.*

Till now no polynomial solution for uniformly random generation of polygon is known. Auer and Held[2], considered a class of heuristics for this problem. In this thesis, we present an algorithm for the generation of simple random polygon in random manner after a preprocessing which takes $O(n^3 logn)$ time. Though the worst time required to generate a polygon may be $O(n^3)$, the actual CPU time needed for the generation is very less.

We also considered finding the visibility graph of those generated random simple polygons which will be helpful for many problems. We considered already existing algorithms for this purpose.

Contents

1 Introduction

Visibility graphs of a simple polygon plays important role in many geometric problems, for example, in robat path planing, morphing, communicating animation data, to name few. In this report we will study the problem of computing visibility graph of a simple polygon. In order to test the algorithm we need to generate simple random polygon of arbitrary shape. This problem is important in its own right and has many other applications apart from those indicated earlier. We first discuss the problem of generating random polygons, then we will discuss the problem of computing visibility graphs.

1.1 Generation of random polygons

Given a set $S = \{s_1, \ldots, s_n\}$ of n points, we would like to generate a simple polygon P, such that the points of S form the vertices of P. A polygon is said to be randomly generated on a point set S, if probability of getting that is $\frac{1}{k}$ if there exist k possible simple polygons with the point set S. Since no polynomial time solution is known for the generation of random simple polygons, we focus on a algorithm that offer a good time complexity and still generates a rich variety of different polygons.

In addition to the theoretical interest of its own, the generation of random simple polygons is an useful tool for testing and verification of time complexity and CPU-time consumption of algorithms that operate on polygons. For testing the correctness of algorithm, we need to generate diverse set of input data such that all branches of the algorithm will be executed with high probability. The same motivation applies to the practical testing of an algorithm's CPU-time consumption.

Recently, the generation of random polygons has received some attention by researchers working on the applied problems in Computational Geometry. A heuristic for generation of simple polygons has been investigated by J.O'Rourke and Viramani [10]. Their algorithm moves the vertices while creating a polygon. Thus it does not fall in to the class of algorithm presented in this report.

An $O(n^2)$ algorithm for the generation of x-monotone polygons was described by Zhu et al [14]. For arbitrary simple polygons, a class of heuristics are proposed by T. Auer and M. Held [2, 3]. These are

Steady growth algorithm: It is incremental algorithm adding points one after the other;

Space partitioning algorithm: It is a divide and conquer algorithm;

Permute & reject algorithm: It creates random permutations of (polygons) until a simple polygon is encountered;

2-opt moves algorithm: It generates a random (may be non-simple) polygon and then repairs it (if necessary) by swapping the end points of crossing edges so-called 2-opt moves; and

Incremental construction & backtracking algorithm: It goes on constructing the polygonal chain as long as it remains simple. Backtracking is applied when a non-simple chain is encountered.

Although the worst case complexity of first two algorithms is $O(n^2)$, they will not generate all possible simple polygons on set S. The third method (Permute & reject algorithm) generates a polygon in $O(n \log n)$ time, but there is no guarantee that it will be a simple polygon. Also, no bound on number of times the algorithm is to be run to get a simple polygon is given. The time complexity of the fourth method is $O(n^4)$. The efficiency of the fifth method depends on the amount of backtracking. Fourth and fifth methods are not suitable for practical problems. A simple algorithm for the generation of star shaped polygons is also discussed in that paper.

In this report we propose an algorithm for generating random simple polygons. Given the set of n points, we construct a data structure of size $O(n^3)$ in $O(n^3 \log n)$ time. Next generation of each polygon on the same point set needs $O(n^3)$ in worst case. We implemented our algorithm on different instances of randomly generated point sets. Though the worst case time complexity of our algorithm is $O(n^3)$, the actual CPU-time needed is observed to very less.

1.2 Generation of visibility graph

The visibility graph is a fundamental combinatorial structure in computational geometry; it is used, for example, in the application such as computing shortest paths amidst polygonal obstacles in the plane. The *visibility graph* of a set of non-intersecting polygonal obstacle in the plane is an undirected graph whose vertex set consists of the vertices of the obstacles and whose edges are the pairs of vertices (u, v) such that the open line segment between u and v doesn't intersect any of the obstacles. In case of polygons, the *visibility graph* consists of vertex set corresponding to the vertices of the polygon and the edges are pairs of vertices (u, v) such that the open line segment between u and v completely lies inside the polygon and does not intersect any of the edges of the polygon.

Till now lot of work is done on the computation of visibility graph of polygons. In the worst case the visibility graph of a set of obstacles with n vertices may contain $O(n^2)$ edges. An $O(n^2 \log n)$ algorithm for this problem was given by Sharir and Schorr [12]. Later, worst case optimal $O(n^2)$ algorithms were given by Asano et al. [1] and Welzl [13]. Hershberger [9] has described a output-sensitive algorithm for the case of computing the visibility graph of a triangulated simple polygon. Overmars and Welzl [11] have given an algorithm for computing the visibility graph for a set of disjoint polygonal obstacles whose running time is $O(E \log n)$ and space is $O(n)$, where E is the number of edges in the visibility graph. Later Ghosh and Mount [8] have given algorithm which computes the visibility graph of an arbitrary set of disjoint obstacles. The worst case time complexity of their algorithm is $O(E + n \log n)$, where $O(n \log n)$ time is required to compute the triangulation of obstacle free space, and $O(E)$ time is required to compute the visibility graph. The space complexity

of the algorithm is $O(E + n)$. All the above mentioned algorithms can be used to find the visibility of simple polygon.

We have implemented the algorithm given in the book by Overmars et.al.[4], which takes $O(n^2 \log n)$ time.

In sections 2 and 3 , we discuss the generation of random simple polygons, its algorithm, complexity issues and some experimental results. In section 4 we discuss visibility graphs, its algorithm and complexity issues.

2 Generation of random polygons

Let $S = \{s_1, ..., s_n\}$ be set of n points. Initially we will give you a brief outline of the algorithm. Later we will explain each step in more detail. The main idea of our algorithm for generating random simple polygons is that, starting with a random empty triangle, we grow it to a random simple polygon, by adding new empty triangle randomly, which is adjacent to the triangles we have already chosen. The outer boundary of union of these adjacent triangles forms the random simple polygon.

The algorithm works as follows:

- In Preprocessing step, to each pair of points u, $v \in S$, associate all possible empty triangles[1] on the plane whose one of the sides is the line segment $\overline{uv}$.
- During the generation of polygon,
  1. First we choose a pair (u, v) (i.e., the edge $\overline{uv}$) randomly from the set of $\frac{n(n-1)}{2}$ pairs.
  2. Next, randomly select a triangle $\triangle$ associated with $\overline{uv}$. We drop all triangles associated with three edges of the triangle $\triangle$, which are overlapping with triangle $\triangle$. Still if any triangle remains associated with any of these edges, then that edge will be added to *Future_Candidate_List*, which is empty initially.
  3. for ($|S| - 3$) times, repeat the following 3 steps:
     - I Choose an edge e randomly from *Future_Candidate_List*.
     - II Choose a triangle $\triangle$ still associated with e randomly, so that it does not overlap with already chosen triangles and also the third vertex of the triangle $\triangle$ is not an end vertex of any edge we have already chosen.Delete e from *Future_Candidate_List*.
     - III Delete all triangles associated with these three edges of triangle $\triangle$ which are overlapping with $\triangle$. Still if any triangle remains associated with any of these edges, those edges will be added to *Future_Candidate_List*.
- Now, report the edges of the selected triangles, which appear on the boundary of only one selected triangle.

In the next two subsections we will discuss the above two steps in detail.

2.1 Preprocessing

In the preprocessing step, with each pair of points $(u, v) \in S$, we attach possible empty triangles whose two vertices are u and v. Here we use a modified version of algorithm for

[1] whose all the three vertices are points of S and no other points of S are inside it.

finding *Empty_Convex_Polygons* by Dobkin, Edelsbrunner and Overmars [6], which is given below.

For each pair of points u and v in S,

1. keep two empty list *RightSideTriangles* and *LeftSideTriangles*.

2. Sort all other ($|S| - 2$) points in S by angle w.r.t. the line $\overline{uv}$ around u in counter clockwise order[2]. Let the sorted list be $SortedList = \{p_1, \ldots, p_{|S|-2}\}$ in the increasing order of angle.

3. ***if*** no point lies on line segment $\overline{uv}$ ***do***
   if any point lies on right side[3] of $\overline{uv}$, ***do***
   set i = ($|S| - 2$), add p_i to *RightSideTriangles* list,
   set *RefAngle*= $\angle uvp_i$, decrement i by 1
   while p_i is on right side of line $\overline{uv}$ ***do***
   if $\angle uvp_i < RefAngle$ ***do***
   add p_i to the *RightSideTriangles*, set *RefAngle*= $\angle uvp_i$,
   and decrement i by 1.
   endif
   endwhile
   endif
   if any point lies on left side[4] of $\overline{uv}$, ***do***
   set i = 0
   while area of $\triangle uvp_i$ is ***zero do***
   increment i.
   endwhile
   if p_i lies on left side of $\overline{uv}$ ***do***
   add p_i to *LeftSideTriangles* list.
   set *RefAngle*= $\angle uvp_i$, increment i by 1.
   endif
   while p_i is on left side of line $\overline{uv}$ ***do***
   if $\angle uvp_i < RefAngle$ ***do***
   add p_i to the *LeftSideTriangles*, set *RefAngle*= $\angle uvp_i$ and
   increment i by 1.
   endif
   endwhile
   endif
   endif

---

[2]If two points make the same angle with $\overline{uv}$ at u, the point which is near to u will come first in the *SortedList*, if angle made by them is greater then π else the point which is far from u will come first in the *SortedList*.

[3]angle made is greater then π

[4]angle made is less then π

Here we observe that when we sort all other points by angle around the point u with respect to line $\overline{uv}$, the points which makes angle less than π lies on left side and the points which makes angle greater then π lies on right, where the orientation of the line $\overline{uv}$ is from u to v.

2.2 Generation of Polygon

In this step we generate a simple polygon randomly using the empty triangles generated in the preprocessing step. We keep two lists *Polygon_Edge_List* - to keep the edges of simple polygon as it is growing, either clockwise or counter clockwise direction[5] and *Future_Candidate_List* - to keep edges of triangles chosen which still has some empty triangles associated with them.

The algorithm for generation of simple polygon is as follows:

Step–I Choose a pair of points u and v randomly from S, and hence edge $\overline{uv}$.

Step–II Choose a empty triangle $\triangle$, associated with edge $\overline{uv}$ randomly. Add the three edges of the $\triangle$ to *Polygon_Edge_List* in counter clockwise direction. Drop all the triangles associated with these three edges of $\triangle$, which are overlapping with $\triangle$. If any of these edges are still associated with one or more triangles, those edges are added to *Future_Candidate_List*.

Step–III set i = 1

while i $<$ ($|S| - 2$) ***do***

A: Choose a edge e randomly from *Future_Candidate_List*

B: Choose randomly a triangle $\triangle$, if exist, which is still associated with edge e, and does not overlap with triangles chosen in previous steps (i.e. not intersecting polygon generated so far), also third vertex of $\triangle$is not the end vertex of any edge in *Polygon_Edge_List*.
delete e from *Future_Candidate_List*.
if No such triangle $\triangle$ exist for e, ***do***
goto A:
endif.

C: Delete edge e from *Polygon_Edge_List*, and insert other two edges of $\triangle$ to *Polygon_Edge_List* thus we get a polygon with i+3 edges in counter clockwise order. Delete all the triangles associated to the newly inserted edges of $\triangle$ in the *Polygon_Edge_List*, which overlaps with $\triangle$. If any of these edges are still associated with atleast one triangle, we add those edges in *Future_Candidate_List*.
increment i by 1.

endwhile

Step–IV output the edges in *Polygon_Edge_List*.

[5] we use counter clockwise direction.

Whenever an algorithm for generation of random polygon is proposed, the first question to ask is, whether it generates all possible simple polygons uniformly and randomly on given set of points. Note that, given a set of n points, any simple polygon with those points as vertices, can be split into empty triangles whose vertices are the vertices of the polygon. Thus, every possible simple polygon with these n vertices can be mappe to at least one set of empty triangles such that each triangle is adjacent with at least one other triangle in that set.

In the preprocessing step, we generate all empty triangles and associate each triangle with three of its edges. In the reporting step (i.e., generation of random polygon), we generate a random sequence of edges and a triangle corresponding to each edge such that we get a set of triangles in which each triangle is adjacent with at least one other triangle in that set. Thus, the outer boundary of union of those triangles gives us a simple polygon.

The second question which we have to answer is, on a given set S of n points after the preprocessing, is it possible to get a simple polygon of n edges with the random selection of edges and empty triangles as proposed in step 2? The answer is NO. Considered the following example Figure 1 of 10 points. In our random sequence of selection of edges and empty triangles, the selected triangles are in the order a,b,c,d,e,g,f. Then we can observe that there exist no empty triangle by which we can include point 5 to the simple polygon generated on remaining 9 points.

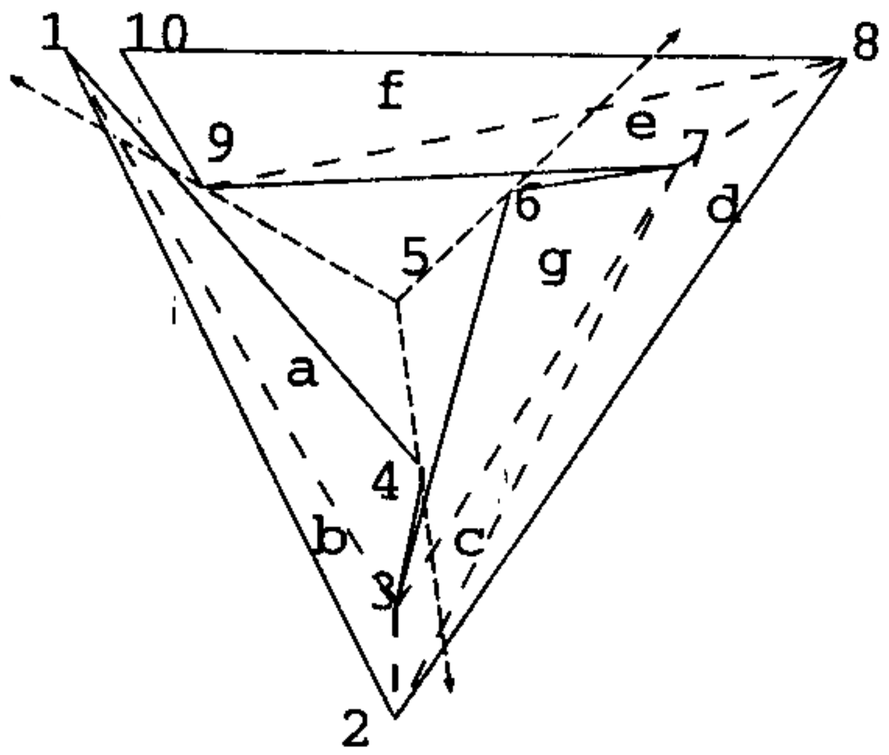


Figure 1: Example of random selection of empty triangles which doesn't yield simple polygon on given set.

So we have to replace the step A of Step-III by

A: ***if*** *Future_Candidate_List* is *empty*
    goto the Step-I and Start fresh.
endif
Choose an edge e randomly from *Future_Candidate_List*.

2.3 Implementation details and Complexity

Here we consider the implementation issues and complexity of algorithm in two sections. In first section we discuss the complexity of preprocessing step. In the next section we discuss the implementation and complexity of random simple polygon generation step.

2.3.1 Preprocessing Step:

Consider the set S of n points. There are $O(n^2)$ pair of points. In preprocessing step, sorting of the other points by angle around a point u, with respect to an edge $\overline{uv}$, can be done in $O(n \log n)$ time. Computation of all possible empty triangles associated with $\overline{uv}$ can be done by scanning the *SortedList* only once; so the time required is $O(n)$. So, for each pair of points u and v, we can associate all possible empty triangles in $O(n \log n)$ time. Since there are $O(n^2)$ pairs, we can associate all possible empty triangles to all pairs of points in S, in $O(n^3 \log n)$ time.
Using the results in [5, 7], it is possible to do the sorting around all the points simultaneously in time $O(n^2)$. So with small modifications in the preprocessing algorithm, we can associate all possible empty triangles to each pair of points in a given set S, in $O(n^3)$ time.

On a set of n points, the number of empty triangles associated with a pair of points is $O(n)$ [6]. Since with each pair of points we are keeping all possible empty triangles, each empty triangle appears exactly 3 times in the list of empty triangles. As the number of empty triangles on a set of n points is $O(n^3)$, the space required will be $O(n^3)$.

2.3.2 Generation of simple polygon

We implemented above algorithm and tested on different instances. We used double link list data structure to store the edges of growing simple polygon, i.e., *Polygon_Edge_List*, and an array *Future_Candidate_List*, which contains pointers to the edges in *Polygon_Edge_List*. To generate random points on plane and for the selection of edges and triangles randomly, we used *rand()* function in standard C library.

Now we will discuss the complexity issues. With each edge $\overline{uv}$ we kept the triangles which are on left and right side of $\overline{uv}$ separately, so whenever we choose an empty triangle $\triangle$ associated with an edge $\overline{uv}$ in Step-II or Step-III.B, we drop all the triangles associated with $\overline{uv}$ which are overlapping with $\triangle$ (i.e., in the same side of $\triangle$) in constant time. So, the complexity of the algorithm depends on the number of times Step-III.B is executed, where the selection of new empty triangle associated with edge $\overline{uv}$, which is non-intersecting with the simple polygon generated so far, is done.

Now we discuss about Step-III.B in more detail. Whenever an edge $\overline{uv}$ is selected randomly in Step-III.A, few empty triangles, associated with $\overline{uv}$, may overlap with the interior of the polygon, generated so far. In Figure 2, empty triangles attached to $\overline{uv}$ are numbered

[6] This bound will achieve to each pair if all points in the set are on convex hull boundary of that set

from 1 to 11, and the shaded area indicates the interior of the already generated polygon. Note that, some of those triangles intersect the shaded area.

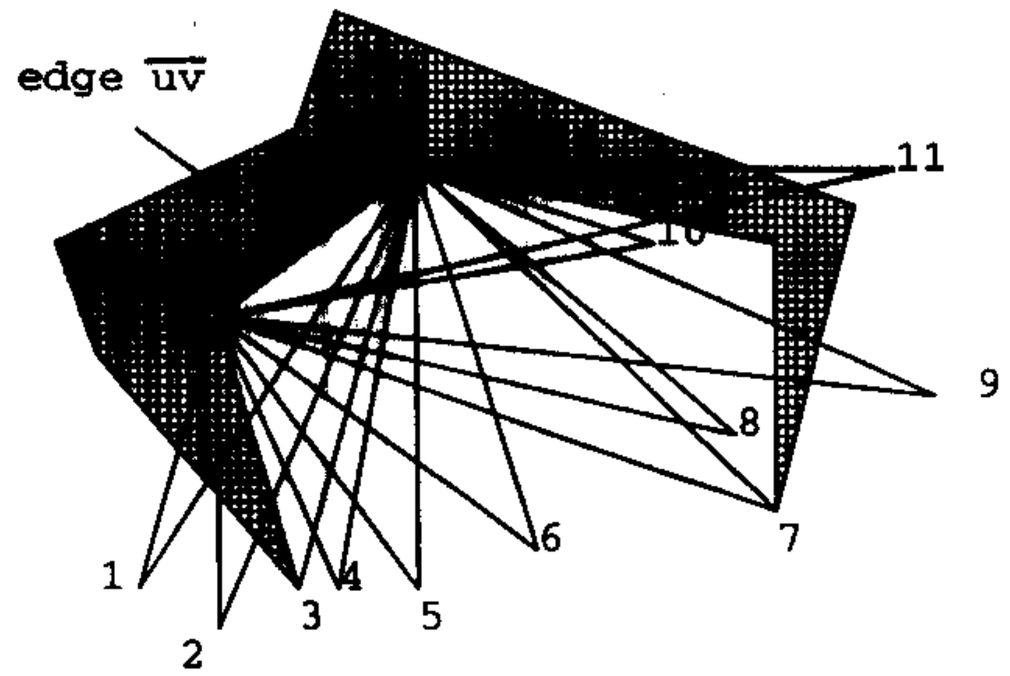


Figure 2: Empty triangles associated with edge $\overline{uv}$ and polygon generated so far.

Since the empty triangles associated with $\overline{uv}$ are ordered with respect to angles, we can drop few triangles in $O(\log n)$ time. For example, the triangles 1, 2, 3, and 11, associated with edge $\overline{uv}$, which are intersected by the edges adjacent to $\overline{uv}$ in *Polygon_Edge_List* in $O(\log n)$ are the members in this subset. But it may be possible that we may choose a triangle $\triangle$ (say triangle 9) in Step-III.B, which may be intersected by a polygonal edge which is not adjacent to edge $\overline{uv}$ in *Polygon_Edge_List*. Detecting such an edge (with respect to $\triangle$) requires $O(n)$ time in the worst case. If such an event occurs, we have to drop $\triangle$ from the empty triangle list associated with $\overline{uv}$, and we have to select another triangle randomly among the remaining, if any, associated with $\overline{uv}$.

Step-III.B is executed atleast $|S| - 3$ many times. At i_{th} iteration of Step-III.B, we choose a triangle $\triangle$ still associated with edge $\overline{uv}$ randomly and we check whether the chosen triangle $\triangle$ intersects with the polygon so far generated. This needs $O(i)$ time since there are at most i+2 edges in the already generated polygon. If such an intersection is empty, then we may proceed for inclusion of another point (if any). Thus, the lower bound on the complexity of our algorithm is $\sum_{i=1}^{|S|-3} i = O(n^2)$.

In the Step-II and Step-III.C, we add the edges of triangle, choosen in Step-III.B, which are still associated with atleast one empty triangle, after droping the triangles associated with them which are overlapping with the choosen one, to the *Future_Candidate_List*. Since we are choosing exactly n-2 triangles, we are adding atmost $O(n)$ edges to the *Future_Candidate_List*. So, in worst case $O(n)$ triangles associated with each of the edge in *Future_Candidate_List* will be checked in Step-III.B. Since the checking in Step-III.B will take $O(n)$ time for each of the triangle, the worst case complexity of our algorithm will be $O(n^3)$.

In the next section we are giving experimental results, like, CPU-consumption, average time taken to generate one random simple, etc.

3 Experimental Results

We implemented our algorithm on Sun Server 3000 running on Sun Solaries 2.6, and run on randomly generated point sets of varying sizes (n). For each n, we have generated three sets of points. For each such example, we computed 50 different simple polygons.

A set of 50 random simple polygons are generated on each of the above mentioned data sets, after computing all empty triangles, and avarege time (in milliseconds) taken to generate a simple random polygon is plotted against number of points which given in Figure 3.

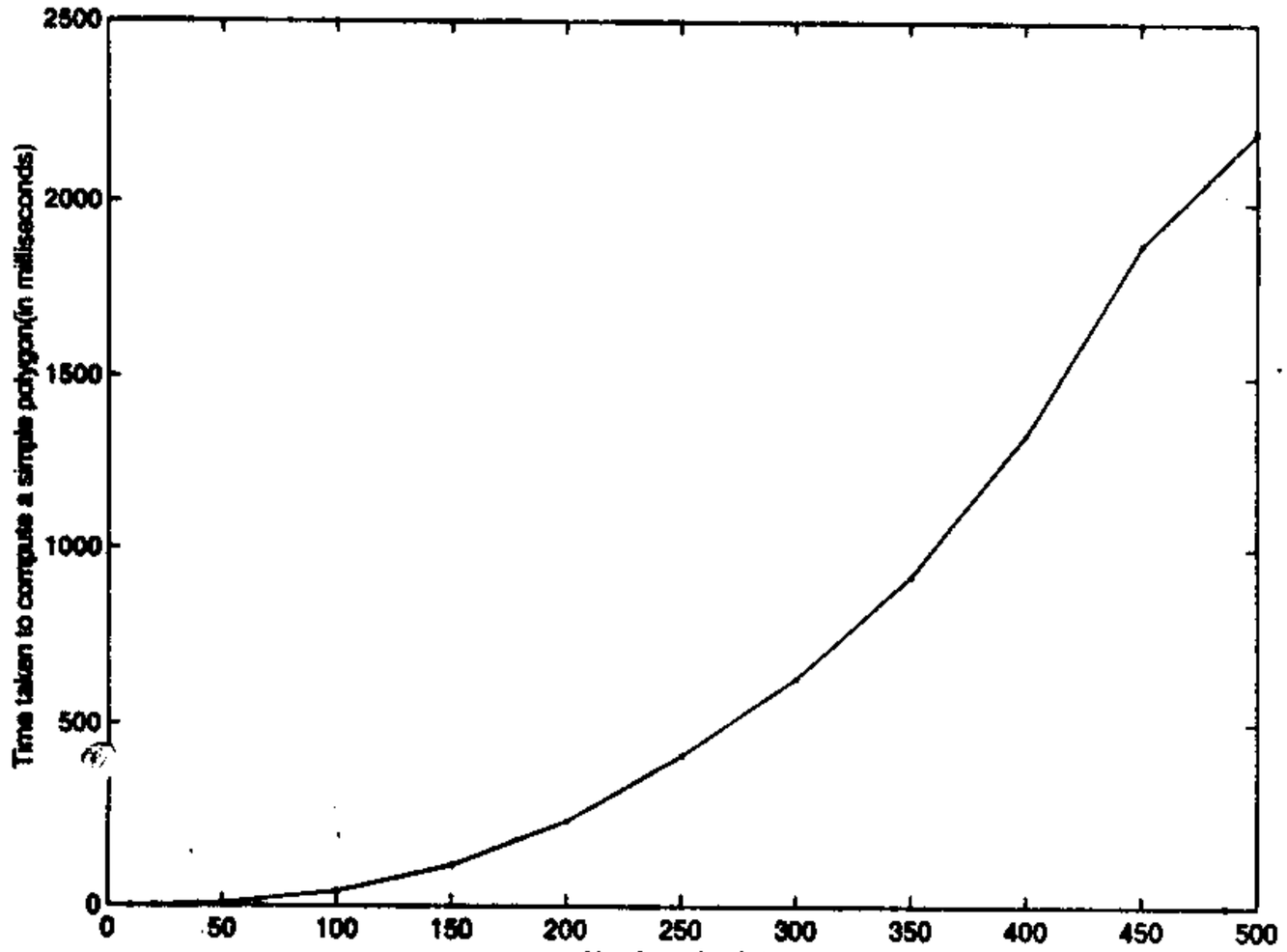


Figure 3: Time taken by CPU against No. of points.

After the preproscessing (i.e., computing all possible empty triangles) on a data set of 500 points, the average time taken to generate a simple random polygon is around 2 seconds.

At the end of the this report we included snapshots of randomly generated simple polygons by our algorithm on same data set.

4 Visibility Graph of Simple Polygon

Let P be a simple polygon with n vertices. The visibility graph $G = (V, E)$ of polygon P is an undirected graph whose vertex set V is vertices of polygon P and edge set E is set of vertex pairs, such that a pair (u, v) in E means the line segment $\overline{uv}$ lies completely inside the polygon P and doesn't intersect any edges of polygon P.

The naive method is, consider all possible $O(n^2)$ pairs of vertices of G and check their intersection with n edges of P, which leading to $O(n^3)$ running time complexity.

Here we are discussing a algorithm given in book by Overmars et. al.[4], which takes $O(n^2 \log n)$ time to compute the visibility graph of given simple polygon. The main idea is that, while computing visibility of all other vertices against a vertex p, in a simple polygon, to check the visibility of a vertex, it is better to use the information which we got while checking visibility of some previous vertex in some order. So here best way is to consider all the other vertices in clockwise direction with respect to vertex p.

A point w will be not visible to p if the line segment $\overline{pw}$ intersects atleast one edge of P or that line segment $\overline{pw}$ will be completely out side the polygon. So we consider a half line ρ from p and passing through w, ρ must hit a edge of P before it reaches w or it is fully out side the polygon P.

While treating the vertices in cyclic order around p we therefore maintain the polygon edges intersected by ρ in a balanced search tree T. The leaves of T store the intersected edges in order: the leftmost leaf stores the first segment intersected by ρ, the next leaf stores the segment that is intersected next, and so on. Any interior node v, which guide the search in T, store the rightmost edge in its left subtree, as in the example Figure 4.

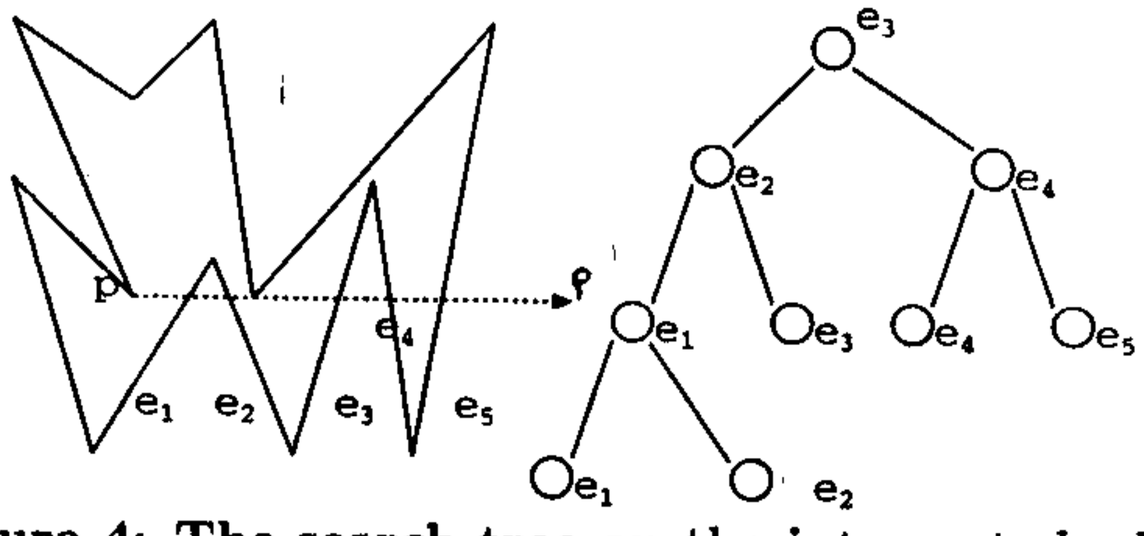


Figure 4: The search tree on the intersected edges.

Treating the vertices in cyclic order effectively means that we rotate the half-line ρ around p. So the approach is *rotational plane sweep*. The events in the sweep are the vertices of P. While swapping the half-line ρ, whenever it touches a vertex w, we check for the visibility of w by searching the the tree structure T ,if not empty, else we check whether the half-line is outside or not. And at the same time we update the tree structure T.

The algorithm of computing the ***visibility graph*** is as follows:

1. Initialize a Graph $G = (V, E)$ where V is the set of all vertices of simple polygon P, and $E = 0$.
2. ***for*** all vertices of P

3. Sort all the other vertices of P according to clockwise angle that the half-line from p to each vertex makes with positive x-axis.In case of tie, vertices closer to p should come before the vertices farther from p. Let $\{ w_1, \ldots, w_n \}$ be the sorted list.

4. Let ρ be the half-line parallel to positive x-axis starting at p. Find the edges of polygon P that are properly intersected by ρ, and store them in a balanced search tree T in the order in which they are intersected by ρ.

5. ***for*** $i \leftarrow 0$ ***to*** n
6. ***do if Visible***(w_i) ***then*** add a edge $\overline{pw_i}$ in E
7. Insert the edges of P incident to w_i that lie on the clockwise side of the half-line ρ from p to w_i into tree T.
8. Delete the edges of P incident to w_i that lie on the counter clockwise side of the half-line ρ from p to w_i from tree T.
9. ***endif***
10. ***endfor***
11. ***endfor***

The subroutine ***Visible*** must decide whether a vertex w_i is visible to p. It returns ***true*** if w_i is visible to p. There are three cases we have to consider in this ***Visible*** routine. They are, 1) The edge $\overline{pw_i}$ may be completely out side the polygon, 2) the vertex w_{i-1} is on the line $\overline{pw_i}$, 3) the vertex w_{i-1} is not the line $\overline{pw_i}$.

Since we keep the polygon edges either in clockwise or counter clockwise direction, we can easily decide case 1. In case 2) we once again have two cases *i*) w_{i-1} is visible to p, in which case we have check in the tree T for an edge, if any, intersecting $\overline{pw_i}$, *ii*) w_{i-1} is not visible to p, then w_i is also not visible to p, In case 3 we have to check in T for an edge, if any, which intersects the line segment $\overline{pw_i}$.

The algorithm for ***Visible*** is as follows:

1. ***if*** the edge $\overline{pw_i}$ lies completely outside the polygon
2. ***return false***.
3. ***elseif*** $i = 1$ or w_{i-1} is not on the edge $\overline{pw_i}$
4. search in T for the edge e in the leftmost leaf.
5. ***if*** e exists and $\overline{pw_i}$ intersects e
6. ***return false***
7. ***else***
8. ***return true***
9. ***endif***
10. ***elseif*** w_{i-1} is not visible to p
11. ***return false***
12. ***else*** search in the tree T for an edge e that intersects $\overline{w_i w_{i-1}}$
13. ***if*** e exists
14. ***return false***

```
15.        else
16.            return true
17.        endif
18.    endif
```

4.1 Complexity

Since balanced search is used, visibility of a vertex w_i to p in ***visible*** takes $O(\log n)$ time. So steps **5** to **10** in ***visibility graph*** takes $O(n \log n)$ time. Also step **3** in ***visibility graph*** takes $O(n \log n)$ time. Therefore the visibility of all other vertices to vertex p takes $O(n \log n)$ time. Since there are n vertices, the total complexity is $O(n^2 \log n)$. The space complexity depends on how we are storing visibility graph G. If we use matrix then complexity is $O(n^2)$, otherwise if we use link list, then it is $O(|E|)$.

5 Conclusion

In this report we considered generation of random simple polygon, and its visibility graph. We given a algorithm for generation of simple random polygon. Though the worst case complexity of our algorithm is $O(n^3)$, CPU-utilization is very less. This algorithm will be very help full for practical purposes, since we can generate many number of simple random polygons on a given data set, after one preprocessing.

References

[1] T. Asano, T. Asano, L. Guibas, J. Hershberger, and H. Imai. Visibility of Disjoint Polygons. *Algorithmica*, 1(1986), pp.49-63.

[2] T. Auer and M. Held. Heuristics for the Generation of Randon Polygons. In *Proc. 8th Canad. Conf. Comput. Geom.*, pages 38-44, Ottawa, Canada, Aug 1996. Carleton University Press

[3] T. Auer. Heuristics for the Generation of Randon Polygons. Master's thesis, Computerwissenschaften, U. Salzburg, A-5020 Salzburg, Ausrta, June 1996.

[4] M.d. Berg, M.v. Kreveld, M. Ovremars and O. Schwarzkopf Computational Geometry: Algorithms and Applications., Second, Revised Edition, *Spinger.*

[5] B. Chazelle, L. Guibas, and D. Lee. The power of Geometric Duality, *BIT* 25(1985), 76-90.

[6] D. P. Dobkin, H. Edelsbrunner and M. H. Overmars. Searching for Empty Convex Polygons. In *Proc. 4th ACM Symposium on Computational Geometry.*, pages 224-228, Urbana, Illinois, June, 1988.

[7] H. Edelsbrunner, J.O'Rouke, and R.Seidel. Constructing arrangements of lines and hyperplanes with applications. *SIAM J. Comput.* 15 (1986), 341-363.

[8] S.K.Ghosh and D.M.Mount. An output-sensitive algorithm for computing Visibility Graphs. *SIAM J. Comput.*, 20(1991), pp.888-910.

[9] J. Hershberger. An optimal visibility graph algorithm for triangulated simple polygons. *Algorithmica*, 4(1989), pp.172-182.

[10] J. O'Rourke and M. Viramani. Generating Random Polygons. Technical Report 011, CS Dept., Smith College, Northampton, MA 01063, July 1991.

[11] M.H.Ovremars and E.Welzl. New methods for constructing visibility graphs. In*Proc. 4th ACM Symposium on Computational Geometry*, urbana, IL, 1988, pp.164-171.

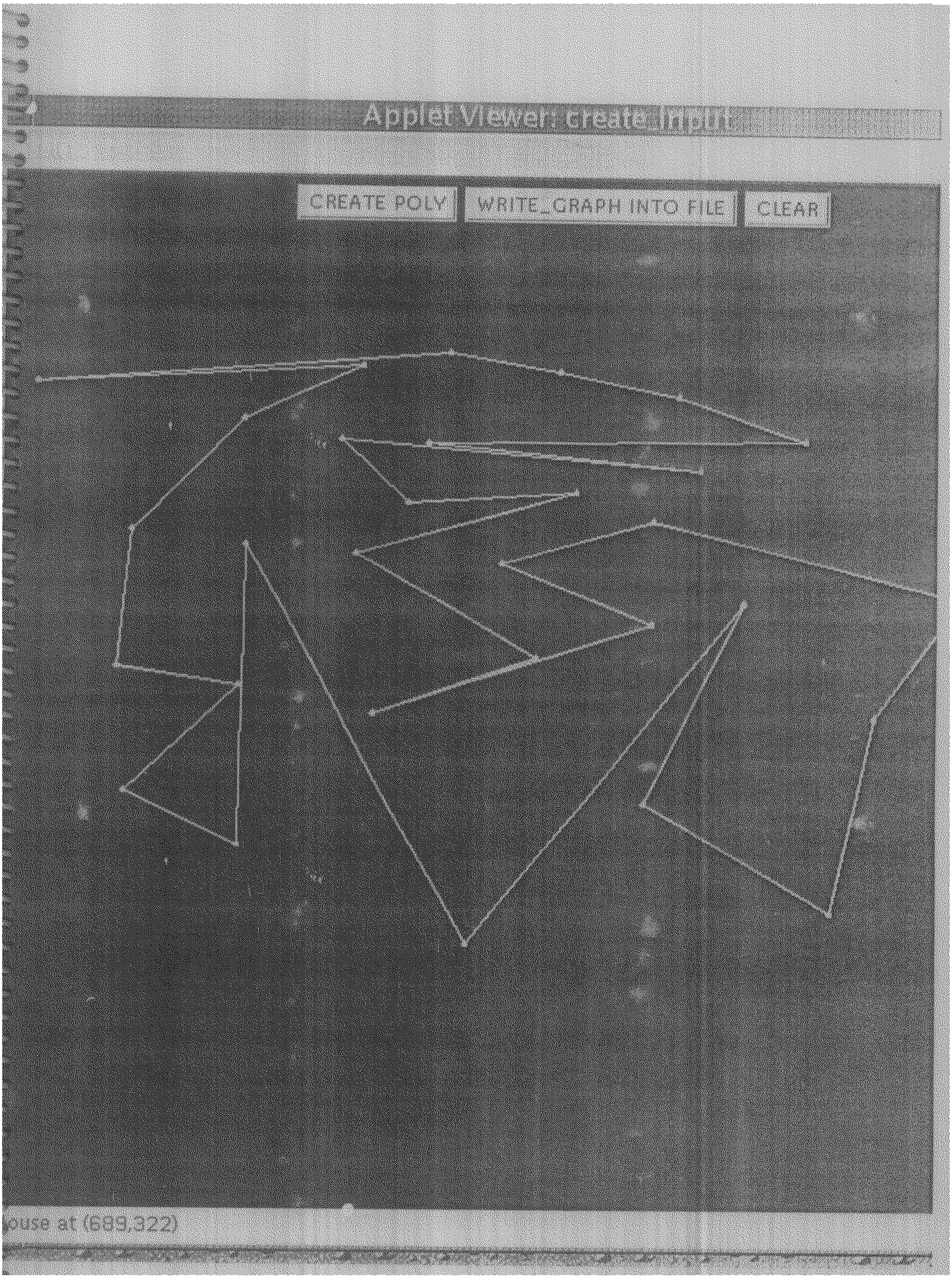
[12] M.Sharir and A.Schorr. On Shortest paths in polyhedral spaces, *SIAM J. Comput.*, 15(1986), pp.193-215.

[13] E.Welzl. Constructing Visibility Graph of n line segments in $O(n^2)$ time. *Infirm. Process.*, Lett., 20(1985), pp167-171.

[14] C. Zhu, G. Sundaram, J. Snoeyink, and J. S. B. Mitchel. Generating random Polygons with Given Vertices. In *Compt. Geom. Theory and Appl.*, 6(5):277-290, Sep 1996.

Applet Maven Graph Input

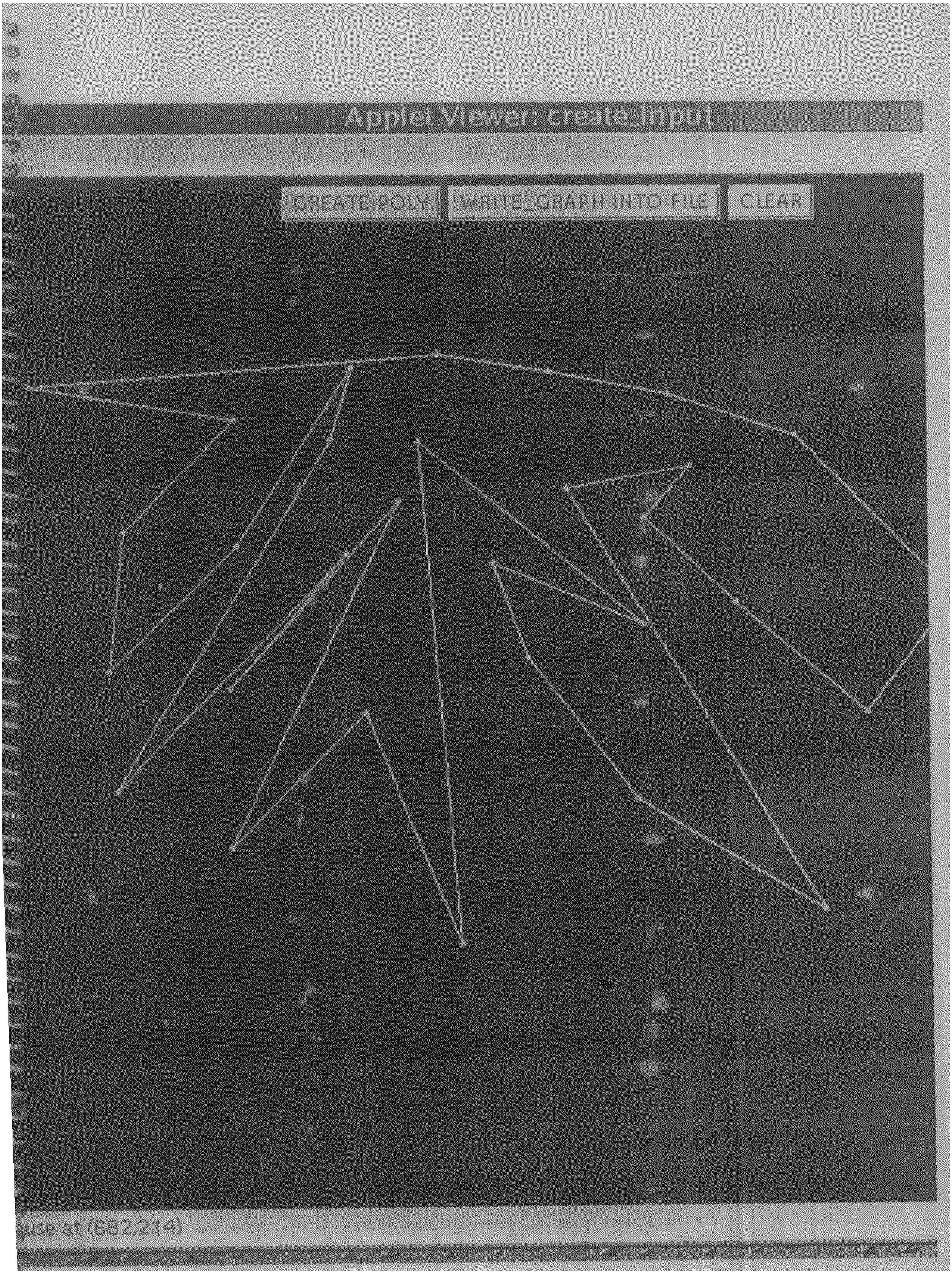
CREATE POLY | WRITE_GRAPH INTO FILE | CLEAR



Mouse at (689,322)

Applet Viewer: Create_Input

CREATE POLY | WRITE_GRAPH INTO FILE | CLEAR



use at (682,214)