International Journal of Computational Geometry & Applications
Vol. 4, No. 4 (1994) 475–481

TRIANGULATING DISJOINT JORDAN CHAINS*

REUVEN BAR-YEHUDA
Computer Science Department, Technion IIT, Haifa 32000, Israel

and

BERNARD CHAZELLE
Department of Computer Science, Princeton University, Princeton, NJ 08544
e-mail: chazelle@Princeton.edu

Received 22 August 1992
Revised 21 July 1994
Communicated by D. G. Kirkpatrick

ABSTRACT

Recent advances on polygon triangulation have yielded efficient algorithms for a large number of problems dealing with a single simple polygon. If the input consists of several disjoint polygons, however, it is often desirable to merge them in preprocessing so as to produce a single polygon that retains the geometric characteristics of its individual components. We give an efficient method for doing so, which combines a generalized form of Jordan sorting with the efficient use of point location and interval trees. As a corollary, we are able to triangulate a collection of p disjoint Jordan polygonal chains in time $O\left(n+p(\log p)^{1+\varepsilon}\right)$, for any fixed $\varepsilon > 0$, where n is the total number of vertices. A variant of the algorithm gives a running time of $O\left((n + p\log p)\log\log p\right)$. The performance of these solutions approaches the lower bound of $\Omega(n + p\log p)$.

Keywords: Simple polygon, triangulation, trapezoidal decomposition, visibility map, Jordan curves

1. Introduction

Simple polygons are among the most ubiquitous objects in computational geometry. As the elementary constituents of the discretized modeling of two-dimensional scenes, they occur naturally in computer graphics, vision, robotics, VLSI, numerical analysis, etc. Little can be done efficiently, however, unless the polygons are triangulated,[7] so a considerable amount of attention has been given to polygon triangulation over the years, e.g., see Refs. 1,2,3,4,5,6,8,11,14. In particular, Chazelle has shown that a simple polygon can be triangulated in linear time.[1]

Often we have to deal with not just one but a whole collection of disjoint simple polygons, or more generally, Jordan chains (i.e., non self-intersecting polygonal curves). For example, a polygon might have holes, or in the case of motion planning, a robot might be moving in a room with several polygonal obstacles, or we might be faced with a set of disjoint polygons as appearing in one layer of a VLSI mask. To triangulate a collection of disjoint Jordan chains, we add line segments so as to merge them into a single connected planar graph, which we can then triangulate in linear time. For this we use the observation[1] that a connected planar graph with straightline edges and no self-intersections can be regarded as a simple polygon by embedding it on a sphere and making a "thickening" of its edges the outside of the polygon.

In keeping with standard practice we do not work on triangulations directly but, instead, on visibility maps. A triangulation of a set P of disjoint Jordan chains is obtained by a maximal addition of non-intersecting line segments joining vertices of the chains. The *visibility map* of P, on the other hand, is the planar map obtained by extending two horizontal segments from each vertex, one in each direction, until each of them hits one of the chains, if at all. As is well-known[2,5] a triangulation can be derived from the visibility map in linear time.

Let P be a collection of p disjoint Jordan chains $C_1, \ldots, C_p$, with a total of n vertices. We define a *visibility tree* for P as follows: first, draw an infinite vertical line to the right of all the chains. Then, for each chain C_i in turn, pick a point in C_i (not necessarily a vertex) and draw a ray to the right until it hits either some C_j or the vertical line, and add the point hit as a new vertex. If we choose the origin of the rays carefully we can ensure that the resulting planar graph is connected and acyclic: in that case it is called a visibility tree for P (Figure 1).

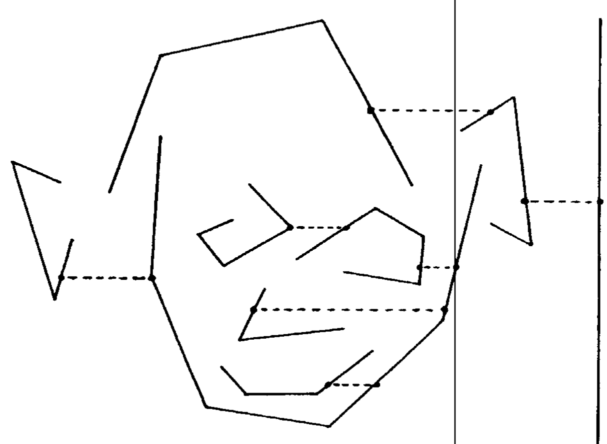

Fig.1

We give an algorithm for computing a visibility tree for P in time $O(n + p(\log p)^{1+\varepsilon})$, for any fixed $\varepsilon > 0$. A variant of the method runs in time $O((n + p \log p) \log\log p)$. The complexity of these algorithms comes close to the lower bound of $\Omega(n + p \log p)$ known for that problem. Note that with a visibility tree in hand, we can compute the complete visibility map of P, and from there, a triangulation of P in linear time.[1] Seidel[13] has given a probabilistic algorithm for triangulating disjoint Jordan chains with an expected running time of $O(n \log^* n + p \log p)$. His algorithm outperforms ours only if p is very close to n. Furthermore, our solution is deterministic. The approach we follow combines a generalized form of Jordan sorting[9] (which is interesting in its own right) with the efficient use of fast point location[4,10] and interval trees.[12]

2. Generalized Jordan Sorting

Let $x_1, \ldots, x_n$ be the intersections of an oriented Jordan curve C with a horizontal line ℓ, given in the order they occur along the curve (Figure 2.A). Jordan sorting is the problem of sorting the coordinates x_i: for this, a linear-time algorithm was proposed by Hoffmann et al.[9] We extend their result in the following manner. Given a collection of p disjoint Jordan curves with a total of n intersections with ℓ, sort the intersections along ℓ in time $O(n + p \log p)$. It is easy to see that this result is optimal by reduction from sorting.

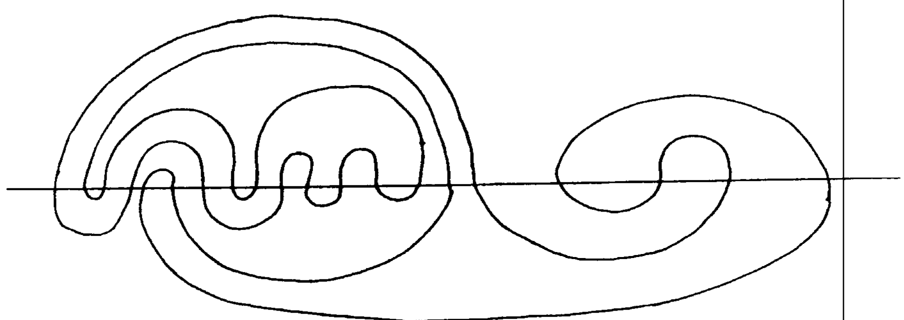

Fig.2A

We briefly review the linear algorithm of Hoffman et al.[9] and modify it to deal with several curves at once. We assume the reader's familiarity with Ref. 9. The line ℓ breaks up the curve C into disjoint arcs entirely below or above the line. The arcs above (resp. below) the line form a parenthesis system which can be represented by a tree T_a (resp. T_b), as indicated in Figure 2.B.

Each node represents an arc and its children correspond to the arcs directly nested within it. The left-to-right order of the child arcs are encoded in a finger tree (see also Ref. 14). The basic idea of the algorithm is to trace the curve and build T_a and T_b incrementally along the way. As the curve crosses ℓ and is about to enter, say, the top halfplane, we already know what arc α above ℓ "covers" the new arc β to be inserted. Let $\alpha_1, \ldots, \alpha_k$ be the child arcs of α: we must identify the interval of arcs $[\alpha_i \ldots \alpha_j]$ that β covers (α_2 and α_3 in the case of Figure 3), which we do by searching through the finger tree for the child arcs of α. Once the search

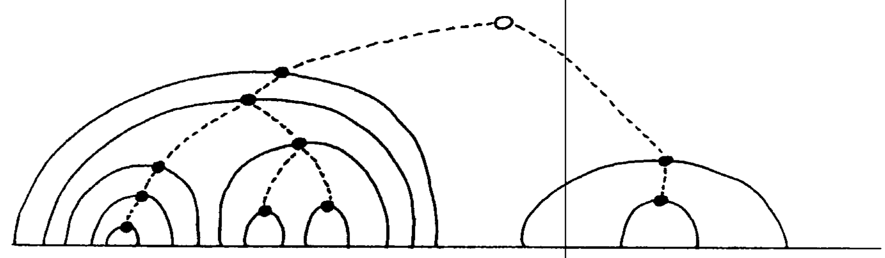

Fig.2B

is completed, we must update the tree T_a and the finger trees associated with the children of α and β. By keeping pointers between matching features in T_a and T_b, we can gain access to a finger in constant time and thus readily jump between the two trees. An amortized analysis shows that the whole algorithm runs in linear time.

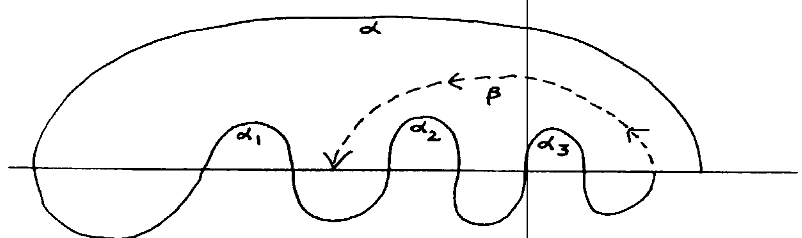

Fig.3

Let us now consider the case of a collection of p disjoint Jordan curves. First, we sort the curves by their leftmost intersections with ℓ and break each curve at that point, which produces at most $2p$ Jordan curves. We process the curves one by one as indicated above, retaining the current T_a and T_b as we switch from one curve to the next. We process the curves in reverse order, i.e., we begin with the curve whose leftmost intersection point is rightmost (breaking ties arbitrarily). In our analysis we can "pretend" that we are sorting a single curve, and the same time bound of $O(n)$ readily follows. Since we have to sort p numbers initially, the total running time is $O(n + p\log p)$, which is optimal.

Note that we can connect the p curves together by adding well chosen segments along ℓ: specifically, we connect the rightmost intersection of each curve to its successor (if it has one) in the sorted list of intersections (Figure 4). This obviously of T_i towards any T_j $(j > i)$ in $O(\log n)$ time. If a hit c occurs within ab, then we call c a "candidate." Among all the Jordan trees T_j $(j > i)$, we identify the leftmost candidate c, and if it exists, we replace ab by ac. Otherwise, we leave the connecting edge ab as is. We follow the same procedure for all the connecting edges of the Jordan trees $T_1, \ldots, T_{k-1}$, which takes a total of $O(n + p(\log p)\log n)$ time.

We claim that the resulting graph is a valid visibility tree for P. Because of the ray-shooting strategy no proper intersections are created, so it suffices to show that the graph is connected and acyclic. We establish this by induction on the number of nodes in $\mathcal{T}$. The case of a single node is obvious: the graph is the Jordan tree of a collection of Jordan chains intersecting the same horizontal line, with in addition an edge connecting it to the infinite vertical line.

Assume now that $\mathcal{T}$ has more than one node. Pick a leaf v and denote by ℓ_v the horizontal line associated with it. Let $C_1, \ldots, C_m$ be the Jordan chains at v sorted in left-to-right order of their rightmost intersections $r_1, \ldots, r_m$ with ℓ_v. To complete the induction, it suffices to show that no cycle can pass through any of the C_i (acyclicity) and that the C_i's are in the same connected component as the other chains (connectivity). This is easily established by induction on m. The key observation is that the only rays shot towards C_i emanate from the points r_j for $j < i$. This shows, in particular, that no ray can hit C_1. Exactly one ray is shot from C_i, namely, the one shot from its rightmost point r_i. These last two facts prove that no cycle can be part of C_1 and that this chain is connected to the rest of the graph. We remove C_1 along with the connecting edge emanating from r_1, which leaves us with $m - 1$ chains. This reasoning takes care of both the basis $(m = 1)$ and the inductive step. This proves our claim.

We can save time by reducing the number of point location queries. Conceptually the previous algorithm can be modeled as follows. For each i $(0 \le i < k)$ we are given p_i points which we must locate in the visibility maps of $T_{i+1}, \ldots, T_k$. Each point location requires $O(\log n)$ time and $\sum p_i \le p$. We have the possibility of merging several T_i's together and computing a point location structure for the resulting visibility map. Consider a balanced tree of degree d whose leaves are associated with $T_1, \ldots, T_k$ from left to right. Each internal node contains the visibility map (preprocessed for point location) of the merge of the T_i's stored at the leaves below. Let n_w and p_w be the total number of vertices and chains, respectively, associated with the leaves below node w. If all the information stored at the children of w is already available, it takes $O(n_w + p_w d \log n)$ time to compute and preprocess the visibility map at w.

Therefore, computing the visibility map at the root (which is our ultimate goal) takes time $O\big((n + pd\log n)\log_d k\big)$. Since $k = O(\log p)$, setting $d = k^\varepsilon$, for any fixed $\varepsilon > 0$, gives $O\big(n + p(\log n)^{1+\varepsilon}\big)$, which is also $O\big(n + p(\log p)^{1+\varepsilon}\big)$. Setting $d = 2$ gives $O\big((n + p\log p)\log\log p\big)$. It is a trivial exercise to show that $\Omega(n + p\log p)$ is a lower bound on the time required to compute the visibility map of P, and hence, by virtue of Ref. 1, on the time for computing any visibility tree for P.

Theorem It is possible to merge p simple Jordan chains with a total of n vertices and compute their visibility map in time $O((n+p\log p)\log\log p)$ or $O(n+p(\log p)^{1+\varepsilon})$, for any fixed $\varepsilon > 0$.

Acknowledgment: The first author wishes to thank R. Grinwald and A. Efrat for helpful discussions.

References

1. Chazelle, B. *Triangulating a simple polygon in linear time*, Disc. Comput. Geom. 6 (1991), 485–524.
2. Chazelle B., Incerpi, J. *Triangulation and shape-complexity*, ACM Trans. on Graphics 3 (1984), 135–152.
3. Clarkson, K., Tarjan, R.E., Van Wyk, C.J. *A fast Las Vegas algorithm for triangulating a simple polygon*, Disc. Comput. Geom. 4 (1989), 432–432.
4. Edelsbrunner, H., Guibas, L.J., Stolfi, J. *Optimal point location in a monotone subdivision*, SIAM J. Comput. 15 (1986), 317–340.
5. Fournier, A., Montuno, D.Y. *Triangulating simple polygons and equivalent problems*, ACM Trans. on Graphics 3 (1984), 153–174.
6. Garey, M.R., Johnson, D.S., Preparata, F.P., Tarjan, R.E. *Triangulating a simple polygon*, Inform. Process. Lett. 7 (1978), 175–180.
7. Guibas, L.J., Hershberger, J., Leven, D., Sharir, M., Tarjan, R.E. *Linear time algorithms for visibility and shortest path problems inside triangulated simple polygons*, Algorithmica 2 (1987), 209–233.
8. Hertel, S., Mehlhorn, K. *Fast triangulation of a simple polygon*, Proc. Conf. Found. Comput. Theory, New York, Lecture Notes on Computer Science 158 (1983), 207–218.
9. Hoffman, K., Mehlhorn, K., Rosenstiehl, P., Tarjan, R. *Sorting Jordan sequences in linear time using level-linked search trees*, Inform. and Control 68 (1986), 170–184.
10. Kirkpatrick, D.G. *Optimal search in planar subdivisions*, SIAM J. Comput. 12 (1983), 28–35.
11. Kirkpatrick, D.G., Klawe, M.M., Tarjan, R.E. *Polygon triangulation in $O(n\log\log n)$ time with simple data structures*, Disc. Comput. Geom. 7 (1992), 329–346.
12. Mehlhorn, K. *Data Structures and Algorithms 3: Multidimensional Searching and Computational Geometry*, Springer-Verlag, Heidelberg, Germany, 1984.
13. Seidel, R. *A simple and fast incremental randomized algorithm for computing trapezoidal decompositions and for triangulating polygons*, Comput. Geom.: Theory and Appl. 1 (1991), 51-64.
14. Tarjan, R.E., Van Wyk, C.J. *An $O(n\log\log n)$-time algorithm for triangulating a simple polygon*, SIAM J. Comput. 17 (1988), 143–178.