Incremental Generation of Voronoi Diagrams of Planar Shapes

Martin Held*

1 Introduction

Consider a planar, bounded, and closed area $\mathcal{A}$ which is simply-connected, and suppose that its boundary C consists of n straight lines and circular arcs. For every point $p \in \mathcal{A}$, the *(contour) clearance* $d(p, C)$ is defined as the minimum Euclidean distance $d(p, q)$, where q is on the boundary C. The *clearance disk* is the disk with radius $d(p, C)$ centered at p. The *internal skeleton* with respect to C and $\mathcal{A}$ is the set of points out of the interior of $\mathcal{A}$ whose clearance disks touch C in at least two disjoint points. From this structure the *Voronoi diagram* $\mathcal{VD}(C)$ of C is derived by adding straight line segments at the reflex vertices of C (normal to the incident boundary segments).

In the following we describe a fast incremental algorithm for computing the Voronoi diagram of a Jordan curve C consisting of straight lines and circular arcs. For a detailed survey of the algorithm and its analysis we refer to [Hel93]. The algorithm's underlying incremental construction scheme was first sketched by Persson [Per78].

The algorithm requires only $O(n)$ arithmetic floating-point operations for the generation of Voronoi diagrams of the boundaries of so-called monotonous areas, which are a generalization of convex areas; in addition to up to $O(n \log h)$ computationally less expensive comparisons and assignments of floating-point values spent on maintaining a priority queue of size $h \leq n$. For general shapes these worst-case complexities have to be multiplied by a shape-dependent factor, which is small in all except perhaps very few practically relevant applications. Practical tests gave clear evidence that this incremental algorithm usually is significantly faster than Lee's divide&conquer algorithm [Lee82].

2 Incremental Algorithm

2.1 Outline of the Algorithm

In the first step of the algorithm a bisector $b(s_1, s_2)$ between every pair of consecutive contour segments (s_1, s_2) is computed. We call these bisectors *contour bisectors* because they originate at the boundary contour, cf. Fig. 1. Secondly, for every triple (s_1, s_2, s_3) of consecutive contour segments, the corresponding contour bisectors $b(s_1, s_2)$ and $b(s_2, s_3)$ are intersected. If an intersection exists then both bisectors are terminated at the point of intersection and the intersection is associated with both bisectors.

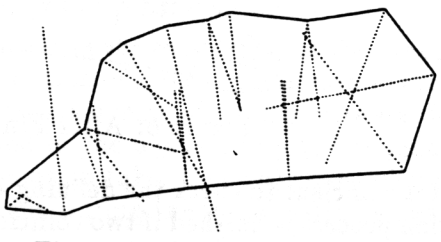

Figure 1: Contour Bisectors.

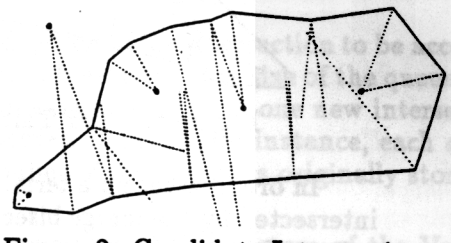

Figure 2: Candidate Intersections.

In the third step, a subset of all intersections is selected as possible candidates for the subsequent incremental merging. An intersection between two bisectors is a suitable *candidate* if and only if no other intersection with a smaller contour clearance is associated with any of the two bisectors.

*Universität Salzburg, Institut für Computerwissenschaften, A–5020 Salzburg, Austria.

All candidates are stored in a priority queue whose elements are ordered according to increasing clearance; i.e., the front end of the queue represents the intersection with smallest clearance. In Fig. 2, the candidate intersections are denoted by small bullets.

After the priority queue has been initialized the actual incremental construction starts. The front element is deleted from the queue and the corresponding intersection is *accepted* as a valid bisector intersection. Accepting a bisector intersection means that this point is used as a start point for the next merge bisector to be computed. This new merge bisector is defined by those contour segments defining the intersecting bisectors which do not define both of them; i.e., if the intersection between $b(s_1, s_2)$ and $b(s_2, s_3)$ is accepted, then the new merge bisector is defined by (s_1, s_3). Similarly to the divide&conquer algorithms, the new merge bisector $b(s_1, s_3)$ is intersected with the chains of bisectors incident in s_1 and s_3. If one intersection exists then this intersection is stored in the priority queue. If two intersections exist then the intersection encountered first when moving away from the start point (along the merge bisector) is stored.

An intersection needs only be stored if the actual front end of the queue corresponds to an intersection which has a smaller clearance than the new intersection. Otherwise, the new intersection can be accepted without any further manipulation of the queue. This subsequent accepting and computing of intersections until an insert into the queue is inevitable is called a *run*. Fig. 3 depicts the actual status of the Voronoi diagram under construction after the end of the first run.

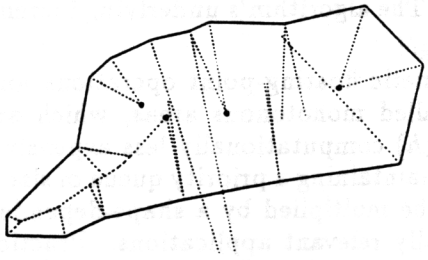

Figure 3: After First Run.

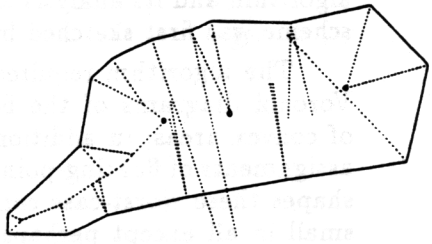

Figure 4: After Second Run.

Individual runs are carried out until the total Voronoi diagram is eventually constructed. In our example, Fig. 4 and Fig. 5 depict the situations after the end of the second respectively third run. Of course, at the start of a new run an intersection fetched from the priority queue may only be accepted if it still constitutes a *valid intersection*. Otherwise, it can be discarded and the next intersection has to be fetched from the priority queue. For instance, an intersection stored has become invalid if one of its associated bisectors has been discarded or shortened by an intersection with another bisector encountered during the construction process.

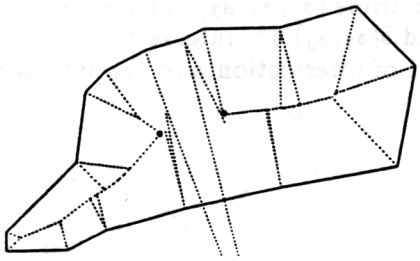

Figure 5: After Third Run.

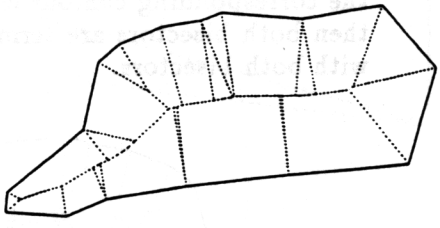

Figure 6: After Final Run.

In order to set up a termination condition it is sufficient to check the defining contour segments intersected by the merge bisector: the construction process is finished if two coinciding intersections exist and if the bisectors intersected by the merge bisector share a defining contour segment. Fig. 6 depicts the completed Voronoi diagram after the final run.

2.2 Voronoi Diagrams and the Concept of Monotonous Areas

The concept of monotonous areas was originally developed by the author a number of years ago when dealing with the automatic generation of tool paths for pocket machining. Monotonous areas

play a key role in the analysis of the algorithm presented here. In order to make this paper self-contained we therefore include a short review of monotonous areas. For a more detailed introduction we refer to [Hel91].

As illustrated in Fig. 7, we call a simply-connected and bounded area a *monotonous area* if its closed boundary contour can continuously and uniformly be shrunk to a point without splitting the area into separate subareas. Convex areas are also monotonous, but monotonous areas need not be convex-shaped – just think of a banana-shaped area. We call those points of a monotonous area which have maximal clearance *innermost points*.

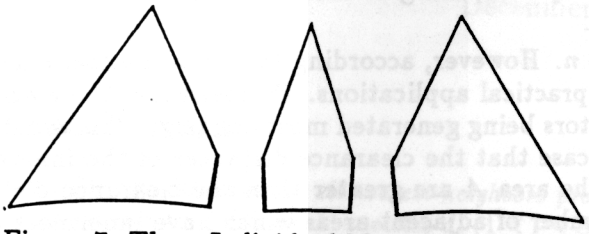

Figure 7: Three Individual Monotonous Areas.

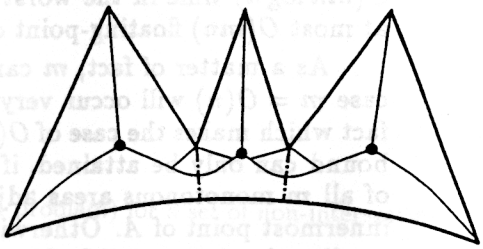

Figure 8: Subdivision Into Monotonous Areas.

An arbitrary multiply-connected planar area $\mathcal{A}$ can be partitioned into its monotonous subareas in time linear in the number n of its boundary segments, provided that the Voronoi diagram of its boundary is available. Fig. 8 depicts a simply-connected area subdivided into three monotonous areas; innermost points are depicted by bullets, straits between adjacent monotonous areas are depicted by dashed lines.

2.3 Complexity Results

What is the worst-case complexity of this incremental scheme? Assume that all nodes of $\mathcal{VD}(\mathcal{C})$ are of degree three and let us start with considering the boundary $\mathcal{C}$ of a monotonous area. It can be proved for monotonous areas that no bisector intersection accepted during the incremental merge process will be discarded by a latter merge step. In other words, any bisector ever constructed, or at least a portion of it, is guaranteed to be part of the final Voronoi diagram.

As a consequence, it is not necessary to scan more than two bisectors – one on the left-hand side and one on the right-hand side of the merge bisector – when looking for intersections: Suppose that an intersection between the merge bisector and some bisector encountered after repeated scanning would exist. The existence of an intersection would result in the discarding of at least one bisector intersection accepted previously, which cannot happen according to the remark stated above.

If there are k bisectors in the final Voronoi diagram, only k bisectors have to be handled during the entire construction process. Handling a bisector means accepting its start point, computing it, and intersecting it with exactly two other bisectors, and in addition possibly storing a new intersection point in the priority queue. Obviously, the arithmetic bisector operations can be carried out in constant time.

As far as the complexity of the queue manipulations – deletion of the intersection to be accepted and insertion of a new intersection – is concerned, one should observe that the size of the queue does not increase during the merge process: for each accepted intersection at most one new intersection is stored in the queue. As priority queues can be implemented as heaps, for instance, each queue manipulation takes $O(\log h)$ time, where h denotes the number of intersections originally stored in the queue during the initialization stage.

Hence, we conclude that our incremental algorithm computes all k bisectors of the Voronoi diagram of a monotonous area in time $O(k + r \log h)$, where r denotes the number of runs. In particular, only $O(k)$ computationally expensive floating-point operations associated with the handling of bisectors have to be executed. The number k ranges between n and $4n - 3$. Since $r < k$ and $h \leq \lfloor n/3 \rfloor$, we get an overall worst-case complexity of $O(n \log n)$.

What about the complexity of applying the incremental algorithm to a general simply-connected area $\mathcal{A}$? First of all, one should observe that in the case of general areas it is no longer valid that

no accepted bisector intersections will ever be discarded during subsequent runs. Rather, as in the case of the divide&conquer algorithm the entire Lee/Drysdale scanning scheme – cf. [Lee82, Hel91] – has to be applied when looking for intersections in order avoid a re-scanning of bisectors; repeated scanning of bisectors may clearly result in a quadratic complexity even in the case of simply-shaped areas.

Now suppose that, for all monotonous areas A of $\mathcal{A}$, m is an upper bound on the number of monotonous areas adjacent to A. In other words, if the monotonous areas are regarded as nodes of a graph which are interconnected by edges if and only if they share common straits, then m is the degree of this graph. It is proved in [Hel93] that the incremental algorithm takes less than $O(mn \log h)$ time in the worst case, thereby computing at most $O(mn)$ bisectors and performing at most $O(mn)$ floating-point operations.

As a matter of fact, m can go up to n. However, according to our own experience, the worst case $m = O(n)$ will occur very rarely in practical applications. Furthermore, there exists another fact which makes the case of $O(mn)$ bisectors being generated most unlikely: This worst-case upper bound can only be attained, if at all, in case that the clearance distances of the innermost points of all m monotonous areas adjacent to the area A are greater than the clearance distance of the innermost point of A. Otherwise, the number of adjacent areas which have innermost points with smaller clearance can be subtracted from m. Usually, m is much smaller than n, and it does not increase when n is increased as long as the shape of the area under consideration is preserved.

Up to now we unfortunatetly did not succeed in carrying out a full-scale average-case analysis. However, the term $O(n \log h \log m)$ serves as a very rough upper bound on the average-case complexity, with at most $O(n \log m)$ bisectors being constructed and with at most $O(n \log m)$ floating-point operations being performed. As witnessed by our practical tests this bound still seems to be too crude for most practical applications.

3 Concluding Remarks

The incremental algorithm presented and a version of Lee's [Lee82] divide&conquer algorithm, cf. [Hel91], have been implemented and extensively tested. During our work on pocket machining Lee's divide&conquer algorithm formed the algorithmic basis for the generation of offset curves. Except for the procedure governing the overall strategy of the algorithms – either divide&conquer or incremental – both algorithms rely on the same code for performing the rest of their tasks.

Our tests clearly demonstrated that the incremental algorithm usually is much faster than Lee's divide&conquer algorithm, with savings of CPU-time reaching up to about 60 percent. In particular, for the incremental algorithm the ratio between n and the CPU-time consumed remains roughly constant when increasing n. This is a clear indication that a roughly linear growth of the CPU-consumption can be expected for practical applications of the incremental algorithm! The CPU-consumption of the divide&conquer algorithm, however, usually shows a non-linear growth.

Apart from its practical advantages this algorithm is also interesting from a theoretical point of view: we suspect that it is amendable for a generalization to 3D, which may clear the road towards the efficient generation of Voronoi diagrams of polyhedrons.

References

[Hel91] M. Held. *On the Computational Geometry of Pocket Machining*, volume 500 of *Lecture Notes in Computer Science*. Springer-Verlag, June 1991. ISBN 3-540-54103-9.

[Hel93] M. Held. Incremental Computation of Voronoi Diagrams of Planar Shapes. Technical Report CS-93-1.0, U. Salzburg, CS Dept., A-5020 Salzburg, Austria, Jan 1993.

[Lee82] D.T. Lee. Medial Axis Transformation of a Planar Shape. *IEEE Trans. Pattern Analysis and Machine Intelligence*, PAMI-4(4):363–369, 1982.

[Per78] H. Persson. NC Machining of Arbitrarily Shaped Pockets. *Computer Aided Design*, 10(3):169–174, May 1978.