A Minimalist's Implementation of the 3-d Divide-and-Conquer Convex Hull Algorithm

Timothy M. Chan*
School of Computer Science
University of Waterloo
Waterloo, Ontario N2L 3G1, Canada
tmchan@uwaterloo.ca

June 4, 2003

Abstract

We give a simple interpretation and a simple implementation of the classical divide-and-conquer algorithm for computing 3-d convex hulls (and in particular, 2-d Delaunay triangulations and Voronoi diagrams). The entire C++ code is under 100 lines long, requires no special data structures, and uses only $6n$ pointers for space.

1 Introduction

This article contains a complete 2-page C++ code that implements an optimal $O(n \log n)$ algorithm for one of the core problems in computational geometry: the construction of the convex hull of n points in 3-d. Applications for this problem are well-known and numerous. For example, we mention intersections of halfspaces, shape analysis and bounding volumes in 3-d, as well as Delaunay triangulations and Voronoi diagrams in 2-d, the latter of which in turn have applications to nearest/farthest neighbors, Euclidean minimum spanning trees, largest empty circles, etc. (see any standard computational geometry textbook for more information [13, 16, 26, 27, 29]).

Disclaimer: The intended goal here is not the *fastest* code for the 3-d convex hull problem. Many efficient, robust implementations are already available [1, 4, 19, 22, 33]. Our implementation in fact ignores degenerate cases and robustness issues (to keep code simple) and is hardly of "professional quality."

Our challenge, however, is in finding the *shortest* program with a reasonable worst-case performance. The motivation is pedagogical. Considering the fundamental nature of the convex hull problem, it would be nice to have a complete solution (from scratch) that students can "see" and understand. Simpler programs are easier to learn, adapt, and experiment with. For the 3-d convex hull problem in particular, a self-contained implementation also has less overhead compared to the use of existing code or library routines, due to the variety of output formats an application may desire (a list of vertices, a graph formed by the edges, a planar subdivision under various representation schemes, etc.)

*This work was supported in part by an NSERC Research Grant.

Revisiting known algorithms from the "minimalist" point of view can sometimes shed new light on old problems (for example, as in Andersson's work on balanced search trees [3]). For 3-d convex hulls, we discover a different interpretation of the classic divide-and-conquer algorithm, showing that it is not as difficult to implement as previously thought, even for the novice; the new description (see Section 3) can be understood by non-geometers.

2 A commentary on known 3-d convex hull algorithms

We first give a quick survey on the existing options, to assess and compare their relative ease of implementation. (Detailed explanations of these algorithms can be found in computational geometry texts.)

Brute-force methods. It is actually easy to write a complete program to compute the 3-d convex hull if the input is small (and non-degenerate). O'Rourke's book [27] gave concise code by testing each triple of points as a possible facet. The running time is $O(n^4)$; a slight variation improves this to $O(n^3)$ by instead testing each pair of points as a possible edge. Our implementation is 26 lines long.[1]

Gift-wrapping. The gift-wrapping algorithm [9, 29] computes the convex hull in $O(nh)$ time by generating facets one at a time via an implicit breadth- or depth-first search. Here and throughout, h denotes the number of output facets. At first glance, the algorithm appears to require a queue or stack for the graph search and a dictionary to prevent revisiting old facets. Actually, both data structures can be avoided in the 3-d case, by using recursion for depth-first search and a linear scan for checking duplicate facets (the cost is still $O(n)$ per facet). Our implementation is 38 lines long (again, assuming non-degeneracy).

Incremental methods. Incremental methods are conceptually simple and maintain the convex hull as points are inserted one at a time. To insert a point outside the current hull, old "visible" facets are identified and removed, and new facets (attached to the new point) are generated, by somehow locating an initial new facet and subsequently performing breadth- or depth-first search. Despite their $O(n^2)$ worst-case performance, incremental methods have gained popularity over the years, due to the discovery by Clarkson and Shor [11, 13, 26] that an optimal $O(n \log n)$ expected running time can be obtained by randomizing the order of insertion and maintaining a suitable pointer structure as points are inserted.

Incremental methods are actually not as straightforward to implement as one might think. First, the current hull needs to be stored in a planar subdivision structure to support the necessary graph traversal and maintenance operations. Many such structures have been proposed in the literature (e.g., doubly-connected edge lists, the winged-edge or the quad-edge representation, as well as triangle-based representations), but all require delicate manipulations of pointers and primitives. Then there is the problem of finding the initial new facet at each iteration (a point location problem); this requires storing an additional "conflict graph" or "history dag," if the optimal randomized versions are adopted (although there are simpler suboptimal approaches [25]). See [10, 14, 22, 27] for implementations.

[1] In this article, line counts include everything (e.g., I/O, declarations, etc.) but comments and blank lines. Source code is available at `http://www.cs.uwaterloo.ca/~tmchan/`.

Sweeping. Fortune [19, 13] gave a nice $O(n \log n)$ algorithm for computing 2-d Voronoi diagrams, based on a process that can viewed as tracking a 1-d parabolic front moving from left to right. This "sweep-line" algorithm requires a priority queue and a balanced search tree to ensure optimal worst-case performance (though Fortune's original implementation opted for a hash table instead of a search tree). Assuming the availability of these data structures, the algorithm is not hard to code up; in particular, a planar subdivision structure is not necessary. Unfortunately, the algorithm does not generalize to compute arbitrary 3-d convex hulls.

Divide-and-conquer. The divide-and-conquer method, proposed by Shamos and Hoey [32] for 2-d Voronoi diagrams and Preparata and Hong [28] for 3-d convex hulls, was the earliest algorithm to achieve $O(n \log n)$ running time. The algorithm resembles mergesort: divide the point set into two halves by a vertical plane, recursively compute the hull of each half, and merge the two hulls into one. To merge, new "bridge" facets along the dividing vertical plane are generated by a graph search, and old facets underneath are removed.

O'Rourke [27] commented in his popular book: "this algorithm is both theoretically important, and quite beautiful. It is, however, rather difficult to implement and seems not used as frequently in practice as other asymptotically slower algorithms..." (For these reasons, O'Rourke decided to illustrate instead the implementation of the incremental algorithm in his book; similarly, Fortune's survey [20] chose not to detail the divide-and-conquer algorithm for Voronoi diagrams.)

The divide-and-conquer algorithm has been somewhat overshadowed by Fortune's and Clarkson and Shor's discoveries. Indeed, its implementation appears to require not only a planar subdivision structure capable of the necessary stitching operations, but also the handling of some very tricky details for the bridge computation. Part of the blame is perhaps of a historical nature: the original presentations in both the Shamos–Hoey and Preparata–Hong paper were sketchy and did not address all the subtleties (O'Rourke [27] mentioned some potential problems). Lee and Schachter [24] gave a more detailed account of the divide-and-conquer algorithm for 2-d Voronoi diagrams. The first full description for 3-d convex hulls appeared in Edelsbrunner's book [16] and was 15 pages long. Day [12] described an implementation that was even lengthier (and worse, his code didn't appear to guarantee $O(n \log n)$ running time). An elegant implementation was finally given by Guibas and Stolfi [22], who demonstrated that much terser code is possible, if done carefully. A proper choice of planar subdivision structure is still a prerequisite, and in fact, a large part of Guibas and Stolfi's paper was devoted to the development of their quad-edge representation scheme. Another thorough implementation was done by Shewchuk [33], using a triangle-based representation scheme.

Although the divide-and-conquer algorithm is perceived as complicated, it is not only the sole deterministic optimal 3-d convex hull algorithm known in theory, but also the fastest in practice, according to experimental studies by Shewchuk [33] (see also [36]), especially if a variation by Dwyer [15] is adopted. From a robustness standpoint, the algorithm uses algebraic predicates of the lowest degree, in contrast to Fortune's algorithm, for example. (From an academic standpoint, it is also one of the best illustrations of the power of the divide-and-conquer paradigm, in the author's opinion.)

Other algorithms. In the literature, one can also find output-sensitive $O(n \log^2 h)$ algorithms [8, 18], optimal $O(n \log h)$ algorithms [7, 11], and parallel algorithms [2] for 3-d convex hulls, some using more sophisticated forms of divide-and-conquer. We omit their discussion, as these algorithms are more complicated to implement.

3 A new description of the divide-and-conquer algorithm

In this section, we present a new explanation of the 3-d divide-and-conquer algorithm that makes its details more transparent and intuitive. Except for the remarks at the end, our presentation is meant to be self-contained and accessible to readers with no background in 3-d computational geometry (just familiarity with 2-d convex hulls).

Viewing as a kinetic 2-d problem. We begin with the problem statement and basic observations (which one can take as definitions). The input is a 3-d point set $P = \{p_1, \ldots, p_n\}$, where $p_i = (x_i, y_i, z_i)$. For simplicity, we assume that the points are non-degenerate, e.g., no four points lie on a common plane, and no three points lie on a common vertical plane.

The output is the *convex hull* of P, a convex polytope made out of *vertices* (0-faces), *edges* (1-faces), and *facets* (2-faces), defined as follows. A *j-face of the lower hull* is a tuple of $j+1$ points in P that lie on some plane $z = sx + ty + b$ such that all other points in P lie above the plane. Faces of the *upper hull* are similarly defined. The faces of the convex hull are the faces of the upper and lower hull combined. By symmetry, it suffices to describe an algorithm for the lower hull.

Since it is easier to visualize in 2-d than in 3-d, we avoid dealing with a 3-d object by taking (an infinite number of) projections in 2-d. For each point p_i, define

$$\hat{p}_i(t) = (x_i, z_i - ty_i).$$

Let $\hat{P}(t) = \{\hat{p}_1(t), \ldots, \hat{p}_n(t)\}$. By a change of variable $y' = z - ty$, a point p_i is a vertex of the lower hull of P iff for some t, $\hat{p}_i(t)$ lies on some line $y' = sx + b$ while all other $\hat{p}_\ell(t)$'s lie above the line, i.e., iff $\hat{p}_i(t)$ is a vertex of the lower hull of the 2-d point set $P(t)$ for some t. Similarly, p_ip_j is an edge of the lower hull of P iff $\hat{p}_i(t)\hat{p}_j(t)$ is an edge of the lower hull of $\hat{P}(t)$ for some t. Therefore, our problem is reduced to the maintenance of the lower hull of $\hat{P}(t)$ as *time* t goes from $-\infty$ to ∞. In other words, we want to *make a movie of a kinetic 2-d lower hull*, as input points move vertically (at different but fixed speeds) in the plane. This is the setting in which we will study the problem; from now on, we can forget about (most of the) 3-d terminologies from the previous paragraph.

How is the movie represented? As points move, the kinetic 2-d lower hull gains and loses vertices. In an *insertion event*, a new vertex $\hat{p}_j$ just appears between $\hat{p}_i$ and $\hat{p}_k$ on the 2-d hull, creating two new edges $\hat{p}_i\hat{p}_j, \hat{p}_j\hat{p}_k$ and destroying the old edge $\hat{p}_i\hat{p}_k$. In a *deletion event*, a vertex $\hat{p}_j$ disappears from the 2-d hull, destroying two edges $\hat{p}_i\hat{p}_j, \hat{p}_j\hat{p}_k$ and creating a new edge $\hat{p}_i\hat{p}_k$. (In either case, the triple $p_ip_jp_k$ makes a facet of the 3-d lower hull.) Each point is inserted at most once and deleted at most once, so the total number of events (and thus facets) is at most $2n$. The desired movie can be specified by the sequence of events, sorted in time, along with the initial hull at $t = -\infty$.

Applying divide-and-conquer. Sort the points $\hat{p}_1, \ldots, \hat{p}_n$ in increasing x-coordinates (recalling that points only move vertically). We use divide-and-conquer to solve the kinetic 2-d problem. Recursively create a movie for the lower hull L of $\hat{p}_1, \ldots, \hat{p}_{\lfloor n/2 \rfloor}$ and a movie for the lower hull R of $\hat{p}_{\lfloor n/2 \rfloor + 1}, \ldots, \hat{p}_n$. We will create a movie for the lower hull A of all points $\hat{p}_1, \ldots, \hat{p}_n$ by a merging process.

The initial hull A can be computed from the initial hulls L and R by identifying the common tangent uv, called the *bridge*, and removing the part of L to the right of u and the part of R to the left of v. (See Figure 1.) The initial bridge uv can be found in $O(n)$ time by a simple iterative algorithm that starts with $u = \hat{p}_{\lfloor n/2 \rfloor}$ and $v = \hat{p}_{\lfloor n/2 \rfloor + 1}$ and repeatedly advances u leftward on L and

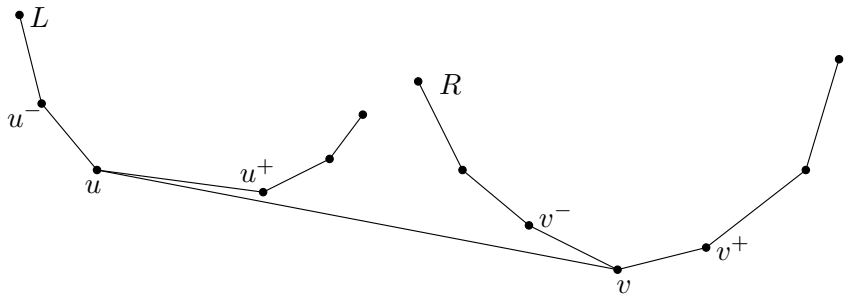

Figure 1: Merging two 2-d lower hulls.

v rightward on R, until u^-uv and uvv^+ both form counterclockwise turns. The notation w^- (resp. w^+) represents the vertex immediately to the left (resp. right) of w on either the hull L or the hull R.

As time progresses, we keep track of the current hulls L and R and the current bridge uv. The movies for L and R tell us when and how L and R change:

- If L undergoes an insertion or deletion of a vertex w, then A undergoes the same event provided that w is to the left of u.
- If R undergoes an insertion or deletion of a vertex w, then B undergoes the same event provided that w is to the right of v.

The pair uv remains a bridge as long as u^-uv and uvv^+ are counterclockwise turns and uu^+v and uv^-v are clockwise turns.

- When u^-uv turns clockwise, u^-v becomes the new bridge. Then A undergoes deletion of u (between u^- and v).
- When uu^+v turns counterclockwise, u^+v becomes the new bridge. Then A undergoes insertion of u^+ between u and v.
- When uvv^+ turns clockwise, uv^+ becomes the new bridge. Then A undergoes deletion of v (between u and v^+).
- When uv^-v turns counterclockwise, uv^- becomes the new bridge. Then A undergoes insertion of v^- between u and v.

To determine which of the above six possibilities should be executed next, we compute the six time values at which they occur and take the minimum. Continuing until there are no changes, we obtain the entire event sequence for the hull A. The description of the algorithm is complete and its correctness is self-evident.

The running time satisfies the standard mergesort recurrence:

$$T(n) = \begin{cases} O(1) & \text{if } n = 1 \\ T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + O(n) & \text{if } n > 1. \end{cases}$$

We conclude that the algorithm runs in optimal $O(n \log n)$ time.

Remarks. The Delaunay triangulation of a 2-d point set (and by duality, the Voronoi diagram) [13, 16, 29] can now be computed by setting $z_i = x_i^2 + y_i^2$.

The kinetic approach was popularized by Basch *et al.* [5]. Their first paper actually contained a discussion on how to maintain the 2-d convex hull for points moving in arbitrary directions (rather than vertically). This is a more general problem and requires near-quadratic number of events in the worst case.

The kinetic approach is related to the classical idea of a plane/space sweep. In fact, when translated to *dual* space [13, 16, 29] (in terms of the 3-d halfspace intersection problem or the 2-d Voronoi diagram problem), the time parameter corresponds precisely to one of the coordinates. So, our merging algorithm can indeed be thought of as a space sweep. The order of the sweep is different from Fortune's algorithm for Voronoi diagrams [19] but is identical to Seidel's shelling method for higher-dimensional convex hulls [31] (once dualized). Seidel's method performs only one pass to compute the entire hull (ours performs recursion), but uses a priority queue (ours avoids an explicit heap because of the recursion) and, more importantly, needs an initial $O(n^2)$ preprocessing to predict the insertion time of each vertex.

Compared to the original 3-d divide-and-conquer algorithm, the main difference that allows us to bypass planar subdivision structures is that we keep the facets (i.e., events) ordered by their time values.

More remarks, on querying. The divide-and-conquer algorithm is particularly suitable for answering 3-d extreme point queries (a dual point location problem [16, 29, 35]): given values s and t, find a point (x_i, y_i, z_i) minimizing $z_i - sx_i - ty_i$. Such queries can be used to find nearest neighbors in 2-d: given a point (a, b), minimizing the Euclidean distance $\sqrt{(x_i - a)^2 + (y_i - b)^2}$ is equivalent to minimizing $z_i - 2ax_i - 2by_i$ with $z_i = x_i^2 + y_i^2$.

We can answer such a query by "remembering" what the 2-d hull A looks like at time t and performing a 2-d extreme point query. By comparing s with the slope of the bridge uv, we can decide which side L or R to search. The approach leads to an $O(\log n)$ query time if appropriate pointers are kept between the event sequence of A and the event sequences of L and R; the data structure uses $O(n \log n)$ space.

This is analogous to the layered version of Lee and Preparata's chain method of point location [23] (as the bridges form a monotone chain in the dual plane), but we have managed to obtain a point-location data structure directly while the hull is being constructed. Guibas, Knuth, and Stolfi [21], for example, were interested in obtaining a data structure for 2-d nearest neighbor queries without the application of two separate methods, one for Voronoi diagram construction and one for point location; their randomized incremental approach gave an $O(n)$-size structure that guaranteed an $O(\log^2 n)$ expected query time only.

Remembering the past is another classical idea, called persistence. In fact, our version of the divide-and-conquer algorithm combines nicely with Sarnak and Tarjan's method of point location using persistent search trees [30]. This method achieves optimal $O(n)$ space and $O(\log n)$ query time, and requires the extraction of the sequence of changes to a list formed by moving a sweep line across the (dual) planar subdivision, but the output of our convex hull algorithm gives us precisely this sequence (in contrast, Fortune's sweep is done in a different order, so a separate sorting step is still required if Sarnak and Tarjan's method is to be used).

4 A minimalist's implementation

The algorithm description in the previous section is simple yet detailed enough that a competent programmer should be able to implement it without too much effort (we invite the "do-it-yourselfers" to try their own version). For example, event sequences can be stored in arrays, and the current hulls L and R can be stored as doubly linked lists. There is no need to learn a planar-subdivision representation scheme.

In this section, we comment on one particular implementation, given in the appendix, that aims at saving space (and is thus not the most straightforward version possible). This version requires only $6n$ pointers for working storage in addition to an array holding the input coordinates; the output is not stored explicitly. The amount is small, considering that an explicit array of facets or an array of edges would require $6n$ pointers already (recall that in the worst case there are about $3n$ facets and $2n$ edges). A quad-edge data structure, used in Guibas and Stolfi's implementation of the divide-and-conquer algorithm [22], would require $15n$ pointers (an edge points to 4 adjacent edges and an incident vertex); on the other hand, Shewchuk's triangle-based approach boasted an improvement to $12n$ pointers (a triangle points to 3 adjacent triangles and 3 incident vertices).

A guided tour of the code. Our 2-page implementation is completely self-contained. We even include code for sorting; appropriately, we choose mergesort (lines 40–53), and to avoid data movement and extra space, we choose a version that returns a linked list (using the `next` field).

The algebraic primitives (lines 30–38) are simple (testing whether three points form a clockwise turn at $t = -\infty$, and determining the time when three points switch from a clockwise to a counterclockwise turn or vice versa).

In the main recursive procedure `hull(list, n, A, B)`, the input is a sorted list of points `list` and the number of points `n`. The output is an array of $\leq 2n$ events stored in `A`; the array `B`, also of size $\leq 2n$, is used as temporary storage. The initial hull at $t = -\infty$ is stored from left to right as a doubly linked list using the `prev` and `next` fields.

To specify an insertion event fully, we appear to need a record of three points, to tell which pair of vertices $\hat{p}_i$ and $\hat{p}_k$ the new vertex $\hat{p}_j$ is inserted between. In our space-economical version, we decide to store just $\hat{p}_j$ in the event array `A`; but we will use the `prev` and `next` fields of $\hat{p}_j$ to point to $\hat{p}_i$ and $\hat{p}_k$ (since $\hat{p}_j$ is not in the initial hull, the fields have not been used). For a deletion event for vertex $\hat{p}_j$, we again store only $\hat{p}_j$ in the event array (as it turns out, an extra bit to differentiate insertions from deletions is unnecessary). As a result, the event array `A` is just an array of $\leq 2n$ points, representing the order in which points are inserted and/or deleted.

Using this compact storage scheme, pointer manipulations actually get easier. The base case (line 61) is trivial (unlike in Guibas and Stolfi's or Shewchuk's implementation). The two recursive calls are made in lines 63–66. The bridge at $t = -\infty$ is found in lines 68–71. In the main loop (lines 74–92), time t progresses and we maintain doubly linked lists for the two hulls L and R using the `prev` and `next` fields. At the same time, we track the bridge uv and generate the output event array `A`.

One problem arises with this compact scheme. During an insertion event, although we know which pair of vertices $\hat{p}_i$ and $\hat{p}_k$ a vertex $\hat{p}_j$ is inserted between, we cannot store $\hat{p}_i$ and $\hat{p}_k$ in the `prev` and `next` fields yet, because they are still in use in the linked list for L or R. This problem can be fixed by performing a second pass (lines 95–104), where time t goes backwards and the role of insertions and deletions is reversed. Here, we use the `prev` and `next` fields to maintain the doubly linked list for the output hull A. At the same time, we track the bridge uv. At the end, the `prev` and

`next` fields will store the positions where a vertex is last seen, i.e., inserted, if it leaves the list, and the list itself will store the surviving hull, i.e., the hull at $t = -\infty$. The desired output requirement is thus fulfilled. (If space is not crucial, this second pass can be eliminated by a more straightforward implementation.)

This version of the code prints the facets of the 3-d lower hull (lines 119–120). This is done by processing the events in the output array `A` and tracking the linked list for the 2-d hull using the same `prev` and `next` fields. The code can be modified to print the vertices and/or edges of the 3-d lower hull by a similar process. For example, we can do the following to print each edge exactly once:

```
for (u = list; u->next != NIL; u = u->next)
  cout << u-P << " " << u->next-P << "\n";
for (i = 0; A[i] != NIL; A[i++]->act())
  if (A[i]->prev->next != A[i]) {  // insertion event
    cout << A[i]->prev-P << " " << A[i]-P << "\n";
    cout << A[i]-P << " " << A[i]->next-P << "\n";
  }
  else cout << A[i]->prev-P << " " << A[i]->next-P << "\n";  // deletion event
```

Remarks. Excluding blank lines and comments, the code is 91 lines long and is significantly shorter than all implementations [1, 4, 10, 19, 22, 33] that we are aware of for 3-d convex hulls and 2-d Voronoi diagrams with comparable performance. To be fair, these implementations are designed not with the same "educational" purpose in mind, and neither for minimizing line counts.

Since our goal is not in optimizing speed, we will skip a full report on runtimes. We did test the program to compute 2-d Voronoi diagrams for 100,000 points uniformly distributed on the unit square; its output matched with the output of other programs, and it ran in under 20 seconds on a Sun Ultra 10.

There are some obvious ways to optimize the program at the expense of increasing its length; for example, in the main loop (lines 75–92), we can avoid computing all 6 time values at each iteration, because some have already been computed in the previous iteration. We don't believe, though, that our program could be made faster than Guibas and Stolfi's or Shewchuk's implementation in practice, for the following reason. In the original divide-and-conquer algorithm using planar subdivisions, the merging process requires only the examination of facets near the bridge, whereas in our version, the merging process must go through all facets of the left and right hull.

There are more degenerate cases in our version of the divide-and-conquer algorithm: two facets with the same time values can cause potential problems. We don't see an easy way to resolve these problems other than to apply a general perturbation scheme (such as [17]). For a robust implementation using exact arithmetic, and perhaps adaptive filters (e.g., see [34]), we have to deal with a more involved algebraic primitive, the comparison of time values. The degree of this primitive is higher than the degree in the original divide-and-conquer algorithm (4 instead of 3 for 3-d convex hulls, and 6 instead of 4 for 2-d Voronoi diagrams), but lower than in Fortune's sweep algorithm.

In terms of space, we can actually get by with $5n$ pointers under the same setup: in the final merge, don't store the output event array but print the answer instead (the code becomes longer, however).

5 Conclusions

By transforming a 3-d problem into a kinetic 2-d problem, we have given a more accessible description of the 3-d divide-and-conquer convex hull algorithm. This simpler description enables a short implementation that uses no special data structures other than arrays and linked lists.

It would be interesting to see minimalistic implementations for other basic problems in computational geometry. For 3-d convex hulls, it would be intriguing to find algorithms that use as little extra space as possible (as sought also by Brönnimann *et al.* [6]).

References

[1] The CGAL home page. `http://www.cgal.org/`.

[2] N. M. Amato, M. T. Goodrich, and E. A. Ramos. Parallel algorithms for higher-dimensional convex hulls. In *Proc. 34th IEEE Sympos. Found. Comput. Sci.*, pages 683–694, 1994.

[3] A. Andersson. Balanced search trees made simple. *Proc. 3rd Workshop Algorithms Data Struct.*, Lect. Notes in Comput. Sci., vol. 709, Springer-Verlag, pages 60–71, 1993.

[4] C. B. Barber, D. P. Dobkin, and H. T. Huhdanpaa. The Quckhull algorithm for convex hulls. *ACM Trans. on Math. Software*, 22:469–483, 1996. (Source code available at `http://www.geom.umn.edu/software/qhull/`.)

[5] J. Basch, L. J. Guibas, and J. Hershberger. Data structures for mobile data. *J. Algorithms*, 31:1–28, 1999.

[6] H. Brönnimann, J. Iacono, J. Katajainen, P. Morin, J. Morrison, and G. Toussaint. Optimal in-place planar convex hull algorithms. In *Proc. Latin American Sympos. on Theoretical Informatics*, pages 494–507, 2002.

[7] T. M. Chan. Optimal output-sensitive convex hull algorithms in two and three dimensions. *Discrete Comput. Geom.*, 16:361–368, 1996.

[8] T. M. Chan, J. Snoeyink, and C.-K. Yap. Primal dividing and dual pruning: output-sensitive construction of four-dimensional polytopes and three-dimensional Voronoi diagrams. *Discrete Comput. Geom.*, 18:433–454, 1997.

[9] D. R. Chand and S. S. Kapur. An algorithm for convex polytopes. *J. ACM*, 17:78–86, 1970.

[10] K. L. Clarkson, K. Mehlhorn, and R. Seidel. Four results on randomized incremental constructions. *Comput. Geom. Theory Appl.*, 3:185–212, 1993. (Source code available at `http://cm.bell-labs.com/netlib/voronoi/hull.html`.)

[11] K. L. Clarkson and P. W. Shor. Applications of random sampling in computational geometry, II. *Discrete Comput. Geom.*, 4:387–421, 1989.

[12] A. M. Day. The implementation of an algorithm to find the convex hull of a set of three-dimensional points. *ACM Trans. on Graphics*, 9:105–132, 1990.

[13] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, Heidelberg, 1997.

[14] O. Devillers. Improved incremental randomized Delaunay triangulation. In *Proc. 14th ACM Sympos. Comput. Geom.*, pages 106–115, 1998.

[15] R. A. Dwyer. A faster divide-and-conquer algorithm for constructing Delaunay triangulations. *Algorithmica*, 2:137–151, 1987.

[16] H. Edelsbrunner. *Algorithms in Combinatorial Geometry*. Springer-Verlag, Berlin, 1987.

[17] H. Edelsbrunner and E. P. Mücke. Simulation of simplicity: a technique to cope with degenerate cases in geometric algorithms. *ACM Trans. on Graphics*, 9:66–104, 1990.

[18] H. Edelsbrunner and W. Shi. An $O(n \log^2 h)$ time algorithm for the three-dimensional convex hull problem. *SIAM J. Comput.*, 20:259–277, 1991.

[19] S. Fortune. A sweepline algorithm for Voronoi diagrams. *Algorithmica*, 2:153–174, 1987. (Source code available at `http://cm.bell-labs.com/cm/cs/who/sjf/`.)

[20] S. Fortune. Voronoi diagrams and Delaunay triangulations. In *Euclidean Geometry and Computers* (D. A. Du and F. K. Hwang, eds.), pages 193–233, World Scientific Publishing Co., 1992.

[21] L. J. Guibas, D. E. Knuth, and M. Sharir. Randomized incremental construction of Delaunay and Voronoi diagrams. *Algorithmica*, 7:381–413, 1992.

[22] L. J. Guibas and J. Stolfi. Primitives for the manipulation of general subdivisions and the computation of Voronoi diagrams. *ACM Trans. on Graphics*, 4:74–123, 1985. (Source code at `http://www.dcc.unicamp.br/~stolfi/EXPORT/software/c/Index.html`.)

[23] D. T. Lee and F. P. Preparata. Location of a point in a planar subdivision and its applications. *SIAM J. Comput.*, 6:594–606, 1977.

[24] D. T. Lee and B. J. Schachter. Two algorithms for constructing a Delaunay triangulation. *Int. J. Comput. Inf. Sci.*, 9:219–242, 1980.

[25] E. P. Mücke, I. Saias, and B. Zhu. Fast randomized point location without preprocessing in two- and three-dimensional Delaunay triangulations. In *Proc. 12th ACM Sympos. Comput. Geom.*, pages 274–283, 1996.

[26] K. Mulmuley. *Computational Geometry: An Introduction Through Randomized Algorithms*. Prentice-Hall, Englewood Cliffs, NJ, 1994.

[27] J. O'Rourke. *Computational Geometry in C* (2nd ed.). Cambridge University Press, 1998.

[28] F. P. Preparata and S. J. Hong. Convex hulls of finite sets of points in two and three dimensions. *Commun. ACM*, 20:87–93, 1977.

[29] F. P. Preparata and M. I. Shamos. *Computational Geometry: An Introduction*. Springer-Verlag, New York, 1985.

[30] N. Sarnak and R. E. Tarjan. Planar point location using persistent search trees. *Commun. ACM*, 29:669–679, 1986.

[31] R. Seidel. Constructing higher-dimensional convex hulls at logarithmic cost per face. In *Proc. 18th ACM Sympos. Theory Comput.*, pages 404–413, 1986.

[32] M. I. Shamos and D. Hoey. Closest-point problems. In *Proc. 16th IEEE Sympos. Found. Comput. Sci.*, pages 151–162, 1975.

[33] J. R. Shewchuk. Triangle: engineering a 2D quality mesh generator and Delaunay triangulator. In *Proc. 1st Workshop on Applied Comput. Geom.*, pages 124–133, 1996. (Source code available at `http://www-2.cs.cmu.edu/~quake/triangle.html`.)

[34] J. R. Shewchuk. Adaptive precision floating-point arithmetic and fast robust geometric predicates. *Discrete Comput. Geom.*, 18:305–363, 1997.

[35] J. Snoeyink. Point location. In *Handbook of Discrete and Computational Geometry* (J. E. Goodman and J. O'Rourke, eds.), CRC Press, Boca Raton, FL, pages 559–574, 1997.

[36] P. Su and R. L. S. Drysdale. A comparison of sequential Delaunay triangulation algorithms. *Comput. Geom. Theory Appl.*, 7:361–386, 1997.

Appendix: Complete Code

```
 1  // Timothy Chan    "ch3d.cc"    12/02    3-d lower hull (in C++)
 2
 3  // a simple implementation of the O(n log n) divide-and-conquer algorithm
 4
 5  //   input: coordinates of points
 6  //     n x_0 y_0 z_0 ... x_{n-1} y_{n-1} z_{n-1}
 7
 8  //   output: indices of facets
 9  //     i_1 j_1 k_1 i_2 j_2 k_2 ...
10
11  //   warning: ignores degeneracies and robustness
12  //   space: uses 6n pointers
13
14
15  #include <stream.h>
16
17  struct Point {
18    double x, y, z;
19    Point *prev, *next;
20    void act() {
21      if (prev->next != this) prev->next = next->prev = this;  // insert
22      else { prev->next = next; next->prev = prev; }  // delete
23    }
24  };
25
26  const double INF = 1e99;
27  static Point nil = {INF, INF, INF, 0, 0};
28  Point *NIL = &nil;
29
30  inline double turn(Point *p, Point *q, Point *r) {  // <0 iff cw
31    if (p == NIL || q == NIL || r == NIL) return 1.0;
32    return (q->x-p->x)*(r->y-p->y) - (r->x-p->x)*(q->y-p->y);
33  }
34
35  inline double time(Point *p, Point *q, Point *r) {  // when turn changes
36    if (p == NIL || q == NIL || r == NIL) return INF;
37    return ((q->x-p->x)*(r->z-p->z) - (r->x-p->x)*(q->z-p->z)) / turn(p,q,r);
38  }
39
40  Point *sort(Point P[], int n) {  // mergesort
41
42    Point *a, *b, *c, head;
43
44    if (n == 1) { P[0].next = NIL; return P; }
45    a = sort(P, n/2);
46    b = sort(P+n/2, n-n/2);
47    c = &head;
48    do
49      if (a->x < b->x) { c = c->next = a; a = a->next; }
50      else { c = c->next = b; b = b->next; }
51    while (c != NIL);
52    return head.next;
53  }
54
55  void hull(Point *list, int n, Point **A, Point **B) {  // the algorithm
56
57    Point *u, *v, *mid;
58    double t[6], oldt, newt;
59    int i, j, k, l, minl;
60
```

```
61    if (n == 1) { A[0] = list->prev = list->next = NIL; return; }
62
63    for (u = list, i = 0; i < n/2-1; u = u->next, i++) ;
64    mid = v = u->next;
65    hull(list, n/2, B, A);  // recurse on left and right sides
66    hull(mid, n-n/2, B+n/2*2, A+n/2*2);
67
68    for ( ; ; )  // find initial bridge
69      if (turn(u, v, v->next) < 0) v = v->next;
70      else if (turn(u->prev, u, v) < 0) u = u->prev;
71      else break;
72
73    // merge by tracking bridge uv over time
74    for (i = k = 0, j = n/2*2, oldt = -INF; ; oldt = newt) {
75      t[0] = time(B[i]->prev, B[i], B[i]->next);
76      t[1] = time(B[j]->prev, B[j], B[j]->next);
77      t[2] = time(u, u->next, v);
78      t[3] = time(u->prev, u, v);
79      t[4] = time(u, v->prev, v);
80      t[5] = time(u, v, v->next);
81      for (newt = INF, l = 0; l < 6; l++)
82        if (t[l] > oldt && t[l] < newt) { minl = l; newt = t[l]; }
83      if (newt == INF) break;
84      switch (minl) {
85      case 0:  if (B[i]->x < u->x) A[k++] = B[i];  B[i++]->act();  break;
86      case 1:  if (B[j]->x > v->x) A[k++] = B[j];  B[j++]->act();  break;
87      case 2:  A[k++] = u = u->next;  break;
88      case 3:  A[k++] = u;  u = u->prev;  break;
89      case 4:  A[k++] = v = v->prev;  break;
90      case 5:  A[k++] = v;  v = v->next;  break;
91      }
92    }
93    A[k] = NIL;
94
95    u->next = v;  v->prev = u;  // now go back in time to update pointers
96    for (k--; k >= 0; k--)
97      if (A[k]->x <= u->x || A[k]->x >= v->x) {
98        A[k]->act();
99        if (A[k] == u) u = u->prev; else if (A[k] == v) v = v->next;
100     }
101     else {
102       u->next = A[k]; A[k]->prev = u; v->prev = A[k]; A[k]->next = v;
103       if (A[k]->x < mid->x) u = A[k]; else v = A[k];
104     }
105 }
106
107 main() {
108
109   int n, i;
110   cin >> n;
111
112   Point *P = new Point[n];  // input
113   for (i = 0; i < n; i++) { cin >> P[i].x; cin >> P[i].y; cin >> P[i].z; }
114
115   Point *list = sort(P, n);
116   Point **A = new Point *[2*n], **B = new Point *[2*n];
117   hull(list, n, A, B);
118
119   for (i = 0; A[i] != NIL; A[i++]->act())  // output
120     cout << A[i]->prev-P << " " << A[i]-P << " " << A[i]->next-P << "\n";
121   delete A;  delete B;  delete P;
122 }
```