The Journal of Graphics Tools, 1(1), 1996, pp. 33–47

Improving Accuracy in a Robust Algorithm for Three-Dimensional Voronoi Diagrams *

Philip M. Hubbard
Program of Computer Graphics
Cornell University†

Abstract

This paper describes extensions to a previous algorithm that robustly builds three-dimensional Voronoi diagrams in the presence of inexact numerical computations. The extensions improve the algorithm's accuracy, making its results more nearly represent the proximity properties of an ideal Voronoi diagram. In empirical tests, these extensions have improved accuracy by more than eight orders of magnitude. Complete pseudocode for the algorithm appears in an appendix of this paper.

1 Introduction

Geometric relationships appear throughout computer graphics, and a fundamental relationship is proximity. For many problems involving proximity, efficient solutions are possible using *Voronoi diagrams* (also known as *Dirichlet tessellations*). Given a finite set of points, P, a Voronoi diagram associates with each point, $p \in P$, the region of space closer to p than to any of the other points, $p' \in P$. Preparata and Shamos [14] discuss several general problems solvable with Voronoi diagrams: finding the nearest neighbor of each point in P; finding the point in P closest to an arbitrary query point, q; building the convex hull of P; and building the *Delaunay triangulation* of P, in which all the triangles are as close to equilateral as possible. These capabilities make Voronoi diagrams useful in more specific graphics problems: Joseph [11] uses them for direct volume rendering of irregularly sampled data; and Goldak *et al.* [5] use them to approximate the medial-axis surfaces or "skeletons" of polyhedral models, a construction useful in the approximation of the models' shapes for time-critical collision detection [9].

To be useful in real applications, an algorithm that constructs a Voronoi diagram must meet two requirements. It must be *robust* in that it must produce a topologically consistent diagram for any input P. Of particular importance is avoiding inconsistencies that cause the algorithm to terminate prematurely. The algorithm must also be *accurate* in that the diagram it produces must partition space into regions as correctly as possible. Specifically, the region associated with each $p \in P$ should include minimal space that is actually closer to some other $p' \in P$.

†Mailing address: 580 Frank H. T. Rhodes Hall, Cornell University, Ithaca, NY 14853-3801. Email address: pmh@graphics.cornell.edu. URL: http://www.graphics.cornell.edu/~pmh.

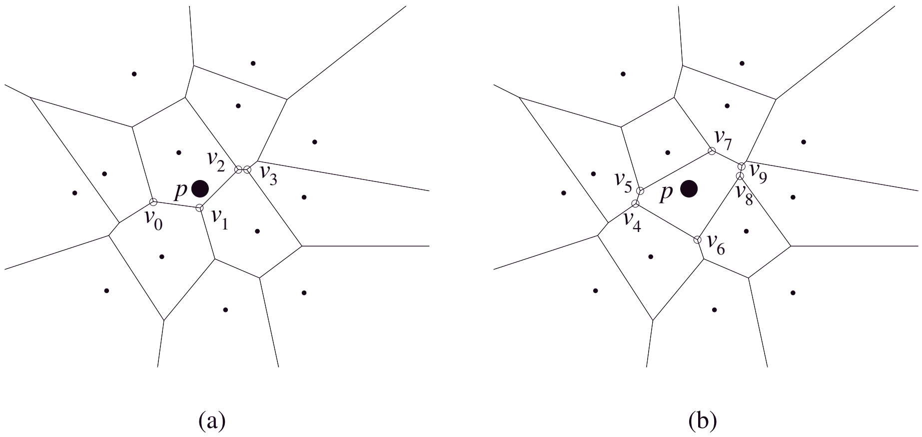

Figure 1: Adding the darkened point, p, causes (a) the deletion of vertices v_0 through v_3, followed by (b) the creation of new vertices v_4 through v_9. Note that in this two-dimensional example, cells are convex polygons instead of polyhedra.

Robustness and accuracy are challenging goals, however, not just for Voronoi diagrams but for many geometric problems. Hoffmann [7] explores the factors that make robustness and accuracy elusive. A basic difficulty is that in theory, geometric problems deal with objects from a continuous domain, but in practice, geometric algorithms use discrete computation. The literature describes techniques, such as the work of Edelsbrunner and Mücke [3], that address this difficulty in a general manner; applying these techniques to specific problems introduces complications, however, as discussed by Schorn [15]. A more direct approach is to develop solutions tailored to specific problems, such as the work by Karasick, Lieber and Nackman [12] and by Sugihara and Iri [17] on robust algorithms for two-dimensional Voronoi diagrams.

Building *three-dimensional* Voronoi diagrams efficiently is a popular topic in the literature, as exemplified by the recent work of Fang and Piegl [4], but to our knowledge, the only algorithms that guarantee robustness are the work of Dey *et al.* [2] and of Inagaki *et al.* [10]. We prefer the latter algorithm because its implementation seems more straightforward. This algorithm has the advantage that it uses only combinatorial computations, avoiding inexact numerical computations that compromise robustness. Its disadvantage is that it does not address issues of accuracy, producing diagrams that do not necessarily embody the proximity properties of true Voronoi diagrams. Empirical tests indicate that the inaccuracy can be significant on input data from a practical application. As a remedy, we describe extensions to the algorithm, heuristics that increase accuracy while maintaining robustness. In empirical tests, this approach provides significant improvements in accuracy over the Inagaki *et al.* algorithm; in one case, for example, the new approach improves accuracy by more than eight orders of magnitude.

2 A Robust Algorithm

The algorithm of Inagaki *et al.* extends previous algorithms that build Voronoi diagrams incrementally, an example being the work of Bowyer [1]. Each increment adds a point, $p \in P$, to the diagram for the points already added. Adding p creates one new *cell* in the diagram, the region of space closer to p than to the already-added points. This space was formerly divided amongst cells associated with already-added points, so adding the new cell must cause existing cells to shrink. Each cell is a convex polyhedron, and shrinking a cell involves deleting some of its *vertices* and replacing them with new ones. Figure 1 gives a two-dimensional example of this process. Each new vertex becomes a shared corner of the cells associated with p and three already-added points. These four points are the vertex's *forming points*. By definition, this corner must be equidistant from the four points, so the position of a new vertex is the circumcenter of its forming points (i.e., the center of the sphere whose surface touches the forming points).

Ideally, the algorithm should delete any vertex that is closer to the new point than to its forming points. A vertex that meets this criterion is called *theoretically deletable*. Such a vertex should be deleted because it is a corner of its forming points' cells, and those cells must shrink so they do not contain space closer to the new point. In practice, however, the algorithm cannot measure closeness exactly, due to the nature of numeric computation. Even small errors can cause the algorithm to delete too many or too few vertices, producing topological inconsistencies from which the algorithm cannot recover. Since specific examples of how these inconsistencies can occur are lengthy and tedious, we do not repeat examples that appear elsewhere [8].

To eliminate these threats to robustness, Inagaki *et al.* modify the basic incremental algorithm so that its deletion of vertices[1] explicitly avoids topological inconsistencies. The modifications involve properties of the Voronoi diagram interpreted as a three-dimensional graph. This interpretation is not surprising: the vertices and edges of the graph are the corners and edges of the polyhedral cells[2]. Inagaki *et al.* state that the deletable vertices must satisfy five constraints in terms of the topology of this graph:

1. the set of deletable vertices is nonempty;
2. the deletable vertices form a connected subgraph;
3. at least one vertex from each cell is *not* deletable;
4. the deletable vertices on each cell form a connected subgraph;
5. the undeletable vertices on each cell form a connected subgraph.

Figure 2 shows an example that satisfies these constraints. Checking these constraints requires no numeric computation, and deleting only vertices that satisfy these constraints preserves the topological properties of a Voronoi diagram. A vertex whose addition to the current set of deletable vertices does not violate the constraints is called *legally deletable*.

Inagaki *et al.* suggest a greedy search to find the legally deletable vertices that are also theoretically deletable. The search starts at the vertex closest to the new point being added; any single vertex, like this one, meets the constraints and is thus deletable. From this vertex the search branches out to neighboring vertices, adding to the deletable set any theoretically deletable neighbors that do not violate the constraints. The search continues spreading through the vertices until no neighboring vertex is legally deletable.

The disadvantage of this approach is its loss of accuracy. To maximize accuracy, the greedy search should choose vertices whose deletion causes cells to shrink so that the cell for each already-added point contains minimal space closer to the new point or any other point. The greedy search does not consider this criterion, however, when choosing between vertices of equal topological validity for deletion. The greedy search may choose a vertex associated with cells that need to shrink only small amounts, and by making this vertex deletable, the search may prevent itself from later choosing another vertex whose cells require more drastic shrinking. Specific examples of such problems are difficult to isolate in practice, but they do seem to occur, because the Inagaki *et al.* algorithm is significantly less accurate than Bowyer's algorithm on point sets that do not make the latter fail.

[1] Inagaki *et al.* use the term "point" for what Bowyer calls a "vertex" (i.e., a corner of a Voronoi cell), and the term "generator" for what Bowyer calls a "point" (i.e., a member of the original data set, P). We use Bowyer's terminology.

[2] Note that if two cells touch at a vertex or edge, the vertex or edge appears only once in the graph.

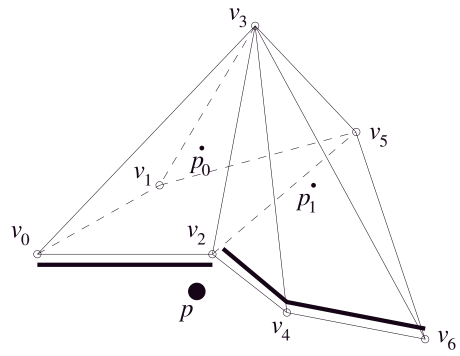

Figure 2: The cell for point p_0 consists of vertices v_0, v_1, v_2, v_3 and v_5, and the cell for p_1 consists of vertices v_2, v_3, v_4, v_5 and v_6. Adding the darkened point, p, forces the deletion of vertices v_0, v_2, v_4 and v_6. These vertices form a connected subgraph, as do the subsets of these vertices on each cell, as indicated by the heavy lines. The undeleted vertices on each cell also form connected subgraphs.

3 Improving Accuracy

To produce accurate results, a robust algorithm must do more than just meet the Inagaki *et al.* constraints. It must replace the greedy search with a more elaborate search, one that identifies vertices whose deletion will compromise the geometric properties of the Voronoi diagram as little as possible. One approach is to define a "value" for the deletion of a vertex, where the value increases with the likelihood that deleting the vertex will produce an accurate diagram. The search for deletable vertices thus becomes a constrained maximization problem: finding the set of vertices with the maximum value that also meets the Inagaki *et al.* constraints.

A suitable definition of a vertex's value is its distance to its forming points minus its distance to the new point. This value is positive for any theoretically deletable vertex. The value increases with the extent to which the cells of the vertex's forming points intrude on the region which should be associated with the new point. A high value means that the forming points' cells contain a substantial amount of space that is actually closer to the new point. Thus, whenever the algorithm does not delete a vertex with high value it decreases the accuracy of the Voronoi diagram. The algorithm's computation of a vertex's value is approximate in that it involves inexact numerical computations. Nevertheless, it does not affect the Inagaki *et al.* constraints that ensure robustness, and the empirical evidence from Section 4 suggests that the use of even approximate values provides significant benefits.

Finding the set of legally deletable vertices with the maximum possible value is a difficult problem. A simpler problem is finding a set with maxi*mal* value, meaning a set to which it is impossible to add another vertex that increases the overall value. A set with maximal value allows more inaccuracy than a set with maximum value, but it is preferable to the set found by the Inagaki *et al.* approach, which does not consider value at all.

The simplest approach for finding a set of legally deletable vertices with maximal value is to use a greedy algorithm. This algorithm is greedy in the sense that when choosing vertices to add to the deletable set, it always takes the highest-valued vertex that it can find. We call this strategy a *highest-first* approach to distinguish it from the greedy approach of Inagaki *et al.*.

The highest-first algorithm's primary data structure is a priority queue of vertices, which releases vertices with high values first. The algorithm initializes this queue to the vertex with maximum value; our implementation finds this vertex by a simple linear search. The main loop of the algorithm starts by taking the first vertex from the priority queue. If this vertex is not legally deletable, the algorithm adds it to a set of "illegal" vertices and takes the next vertex from the queue until it finds a legally deletable vertex. (Determining if any single vertex is legally deletable is the same as in the Inagaki *et al.* algorithm.) The algorithm adds this vertex to the deletable set. It then adds to the queue the vertex's positive-valued neighbors. The last step of the main loop adds back onto the queue the set of illegal vertices found at the beginning of the loop; the algorithm should reconsider these vertices because they have high value, and it is possible some of them will no longer violate the constraints once other vertices have been added to the deletable set. The algorithm then repeats its main loop. The loop terminates when the algorithm finds no legally deletable vertex on the queue. Note that this approach is reminiscent of Kruskal's algorithm for finding the minimum spanning tree of a graph [13, 16].

Another accuracy issue concerns the position assigned to a new vertex. Using the circumcenter of the four forming points can fail in practice, because imprecise choices of the deletable vertices can lead to forming points that are coplanar. Inagaki *et al.* thus suggest switching to a purely topological criterion—that each new vertex can lie anywhere on the edge between a deletable vertex, v_d, and an undeletable vertex, v_u—but this approach allows vertex positions that compromise accuracy. A better approach is to use the circumcenter when the forming points are not coplanar, resorting to the topological criterion only when coplanarity occurs. In the latter case, the new vertex should have the same position as v_u. As justification, note that the "true" circumcenter of four coplanar points is infinitely far from the points. To approximate going infinitely far while staying on the edge between v_d and v_u, the algorithm must stop at v_d or v_u; v_u is preferable because it is not closer to the new point than to its forming points.

Pseudocode for the complete algorithm, including the relevant parts of the Bowyer and Inagaki *et al.* algorithms, appears in the Appendix at the end of this paper.

4 Results

In a true Voronoi diagram, each cell should be a convex polyhedron containing exactly one $p \in P$, and every location in that cell should be closer to p than to any other $p' \in P$. To empirically test this property, we use a postprocess that generates sample locations, q, on the line segment from each p to each corner of its cell; our current implementation uses three evenly-spaced samples per segment. The postprocess computes d, the distance from q to p, and d', the minimum distance from q to any other $p' \in P$. If $d' < d$, then the diagram is inaccurate. A measure of the inaccuracy at q is $d - d'$, for which a higher value means more inaccuracy. The postprocess sums all positive values of $d - d'$ and divides by the total number of samples to obtain the mean inaccuracy for the Voronoi diagram.

We tested the Inagaki *et al.* algorithm and the new algorithm on two sets of data. The data comes from a real application mentioned in Section 1: using the Voronoi diagram for points sprinkled on the surface of a model to approximate the model's medial-axis surface, as a step in approximating the model's shape for time-critical collision detection [9]. Figures 3 and 4 show the two data sets. Bowyer's algorithm, which does not consider robustness issues, fails on both data sets, so these examples emphasize that robustness is important in practical applications and not just for pathological test cases.

Figure 5 shows the mean inaccuracy for the two robust algorithms on these data sets. Recall that the inaccuracy is a spatial distance; to make the scale of this distance meaningful, the inaccuracies are expressed as fractions of the width of the data set's bounding box. The inaccuracy of the Inagaki *et al.* algorithm is significant, being between five and ten percent of the scale of the data. The inaccuracy of the new algorithm is substantially better, being more than four orders of magnitude lower on the first data set, and more than eight orders of magnitude lower on the second.

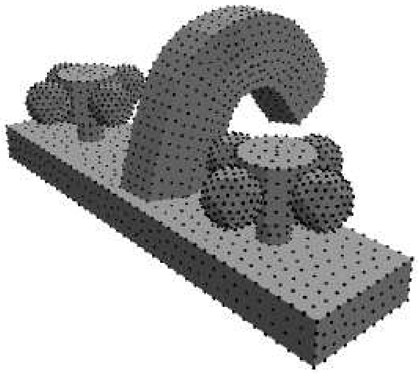

Figure 3: 3325 points on a bathroom faucet.

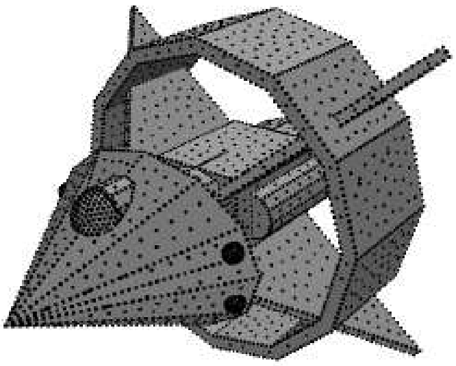

Figure 4: 6792 points on a spaceship.

Further evidence of the new algorithm's improved accuracy comes from the number of times the algorithm has to approximate the circumcenter of coplanar forming points: it makes this approximation 52 times for the first test and 5 times for the second, whereas the Inagaki *et al.* algorithm makes it 4066 and 6452 times, respectively. The diagrams produced by the new algorithm are not only more accurate but also more compact: in the first test, the number of vertices it produces is 59 percent of the number produced by the Inagaki *et al.* algorithm, and in the second test it is 70 percent.

For the question of performance, the two tests are not conclusive. In the first test, the new algorithm is faster, taking 154 seconds on an Hewlett Packard 9000 Model 735 compared to the 240 seconds required by the Inagaki *et al.* algorithm. The new algorithm is slower in the second

	mean inaccuracy	
data set	Inagaki *et al.* algorithm	new algorithm
faucet	0.07	6.02×10^{-6}
spaceship	0.09	8.31×10^{-10}

Figure 5: Mean inaccuracy, as a fraction of the width of the data set's bounding box.

test, however, taking 963 seconds compared to the Inagaki *et al.* algorithm's 859 seconds. The performance of our implementations of both algorithms would likely improve with optimization. One operation deserving optimization is the search for the first deletable vertex corresponding to a point being added to the diagram; our implementations currently use the simplest approach for this operation, a linear search.

5 Acknowledgements

This research was conducted as part of the author's doctoral studies at Brown University. The guidance of John "Spike" Hughes was essential to the completion of this work. James Stewart and Peter Shirley also made valuable suggestions on the presentation of this material. At Brown, this work was supported in part by: the NSF/ARPA Science and Technology Center for Computer Graphics and Scientific Visualization; Sun Microsystems; Autodesk; Taco, Inc.; ONR grant N00014-91-J-4052 ARPA order 8225; NCR; Hewlett Packard; IBM; Digital Equipment Corporation; Apple; and Microsoft.

References

[1] Adrian Bowyer. Computing Dirichlet tessellations. *The Computer Journal*, 24(2):162–166, 1981.

[2] Tamal K. Dey, Kokichi Sugihara, and Chandrajit L. Bajaj. Delaunay triangulations in three dimensions with finite precision arithmetic. *Computer Aided Geometric Design*, 9:457–470, 1992.

[3] Herbert Edelsbrunner and Ernst Peter Mücke. Simulation of simplicity: A technique to cope with degenerate cases in geometric algorithms. *ACM Transactions on Graphics*, 9(1):66–104, January 1990.

[4] Tsung-Pao Fang and Les A. Piegl. Delaunay triangulation in three dimensions. *IEEE Computer Graphics and Applications*, 15(5):62–69, September 1995.

[5] John A. Goldak, Xinhua Yu, Alan Knight, and Lingxian Dong. Constructing discrete medial axis of 3-D objects. *International Journal of Computational Geometry and Applications*, 1(3):327–339, 1991.

[6] Ronald Goldman. Intersection of three planes. In Andrew S. Glassner, editor, *Graphics Gems*, page 305. Academic Press, Boston, Massachusetts, 1990.

[7] Christoph M. Hoffmann. The problems of accuracy and robustness in geometric computation. *IEEE Computer*, 22(3):31–41, March 1989.

[8] Philip M. Hubbard. *Collision Detection for Interactive Graphics Applications*. PhD thesis, Department of Computer Science, Brown University, 1994. Available at ftp://ftp.cs.brown.edu/pub/techreports/95/cs95-08.ps.Z.

[9] Philip M. Hubbard. Approximating polyhedra with spheres for time-critical collision detection. *ACM Transactions on Graphics*, 15(3), July 1996.

[10] Hiroshi Inagaki, Kokichi Sugihara, and Noboru Sugie. Numerically robust incremental algorithm for constructing three-dimensional Voronoi diagrams. In *Proceedings of the Fourth Canadian Conference on Computational Geometry*, pages 334–339, 1992.

[11] Jonathan Joseph. Direct volume rendering of irregularly sampled data using Voronoi decomposition. Master's thesis, Program of Computer Graphics, Cornell University, 1996.

[12] Michael Karasick, Derek Lieber, and Lee R. Nackman. Efficient Delaunay triangulation using rational arithmetic. *ACM Transactions on Graphics*, 10(1):71–91, January 1991.

[13] Joseph B. Kruskal, Jr. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical Society*, 7(1):48–50, February 1956.

[14] Franco P. Preparata and Michael I. Shamos. *Computational Geometry: An Introduction*. Springer-Verlag, New York, New York, 1985.

[15] Peter Schorn. Degeneracy in geometric computation and the perturbation approach. *The Computer Journal*, 37(1):35–42, 1994.

[16] Robert Sedgewick. *Algorithms*. Addison-Wesley, Reading, Massachusetts, 2nd edition, 1988.

[17] Kokichi Sugihara and Masao Iri. Constructing the Voronoi diagram for "one million" generators in single-precision arithmetic. *Proceedings of the IEEE*, 80(9):1471–1484, September 1992.

Appendix: Pseudocode

Figure 6 presents pseudocode for the main routine that builds a Voronoi diagram for a set of three-dimensional points, P. The main routine follows the general model suggested by Bowyer [1]. Figures 7 through 11 describe routines called by the main routine. The routine in Figure 10 checks the Inagaki *et al.* constraints [10]; the routines in Figures 8, 9 and 11 implement the extensions that make the robust algorithm more accurate.

Certain data structures are implicit in the pseudocode. Each point, p, is marked with references to the vertices for which it is a forming point (as defined in Section 2); these vertices are sometimes referred to as the vertices "on p's cell." Each vertex, v, is marked with references to its four forming points and the four vertices that are its neighbors[3] in the Voronoi diagram; note that the structure and useful properties of a Voronoi diagram can thus be recovered from the set, V, of all its vertices. Vertices can also be marked as "deleted" and "visited" for temporary bookkeeping purposes. Variable names follow the convention that sets and other collections begin with capital letters.

This pseudocode builds the Voronoi diagram incrementally, adding each point, $p \in P$, to the diagram for the previous points. Bowyer observes that this approach involves fewer special cases if each point lies within the convex hull of the previous points. Thus, our pseudocode adopts Bowyer's strategy of starting with the Voronoi diagram for a tetrahedron of "dummy" points that bounds all points in P. The $\mathsf{initialize}(P)$ routine in Figure 7 adds these points and builds their Voronoi diagram. The four cells of this diagram are unbounded, meeting at one vertex in the center of the tetrahedron and extending infinitely far away from this vertex. Representing the unbounded sides of these cells by four vertices "at infinity" simplifies the rest of the pseudocode; this approach requires only that the pseudocode take care to never delete a vertex "at infinity." Note that the addition of the dummy points does subtly change some parts of the final Voronoi diagram. The cells associated with the dummy points bound the cells associated with points on the convex hull of P, cells that would otherwise be unbounded. For applications in which these artificial boundaries matter, they can be detected as they occur at vertices with a dummy point as a forming point.

[3]Situations in which a Voronoi diagram might seem to have a vertex with more than four neighbors are actually situations in which multiple four-neighbor vertices coincide with each other, being connected by zero-length edges.

```
voronoi(P)
{
   /* V will contain the Voronoi diagram's vertices. */
   V ← empty set;
   initialize(P);
   for (p ← each point in P) {
      V_d ← deletableVertices(p);
      /* V_n will contain the new vertices for p. */
      V_n ← empty set;
      for (v_d ← each vertex in V_d) {
         for (v_u ← each neighbor of v_d not in V_d) {
            P_form ← forming points shared by v_d and v_u;
            if (P_form contains 3 points) {
               c ← circumcenter(P_form, p, v_u);
               v_n ← new vertex at position c;
               unmark v_d and v_u as neighbors;
               mark v_n and v_u as neighbors;
               mark P_form and p as forming points for v_n;
               put v_n in V_n;
               put v_n in V;
            }
         }
         unmark forming points of v_d;
         remove v_d from V;
      }
      for (v'_n, v''_n ← each pair from V_n)
         if (v'_n and v''_n share 3 forming points)
            mark v'_n and v''_n as neighbors;
   }
}
```

Figure 6: This routine builds a Voronoi diagram for the set of points P. The result is the set of vertices V for the diagram.

```
initialize(P)
{
   P_tetr ← tetrahedron of dummy points that bound P;
   v_n ← new vertex at tetrahedron center;
   V_∞ ← four new vertices "at infinity";
   mark P_tetr as forming points for v_n;
   mark V_∞ and v_n as neighbors;
}
```

Figure 7: This routine initializes the Voronoi diagram for the set of points P.

```
deletableVertices(p)
{
   p0 ← already-added point closest to p;
   v0 ← vertex on p0's cell that is most closer to p than to p0;
   /* Vd will contain the deletable vertices. */
   Vd ← empty set;
   /* Q releases a vertex, v, with higher deleteValue(v, p) earlier. */
   Q ← priority queue initialized with v0;
   do {
       /* Vill will contain the high-valued vertices that are not legally deletable. */
       Vill ← empty set;
       /* When Q is empty, v will evaluate to FALSE. */
       while ((v ← head removed from Q) and (not legallyDeletable(v)))
           put v in Vill;
       if (v) {
         put v in Vd;
         mark v as deleted;
         for (vneigh ← each neighbor of v)
             if ((deleteValue(vneigh, p) > 0) and (vneigh is not "at infinity") and
                 (vneigh is not already on Q))
               put vneigh on Q;
         put each vertex from Vill onto Q;
       }
   } while (v);
   return Vd;
}
```

Figure 8: This routine returns the set V_d of vertices to delete when adding the point p.

```
deleteValue(v, p)
{
   df ← distance from v to its forming points;
   dp ← distance from v to p;
   return (df − dp);
}
```

Figure 9: This routine returns the value of deleting the vertex v when adding the point p.

```
legallyDeletable(v)
{
   /* Inagaki et al. constraint 1 is met because deletableVertices(p) makes V_d contain
      at least one vertex. */
   /* Inagaki et al. constraint 2 is met because deletableVertices(p) puts only
      neighboring vertices in V_d. */
   for (p ← each forming point of v) {
      /* Inagaki et al. constraint 3. */
      if (every vertex on p's cell is marked deleted)
         return FALSE;
      /* Inagaki et al. constraint 4. Checking this constraint is simplified for the same
         reason constraint 2 is satisfied. */
      if ((no neighbor of v on p's cell is marked deleted) and
          (v is not the only vertex on p's cell marked deleted))
         return FALSE;
      /* Inagaki et al. constraint 5. */
      unmark as visited all vertices on p's cell;
      v_u ← any vertex on p's cell not marked deleted;
      for (v_reach ← each vertex reachable from v_u by a path of
             neighboring vertices on p's cell not marked deleted)
           mark v_reach as visited;
      if (any vertex on p's cell is not marked deleted and not marked visited)
         return FALSE;
   }
   return TRUE;
}
```

Figure 10: This routine returns TRUE if the vertex v is legally deletable, according to the Inagaki *et al.* constraints.

```
circumcenter(P_form, p, v_u)
{
   for (i from 1 to 3) {
      m_i ← midpoint of segment from p to ith point in P_form;
      n_i ← vector from p to ith point in P_form, normalized;
   }
   d ← determinant(n_1, n_2, n_3);
   /* ε is a small number that acts like 0 for the numerical precision of the machine. */
   if (|d| > ε)
      /* Use Goldman's algorithm [6] to find the intersection of 3 planes with normals n_i
         at positions m_i. */
      return [(m_1 · n_1)(n_2 × n_3) + (m_2 · n_2)(n_3 × n_1) + (m_3 · n_3)(n_1 × n_2)]/d;
   else
      return position of v_u;
}
```

Figure 11: This routine returns the circumcenter of the three points in the set P_{form} and the new point p. If these points are coplanar, it approximates the circumcenter based on the undeleted vertex v_u.