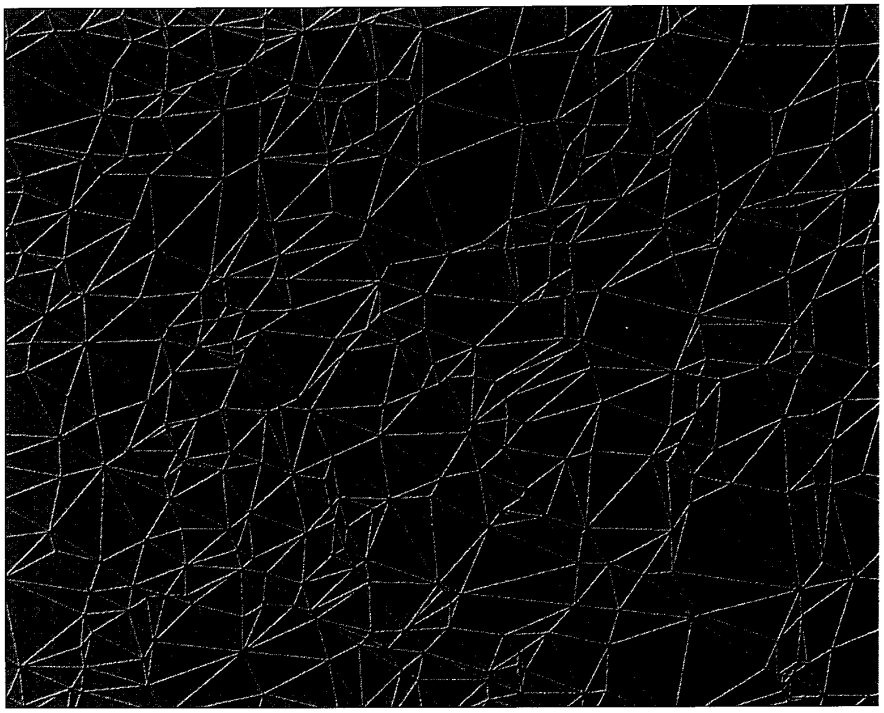
Delaunay Triangulation Using a Uniform Grid

Tsung-Pao Fang and Les A. Piegl
University of South Florida



Our direct algorithm for computing the Delaunay triangulation of 2D points permits dynamic update of the internal grid data structure and simultaneous computation of the convex hull.

Triangulations of scattered points on the plane have been the subject of significant research in the past few decades. The origins go back to Voronoi[1] and Delaunay.[2] Textbooks[3,4] and papers[5-11] have extensively covered their properties and the algorithms for their construction. Most of this work deals with the algorithms' theoretical aspects and gives upper bounds on their complexity. In this article we present a new algorithm and its implementation. Instead of providing a theoretical analysis, we present implementation details, tests, and examples. Our experience shows that implementation is not a trivial exercise, which agrees with Forrest's opinion[12]: If researchers have not implemented an algorithm, their study of it is incomplete.

Our algorithm has a number of new features:

- It is a direct method without the overhead of the popular divide-and-conquer algorithms (that is, stack management, splitting the data set, and merging the partial results). Also, it does not require data sorting.
- It uses a uniform grid structure to form triangles and find boundary edges and nearest points.
- It creates the triangles circularly through shelling. This ensures that the triangulation is complete and correct, and helps the algorithm create the output data structure as it computes the triangles.
- It uses an edge list to govern the triangulation and dynamically decrease the complexity of the internal grid data structure.
- It is numerically stable and handles coincident and collinear points automatically.
- It produces the convex hull of the data set at no extra cost.
- For all the randomly generated data we tested, the algorithm exhibited linear time complexity. This is a significant property because the problem of triangulation in two dimensions is inherently $\Theta(n^2)$.

In the next section we discuss how to preprocess the data. Then we show how to set up an internal data structure based on a uniform grid, give details about the triangulation process, and show how to use the algorithm to compute the convex hull. We also present solutions to degeneracies, tests, and examples, and propose a method for output generation.

Preprocessing the data

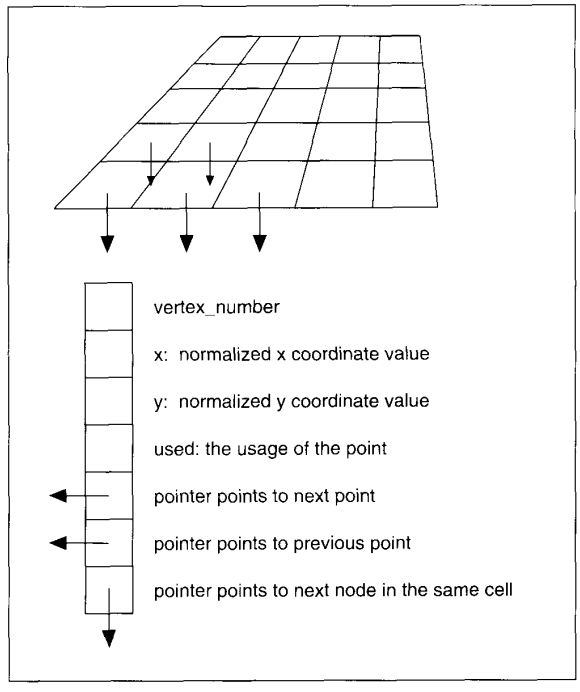
Suppose we have a set of 2D points

$$(x_i, y_i),\ i = 0, \ldots, n$$

We want to put these points into a data structure that permits fast searching for points making up triangles. More precisely, given an edge $<a, b>$, we want the algorithm to find a point c such that the triangle $<a, b, c>$ is a Delaunay triangle. (A Delaunay trian-

0272-17-16/93/0500-0036$03.00 © 1993 IEEE

Figure 1. Uniform grid data structure.



gle has the property that its circumscribing circle does not contain any other point.) For its data structure, our algorithm uses a uniform grid structure laid over the 2D points.

To compute the uniform grid, first step get the min-max box of the data set. Then offset the box outward by the point coincidence tolerance *TOL*. That is, compute

$$x\min = x\min - TOL$$
$$x\max = x\max + TOL$$
$$y\min = y\min - TOL$$
$$y\max = y\max + TOL$$

This step is necessary to resolve conflicts when points lie on grid lines. Next, compute the grid size:

$$size = \sqrt{\frac{(x\max - x\min)(y\max - y\min)}{n}}$$

(We recently found that $size \times 4/3$ results in the fastest algorithm.) This choice of the grid size results in the following number of grid cells in the x and y directions (the x and y resolutions of the 2D domain):

$$x\_res = \text{int}\left(\frac{x\max - x\min}{size}\right) + 1$$
$$y\_res = \text{int}\left(\frac{y\max - y\min}{size}\right) + 1$$

where int() is the cast operator that converts a double-precision number into an integer.

After the grid structure has been computed, to put each data point in one and only one grid cell, do the following for each point (x_i, y_i), $i = 0, \ldots, n$:

1. Compute

$$grid\_x = \frac{x_i - x\min}{size}$$
$$grid\_y = \frac{y_i - y\min}{size}$$
$$i\_cell = \text{int}(grid\_x)$$
$$i\_cell = \text{int}(grid\_y).$$

Here *grid_x* and *grid_y* are coordinates normalized with respect to the origin (*x*min, *x*max).

2. If the cell at the index (*i_cell*, *j_cell*) is empty, then put (x_i, y_i) into it.
3. If the cell already has points in it, then check for point coincidence. If the current point coincides with any point already in the cell, then ignore this point. Otherwise, put it in the cell.

This process associates each data point with one and only one cell—even for points that lie along the lines or at the corners of the grid structure. Since *x*min and *y*min were offset by *TOL*, if points lie on grid lines, then the algorithm chooses the grid to the right of the vertical line and above the horizontal. If points lie along the right or along the top side of the min-max box, then since *xmax* and *ymax* were also offset, the algorithm considers the cell immediately left of or below the grid line.

Data structure

The data structure used in our algorithm is a 2D matrix of pointers, each pointing to a linked list of points that fall into the same cell (see Figure 1). The algorithm associates each data point with the following structure:

```
struct cellnode
{
   int vertex_number;
   double x, y;
   int used;
   struct cellnode *previous_point;
   struct cellnode *next_point;
   struct cellnode *next_node;
}
```

Here *vertex_number* is a number between 0 and *n*, *x* and *y* are the normalized coordinates of the point, *used* is a flag that shows whether a point was used to form a triangle, *previous_point* and *next_point* are pointers incorporating a certain ordering into the triangulation process, and *next_node* simply points to the next node associated with the same cell.

Triangulation

The triangulation process consists of three major steps: find-

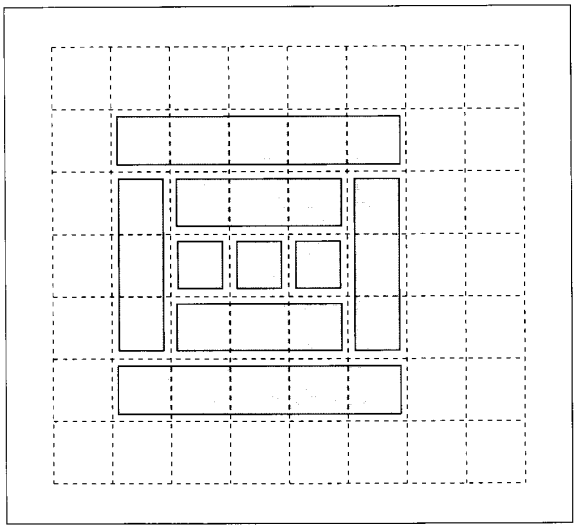


Figure 2. Procedure to find the first point.

ing a start point and the first edge, forming triangles, and putting triangles together.

Finding a start point and the first edge

An interesting question in triangulation is, Where should the triangulation start? Specifically, which are the first point and edge the algorithm should consider? Our algorithm could start at any point. However, for efficiency it starts at a point more or less in the middle of the data set. The algorithm is as follows (see Figure 2):

1. Find the middle cell at the cell index

   $(m, n) = (x\_res/2, y\_res/2)$

2. If the cell at (m, n) is not empty, then pick any point (the top element of the linked list).
3. If the cell is empty, then search the neighboring cells.

Step 3 can be done a variety of ways. The current implementation first searches the top three cells, then the bottom three, followed by the left and the right cells. It stops as soon as it finds a point. If it finds no point, it continues by searching the top five cells, the bottom five, the left three, and the right three, and so on.

This procedure is rather simple. Nevertheless, it works because a start point somewhat off the data set's physical center gives the same performance as the one closest to the center. A major difference arises if the algorithm chooses a point close to a boundary of the min-max box rather than one close to the center (you will understand this point by seeing how to put triangles together).

After it picks the start point P_1, the algorithm computes the first edge by using the following procedure to find the nearest point (see Figure 3):

1. Set the minimum distance d_{min} to a large number, for example, the diagonal of the min-max box.
2. If the cell containing the start point has more than one point, then find the one closest to P_1 and compute the distance d between the two points. If d is less than d_{min}, then set d_{min} to d.
3. Search rows and columns around the cell just as in the previous procedure (that is, search the top three, the bottom three, and the left and right cells, and so on).
4. For each row and column do the following:
   - Drop a perpendicular from P_1 to the side closest to P_1.
   - If the length of the perpendicular is less than d_{min}, then search the row or column. If not, mark the direction (top row, bottom row, left column, or right column) as invalid.
   - If points are found, compute the distance from P_1 and update d_{min} (if the distance is less than the current minimum).
5. Stop the search when all directions are marked invalid; that is, all points in unsearched rows and columns are farther from P_1 than the point used to compute d_{min}.

The found point P_2 is the closest point to P_1, and the edge $<P_1, P_2>$ is used to start the triangulation.

Forming triangles

Assume the algorithm has found the edge $<P_1, P_2>$ (see Figure 4). To find a third point P_i so the triangle $<P_1, P_i, P_2>$ satisfies the Delaunay criterion, the algorithm does the following:

1. Search on the right-hand side of P_1P_2.
2. Find the cells with indexes (i_1, j_1) and (i_2, j_2) that are either the cells of the endpoints or their immediate neighbors. These cells can contain points on the right-hand side of P_1P_2. Compute the index i_1 by intersecting the bottom line of the cell of P_1 with P_1P_2 and finding the column index of the cell that the intersection point falls in (see Figure 4). Compute the index j_2 similarly using the grid line bounding the cell of P_2 from the right.
3. Form a triangular area covered by cells. The vertices of the

Figure 3. Procedure to find the first edge.

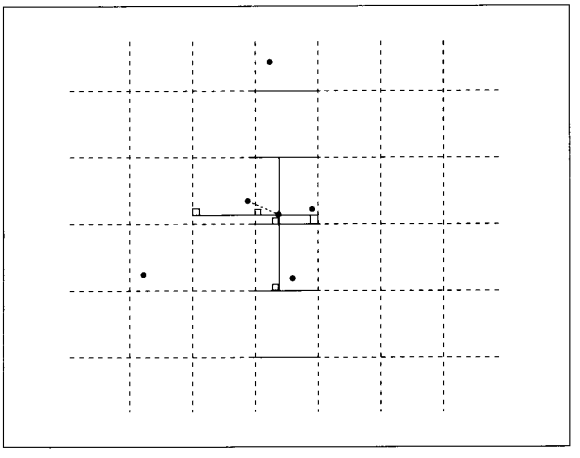


Figure 4. Procedure to form a triangle, given an edge.

cell triangle are (i_1, j_1), (i_2, j_1), and (i_2, j_2), and the boundaries are cells arranged horizontally, vertically, and diagonally, as Figure 4 shows. In the diagonal direction the algorithm chooses all cells that intersect the line P_1P_2. (The line-cell intersection is very efficient. Because of the uniform grid applied, it can be done by one addition. That is, for each line the algorithm computes the slope of the line and the first intersection. Then each grid selection requires only one addition to move to the next intersection point.)

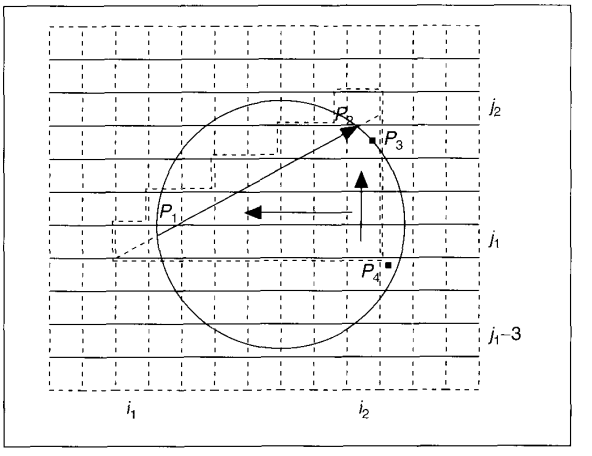
4. After the triangular area is formed, check each cell inside the area. The search can be done in a variety of ways. For example, the algorithm can search each column starting at (i_2, j_1) and move to the left until it reaches (i_1, j_1) (see Figure 4). Or it can search (i_2, j_1), then search diagonally by visiting $(i_2, j_1 + 1)$ and $(i_2 - 1, j_1)$, then $(i_2, j_1 + 2)$, $(i_2 - 1, j_1 + 1)$, and $(i_2 - 2, j_1)$, and so on. Thus the algorithm first searches the cell most likely to have the point that yields the triangle with the largest angle at the found point and as close to equiangular as possible. We kept the searching strategy simple, because in practical situations the average number of cells to be visited is less than 10.
5. If points are found, then choose the one that gives the largest angle (that is, the smallest cosine), get the circle defined by P_1, P_2 and the point with the minimum cosine, and set the min-max box of the circle. (The sides of the box are bounded by grid lines.) If no points are found, search rows from $(i_1 - k, j_1 - k)$ to $(i_2 + k, j_1 - k)$, and columns from $(i_2 + k, j_1 - k)$ to $(i_2 + k, j_2 + k)$, $k = 1, 2, \ldots$, until points are found.
6. After the first min-max box is set, continue searching along rows and columns inside the min-max box and do the same thing as above—that is, select the point with the smallest cosine, draw the circle, and update the min-max box (see Figure 5).
7. Stop the search when there are no unvisited rows and columns inside the current min-max box.

The dynamic update of the circle bounding box is an essential part of the algorithm. Points close to the line P_1P_2 can result in huge boxes that might lead to searching for points on the entire data set. Our goal is to limit the search to the immediate vicinity of the edge P_1P_2 so that points are found within a few steps.

If the edge P_1P_2 happens to be horizontal or vertical, the algorithm uses simple boxes instead of triangles.

Putting triangles together

The most intriguing question is how to put triangles together to ensure that the triangulation is complete and correct. A good way to ensure completeness is to start somewhere, find the first triangle, and then add triangles successively so that the added triangles are simply connected; that is, there are no holes and bridges with the already chosen triangles. Our triangulation strategy adds triangles in a circular fashion. We explain all the steps using the practical example illustrated in Figure 6. The dashed lines show the uniform grid.



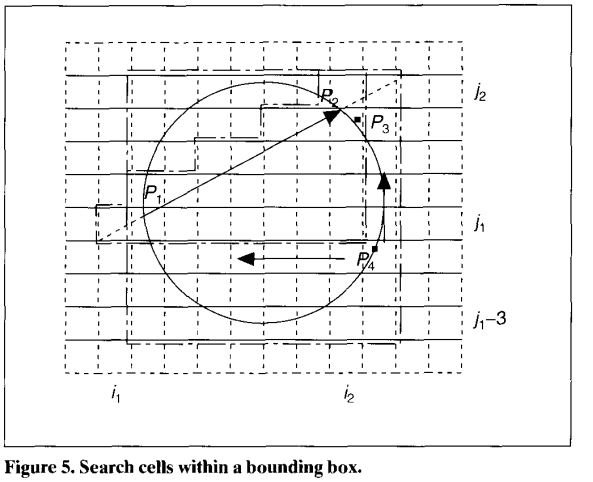


Figure 5. Search cells within a bounding box.

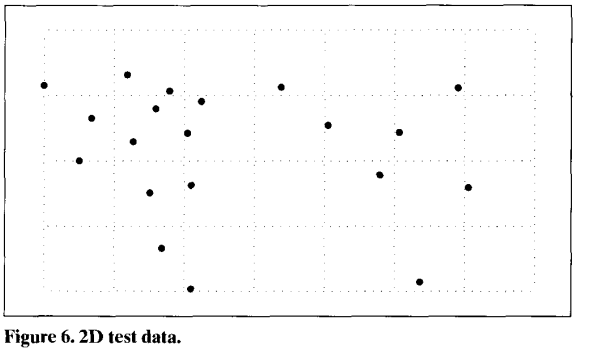


Figure 6. 2D test data.

Figure 7 shows the first step: finding the first edge. The algorithm subdivides the edge and puts the two half edges P_1P_2 and P_2P_1 into an edge list that is actually a *circular queue*. The algorithm uses the queue throughout the triangulation process to keep track of the current edge, which it uses to find the next triangle. The queue's initial content is

P_2P_1, P_1P_2

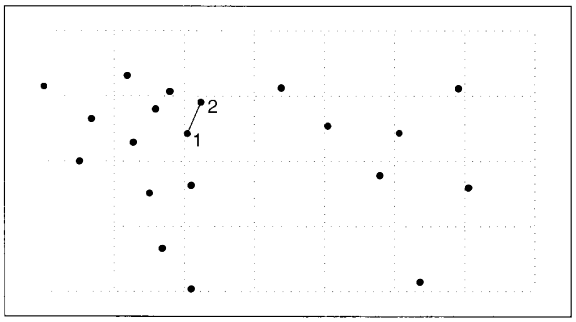


Figure 7. The first edge generated.

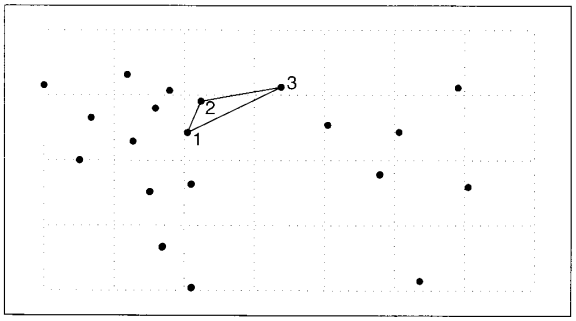


Figure 8. The first triangle generated.

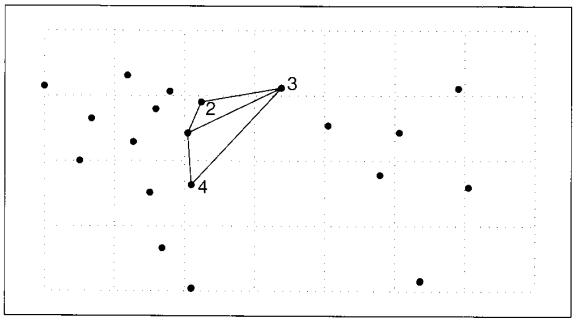


Figure 9. The second triangle generated using the new half edges.

where the current edge is always at the rear of the queue. In all the edge lists we present, the front is the first item in the list, and the back is the last item in the list. With the current edge P_1P_2, the algorithm finds the point P_3 and creates two new half edges P_3P_2 and P_1P_3 (see Figure 8). Lists of new half edges are always orderly: The first half edge points from the found point to the end of the current half edge, and the second half edge points from the beginning of the current half edge to the found point. Since the half edge P_1P_2 is not used anymore, the algorithm deletes it from the queue and adds the two new half edges to the rear in the order

$$P_2P_1,\ P_3P_2,\ P_1P_3$$

The current half edge is P_1P_3. The algorithm uses it to find the point P_4 and the half edges P_4P_3 and P_1P_4 (see Figure 9). After deletion of the current edge and addition of the new half edges, the edge list becomes

$$P_2P_1,\ P_3P_2,\ P_4P_3,\ P_1P_4$$

Continuing this process—that is, using the current half edge to create two new half edges and updating the edge list—the algorithm arrives at the important configuration shown in Figure 10. Here the edge list is

$$P_2P_1,\ P_3P_2,\ P_4P_3,\ P_5P_4,\ P_6P_5,\ P_7P_6,\ P_1P_7$$

Using the current edge P_1P_7, the algorithm finds the point P_2 and creates two new half edges P_2P_7 and P_1P_2. The half edge P_1P_2 is the dual of the half edge P_2P_1 at the front of the edge list. Since both half edges have been processed, they can be eliminated from the edge list. The new edge list is then (see Figure 11)

$$P_3P_2,\ P_4P_3,\ P_5P_4,\ P_6P_5,\ P_7P_6,\ P_2P_7$$

We call the situation in which a found half edge is the dual of the one at the front of the edge list a *right touch*. A *left touch* is

Figure 10. Configuration before triangulation closes.

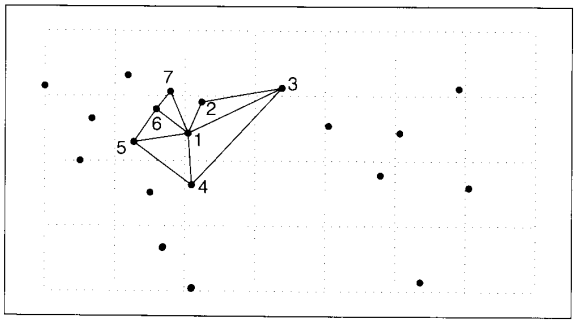
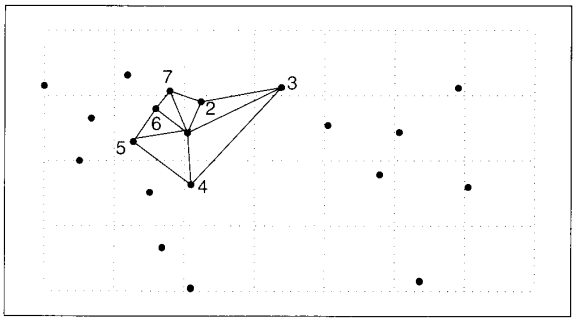


Figure 11. Point 1 is deleted from the data set.



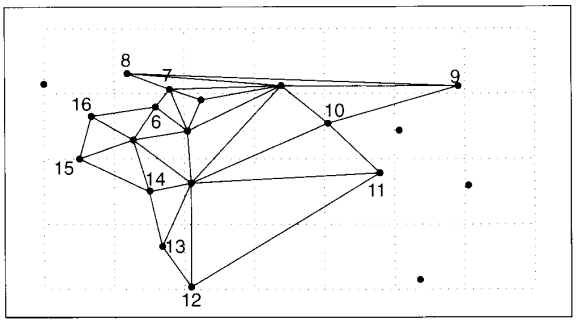


Figure 12. Triangulation configuration showing the "cave" situation.

a situation in which the found half edge is the dual of the one before the rear edge in the edge list (as we show in a later example). A touch situation occurs when the algorithm has previously used the found point. Therefore, the first time it finds each point the algorithm flags it as used. If the algorithm subsequently finds used points, it checks for touch cases. Otherwise, no touch check is necessary.

Figure 11 shows that the point P_1 is no longer needed and therefore can be eliminated from the grid structure (hence the absence of the point number). This step is essential, as it decreases the time needed to find points to form triangles (see Figure 4).

Figure 12 shows the next important stage of the triangulation process. The edge list is

P_7P_6, P_8P_7, P_9P_8, $P_{10}P_9$, $P_{11}P_{10}$, $P_{12}P_{11}$, $P_{13}P_{12}$, $P_{14}P_{13}$, $P_{15}P_{14}$, $P_{16}P_{15}$, P_6P_{16}

Using the rear edge, the algorithm finds the point P_8 and generates the half edges P_8P_{16} and P_6P_8. Since P_8 is a used point, the algorithm checks for touch cases. Unfortunately, it finds no touch with either the front edge or the one before the rear edge. This case is remedied by skipping the current edge P_6P_{16} and moving the front edge to the rear. This step is essential to avoid creating a hole or a bridge. Now the edge list is

P_8P_7, P_9P_8, $P_{10}P_9$, $P_{11}P_{10}$, $P_{12}P_{11}$, $P_{13}P_{12}$, $P_{14}P_{13}$, $P_{15}P_{14}$, $P_{16}P_{15}$, P_6P_{16}, P_7P_6

Using P_7P_6, the algorithm finds the point P_8 and creates two new half edges P_8P_6 and P_7P_8. Since P_8 is used, the algorithm checks for touch cases. It finds a right touch with the front edge P_8P_7 and therefore deletes the point P_7 from the data set and the half edges P_7P_6 and P_8P_7 from the edge list. The new edge list corresponding to the triangulation in Figure 13 is

P_9P_8, $P_{10}P_9$, $P_{11}P_{10}$, $P_{12}P_{11}$, $P_{13}P_{12}$, $P_{14}P_{13}$, $P_{15}P_{14}$, $P_{16}P_{15}$, P_6P_{16}, P_8P_6

Using the current edge in the list, the algorithm finds the point P_{16} and generates the new half edges $P_{16}P_6$ and P_8P_{16}. Since P_{16} is used, it checks for touch cases. Although it finds no right touch with the front edge, it detects a left touch with the edge before the rear edge (P_6P_{16}). The algorithm removes this edge and the current rear one from the list, and adds the newly created edge P_8P_{16} to the rear. The new edge list corresponding to the configuration in Figure 14 is

P_9P_8, $P_{10}P_9$, $P_{11}P_{10}$, $P_{12}P_{11}$, $P_{13}P_{12}$, $P_{14}P_{13}$, $P_{15}P_{14}$, $P_{16}P_{15}$, P_8P_{16}

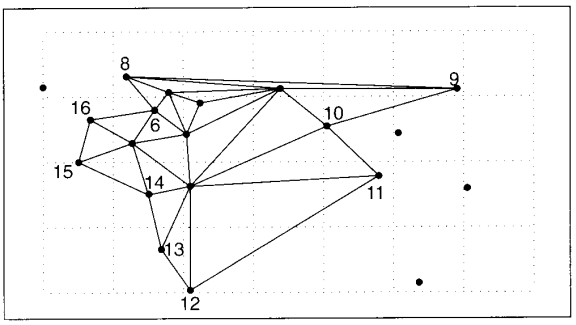


Figure 13. Right touch after the edge P_6P_{16} is skipped.

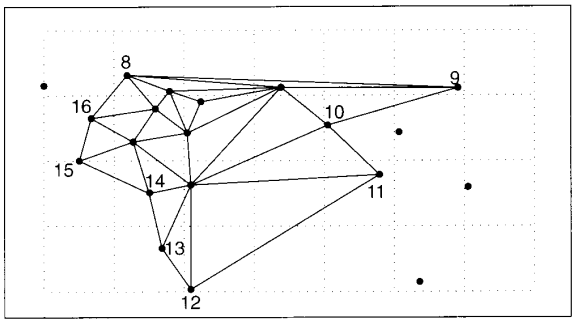


Figure 14. An example of left touch.

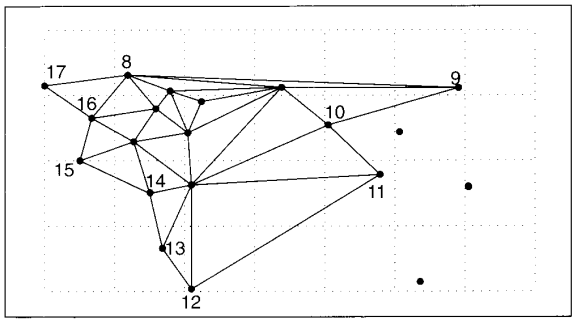


Figure 15. Boundary edge is found.

Using P_8P_{16}, the algorithm finds the point P_{17}. Since this point does not have the "used" flag set, the algorithm does not check for touch situations and adds the new half edges $P_{17}P_{16}$ and P_8P_{17} to the rear of the list, after removing P_8P_{16}. The edge list corresponding to Figure 15 is

P_9P_8, $P_{10}P_9$, $P_{11}P_{10}$, $P_{12}P_{11}$, $P_{13}P_{12}$, $P_{14}P_{13}$, $P_{15}P_{14}$, $P_{16}P_{15}$, $P_{17}P_{16}$, P_8P_{17}

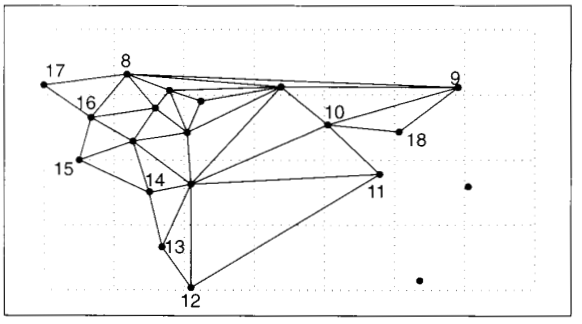
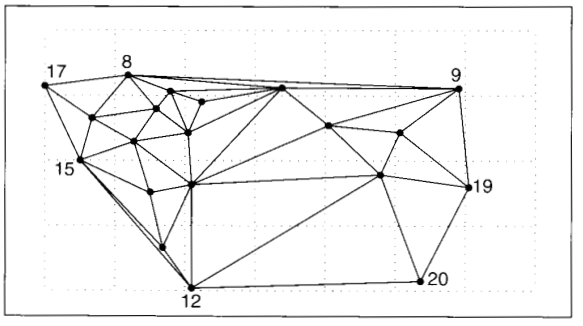


Figure 17. The triangulation is completed.



Using the rear edge P_8P_{17}, the algorithm finds no point. We refer to such an edge as a *boundary edge*. The algorithm simply deletes it from the edge list, simultaneously moving the front edge to the rear. The new edge list becomes

$$P_{10}P_9, P_{11}P_{10}, P_{12}P_{11}, P_{13}P_{12}, P_{14}P_{13}, P_{15}P_{14}, P_{16}P_{15}, P_{17}P_{16}, P_9P_8$$

The half edge P_9P_8 is again a boundary edge and is deleted from the list. The algorithm moves the front edge to the rear. Now, using $P_{10}P_9$, it finds the unused point P_{18} and generates the new half edges $P_{18}P_{10}$ and P_9P_{18}, resulting in the edge list corresponding to Figure 16:

$$P_{11}P_{10}, P_{12}P_{11}, P_{13}P_{12}, P_{14}P_{13}, P_{15}P_{14}, P_{16}P_{15}, P_{17}P_{16}, P_{18}P_9, P_{10}P_{18}$$

At this point we have presented all the possible cases. The algorithm proceeds and considers all the cases mentioned above until the edge list becomes empty, at which point the triangulation is complete. As Figure 17 shows, by the time the algorithm reaches the end, it has deleted all the data points except the boundary ones. Hence the searching part of the algorithm, which is the most time-consuming part, becomes faster and faster.

The Delaunay criterion

Here we give a proof that the algorithm produces a valid Delaunay triangulation: For each triangle computed, no data point is found inside the circumscribing circle. Since the searching part of the algorithm considers only the right-hand side of each edge, all we need to prove is that no point lies inside the circumscribing circle on the left-hand side of an edge. (Since the algorithm picked the point with the smallest cosine, no point lies inside the circle on the right-hand side of the edge.) The proof

Figure 16. Forming a new triangle with the unused point P_{18}.

has two steps. First, consider the first edge $<A, B>$ (see Figure 18a). Since B is the nearest point to A, for any point C that forms the first triangle with $<A, B>$, the edge $<A, C>$ is greater than or equal to $<A, B>$. Therefore, the angle ACB is less than 90 degrees and hence the arc not containing the point C is less than a semicircle. If the point D happens to lie inside the circle and to the left of $<A, B>$, then $<A, D>$ must be shorter than $<A, B>$, which contradicts the fact that B is the nearest point to A.

Next, consider the edge $<A, C>$ of the triangle $<A, B, C>$ computed above (see Figure 18b). We must prove that any point D that lies on the right of $<A, C>$ and outside the circle $<A, B, C>$ will produce a triangle $<A, C, D>$, whose circumscribing circle does not contain any point that lies on the left of $<A, C>$. (The circle $<A, B, C>$ does not contain any point because $<A, B, C>$ is already a Delaunay triangle.) From elementary geometry it follows that if D is outside the circle $<A, B, C>$, then we

Figure 18. Verification of the Delaunay criterion.

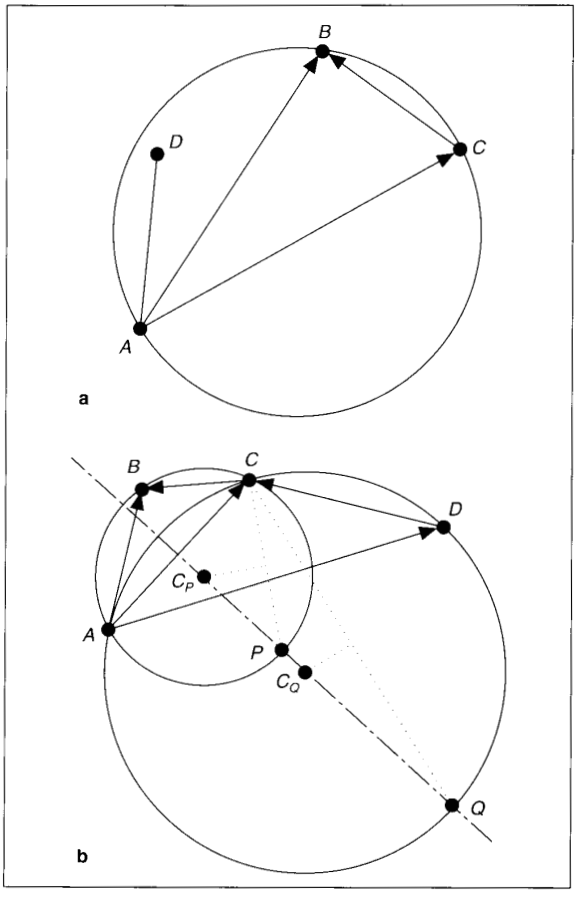


Figure 19. Structure of the algorithm.

```
initialize the half edge list with the first two half edges <P1, P2>
  and <P2, P1>;
P1.previous=P1.next=P2; P2.previous=P2.next=P1;
while(edge list is not empty)
{
  get current edge <Ps, Pe>;
  found←find_third_point(..., P, ...);
  /*third point found is P */
  if (found)
  {
    if (third point is used)
    {
      touch←check_touch_case (Ps, P, Pe);
      switch (touch)
      {
        case RIGHT_TOUCH:
          delete front half edge from the edge list;
          delete current rear half edge from the edge list;
          append first half edge to the edge list;
          Pe.previous=P; P.next=Pe;
          break;
        case LEFT_TOUCH:
          delete current rear half edge from the edge list;
          delete half edge before current one from the edge
            list;
          append second half edge to the edge list;
          Ps.next=P; P.previous=Ps;
          break;
        case NO_TOUCH:
          move front half edge to the rear;
      }
    } else /*third point was not used*/
    {
      mark third point as used;
      delete rear half edge from the edge list;
      append first half edge to the edge list;
      append second half edge to the edge list;
      P.previous=Ps; P.next=Pe; Ps.next=P; Pe.previous=P;
    }
  } else /* no point was found*/
  {
    delete rear half edge from the edge list;
    move front half edge to the rear of the edge list;
  }
}
```

can define the circle $<A, C, D>$ by a point Q lying on the perpendicular bisector of $<A, C>$ beyond the point P, where P is the intersection of the bisector with the circle $<A, B, C>$. The complementary arc of the circle $<A, Q, C>$ is contained entirely in the circle $<A, B, C>$ and is to the left of the edge $<A, C>$. Therefore, this circle cannot contain any point on the left-hand side of $<A, C>$ because the triangle $<A, B, C>$ does not contain any other data point.

Applying this argument to the edges $<A, D>$, $<A, E>$, and so on, shows that the triangulation produced by our algorithm is Delaunay triangulation.

The structure of the algorithm

Conceptually, the algorithm is very simple. Figure 19 shows the pseudocode. The check_touch_case (Ps, P, Pe) routine is

```
check_touch_case(Ps, P, Pe)
{
  if(Ps == P.next) return(RIGHT_TOUCH);
  if(Pe == P.previous) return(LEFT_TOUCH);
  return(NO_TOUCH);
}
```

Computing the convex hull

The algorithm offers two effortless ways to compute the convex hull: the first based on the edge list, and the second on the grid structure. The edge-list-based method saves all boundary edges before deleting them from the edge list. With the example we presented earlier for triangulation, the algorithm finds the boundary edges in the following order:

$$P_8P_{17},\ P_9P_8,\ P_{12}P_{20},\ P_{15}P_{12},\ P_{17}P_{15},\ P_{19}P_9,\ P_{20}P_{19}$$

Since these edges are ordered consistently to begin with, they hook up automatically. That is, P_8P_{17} is followed by $P_{17}P_{15}$, which is joined to $P_{15}P_{12}$, and so on, and the sequence of edges is closed by the edge P_9P_8.

The grid-structure-based method is even simpler. Figure 17 shows that all the points remaining in the grid structure at the end of the triangulation process are boundary points. In addition, each point is associated with two pointers, previous and next, as explained in earlier sections. Now, starting at any point remaining in the grid structure, the algorithm can march along the convex hull either forward (always following the direction defined by the pointer next) or backward (following the path suggested by the pointer previous). As a practical concern, the algorithm computes the directed convex hulls; that is, it outputs the closed polygon either clockwise or counterclockwise.

Degeneracies

We need to deal with two degeneracies: collinear points and coincident points. If the entire data set consists of collinear points, then we have special considerations because the algorithm finds no points on the left or right of the first edge. Since in this case the edge list is empty, the program quits without computing the edges along the collinear points. A simple fix is to check for such a situation and to output the line segments based on the ordering inherent in the grid structure.

If the data set is not entirely collinear, then no special treatment is necessary. The algorithm finds points not lying on the line, and they form triangles that make the subset of points noncollinear.

Coincident points do not cause any problem, as they are eliminated during building of the grid structure.

Time complexity

Computational geometers consider time complexity as a theoretical exercise: They look for the order of algorithm growth from a purely theoretical point of view. The formula they arrive at may or may not reflect the algorithm's actual cost, because

Table 1. Timing examples in CPU seconds for randomly distributed points.

Number of Points	Building the Grid Structure	Triangulation
500	0.274725	1.351648
1,000	0.494505	2.681319
1,500	0.714286	3.989011
2,000	0.879121	5.285714
2,500	1.043956	6.670330
3,000	1.318681	7.989011
3,500	1.483516	9.329670
4,000	1.648352	10.670329
4,500	1.868132	11.956044
5,000	2.087912	13.230769

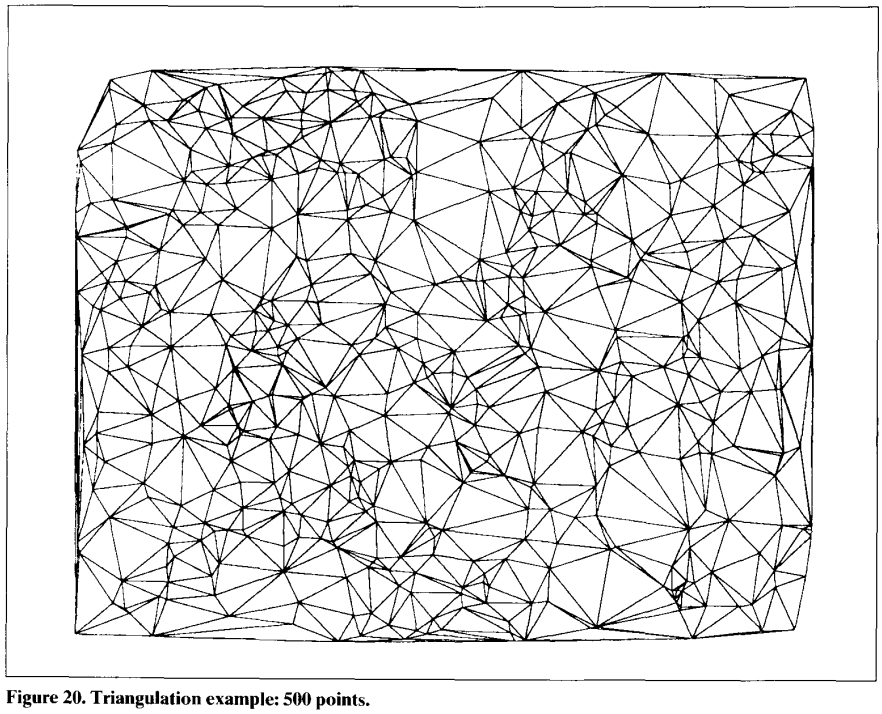


Figure 20. Triangulation example: 500 points.

they have not considered many implementation details. In this section we present a practical complexity analysis based on actual testing of the algorithm for a variety of data. The hardware we used was an IBM PS/2 486/25 running under IBM DOS 5.0. Table 1 shows timing examples, in CPU seconds, for randomly distributed points ranging in number from 500 to 5,000. Since randomly distributed points give different results for different data sets, we ran five examples of each—for example, 5 times 1,000 points—and averaged the times.

Figure 20 illustrates a data set typical of the ones we used. Figure 21 shows the graphical relationship between the number of points and the triangulation times. We tested the algorithm with a large number of data points and found the practical time complexity to be (steadily) linear.

Output generation

For the triangulation to be useful for further processing, we must store the computed triangles. We could create the output data structure in a number of ways—for example, using the concept of the winged-edge data structure. The method we propose here is based on a point list. We chose this technique because it permits output creation on the fly. In fact, we can replace the edge list by the point list and control the triangulation with this list. We introduce our new data structure and show how it can be used to control the triangulation and to form the output at the same time.

Consider the example shown in Figure 22. The algorithm uses two point lists: In the first it lists all interior points; in the second, all boundary points. For each point, it lists in clockwise order all points that surround that point. Figure 23 shows the data structure representing the triangles, edges, and points of the triangulation shown in Figure 22.

The points in the first column below the double line are boundary points, whereas those above the double line are internal ones. The data structure lets us answer all geometric queries, for example

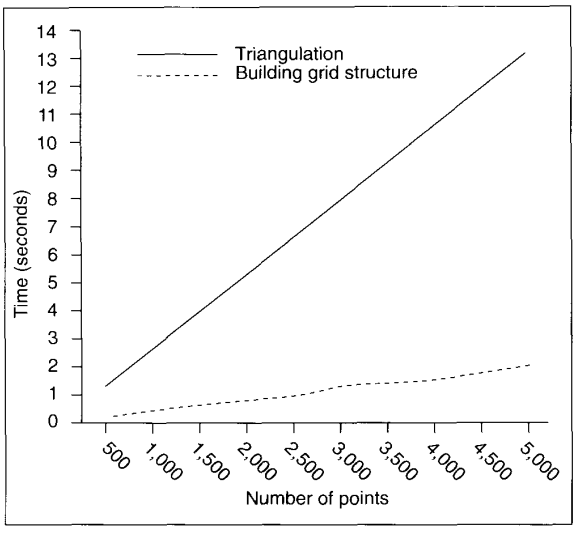


Figure 21. Graphical illustration of time complexity.

- All triangles incident at a point. For point 3: <3, 5, 6>, <3, 6, 4>, <3, 4, 1>, <3, 1, 2>, and <3, 2, 5>. For the boundary point 5: <5, 8, 6>, <5, 6, 3>, and <5, 3, 2>.
- All triangles incident on an edge. For edge <3, 6> we get <3, 6, 4> from 3's list and <6, 3, 5> from 6's list. If the edge is <2, 1>, then from 2's list it follows that <2, 1> is a boundary edge, and from 1's list we obtain <1, 2, 3>.
- All edges incident at a point. For 7: <7, 4>, <7, 6>, and <7, 8>.
- All triangles adjacent to a given triangle. If the given triangle is <3, 6, 4>, then we seek the triangles incident on the half edges <3, 6>, <6, 4>, and <4, 3>. From 6's list we get <6, 3, 5>, from 4's list <4, 6, 7>, and from 3's list <3, 4, 1>.

The data structure shown in Figure 23 is very compact. We can write simple subroutines that perform most data-structure-browsing operations.

We now use the figures in the subsection "Putting triangles together" to explain how to form triangles and get the data structure simultaneously.

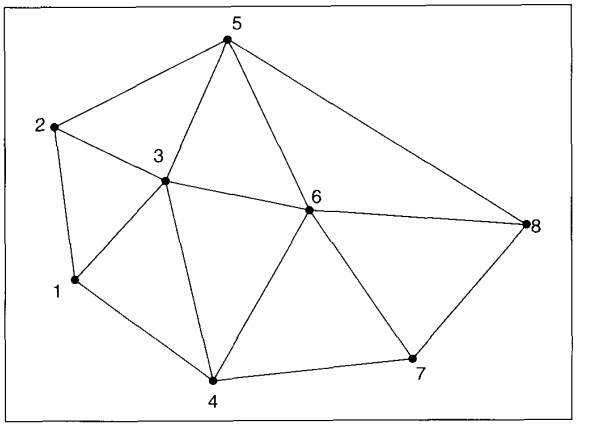
In Figure 7 the first edge is obtained. The algorithm puts the two half edges into the lists shown in Figure 24a. The current edge is <1, 2>, which is $F(1)L(1)$. After it finds point 3 (see Figure 8), the algorithm updates the list:

1. Append 3 to the list of F.
2. Insert 3 into the second position of L's list.
3. Create a new list and add 3, and append $F(1)$ and $L(1)$ to it.
4. Advance the pointer L.

Figure 24b shows the new list corresponding to Figure 8. The current edge is $F(1)L(1)$ = <1, 3> and the new point found is 4 (see Figure 9). Following the same procedure, the algorithm updates the edge list as shown in Figure 24c.

The first important case is the right touch shown in Figure 10. Figure 24d shows the corresponding edge list. The current edge is <1, 7>. The algorithm finds the point 2, detects a right touch (using the simple check_touch_case routine described earlier, and updates the list:

Figure 22. Example of triangular network to illustrate the data structure used.



1. Save the list of F.
2. Append $L(1)$ to the end of the list following F.
3. Insert $F(2)$ into the second position of L.
4. Advance F.

Figure 24e shows the edge list, corresponding to Figure 11, after right-touch processing.

Next is the no-touch case shown in Figure 12. Figure 24f shows the corresponding point lists. The current edge is <6, 16>. The algorithm finds the point 8 and processes the no-touch case:

1. Copy F's list to the end.
2. Advance F and L.

3	5	6	4	1	2
6	8	7	4	3	5
5	8	6	3	2	
8	7	6	5		
7	4	6	8		
4	1	3	6	7	
1	2	3	4		
2	5	3	1		

Figure 23. Data structure representing the triangulation shown in Figure 22.

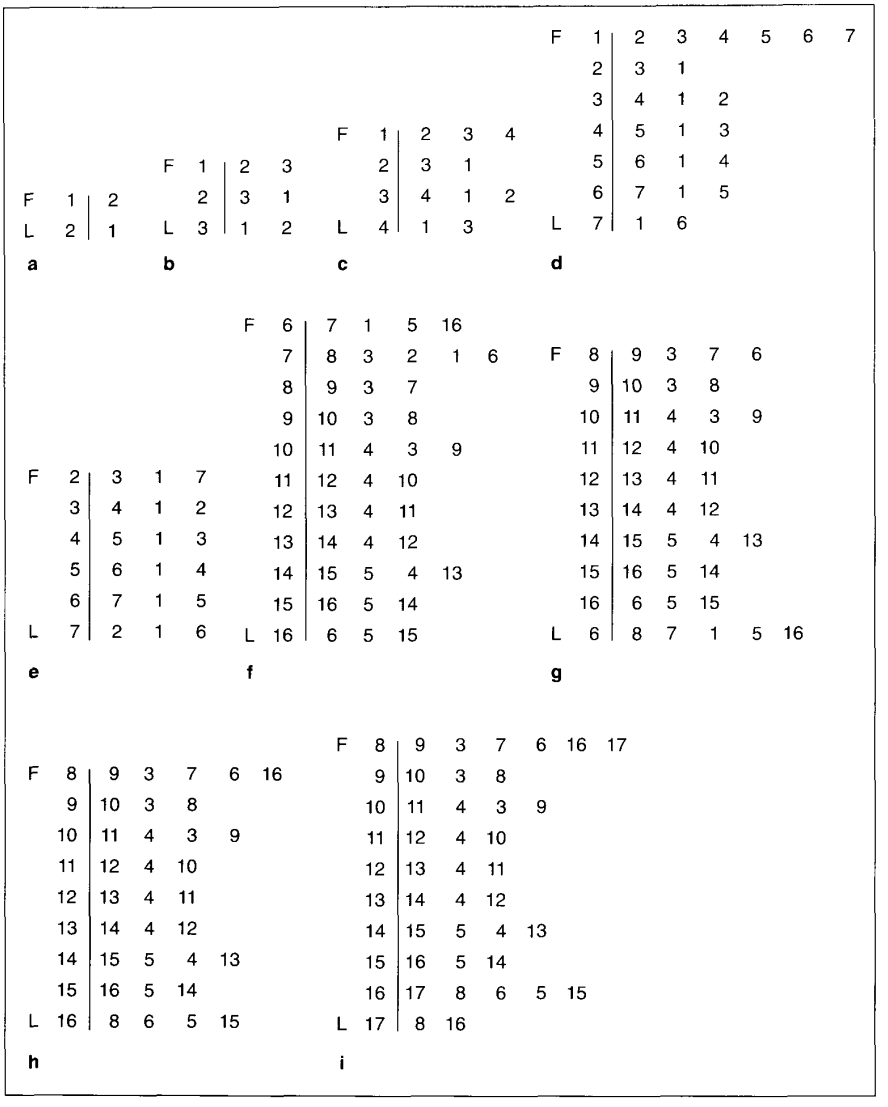


Figure 24. Updating the edge lists.

If the lists are arranged in a circular fashion, as we recommend, then the pointers F and L are simply advanced.

Figure 13 illustrates the left touch. Figure 24g shows the corresponding lists. The current edge is <8, 6>. The algorithm finds the point 16, detects the left touch, and processes it:

1. Save the list of L.
2. Append L (last) to the end of F's list.
3. Decrement L.
4. Insert $F(1)$ into the second position of the list of L.

Figure 24h shows the point lists corresponding to Figure 14.

Figure 15 shows the final configuration, and Figure 24i shows the corresponding data structure. The current edge is <8, 17>. The algorithm finds no point: The edge is a boundary edge. In this case it simply advances both pointers F and L.

At the end of the triangulation process we have two kinds of data:

1. Point lists that are saved (after each left and right touch).
2. Lists remaining in the internal circular structure.

Figure 25a shows the saved point lists according to Figure 17. Figure 25b shows the points remaining in the circular list.

This list (in Figure 25b) contains all the boundary points. Traversing them from

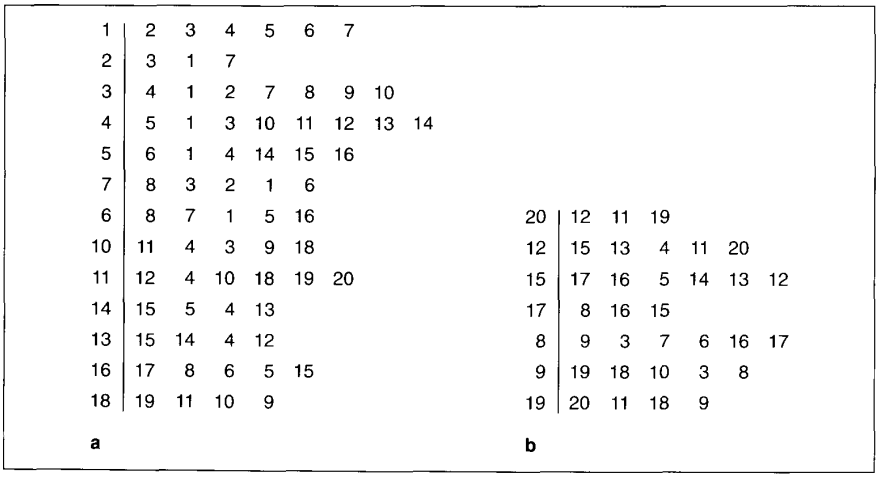


Figure 25. (a) Saved point lists. (b) Boundary point lists.

Figure 26. Full data structure.

1	2	3	4	5	6	7		
2	3	1	7					
3	4	1	2	7	8	9	10	
4	5	1	3	10	11	12	13	14
5	6	1	4	14	15	16		
7	8	3	2	1	6			
6	8	7	1	5	16			
10	11	4	3	9	18			
11	12	4	10	18	19	20		
14	15	5	4	13				
13	15	14	4	12				
16	17	8	6	5	15			
18	19	11	10	9				
20	12	11	19					
12	15	13	4	11	20			
15	17	16	5	14	13	12		
17	8	16	15					
8	9	3	7	6	16	17		
9	19	18	10	3	8			
19	20	11	18	9				

top to bottom gives the convex hull of the data set. The algorithm obtains the full data structure that represents the triangulation (Figure 17) by merging the two lists, as shown in Figure 26.

Conclusions

Our simple algorithm for triangulating 2D data points is based on a uniform grid structure and a new triangulation strategy that builds triangles in a circular fashion. This new strategy lets the algorithm eliminate points from the internal data structure and hence decreases the time to find points to form triangles, given an edge. The algorithm has a tested linear time complexity that significantly improves upon other methods. As a by-product, the algorithm produces the convex hull of the data set at no extra cost. ❑

Acknowledgments

The work reported in this article was supported by the Florida High Technology and Industry Council. We thank the referees for their valuable comments and suggestions.

References

1. G. Voronoi, "Nouvelles applications des paramätres continus à la théorie des formes quadratiques. Premier Mémoire: Sur quelques proprieteés des formes quadratiques positives parfaites," *J. reine angewandte Mathematik*, Vol. 133, 1907, pp. 97-178.
2. B. Delaunay, "Neue Darstellung der geometrischen Krystallographie," *Zeitschrift Krystallographie*, Vol. 84, 1932, pp. 109-149.
3. F. Preparata and M. Shamos, *Computational Geometry—An Introduction*, Springer-Verlag, New York, 1985.
4. H. Edelsbrunner, *Algorithms in Combinatorial Geometry*, Springer-Verlag, New York, 1987.
5. P.J. Green and R. Sibson, "Computing Dirichlet Tessellations in the Plane," *Computer J.*, Vol. 21, No. 2, Feb. 1978, pp. 168-173.
6. R. Sibson, "Locally Equiangular Triangulations," *Computer J.*, Vol. 21, No. 3, March, pp. 243-245. [**Au: Year?**]
7. A. Bowyer, "Computing Dirichlet Tessellations," *Computer J.*, Vol. 24, No. 2, Feb. 1981, pp. 162-166.
8. D.F. Watson, "Computing the *n*-dimensional Delaunay Tessellation with Applications to Voronoi Polytopes," *Computer J.*, Vol. 24, No. 2, Feb. 1981, pp. 167-172.
9. D.T. Lee and B.J. Schachter, "Two Algorithms for Constructing Delaunay Triangulation," *Int'l J. Computer and Information Science*, Vol. 9, No. 3, 1980, pp. 219-242.
10. L. Guibas and J. Stolfi, "Primitives for the Manipulation of General Subdivisions and the Computation of Voronoi Diagrams," *ACM Trans. Graphics*, Vol. 4, No. 2, April 1985, pp. 74-123.
11. L. Guibas, D.E. Knuth, and M. Sharir, "Randomized Incremental Construction of Delaunay and Voronoi Diagrams," Tech. Report 481, Computer Science Dept., Courant Inst. of Mathematical Sciences, New York Univ., 1990.
12. A.R. Forrest, "Computational Geometry and Software Engineering: Towards a Geometric Computing Environment," in *Techniques for Computer Graphics*, D.F. Rogers and R.A. Earnshaw, eds., Springer-Verlag, New York, 1987, pp. 23-37.

Tsung-Pao Fang is a PhD candidate in the Department of Computer Science and Engineering at the University of South Florida. He holds a BS degree in earth sciences from the National Cheng Kung University in Taiwan and an MS degree in computer science from the University of South Florida. His research interests are in applied computational geometry, computer graphics, and algorithm design.

Les A. Piegl is a professor in the Department of Computer Science and Engineering at the University of South Florida. His research interests are CAD/CAM, geometric modeling, user interface design, data structures and algorithms, and computer graphics. He has recently turned to applied computational geometry, in particular triangulation and mesh generation in two and three dimensions.

Piegl holds an MS in mathematics and a PhD in applied computing. He is a member of ACM, SIAM, IEEE Computer Society, and Eurographics. He serves as editor for *Computer-Aided Design*.

Contact Piegl at the Department of Computer Science and Engineering, University of South Florida, 4202 E. Fowler Ave., ENG 118, Tampa, FL 33620.