# Problem 1. Algorithm

## Describe the representation of a convex polyhedron. Show the representation of a tetrahedron.

Since a convex polyhedron can be interpreted as a planar graph, we can use a doubly-connected edge list to represent a convex polyhedron. One example of the representation of a tetrahedron by using a doubly-connected edge list is shown in Fig. 1.

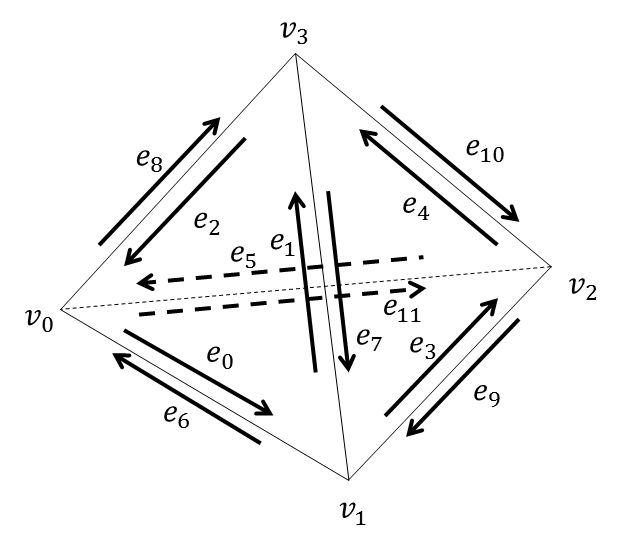


Figure 1. One example of the representation of a tetrahedron.

Around the each vertex, the half edge is ordered by the clockwise order. In the case of Fig. 1, edges is the clockwise order around the vertex , for instance. The entire ordered list is shown in the Table. 1.

Table 1. The ordered list of edges around each vertex

|  |  |
| --- | --- |
| Vertex | Ordered list of edges |
|  |  |
|  |  |
|  |  |
|  |  |

## Explain how to represent the conflict graph by adding pointers to this representation.

The conflict graph is represented by adding the pointers of the visible faces to each unadded vertices and the pointers of the visible vertices to each hull faces. In this manner, each vertex can look up all the visible faces by just scanning these pointers, and each face can look up all the visible vertices by just scanning these pointers. The declaration of the classes Vertex and Face should be modified as shown in Fig. 2 and Fig. 3.

class Vertex {

public:

vector<Face\*> visibleFaces;

...(snip)...

};

Figure 2. The declaration of class Vertex

class Face {

public:

vector<Vertex\*> visibleVertices;

...(snip)...

};

Figure 3. The declaration of class Face

## Which vertices, edges, and faces are removed when a vertex is added to the hull?

When a vertex is added, we want to remove all the visible faces from . We also want to remove all the edges that constitute those faces except the horizon edges. For vertices, we do not explicitly remove the visible vertices, because those vertices are implicitly removed from the hull when the corresponding edges are removed.

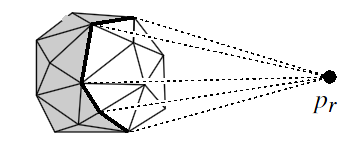


Figure 4. All the visible faces are to be removed when a vertex is added.

To find the visible faces, we just need to look up the conflict graph.

## How can one horizon edge be found? How can be the next one in clockwise order by found?

One horizon edge can be found by traversing all the constituting edges of the visible faces until one horizon edge is found. For each edge, we check whether the face of its twin edge is visible or not. If it is not visible, then this edge is a horizon edge, so we stop traversing.

Once we find the first horizon edge, we can get the entire horizon edges by the following process. Let be the newly added vertex and be the first horizon edge we found. Its twin edge is directed in the clockwise order from the view of . Then we traverse all the edges which go out from the tail of until we find the next horizon edge. The pseudo code of these steps are shown in Fig. 5.

FindNextHorizonEdge(vertex , edge )

1 = ->twin

2 e =

3 while (true)

4 if e->twin->face is visible from then

5 break;

6 e = e->next

6 end while

7 return e

Figure 5. The pseudo code to find the next horizon edge

## Which vertices, edges, and faces are added to the hull? Which are removed?

When a vertex is added, we add edges that connect with all the vertices on the horizon, and add the corresponding faces as shown in Fig. 6. When we add these edges and faces, we also have to maintain the representation of the convex hull, especially the clockwise order of edges around each vertex. Let and be the vertices on one of the horizon edges, be the edge from to , be the edge from to , be the edge from to , be the edge from to , be the edge from to , be the edge from to , and be the face surrounded by the edge , , and . Then, the order of the edges will be updated as follows:

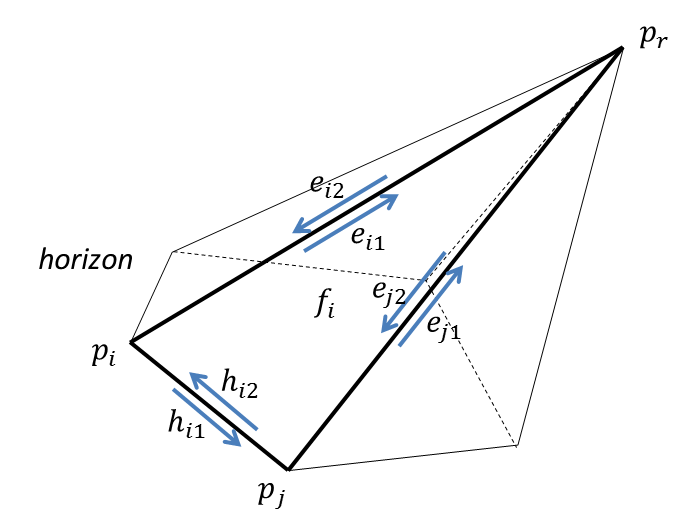


Figure 6. After all the visible faces are removed, edges that connect horizon with the newly added vertex and the corresponding faces are to be added, and these edges are to be ordered in counter-clockwise order around .

For vertices, since all the vertices are added to the arrangement at the beginning, we do not need to add the vertex any more.

## How does the conflict graph change?

First, we list up all the vertices that are visible from faces that are adjacent to the horizon edges. These vertices are to be used for the visibility test for the newly added faces later. Since we also want to remove the newly added vertex, we exclude this vertex from the list. Next, we remove all the nodes that correspond to the visible faces, and remove all the outgoing edges from those nodes. Since we maintain the conflict graph by the pointers as I explained in (b), we do not explicitly store the edges of the conflict graph. Instead, we remove all the visible faces from the arrangement and from the pointers of the vertices that are visible from those visible faces. Then, we want to remove the newly added vertex from the conflict graph, but we do not do anything for this, because the newly added vertex does not have any edges any more. Lastly, for all the newly added faces, we check if they are visible from the vertices that were listed up in the first step. If a face is visible from a vertex, the vertex is added to the pointers of the face, and the face is added to the pointers of the vertex. The pseudo code of these steps is shown in Fig. 7.

UpdateConflictGraph

1 List up all the vertices that are visible from faces that are adjacent to the horizon edges.

2 For each visible face

3 For each vertex ->visibleVertices

4 ->removeFace()

5 End for

6 arr->removeFace()

7 End for

8 For each vertex list retrieved in line 1

9 For each newly added face

10 If is visible from Then

11 ->visibleFaces.push\_back()

12 ->visibleVertices.push\_back()

13 End for

14 End for

Figure 7. The pseudo code of updating the conflict graph

# Problem 2. Implementation

Please type the following command to compile and execute my implementation:

$ make

$ ./ps3-nishida-1

One of the test data that I used to test is as follows:

$ ./ps3-nishida-1

10 0 0 0 1 1 1 5 5 10 5 5 -5 10 10 0 8 8 -1 0 10 0 7 3 1 5 5 -10 10 0 0

Then, the program outputs the following line.

8 9 0 8 2 0 9 9 8 4 2 9 4 0 2 6 2 4 6 4 8 6 8 0 6

The resulting convex hull is shown in Fig. 8, which was visualized by *ParaView*.

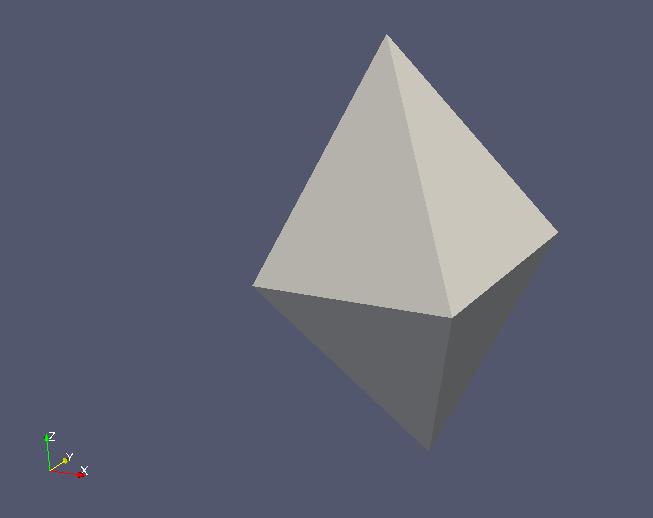


Figure 8. The resulting convex hull is visualized by *ParaView*.

# Problem 3. Delaunay Triangulation

Please type the following command to compile and execute my implementation:

$ make

$ ./ps3-nishida-2

One of the test data that I used to test is as follows:

$ ./ps3-nishida-2

6 0 0 5 -5 10 5 5 5 10 15 0 10

Then, the program outputs the following line.

5 1 3 0 2 3 1 3 5 0 2 4 3 3 4 5

The resulting Delaunay triangulation is shown in Fig. 9.

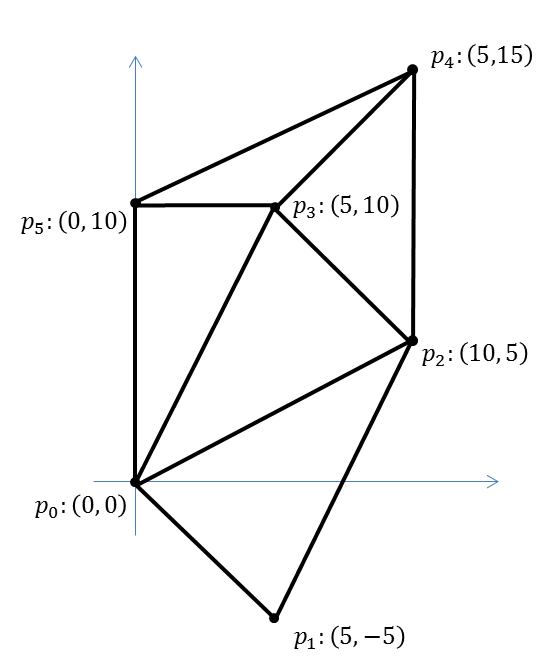


Figure 9. The result of the Delaunay triangulation of the test data.