

Trabajo Final Algoritmos II

Triangulación de Delaunay y Diagrama de Voronoi



Alumno:

Emiliano Migliorata (emigliorata@gmail.com)

1. Introducción

En el presente informe se explica la implementación de una aplicación que, mediante el ingreso de una nube de puntos, procede a calcular la triangulación de Delaunay. El algoritmo elegido para la implementación es el llamado “Incremental” donde comenzamos con un triángulo ficticio infinitamente grande y vamos agregando un vértice a la vez. Al agregar el vértice se realizan los chequeos y modificaciones de aristas para que al finalizar, la triangulación cumpla las propiedades correspondientes.

Por otra parte, la aplicación desarrollada permite calcular el diagrama de Voronoi a partir de una triangulación de Delaunay. Además, la aplicación posee una pequeña interfaz gráfica la cual nos permite observar con mejor claridad la salida de los algoritmos.

Delaunay:

Una triangulación de Delaunay es un tipo especial de triangulación donde cada triángulo cumple la propiedad de que la circunferencia definida por sus vértices no contiene a otros puntos de la triangulación.[1]

En la *Figura 1*, la triangulación 1 no es Delaunay, mientras que la triangulación 2 si lo es.

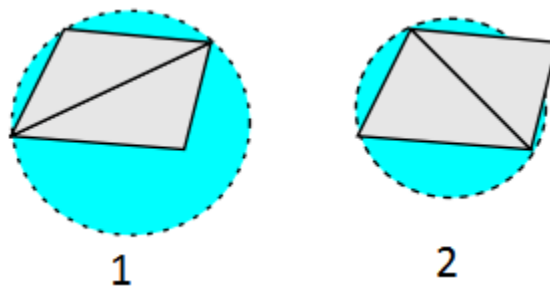


Figura 1. Triangulación de Delaunay

Dentro todas las triangulaciones de una nube de puntos, las de Delaunay son una alternativa interesante ya que tienden a generar triángulos de buena calidad maximizando el ángulo mínimo de todos los triángulos de la triangulación.

Voronoi:

Una de las herramientas fundamentales de la Geometría Computacional son los Diagramas de Voronoi. La idea detrás de los mismos es la de, dado un conjunto de objetos geométricos S en un cierto espacio E , descomponer E en "regiones de influencia" de los mismos. A estas regiones de influencia se las llama regiones de Voronoi.[2]

Las aplicaciones que surgen a partir de este diagrama son muchas, todos los problemas conocidos como "de la oficina postal". El problema consiste en determinar a qué región pertenece un punto dado, en nuestra analogía las oficinas postales son los puntos y la ciudad corresponde con la región en sí. Otros ejemplos concretos son la ubicación de torres celulares, ubicación de centros de salud en una ciudad, etc.

En la Figura 2 se observa la clara relación que existe entre la triangulación de Delaunay y el diagrama de Voronoi. Siendo Voronoi representado por las líneas punteadas y Delaunay las líneas gruesas.

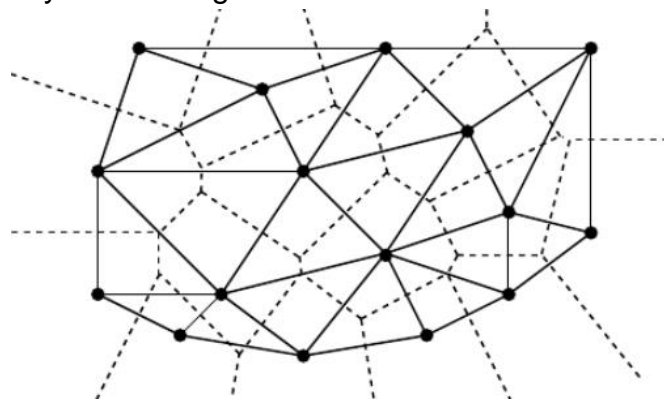


Figura 2. Delaunay y Voronoi en el mismo gráfico.

Por la *Figura 2* obtenemos que los vértices de Delaunay son los puntos Voronoi y las aristas Delaunay son segmentos que unen puntos Voronoi cuyas regiones son adyacentes.

2. Diagrama de clases

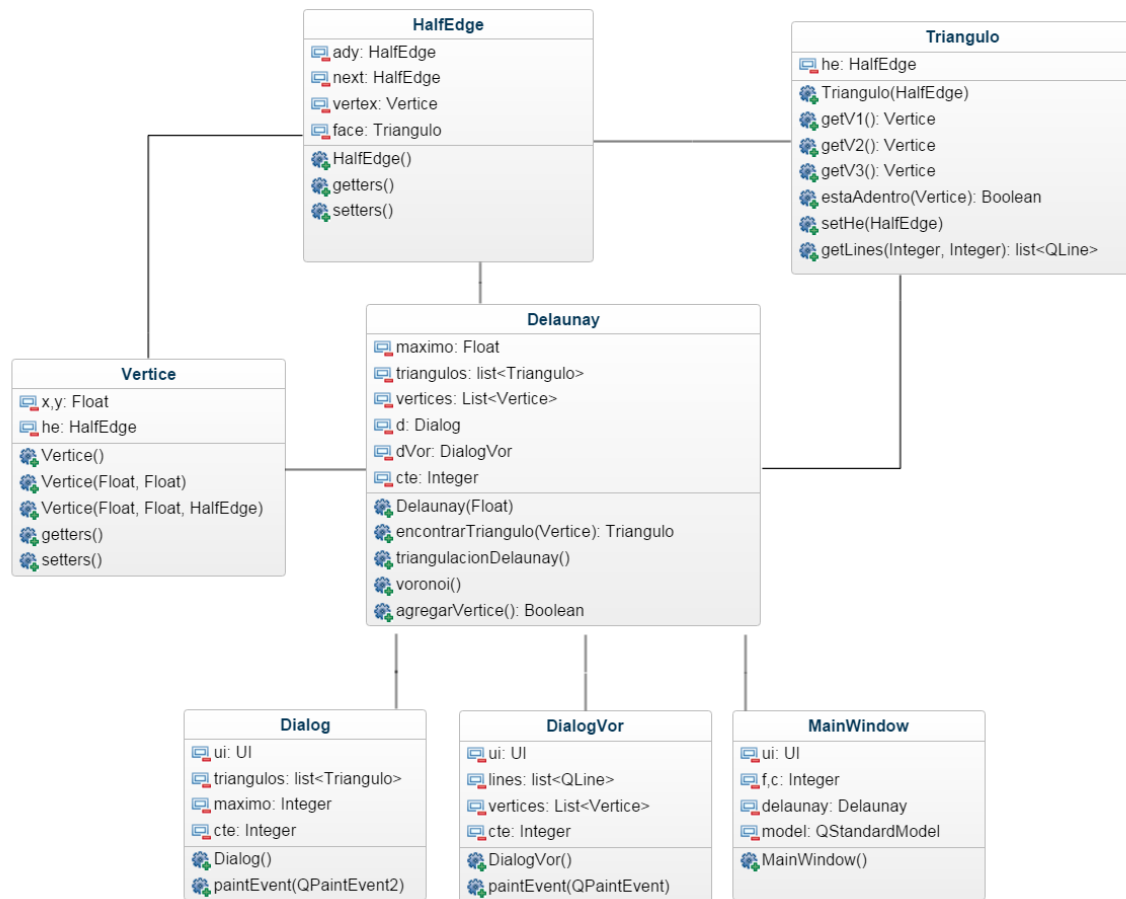


Figura 3. Diagrama de Clases.

HalfEdge: es la clase encargada del manejo de la estructura, posee dos punteros HalfEdge, un vértice asociado y un triángulo asociado. En la siguiente sección se explica mejor todo lo relacionado con la estructura antes mencionada.

Delaunay: posee los algoritmos que permiten la triangulación en sí, mediante el método *triangulacionDelaunay()* obtenemos los triángulos asociados a Delaunay. Por otro lado si ejecutamos la función *voronoi()* esta calculará Delaunay y procederá, a partir de la triangulación obtenida a calcular el diagrama de Voronoi. Los vértices para realizar la triangulación se agregan mediante el método *agregarVertice()*.

Dialog: permite dibujar en un cuadro de dialogo la triangulación de Delaunay. Mediante la utilización de QPainter se trazan todos los triángulos correspondientes, estos los setea antes de mostrar el diálogo la clase Delaunay al finalizar la triangulación.

DialogVor: de la misma forma que el Dialog, permite dibujar líneas para formar el diagrama de Voronoi, en este caso los polígonos a dibujar no son triángulos sino que pueden tener más de tres lados.

MainWindow: es la encargada de manejar la interfaz principal, ejecutando los métodos correspondientes según la funcionalidad a realizar. En las próximas secciones se explicarán las funcionalidades que se pueden realizar desde la pantalla principal.

3. Implementación

Estructura de Datos:

Luego de investigar qué representación era la más conveniente para calcular la triangulación de Delaunay y el diagrama de Voronoi se decidió utilizar la estructura Half Edge propuesta por Müller en 1978, una estructura muy utilizada en algoritmos de geometría computacional. El principio de Half Edge es dividir cada arista en dos aristas dirigidas como se observa en la *Figura 2*. [7]

Cada Half-edge almacena la siguiente información:

- Half-edge adyacente.
- Siguiendo Half-Edge en sentido anti horario.
- Face o Cara a la izquierda del half-edge (en nuestro caso un triángulo).
- Vértice asociado (al que apunta).

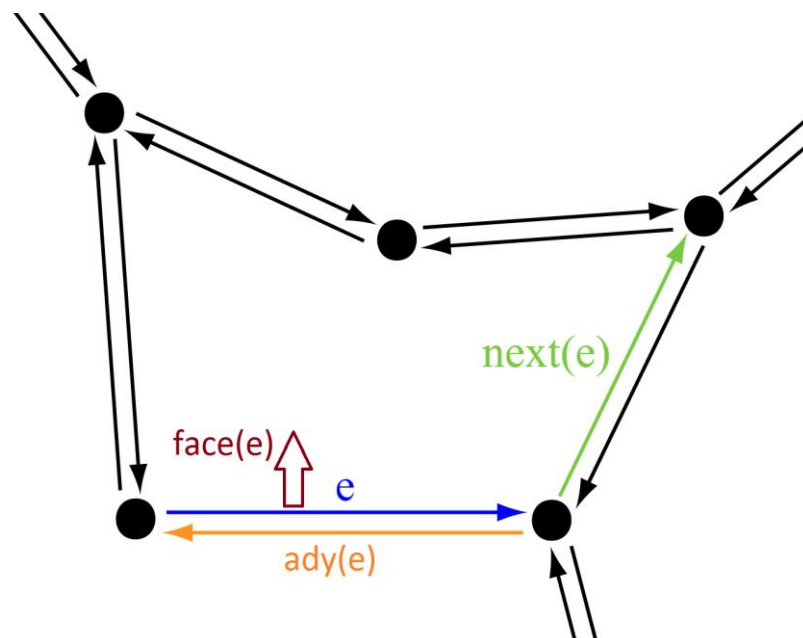


Figura 2. Representación de un polígono con Half-Edge.

Cabe aclarar que en la Figura 2 se muestra a modo de ejemplo para la representación de un polígono de cinco lados, en el caso de Delaunay al ser una triangulación obtendremos siempre polígonos de tres lados.

Para comprender un poco más y observar la simplicidad con la que se puede recorrer la estructura, se ejemplifican ciertos casos que fueron muy utilizados para la implementación de la triangulación de Delaunay y el diagrama de Voronoi.

Dado un HalfEdge obtener los tres vértices de un triángulo(sentido anti horario):

```
Vertice* v1=he->getVertex();  
Vertice* v2=he->getNext()->getVertex();  
Vertice* v3=he->getNext()->getNext()->getVertex();
```

Dado un HalfEdge obtener el vértice opuesto a una arista (legalización):

```
Vertice* vAdy=he->getAdy()->getNext()->getVertex();
```

Dado un HalfEdge obtener el triángulo adyacente a la arista (Voronoi):

```
Triangulo* tAdy=he->getAdy()->getFace();
```

Como se puede observar estas operaciones nos resultan muy conveniente en nuestro caso, la complejidad temporal para obtener los vértices anteriores es $O(1)$, mientras que si se utilizara una estructura de grafos normal se deberían realizar búsquedas exhaustivas sobre todos los vértices/triángulos. También, podemos observar en la *Figura 2* que recorriendo la estructura mediante los punteros he, siempre obtenemos los vértices en sentido anti horario. Esto nos simplifica mucho a la hora de calcular *dentroDelCirculo()* ya que necesita los vértices dados en sentido anti horario.

Pseudocódigo

Delaunay:

Proceso de inserción de un vértice y generación de la triangulación de Delaunay:

```
insertar(v){  
    tabc=encontrarTriangulo(v);  
    obtengoVerticesTriangulo a,b,c;  
    creoTriangulosNuevos tva,tvb,tvc;  
    creoYajustoHalfEdges();  
    legalizar(tva);  
    legalizar(tvb);  
    legalizar(tvc);  
}  
  
legalizar(triangulo){  
    obtengoVertices(t);
```

```

for(v:v1,v2,v3){
    d=verticeOpuesto(v);
    if(dentroDeCirculo(d)){
        flipArista(vd);
    }
}

```

Para la función *flipArista(vd)* por simplicidad se explicará mediante un ejemplo:

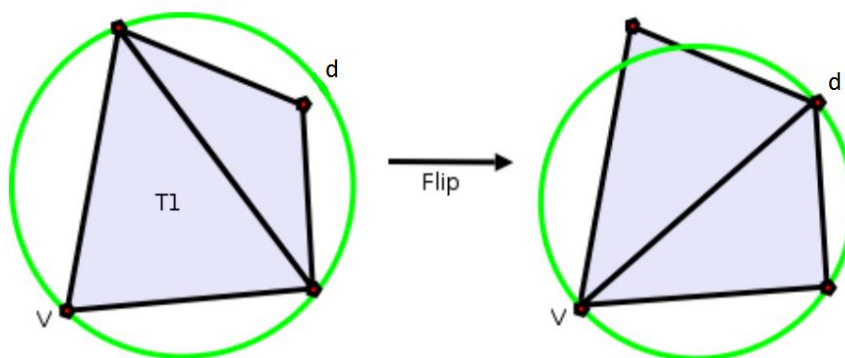


Figura 3. *flipArista(vd)*.

La función *flipArista()* entra en juego cuando queremos realizar el chequeo donde un vértice que se encuentra en los alrededores de nuestro nuevo triángulo cae dentro de la circunferencia formada por los tres puntos (rompe la propiedad de Delaunay). Entonces se procede a realizar el “flip” (rotar) de la arista ilegal para legalizar la triangulación. Debido a la estructura que estamos manejando debemos cambiar varios punteros para dejar nuestra representación de manera consistente. Cabe aclarar que luego de realizar el flip se debe volver a chequear que los nuevos triángulos formados sigan cumpliendo la propiedad de Delaunay.[3]

Para la función *dentroDeCirculo(A,B,C,D)* que se encarga de realizar el chequeo de si el vértice D cae dentro de la circunferencia formada por los vértices A,B,C, se realiza el determinante de la Figura 4 y se verifica si es positivo.

$$\begin{vmatrix} A_x & A_y & A_x^2 + A_y^2 & 1 \\ B_x & B_y & B_x^2 + B_y^2 & 1 \\ C_x & C_y & C_x^2 + C_y^2 & 1 \\ D_x & D_y & D_x^2 + D_y^2 & 1 \end{vmatrix} = \begin{vmatrix} A_x - D_x & A_y - D_y & (A_x^2 - D_x^2) + (A_y^2 - D_y^2) \\ B_x - D_x & B_y - D_y & (B_x^2 - D_x^2) + (B_y^2 - D_y^2) \\ C_x - D_x & C_y - D_y & (C_x^2 - D_x^2) + (C_y^2 - D_y^2) \end{vmatrix} > 0$$

Figura 4.

En la inicialización del triángulo “infinito” se concluyó que nuestro triángulo va a estar formado por los vértices $(3*M,0);(0,3*M);(-3*M,-3*M)$. Estos valores son

suficientemente grande para contener a todos los puntos que se ingresen a la triangulación sin que se produzca ningún error.

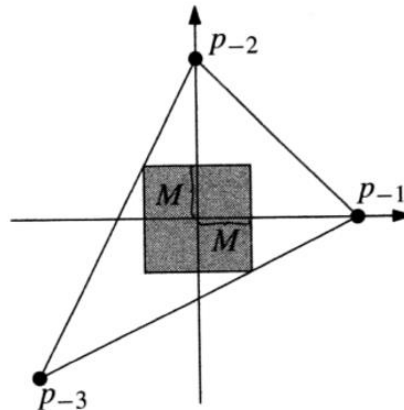


Figura 5. Triángulo "infinito" que contiene todos los puntos

Esto es lo que se realiza antes de comenzar la triangulación (*inicialización*), ya que luego se van agregando los vértices que queremos triangular verdaderamente y el triángulo ficticio nos ayuda a poder lograrlo.

Voronoi:

```
voronoi(){
    triangulacionDelaunay();
    for(t:triangulosDelaunay){
        centro=getCentroTriang(t);
        agregarLinea(he,centro);
        agregarLinea(he->getNext(),centro);
        agregarLinea(he->getNext()->getNext(),centro);
    }
}
```

Para aclarar un poco más este algoritmo, se explicará la función *agregarLineaCentro(he,centro)*. La funcionalidad que posee es, dado un HalfEdge y un punto que es el centro del triángulo de la iteración en la que estamos, se procede a calcular el centro del triángulo al que pertenece el vértice adyacente al HalfEdge. Entonces creamos la línea entre los dos centros de las circunferencias que forman los dos triángulos (el de la iteración y el que forma un vértice del mismo). Esto se realiza para los tres triángulos adyacentes al triángulo de Delaunay dado, por eso se llama a la función con *he,he->getNext(),etc.*

Por simplicidad no se muestra lo relacionado con la interfaz en ninguno de los pseudocódigos.

Complejidad

Para el análisis de complejidad en Delaunay debemos analizar las operaciones más costosas, las cuales son encontrar el triángulo donde cae el vértice a insertar y legalizar la arista. El resto de operaciones y chequeos gracias a la estructura HalfEdge se realizan en tiempo constante.

Tomemos los dos casos entonces:

1. Encontrar el triángulo en el que cae v : para obtener el triángulo en el que cae no tenemos otra opción que recorrer todos los triángulos que existen en la triangulación. Siendo $n-2$ la cantidad de triángulos en una triangulación tenemos que la complejidad temporal de esta operación es $O(n)$ siendo n la cantidad de vértices.
2. Para la legalización se debe hacer un análisis más complejo, se realizará un análisis inverso. Suponemos que p fue el último vértice insertado, ¿Cuánto trabajo requirió insertar p ? Al insertar p tenemos la creación de 3 aristas incidentes a p y de tener que hacer un flip se elimina una arista y se crea otra. Estos son los únicos cambios que el algoritmo realiza, entonces al tener como máximo $6n$ aristas (según la fórmula de Euler[4]) obtenemos que la complejidad es $O(n)$.

Para concluir el análisis, siendo que por cada vez que encontramos un triángulo debemos legalizarlo, tenemos que el algoritmo de triangulación de Delaunay propuesto posee una complejidad temporal $O(n^2)$.

Para el algoritmo de Voronoi la complejidad calculada que surge es la misma que Delaunay, ya que primero se debe calcular dicha triangulación y es la operación más costosa del algoritmo (la otra operación del algoritmo es recorrer todos los triángulos).

4. Uso de la aplicación

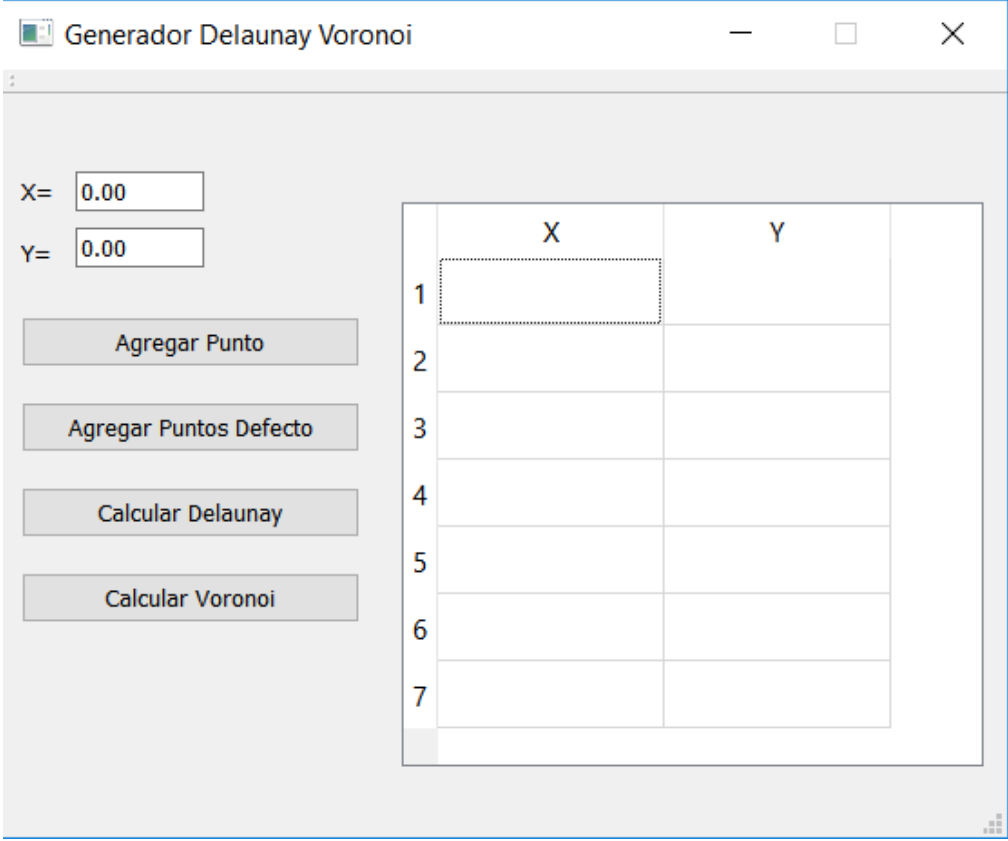
El uso de la aplicación es bastante intuitivo, como se observa en la Figura 4 la pantalla principal nos provee de varias opciones las cuales serán descritas a continuación:

Agregar Punto: agrega un nuevo punto para luego poder realizar la triangulación. Si el punto que se desea ingresar ya está contenido en la tabla, no se ingresa. Los valores a ingresar son reales y deben ser mayor a 0.0 y menor a 10.00.

Agregar Puntos Defecto: agrega catorce puntos por defecto para no tener que ingresar tanta cantidad de puntos manualmente. Luego de ejecutar esta función, se puede continuar agregando puntos normalmente.

Calcular Delaunay: procede a realizar la triangulación con los puntos que se observan dentro de la tabla y cuando finaliza lanza un cuadro de diálogo con la solución correspondiente.

Calcular Voronoi: luego de obtener la triangulación de Delaunay, realiza el cálculo de Voronoi. Al igual que el ítem anterior muestra un cuadro de diálogo en el que se representa el diagrama correspondiente, y cabe aclarar que detrás está el diálogo de Delaunay. Se seteo la transparencia del diálogo de Voronoi a un 93% para poder observar la relación que se da entre ellos, mencionada en la introducción.



	X	Y
1		
2		
3		
4		
5		
6		
7		

Figura 6.

5. Conclusión

La realización del trabajo requirió mucha investigación ya que se debía trabajar con estructuras de datos geométricas para lograr una buena complejidad. Leyendo y buscando varias fuentes se pudo lograr manejar muy bien la nueva estructura y poder sacar provecho de todas las ventajas que esta poseía frente a las convencionales.

En cuanto a la complejidad temporal obtenida, resultó ser un algoritmo relativamente eficiente, pudiéndose mejorar ya que la mejor implementación conocida pertenece a $O(n \log n)$. Esta complejidad se obtiene mediante el método de divide y conquista [6] o realizando una optimización a la hora de buscar el triángulo a insertar en nuestro caso.[5]

Por el lado de la interfaz, es la primera vez que me enfrentaba a realizar una ya que en la carrera no se suele enseñar y la verdad es que quede muy satisfecho de haber podido representar (aunque de una forma sencilla) muy bien los dos algoritmos y sus similitudes.

6. Bibliografía

- [1]Mark de Berg, Otfried Cheong, Marc van Kreveld and Mark Overmars.
Computational Geometry:Algorithms and Applications.Springer, 3rd edition (April 16, 2008).
- [2]Guibas, Lenoidas; Stolfi, Jorge (1985-04-01). "Primitives for the manipulation of general subdivisions and the computation of Voronoi".
- [3]Berg, Mark; Otfried Cheong, Marc van Kreveld, Mark Overmars (2008).
Computational Geometry: Algorithms and Applications. Springer-Verlag.
- [4]Antonio García Martín,Manuel Rosique Campoy,Francisco E. Segado Vázquez;
Topografía básica para ingenieros; 1994.
- [5] G. Leach: Improving Worst-Case Optimal Delaunay Triangulation Algorithms.
June 1992.
- [6]A Comparison of Sequential Delaunay Triangulation Algorithms
<http://www.cs.berkeley.edu/~jrs/meshpapers/SuDrysedale.pdf>
- [7]The Halfedge Data Structure <https://www.openmesh.org/Daily-Builds/Doc/a00016.html>.