UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

IMPLEMENTACIÓN DE UNA BIBLIOTECA DE TRIANGULACIÓN DE POLÍGONOS BASADA EN EL ALGORITMO LEPP DELAUNAY

MEMORIA PARA OPTAR AL TÍTULO DE INGENIERO CIVIL EN COMPUTACIÓN

PEDRO DANIEL VALENZUELA SALVATIERRA

PROFESORA GUÍA:
MARÍA CECILIA RIVARA ZÚÑIGA.

MIEMBROS DE LA COMISIÓN:
NANCY HITSCHFELD KAHLER.
PATRICIO INOSTROZA FAJARDIN.

SANTIAGO DE CHILE
DICIEMBRE DE 2009

Resumen

Las mallas de triángulos son ampliamente utilizadas en aplicaciones científicas e ingeniería. Una triangulación de un conjunto de puntos puede construirse utilizando diversos algoritmos, pero usualmente se prefiere aquellos que además de ser eficientes en tiempo y espacio, entreguen triángulos cuyo menor ángulo se encuentre sobre cierta cota.

La triangulación de Delaunay de un conjunto de puntos maximiza el ángulo mínimo de todos los triángulos de la triangulación. Sin embargo, el ángulo mínimo en una triangulación de Delaunay puede ser menor que el valor requerido en una aplicación dada.

Existen métodos de refinamiento de triangulaciones basados en la inserción de nuevos puntos en la malla que incrementalmente mejoran el ángulo mínimo de una triangulación. De especial interés es la familia de métodos de refinamiento basados LEPP, que recorren los triángulos de una malla a través de las aristas más largas de los triángulos.

El uso de los métodos de refinamiento basados en LEPP-Delaunay en aplicaciones, requiere la implementación de estructuras de datos para la representación de la malla, primitivas geométricas, algoritmos de triangulación y algoritmos de manipulación de los datos. El costo de escribir una aplicación desde cero se eleva al considerar los requisitos anteriormente mencionados.

El presente informe describe la implementación de una biblioteca reusable y general que ofrece la funcionalidad de los métodos LEPP-Delaunay. De esta manera, es posible crear aplicaciones que utilicen dichos métodos sin la necesidad de invertir tiempo de desarrollo en los algoritmos y estructuras de datos asociadas. En otras palabras, el foco del desarrollo puede estar completamente en la aplicación de los métodos LEPP-Delaunay y no en sus detalles de implementación.

La biblioteca LEPP-Delaunay fue construida utilizando las estructuras de datos de OpenMesh, proyecto que ofrece una implementación extensible y flexible de la representación de una malla en base a la estructura de datos halfedge.

Para mostrar las capacidades de la biblioteca LEPP-Delaunay se implementó una herramienta gráfica para el análisis de mallas que hace uso de la funcionalidad provista por la biblioteca LEPP-Delaunay.

Índice general

1. Introducción **3**
1.1. Estructura del informe 6

2. La estructura de datos Halfedge **7**
2.1. Representación de la triangulación 7
2.2. La estructura de datos *halfedge* 7
2.3. Vértices, caras y aristas 8
2.4. Conectividad 8
2.4.1. Alcance de la biblioteca OpenMesh 10
2.5. Tipos de datos 10

3. Definiciones **11**
3.1. Definiciones 11
3.2. Primitivas geométricas 12
3.3. Triangulación Delaunay 14
3.4. LEPP 15
3.4.1. Arista terminal 15
3.4.2. Longest edge propagation path 15
3.4.3. Refinamiento de triangulaciones mediante el algoritmo Lepp Delaunay 16
3.5. Funciones geométricas disponibles 16

4. Triangulación de Delaunay **18**
4.1. Algoritmo incremental 18
4.1.1. Point location 20
4.2. Algoritmo basado en dividir para conquistar 20
4.2.1. Procedimiento de unión de dos triangulaciones Delaunay 22
4.2.2. Elección de vértices candidatos en el proceso de unión 24
4.3. Funcionalidad disponible 27
4.3.1. Algoritmos implementados 28

5. Triangulación de Delaunay restringida **31**
5.1. Inserción de aristas restringidas 32
5.1.1. Funcionalidad disponible 34
5.1.2. Triangulación de polígonos 35
5.1.3. Triangulación de polígonos con agujeros 35
5.1.4. Funciones para definición de polígonos 36

6. Refinamiento basado en LEPP **38**
6.1. LEPP 38
6.1.1. Selección del punto a insertar: Punto medio 39
6.1.2. Selección del punto a insertar: Centroide 39
6.2. Aplicación de ejemplo 40
6.3. Herramienta de refinamiento LEPP-Delaunay 42
6.4. Controlador 43
6.4.1. Interfaz del controlador 44
6.5. Capacidades 47

7. Validación **49**
7.1. Validación de los algoritmos geométricos 49
7.2. Validación del Controlador 51
7.3. Validación de la interfaz de usuario 51

8. Conclusiones **52**
8.1. Objetivo general 52
8.1.1. Objetivos específicos 53

Apéndice

A. Comandos disponibles **55**

Capítulo 1

Introducción

Las mallas de triángulos son ampliamente utilizadas en simulaciones de ingeniería, diseño asistido por computador (CAD), computación gráfica, métodos de elementos finitos para interpolación de superficies y visualizaciones científicas, entre otras aplicaciones.

La calidad de los triángulos en las mallas de triángulos es crucial pues se encuentra asociada a la precisión de los resultados de las aplicaciones mencionadas anteriormente. En general, un triángulo se considera de buena calidad si su menor ángulo se encuentra acotado por abajo con respecto a cierto valor deseable. En otras palabras, se trata de evitar el uso de triángulos con ángulos pequeños.

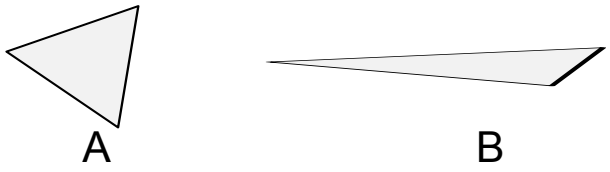

Figura 1.1: El menor ángulo del triángulo A es mayor que el menor ángulo del triángulo B.

En la figura 1.1 se muestran dos triángulos a modo de ejemplo. Se busca obtener triangulaciones con triángulos como A, cuyo menor ángulo es mayor que el ángulo del triángulo B.

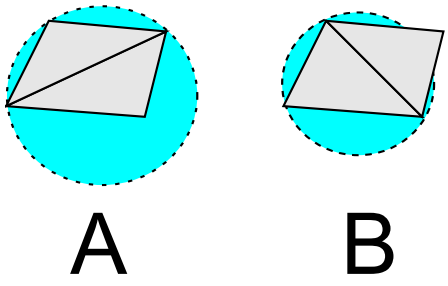

Figura 1.2: Dos triangulaciones. A es no Delaunay. B es Delaunay.

Una triangulación de Delaunay es un tipo especial de triangulación en que cada triángulo cumple con la propiedad de que el círculo definido por sus vértices no contiene a otros puntos de la triangulación. En la figura 1.2, la triangulación A no es Delaunay, mientras que la triangulación B sí lo es.

Dentro de las posibles triangulaciones de un conjunto de puntos, las triangulaciones de Delaunay son una alternativa interesante pues tienden a generar triángulos de buena calidad ya que maximizan el ángulo mínimo de todos los triángulos de la triangulación. Adicionalmente, el desempeño de las triangulaciones de Delaunay es aceptable en la práctica pues se cuenta con algoritmos con un costo en tiempo de $O(n \log n)$, con n el número de vértices a triangular.

Sin embargo, la calidad de los triángulos resultantes de una triangulación de Delaunay puede no ser suficientemente buena. Para resolver este problema se utilizan estrategias de refinamiento basadas en la inserción incremental de puntos en la malla. La idea principal de dichas estrategias consiste en que los nuevos puntos insertados en la malla ayuden a formar triángulos con ángulos cada vez mayores de manera que luego de realizar un cierto número de inserciones, todos los triángulos tengan su ángulo menor sobre cierta cota. De especial interés en este trabajo de memoria es la estrategia basada en el *Longest edge propagation path*, LEPP[9].

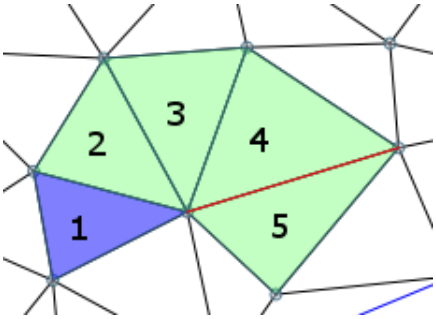

Figura 1.3: Ejemplo de LEPP de un triángulo dado.

El *Longest edge propagation path* de un triángulo t, corresponde al conjunto de triángulos resultante de recorrer la malla partiendo en t y avanzar en cada iteración al triángulo vecino al triángulo actual por la arista más larga. Este recorrido termina cuando la arista por la cual dos triángulos son vecinos es la más larga para ambos triángulos, o se ha alcanzado una arista de la envoltura convexa de la triangulación. En ambos casos, se obtiene como resultado una arista terminal.

La figura 1.3 muestra el LEPP de un triángulo en una triangulación. En dicha figura, los triángulos 4 y 5 son vecinos por la arista más larga en ambos triángulos por lo que el recorrido termina con el triángulo 5.

Las técnicas de refinamiento basadas en LEPP seleccionan a los triángulos cuyo menor ángulo se encuentre bajo una cota dada y calculan su LEPP. A partir de la arista terminal asociada al LEPP de cada triángulo, se obtiene un punto a ser insertado según diversos criterios. Los criterios de selección del punto a insertar son discutidos en este informe, en el capítulo dedicado al refinamiento basado en LEPP.

El caso de uso más común es contar con un número de puntos, construir su triangulación de Delaunay y luego aplicar el refinamiento basado en LEPP, procurando que luego de la inserción de nuevos puntos, la triangulación siga siendo Delaunay.

La utilización de triangulaciones de Delaunay en conjunto con la técnica de refinamiento LEPP, definen a los métodos LEPP-Delaunay. La implementación de los algoritmos asociados requiere contar con una representación de los datos de la malla de triángulos acorde a las necesidades de los métodos, así como la implementación de los algoritmos base que realizan

las triangulación de Delaunay y manipulan la malla según los métodos de refinamiento LEPP-Delaunay.

Figura 1.4: Jerarquía de módulos involucrados en LEPP-Delaunay.

Una vez disponibles las estructuras de datos y los algoritmos anteriormente mencionados, es posible construir aplicaciones que hagan uso de los métodos LEPP-Delaunay o permitan su estudio y extensión. La jerarquía de módulos de la figura 1.4 representa los distintos módulos involucrados en el proceso de implementación de los métodos LEPP-Delaunay.

Cada módulo en la jerarquía de la figura 1.4 puede ser brevemente descrito como,

- *Representación de la malla*: Estructuras de datos para representar triángulos, vértices y aristas de una malla de triángulos. También considera el almacenamiento en memoria de los datos.
- *Conectividad*: Rutinas para modificar y recorrer la malla de triángulos. Útiles para definir qué vértices forman un triángulo y recorrer los elementos de la malla según su conectividad (si son elementos vecinos) o de manera secuencial.
- *Algoritmos de triangulación y manipulación de la malla*: Este módulo considera los algoritmos que realizan triangulaciones Delaunay e implementan estrategias de refinamiento basadas en LEPP. De la misma manera se considera parte de este módulo la especificación de restricciones en la triangulación y el manejo de propiedades especiales de los elementos de la malla.
- *Aplicaciones*: Este módulo es el más general y corresponde a la sección que utiliza los métodos LEPP-Delaunay para algún propósito particular.

Típicamente el interés se encuentra al nivel de las aplicaciones pues usualmente van de la mano a problemas reales. El principal problema es que la tarea de diseñar e implementar los tres primeros niveles de la jerarquía es no trivial y representa un alto costo en el desarrollo.

El presente trabajo de memoria tiene como objetivo proveer de una implementación robusta y general de los algoritmos para la construcción de triangulaciones de Delaunay y los métodos de refinamiento de dichas triangulaciones, basados en el algoritmo LEPP-Delaunay[9].

La implementación considera la elección de una estructura de datos acorde al problema y la correspondiente capa de conectividad. El resultado de este trabajo de memoria es una biblioteca reusable e independiente capaz de ser incluida en diversas aplicaciones.

Adicionalmente, con el objetivo de proveer un ejemplo concreto de la utilización de la biblioteca que implementa las tres primeras capas de la jerarquía, se construyó una aplicación para el análisis de los métodos LEPP-Delaunay.

1.1. Estructura del informe

El presente informe describe la implementación de las capas de representación de la malla, manejo de conectividad y los algoritmos de triangulación, refinamiento y manipulación de la malla. La implementación de estas capas fue realizada en forma de biblioteca, utilizando el lenguaje de programación C++. Adicionalmente, se describe la implementación de una herramienta para el análisis de los métodos LEPP-Delaunay.

Al finalizar cada capítulo se especifica la funcionalidad ofrecida por la biblioteca implementada, relativa a los contenidos del capítulo. Por ejemplo, en el capítulo de definición de primitivas geométricas, se especifican las funciones disponibles que implementan dichas primitivas.

Los capítulos que siguen describen,

- La representación de los datos de la malla y las estructuras de datos relativas a su conectividad. En este capítulo se presenta la estructura de datos *halfedge* y la biblioteca OpenMesh. El uso de OpenMesh se restringe al almacenamiento de los datos e información de conectividad. La biblioteca OpenMesh fue escogida por su generalidad y por permitir el intercambio de datos con otras aplicaciones clientes de OpenMesh.

- Definiciones geométricas. Se describen las herramientas geométricas básicas para poder implementar los algoritmos de triangulación de Delaunay y refinamiento. En este capítulo se definen los conceptos de triangulación de de Delaunay y LEPP. Al finalizar el capítulo se describen las funciones geométricas básicas disponibles en la biblioteca implementada.

- Triangulación de Delaunay. Se describen los algoritmos implementados para construir triangulaciones de Delaunay y las funciones disponibles en la biblioteca para ejecutar los algoritmos sobre un conjunto de vértices.

- Triangulación de Delaunay restringida. Se definen los conceptos de triangulación de Delaunay restringida y su uso para triangular polígonos e insertar aristas restringidas. Se presentan las funciones geométricas necesarias para construir una triangulación de Delaunay restringida, y la funcionalidad disponible para definir y triangular polígonos e insertar aristas restringidas.

- Refinamiento basado en LEPP. Se describen las distintas estrategias de selección del punto a insertar y la manera en que dicha funcionalidad fue implementada en la biblioteca. Se describe la herramienta implementada para realizar refinamiento basado en LEPP-Delaunay, sus capacidades y los comandos aceptados por dicha herramienta.

- Validación. Se describe como fue probado el correcto funcionamiento de las funciones geométricas, los algoritmos de triangulación y refinamiento, y las limitaciones de la implementación.

- Conclusiones.

Capítulo 2

La estructura de datos Halfedge

2.1. Representación de la triangulación

La estructura de datos escogida para representar los elementos de la triangulación es el *halfedge*[2]. En esta estructura de datos cada arista es dividida en dos aristas dirigidas y opuestas, llamadas *halfedges*.

La implementación particular de dicha estructura de datos, utilizada en este trabajo de memoria, corresponde a la provista por la biblioteca OpenMesh[4]. La licencia de OpenMesh es la *GNU Lesser General Public License*, LGPL, versión 2.1.

Cada vez que se mencione el concepto de referencia, este corresponderá al concepto de *handle*, en el contexto de OpenMesh. Un *handle* es un objeto utilizado para hacer referencia a un elemento en particular almacenado por la biblioteca OpenMesh. Es así como existen *handles* que referencian vértices, aristas, caras (triángulos) y *halfedges*.

2.2. La estructura de datos *halfedge*

En cada *halfedge* se almacena[2],

- El vértice de `destino`, es decir, el vértice al que apunta el *halfedge*.
- La `cara` adjacente o una referencia inválida si el *halfedge* pertenece a la envoltura convexa del conjunto de puntos. La cara adjacente referenciada en el *halfedge* siempre se encuentra a la izquierda de éste debido a la orientación CCW de las caras.
- El `siguiente` *halfedge* de la cara en orden CCW.
- Su `inverso`, la otra mitad de la arista.
- El `anterior` *halfedge* si se busca optimizar el tiempo a cambio de uso de memoria. El uso de este miembro en la estructura de datos es opcional.

Todos los *halfedges* son orientados consistentemente en orden CCW alrededor de cada cara y a lo largo de la envoltura convexa del conjunto de puntos.

En la figura 2.1 se muestran los *halfedges* asociados a una malla de triángulos como aristas dirigidas. La orientación los *halfedges* dentro de una misma cara es en orden CCW.

La información necesaria para describir los elementos de una malla de polígonos se detalla a continuación.

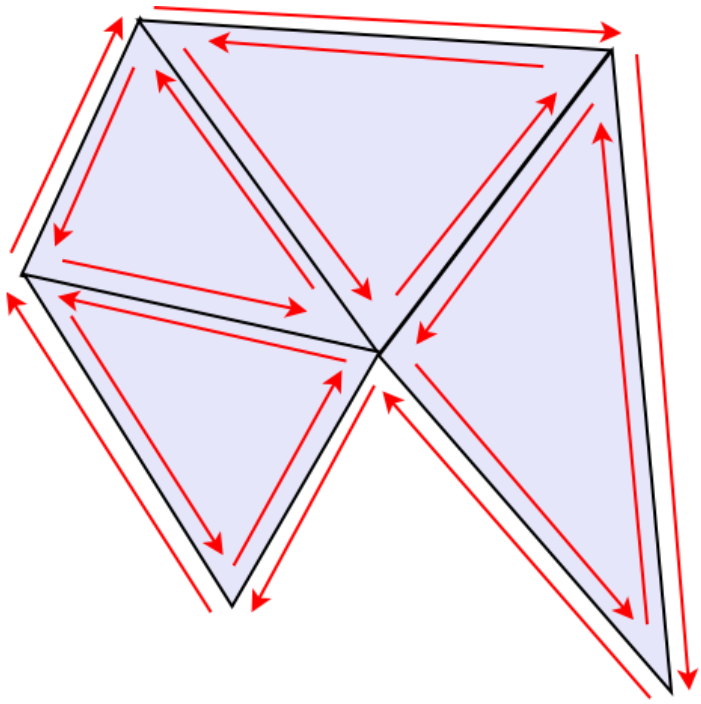

Figura 2.1: Halfedges de una malla de triángulos.

2.3. Vértices, caras y aristas

- *Vértice*: Contiene datos de coordenadas en el plano y una referencia a un *halfedge* saliente desde dicho vértice. Si el vértice se encuentra aislado, es decir, no es parte de alguna arista, entonces la referencia al *halfedge* correspondiente es inválida. Siempre que sea posible, el *halfedge* referenciado deberá corresponder a un *halfedge* de la envoltura convexa si el vértice se encuentra en la envoltura convexa del conjunto de puntos.
- *Cara*: Contiene una referencia a un *halfedge* arbitrario adyacente a la cara. Esto implica que si la cara es C, entonces se cumple que $C \rightarrow \texttt{halfedge} \rightarrow \texttt{cara} == C$.
- *Arista*: Contiene una referencia a alguno de los dos *halfedges* asociados a la arista.

2.4. Conectividad

El principal motivo para escoger la estructura de datos *halfedge* es que ofrece una manera eficiente de recorrer los elementos de una malla. En los algoritmos 2.1 y 2.2 se presentan ejemplos de recorrido de los elementos de la triangulación.

Como se puede apreciar, el recorrido en el algoritmo 2.1 es en forma de estrella. En cada iteración se visita un *halfedge* que sale del vértice v y que apunta a un vecino de v. Luego de obtener el vértice vecino, el paso para seguir visitando los vecinos de v consiste en moverse al siguiente *halfedge* saliente de v. Dada la conectividad, el siguiente *halfedge* saliente de v estará dado por el `siguiente` *halfedge* del `inverso` del *halfedge* actual.

El recorrido de las aristas de una cara es trivial. Dado que las aristas de una misma cara se encuentran conectadas según la orientación CCW de la cara, basta seguir la referencia al

Algoritmo 2.1: Vértices vecinos a un vértice dado.

Input: Vértice v
Halfedge inicio = v →halfedge;
Halfedge actual = inicio;
repeat
  Vertice vecino = actual → destino;
  actual = actual→inverso→siguiente;
until *actual == inicio* ;

Algoritmo 2.2: Aristas de una cara.

Input: Cara c
Halfedge inicio = c →halfedge;
Halfedge actual = inicio;
repeat
  actual = actual→siguiente;
until *actual == inicio* ;

halfedge `siguiente` de manera iterativa hasta que el recorrido vuelva a la arista de inicio.

De la misma manera, los vértices de una arista son,

- El vértice al que apunta el *halfedge* almacenado en la arista, es decir, si la arista es A, $A \rightarrow$ `halfedge` $\rightarrow$ `destino`.
- El vértice al que apunta el *halfedge* opuesto al almacenado en la arista, es decir, si la arista es A, $A \rightarrow$ `halfedge` $\rightarrow$ `inverso` $\rightarrow$ `destino`.

Diversas variantes de los recorridos pueden ser obtenidas a partir de los algoritmos presentados anteriormente. La biblioteca OpenMesh establece dos categorías principales de objetos para recorrer los elementos de la triangulación: *Circulators* e *Iterators*.

Un *Circulator* es un objeto que permite recorrer la vecindad de un elemento dado. Ejemplos de *Circulators* son aquellos utilizados para,

- Recorrer los vértices de una cara.
- Recorrer las aristas de una cara.
- Recorrer las caras vecinas a un vértice.
- Recorrer las aristas vecinas a un vértice.
- Recorrer los vértices vecinos a un vértice.

Un *Iterator* es un objeto que permite recorrer todos los elementos de la malla. Existen *Iterators* para,

- Recorrer todos los vértices.
- Recorrer todas las aristas (y también todos los *halfedges*).

- Recorrer todas las caras.

Toda esta funcionalidad es parte de OpenMesh, y se encuentra implementada de una manera muy similar a los iteradores de la *Standard Template Library* de C++.

Adicionalmente, cada vez que una cara es insertada o removida, la información de conectividad de todos los elementos asociados a la operación son actualizados automáticamente por la biblioteca. Esto facilitó la implementación de los algoritmos pues el foco no estuvo centrado en los detalles acerca de la manipulación de las estructuras de datos.

2.4.1. Alcance de la biblioteca OpenMesh

Es necesario notar que el uso de la biblioteca OpenMesh se restringe a la representación de los datos y el manejo de gran parte de la información de conectividad de la malla. Sin embargo, las operaciones geométricas, y los algoritmos de triangulación, inserción de restricciones, refinamiento y manipulación general de la malla no se encuentran disponibles en OpenMesh y representan gran parte del trabajo realizado en este trabajo de memoria.

Para completar algunas secciones de los algoritmos implementados, fue necesario extender la funcionalidad de OpenMesh. Las extensiones realizadas se encuentran descritas en este documento en las secciones donde dichas extensiones fueron necesarias.

2.5. Tipos de datos

En el resto de este informe se hace mención a los distintos elementos de la malla, en términos de la estructura de datos `Mesh`, que representa la malla de triángulos.

Las siguientes son las definiciones más importantes a considerar a la hora de utilizar la biblioteca implementada.

- `Scalar`: Tipo de datos escalar. En la biblioteca este tipo de datos corresponde a un número de punto flotante IEEE 754. Sin embargo, se recomienda encarecidamente el uso de la definición `Scalar`, en caso de que la definición de la biblioteca cambie en el futuro.
- `Point`: Representación de un punto. Si p es una variable del tipo `Point`, sus coordenadas son $p[0]$ y $p[1]$.
- `VertexHandle`: Objeto utilizado para referenciar un vértice en la malla. Es importante notar que un vértice, adicionalmente a estar asociado a un objeto del tipo `Point`, contiene información de conectividad.
- `FaceHandle`: Objeto utilizado para referenciar una cara en la malla. La mayor parte del tiempo, una cara corresponderá a un triángulo en la malla.
- `EdgeHandle`: Objeto utilizado para referenciar una arista en la malla.
- `HalfedgeHandle`: Objeto utilizado para referenciar un *halfedge*.

La definición de los tipos de datos enumerados anteriormente se encuentra en la cabecera `Geometry.h`.

Capítulo 3

Definiciones

3.1. Definiciones

Sea $\mathcal{P} = \{P_i = (x_i, y_i)\}_{i=1}^{N}$ un conjunto de N puntos, P_i, en el plano XY, y sea Ω la envoltura convexa de $\mathcal{P}$[6].

Definición 1. Un conjunto $\mathcal{T} = \{(\alpha_j, \beta_j, \gamma_j) : 1 \leq \alpha_j, \beta_j, \gamma_j \leq N\}$ formado por M 3-tuplas de enteros $(\alpha_j, \beta_j, \gamma_j)$ define una triangulación de $\mathcal{P}$ si cumple que,

a. Por cada $j = 1..M$, los puntos P_{α_j}, P_{β_j}, P_{γ_j} son no colineales y definen al triángulo T_j.

b. Cada triángulo es definido por exactamente 3 puntos en $\mathcal{P}$, los cuales son los vértices del triángulo.

c. La intersección del interior de dos triángulos T_j, T_k es vacía siempre que $j \neq k$.

d. La intersección de dos triángulos vecinos es una arista común o un vértice común.

e. La unión de todos los triángulos es la envoltura convexa de $\mathcal{P}$.

Ya en el caso de 4 puntos en el plano existen dos triangulaciones posibles por lo que inmediatamente surge la pregunta de cómo escoger la mejor triangulación, según algún criterio.

Criterios posibles para establecer si una triangulación es mejor que otra incluyen,

- *Criterio de la diagonal más corta*: La triangulación $\mathcal{T}_1$, de cuatro puntos, es *mejor* que la triangulación $\mathcal{T}_2$, si $d_1 < d_2$, donde d_i es la diagonal de la triangulación $\mathcal{T}_i$. Si bien este criterio es simple de implementar, no previene la existencia de triángulos largos y delgados, los cuales no son deseables en los métodos de elementos finitos para interpolación de superficies.

- *Criterio de mínimización del ángulo máximo*: Una triangulación $\mathcal{T}_1$ es *mejor* que otra triangulación $\mathcal{T}_2$ si $\alpha_1 < \alpha_2$, con $\alpha_i = \max\{\alpha(T_j) : T_j \in \mathcal{T}_i\}$, donde $\alpha(T_j)$ es el ángulo más grande en el triángulo T_j.

Sin embargo, el criterio de interés en este trabajo de memoria se encuentra en la siguiente definición.

Definición 2. La triangulación $\mathcal{T}_1$ es *mejor* que otra triangulación $\mathcal{T}_2$, según el criterio de *maximimización del ángulo mínimo*, si $\alpha_1 > \alpha_2$, con $\alpha_i = \text{mín}\{\alpha(T_j) : T_j \in \mathcal{T}_i\}$, donde $\alpha(T_j)$ es el ángulo más pequeño en el triángulo T_j.

El criterio de *maximimización del ángulo mínimo* es utilizado para construir triangulaciones Delaunay, definidas posteriormente en este documento.

3.2. Primitivas geométricas

Las primitivas geométricas detalladas a continuación son utilizadas en la definición de una triangulación Delaunay, así como en la implementación de los algoritmos para construir triangulaciones de Delaunay sobre puntos en el plano XY.

Orientación CCW

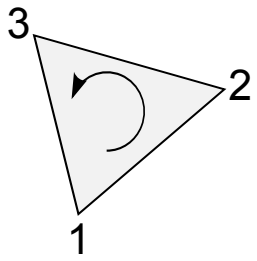

Figura 3.1: Orientación CCW.

Una de las operaciones base en las aplicaciones geométricas es verificar la orientación de 3 puntos. Para verificar si 3 puntos A, B y C en el plano XY se encuentran en orientación CCW[1], se debe calcular el producto cruz entre los vectores V_{AB}, el vector $\vec{AB}$, y V_{AC}, el vector $\vec{AC}$, tal como se aprecia en la figura 3.2.

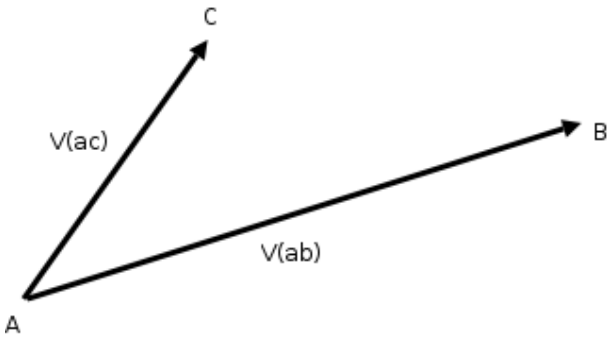

Figura 3.2: Vectores V_{AB} y V_{AC} utilizados en el test de orientación CCW.

[1] *Counter clockwise*(ver figura 3.1), en el sentido opuesto al movimiento de las agujas del reloj.

$$\begin{bmatrix} \hat{i} & \hat{j} \\ V_{AB_x} & V_{AB_y} \\ V_{AC_x} & V_{AC_y} \end{bmatrix} = (V_{AB_x}V_{AC_y} - V_{AB_y}V_{AC_x})\hat{k}$$
$$= ((B_x - A_x)(C_y - A_y) - (B_y - A_y)(C_x - A_x))\hat{k}$$

Por regla de la mano derecha, A, B y C se encuentran en la orientación CCW si,

$$(B_x - A_x)(C_y - A_y) - (B_y - A_y)(C_x - A_x) > 0$$

O equivalentemente,

$$\det \begin{bmatrix} (B_x - A_x) & (B_y - A_y) \\ (C_x - A_x) & (C_y - A_y) \end{bmatrix} > 0$$

inCircle

La principal primitiva geométrica a usar en la construcción de triangulaciones Delaunay es `inCircle`. Esta primitiva es aplicada a cuatro puntos distintos en el plano, A, B, C y D (figura 3.3), y su significado se encuentra en la siguiente definición.

Definición 3. La primitiva geométrica `inCircle`(A, B, C, D) es definida verdadera si y sólo si el punto D es interior a la región del plano delimitada por el círculo orientado ABC y se encuentra a la izquierda del mismo.

La definición anterior puede ser expresada como,

$$\det \begin{bmatrix} A_x & A_y & A_x^2 + A_y^2 & 1 \\ B_x & B_y & B_x^2 + B_y^2 & 1 \\ C_x & C_y & C_x^2 + C_y^2 & 1 \\ D_x & D_y & D_x^2 + D_y^2 & 1 \end{bmatrix} > 0$$

O equivalentemente,

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

Si trasladamos el origen de coordenadas al punto D.

Es importante notar que la primitiva `inCircle` considera la orientación de los puntos A, B y C. En las secciones siguientes, relativas a la definición de una triangulación Delaunay, dichos puntos cumplen `CCW`(A, B, C) pues corresponden a un triángulo en la misma orientación.

Lema 1. Si A, B, C y D son cuatro puntos no cocirculares en el plano, entonces transponer cualquier par adyacente en `inCircle`(A, B, C, D) cambia el valor de la primitiva de verdadera a falsa, o viceversa. En particular, la secuencia (`inCircle`(A, B, C, D), `inCircle`(B, C, D, A)) es (verdadera, falsa) o (falsa, verdadera).

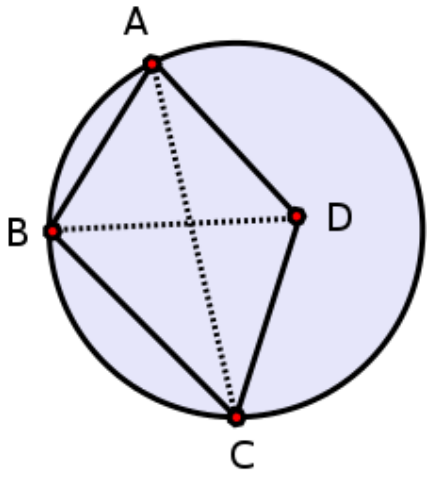

Figura 3.3: Test del círculo.

La demostración del lema anterior puede encontrarse en [5]. De la misma manera, si `inCircle`(A, B, C, D) es cierta, entonces `inCircle`(C, B, A, D) es falsa. Esto que implica que invertir la orientación del círculo cambia el valor de la primitiva. Sin embargo, la primitiva `inCircle` es siempre falsa si los cuatro puntos A, B, C y D son cocirculares, sin importar su orden.

3.3. Triangulación Delaunay

Definición 4. Una triangulación $\mathcal{T}$ de un conjunto de N puntos $\mathcal{P}$ es considerada Delaunay si se cumple *alguna* de las siguientes condiciones

a. La triangulación de los $N \geq 2$ puntos cumple que para cada arista, ab definida en $\mathcal{T}$, existe un círculo vacío. Es decir, existe un círculo que pasa por los extremos de la arista, a y b, de modo que los otros puntos en $\mathcal{P}$ son exteriores a dicho círculo.

b. La triangulación de los $N \geq 2$ puntos cumple que para cada triángulo, T definido en $\mathcal{T}$, el interior del círculo definido por los vértices de T no contiene puntos de $\mathcal{P}$.

Una triangulación de Delaunay maximiza el ángulo mínimo de todos los triángulos que la componen.

En particular, las aristas de la envoltura convexa de $\mathcal{P}$ son parte de la triangulación de Delaunay de $\mathcal{P}$. En el caso no degenerado en que no existe cuatro o más puntos cocirculares, la triangulación de Delaunay de un conjunto de puntos es única. En caso contrario, alternativas son aceptadas.

Los siguientes lemas, demostrados en [5] permiten caracterizar una triangulación de Delaunay utilizando las primitivas geométricas definidas anteriormente.

Lema 2. Sea $\mathcal{T}$ una triangulación arbitraria de $\mathcal{P}$, y sea XY una de sus aristas. Se dice que XY *pasa el test del círculo* si es la arista compartida entre dos triángulos AXY y YXB de $\mathcal{T}$ que cumplen `CCW`(A, X, Y), `CCW`(Y, X, B) y para los cuales `inCircle`(A, X, Y, B) es falsa.

Lema 3. Una triangulación $\mathcal{T}$ es Delaunay si y sólo si todas sus aristas pasan el test del círculo.

Diversos algoritmos existen para calcular la triangulación de Delaunay de un conjunto de puntos. Entre ellos, los dos algoritmos relevantes en este trabajo de memoria son,

- *Algoritmo incremental*: Este algoritmo comienza con un super triángulo o la triangulación de un rectángulo cuyo interior contiene todos los puntos de $\mathcal{P}$. En cada iteración, el algoritmo introduce un punto de $\mathcal{P}$ a la triangulación, primero buscando el triángulo que contiene a dicho punto[2], para luego dividir el triángulo que contiene al punto en tres. Los nuevos triángulos y sus vecindades son verificadas para garantizar que la triangulación resultante sea Delaunay.
- *Algoritmo basado en dividir para conquistar*: Este algoritmo divide recursivamente $\mathcal{P}$ en mitades, calcula la triangulación de Delaunay para ambas mitades y une el resultado sus respectivas triangulaciones. Los casos base ocurren cuando un conjunto posee 2 puntos y su triangulación de Delaunay es simplemente la arista que une ambos puntos, y cuando un conjunto posee 3 puntos y su triangulación es un único triángulo. La mayor parte del trabajo ocurre en la rutina para unir las triangulaciones de Delaunay resultantes de procesar cada mitad por separado.

3.4. LEPP

Si bien la triangulación de Delaunay maximiza el ángulo mínimo de los triángulos que la componen, en ciertos casos la distribución de los datos de entrada es tal que, aún utilizando una triangulación de Delaunay, el mínimo ángulo se encuentre bajo una cota deseable.

Una manera de resolver este problema es utilizar la técnica de refinamiento basada en *Longest edge propagation path*, LEPP.

3.4.1. Arista terminal

En una triangulación, una arista terminal, E, es la arista más larga de los dos triángulos que la comparten si E es una arista interior en la triangulación. Los triángulos que comparten la arista terminal E son llamados triángulos terminales.

En el caso del algoritmo de refinamiento LEPP Delaunay, se considera también como arista terminal E, a la arista mayor del triángulo t_n de LEPP(t), si E pertenece a la envoltura convexa de la triangulación.

3.4.2. Longest edge propagation path

Se define el *Longest edge propagation path* de un triángulo t_0 perteneciente a una triangulación, LEPP(t_0), como la lista ordenada de triángulos adyacentes, $t_0, t_1, ..., t_n$, en la que la arista más larga de cada triángulo, es mayor a la arista más larga del triángulo que lo precede[9].

Se cumplen las siguientes propiedades relativas al *Longest edge propagation path* en una triangulación[9],

[2]Problema conocido como *point location*

1. Para cualquier triángulo t, LEPP(t) es finito (el caso extremo es que LEPP(t) incluya a toda la triangulación).

2. Los triángulos $t_0, t_1, ..., t_n$, de LEPP(t_0), tienen su arista mayor en orden estrictamente creciente.

3. Para el triángulo t_n, perteneciente a LEPP(t_0), se cumple una de las siguientes afirmaciones,

   a) t_n posee su arista mayor en el borde y dicha arista es mayor que la arista más larga de t_{n-1}.

   b) La arista más larga de t_n es la misma que la arista más larga de t_{n-1}.

3.4.3. Refinamiento de triangulaciones mediante el algoritmo Lepp Delaunay

El algoritmo de refinamiento basado en LEPP busca *romper* los triángulos que posean ángulos menores a un cierto valor límite.

Para este motivo, el primer paso para ejecutar el refinamiento es encontrar un triángulo candidato, t, que posea un ángulo menor a una cota pre establecida. A continuación se debe encontrar la arista terminal, E, a partir de LEPP(t).

Una vez encontrada E, la arista terminal, existen dos criterios para el siguiente paso,

- Insertar el punto medio de E a la triangulación.
- Insertar el centroide del cuadrilátero definido por los triángulos adyacentes a E.

La inserción a realizar es la misma que en el algoritmo incremental para construir triangulaciones de Delaunay. Ambos criterios son explorados posteriormente en la descripción de la implementación de la estrategia de refinamiento basada en LEPP Delaunay.

Adicionalmente, LEPP(t) define un lugar en la triangulación donde otros criterios de selección del punto a insertar pueden ser usados. Esto último define una familia de métodos basados en LEPP, por lo que la implementación de este concepto debe ser lo suficientemente general como para permitir dichas extensiones.

Antes de exponer los detalles de cada algoritmo y su implementación, es necesario describir la estructura de datos utilizada para representar los elementos de la triangulación: vértices, aristas y caras. Adicionalmente se expondrá la manera en que la información de conectividad es manipulada y almacenada.

3.5. Funciones geométricas disponibles

Las siguientes funciones, que corresponden a las primitivas geométricas descritas en este capítulo, fueron implementadas en el espacio de nombres `Geometry`.

- `bool CCW(const Point& a, const Point& b, const Point& c)`
  Prueba si los puntos `a`, `b` y `c` se encuentran en orientación `CCW`.

- `bool left(const Point& a, const Point& b, const Point& c)`
  Prueba si el punto `c` se encuentra estrictamente a la izquierda de la arista dirigida `(a, c)`. Equivalente a `CCW`.

- `bool leftOn(const Point& a, const Point& b, const Point& c)`
  Prueba si el punto `c` se encuentra a la izquierda o en la arista dirigida `(a, c)`.

- `bool collinear(const Point& a, const Point& b, const Point& c)`
  Prueba si tres puntos son colineales.

- `bool intersectProp(Point& a, Point& b, Point& c, Point& d)`
  Prueba si la intersección propia entre el segmento `(a,b)` y el segmento `(c,d)`.

- `bool inCircle(Mesh& mesh, VertexHandle a, ... b, ... c, ... d)`
  Implementación de la primitiva `inCircle`.

- `bool faceContainsPoint(Mesh& mesh, FaceHandle& face, Point& point)`
  Prueba si un punto se encuentra dentro de una cara.

- `Scalar distanceSquared(Point& a, Point& b)`
  Retorna el cuadrado de la distancia entre dos puntos.

- `Scalar getMaxScalar()`
  Retorna el mayor escalar representable. Util para inicializar valores.

- `Scalar getMinScalar()`
  Retorna el menor escalar representable.

Para acceder a estas funciones es necesario incluir la cabecera `Geometry.h`. La mayoría de las funciones considera la definición de epsilon presente en dicha cabecera.

Una clase de utilidad en el desarrollo de aplicaciones es la definición del rectángulo que contiene a un conjunto de puntos. Se implementó esta funcionalidad en la clase `BoundingBox`, siendo sus operaciones más importantes,

- `void BoundingBox::update(const Point& point)`
  Considera el punto especificado en el cálculo de los límites del rectángulo.

- `const Point& BoundingBox::getLowerLeft()`
  Entrega la esquina inferior izquierda del rectángulo.

- `const Point& BoundingBox::getTopRight()`
  Entrega la esquina superior derecha del rectángulo.

- `Scalar BoundingBox::getWidth()`
  Entrega el ancho del rectángulo.

- `Scalar BoundingBox::getHeight()`
  Entrega el alto del rectángulo.

Capítulo 4

Triangulación de Delaunay

La implementación realizada se encuentra en forma de biblioteca, y es suficientemente general como para ser utilizada en otros proyectos. El lenguaje de programación C++ fue utilizado en la implementación de la biblioteca.

A continuación se presentan los detalles de implementación del algoritmo de triangulación de Delaunay utilizando la representación de la malla en base a la estructura de datos *halfedge*.

4.1. Algoritmo incremental

El algoritmo incremental para construir una triangulación de Delaunay de un conjunto de puntos $\mathcal{P}$, consiste en comenzar con un triángulo inicial, el cual contiene a todos los puntos de $\mathcal{P}$. A continuación, se inserta cada uno de los vértices en la triangulación de acuerdo al siguiente procedimiento[13],

Algoritmo 4.1: Inserción Delaunay.

```
Input: Vértice v, Triangulación T
Triángulo t_p = locate(v, T);
Triángulo[] nuevosTriángulos = split(t_p, v);
T.agregarTriángulos(nuevosTriángulos);
Pila triangulosPendientes;
triangulosPendientes.push(nuevosTriángulos);
while ! triangulosPendientes.pilaVacia() do
    Triángulo t_i = triangulosPendientes.pop();
    if ! esVálido(t_i, v) then
        Triángulo[] aVerificar = flip(t_i, v);
        triangulosPendientes.push(aVerificar);
    end
end
```

El procedimiento de inserción parte buscando el triángulo existente en la triangulación $\mathcal{T}$, que contiene al vértice a insertar v.

Una vez localizado el triángulo t_p, se aplica el procedimiento `split`, que divide el triángulo t_p en tres triángulos resultantes de agregar 3 nuevas aristas, desde los vértives de t_p a v. El procedimiento `split` puede apreciarse en la figura 4.1.

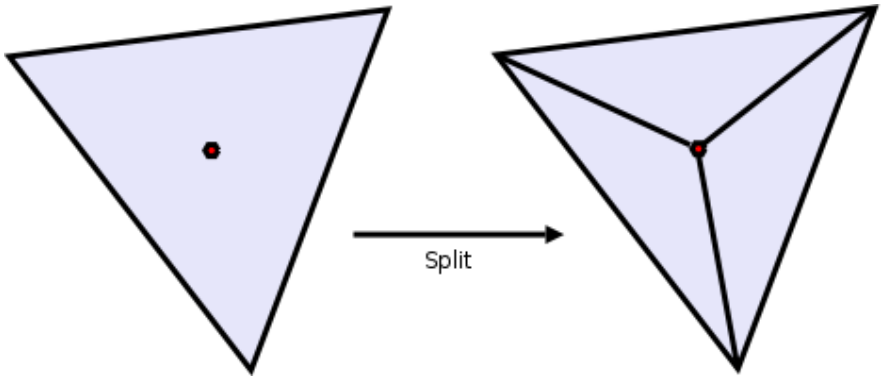

Figura 4.1: Operación split sobre un triángulo y un punto en su interior.

Los triángulos resultantes de realizar `split` sobre t_p deben ser verificados de manera que cumplan con el test del círculo. Si la triangulación $\mathcal{T}$ es Delaunay, entonces la verificación se reduce a comprobar que el vértice del triángulo adyacente al triángulo bajo verificación opuesto a la arista por la cual ambos triángulos son vecinos, no se encuentre dentro del círculo definido por los vértices del triángulo bajo verificación. La arista por la cual estos triángulos son vecinos es la opuesta al vértice insertado, v. En la figura 4.2 se aprecia la verificación local a realizar.

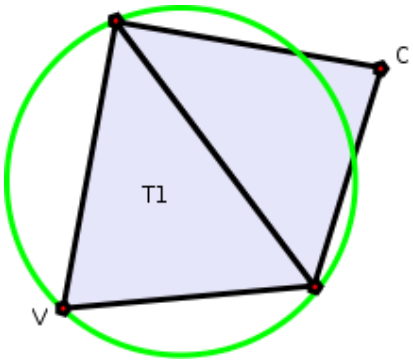

Figura 4.2: Verificación del triángulo $T1$.

Si alguno de estos triángulos no cumple con la verificación local, entonces se aplica la operación `flip` sobre la arista opuesta al vértice insertado. La operación `flip` se detalla en la figura 4.3 y consiste en intercambiar la arista opuesta al vértice V en el triángulo bajo verificación, por la arista VC.

Los nuevos triángulos resultantes de la operación `flip`, deben ser añadidos a la pila de triángulos por verificar. La inserción del vértice v termina cuando la pila de triángulos por verificar está vacía.

El algoritmo 4.1 posee varias simplificaciones con respecto a la función `locate`. En la implementación de la biblioteca, se considera los casos en que el vértice a insertar, v, se encuentra sobre una arista, e, de la triangulación T. En este caso, no se puede realizar la operación `split` sobre un único triángulo (el cual es un `split` de 1 a 3), sino que se debe realizar un `split` que involucre a los dos triángulos adyacentes a la arista e, introduciendo aristas desde v hacia los vértices opuestos a e en los triángulos adyacentes (`split` de 2 a

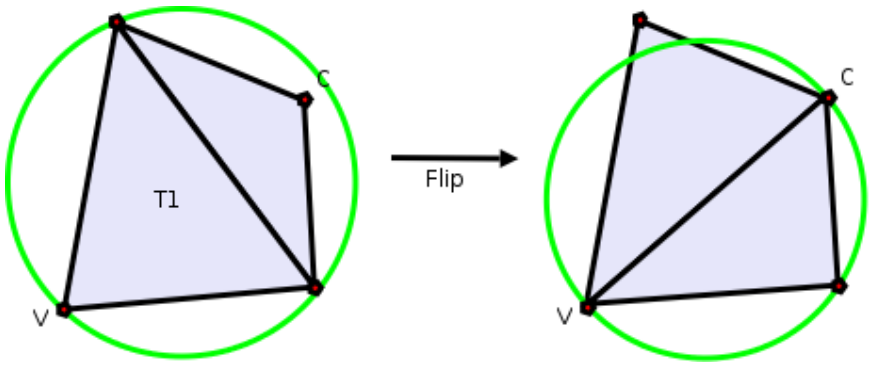

Figura 4.3: Operación flip con respecto al vértice V en el triángulo $T1$.

4)[13].

En el caso de que el vértice v corresponda a un vértice ya existente en la triangulación $\mathcal{T}$, la inserción termina de inmediato (nada se hace).

Un problema del algoritmo incremental es que es más lento que el algoritmo basado en dividir para conquistar, principalmente por el costo de encontrar el triángulo que contiene al punto a insertar (problema conocido como *point location*). Si bien en la actualidad existen diversos métodos que resuelven el problema de encontrar el triángulo que contiene al punto a insertar en tiempo competitivo, usualmente las implementaciones del algoritmo basado en dividir para conquistar, son más rápidas.

Una alternativa interesante en la aplicación del algoritmo LEPP Delaunay es el procesamiento del conjunto inicial de puntos utilizando el algoritmo basado en dividir para conquistar, para luego realizar el refinamiento de la triangulación con las facilidades provistas por el algoritmo incremental.

4.1.1. Point location

Para resolver el problema de *point location*, que encuentra el triángulo que contiene a un punto, p, dentro de una triangulación existente, se implementó el algoritmo clásico de movimiento hacia el punto[3].

Este algoritmo comienza en un triángulo, T_i, aleatorio dentro de la triangulación. En cada iteración el algoritmo se mueve en dirección a p, según p se encuentre a la izquierda de alguna de las caras de T_i. Si dos caras de T_i cumplen que p se encuentra a la izquierda, entonces se escoge una de ellas, aleatoriamente. Esta aplicación del test de orientación entrega una arista E de T_i.

El siguiente triángulo en el recorrido es el triángulo adyacente a T_i por la arista E. Este es el nuevo triángulo a considerar.

El procedimiento termina cuando p está en el interior de T_i, está sobre una de las aristas de T_i, o es uno de los vértices de T_i.

4.2. Algoritmo basado en dividir para conquistar

El algoritmo descrito en esta sección, es el algoritmo presentado por L. Guibas y J. Stolfi en [5].

El algoritmo basado en dividir para conquistar, construye de manera recursiva la triangulación de Delaunay de un conjunto de puntos, construyendo primero la triangulación de Delaunay de dos mitades del conjunto.

Para dividir el conjunto de puntos a triangular y generar instancias más pequeñas del problema, se ordena los puntos según su primera coordenada. Una vez ordenados los puntos, el conjunto es dividido en dos mitades según la primera coordenada de los puntos. Con cada una de las mitades se construye la triangulación de Delaunay correspondiente a los puntos de esa mitad.

Los casos base del algoritmo son dos, cuando el conjunto de puntos a triangular es de tamaño 2, y cuando el conjunto de puntos a triangular es de tamaño 3. En el primer caso, el algoritmo devuelve una arista que conecta a los dos puntos. En el segundo caso base, el algoritmo entrega un triángulo que tiene como vértices a los 3 puntos a triangular.

La versión general del algoritmo se presenta en el algoritmo 4.2. La entrada del algoritmo son los índices de los vértices que definen el íntervalo ordenado de vértices a triangular.

Algoritmo 4.2: Delaunay version dividir para conquistar.

```
Input: int low, int high
size = high − low + 1;
if size == 2 then                                   /* Caso base: Arista */
 | return addEdge(low, high);
else if size == 3 then                             /* Caso base: Triangulo */
 | return addFace(low, low + 1, high);

middle = low + size/2;                            /* Variables para recursion */
ldi = middle − 1;
rdi = middle;
divideAndConquerDelaunay(low, ldi);              /* Recursion mitad izquierda */
divideAndConquerDelaunay(rdi, high);              /* Recursion mitad derecha */
merge(low, ldi, rdi, high);                         /* Mezcla de las mitades */
```

Queda entonces resolver cómo realizar la unión de dos triangulaciones correspondientes a las dos mitades de un conjunto de puntos. Consideremos que la triangulación de la izquierda (según la primera coordenada de los puntos, X) es L, y la triangulación de la derecha es R.

Diremos que las aristas de los triángulos en L son del tipo $L - L$, es decir, parten de un vértice en L y terminan en un vértice en L. De la misma manera, las aristas de los triángulos en R, son del tipo $R - R$. Por razones obvias, el proceso de unión de L y R deberá introducir aristas que crucen desde L a R, es decir del tipo $L - R$, o viceversa.

El algoritmo presentado por L. Guibas y J. Stolfi[5], garantiza que en el procedimiento de unión de L y R, únicamente son insertadas aristas del tipo $L - R$ o $R - L$. Es decir, las aristas del tipo $L - L$ y $R - R$ pueden ser borradas en la unión de L y R, pero no agregadas.

En el algoritmo 4.3 se detallan los pasos del procedimiento de unión de las triangulaciones de L y R. La entrada de dicho procedimiento son los dos índices que definen a L (ldo y ldi), y los dos índices que definen a R (rdi y rdo). En el caso de la mitad izquierda, el índice menor es ldo y el índice mayor es ldi. En el caso de la mitad derecha, el índice menor es rdi y el índice mayor es rdo. Se puede pensar que los índices terminados en la letra i, son los internos

con respecto a el procedimiento de mezcla, mientras que los índices terminados en la letra o son los que delimitan el intervalo completo a ser entregado luego de la unión de L y R.

En la figura 4.4 se muestra un ejemplo de las mitades L, R y los índices ldo, ldi, rdi y rdo, antes de la unión de ambas mitades.

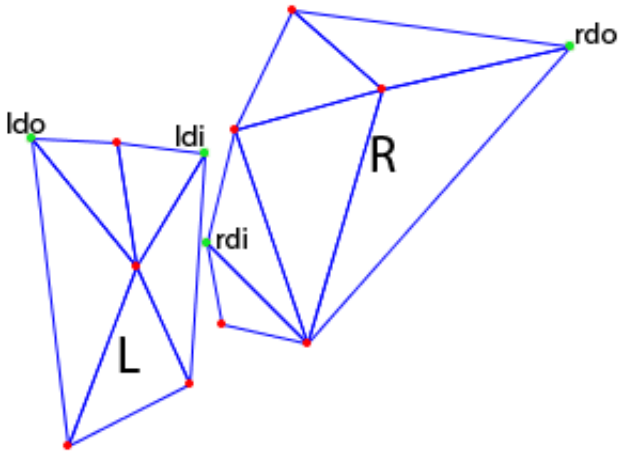

Figura 4.4: Ejemplo de las mitades L, R y sus respectivos índices.

4.2.1. Procedimiento de unión de dos triangulaciones Delaunay

En el algoritmo 4.3 se detalla el procedimiento de unión de dos triangulaciones de Delaunay, en el contexto del algoritmo 4.2. Las entradas de este procedimiento son los índices ldo, ldi, rdi y rdo, descritos en la sección anterior.

El procedimiento de unión comienza buscando la arista $L - R$ base desde la cual se irán añadiendo nuevos triángulos [5]. Para encontrar la arista $L - R$ base, se recorre la mitad derecha, R, partiendo desde rdi en sentido CCW y la mitad izquierda, L, partiendo desde ldi en sentido CW.

La determinación de la arista $L - R$ base hace uso de las funciones `right` y `left`, las cuales verifican si un punto se encuentra a la derecha o izquierda de una arista dirigida definida por otros dos puntos. Si los puntos que definen la arista dirigida son a y b, y el punto que se desea verificar es c, entonces la operación `left`(a, b, c) corresponde a `CCW`(a, b, c) y la operación `right`(a, b, c) es equivalente a `CCW`(a, c, b).

La determinación de la arista $L - R$ base nos entregará los índices de los dos vértices que la componen, $rBase$ y $lBase$, en R y L respectivamente. En la figura 4.5 se muestra la arista $L - R$ base y los vértices $rBase$ y $lBase$ en una triangulación de ejemplo.

El paso siguiente del algoritmo es insertar la arista $L - R$ base en la triangulación. En la implementación del algoritmo utilizando la biblioteca OpenMesh, fue necesario modificar dicha biblioteca para soportar la inserción de la arista $L - R$ base. Esto incluyó la creación

Algoritmo 4.3: Procedimiento `merge`.

```
Input: int ldo, int ldi, int rdi, int rdo

rBase = rdi;
lBase = ldi;
while true do                                   /* Encontrar arista L − R base */
    if right(rBase, nextCCW(rBase), lBase) then
        rBase = nextCCW(rBase);
    else if left(lBase, nextCW(lBase), rBase) then
        lBase = nextCW(lBase);
    else
        break;
end
addEdge(rBase, lBase);
while true do                                   /* Ciclo de mezcla de las mitades */
    lCand = getLeftCandidate(rBase, lBase);
    rCand = getRightCandidate(rBase, lBase);
    if !isValid(lCand) && !isValid(rCand) then
        break;
    if !isValid(lCand) ||
    (isValid(rCand) && inCircle(lBase, rBase, lCand, rCand)) then
        addFace(lBase, rBase, rCand);
        rBase = rCand;
    else
        addFace(lBase, rBase, lCand);
        lBase = lCand;
end
```

de los dos *halfedges* involucrados y el ajuste de la información de conectividad de todas las aristas, caras y vértices vecinos a la arista $L - R$ base.

Una vez insertada la arista $L - R$ base, comienza el ciclo de unión. El ciclo de unión busca vértices candidatos para conectar con la arista $L - R$ base, formar un nuevo triángulo, actualizar la arista $L - R$ base y dar paso a la siguiente iteración. Puede decirse, de manera informal, que la arista $L - R$ *sube* en cada iteración del ciclo de unión. Los detalles teóricos y la demostración formal de la correctitud del método pueden consultarse en [5].

Antes de discutir el proceso de selección de vértices candidatos en L y R, se revisará el proceso de inserción de nuevos triángulos. Dado un vértice candidato en R de índice $rCand$, y un vértice candidato en L de índice $lCand$, el algoritmo verifica la validez de los candidatos mediante la función `isValid`.

La verificación de validez realizada por el algoritmo considera como candidatos válidos a aquellos que se encuentran *arriba* de la arista $L - R$ base (recordemos que dicha arista *sube* en cada iteración del ciclo de unión). Esto equivale a comprobar `left`$(lBase, rBase, rCand)$ y `left`$(lBase, rBase, lCand)$ para el candidato de R y el candidato de L, respectivamente.

Por lo tanto, 3 situaciones pueden ocurrir en cada iteración del ciclo de unión,

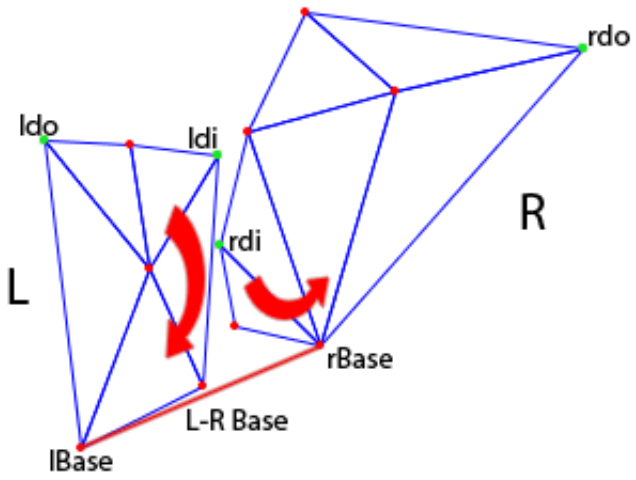

Figura 4.5: La arista $L - R$ base para la unión de L y R. Las flechas indican el sentido en que el algoritmo recorre cada mitad para determinar la arista $L - R$ base.

- Ambos candidatos no son válidos. Esto quiere decir que la arista $L - R$ base no puede seguir subiendo y por lo tanto, el ciclo de unión debe finalizar.
- Sólo uno de los candidatos es válido. En este caso, el triángulo definido por $lBase$, $rBase$ y el índice del candidato es insertado en la triangulación.
- Ambos candidatos son válidos. En este caso, se realiza el test del círculo vacío considerando los triángulos definidos por la arista $L - R$ base y ambos candidatos. Se escoge el candidato que cumpla con el test del círculo vacío. En la figura 4.6 se muestra esta situación. En dicho ejemplo, el vértice candidato escogido para la inserción, es $lCand$, pues el círculo definido por $lBase$, $rBase$ y $lCand$ no contiene a $rCand$.

Luego de la inserción del nuevo triángulo en la malla, se debe actualizar la arista $L - R$ base. Si el vértice candidato escogido para la inserción fue $lCand$, entonces $lBase$ tomará el valor de $lCand$ en la siguiente iteración del ciclo de unión. Análogamente, si el vértice seleccionado fue $rCand$, entonces $rBase$ tomará el valor de $rCand$ en la siguiente iteración del ciclo de unión.

La actualización de la arista $L-R$ base en el ejemplo de esta sección y luego de la primera iteración del ciclo de unión se puede apreciar en la figura 4.7.

La descripción de la elección de candidatos de L y R, es el tema de la siguiente sección.

4.2.2. Elección de vértices candidatos en el proceso de unión

El algoritmo 4.4 es utilizado para encontrar el vértice candidato en L. Se omite el algoritmo análogo para R, en donde cambia el sentido en el que se recorren los posibles candidatos a CW.

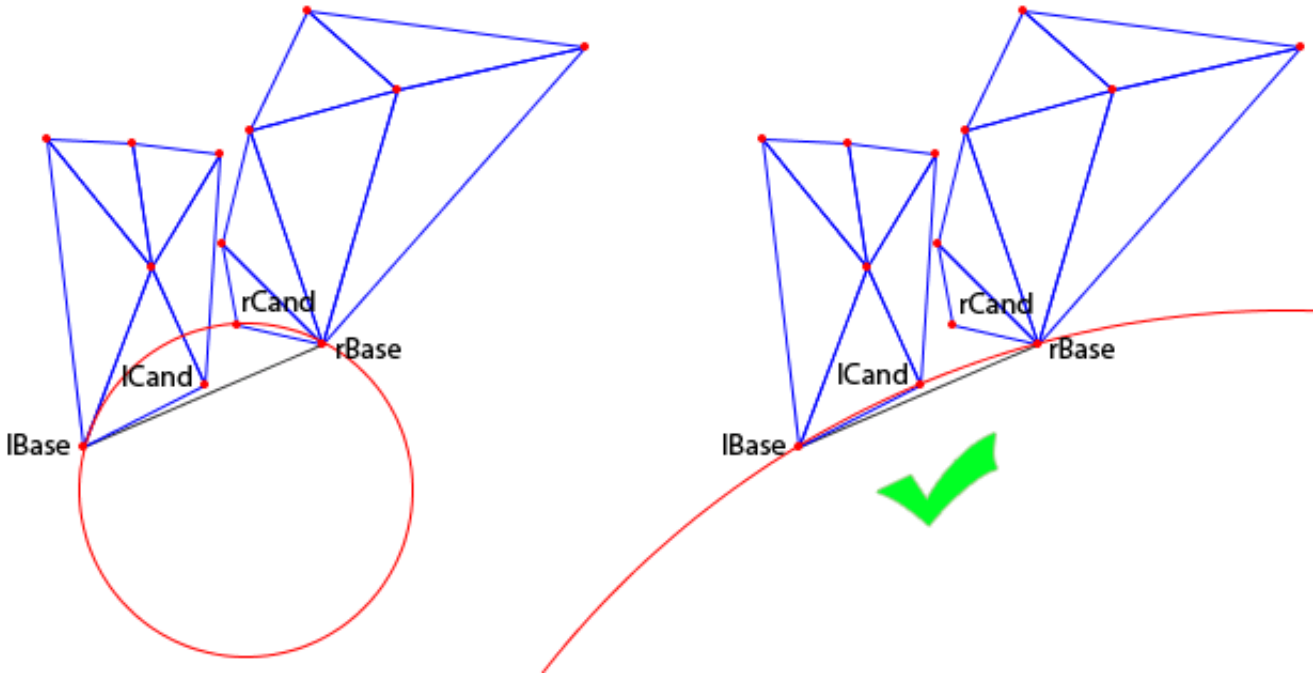

Figura 4.6: Los vértices candidatos $lCand$ y $rCand$. Se escoge $lCand$, pues el círculo que define junto a la arista $L - R$ base, no contiene a $rCand$.

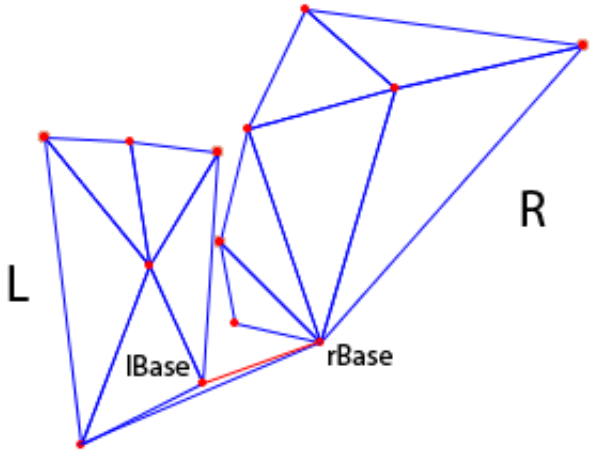

Figura 4.7: Actualización de la arista $L - R$ base luego de la utilizar al vértice candidato $lCand$, en la primera iteración del ciclo de unión.

En la figura 4.8 se muestra el orden en que los posibles candidatos de L son evaluados. El algoritmo verifica que el vértice candidato sea válido y que el círculo definido por el vértice

Algoritmo 4.4: Selección del vértice candidato en L.

Input: **int** $rBase$, **int** $lBase$

$lCand$ = `nextNeighbourCCW`($lBase$);
if *Valid*($lCand$) **then**
    $lNextCand$ = `nextCandidateCCW`($lCand$); **while** *InCircle*($lBase$, $rBase$, $lCand$, $lNextCand$) **do**
        `deleteEdge`($lBase$, $lCand$);
        $lCand = lNextCand$;
        $lNextCand$ = `nextCandidateCCW`($lNextCand$);
    end
return $lCand$;

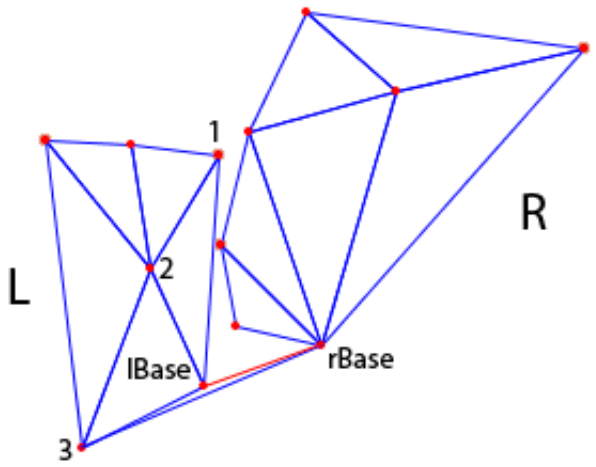

Figura 4.8: Orden en que los posibles candidatos de L son evaluados.

candidato, $lBase$ y $rBase$ no contenga al próximo candidato. Si la segunda condición no se cumple, la arista que une a $lBase$ y al vértice candidato es eliminada y el vértice candidato es descartado.

En la figura 4.9, el vértice candidato 1 no cumple con que el círculo que define junto a $lBase$ y $rBase$ no contenga al vértice candidato 2. Por lo tanto, la arista que une $lBase$ con el vértice candidato 1 es removida y el vértice candidato 1 es descartado.

Los procedimientos de selección de vértices candidatos en L y R pueden retornar vértices no válidos. Sin embargo esto es manejado sin problemas en el ciclo de mezcla.

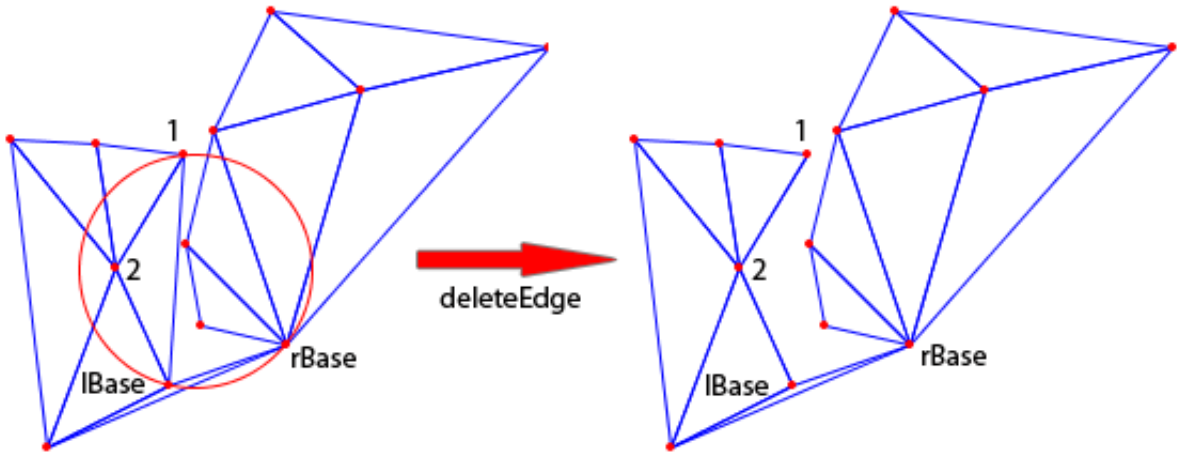

Figura 4.9: El círculo definido por el vértice candidato 1 y la arista $L - R$ base contiene al vértice candidato 2. Por lo tanto, el vértice candidato 1 es descartado.

4.3. Funcionalidad disponible

La clase principal implementada para encapsular el algoritmo de triangulación de Delaunay es `OpenMeshDelaunay`. Las operaciones principales sobre esta clase, relativas a este capítulo, son insertar vértices y luego realizar una triangulación de Delaunay utilizándolos como entrada. A continuación, los métodos relevantes de la clase `OpenMeshDelaunay`.

- `VertexHandle addVertex(Scalar x, Scalar y)`
  Añade un nuevo vértice a la malla y devuelve el `VertexHandle` asociado. Cualquier referencia posterior al vértice, debe realizarse utilizando dicho objeto. Este método no realiza operaciones de conectividad, por lo que el vértice se encuentra aislado del resto de los elementos de la malla.

- `template <typename TriangulationAlgorithm>`
  `void triangulate()`
  Realiza una triangulación utilizando el algoritmo especificado como parámetro, considerando todos los vértices de la malla. Se prefirió la definición del concepto `TriangulationAlgorithm` al uso de herencia, por razones de eficiencia.

- `template <typename TriangulationAlgorithm>`
  `void triangulate(vector<VertexHandle>& verticesToTriangulate)`
  Realiza una triangulación el algoritmo especificado como parámetro, considerando el subconjunto de vértices referenciados por el contenido del vector de `VertexHandle`.

Una vez realizada la triangulación, es posible realizar consultas y modificaciones a la malla. Los métodos disponibles en `OpenMeshDelaunay` para tales efectos son,

- `void locatePoint(const Point& point, FaceHandle& faceHandle)`
  Encuentra el triángulo que contiene al punto especificado. La implementación de este

método utiliza al objeto `PointLocator`, cuyo código fuente se encuentra en `PointLocator.h` y `PointLocator.cpp`. El parámetro `faceHandle` es un parámetro de salida.

- `VertexHandle delaunayInsert(const Point& point)`
  Realiza una inserción Delaunay del punto especificado y devuelve el `VertexHandle` asociado.

- `HalfedgeHandle getLongestEdge(FaceHandle faceHandle)`
  Entrega un `HalfedgeHandle` correspondiente a la arista más larga del triángulo especificado por `faceHandle`.

- `void lepp`
  `(FaceHandle face, vector<FaceHandle>& v, HalfedgeHandle& terminalEdge)`
  Calcula el LEPP del triángulo referenciado por `face`. El vector `v` es llenado con referencias a todos los triángulos que conforman LEPP(`face`). La arista terminal es almacenada en `terminalEdge`.

- `Mesh& getMesh()`
  Retorna una referencia a la representación de la malla.

4.3.1. Algoritmos implementados

Dos algoritmos fueron implementados. Las clases que definen ambos algoritmos son `DivideAndConquerAlgorithm`, para el algoritmo basado en dividir para conquistar, e `IncrementalAlgorithm` para el algoritmo incremental.

Ambos algoritmos cumplen con las características descritas en este capítulo. En el caso del algoritmo basado en dividir para conquistar, fue necesario extender OpenMesh para soportar la existencia de aristas sueltas, necesarias en el procedimiento de unión. Esto es requerido al momento de insertar la primera arista que une las triangulaciones L y R, pero que no se encuentra asociada a alguna cara.

Las extensiones realizadas comprenden,

- `HalfedgeHandle`
  `TriConnectivity::insert_crossing_edge`
  `(HalfedgeHandle _prev_heh, HalfedgeHandle _next_heh)`
  Inserta una arista que une las triangulaciones L y R en el procedimiento de unión. Esta es la arista que une a *lBase* con *rBase*.

- `void`
  `TriConnectivity::remove_edge`
  `(HalfedgeHandle heh0)`
  Borra una arista de la manera requerida en el procedimiento de unión, donde dicho evento depende de los vértices candidatos.

Con lo presentado anteriormente, es posible escribir un simple programa, a manera de ejemplo, mostrando las capacidades de la biblioteca hasta este punto.

```
#include <iostream>

#include <Delaunay/OpenMeshDelaunay.h>
#include <Delaunay/Util/Util.h>

typedef Mesh::ConstVertexIter CVertexIter;
typedef Mesh::ConstFaceIter   CFaceIter;

int main(int argc, char *argv[])
{
    OpenMeshDelaunay omDelaunay;

    omDelaunay.addVertex(61, 17);
    omDelaunay.addVertex(-23, -95);
    omDelaunay.addVertex(55, 60);
    omDelaunay.addVertex(-77, 4);
    omDelaunay.addVertex(57, -114);

    /* Triangular */
    omDelaunay.triangulate<DivideAndConquerAlgorithm>();

    Mesh& mesh = omDelaunay.getMesh();

    /* Escribir vértices */
    CVertexIter vEnd = mesh.vertices_end();

    for(CVertexIter vIter = mesh.vertices_begin(); vIter != vEnd; ++vIter)
        Util::dump(mesh, vIter.handle());

    /* Escribir triángulos */
    CFaceIter fEnd = mesh.faces_end();

    for(CFaceIter fIter = mesh.faces_begin(); fIter != fEnd; ++fIter)
        Util::dump(mesh, fIter.handle());

    return 0;
}
```

El programa anterior hace uso de las funciones utilitarias implementadas en `Util.h`. La salida de este programa es,

```
Vertex [0] -> (61, 17)
Vertex [1] -> (-23, -95)
Vertex [2] -> (55, 60)
Vertex [3] -> (-77, 4)
Vertex [4] -> (57, -114)
Face [ 3 0 2 ]
```

```
Face [ 1 4 0 ]
Face [ 1 0 3 ]
```

Capítulo 5

Triangulación de Delaunay restringida

Las aristas restringidas, en una triangulación, son un conjunto de aristas que forzosamente deben aparecer en la triangulación, no pudiendo ser removidas mediante operaciones como `flip`. En la implementación realizada, dos aristas restringidas no pueden intersectar.

Definición 5. Sea $\mathcal{T}$ una triangulación arbitraria de $\mathcal{P}$, y sea XY una de sus aristas. Se dice que XY *pasa el test restringido del círculo* si es la arista compartida entre dos triángulos AXY y YXB de $\mathcal{T}$ que cumplen $\texttt{CCW}(A, X, Y)$, $\texttt{CCW}(Y, X, B)$ y para los cuales $\texttt{inCircle}(A, X, Y, B)$ es falsa, o se tiene simultáneamente que $\texttt{inCircle}(A, X, Y, B)$ es verdadera, y XY es una arista restringida.

Definición 6. Una triangulación $\mathcal{T}$ es Delaunay restringida si y sólo si todas sus aristas pasan el test restringido del círculo.

Para construir una triangulación restringida, es necesario describir el procedimiento de inserción de una arista restringida.

Fue necesario extender la definición de una arista en la biblioteca OpenMesh para incluir la propiedad Booleana que indica si dicha arista es restringida o no. La extensión fue realizada utilizando la definición de una clase de *Traits* particulares para la aplicación, técnica común en programación genérica basada en *templates* de C++.

La especificación completa de *Traits* para la definición de la malla es,

```
struct LeppDelaunayTraits : public OpenMesh::DefaultTraits
{
    typedef OpenMesh::Vec2f  Point;

    VertexAttributes(OpenMesh::Attributes::Status);
    EdgeAttributes  (OpenMesh::Attributes::Status);
    FaceAttributes  (OpenMesh::Attributes::Status);

    EdgeTraits
    {
      private:
        bool  isConstrained_;
      public:
```

```
        EdgeT() : isConstrained_( false ) { }

        bool is_constrained() const      { return isConstrained_;  }
        void set_constrained(bool value) { isConstrained_ = value; }
    };
};

typedef OpenMesh::TriMesh_ArrayKernelT<LeppDelaunayTraits> Mesh;
```

5.1. Inserción de aristas restringidas

Si la arista restringida AB ya existe en la triangulación, el algoritmo únicamente cambia la propiedad que indica que AB es una arista restringida, y termina. En caso contrario, el algoritmo para insertar una arista restringida, AB, puede ser dividido en tres etapas[1]

1. Remover los triángulos $T_1, ..., T_k$, por los que cruza AB, generando una region sin triangular.
2. Añadir la arista AB al resultado. Dicha arista posee la propiedad de ser restringida.
3. Volver a triangular las regiones a la izquierda y a la derecha de AB, que fueron no trianguladas en el primer paso.

El primer paso, determinar los triángulos $T_1, ..., T_k$ es simple, pues el primer triángulo tiene como vértice a A, por lo que T_1 se puede encontrar mediante el recorrido de las caras asociadas a A y el test de intersección de segmentos.

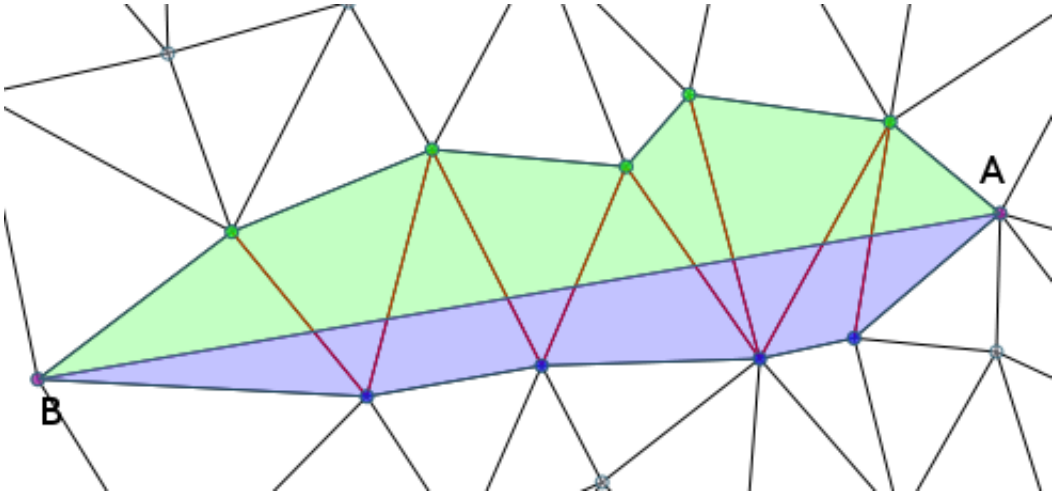

Figura 5.1: Inserción de una arista restringida.

Una vez determinado T_1, se realiza un recorrido similar al de *point location*, esta vez, en dirección a B. Una vez determinados $T_1, ..., T_k$, se clasifican los vértices de los triángulos a remover dependiendo si están a la izquierda o a la derecha de AB. A continuación se remueven los triangulos $T_1, ..., T_k$. Fue necesario extender OpenMesh para soportar esta operación en mallas basadas en triángulos, donde la cara resultante de remover $T_1, ..., T_k$, es un polígono.

La figura 5.1 muestra los triángulos $T_1, ..., T_k$. En dicha figura, las aristas en rojo son intersectadas por AB. Los vértices clasificados a la izquierda son mostrados en azul mientras que los vértices clasificados a la derecha son mostrados en verde. Luego de remover los triángulos $T_1, ..., T_k$, la arista AB es insertada y las regiones en azul y en verde deben ser trianguladas.

Para triangular los vértices a la izquierda, $i_1, ..., i_n$, se utiliza el siguiente procedimiento recursivo,

a. La serie de vértices $A, i_1, ..., i_n, B$ definen un pseudo polígono, en el cual se busca construir un triángulo formado por los vértices A, i_j y B. Dicho valor de j es encontrado para el triángulo cuyo circumcírculo no contiene a algún otro vértice i_k.

b. Una vez determinado i_j, se aplica recursivamente el procedimiento a $A, i_i, ..., i_j$ y a $i_j, ..., i_n, B$. El caso base está dado cuando el rango de vértices a analizar define únicamente un triángulo.

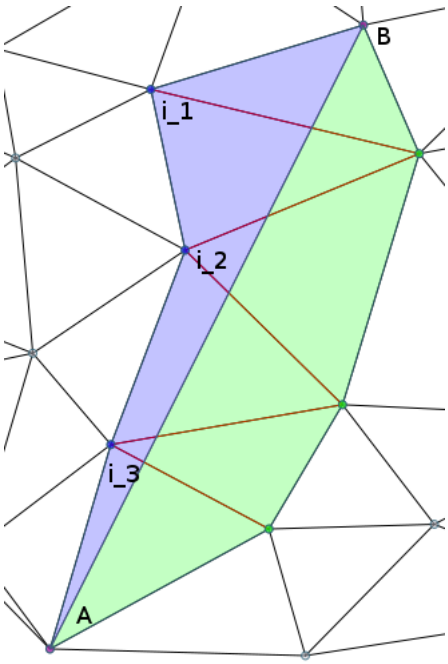

Figura 5.2: Triangulación del lado izquierdo.

En la figura 5.2, el resultado de aplicar el algoritmo corresponde a seleccionar i_3 pues su circumcírculo no incluye a i_1 ni a i_2. Por lo tanto, la arista que conecta B con i_3 es insertada, y el procedimiento recursivamente aplicado, esta vez considerando la arista i_3B y los vértices i_1, i_2.

Análogamente, se realiza el mismo procedimiento para los vértices a la derecha de AB, $d_1, ..., d_m$.

El resultado de finalizar la inserción de la arista restringida de la figura 5.2 se puede apreciar en la figura 5.3. Se debe notar que la existencia de aristas restringidas puede implicar la existencia de triángulos de mala calidad, tal como los resultantes en la figura 5.3, por lo que en dicho escenario las estrategias de refinamiento son muy importantes.

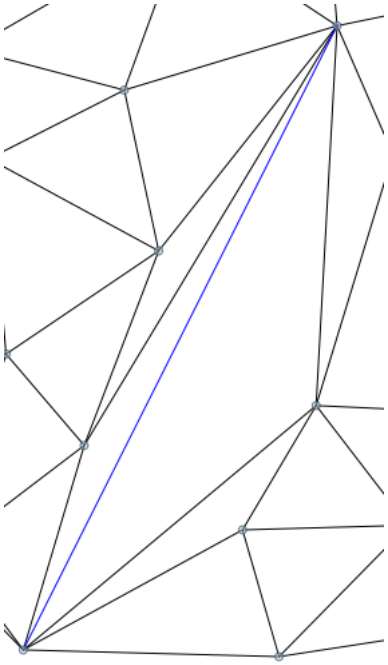

Figura 5.3: Resultado de la inserción de la arista restringida.

5.1.1. Funcionalidad disponible

La primera función a incorporar corresponde al test del círculo restringido, implementada en el espacio de nombres `Geometry`.

- `bool constrainedInCircle(Mesh& mesh, VertexHandle a, ... b, ... c, ... d)`
  Realiza el test del círculo restringido, considerando si la arista `(a,c)` es restringida o no.

La funcionalidad de especificar aristas restringidas fue añadida a la clase `OpenMeshDelaunay` en los siguientes métodos,

- `void prepareConstrainedEdgeInsertion(VertexHandle v1, VertexHandle v2)`
  Prepara la inserción de la arista restringida `(v1, v2)`.

- `void getConstrainedEdgeData(ConstrainedEdgeData &data)`
  Obtiene los datos asociados a la inserción de una arista restringida. Dichos datos incluyen las aristas intersectadas por la arista restringida y el conjunto de vértices a la izquierda y a la derecha de la arista. Un uso de esta información es generar interfaces como las mostradas en las figuras 5.2 y 5.3.

- `EdgeHandle executeConstrainedEdgeInsertion(VertexHandle v1, ... v2)`
  Ejecuta la inserción de una arista restringida. Se asume que la inserción de dicha arista fue preparada utilizando el método `prepareConstrainedEdgeInsertion`.

- `EdgeHandle insertConstrainedEdge(VertexHandle v1, ... v2)`
  Equivale a la ejecución de `prepareConstrainedEdgeInsertion` y `executeConstrainedEdgeInsertion`.

La existencia de dos flujos para inserción de aristas restringidas se justifica por la capacidad de obtener la información relativa a la inserción de la arista restringida, usando las tres primeras funciones (en ese mismo orden).

Si la información asociada a la arista restringida no es requerida, puede usarse directamente la función `insertConstrainedEdge`.

5.1.2. Triangulación de polígonos

Con la capacidad de insertar restricciones en la triangulación, es posible triangular polígonos, definiendo cada arista en el contorno del polígono, como una arista restringida en la triangulación. Esta operación es conocida como la definición del contorno exterior de $\mathcal{P}$.

La biblioteca implementada considera la definición del contorno exterior de $\mathcal{P}$ para ambos algoritmos. En el algoritmo incremental, la definición se realiza inmediatamente, como un preproceso del algoritmo. Es decir, para cada arista restringida A_iB_i en el contorno exterior,

a. El vértice A_i es insertado en la triangulación.

b. El vértice B_i es insertado en la triangulación.

c. La arista restringida A_iB_i es insertada en la triangulación.

La ventaja de que esto ocurra como un preproceso es que las restricciones son aplicadas cuando el número de triángulos en la triangulación es acotado, por lo que el número de triángulos cruzados por cada arista restringida a insertar es menor que si la inserción de cada arista restringida fuese ejecutada al finalizar la triangulación. En otras palabras, menos triangulaciones de los vértices a la derecha y a la izquierda de la arista restringida, ocurren.

En el caso del algoritmo basado en dividir para conquistar, la inserción de las aristas restringidas ocurre al finalizar la triangulación Delaunay sin restricciones. En este caso, la única operación a realizar a la inserción de las aristas restringidas A_iB_i que definen el contorno exterior de $\mathcal{P}$.

La figura 5.4 muestra un ejemplo de definición de polígono.

5.1.3. Triangulación de polígonos con agujeros

De la misma manera en que es posible definir el contorno exterior del polígono triangulado, también es posible definir agujeros. Los agujeros son expresados como un conjunto de vértices contenidos en el polígono triangulado, ordenados en el sentido CW.

Una vez definidos los vértices del agujero en CW, se insertan las aristas del agujero como aristas restringidas. Finalmente, se remueven las aristas y triángulos que se encuentren a la derecha de las aristas que definen al agujero.

La inserción de los agujeros ocurre como un preproceso en el algoritmo incremental, luego de la definición del contorno exterior del polígono.

En el caso del algoritmo basado en dividir para conquistar, la definición de los agujeros en el polígono, ocurre inmediatamente despues de la definición del contorno exterior del polígono, nuevamente, como un postproceso.

La figura 5.5 muestra un ejemplo de definición de polígono con agujeros.

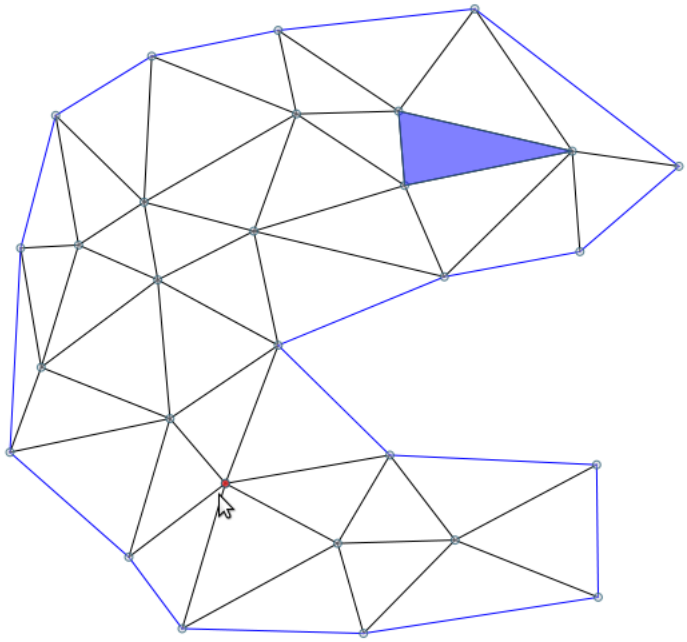

Figura 5.4: Ejemplo de definición de polígono.

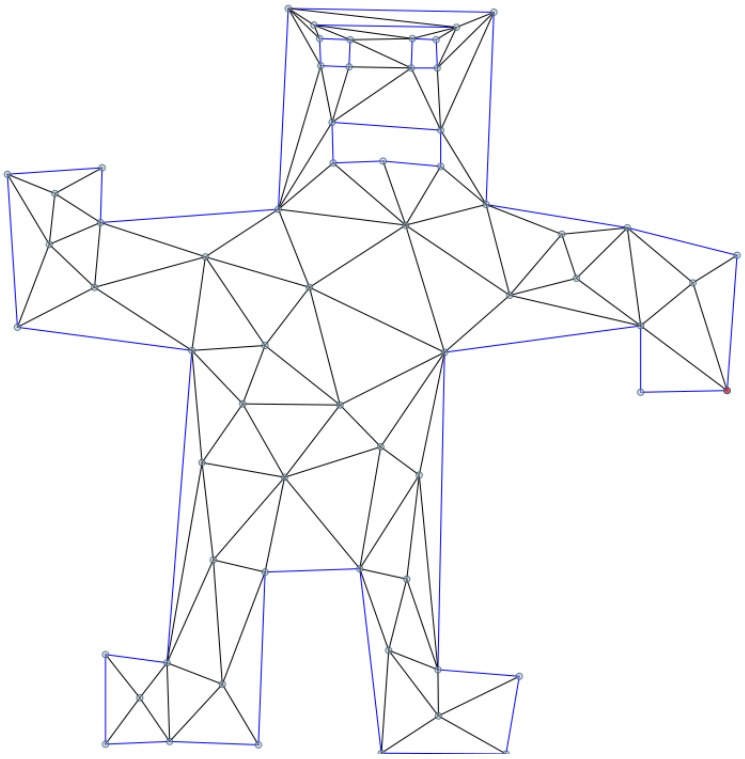

Figura 5.5: Ejemplo de definición de polígono con agujeros.

5.1.4. Funciones para definición de polígonos

La funcionalidad para definir polígonos y agujeros fue agregada a la clase `OpenMeshDelaunay` en los siguientes métodos,

- `void defineOuterContour(vector<VertexHandle>& ccwPolygon)`
  Define el polígono externo de un conjunto de puntos. Las referencias a los vértices que componen el polígono deben encontrarse en order `CCW`.

- `void addInnerContour(vector<VertexHandle>& cwPolygon)`
  Agrega un agujero en el conjunto de puntos. El agujero debe estar completamente contenido en el polígono externo, si este estuviese definido. El orden de las referencias a los vértices que componen el agujero debe ser el opuesto a `CCW`.

Capítulo 6

Refinamiento basado en LEPP

Obtener LEPP(t) es bastante simple utilizando la representación de la malla basada en *halfedges*. Las aristas de cada cara se pueden recorrer utilizando el algoritmo 2.2. Una vez determinada la arista más larga, dada por un cierto *halfedge*, h, la cara siguiente a examinar está dada por el *halfedge* opuesto a h. Si la cara asociada al *halfedge* opuesto a h es inválida, entonces se ha llegado al borde de la envoltura convexa de $\mathcal{P}$ o al contorno exterior del polígono $\mathcal{P}$.

6.1. LEPP

La biblioteca implementada permite construir una triangulación de Delaunay inicial utilizando el algoritmo basado en dividir para conquistar o el algoritmo incremental. También es posible definir el polígono exterior de $\mathcal{P}$, así como agujeros dentro del mismo.

Luego de obtener una triangulación de Delaunay restringida, es posible obtener un iterador de los triángulos de la triangulación, con el objetivo de seleccionar aquellos que posean ángulos menores a cierto ángulo mínimo. Las aplicaciones que utilicen la biblioteca no están limitadas a otros criterios de ordenamiento de los triángulos.

La biblioteca provee procedimientos para calcular el LEPP[12] de dichos triángulos, con la opción de solamente obtener la arista terminal, o la secuencia completa de triángulos de un cierto LEPP(T). La secuencia completa de triángulos de un cierto LEPP(T), es útil en el estudio de las propiedades de los métodos de refinamiento basados en LEPP-Delaunay[11].

En los métodos LEPP, usualmente se inserta el punto medio de la arista terminal e_t del LEPP[7], o el centroide del cuadrilátero definido por los triángulos adyacentes a la arista terminal e_t. En ambos casos, se conoce el triángulo que contiene al punto a insertar. La biblioteca provee funciones para la inserción de dichos puntos, ya sea ejecutando un `split` de 1 a 3, o un `split` de 1 a 4.

La rutina más importante en el cálculo de LEPP es `SelectPoint`, que selecciona el punto a insertar, relativo a la arista terminal. Se añadieron las siguientes funciones a la clase `OpenMeshDelaunay`,

- `bool constrained(HalfedgeHandle h)`
  Retorna si la arista es restringida o no. La referencia a una arista inválida se considera no restringida.

- `HalfedgeHandle second(FaceHandle f)`
  Retorna una referencia a la segunda arista más larga del triangulo referenciado por `f`.

- `Point midPoint(HalfedgeHandle h)`
  Retorna el punto medio de la arista referenciada por `h`.
- `Point centroid(Facehandle f1, FaceHandle f2)`
  Retorna el centroide del cuadrilátero formado por `f1` y `f2`.

6.1.1. Selección del punto a insertar: Punto medio

A continuación se presenta la implementación del algoritmo de selección del punto a insertar para el caso de inserción del punto medio[8].

```
void selectPoint(FaceHandle t1, FaceHandle t2, HalfedgeHandle l, Point& p)
{
  if( (!constrained(second(t1)) && !constrained(second(t2))) ||
        constrained(l))
  {
    P = midPoint(l);
    return;
  }
  else
  {
    if(t1 != InvalidFaceHandle && constrained(second(t1)))
    {
      p = midPoint(second(t1));
      return;
    }

    if(t2 != InvalidFaceHandle && constrained(second(t2)))
    {
      p = midPoint(second(t2));
      return;
    }
  }
}
```

6.1.2. Selección del punto a insertar: Centroide

A continuación se presenta la implementación del algoritmo de selección del punto a insertar para el caso de inserción del centroide[8].

```
void selectPointCentroid(FaceHandle t1, FaceHandle t2, HalfedgeHandle l, Point& p)
{
  if(constrained(l))
  {
    P = midPoint(l);
    return;
  }
  if(!constrained(second(t1)) && !constrained(second(t2)))
```

```
  {
    P = centroid(t1, t2);
    return;
  }
  else
  {
    if(t1 != InvalidFaceHandle && constrained(second(t1)))
    {
      SelectConstrainedQuad(t1, t2, l, second(t1));
      return;
    }

    if(t2 != InvalidFaceHandle && constrained(second(t2)))
    {
      SelectConstrainedQuad(t1, t2, l, second(t2));
      return;
    }
  }
}
```

Las anteriores son funciones privadas de la clase `OpenMeshDelaunay`. La función pública para la inserción LEPP-Delaunay es

```
void OpenMeshDelaunay::leppInsert(EdgeHandle edgeHandle, bool useCentroidStrategy);
```

La función `leppInsert()` realiza la selección del punto a insertar dependiendo del parámetro
`useCentroidStrategy`. Si dicho parámetro es verdadero, se utiliza la estrategia del centroide. Si el parámetro es falso, se utiliza la estrategia del punto medio. Una vez determinado el punto a insertar, se realiza una inserción Delaunay del mismo.

Una ventaja de separar el proceso de cálculo del LEPP de un triángulo, es decir, el cálculo del conjunto de los triángulos recorridos hasta la arista terminal, de la selección e inserción de un punto utilizando la arista terminal[10] es la capacidad de agregar nuevos criterios de selección de puntos a insertar. En particular, una futura extensión de la biblioteca podría comprender la mezcla de LEPP con la inserción de puntos basados en *off-centers*, descritos en [14].

Luego de la inserción, la biblioteca verifica que la triangulación siga siendo Delaunay restringida o de lo contrario aplica las operaciones `flip` necesarias para que así sea.

6.2. Aplicación de ejemplo

Con la funcionalidad de los algoritmos de triangulación y la capacidad de utilizar refinamiento LEPP, es posible escribir aplicaciones que ejecuten el refinamiento basado en LEPP-Delaunay. A continuación se presenta una aplicación de ejemplo que inserta vértices a la malla, construye una triangulación Delaunay y finalmente aplica el refinamiento LEPP-Delaunay hasta que el ángulo mínimo de la triangulación sea mayor que 32 grados.

```
#include <iostream>
#include <vector>

#include <Delaunay/OpenMeshDelaunay.h>
#include <Delaunay/Util/Util.h>

typedef Mesh::ConstFaceIter   CFaceIter;
typedef Mesh::FaceHandle      FaceHandle;
typedef Mesh::Scalar          Scalar;
typedef Mesh::HalfedgeHandle  HalfedgeHandle;

FaceHandle getBadTriangle(const Mesh& mesh, Scalar limit)
{
    for(CFaceIter fIter = mesh.faces_begin(); fIter != fEnd; ++fIter)
    {
        if(Util::triangleSmallestAngle(mesh, fIter.handle()) < limit)
            return fIter.handle();
    }

    return Mesh::InvalidFaceHandle;
}

int main(int argc, char *argv[])
{
    const Scalar SMALL_ANGLE = 32.0f;
    OpenMeshDelaunay omDelaunay;

    /* Agregar todos los vertices necesarios */
    omDelaunay.addVertex(...);
    ...

    /* Triangular */
    omDelaunay.triangulate<DivideAndConquerAlgorithm>();

    Mesh& mesh = omDelaunay.getMesh();


    /*
     * Refinar hasta que todos los triangulos tengan
     * menor angulo mayor que SMALL_ANGLE
     */
    std::vector<FaceHandle> leppTriangles;

    for(;;)
    {
        FaceHandle badTriangle = getBadTriangle(mesh, SMALL_ANGLE);
```

```
        if(badTriangle == Mesh::InvalidFaceHandle)
            break;

        HalfedgeHandle terminalEdge;

        /* Obtener arista terminal */
        omDelaunay.lepp(badTriangle, leppTriangles, terminalEdge);

        /* Insertar usando centroide */
        omDelaunay.leppInsert(terminalEdge, true);
    }

    /* Escribir triángulos */
    CFaceIter fEnd = mesh.faces_end();

    for(CFaceIter fIter = mesh.faces_begin(); fIter != fEnd; ++fIter)
        Util::dump(mesh, fIter.handle());

    return 0;
}
```

Es interesante notar que el proceso de refinamiento se reduce a unas pocas líneas, mientras que la triangulación inicial de los puntos es una simple llamada a la funcion `triangulate()`. Adicionalmente, el uso de iteradores para recorrer los elementos de la malla aumenta la legibilidad y mantenibilidad del código.

6.3. Herramienta de refinamiento LEPP-Delaunay

La biblioteca implementada considera primitivas geométricas, algoritmos de construcción de triangulaciones Delaunay, definición de polígonos y aristas restringidas, facilidades para el recorrido de los elementos de la malla, cálculo del LEPP de un triángulo e inserción de puntos utilizando las estrategias de selección de puntos de los métodos de refinamiento LEPP-Delaunay.

Para verificar el correcto funcionamiento de la biblioteca y sus componentes, se implementó una herramienta de refinamiento basada en LEPP-Delaunay. La herramienta ofrece la posibilidad de definir y editar mallas, así como construir triangulaciones, definir polígonos y refinar mallas de triángulos utilizando LEPP-Delaunay.

Como objetivo secundario, la herramienta de refinamiento y su interfaz de usuario ofrecen un ejemplo concreto de uso de la biblioteca.

La interfaz de usuario de la herramienta de refinamiento fue implementada utilizando la biblioteca QT, versión 4.5.

La implementación se basó en el patrón modelo-vista-controlador. En este caso particular, el modelo corresponde a la biblioteca para triangulaciones LEPP-Delaunay, la vista a una colección de *widgets* de QT programados para mostrar los resultados de las operaciones geométricas, y un controlador capaz de unir ambas interfaces.

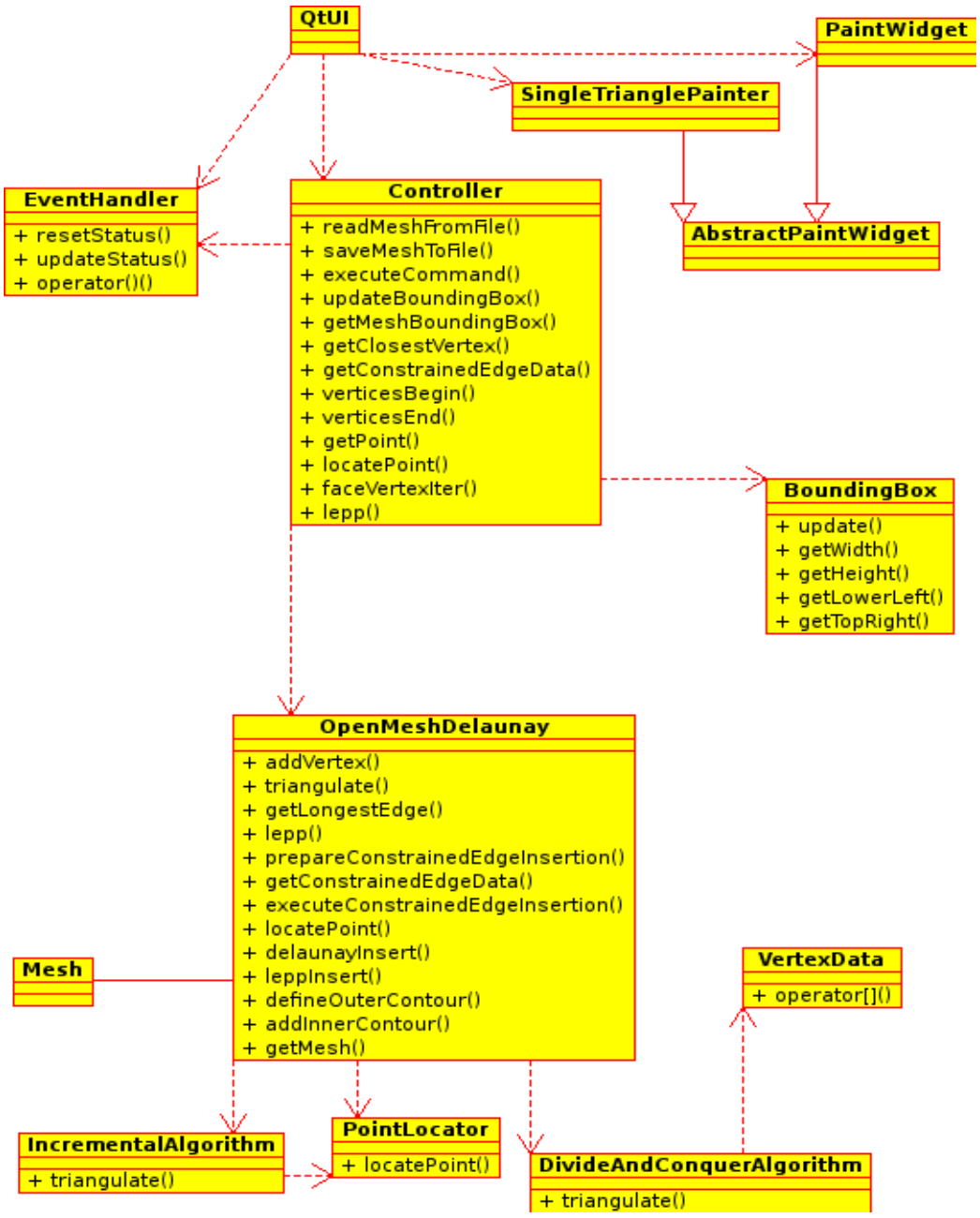

Figura 6.1: Clases principales de este trabajo de memoria.

El diagrama de clases simplificado, con las clases más importantes de este trabajo de memoria se encuentra en la figura 6.1. Para un diagrama de clases detallado, consúltese la documentación generada por *Doxygen* en la distribución del código fuente de la herramienta de refinamiento.

6.4. Controlador

El controlador es el objeto que une la implementación de todas las operaciones geométricas (el modelo), con la interfaz de usuario. Su tarea principal es la de recibir peticiones desde la

interfaz de usuario y ofrecer acceso a los elementos de la malla, encapsulando la representación misma de la malla de triángulos.

La principal ventaja del patrón modelo-vista-controlador es la posibilidad de cambiar una capa sin la necesidad de tocar las capas restantes. De esta manera, es posible escribir una interfaz de usuario completamente nueva, sin tener que intervenir el controlador ni los algoritmos geométricos.

La parte más interesante del controlador, es la inclusión de un *parser* de comandos. Los comandos actualmente disponibles permiten,

1. Añadir vértices.
2. Definir el contorno externo de un polígono.
3. Definir agujeros dentro de un polígono.
4. Insertar aristas restringidas.
5. Ejecutar triangulaciones, especificando el algoritmo a usar.
6. Refinar triangulaciones mediante LEPP-Delaunay.

Lo anterior establece una jerarquía de comandos, permitiendo guardar una sesión completa de la interfaz de usuario en archivos de texto con los comandos ejecutados en la sesión.

El parser utiliza *streams* de la biblioteca estándar de C++, por lo que su uso se puede extender a interfaces de texto o a procesamiento no atendido de archivos de comandos autogenerados.

Otro punto importante del controlador es la incorporación de *Functors* en las interfaces de comunicación. Esto permite desacoplar al Controlador de una biblioteca de interface de usuario particular como QT, abriendo la posibilidad de escribir una nueva interfaz utilizando algún otro *toolkit* como GTK.

6.4.1. Interfaz del controlador

El controlador provee una extensa variedad de métodos para acceder a los datos de la malla sin exponer la malla de triángulos a las capas superiores.

Las funciones más importantes de la clase `Controller` son las de carga de archivos, ejecución de comandos y guardado de archivos.

```
template<typename DisplayMessageFunctor, typename EventHandler>
bool readMeshFromFile(const char *fileName,
                      DisplayMessageFunctor& commandFunctor,
                      EventHandler& eventHandler);

bool saveMeshToFile(const char *fileName);

template<typename DisplayMessageFunctor, typename EventHandler>
bool executeCommand(const std::string& commandString,
                    DisplayMessageFunctor& commandFunctor,
                    EventHandler& eventHandler);
```

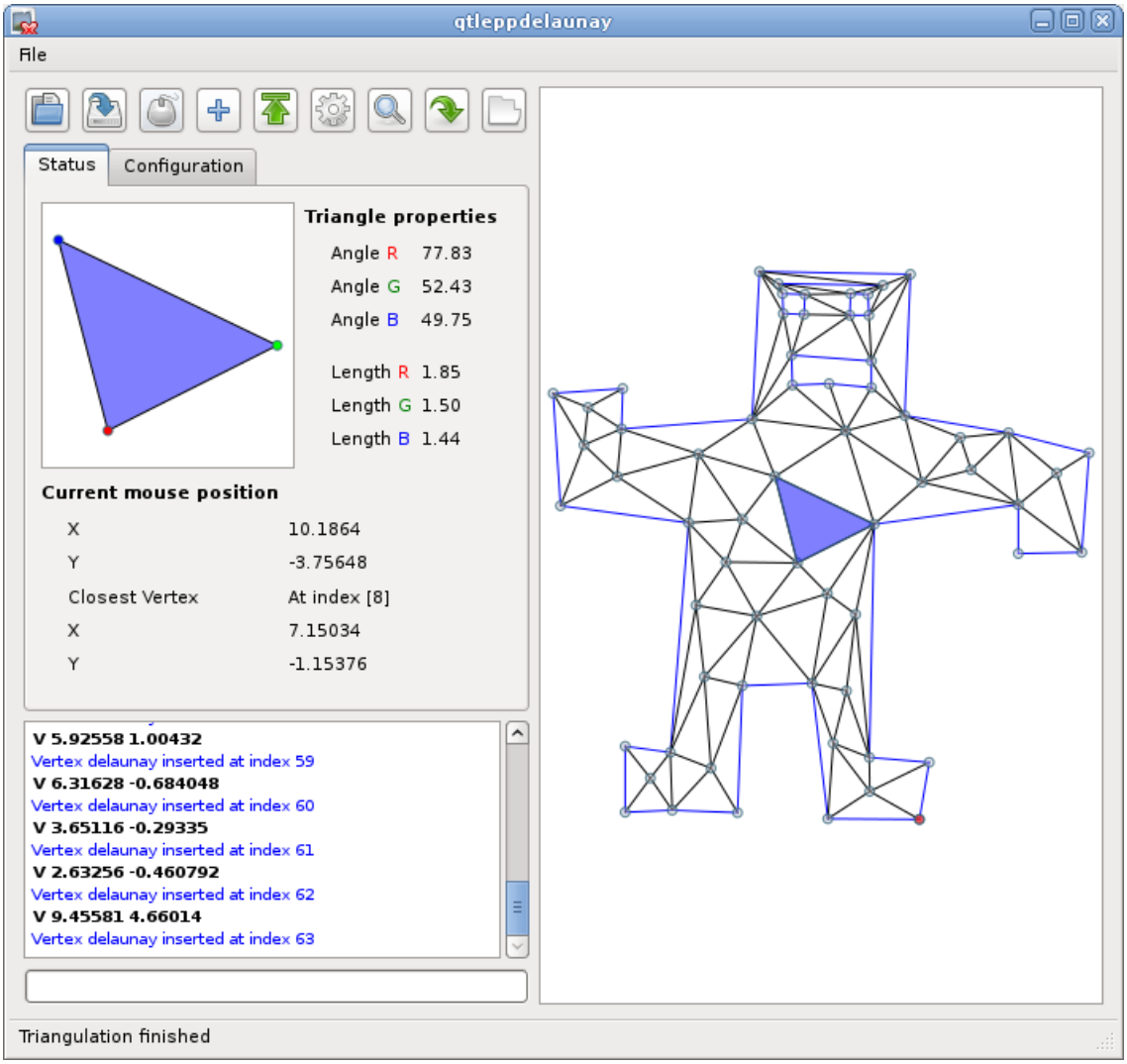

Figura 6.2: Interfaz de usuario: Propiedades de un triángulo.

Se definen dos tipos de conceptos para interactuar con el controlador. El primero es un *Functor* encargado de mostrar mensajes de parte del controlador cuando los comandos son ejecutados. Una clase que implemente dicho concepto debe proveer el operador().

```
void operator()(const char *message,
                const Controller::MessageType messageType);
```

Donde `Controller::MessageType` es una enumeración con valores posibles para mensajes de los tipos:

- `CommandLineMessage`: Usualmente para hacer eco de los comandos. En la interfaz de usuario estos mensajes son mostrados de color negro.
- `InformationMessage`: Mensajes informativos, como índices de elementos. En la interfaz de usuario estos mensajes son mostrados de color azul.

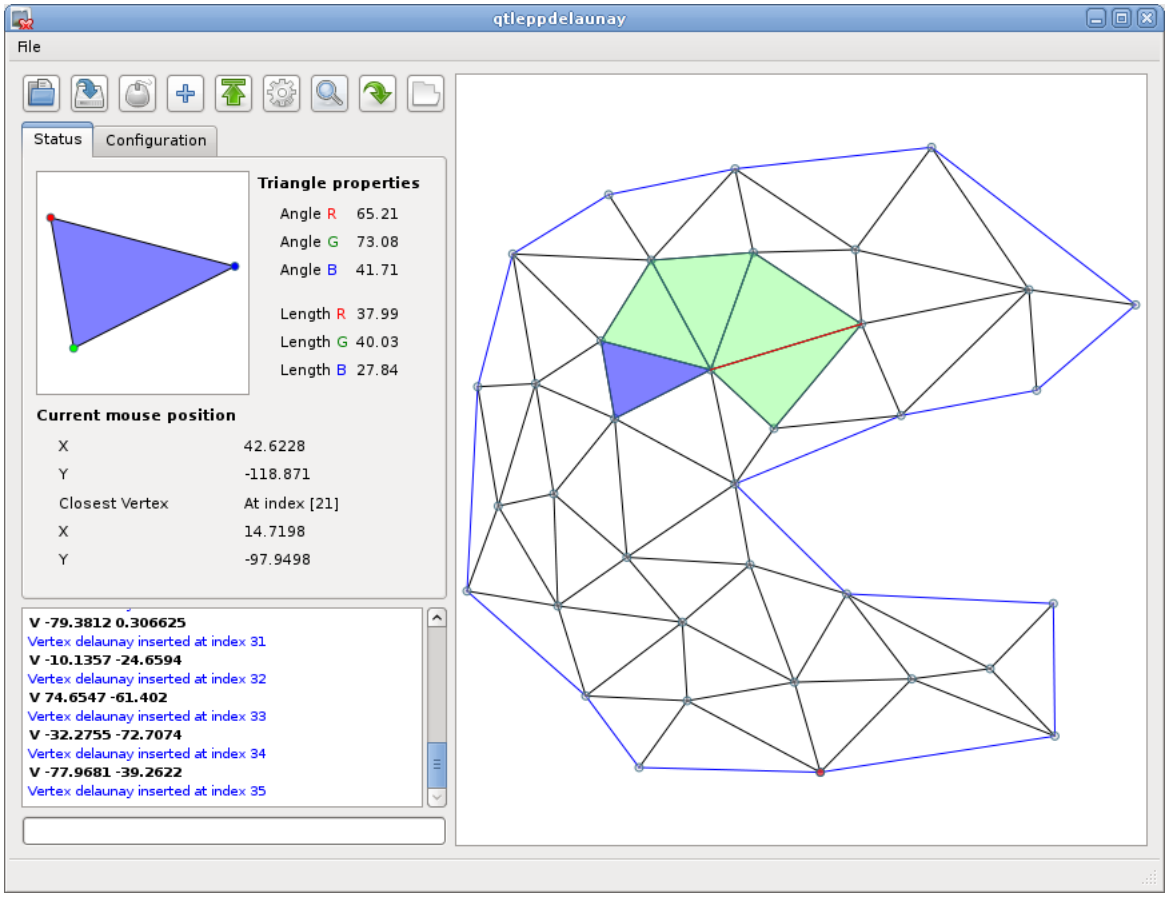

Figura 6.3: Interfaz de usuario: Visualización de LEPP.

- `ErrorMessage`: Mensajes de error. En la interfaz de usuario estos mensajes son mostrados de color rojo.

El segundo concepto corresponde a un manejador de eventos, el cual debe implementar las siguientes funciones,

- `void resetStatus()`
  Función que es llamada cuando la representación de la malla será reiniciada. La semántica de esta llamada corresponde a un aviso de que cualquier *handle* que otros objetos pudiesen tener guardados, dejarán de ser válidos. En la interfaz de usuario de la herramienta de refinamiento esta llamada envía una señal a todos los *Widgets* de QT que pudiesen tener variables de estado asociadas a la malla, pidiendo que dichas variables sean reiniciadas.

- `void updateStatus()`
  Función que es llamada cuando la malla cambia. La semántica de esta llamada corresponde a un aviso de que hay elementos nuevos (o distintos, por ejemplo a causa de un `flip` de arista), que pueden ser mostrados. En la interfaz de usuario de la herramienta de refinamiento esta llamada envía una señal a todos los *Widgets* de QT que muestran elementos de la malla para que invaliden su estado y dibujen nuevamente sus contenidos.

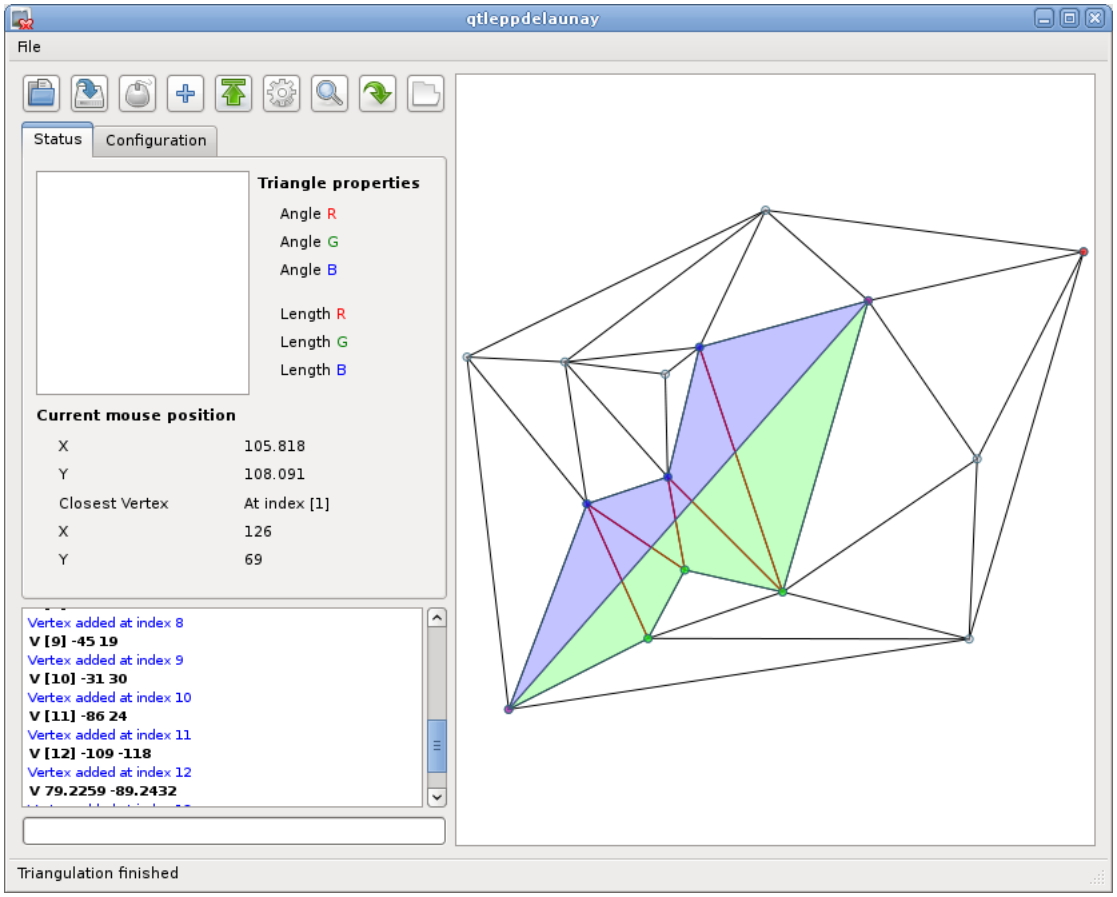

Figura 6.4: Interfaz de usuario: Inserción de arista restringida.

Si bien el uso de estos dos conceptos tiene una complejidad extra en los clientes de la clase `Controller`, su uso garantiza la separación de un *toolkit* particular como QT.

6.5. Capacidades

La interfaz de usuario de la herramienta de refinamiento permite la inserción de vértices, la definición del contorno exterior de un polígono, así como agujeros dentro del mismo de manera interactiva, pudiendo un usuario inspeccionar y manipular triángulos de la malla.

La interfaz de usuario incluye la capacidad de visualizar el círculo definido por cada triángulo, para así verificar el test del círculo, y la facilidad de mostrar las aristas restringidas dentro de la triangulación. La definición de aristas restringidas se puede realizar mediante comandos en la consola de la interfaz de usuario o interactivamente seleccionando pares de vértices.

En el caso de LEPP, la interfaz provee la visualización del LEPP de cualquier triángulo, así como la inserción de puntos en el punto medio de la arista terminal, o el centroide asociado a dicha arista.

Cada vez que una acción es ejecutada en la interfaz de usuario, el comando asociado al Controlador es mostrado en pantalla a manera de referencia. Los comandos enviados al Controlador pueden ser guardados y cargados en la interfaz de usuario posteriormente.

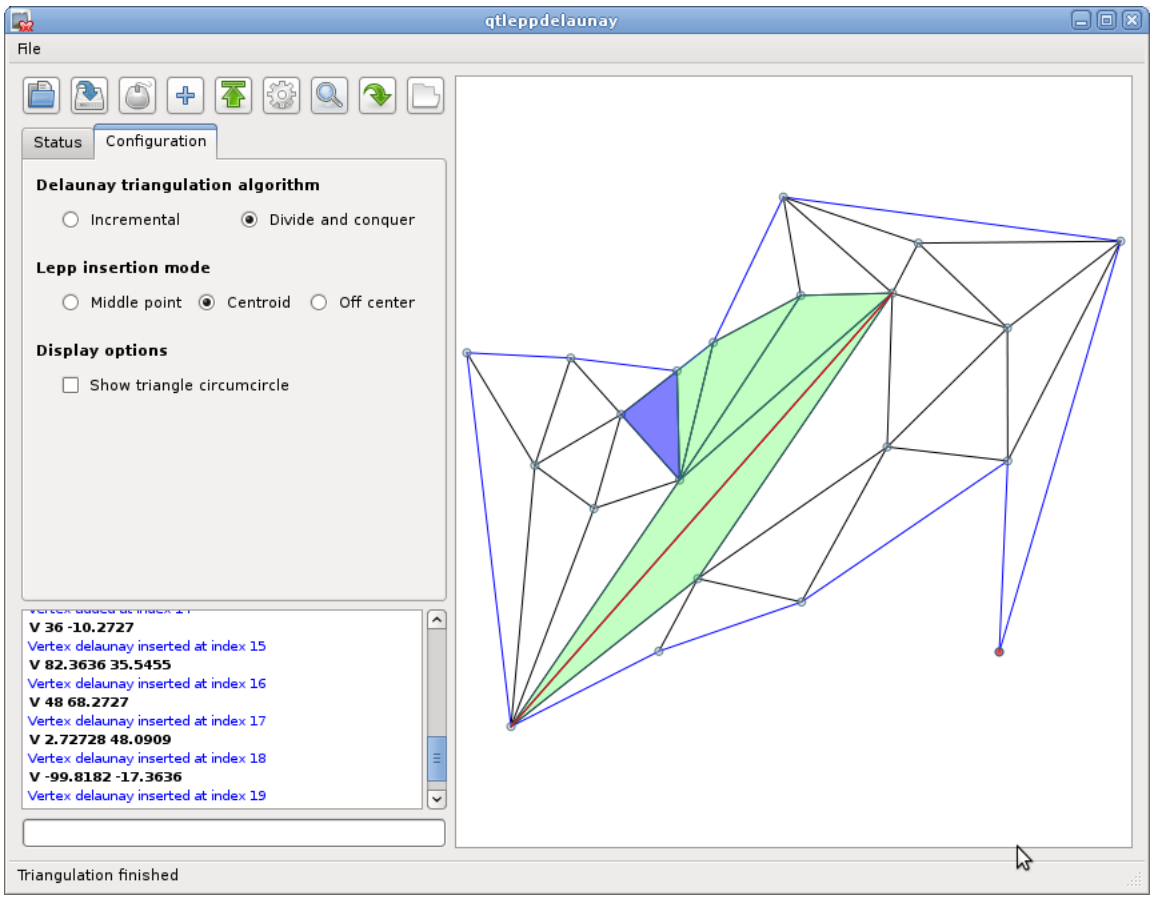

Figura 6.5: Interfaz de usuario: Lepp en un polígono con aristas restringidas. Tab de configuración.

La primera versión de la interfaz de usuario no es multi hilo. Como resultado, operaciones costosas en CPU producto de la ejecución de un comando en el controlador pueden bloquear o ralentizar la respuesta de la interfaz.

Capítulo 7

Validación

La validación de los elementos programados para este trabajo de memoria se realizó principalmente utilizando casos de prueba unitarios en el periodo de desarrollo, y casos de prueba generados automáticamente una vez que la implementación de cada modulo era terminada.

Los casos de prueba unitarios se encuentran junto al código fuente de la biblioteca y la herramienta de refinamiento, bajo el directorio `TestCases`. Algunos casos de prueba cuentan con comentarios sobre situaciones particularmente delicadas en la ejecución de los algoritmos.

Los casos de prueba construídos, además de ayudar en la verificación del trabajo realizado, son un excelente punto de partida para conocer los comandos soportados por el controlador.

7.1. Validación de los algoritmos geométricos

La validación de los algoritmos geométricos fue realizada mediante pruebas unitarias construídas a lo largo del desarrollo para así detectar tempranamente errores. La colección completa de pruebas unitarias comprende 27 casos donde el número de triángulos resultante es menor a diez.

El objetivo de los casos unitarios fue probar:

- Primitivas geométricas. Principalmente se buscó verificar el comportamiento ante la inserción de puntos colineales.
- Diversas configuraciones de triángulos para el algoritmo basado en dividir para conquistar.
- Inserción de puntos en triangulaciones sin aristas restringidas.
- Inserción de puntos en triangulaciones con aristas restringidas.
- Cálculo de LEPP las triangulaciones de los dos puntos anteriores.
- Selección de puntos utilizando LEPP, especialmente en el caso de que el o los triángulos terminales posean aristas restringidas.

Los problemas de precisión fueron abordados mediante la definición de un valor epsilon para las operaciones geométricas. La definición del epsilon utilizado se encuentra en el archivo de cabecera `Geometry.h`. El valor particular de epsilon debe ser establecido por cada

aplicación, principalmente pues la definición del tipo `Scalar`, usado en todas las coordenadas de la malla, puede corresponder a un número de punto flotante de precisión simple o doble.

Se recomienda utilizar los casos unitarios disponibles en posibles modificaciones a los algoritmos geométricos, pues permiten encontrar y depurar rápidamente errores en el código, dado su tamaño reducido.

La segunda etapa de validación fue realizada generando entradas aleatorias, de hasta diez mil puntos. El programa encargado de la validación es `testdelaunay`, el cual genera conjuntos de puntos aleatorios y los utiliza como entrada para los algoritmos de triangulación y refinamiento.

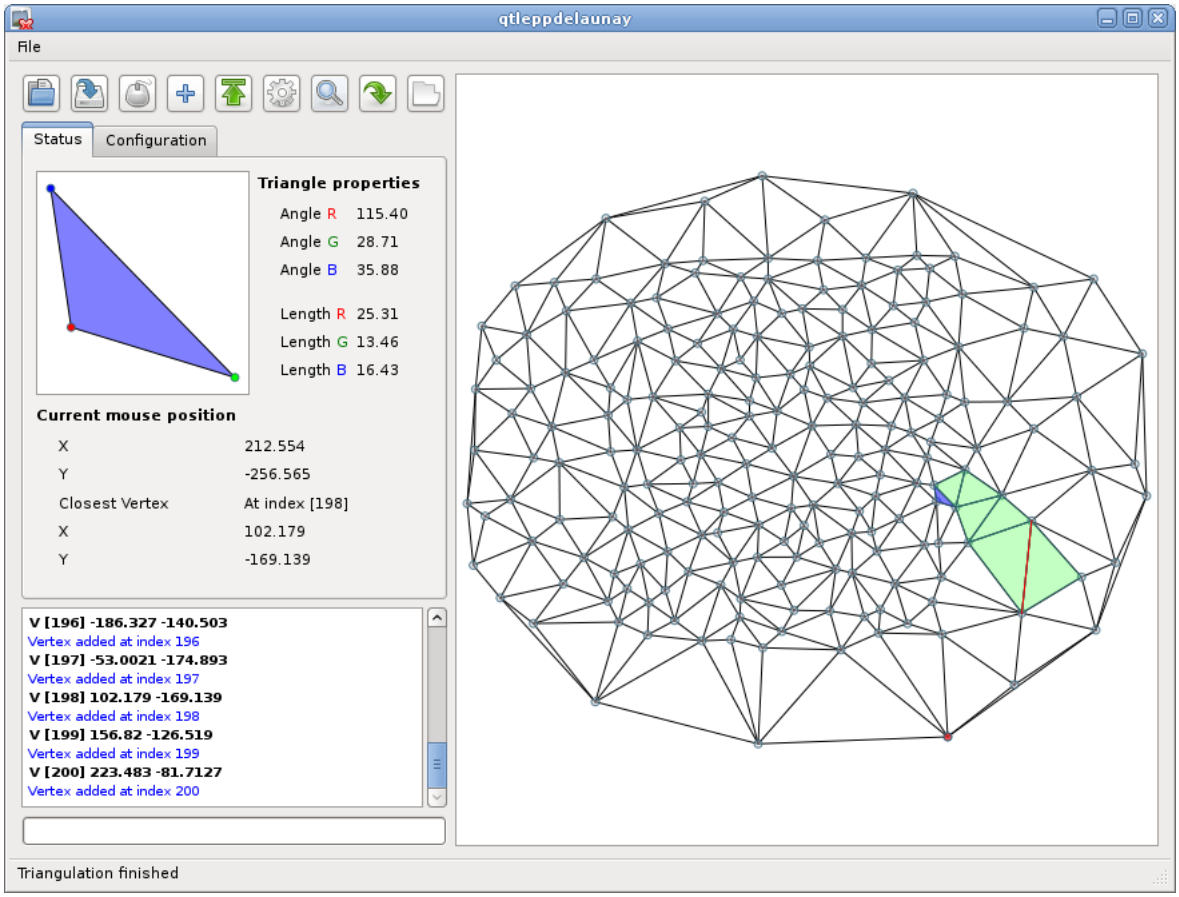

Figura 7.1: Caso de prueba de 200 vértices.

La validación de las triangulaciones generadas mediante conjuntos aleatorios de puntos se implementó en la función `checkTriangles` de la clase `OpenMeshDelaunay`. Esta función verifica que todos los triángulos de la triangulación pasen el test del círculo restringido.

La inserción de nuevos triangulos y la ejecución de los algoritmos Delaunay están sujetas a las pre-condición de que cada triángulo se encuentre en orientación `CCW`. Si esta condición no se cumple debido a un error en la biblioteca, la ejecución se detiene arrojando un mensaje de diagnóstico. En otras palabras, si un triángulo no es correctamente definido por los algoritmos implementados, el programa termina, no generando resultados que pudiesen contener triángulos inválidos.

7.2. Validación del Controlador

La validación del controlador fue realizada utilizando pruebas unitarias sobre cada comando disponible.

Las pruebas principalmente consideraron que la acción apropiada fuese realizada sobre la malla de triángulos, es decir, que cada comando fuese mapeado de manera correcta a alguna función de la biblioteca.

La segunda validación del Controlador consistió en utilizar la interfaz de usuario para generar diversos archivos de comandos, especialmente en modo interactivo, seleccionando vértices, realizando inserciones, inspeccionando triángulos y definiendo polígonos. Los archivos de comandos generados fueron cargados nuevamente, dando como resultado la misma geometría obtenida al utilizar la interfaz de usuario en modo interactivo.

7.3. Validación de la interfaz de usuario

La validación de la interfaz de usuario se realizó utilizando repetidamente sus capacidades. Sin embargo, puede existir casos de uso no cubiertos en la validación realizada. Se recomienda avisar al autor de este trabajo de memoria o a su profesor guía en caso de encontrar algún *bug*. Lo más útil en este tipo de reportes es incluir un archivo de comandos para reproducir el problema.

Capítulo 8

Conclusiones

La principal motivación de este trabajo de memoria fue estandarizar el desarrollo de aplicaciones basadas en triangulaciones Delaunay y estrategias de refinamiento basadas en LEPP. Para reducir los tiempos de desarrollo de dichas aplicaciones y permitir a sus autores enfocarse en problemas de mayor nivel, se implementó una biblioteca en el lenguaje de programación C++ que incluye las primitivas geométricas y los algoritmos base para el empleo de los métodos LEPP-Delaunay.

Figura 8.1: Jerarquía de módulos involucrados en LEPP-Delaunay.

Se puede argumentar que el desarrollo de este trabajo de memoria corresponde a las tres primeras capas para aplicaciones basadas en triangulaciones de Delaunay pues define las estructuras de datos básicas, el manejo de la conectividad y los algoritmos fundamentales para la manipulación de la malla de triángulos (ver figura 8.1).

A continuación se detallan los objetivos cumplidos en este trabajo de memoria.

8.1. Objetivo general

Se implementó una biblioteca de algoritmos de construcción de triangulaciones LEPP Delaunay en el lenguaje de programación C++. Adicionalmente, se construyó una herramienta de refinamiento para experimentar con la biblioteca, y proveer un entorno apropiado para el estudio de los métodos de refinamiento basados en LEPP-Delaunay. La herramienta

implementada se encuentra en la capa de las aplicaciones y ofrece la posibilidad de generar y guardar sesiones completas en archivos de comandos.

8.1.1. Objetivos específicos

Los objetivos específicos alcanzados en este trabajo de memoria fueron,

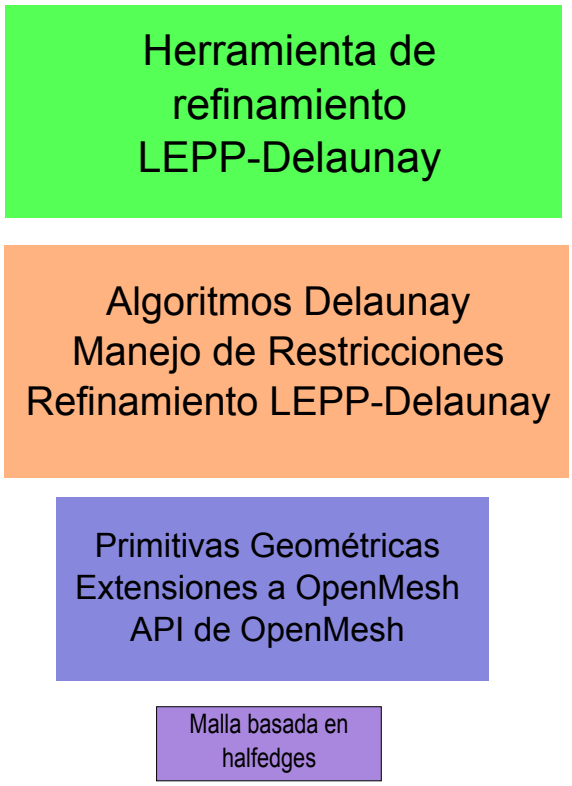

Figura 8.2: Jerarquía de módulos involucrados en LEPP-Delaunay.

1. La implementación de primitivas geométricas utilizadas en los algoritmos de triangulación de Delaunay y en el procesamiento general de mallas de triángulos.

2. La extensión de la biblioteca OpenMesh para soportar los algoritmos de triangulación y propiedades especiales de las aristas.

3. La implementación de los algoritmos de construcción de triangulaciones Delaunay incremental y el algoritmo basado en dividir para conquistar. La representación escogida se basa en la estructura de datos *halfedge* y la biblioteca OpenMesh.

4. La extensión de las primitivas geométricas y de los algoritmos de triangulación para soportar triangulaciones de Delaunay restringidas. Se implementó además la capacidad de definir polígonos.

5. La implementación de los métodos de refinamiento basados en LEPP-Delaunay, separando la determinación de aristas terminales de la selección e inserción de nuevos puntos. Esto permite futuras extensiones de los métodos a nivel de la selección de los puntos.

6. La creación de una biblioteca en el lenguaje de programación C++ con los algoritmos anteriormente mencionados, documentada detalladamente en su forma de uso y en las particularidades de la implementación. La documentación de la implementación fue generada utilizando la herramienta *Doxygen* y se encuentra disponible junto al código fuente de este trabajo de memoria.

7. La implementación de una herramienta para la creación y edición de triangulaciones de Delaunay y los métodos de refinamiento basados en LEPP. Dicha herramienta utiliza de manera activa la biblioteca de triangulaciones implementada y ofrece un ejemplo práctico de su uso. La interfaz de usuario de la herramienta de refinamienta se encuentra implementada utilizando QT versión 4.5.

Los objetivos mencionados anteriormente pueden ser visualizados en la figura 8.2 donde cada elemento de la jerarquía de la figura 8.1 fue reemplazado por las distintas capas implementadas en este trabajo de memoria. Las tres primeras capas corresponden a la biblioteca LEPP-Delaunay, mientras que la capa de aplicaciones se encuentra representada por la herramienta de refinamiento basada en LEPP-Delaunay.

Apéndice A

Comandos disponibles

Utilizando los siguientes comandos es posible definir triangulaciones sin la necesidad de escribir una línea de código, utilizando las facilidades provistas por el Controlador.

- `Clear` : Borra todos los contenidos de la sesión.
- `Reset` : Borra todas los triángulos y aristas restringidas de la sesión. Este comando es de utilidad cuando se desea volver a empezar, con el mismo conjunto de vértices existente en la sesión.
- `V [índice] X Y` : Añade un nuevo vértice a la triangulación en el punto (X,Y). El índice es importante pues identifica al vértice en el contexto del Controlador. Los vértices deben ser especificados en orden estrictamente creciente, comenzando desde 0.
- `Outer N i1 ... iN` : Define el contorno externo del conjunto de vértices. N es el número de vértices que definen el contorno externo. Siguen N índices de vértices previamente insertados con el comando `V` en orden CCW, que definen el borde exterior. Este comando debe ser ejecutado antes de realizar la triangulación.
- `Inner N i1 ... iN` : Define un contorno interior. Este comando debe ser ejecutado antes de realizar la triangulación.
- `Triangulate [DAC|INC]` : Ejecuta el algoritmo basado en dividir para conquistar si el argumento es `DAC` y el algoritmo incremental si el argumento es `INC`.
- `LEPP cotaInf [CEN|MID]` : Ejecuta el refinamiento basado en LEPP buscando el triángulo con menor ángulo en la triangulación, que sea menor que `cotaInf`. El punto a insertar es el centroide si el segundo argumento es `CEN`, o el punto medio, si el segundo argumento es `MID`.

Bibliografía

[1] Marc Vigo Anglada. An improved incremental algorithm for constructing restricted delaunay triangulations. 1997.

[2] Mario Botsch, Mark Pauly, Leif Kobbelt, Pierre Alliez, Bruno Lévy, Stephan Bischoff, and Christian Rössl. Geometric modeling based on polygonal meshes. In *SIGGRAPH Course Notes*, San Diego, California, 2007. ACM. revised course notes.

[3] P. J. C. Brown and C. T. Faigle. A robust efficient algorithm for point location in triangulations. *Technical report, Cambridge University*, 1996.

[4] RWTH Aachen. Computer Graphics Group. Project OpenMesh, 2009. http://www.openmesh.org/.

[5] Leonidas Guibas and Jorge Stolfi. Primitives for the manipulation of general subdivisions and the computation of voronoi. *ACM Trans. Graph.*, 4(2):74–123, 1985.

[6] Josef Hoschek and Dieter Lasser. *Fundamentals of computer aided geometric design*. A. K. Peters, Ltd., Natick, MA, USA, 1993. Translator-Larry L. Schumaker.

[7] Rivara MC and Palma N. New lepp algorithms for quality polygon and volume triangulation: Implementation issues and practical behavior. volume 220, pages 1–8, 1997.

[8] Maria-Cecilia Rivara and Carlo Calderon. Lepp terminal centroid method for quality triangulation. *Computer Aided Design*, 42(1):58–66, 2010.

[9] Maria Cecilia Rivara and Nancy Hitschfeld. Lepp-delaunay algorithm: a robust tool for producing size-optimal quality triangulations. In *Proc. of the 8th Int. Meshing Roundtable*, pages 205–220, 1999.

[10] Maria-Cecilia Rivara, Nancy Hitschfeld, and Bruce Simpson. Terminal-edges delaunay (small-angle based) algorithm for the quality triangulation problem. 2001.

[11] María Cecilia Rivara. New mathematical tools and techniques for the refinement and/or improvement of unstructured triangulations. In *Proceedings of 5th International Meshing Roundtable, Sandia National Laboratories*, pages 77–86, 1996.

[12] María Cecilia Rivara. New longest-edge algorithms for the refinement and/or improvement of unstructured triangulations. 1997.

[13] Jonathan Richard Shewchuk. Triangle: Engineering a 2D Quality Mesh Generator and Delaunay Triangulator. In Ming C. Lin and Dinesh Manocha, editors, *Applied Computational Geometry: Towards Geometric Engineering*, volume 1148 of *Lecture Notes in*

Computer Science, pages 203–222. Springer-Verlag, May 1996. From the First ACM Workshop on Applied Computational Geometry.

[14] Alper Üngör. Off-centers: A new type of steiner points for computing size-optimal quality-guaranteed delaunay triangulations. *Comput. Geom. Theory Appl.*, 42(2):109–118, 2009.