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

SOFTWARE DE COMPARACIÓN DE ALGORITMOS DELAUNAY DE REFINAMIENTO DE TRIANGULACIONES

MEMORIA PARA OPTAR AL TÍTULO DE INGENIERO CIVIL EN COMPUTACIÓN

FRANCISCA DANIELA GALLARDO PALACIOS

PROFESOR GUÍA:
MARÍA CECILIA RIVARA ZUÑIGA

MIEMBROS DE LA COMISIÓN:
BENJAMÍN BUSTOS CÁRDENAS
MAURICIO PALMA LIZAMA

SANTIAGO DE CHILE
JUNIO 2012

Resumen

Existen aplicaciones en donde una triangulación de buena calidad es esencial, entendiéndose por calidad que el ángulo mínimo de cada triángulo esté acotado inferiormente. El método de elementos finitos corresponde a una de las aplicaciones más importantes.

Los algoritmos de refinamiento de triangulaciones eliminan aquellos triángulos que poseen algún ángulo interior menor a un umbral deseado, mediante la inserción de nuevos puntos en la triangulación original. Un subconjunto de estos algoritmos, que reciben el nombre de algoritmos de refinamiento Delaunay, toman como punto de partida una triangulación de Delaunay restringida de los datos de entrada, y mediante inserciones Delaunay de los nuevos vértices mantienen la condición de Delaunay tras cada inserción. Estos algoritmos son los más utilizados para obtener triangulaciones de calidad.

En esta memoria se desarrolló un nuevo software de comparación de algoritmos de refinamiento llamado *Compare2DMesh*, el cual permite: hacer comparaciones experimentales rigurosas de los diferentes algoritmos de refinamiento Delaunay, manejar cualquier tamaño de mallas y geometrías complejas, visualizar cada inserción de un nuevo vértice mientras la triangulación está siendo refinada, refinar sin visualización del progreso, y configurar y ejecutar variaciones de los algoritmos. Este software usó como base un prototipo llamado *MeshSuite*.

Para validar el desempeño de *Compare2DMesh* se realizaron experimentos de comparación entre los algoritmos implementados, y también se comparó con otros software de refinamiento. Se concluyó que *Compare2DMesh* supera considerablemente el rendimiento del prototipo del que fue originado, y que procesa mallas grandes en tiempos razonables, lo que permite que *Compare2DMesh* pueda ser utilizado para los fines de probar, comparar y afinar algoritmos.

Agradecimientos

A mi profesora guía, Sra. María Cecilia Rivara, por su paciencia y por haber estado siempre disponible a lo largo de todo el desarrollo de esta memoria, guiando, corrigiendo e incluso animándome en los momentos difíciles.

A Alejandro Karaciolo, quien ha sido mi principal soporte durante todos estos años, por mantenerme con los pies en la tierra en los momentos más caóticos, apoyarme en los momentos más duros y celebrar conmigo los momentos satisfactorios.

A Esteban Allende, por enseñarme C++, ayudarme a depurar errores, discutir implementaciones conmigo, revisar mis informes, y todo el resto de la ayuda brindada a lo largo del proceso de titulación.

A mi familia, en especial a mi abuela Inés Urrutia, pues gracias a ella es que yo pude estudiar.

A mis amigos y amigas, por escucharme, por darme ánimos, por el simple hecho de estar ahí.

Y a todas las demás personas que no he nombrado y que me han apoyado de alguna forma durante esta importante etapa de mi vida.

Índice general

Índice de figuras

Capítulo 1

Introducción

1.1. Motivación

La importancia de los algoritmos de refinamiento de mallas radica en el hecho de que existen aplicaciones en donde una triangulación de buena calidad es esencial, entendiéndose por calidad que el ángulo mínimo de cada triángulo esté acotado inferiormente. El método de elementos finitos, que permite resolver numéricamente problemas modelados por ecuaciones diferenciales parciales para el análisis de complejos problemas físicos, corresponde a una de las aplicaciones más importantes.

Los diversos métodos que se han desarrollado con el fin de mejorar la calidad de una malla, producen triangulaciones de salida con diferentes características. Sin embargo, hasta el momento no se ha realizado un estudio empírico riguroso de comparación de los algoritmos de refinamiento de triangulaciones más conocidos. Los resultados de un estudio como éste permitirían: entregar información para elegir, de acuerdo a las características del problema, el algoritmo adecuado para su resolución; obtener información práctica que pueda ser utilizada para analizar posibles mejoras en base al comportamiento empírico de los algoritmos; conocer las fortalezas y debilidades de cada estrategia de refinamiento, etc. Para ello, es necesario contar con una herramienta que permita hacer comparaciones experimentales bajo las mismas condiciones de implementación.

Un prototipo de software de comparación de algoritmos de refinamiento fue desarrollado en una memoria previa por Álvaro Faúndez. El software, llamado *MeshSuite*, posibilita la ejecución de diversos algoritmos de refinamiento conocidos y variaciones de éstos, mediante una interfaz gráfica que permite configurar manualmente el método a utilizar, combinando diferentes estrategias de priorización de triángulos, selección e inserción de puntos. *MeshSuite* provee de una visualización gráfica de las inserciones de puntos realizadas a través del proceso de refinamiento en tiempo real, de manera muy didáctica. El diseño de esta herramienta privilegió la simplicidad del manejo de algoritmos a través de la interfaz gráfica, así como el logro de un código fuente que facilite añadir nuevas opciones de priorización de triángulos y criterios de selección e inserción de puntos a la aplicación. Sin embargo, la

herramienta no incluye implementaciones eficientes de los algoritmos y solo permite refinar mallas relativamente pequeñas (miles de puntos).

En esta memoria se desarrolló una nueva herramienta de software de comparación de algoritmos de refinamiento, con el fin de utilizarla en una comparación experimental rigurosa entre diferentes algoritmos de refinamiento Delaunay de triangulaciones. Para ello, fue necesario revisar y reimplementar parte importante de los algoritmos incluidos en *MeshSuite*. Estos algoritmos fueron reimplementados de manera eficiente, aprovechando las ventajas y características propias de cada uno. Se mantuvo solo la interfaz gráfica y las características de diseño generales de *MeshSuite*. El software desarrollado recibe el nombre de ***Compare2DMesh***.

1.2. Objetivos

1.2.1. Objetivo general

El objetivo general de esta memoria consiste en el desarrollo de un software que permita analizar las características de las triangulaciones obtenidas a partir de diferentes algoritmos de refinamiento de mallas, mediante la realización de pruebas experimentales que permitan hacer una comparación rigurosa de los algoritmos, con la finalidad de establecer, en base a los resultados obtenidos, las condiciones bajo las cuales una estrategia es mejor que otra. Esto requiere de la revisión y mejoramiento de una herramienta de software de refinamiento de mallas existente (MeshSuite), de modo de adaptarla para cumplir con los requerimientos de implementación de las pruebas.

1.2.2. Objetivos específicos

- Investigar los diferentes algoritmos de tipo Delaunay, que se han desarrollado para refinar triangulaciones (Revisión bibliográfica).
- Adaptar software de comparación de estrategias de refinamiento de mallas MeshSuite, desarrollado por Álvaro Faúndez, considerando las siguientes necesidades:
    i. Eficiencia en los tiempos de ejecución de los algoritmos estrategia de refinamiento.
    ii. Eficiencia en el uso de memoria.
- Validar la herramienta desarrollada, a través de un estudio de las siguientes características de las triangulaciones obtenidas a partir de diferentes algoritmos de refinamiento:
    i. Evolución de los ángulos a medida que avanza el proceso de refinamiento.
    ii. Tamaño del menor ángulo obtenido.
    iii. Número de vértices y triángulos obtenidos.
    iv. Tiempo de ejecución.

1.2.3. Objetivos a futuro en estudio riguroso

- Estudiar, utilizando la misma herramienta, nuevas combinaciones de criterios de selección e inserción de puntos.
- Comparar todos los algoritmos de refinamiento estudiados y sus variaciones, para finalmente concluir en qué casos una estrategia es mejor que otra.

Capítulo 2

Triangulación de Delaunay

2.1. Triangulación

Una triangulación en dos dimensiones sobre un conjunto de puntos, se define como una subdivisión del plano en caras triangulares cuyos vértices corresponden a los puntos del conjunto inicial. En tres dimensiones, una triangulación es un conjunto de triángulos que aproximan una superficie en el espacio a partir de un conjunto de puntos sobre esa superficie. Ambos tipos de triangulación se construyen formando caras planas triangulares, uniendo pares de puntos mediante aristas que nunca se cruzan entre si.

Las triangulaciones también se conocen por el nombre de mallas de triángulos. En la Figura 2.1 se muestran dos ejemplos de mallas, una sobre el plano y otra sobre una superficie en tres dimensiones.

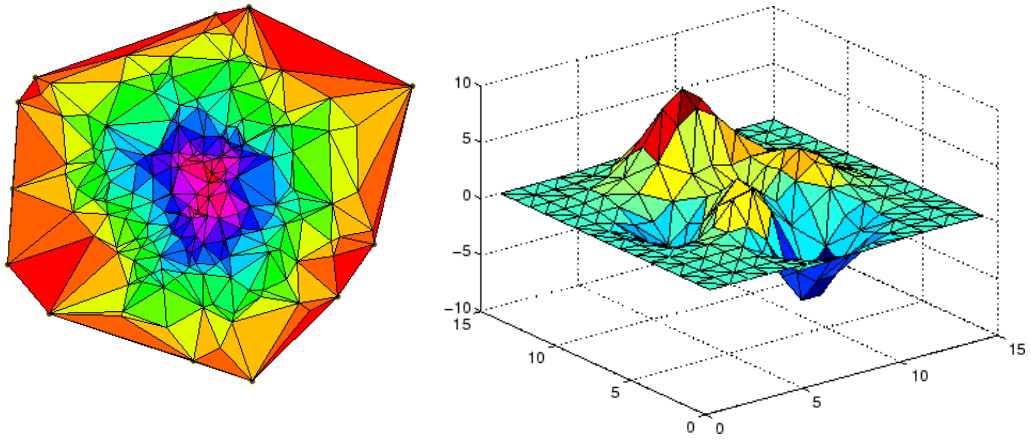

Figura 2.1: Aproximación de superficies mediante mallas de triángulos, en dos y tres dimensiones.

Se dice que una triangulación o malla de triángulos es válida si cumple con las siguientes

características:

I. Todos los triángulos construidos poseen área mayor que cero.

II. La intersección del interior de dos triángulos cualesquiera del conjunto es siempre vacía.

III. La intersección de dos triángulos cualesquiera del conjunto corresponde a una arista común o a un vértice común.

IV. El conjunto define una superficie conexa, abierta o cerrada.

Las triangulaciones son utilizadas en aplicaciones tales como: modelación de terrenos, análisis de fenómenos físicos modelados mediante ecuaciones diferenciales, computación gráfica y animaciones, entre otros. Un problema de particular interés para la geometría computacional y otras aplicaciones tales como cartografía, análisis de datos espaciales y el método de elementos finitos [10], es encontrar la triangulación de un conjunto de puntos sobre el plano que maximice el menor ángulo interno de los triángulos que la conforman. Debido a la gran cantidad de aplicaciones científicas en las que se utilizan mallas en dos dimensiones, y que en muchos casos un problema sobre una superficie tridimensional se resuelve a través de la proyección de puntos de dicha superficie sobre el plano, el resto de la memoria se centra en la generación de triangulaciones en **dos dimensiones**.

2.2. Triangulación de Delaunay

La triangulación de Delaunay [1, 2] ha sido ampliamente estudiada en el ámbito de la geometría computacional debido a sus aplicaciones en diferentes áreas de la ciencia y la ingeniería. Su definición formal se presenta a continuación:

Definición: [**Triangulación de Delaunay**]. Una triangulación del conjunto P de puntos sobre el plano es de Delaunay, si y solo si el circuncírculo[1] de cualquier triángulo de la malla no contiene un punto de P en su interior (ver Figura 2.2).

2.2.1. Propiedades

La triangulación de Delaunay satisface las siguientes propiedades:

I. La triangulación maximiza el menor ángulo de la malla, es decir, el menor de los ángulos internos de los triángulos que la conforman.

[1]Circuncírculo corresponde a la circunferencia que pasa por todos los vértices de un polígono regular. También se conoce como "circunferencia circunscrita".

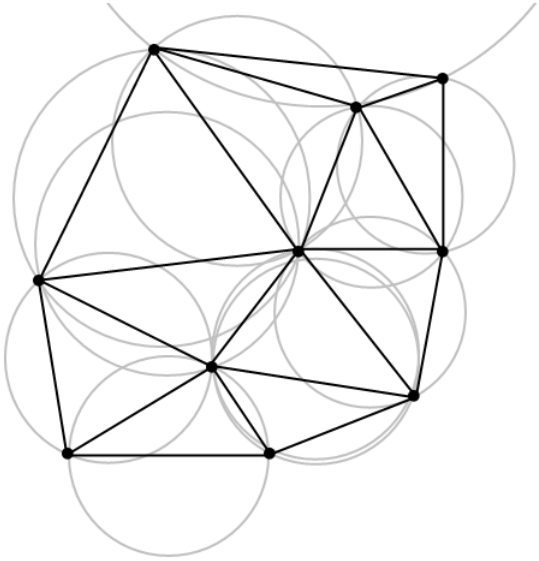

Figura 2.2: Triangulación de Delaunay sobre un conjunto de 10 puntos.

II. La frontera de la triangulación es la envolvente convexa de los puntos, en otras palabras, las aristas del borde de la triangulación forman un polígono convexo que contiene todos los demás puntos.

III. La triangulación es única cuando ningún borde de circunferencia circunscrita contiene más de tres vértices de la malla.

La primera propiedad es de mucha importancia, pues la mayor parte de las aplicaciones de las triangulaciones se ve afectada por la forma de los triángulos, en particular, cuando éstos son muy delgados (en otras palabras, cuando poseen algún ángulo interior muy pequeño). Un ejemplo de este problema ocurre al tratar de aproximar un terreno. Si la malla tiene triángulos muy delgados, el valor de la altura de un punto sobre una arista de la triangulación podría estar muy alejado del valor real en esa posición del terreno. Esto se debe a que en la triangulación que aproxima un terreno, la altura de cada posición se calcula en base al valor de la altura de los vértices cercanos mediante interpolación lineal, entre más lejos estén los vértices del punto que se está interpolando, más probabilidades hay de que la aproximación sea incorrecta, pues en general, en un terreno real la altura entre dos puntos cercanos no varía mucho, por lo tanto si los vértices opuestos a la arista están más cerca de ella que sus vértices extremos, los valores interpolados sobre la arista podrían ser muy diferentes a las alturas de sus vértices más cercanos, lo cual no se ajusta a la variación de alturas que presenta un terreno en la realidad.

2.2.2. Algoritmo básico de construcción

Existen diferentes algoritmos para construir la triangulación de Delaunay de un conjunto P de puntos dado. El algoritmo más simple permite *Delaunizar* una triangulación válida

cualquiera de P, modificando sus aristas para obtener una triangulación de Delaunay. El criterio para decidir qué aristas deben modificarse se basa en la siguiente definición.

Definición: **[Arista ilegal].** Sean t_1 y t_2 dos triángulos adyacentes que pertenecen a una triangulación, y sean p_i y p_j los vértices de la arista compartida por ambos triángulos. Se dice que la arista $\overline{p_i p_j}$ es ilegal si el círculo circunscrito de uno de los triángulos contiene en su interior un vértice del otro (ver Figura 2.3). Además, se cumple que si t_1 y t_2 forman un cuadrilátero convexo cuyos vértices no son cocirculares, entonces una y solo una de las diagonales de dicho cuadrilátero es una arista ilegal [1].

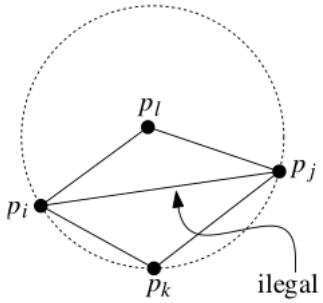

Figura 2.3: Ejemplo de arista ilegal

El algoritmo básico de construcción de la triangulación de Delaunay consiste en obtener primero una triangulación válida cualquiera del conjunto de puntos inicial, y eliminar todas las aristas ilegales de esa triangulación, pues por definición, la triangulación de Delaunay no tiene aristas ilegales. Para ello, se realizan las siguientes operaciones a cada triángulo:

I. **Test del círculo:** Para el triángulo t, esta operación verifica que el circuncírculo de t no contenga un vértice de otro triángulo en su interior, es decir, el test valida que t cumpla la condición de Delaunay.

II. **Intercambio de diagonales (para *Delaunizar* pares de triángulos):** Si el test del círculo falla, entonces el circuncírculo de t contiene un vértice de algún triángulo adyacente t_2, por lo tanto, la arista que comparten t y t_2 es ilegal y debe ser reemplazada por la otra diagonal del cuadrilátero formado por ambos triángulos, como se muestra en la Figura 2.4.

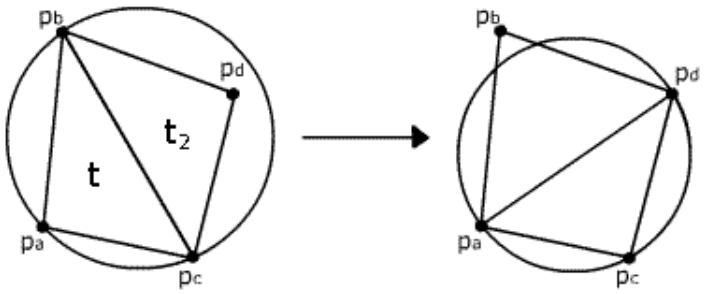

Figura 2.4: Intercambio de diagonales.

Las operaciones anteriores se repiten hasta que ya no quede ninguna arista ilegal en la triangulación.

En pseudocódigo, este algoritmo queda de la siguiente forma:

```
DelaunizarTriangulacion(T):
    Continuar = True
    while( Continuar )
        Continuar = False
        foreach( Triangulo t perteneciente a T )
            if( TestDelCirculo(t) == False )
                E = ObtenerAristaIlegal(t)
                IntercambiarDiagonales(E,T)
                Continuar = True
                break /* Salir del bucle y recorrer nuevamente T */
    return T
```

El algoritmo descrito resuelve bien el problema de construir una triangulación de Delaunay, pero su tiempo de ejecución es de $O(n^2)$, lo cual es muy lento. Esto se debe a que durante el proceso, cada vez que se realiza la operación de intercambio de diagonales, la triangulación es modificada y por ende debe ser recorrida nuevamente en búsqueda del siguiente triángulo a *Delaunizar*.

El mejor algoritmo para construir una triangulación de Delaunay se basa en la estrategia "Dividir para reinar" y en la práctica construye la triangulación en $O(n \log n)$. Sin embargo, los algoritmos más utilizados construyen la malla de forma incremental.

2.2.3. Algoritmo incremental para construir triangulaciones de Delaunay

En el algoritmo incremental, la triangulación de Delaunay se computa insertando uno a uno los vértices en la malla. Inicialmente, se construye un gran triángulo que contiene todos los puntos del conjunto, esto implica agregar tres puntos adicionales que al final del proceso son eliminados. Dicho triángulo se considera como la triangulación inicial del conjunto. Posteriormente, se repite el siguiente proceso hasta tener una triangulación de Delaunay que contenga todos los vértices del conjunto inicial:

I. Se escoge aleatoriamente un punto del conjunto inicial, que actualmente no pertenece a la malla

II. Se realiza una inserción de dicho punto, mediante un algoritmo conocido como Inserción Delaunay por intercambio de diagonales, el cual se describe a continuación.

Inserción Delaunay (por intercambio de diagonales): Corresponde a un método para insertar un nuevo vértice a una triangulación de Delaunay [11], el cual consiste en:

I. Encontrar el triángulo t de la malla que contiene al punto p que se desea insertar.

II. Una vez identificado t, realizar la inserción de p de acuerdo a alguno de los siguientes criterios:

   a) Si el punto p se encuentra al interior del triángulo, unir cada vértice de t a dicho punto, formando así tres nuevos triángulos, tal y como se aprecia en la Figura 2.5a.

   b) Si el punto p se encuentra sobre una de las aristas del triángulo, considerar el triángulo t_2 que comparte esa arista con t y unir los vértices de ambos triángulos a p, formando así cuatro nuevos triángulos como puede verse en la Figura 2.5b.

   c) Si el punto p se encuentra sobre uno de los vértices del triángulo, el punto no se inserta.

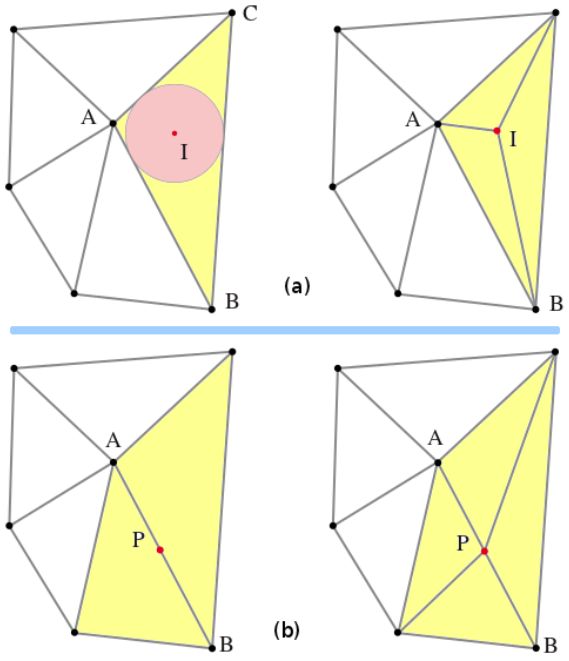

Figura 2.5: (a) Inserción de punto cuando este se encuentra al interior del triángulo. (b) Inserción del punto cuando este se encuentra en una arista.

III. La triangulación resultante tras la inserción del punto p no es necesariamente de Delaunay, por lo que se debe realizar el test del círculo a cada uno de los nuevos triángulos creados en el paso anterior y, en cada caso que falle el test, realizar el intercambio de diagonales correspondiente. Este proceso se repite sobre los triángulos creados por el intercambio de diagonales, y continúa recursivamente hasta obtener nuevamente una triangulación de Delaunay.

Finalmente, cuando se han insertado todos los puntos del conjunto inicial en la triangulación, los tres puntos adicionales que fueron agregados al comienzo del algoritmo son eliminados. Esto involucra también la eliminación de todos aquellos triángulos que contienen dichos puntos en alguno de sus vértices.

En pseudocódigo, este algoritmo queda de la siguiente forma:

```
TriangulacionIncrementalDeDelaunay(P):
    T = Triangulo(v1, v2, v3) /* Triangulación inicial */
    foreach( p perteneciente a P )
        T = InsercionDelaunay(T,p)

    /* Eliminar vértices artificiales y triángulos que los
contienen */
    T = Eliminar(v1, v2, v3)
    return T
InsercionDelaunay(T,p):
    /* Encontrar triángulo que contiene a p */
    t = ObtenerTriangulo(T,p)
    /* Inserción de p */
    if( not(p sobre vertice de t) )
        if( p dentro de t )
            [t1,t2,t3] = Split(t,p)
            T = Delaunizar(T, [t1,t2,t3])
        else if ( p sobre arista de t,t2)
            [t1,t2,t3,t4] = Split(t1,t2,p)
            T = Delaunizar(T, [t1,t2,t3,t4])
    return T
Delaunizar(T, L): /* L = Lista de triángulos */
    foreach( t perteneciente a L )
        if( TestDelCirculo(t) == False )
            E = ObtenerAristaIlegal(t)
            [t1,t2] = IntercambiarDiagonales(E)
            Delaunizar(T, [t1,t2])
    return T
```

Es importante mencionar que el algoritmo de inserción Delaunay de puntos, utilizado en la construcción incremental de la triangulación de Delaunay, es también usado en los algoritmos de refinamiento de triangulaciones, los cuales se detallan más adelante en esta memoria.

En la Figura 2.6 se muestra un ejemplo de inserción Delaunay.

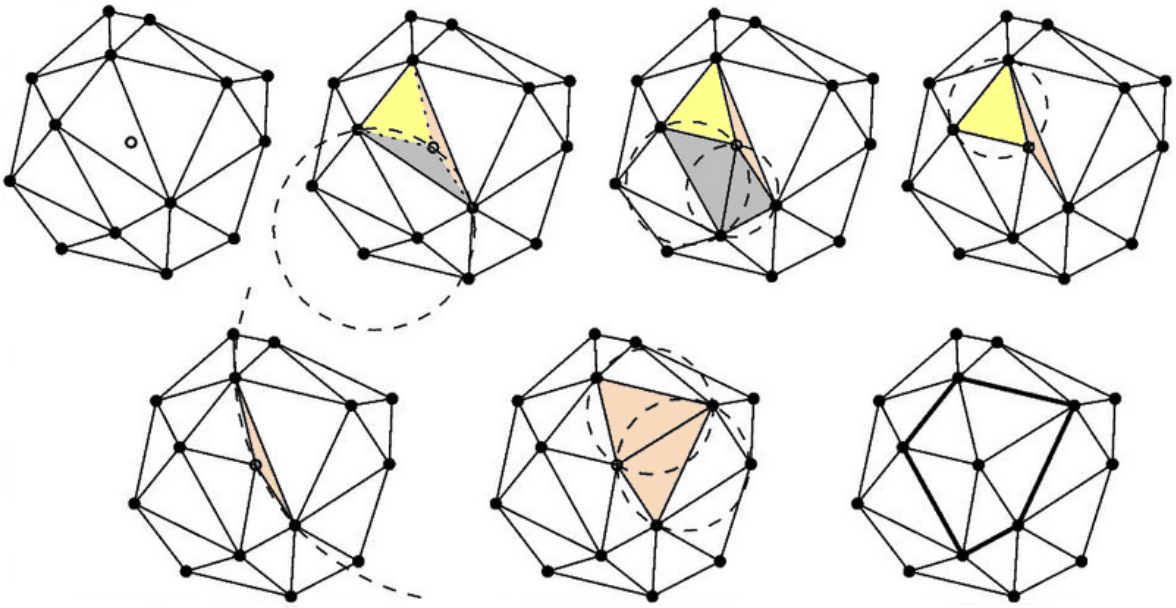

Figura 2.6: Inserción Delaunay de un punto, paso a paso.

2.2.4. Triangulación de Delaunay restringida

En aplicaciones reales, con frecuencia se requiere construir una triangulación de un conjunto de vértices y aristas sobre el plano o PSLG (por sus siglas en inglés para *Planar Straight-Line Graph*), manteniendo las aristas del conjunto inicial en la triangulación resultante. Una triangulación con estas características se conoce como triangulación con **aristas restringidas**. En el problema de triangularizar un polígono (que es un caso particular de PSLG), los bordes de éste también se consideran aristas restringidas.

Es posible construir una triangulación de Delaunay de un PSLG, relajando la condición de Delaunay, la cual exige que el circuncírculo de cualquier triángulo de la malla no contenga otro vértice en su interior.

Se define entonces la triangulación de Delaunay restringida, la cual se describe a continuación.

Definición: **[Triangulación de Delaunay restringida].** Una malla de triángulos construida sobre un conjunto de puntos y aristas G sobre el plano, es una triangulación de Delaunay restringida si para cada triángulo t de la malla se cumple que el circuncírculo de t no contiene un vértice de G en su interior, o el circuncírculo de t posee un vértice de G en su interior pero éste se encuentra en el lado opuesto de una arista restringida de t, actuando dicha arista como una muralla que bloquea la visión al resto de la malla, y por ende, a cualquier punto que esté en el interior del circuncírculo.

En la Figura 2.7 se muestran algunos ejemplos de este tipo de triangulación.

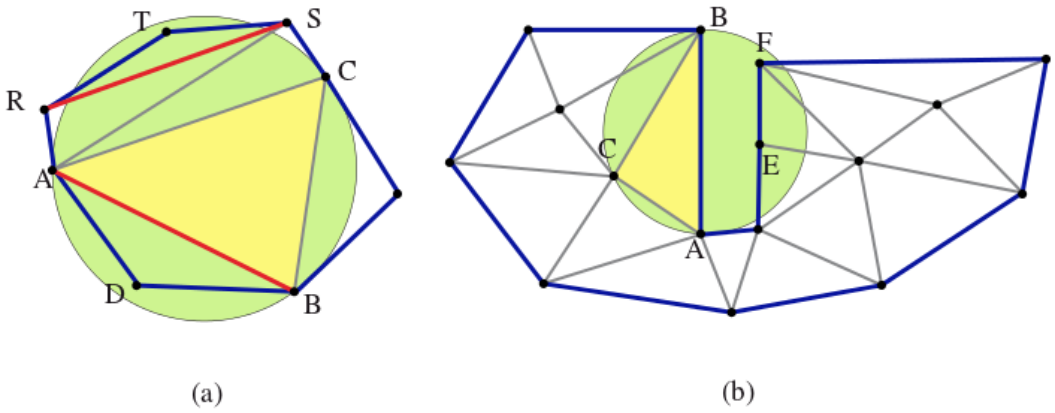

Figura 2.7: Triangulaciones Delaunay Restringidas En ambas figuras, los segmentos rojos, azules y los vértices negros son elementos iniciales. (a) Las aristas restringidas RS y AB en la figura permiten ignorar los vértices T y D respectivamente. (b) El borde de malla AB permite ignorar los vértices E y F.

Capítulo 3

Triangulaciones de calidad: refinamiento Delaunay

Aunque la triangulación de Delaunay maximiza el menor ángulo de la malla, esto no garantiza que en ella no existan ángulos muy pequeños, pues el tamaño de los ángulos de la triangulación de Delaunay depende de la distribución de puntos del conjunto inicial. Si la triangulación además posee aristas restringidas, la probabilidad de tener triángulos de ángulo muy pequeño es mayor. Un ejemplo de esto ocurre al triangularizar un polígono cóncavo o con huecos (ver Figura 3.1).

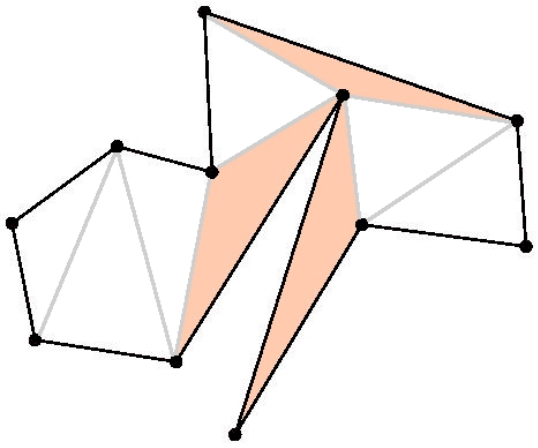

Figura 3.1: Triangulación de un polígono cóncavo. Los triángulos de color poseen ángulos muy pequeños y visualmente se ven alargados.

Existen aplicaciones científicas en las cuales es importante que la triangulación sobre la cual se trabaja sea de "calidad", es decir, que el menor ángulo del conjunto esté acotado inferiormente. Los triángulos deben ser lo más equiláteros posible, debido a que triángulos con ángulos interiores muy grandes o muy pequeños a menudo llevan a errores de cálculo, degradando, por ejemplo, la calidad de la solución numérica de un problema de elementos finitos [9] o, en el caso de la interpolación, produciendo grandes errores en los gradientes

de una superficie interpolada. El criterio para decidir si un ángulo es demasiado pequeño depende de la aplicación para la que se esté utilizando la malla.

Debido a esta necesidad de tener triangulaciones de buena calidad, a lo largo de los años se han desarrollado diferentes algoritmos que buscan incrementar el tamaño del ángulo mínimo de una malla. En estos algoritmos, que reciben el nombre genérico de **algoritmos de refinamiento de triangulaciones**, se define el valor del ángulo mínimo deseado, que corresponde al parámetro de calidad, y mediante la inserción de nuevos puntos en la triangulación original (llamados puntos de Steiner), se eliminan aquellos triángulos que poseen algún ángulo menor al deseado. El resultado es una malla con más vértices y triángulos que la original, pero con una distribución de puntos que produce triángulos de ángulo mínimo acotado por el valor definido antes de comenzar el procedimiento (ver Figura 3.2).

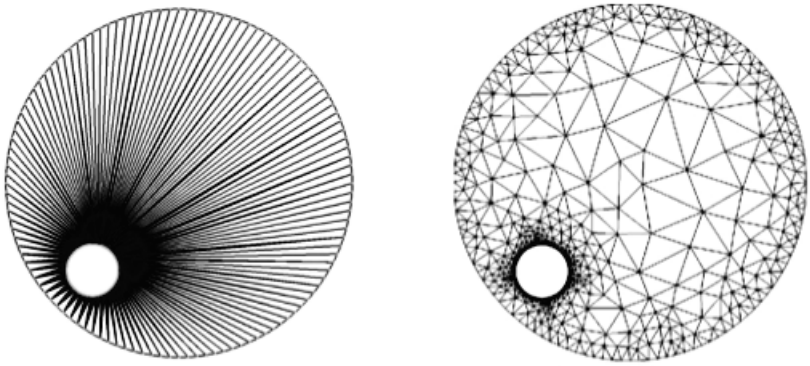

Figura 3.2: Ejemplo de refinamiento de un círculo con un hueco en su interior. Refinamiento obtenido utilizando el algoritmo Lepp-Delaunay centroide. Ángulo mínimo logrado de 32 grados, 493 puntos insertados.

En esta memoria se estudia un subconjunto de estos algoritmos, los cuales toman como punto de partida una triangulación de Delaunay restringida de los datos de entrada, y luego agregan de uno en uno nuevos vértices a la malla mediante inserciones Delaunay, lo que garantiza que la condición de Delaunay se mantiene luego de insertar cada nuevo punto en la triangulación. Dichos algoritmos reciben el nombre de **algoritmos de refinamiento Delaunay**.

Los algoritmos de refinamiento Delaunay permiten mejorar la calidad de una triangulación de Delaunay inicial (que por lo general es restringida) mediante la selección e inserción de puntos de Steiner, manteniendo la condición de Delaunay en la malla resultante.

El procedimiento general que se realiza en este tipo de algoritmos se describe a continuación:

Sea T una triangulación de Delaunay restringida y α el criterio de calidad (valor mínimo deseado para los ángulos de T). Los siguientes pasos se repiten hasta haber eliminado de la triangulación todos los triángulos que contienen algún ángulo menor que α.

1. Seleccionar en la triangulación T un triángulo t que tenga algún ángulo menor que α (*)

II. Seleccionar (calcular) el punto P que será insertado en la triangulación T con el fin de eliminar el triángulo t.

III. Realizar una inserción Delaunay del punto P en la triangulación.

(*) En algunos casos, antes del paso I de la primera iteración, puede ser necesario hacer un preproceso a la triangulación con el fin de evitar o disminuir la probabilidad de que ocurran casos de borde como, por ejemplo, tratar de insertar un punto fuera de la triangulación o que se produzca un ciclo infinito.

En pseudocódigo, el proceso descrito anteriormente es el siguiente:

```
alpha = valor_angulo_minimo_deseado
T = Triangulacion
while (existe triangulo con angulo menor a alpha en T)
    if (primera iteracion)
        /* El paso de preproceso de T es opcional */
        preProcesarTriangulacion(T)
    triangulo t = seleccionarTrianguloMalaCalidad(alpha,T)
    punto p = seleccionarPunto(t,T)
    insertarPunto(p,T)
end while
```

Los distintos algoritmos de refinamiento Delaunay existentes, se diferencian principalmente por el criterio que utilizan para seleccionar el punto P señalado en el paso II del proceso antes descrito. A continuación, se describen los diferentes criterios de selección de punto para los algoritmos de refinamiento Delaunay que se desean estudiar.

3.1. Selección de punto basada en el circuncentro del triángulo a refinar

En esta sección se describen dos algoritmos de refinamiento en los cuales el cálculo del punto a insertar se hace en base al circuncentro del triángulo seleccionado para refinar.

3.1.1. Selección simple del circuncentro: Algoritmo de Ruppert

Para cada triángulo t de mala calidad en la malla, el algoritmo de Ruppert selecciona el circuncentro de t y realiza una inserción Delaunay de este en la triangulación.

En el algoritmo de Ruppert [4] los triángulos de mala calidad se procesan en orden, de menor a mayor ángulo. Además, en este algoritmo es necesario realizar un preproceso a la

malla al inicio de la primera iteración. Dicho preproceso consiste en eliminar ciertas aristas de la triangulación, conocidas como **aristas *encroached***, cuya definición se presenta a continuación.

Definición: **[Arista *encroached*].** Se dice que una arista restringida de una triangulación es una arista *encroached*, siempre y cuando su circunferencia diametral (aquella circunferencia cuyo diámetro se corresponde con la arista) contiene al interior algún vértice de la malla.

Cuando un triángulo posee una arista *encroached*, su circuncentro puede caer fuera del dominio de la triangulación, por lo que el punto no puede ser insertado. Es por esto que se deben eliminar de la malla todos los triángulos que poseen aristas *encroached* y para ello se divide cada una de esas aristas por la mitad, mediante una inserción Delaunay, lo que se repite hasta que ningún segmento de dichas arista sea *encroached* (ver Figura 3.3).

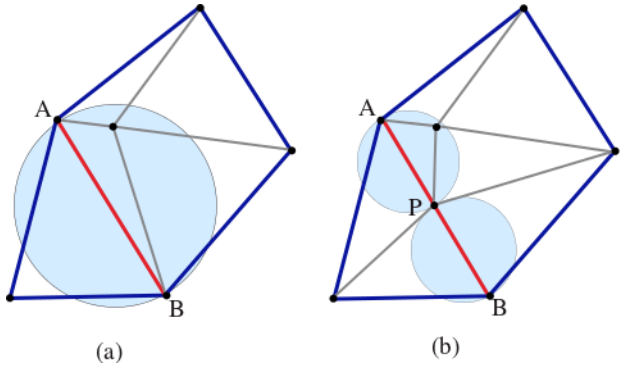

Figura 3.3: Eliminación de aristas *encroached*. (a) Arista *encroached* $\overline{AB}$. (b) Se introduce P, punto medio de $\overline{AB}$ y se obtiene arista *encroached* más pequeña $\overline{AP}$.

Posteriormente, comienza el refinamiento de los triángulos de mala calidad y durante el proceso, antes de cada inserción, se verifica si el circuncentro del triángulo seleccionado producirá una o más aristas encroached. En caso afirmativo, se inserta el punto medio de las aristas afectadas (ver Figura 3.4a) y, en caso contrario, se procede a insertar el circuncentro del triángulo (ver Figura 3.4b).

En pseudocódigo, el algoritmo de Ruppert queda de la siguiente manera:

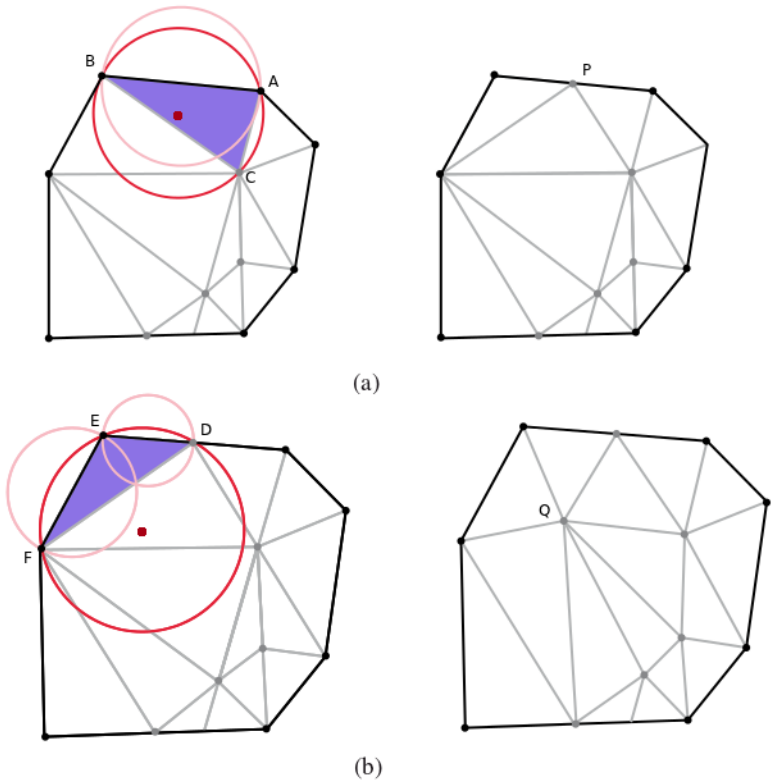

Figura 3.4: Refinamiento de Ruppert. En el caso (a) el circuncentro produce una arista *encroached*, se inserta el punto medio de la arista. En el caso (b) no involucra ninguna arista restringida, se inserta el circuncentro.

```
alpha = valor_angulo_minimo_deseado
T = Triangulacion
while (existe triangulo con angulo menor a alpha en T)
    if (primera iteracion)
        T = eliminarAristasEncroached(T)
    triangulo t = seleccionarTrianguloMenorAnguloMinimo(alpha,T)
    punto p = calcularCircuncentro(t,T)
    triangulo s = encontrarTrianguloContiene(p,T)
    T = insertarPunto(p,s,T)
end while
eliminarAristasEncroached(T):
    while (T posee arista encroached E)
        T = insertarPuntoMedio(E,T)
    return T
insertarPunto(p,s,T):
    if (insercion Delaunay de p dentro de s produce aristas
encroached [E1,...,En])
        foreach (E en [E1,...,En])
            q = PuntoMedio(E)
            T = insercionDelaunay(q,E,T)
    else
        T = insercionDelaunay(p,s,T)
```

3.1.2. Selección del Off-center: Algoritmo de Üngör

El Algoritmo Off-center [7] propone una alternativa a la elección del circuncentro como nuevo punto a insertar, considerando un ángulo umbral de calidad α. El punto propuesto se describe en la siguiente definición:

Definición: **[Off-center].** Dado un triángulo t de mala calidad, se construye un segmento que une el circuncentro P de t con el punto medio M de la arista más pequeña de t. Luego, se selecciona un punto O sobre el segmento de recta $\overline{PM}$ que, unido a los vértices de la arista más pequeña del triángulo a refinar, forme un triángulo de ángulo interior α (ver Figura 3.5). A este punto se le conoce como Off-center.

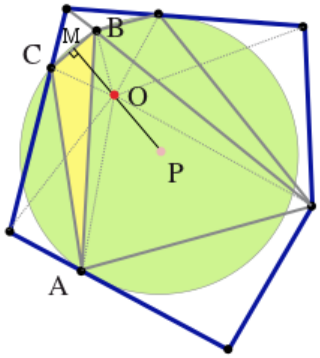

Figura 3.5: El punto O corresponde al Off-center del triángulo ABC.

El objetivo del Off-center es que el triángulo formado por los extremos de la arista de menor tamaño del triángulo a refinar y el mismo Off-center, tenga todos sus ángulos interiores mayores a una cota inferior α, si esto se cumple, entonces se realiza una inserción Delaunay de dicho punto.

Al igual que en el algoritmo de Ruppert, el Off-center de un triángulo puede caer fuera de la triangulación, por lo que es necesario eliminar las aristas *encroached* de la triangulación antes de comenzar las inserciones. En este algoritmo también se cumple la regla de que si el Off-center del triángulo seleccionado produce que alguna arista sea *encroached*, entonces se inserta el punto medio de dicha arista.

El algoritmo de Üngör utiliza también un segundo preproceso, que consiste en refinar primero aquellos triángulos de la malla cuya arista más pequeña sea restringida. Esto produce como resultado que el número de puntos insertados durante el proceso de refinamiento sea mucho menor que en los algoritmos basados en la inserción del circuncentro de los triángulos.

Una vez realizado el preprocesamiento inicial de la triangulación, el resto de los pasos del proceso de refinamiento son los mismos que en el algoritmo de Ruppert, diferenciándose únicamente en el criterio de selección de punto.

En pseudocódigo, el algoritmo de Üngör queda de la siguiente manera:

```
alpha = valor_angulo_minimo_deseado
T = Triangulacion
while (existe triangulo con angulo menor a alpha en T)
    if (primera iteracion)
        T = eliminarAristasEncroached(T)
        T = procesarTriangulosAristaMasPequeñaRestringida(T)
    triangulo t = seleccionarTrianguloMenorAnguloMinimo(alpha,T)
    punto p = calcularOffCenter(t,T)
    triangulo s = encontrarTrianguloContiene(p,T)
    T = insertarPunto(p,s,T)
end while
eliminarAristasEncroached(T):
    while (T posee arista encroached E)
        T = insertarPuntoMedio(E,T)
    return T
procesarTriangulosAristaMasPequeñaRestringida(T):
    while (existe triangulo con arista mas pequeña restringida en
T)
        triangulo t = seleccionarTrianguloMenorAnguloMinimo(alpha,T)
        punto p = calcularOffCenter(t,T)
        triangulo s = encontrarTrianguloContiene(p,T)
        T = insertarPunto(p,s,T)
    return T
insertarPunto(p,s,T):
    if (insercion Delaunay de p dentro de s produce aristas
encroached [E1,...,En])
        foreach (E en [E1,...,En])
            q = PuntoMedio(E)
            T = insercionDelaunay(q,E,T)
    else
        T = insercionDelaunay(p,s,T)
```

3.2. Algoritmos Lepp-Delaunay: Selección de punto asociado al Lepp del triángulo a refinar

Existen varios algoritmos de refinamiento de triangulaciones que basan su criterio de selección de punto en el cálculo del Lepp asociado al triángulo que se desea refinar. En esta sección se describen los criterios de selección de punto asociados a los algoritmos Lepp-Delaunay Punto Medio y Lepp-Delaunay Centroide. Antes de presentar los detalles de cada uno, es necesario entender los conceptos de Lepp asociado a un triángulo y de arista terminal:

Definición: **[Lepp].** El camino de propagación por la arista más larga o LEPP (siglas en inglés para *Longest Edge Propagation Path*) asociado a un triángulo t_0 corresponde a una secuencia de triángulos $(t_0, ..., t_n)$ de la malla que cumplen con lo siguiente:

I. Para todo $i \in (0, ..., n-1)$, el triángulo t_{i+1} es vecino del triángulo de t_i a través de la arista más larga de t_i (ver Figura 3.6)

II. Si existe un triángulo t_n en la secuencia que es vecino de t_{n-1} por la arista más larga de t_{n-1}, pero a su vez el triángulo t_{n-1} es vecino de t_n a través de la arista más larga de t_n, entonces se dice que t_n es el triángulo terminal del Lepp.

III. Si en lugar de lo anterior, la arista más larga del triángulo t_n corresponde a un borde de la malla, también se considera que t_n es el triángulo terminal del Lepp.

IV. La arista más larga del triángulo terminal t_n se conoce como **arista terminal**.

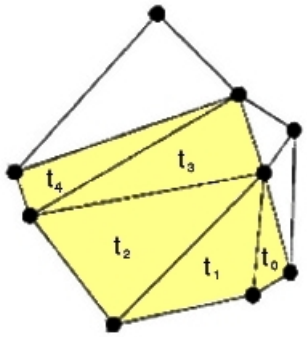

Figura 3.6: El Lepp del triángulo t_0 está formado por la sucesión de triángulos t_0 a t_4.

En este tipo de algoritmos de refinamiento de triangulaciones, el punto a insertar se calcula en base al triángulo terminal del $LEPP(t)$, donde t es el triángulo que se desea refinar.

3.2.1. Algoritmo Lepp-Delaunay punto medio: Selección del punto medio de la arista terminal

En el algoritmo Lepp-Delaunay punto medio [3, 6], no se requiere de procesar los triángulos en un orden particular. En cada iteración del proceso de refinamiento, si t es un triángulo de mala calidad, entonces se obtiene el Lepp L asociado a t y se realiza una inserción Delaunay del punto medio de la arista terminal de L. Si después de la inserción el triángulo t aún existe en la malla, se repite el mismo procedimiento cuantas veces sea necesario hasta eliminar t de la triangulación (ver Figura 3.7).

En pseudocódigo, el algoritmo Lepp-Delaunay punto medio queda de la siguiente manera:

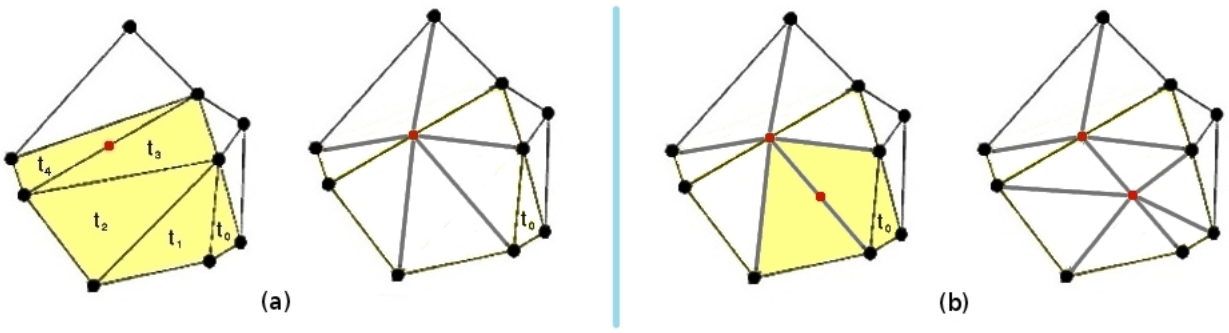

Figura 3.7: Refinamiento Lepp-Delaunay Punto Medio. Se realiza sucesivamente una inserción Delaunay en la arista terminal del Lepp.

```
alpha = valor_angulo_minimo_deseado
T = Triangulacion
while (existe triangulo con angulo menor a alpha en T)
    triangulo t = seleccionarTrianguloMalaCalidad(alpha,T)
    while (t existe en T)
        [t,t1,...,tn] = calcularLepp(t)
        punto p = calcularPuntoMedioAristaMasLarga(tn,T)
        T = insercionDelaunay(p,tn,T)
end while
```

Cabe destacar que en este algoritmo existe un raro caso especial que puede producir que el proceso de refinamiento entre en un ciclo infinito [12]. En la Figura 3.8 se muestra este caso, que ocurre cuando un triángulo MAM_1 similar al triángulo de mala calidad ABC puede ser obtenido mediante la inserción de los puntos M y M_1 en las aristas terminales BA y CA, siendo CA una arista restringida.

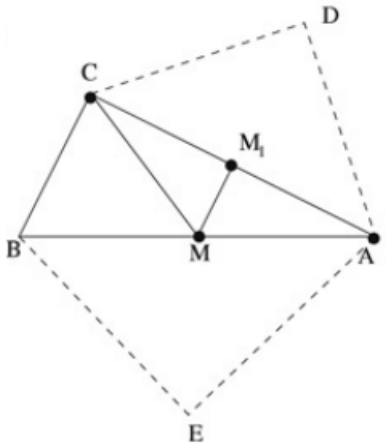

Figura 3.8: Caso especial de inserción en el algoritmo Lepp-Delaunay punto medio, que produce un ciclo infinito.

Para evitar este caso especial se introducen algunas condiciones adicionales. El pseudocódigo del algoritmo incluyendo estas condiciones de borde es el siguiente:

```
alpha = valor_angulo_minimo_deseado
T = Triangulacion
while (existe triangulo con angulo menor a alpha en T)
    triangulo t = seleccionarTrianguloMalaCalidad(alpha,T)
    while (t existe en T)
        [t,t1,...,tn] = calcularLepp(t)
        punto p = seleccionarPunto([t,t1,...,tn],T)
        T = insercionDelaunay(p,tn,T)
end while
seleccionarPunto(L,T): /* Lista de triangulos del Lepp */
    tn = trianguloTerminal(L)
    E = aristaMasLarga(tn)
    if((E es restringida o de borde) o
       (segunda arista mas larga de tn y
        segunda arista mas larga t(n-1)
        no son restringidas))
        p = calcularPuntoMedioAristaMasLarga(tn,T)
    else
        if((segunda arista mas larga de tn es restringida) o
           (t(n-1) comparte E con tn y
            segunda arista mas larga de t(n-1) es restringida))
            t = triangulo_que_cumple_la_condicion
            p = calcularPuntoMedioSegundaAristaMasLarga(t,T)
    return p
```

3.2.2. Algoritmo Lepp Delaunay centroide: Selección del centroide del cuadrilátero terminal

En el algoritmo Lepp-Delaunay centroide [5], se obtiene el Lepp L del triángulo t que se desea refinar, y el punto a insertar corresponde al centroide[1] del cuadrilátero convexo formado por los triángulos que comparten la arista terminal de L (ver Figura 3.9a). En caso de que la arista terminal de L sea una arista de borde, entonces se inserta el punto medio de ésta (ver Figura 3.9b).

En pseudocódigo, el algoritmo Lepp-Delaunay centroide queda de la siguiente manera:

[1]Centroide de un cuadrilátero corresponde al promedio de sus cuatro vértices.

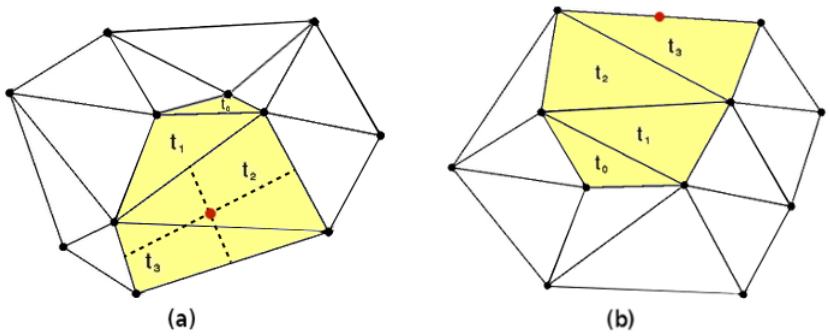

Figura 3.9: Refinamiento Lepp-Delaunay Centroide. En el ejemplo (a) se escoge como punto a insertar el centroide del cuadrilátero formado por t3 y t4. En el ejemplo (b) la arista terminal es un borde de malla, por lo que se selecciona el punto medio de ésta.

```
alpha = valor_angulo_minimo_deseado
T = Triangulacion
while (existe triangulo con angulo menor a alpha en T)
    triangulo t = seleccionarTrianguloMalaCalidad(alpha,T)
    while (t existe en T)
        [t,t1,...,tn] = calcularLepp(t)
        punto p = seleccionarPunto([t,t1,..tn],T)
        T = insercionDelaunay(p,tn,T)
end while
seleccionarPunto(L,T): /* Lista de triangulos del Lepp */
    tn = trianguloTerminal(L)
    E = aristaMasLarga(tn)
    if(E es arista de borde)
        p = calcularPuntoMedioAristaMasLarga(tn,T)
    else
        t(n-1) = vecinoPorAristaMasLarga(tn,T)
        p = calcularCentroideCuadrilateroTerminal(t(n-1),tn,T)
    return p
```

Capítulo 4

Software previo de refinamiento

4.1. Triangle

Triangle es un software de código abierto creado por Jonathan Richard Schewchuk, que permite generar triangulaciones de buena calidad (es decir, sin ángulos demasiado grandes o demasiado pequeños). El programa provee una interfaz de línea de comandos y la sintaxis para ejecutarlo es:

```
./triangle -<opciones> nombre_archivo
```

El archivo que *Triangle* recibe como entrada contiene un conjunto de puntos o una malla, y las opciones permiten especificar que clase de operación desea hacerse sobre los datos del archivo.

Los resultados de la operación quedan almacenados en un archivo de salida, el cual puede ser visualizado mediante una herramienta complementaria llamada *Showme* (la cual también es proveída por Schewchuk), que es un software para desplegar gráficamente el contenido de archivos que almacenan geometrías, en particular aquellas generadas por *Triangle*. Un ejemplo del uso de *Showme* puede verse en la Figura 4.1

Específicamente, *Triangle* permite ejecutar los siguientes algoritmos:

- Construir una triangulación de Delaunay a partir de un conjunto de vértices
- Construir una triangulación de Delaunay restringida a partir de un conjunto de vértices y aristas (PSLG)
- Construir una triangulación de calidad a partir de un conjunto de vértices y aristas, y un criterio de calidad que puede ser un ángulo mínimo deseado o un área de triángulo máxima deseada.

- Mejorar la calidad de una malla existente (refinar una triangulación)
- Otras operaciones como calcular la envolvente convexa de un conjunto de vértices, diagrama de Voronoi, y otros.

A la fecha en que se escribe esta memoria, *Triangle* se encuentra en su versión 1.6 y utiliza un algoritmo de refinamiento Delaunay muy optimizado basado en el trabajo de Paul Chew [13] (en versiones anteriores usaba el algoritmo de Ruppert, el cual aún está disponible en el software) e incluye un manejo mejorado de los dominios con ángulos muy pequeños [14] y una modificación sugerida por Alper Üngör que reduce el número de triángulos generados en la triangulación resultante.

La ventaja de *Triangle* por sobre otros software de refinamiento de triangulaciones es su eficiencia tanto en la rapidez de los algoritmos como en el uso de memoria, además de la robustez de su implementación. Como desventaja, a pesar de ser un software de código abierto, es muy difícil modificarlo o agregar cosas como, por ejemplo, otros algoritmos de refinamiento.

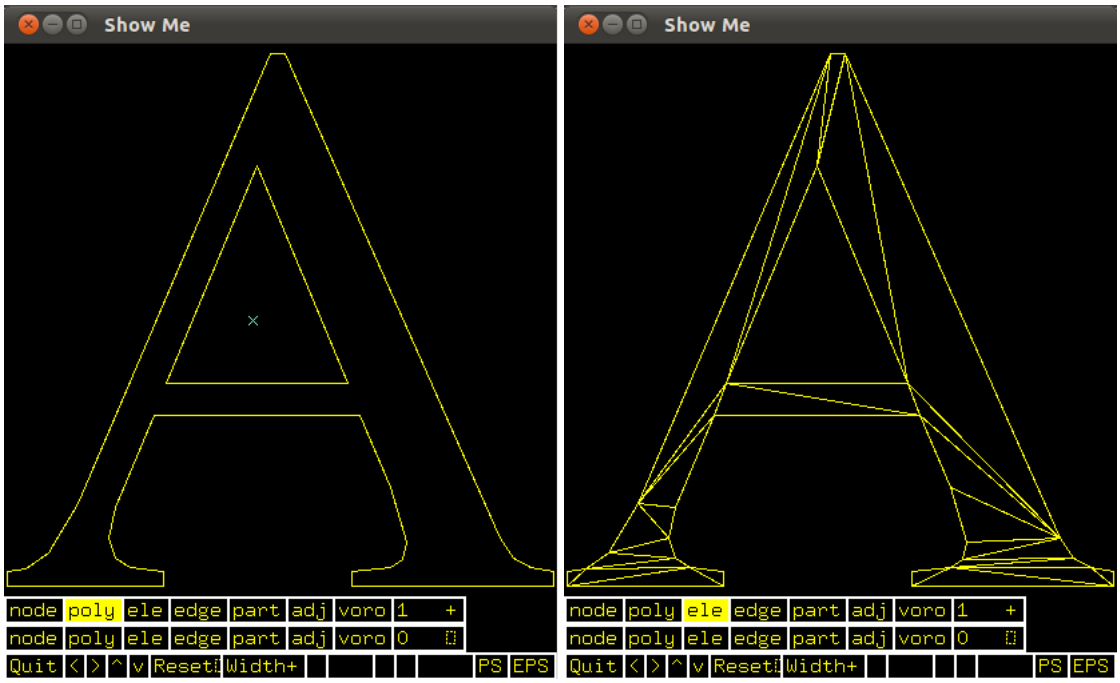

Figura 4.1: Visualización de polígono a triangularizar y triangulación resultante, usando *Showme*.

4.2. MeshSuite

MeshSuite es un prototipo de software, de código abierto, desarrollado por Álvaro Faúndez como parte de su memoria de título [8], el cual permite al usuario seleccionar distintos algoritmos de refinamiento Delaunay para mejorar la calidad de una malla existente. El

software provee una interfaz gráfica o GUI (ver Figura 4.2), a través de la cual el usuario puede cargar un archivo con la triangulación que desea refinar, definir el criterio de calidad (ángulo mínimo deseado) y además configurar la estrategia de refinamiento de mallas a utilizar, mediante la selección independiente de criterios de preprocesamiento de la malla, criterios de selección de triángulos a refinar (es decir, si los triángulos se procesan en algún orden específico), criterios de selección de puntos a insertar, y otras opciones. La evolución del proceso de refinamiento puede verse gráficamente de forma continua y también paso por paso. Los resultados de la ejecución, tales como número de triángulos resultantes o ángulo mínimo logrado, se muestran también en la interfaz.

Específicamente, *MeshSuite* permite ejecutar los siguientes algoritmos:

- Mejorar la calidad de una malla existente mediante la definición de los siguientes criterios:
  1. Ángulo mínimo deseado (umbral)
  2. Proceso pre-inserción: eliminación de aristas *encroached* o ninguno
  3. Priorización de triángulo a refinar, por ejemplo: ordenar de menor a mayor ángulo bajo el umbral, procesar primero triángulos con arista de menor tamaño y con ángulo bajo el umbral, etc.
  4. Selección de punto a insertar. Incluye inserción del circuncentro, inserción del Off-center, y algoritmos Lepp punto medio, centroide y bisección
  5. Algoritmo de inserción de punto (Intercambio de diagonales, cavidad o simple)
  6. Recuperación de aristas restringidas: por unión de aristas o completa

Otra característica importante de *MeshSuite* es el diseño de su código fuente, el cual fue concebido para que sea fácil añadir nuevos criterios y algoritmos. Además, el software permite al usuario elegir manualmente el triángulo a refinar y cambiar la configuración de la estrategia de refinamiento en cualquier iteración del proceso, excepto en el caso de ejecución continua del algoritmo.

La ventaja de *MeshSuite* por sobre otros software de refinamiento de triangulaciones es su flexibilidad para combinar diferentes criterios de preprocesamiento, priorización de triángulos, selección de puntos y tipo de inserción, lo que permite comparar de forma sencilla diferentes algoritmos. Por otro lado, tiene la desventaja de tener un sobrecosto asociado al dibujado de la malla en cada iteración, además de implementaciones poco eficientes de los algoritmos, que se detallan en el siguiente capítulo.

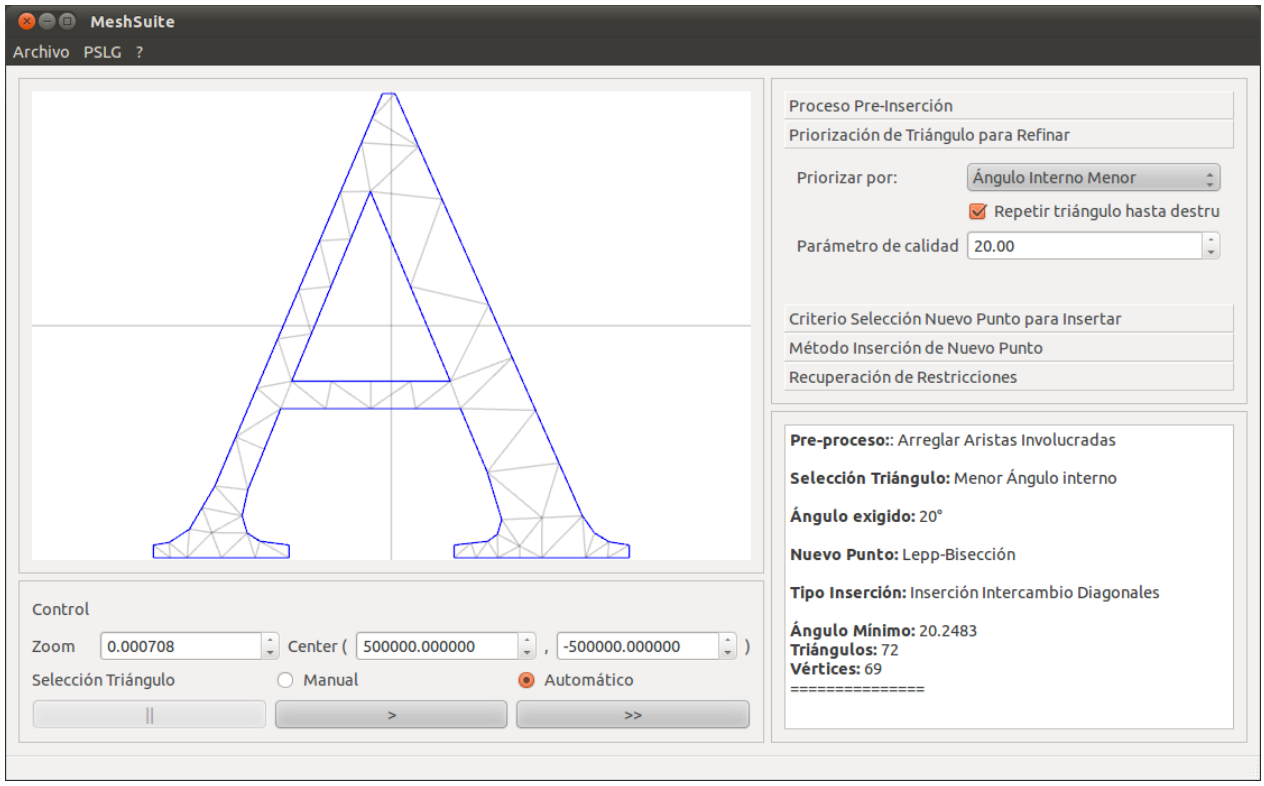

Figura 4.2: Interfaz gráfica de MeshSuite.

Capítulo 5

Diseño e implementación de software Compare2DMesh

En esta parte del informe se describe el diseño de la herramienta de comparación de algoritmos *Compare2DMesh*. En particular, se detallan las modificaciones y correcciones realizadas a *MeshSuite* con el fin de obtener un software adaptado a los objetivos de experimentación planteados en la memoria.

5.1. Creación de una cola de priorización de triángulos a refinar

La implementación de *MeshSuite* presenta un problema de eficiencia en el algoritmo que selecciona el siguiente triángulo a refinar de acuerdo al criterio de priorización definido por el usuario.

En términos generales, la selección ordenada de triángulos de acuerdo a un determinado criterio, consiste en recorrer toda la malla hasta encontrar un triángulo que cumpla con las características deseadas (menor ángulo mínimo, menor arista del borde, etc.). Puesto que esta búsqueda se repite en cada iteración del proceso de refinamiento, la estrategia descrita resulta muy costosa en tiempo de ejecución.

Para mejorar la eficiencia en la selección del triángulo a refinar, el software *Compare2DMesh* incluye una interfaz que representa una cola con punteros a todos los triángulos que van a ser refinados, es decir, aquellos triángulos cuyo ángulo mínimo es inferior al umbral definido por el usuario. La interfaz recibe el nombre de *QueueOfTrianglesToProcess* y para cada criterio de priorización debe implementarse una clase que herede de ella. De este modo, el algoritmo de refinamiento selecciona triángulos de la malla sin que le importe qué tipo de cola se está utilizando. Para procesar los triángulos sin priorizar se implementa usando una cola simple, y para procesar los triángulos en un cierto orden, se implementa usando una cola de prioridad.

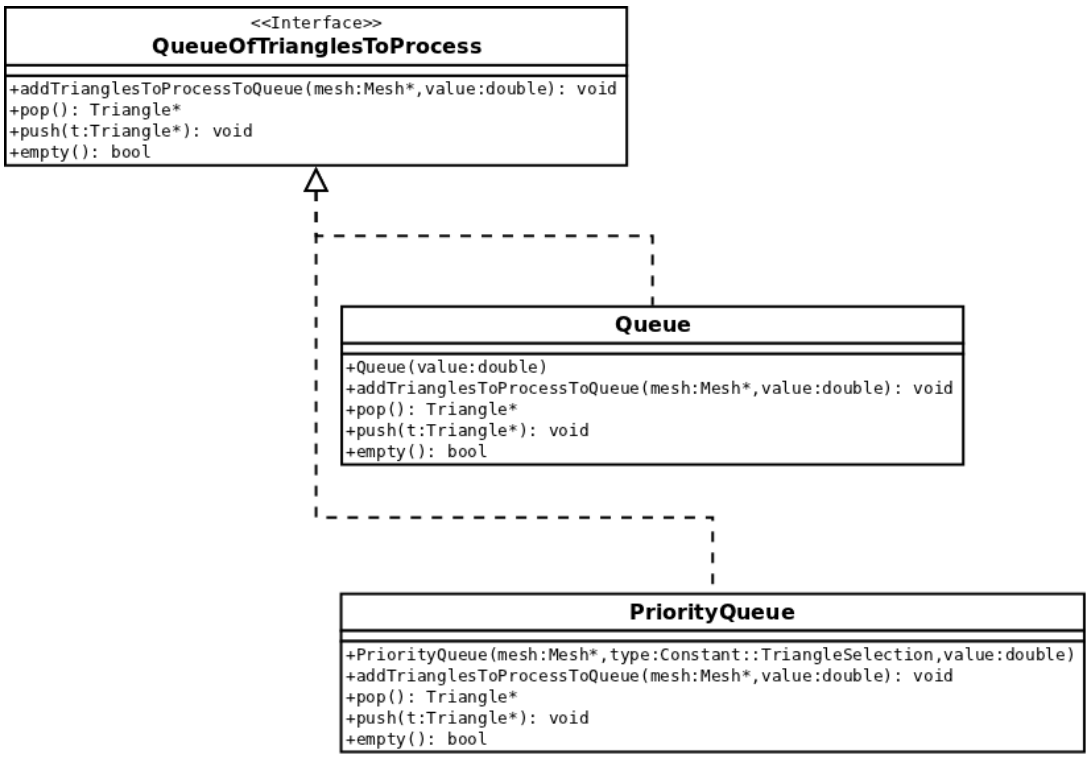

Figura 5.1: Diagrama de clases de la implementación de la interfaz *QueueOfTrianglesToProcess*.

En el diagrama de la Figura 5.1 se muestran las clases creadas para implementar *QueueOfTrianglesToProcess*: la clase *Queue* corresponde a una cola simple, en donde los elementos conservan el orden en que fueron agregados; la clase *PriorityQueue* es, como su nombre dice, una cola de prioridad que además puede ser inicializada con diferentes comparadores de acuerdo al criterio de priorización que se va a utilizar. El siguiente pseudocódigo muestra cómo se definen los diferentes operadores de comparación en la *PriorityQueue*.

```
/* Inicializacion de comparador */
DefinirComparador(int tipoComparador):
    miComparador->tipoComparador = tipoComparador

/* Definicion de comparador (ejemplo) */
booleano comparar()(Triagulo t1, Triangulo t2):
    switch(miComparador->tipoComparador)
    case MENOR_ANGULO:
        return anguloMinimo(t1) > anguloMinimo(t2)
    case MENOR_ARISTA:
        return menorArista(t1) > menorArista(t2)
    case MENOR_CIRCUNCIRCULO:
        return radioCircuncirculo(t1) > radioCircuncirculo(t2)
```

Una vez implementadas las colas descritas anteriormente, el proceso de refinamiento queda

modificado de la siguiente forma: Antes de la primera inserción, se debe llenar *QueueOfTrianglesToProcess* con punteros a todos aquellos triángulos de la malla que se van a refinar. Luego, en cada iteración del refinamiento, la selección del siguiente triángulo a procesar consiste simplemente en sacar el primer elemento de la cola.

Durante el proceso de refinamiento, en cada inserción de un nuevo punto, se crean y eliminan triángulos de la malla, algunos de los nuevos triángulos creados podrían eventualmente tener mala calidad, lo mismo puede ocurrir con los triángulos eliminados a causa de la inserción. Como la cola de procesamiento debe contener todos los triángulos de mala calidad existentes en la malla, es necesario actualizarla en cada inserción para reflejar estos cambios.

La inserción de un punto en la malla genera nuevos triángulos, los cuales pueden eventualmente ser de mala calidad. Si esto ocurre, dichos triángulos deben ser agregados a *QueueOfTrianglesToProcess*.

La inserción de un nuevo punto en la malla produce también la eliminación de triángulos. Cuando alguno de los triángulos a eliminar es de mala calidad, debe borrarse también de la cola de procesamiento. Sin embargo, en una cola solo se puede acceder al primer y al último elemento, por lo que no es posible eliminar directamente un elemento que se encuentre entremedio. Además, si se eliminan de la malla triángulos que también se encuentran en la cola de procesamiento se producirá un error, pues la cola maneja punteros a los triángulos: si se elimina el objeto apuntado por un puntero que aún está en uso, el programa puede caerse o tener comportamientos inesperados. Para resolver esto, se realiza una eliminación lógica de los triángulos y estos solo se eliminan realmente de la memoria al momento de ser extraídos de la cola. Para ello se definen tres estados posibles para un triángulo:

- *ALIVE*: Un triángulo tiene este estado si se encuentra en la malla pero no ha sido ingresado en la cola de procesamiento.
- *IN_DEATH_ROW*: Un triángulo tiene este estado si se encuentra en la malla y también en la cola de procesamiento. El nombre del estado se debe a que los triángulos que se encuentran en la cola necesariamente serán eliminados en algún momento de la malla, pues son de mala calidad.
- *DEAD*: Un triángulo tiene este estado si se ha eliminado lógicamente de la malla.

El proceso de eliminación lógica de triángulos queda entonces como sigue:

I. Inicialmente, todos los triángulos de la malla tienen estado *ALIVE*.

II. El estado de un triángulo cambia a *IN_DEATH_ROW* cuando es agregado a la estructura *QueueOfTrianglesToProcess*.

III. Al refinar un triángulo de la cola de procesamiento, éste es eliminado de la malla y reemplazado por nuevos triángulos. Como el triángulo ya ha salido de la cola al momento de ser eliminado de la triangulación, entonces puede ser destruido completamente de la memoria.

IV. Generalmente, al refinar un triángulo también se eliminan otros, por ejemplo al momento de *Delaunizar* la malla, lo que implica que durante el proceso de refinamiento se eliminan triángulos que no necesariamente son de mala calidad y, por ende, no están en la cola de triángulos a procesar. Es posible saber que un triángulo de la malla no se encuentra en la cola porque en ese caso su estado es *ALIVE*. Por lo tanto, puede ser destruido completamente de la memoria sin problemas

V. Al *Delaunizar* la malla, también pueden eliminarse triángulos de mala calidad que aún no han sido procesados. Estos triángulos están en la cola de triángulos a procesar y tienen estado *IN_DEATH_ROW*. Para eliminar un triángulo que cumple esta condición, se realiza una eliminación lógica que consiste en quitar de la malla el puntero a dicho triángulo y cambiar su estado a *DEAD*.

VI. Como consecuencia del caso descrito en el punto anterior, la cola de procesamiento contiene triángulos con estados *IN_DEATH_ROW* y *DEAD*. Si al momento de extraer de la cola el siguiente triángulo a procesar, este tiene estado DEAD, entonces debe eliminarse físicamente de la memoria y extraer el siguiente triángulo. En caso contrario el triángulo tiene estado *IN_DEATH_ROW* y es refinado. Su posterior eliminación está determinada por el punto III.

De este modo, se mantiene consistente al objeto *QueueOfTrianglesToProcess* durante todo el proceso de refinamiento de la malla.

El procedimiento descrito en esta sección permite procesar los triángulos de manera más eficiente que la utilizada en *MeshSuite*, puesto que sacar el primer elemento de una cola toma tiempo constante en una cola simple, y tiempo logarítmico sobre el tamaño de la cola en una de prioridad, por lo que es mucho más rápido que recorrer toda la malla, en cada iteración, para buscar el siguiente triángulo de mala calidad (tiempo lineal sobre el tamaño de la malla).

5.2. Creación de estructura de datos para representar aristas restringidas

En la implementación de *MeshSuite* no existe una estructura de datos explícita para representar las aristas de una triangulación. Las aristas restringidas se guardan en una lista de pares de vértices, donde cada par de puntos representa los extremos de la arista. Estos pares de puntos no están asociados a los triángulos que los contienen, solamente se guarda dentro del triángulo un arreglo de booleanos que indica si cada arista es restringida o no, pero esta información no es aprovechada en ninguna etapa del proceso de refinamiento.

Durante el proceso de refinamiento en *MeshSuite*, solo se consideran como aristas restringidas aquellas que se encuentran en el borde de la malla; una arista es de borde si el triángulo que la contiene no posee vecino a través de ella. El resto de las aristas restringidas que están indicadas en la malla de entrada se agregan a la triangulación mediante un algoritmo de recuperación de aristas, al final del proceso de refinamiento.

En el software *Compare2DMesh*, en cambio, se crea una estructura de datos explícita para representar las aristas restringidas de la triangulación, la cual recibe el nombre de *RestrictedEdge* y contiene punteros a los vértices extremos de la arista y a los triángulos que la contienen. La creación de esta nueva estructura de datos introduce una modificación en la representación de la malla de triángulos: la lista de pares de vértices que representaba el conjunto de aristas restringidas de la triangulación, es ahora reemplazada por una lista que contiene punteros a objetos de la clase *RestrictedEdge.* Se modifica también la estructura de datos de triángulo, de modo que a cada vértice se le asocia el id de su arista restringida opuesta (si la arista no es restringida, se le asocia -1).

La finalidad de crear esta estructura es proveer información explícita que será aprovechada en el preprocesamiento de aristas *encroached*, algoritmos de selección y también de inserción de puntos en la malla, eliminando la necesidad de un algoritmo de recuperación de aristas restringidas y mejorando la eficiencia del proceso de refinamiento.

5.3. Creación de una cola de preprocesamiento de aristas restringidas

En *MeshSuite*, el proceso de eliminación de aristas *encroached* consiste en recorrer la malla en búsqueda de una arista con las siguientes características: el triángulo que la contiene no posee vecino a través de dicha arista (la arista pertenece al borde de la triangulación); el círculo diametral de la arista contiene un vértice de la malla en su interior (la arista es *encroached*). Luego, se realiza una inserción Delaunay del punto medio de la arista encontrada. El mismo procedimiento se repite hasta que ya no queden más aristas de borde *encroached* en la triangulación. Esta implementación presenta dos problemas: para eliminar dichas aristas se recorre la malla tantas veces como aristas haya que procesar, lo que es poco eficiente, y solo se están procesando las aristas del borde.

Para mejorar el procedimiento descrito, el software *Compare2DMesh* incluye una estructura para representar las aristas restringidas, la cual posee punteros a los triángulos que contienen a dicha arista. Esta estructura de datos está representada por la clase *RestrictedEdge* y su descripción se encuentra en la sección 5.2. Los triángulos de la malla guardan, a su vez, los id de sus respectivas aristas restringidas. También se crea una estructura, de nombre de *QueueOfEncroachedEdges*, que representa una cola de aristas restringidas que van a ser procesadas, en otras palabras, una cola con las aristas *encroached* de la triangulación. La implementación de *QueueOfEncroachedEdges* consiste en una cola simple que almacena punteros a objetos de la clase *RestrictedEdge*.

Una vez implementada la cola descrita anteriormente, el proceso de refinamiento queda modificado de la siguiente forma: Antes de la primera inserción, se debe llenar *QueueOfEncroachedEdges* con punteros a todas las aristas *encroached* de la malla, para ello es necesario recorrer la lista de todas las aristas restringidas de la triangulación, identificando cuales de ellas son *encroached* y agregándolas en *QueueOfEncroachedEdges*. Luego, se procesan todas las aristas de la cola, realizando una inserción Delaunay del punto medio de cada una de

ellas. Una vez finalizado este proceso, comienza el refinamiento de la malla.

Durante el proceso de refinamiento de la triangulación, en cada inserción de un nuevo punto se crean y eliminan triángulos de la malla. Algunos de los nuevos triángulos creados podrían eventualmente producir una arista *encroached*. Si esto ocurre, dichas aristas deben ser agregadas a *QueueOfEncroachedEdges*, manteniendo consistente la cola durante todo el proceso de refinamiento. Luego, antes de cada inserción, se procesan las aristas de la cola.

La nueva forma de procesar las aristas *encroached* es más eficiente que la implementada en *MeshSuite*, puesto que para identificar todas las aristas a procesar se debe recorrer la malla solo una vez y, posteriormente, para procesar las aristas solo basta con sacarlas una a una de la cola, donde el tiempo para sacar cada elemento de la cola es constante. Esto es mucho más rápido que recorrer la malla completa cada vez que se quiere procesar una arista *enroached*, y tiene la ventaja de que se procesan todas las aristas, no solo las de borde.

5.4. Modificaciones a algoritmo de inserción Delaunay de puntos

La inclusión de colas de procesamiento de aristas restringidas y de triángulos en el proceso de refinamiento (secciones 5.3 y 5.1 respectivamente), necesariamente requiere de una modificación del algoritmo de inserción Delaunay de puntos implementado en *MeshSuite*, debido a que cada vez que se inserta un nuevo vértice en la malla, dichas colas deben actualizarse. En el software *Compare2DMesh*, se incluye *QueueOfTrianglesToProcess* y *QueueOfEncroachedEdges* entre los parámetros del algoritmo de inserción de vértices. También se modifica el proceso de inserción para mantener actualizadas las aristas restringidas, las cuales ahora están representadas por la estructura *RestrictedEdges*. A continuación se describe el nuevo algoritmo:

Cuando se va a agregar un nuevo vértice en la malla, primero se debe buscar el triángulo que lo contiene. Una vez encontrado este triángulo, se realiza el siguiente procedimiento:

I. Si el vértice P cae dentro del triángulo, se unen sus vértices a P, creando así tres nuevos triángulos. Si, por el contrario, P cae sobre una arista, se une P a los vértices de los triángulos adyacentes a dicha arista, creando así cuatro nuevos triángulos (o dos, si la arista pertenece al borde de la malla).

II. Se actualiza la asignación triángulo-arista para asociar las aristas restringidas del triángulo original a los nuevos triángulos creados, según corresponda. En el caso de inserción del vértice en una arista restringida o de borde, se realiza un paso adicional que consiste en dividir la arista en dos, lo que se consigue eliminando la arista original y creando dos nuevas aristas, las cuales deben asociarse a los nuevos triángulos creados según corresponda, y también deben ser agregadas a la lista de aristas restringidas de la malla.

III. Se verifica si alguna de las aristas restringidas es *encroached*, en caso afirmativo, se agrega dicha arista a la cola de aristas *encroached*.

IV. Se verifica si alguno de los nuevos triángulos creados es de mala calidad (posee un ángulo menor al umbral deseado), en caso afirmativo, se agrega dicho triángulo a la cola de triángulos a procesar.

V. Finalmente, se realiza intercambio de diagonales para obtener nuevamente una triangulación de Delaunay tras la inserción del nuevo punto. En cada intercambio, se eliminan dos triángulos y se crean dos nuevos, se actualiza la asignación arista restringida-triángulo y se repiten los pasos III y IV. Además, si un par de triángulos comparte una arista restringida, el intercambio de diagonales no se realiza para ese par.

En el proceso descrito no se menciona la eliminación de elementos de las colas de procesamiento de aristas restringidas y triángulos. Esto se debe a que cuando la malla elimina sus elementos, se encarga también de marcar como eliminados dichos objetos en las colas de aristas restringidas y de triángulos según corresponda. Ya que no es posible acceder a los elementos intermedios de una cola, es necesario eliminarlos lógicamente y cuando se extraen del principio de la cola son realmente borrados de la memoria. El detalle de este proceso para el caso de la cola de procesamiento de triángulos es descrito en la sección 5.1. El mismo modelo es utilizado para la eliminación de aristas restringidas.

Esta versión modificada del algoritmo de inserción Delaunay permite mantener las colas de procesamiento de aristas y triángulos (*QueueOfEncroachedEdges* y *QueueOfTrianglesToProcess*), la lista interna de aristas restringidas de la malla y las relaciones entre aristas restringidas y triángulos actualizadas a lo largo de todo el proceso de refinamiento.

5.5. Selección de punto: Algoritmo de búsqueda de triángulo que contiene el punto a insertar

En *MeshSuite*, la implementación de los algoritmos de selección de punto de Ruppert y Üngör, presenta problemas de eficiencia en la búsqueda del triángulo que contiene al punto a insertar. Dicha búsqueda consiste en recorrer toda la malla hasta encontrar el triángulo que contiene al punto. Esta estrategia es poco eficiente, debido a que la búsqueda se repite en cada iteración del proceso de refinamiento.

Para mejorar esto, en *Compare2DMesh* se implementa un algoritmo conocido como "*Straight Walk*" [15], el cual consiste en escoger un vértice Q de la triangulación y recorrer todos los triángulos que son intersectados por el segmento de recta $\overline{QP}$, donde P corresponde al punto que se desea insertar en la malla. Para ello, se utilizan las relaciones de adyacencia entre los triángulos. El recorrido termina al encontrar un triángulo que contenga a P.

En términos más precisos, el algoritmo realiza un paso de inicialización, el cual consiste en recorrer todos los triángulos que contienen el vértice Q hasta encontrar uno que intersecte la

línea $\overline{QP}$, para lo cual a cada uno de los triángulos visitados se le aplica un test de orientación. Luego, en cada iteración del recorrido, se utiliza el test de orientación sobre el triángulo actual para determinar cual de sus aristas es intersectada por $\overline{QP}$ o si el triángulo contiene al punto P. En el primer caso, se selecciona el triángulo adyacente a la arista encontrada y se repite el proceso. En el segundo caso, el algoritmo termina.

Para los algoritmos de Ruppert y Üngör, el paso de inicialización no es necesario. La búsqueda se inicia con el triángulo que va a ser refinado, el resto del algoritmo "*Straight Walk*" se mantiene igual.

El siguiente pseudocódigo representa la nueva implementación de búsqueda del triángulo que contiene el punto seleccionado para insertar en la malla:

```
BuscarTrianguloContiene(t,p): /* t = triangulo a refinar */
    currentTriangle = t
    while (currentTriangle no contiene a p)
        foreach( arista E[i] en currentTriangle )
            if (orientacion(p, E[i]) < 0 )
                currentTriangle = currentTriangle.vecino[i]
                break
    return currentTriangle
```

5.6. Rediseño del algoritmo general de refinamiento

5.6.1. Diseño del proceso de refinamiento de MeshSuite

En la implementación de *MeshSuite*, se utiliza el patrón estrategia para dividir el refinamiento de mallas en 5 pasos generales:

I. Proceso pre-inserción: antes de realizar una nueva inserción, se realizan algunas modificaciones a la malla (Ejemplo: eliminación de aristas *encroached*).

II. Criterio de selección de triángulo para refinar: define el orden en que se procesan los triángulos durante el refinamiento. Usando este criterio se selecciona el siguiente triángulo a refinar.

III. Criterio de selección de nuevo punto a insertar: usando este criterio se calcula el punto a insertar en la malla para mejorar la calidad del triángulo seleccionado.

IV. Método de inserción de nuevo punto a la malla: corresponde al algoritmo de inserción (Ejemplo: Intercambio de diagonales, cavidad). Utilizando el método seleccionado se inserta el nuevo vértice en la triangulación.

v. Postproceso: algoritmo de recuperación de aristas restringidas a utilizar tras terminar el refinamiento de la triangulación. Esto es requerido porque el software no considera las aristas restringidas interiores de la malla durante su refinamiento, por lo que debe incluirlas al finalizar el proceso.

Los pasos I, II, III y IV son repetidos mientras queden dentro de la malla triángulos con ángulo menor al criterio de calidad definido por el usuario. El paso V solo se realiza una vez, al final del proceso de refinamiento.

El objetivo de separar el refinamiento en 5 pasos generales es poder representar de forma abstracta cualquier algoritmo de refinamiento Delaunay de triangulaciones. Para ello, se definen interfaces para cada una de las etapas del proceso (ver diagrama de la Figura 5.2) y cada algoritmo de refinamiento implementa dichas interfaces (Figura 5.3). El uso del patrón estrategia permite además mezclar criterios. Por ejemplo, usar un criterio de selección de triángulos que procesa primero los triángulos con arista más pequeña de borde (Üngör), con el criterio de selección de punto Lepp-Delaunay centroide. Además, agregar un nuevo algoritmo al software es sencillo pues basta con implementar nuevas clases que hereden de las interfaces que representan cada una de las etapas definidas.

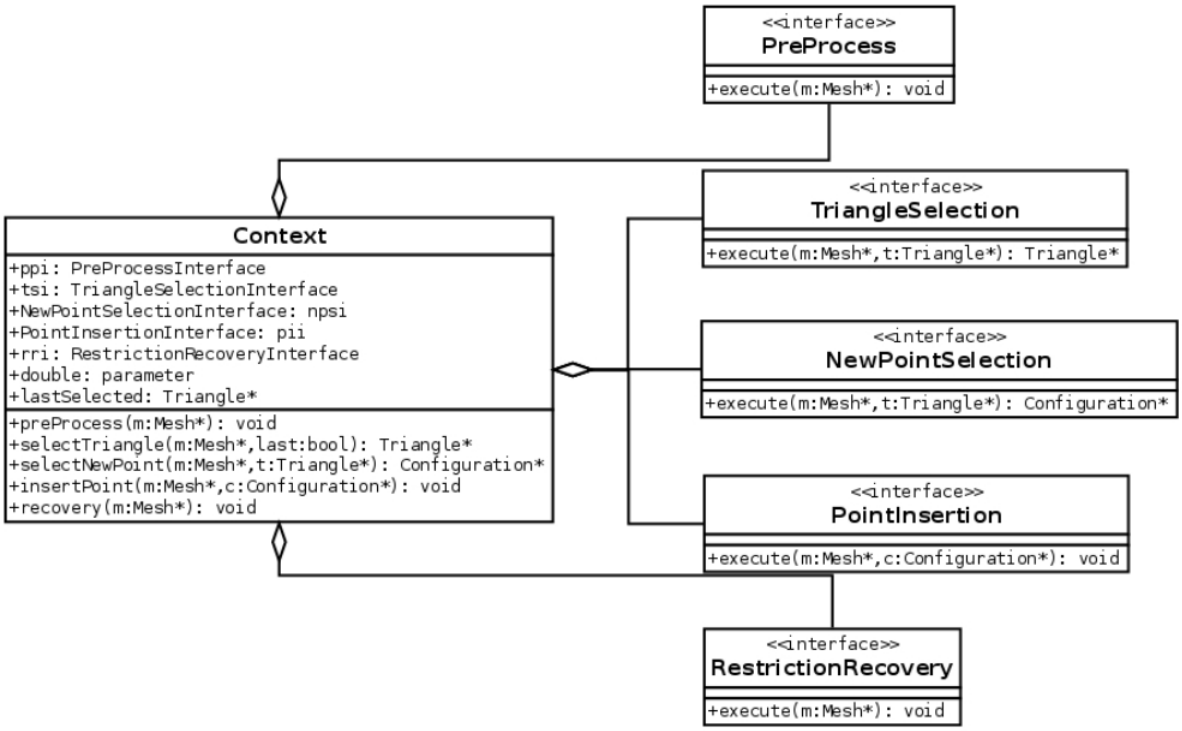

Figura 5.2: Diagrama de clases parcial con uso del patrón estrategia para el proceso de refinamiento. Cada interfaz debe ser implementada con algún algoritmo concreto.

Este diseño en base al patrón estrategia se complementa con el patrón fábrica, el cual permite instanciar cualquiera de las distintas implementaciones de las etapas del refinamiento en tiempo de ejecución. Por ejemplo, la etapa de selección de punto está representada por la interfaz *NewPointMethod*, la cual posee un método abstracto llamado *getConfiguration* que se

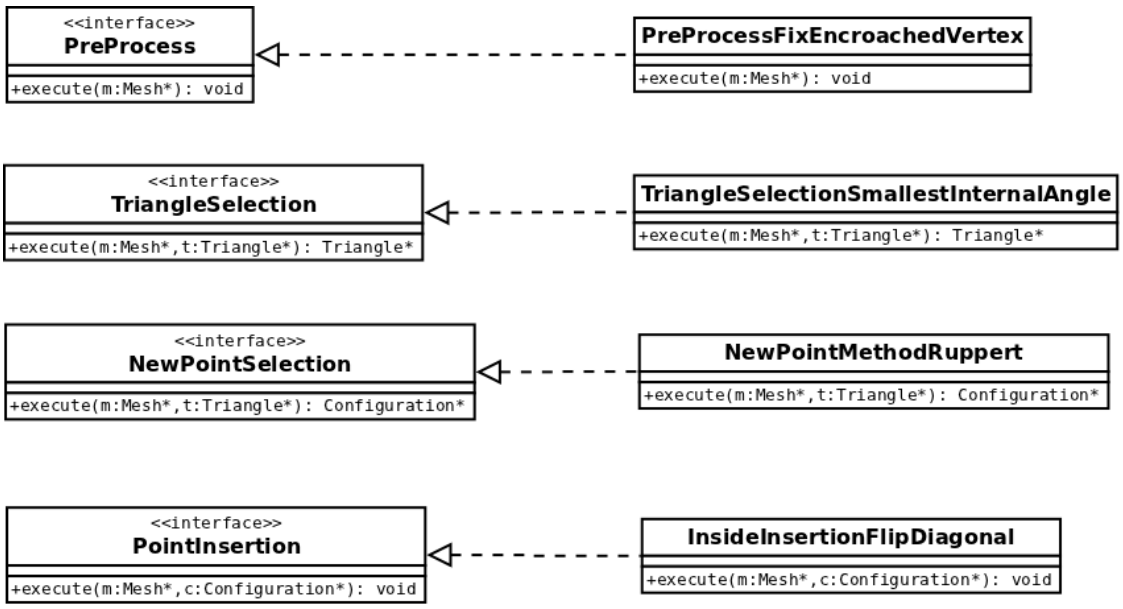

Figura 5.3: Diagrama parcial de clases que ejemplifica como se construye una implementación particular del proceso abstracto de refinamiento de triangulaciones. Por ejemplo, el algoritmo de Ruppert.

encarga de calcular el punto a insertar en la malla. Mediante la clase *FactoryNewPointMethod*, se crea un objeto que corresponde a una implementación específica de *NewPointMethod*, como por ejemplo, una clase que contiene un método que calcula el circuncentro del triángulo que será refinado (selección de punto basada en el algoritmo de Ruppert). El diagrama de clases de la Figura 5.4 muestra el uso del patrón fábrica en el proceso de refinamiento.

5.6.2. Nuevo diseño del proceso de refinamiento en Compare2DMesh

La nueva implementación del proceso de refinamiento incluye el manejo de las colas de procesamiento de aristas restringidas y de triángulos (secciones 5.3 y 5.1 respectivamente), creadas para optimizar las etapas de eliminación de aristas *encroached* y selección de triángulo a refinar. Para ello ha sido necesario modificar los pasos generales del proceso de refinamiento.

MeshSuite requiere de algoritmos de recuperación de aristas restringidas debido a que en el refinamiento no dispone de suficiente información acerca de las aristas restringidas interiores de la triangulación, por lo que la malla resultante se construye sin considerar dichas aristas y éstas son incluidas después de terminar de refinar. En la nueva implementación, se ha eliminado la etapa de recuperación de aristas restringidas. Esto debido a que se agregó al software una estructura de datos para representar las aristas restringidas de forma explícita (sección 5.2) y se modificó el algoritmo de inserción de puntos de manera que se mantengan las aristas restringidas de la malla, íntegras o divididas en segmentos, durante todo el proceso de refinamiento.

Este nuevo proceso general de refinamiento se divide en las siguientes etapas:

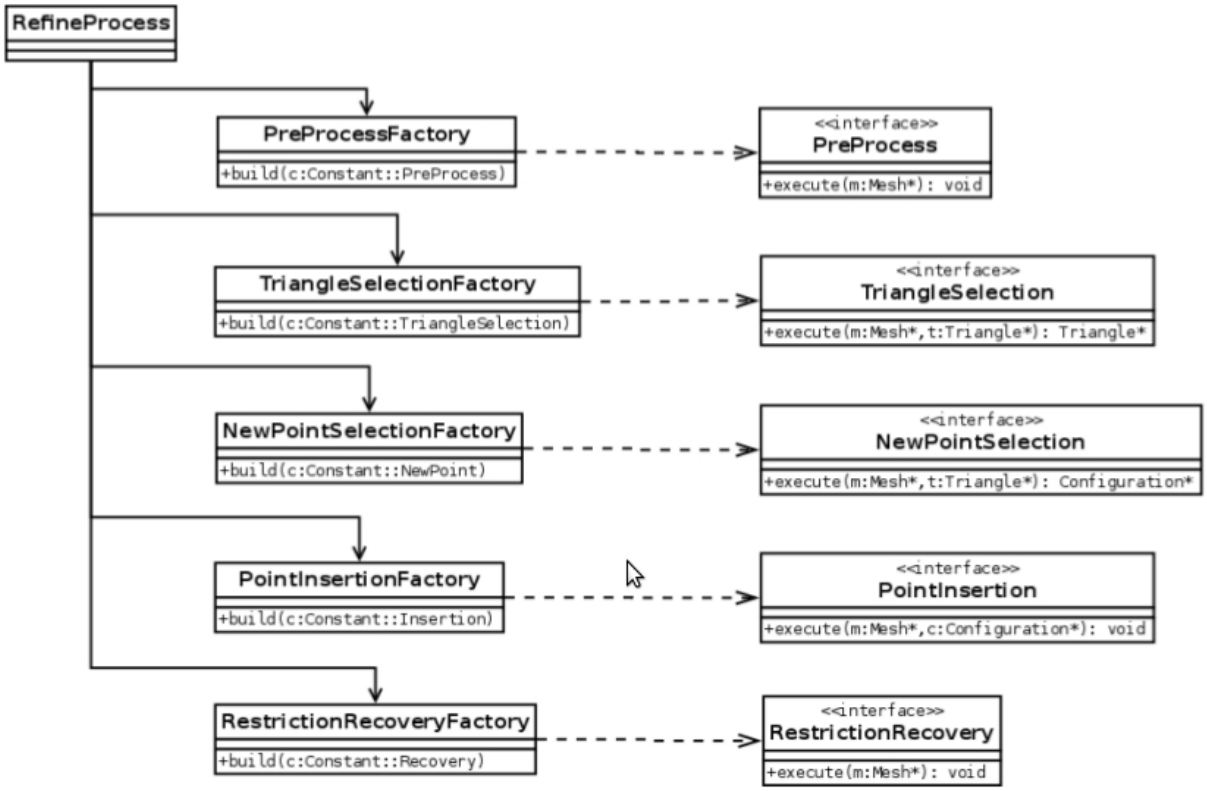

Figura 5.4: Diagrama de clases parcial con uso de patrón fábrica para la creación de algoritmos.

I. Inicialización proceso pre-inserción: obtener todas las aristas *encroached* de la malla y agregarlas a la cola de procesamiento de aristas *encroached*. Este paso solo se realiza en la primera iteración, en el resto del proceso la cola simplemente es actualizada.

II. Proceso pre-inserción: antes de realizar una nueva inserción, se eliminan todas las aristas *encroached* de la triangulación.

III. Inicialización criterio de selección de triángulo: obtener los triángulos que se van a refinar y construir una cola de priorización de acuerdo al criterio de selección escogido. Este paso solo se realiza en la primera iteración, en el resto del proceso la cola simplemente es actualizada.

IV. Selección de triángulo a refinar: obtiene primer elemento de la cola de procesamiento de triángulos.

V. Criterio de selección de nuevo punto a insertar: usando este criterio se calcula el punto a insertar en la malla para mejorar la calidad del triángulo seleccionado.

VI. Inserción de nuevo punto a la malla: inserta el nuevo vértice en la triangulación mediante el método de inserción Delaunay por intercambio de diagonales.

Los pasos II, IV, V y VI son repetidos mientras queden dentro de la malla triángulos con ángulo menor al criterio de calidad definido por el usuario.

Los patrones de diseño utilizados en *MeshSuite* se mantienen en la nueva implementación. De este modo, se conservan las ventajas de flexibilidad para configurar diferentes estrategias de refinamiento y también la facilidad para agregar nuevos algoritmos al software. Por otra parte, la nueva implementación procesa las triangulaciones de forma más eficiente, pues se elimina del proceso de refinamiento la necesidad de recorrer la malla completa cada vez que se va a eliminar una arista *encroached*, cada vez que se debe seleccionar el siguiente triángulo a procesar y cada vez que se debe buscar el triángulo que contiene el nuevo punto a insertar en la triangulación. En el nuevo proceso de refinamiento, se recorre la malla para llenar la cola de aristas *encroached* (*QueueOfEncroachedEdges*) y luego para llenar la cola de triángulos a procesar (*QueueOfTrianglesToProcess*), solo en la primera iteración. Por otro lado, la búsqueda del triángulo que contiene el nuevo punto a insertar recorre solo una porción de la malla, utilizando el algoritmo descrito en la sección 5.5. Por lo tanto, mediante las modificaciones descritas en esta sección, se consigue un algoritmo general de refinamiento flexible y eficiente a la vez.

5.7. Inicialización de la malla

Al igual que en *MeshSuite*, el software *Compare2DMesh* carga los datos de la malla inicial desde un archivo de entrada con las siguientes características:

I. Las primeras líneas corresponden a los vértices de la triangulación y son de la forma:

```
v coordenadaX coordenadaY
```

II. Las siguientes líneas corresponden a los triángulos formados por los vértices anteriores y son de la forma:

```
f idVertice1 idVertice2 idVertice3
```

III. Las últimas líneas corresponden a las aristas restringidas de la triangulación y son de la forma:

```
r idVertice1 idVertice2
```

Las aristas restringidas de borde pueden omitirse en el archivo.

Una vez cargados los vértices, triángulos y aristas restringidas se procede a la asignación de vecinos de cada triángulo.

En *MeshSuite*, por cada triángulo de la malla se recorre toda la malla en busca de sus vecinos y, cuando estos son encontrados, se realizan las asignaciones correspondientes.

En pseudocódigo, el algoritmo de inicialización de la malla es el siguiente:

```
FH = open(file)
while (linea = readLine(FH))
    if (linea.startsWith('v'))
        malla.crearYAgregarVertice(coordenadaX, coordenadaY)
    if (linea.startsWith('f'))
        malla.crearYAgregarTriangulo(malla.vertices(idVertice1),
                                     malla.vertices(idVertice2),
                                     malla.vertices(idVertice3))
foreach (Triangulo T perteneciente a malla.triangulos)
    foreach (Triangulo S perteneciente a malla.triangulos)
        if (T comparte dos vertices con S)
            T.asignarVecino(S)
            S.asignarVecino(T)
close(FH)
```

Dado que el software *Compare2DMesh* posee estructuras explícitas para las aristas restringidas, a este procedimiento se le agrega una etapa de asignación de aristas restringidas a cada triángulo y viceversa. Para ello, por cada triángulo se recorre la lista de aristas restringidas y cuando alguna de ellas pertenece al triángulo, se realizan las asignaciones correspondientes.

Además, *Compare2DMesh* mejora la eficiencia del procedimiento de asignación de vecinos(*), mediante el siguiente algoritmo:

I. En la etapa de construcción de los triángulos a partir del archivo de entrada, crear una tabla que asocia a cada vértice de la malla los triángulos que lo contienen. Debido a que un triángulo posee tres vértices, cada triángulo de la malla está repetido tres veces en la tabla.

II. Luego, para buscar los vecinos de un determinado triángulo t, se obtiene de la tabla todos los triángulos que comparten algún vértice con t (los cuales son posibles candidatos a ser vecinos de t) y se guardan en un conjunto C que acepta elementos repetidos. Puesto que dos triángulos son vecinos si comparten dos vértices, basta contar cuantas veces aparece cada triángulo en el conjunto creado. Sea s un triángulo que pertenece a C:

   a) si s aparece solo una vez en C, significa que solo comparte un vértice con t, por lo que t y s no son vecinos.

   b) si s aparece dos veces en C, significa que comparte dos vértices con t, por lo tanto t y s **son vecinos** y debe hacerse la respectiva asignación de vecindad.

   c) si s aparece tres veces en C, significa que comparte tres vértices con t, por lo que se trata del mismo triángulo t.

(*) Es importante hacer notar que, aunque este algoritmo es más eficiente que el de *MeshSuite* en el caso general, tiene un peor caso muy malo.

Para una malla de tamaño n, sea t un triángulo y k_i el numero de triángulos que comparten con t el vértice v_i. El algoritmo utilizado en *Compare2DMesh* realiza $(k_0 + k_1 + k_2)$ comparaciones para encontrar a los vecinos de t. En el peor caso, hay algún vértice de t que es compartido por muchos triángulos, de modo que $(k_0 + k_1 + k_2) \sim n$, lo que implica que se realizan $O(n)$ comparaciones para encontrar a los vecinos de t, y cuando este caso ocurre, se tiene que para todos los triángulos que comparten dicho vértice se realizan $O(n)$ comparaciones. Esto quiere decir que en el peor caso la asignación de vecinos de toda la malla requiere $O(n^2)$ comparaciones. Por otro lado, al inicio del algoritmo se almacena en una estructura de datos los punteros de los candidatos a vecinos de cada triángulo t de la malla, los que corresponden a los triángulos que comparten algún vértice con t. Esto significa que en el peor caso la estructura contiene $O(n^2)$ punteros.

En el caso de *MeshSuite*, se realizan siempre $O(n^2)$ comparaciones, pues en este software la asignación de vecinos se realiza comparando cada triángulo contra todos los demás de la malla y no requiere del uso de espacio adicional al de la malla.

El análisis anterior indica que el peor caso del algoritmo de asignación de vecinos de *Compare2DMesh* es peor que el caso general del algoritmo de *MeshSuite*. Sin embargo, en las mallas que se utilizan en la práctica el número promedio de triángulos que comparten un vértice es menor que 8, y el costo total de encontrar los vecinos es aproximadamente lineal.

5.8. Desactivación de interfaz gráfica de usuario en Compare2DMesh

Debido a que dibujar la triangulación en cada iteración del proceso de refinamiento es un costo adicional para los algoritmos, se modifica el software de modo de poder desactivar opcionalmente la interfaz gráfica. De este modo, el usuario puede elegir si desea usar el software solamente para refinar una malla, o para observar gráficamente la evolución del proceso de refinamiento.

Para ejecutar *Compare2DMesh* desactivando la GUI, se debe pasar los siguientes parámetros mediante línea de comandos:

```
./Compare2DMesh archivo_mesh preproceso priorizacion_triangulos angulo
                                algoritmo
```

Donde los valores de cada parámetro son los siguientes:

- **archivo_mesh**: corresponde a la ruta del archivo .mesh que contiene los datos de la malla a cargar
- **preproceso**: indica si se procesan o no las aristas *encroached* de la triangulación

    - 0 = sin preproceso
    - 1 = con preproceso
- **priorizacion_triangulos**: indica en qué orden deben procesarse los triángulos de mala calidad
    - 0 = sin ordenar
    - 1 = ordenar triángulos de menor a mayor ángulo
- **angulo**: valor del ángulo mínimo al que se desea llegar
- **algoritmo**: algoritmo a utilizar para refinar la triangulación
    - 1 = Circuncentro Ruppert
    - 2 = Lepp-Delaunay punto medio
    - 3 = Lepp-Delaunay centroide
    - 4 = Off-center Üngör

Para refinar una malla utilizando la GUI, simplemente se ejecuta *Compare2DMesh* sin parámetros.

Capítulo 6

Validación y pruebas realizadas con Compare2DMesh

En este capítulo se presentan los resultados de diferentes pruebas de comparación realizadas entre los software *Compare2DMesh*, *MeshSuite* y *Triangle*. También se presentan algunas comparaciones entre los diferentes algoritmos de refinamiento implementados en *Compare2DMesh*. Las pruebas fueron realizadas en un computador con **procesador Core 2 Duo de 2.2 GHz y 3 GB de memoria RAM**.

6.1. Comparación de tiempo de proceso de refinamiento con y sin uso de GUI en Compare2DMesh

Una de las características importantes de *Compare2DMesh* es que permite visualizar cada inserción de un nuevo vértice mientras la triangulación está siendo refinada. Sin embargo, redibujar la malla después de cada inserción implica un costo adicional en tiempo de ejecución de los algoritmos de refinamiento. Con el fin de obtener una medida aproximada del sobrecosto de mostrar gráficamente el proceso de refinamiento, se desactiva la GUI en Compare2DMesh (ver sección 5.8), de manera de poder determinar cuánto tiempo toma realmente para el software ejecutar cada uno de los algoritmos.

Los datos de la Tabla 6.1 muestran que los tiempos de ejecución de *Compare2DMesh* utilizando GUI son mucho más altos que los tiempos de ejecución sin utilizarla. Por ejemplo, el tiempo que toma refinar una malla de cincuenta mil puntos ejecutando el software con GUI es de 1 hora y cuarto, mientras que refinarla sin utilizar la interfaz toma tan solo 18 segundos. En otras palabras, para cincuenta mil puntos el refinamiento usando GUI es 250 veces más lento.

En la Figura 6.1 se muestra como el tiempo de ejecución de *Compare2DMesh* utilizando la GUI siempre es mayor al tiempo que toma el software en refinar cuando se utiliza solo

N° vértices	C2DM GUI (ms)	C2DM cmd (ms)
100	200	5
500	800	40
1000	2000	70
5000	45000	800
10000	180000	2000
50000	4500000	18000

Cuadro 6.1: Tabla comparativa de tiempo promedio de ejecución de refinamiento de mallas entre *Compare2DMesh* corriendo con interfaz gráfica (GUI) y sin ella (cmd), utilizando el algoritmo de Ruppert para las pruebas. La columna "N° vértices" corresponde a la cantidad de puntos inicial de la malla testeada.

mediante linea de comandos, independiente del tamaño inicial de la malla. Por otro lado, en la Figura 6.2 se puede apreciar que el tiempo que le toma a *Compare2DMesh* refinar una malla utilizando la GUI crece más rápido (en función del número de puntos inicial de la malla) que el tiempo que le toma hacerlo ejecutando el software solo por línea de comandos. Esto quiere decir que para realizar comparaciones de tiempo, número de vértices, número de triángulos y otros parámetros, y otras pruebas que requieren la repetición de varios experimentos con mallas grandes, conviene ejecutar *Compare2DMesh* sin interfaz gráfica. Pero, si lo que se desea es estudiar paso a paso el comportamiento de los algoritmos, observar visualmente como progresa el refinamiento, escoger manualmente los triángulos a refinar u otras características que pueden ser analizadas mediante la visualización del proceso de refinamiento; entonces la ejecución de *Compare2DMesh* utilizando interfaz gráfica es de gran utilidad, con la restricción de que las mallas a procesar deben ser pequeñas para que el refinamiento termine en un tiempo razonable (minutos).

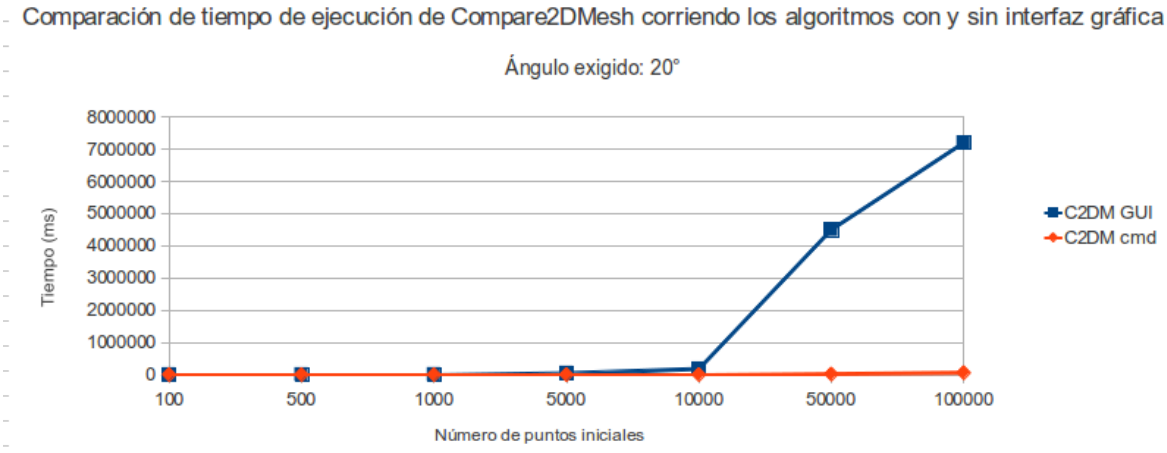

Figura 6.1: Gráfico comparativo de tiempo de ejecución de refinamiento de mallas entre *Compare2DMesh* corriendo con interfaz gráfica (GUI) y sin ella (cmd), en función del número de puntos iniciales de la malla.

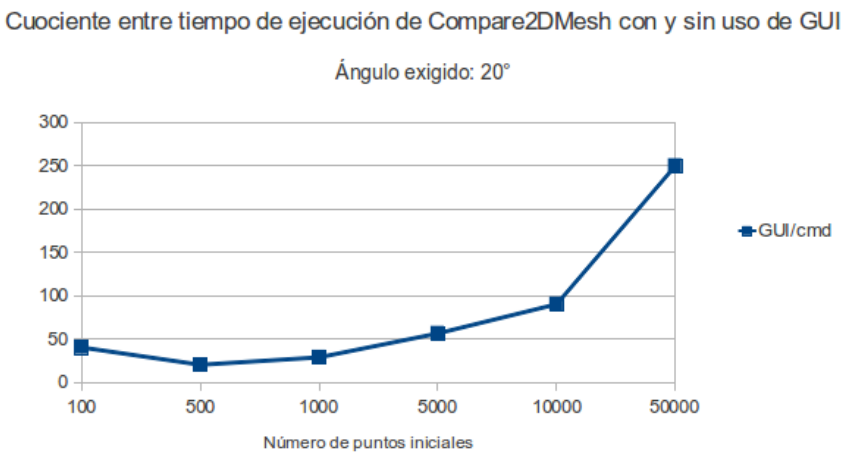

Figura 6.2: Cuociente entre los tiempos de ejecución de *Compare2DMesh* utilizando GUI y *Compare2DMesh* sin GUI (cmd).

6.2. Comparación de tiempo de proceso de refinamiento entre Compare2DMesh y MeshSuite

Para validar la mejora en la eficiencia de los tiempos de ejecución que el software *Compare2DMesh* introduce al modificar la implementación de los algoritmos en *MeshSuite*, se compara el tiempo que toma refinar mallas de diferente tamaño, utilizando el algoritmo de Ruppert y exigiendo un ángulo de 20º. Para que la comparación sea justa, se ejecuta *Compare2DMesh* con interfaz gráfica.

puntos	C2DM GUI (ms)	MeshSuite (ms)
100	200	1000
500	800	18000
1000	2000	180000
5000	45000	$>$ 10800000
10000	180000	?
50000	4500000	?

Cuadro 6.2: Tabla comparativa de tiempo de ejecución de refinamiento de mallas entre *Compare2DMesh* corriendo con interfaz gráfica (GUI) y *MeshSuite*.

Como se puede apreciar en la Tabla 6.2, la diferencia entre los tiempos de ejecución de *MeshSuite* y *Compare2DMesh* es muy grande. El tiempo de refinamiento de una malla de cinco mil puntos en *Compare2DMesh* utilizando GUI es de unos 45 segundos, mientras que en *MeshSuite* la misma malla estuvo siendo procesada por más de 3 horas y en ese tiempo el refinamiento no llegó a término. Observar además que *MeshSuite* se demora en procesar cinco mil puntos, más del doble del tiempo que a *Compare2DMesh* le toma procesar una malla 10 veces más grande. Estos resultados permiten validar que la implementación de los algoritmos de refinamiento en el software *Compare2DMesh* efectivamente es mucho más eficiente que la de su predecesor *MeshSuite*.

6.3. Comparación de algoritmos de refinamiento implementados en Compare2DMesh y Triangle

Para validar la utilidad de *Compare2DMesh* como herramienta de comparación se realizan pruebas utilizando los algoritmos implementados en el software para refinar diferentes triangulaciones. Para ello, se utilizan mallas con vértices generados de forma aleatoria. Los resultados obtenidos se comparan también con el algoritmo de refinamiento implementado en *Triangle*, que es la herramienta de refinamiento de mallas de código abierto más conocida y eficiente que existe actualmente. Para que la comparación sea justa, se ejecuta *Compare2DMesh* sin interfaz gráfica.

6.3.1. Comparación de tiempo de ejecución de los algoritmos

Los gráficos de las Figuras 6.3 a 6.7 presentan los datos obtenidos a partir de la medición del tiempo que toma el refinamiento de mallas de diferente tamaño para diferentes ángulos exigidos. Las pruebas fueron realizadas para los algoritmos de Ruppert, Üngör, Lepp-Delaunay punto medio y Lepp-Delaunay centroide implementados en *Compare2DMesh*; y el algoritmo de refinamiento Delaunay de Paul Chew implementado en *Triangle*.

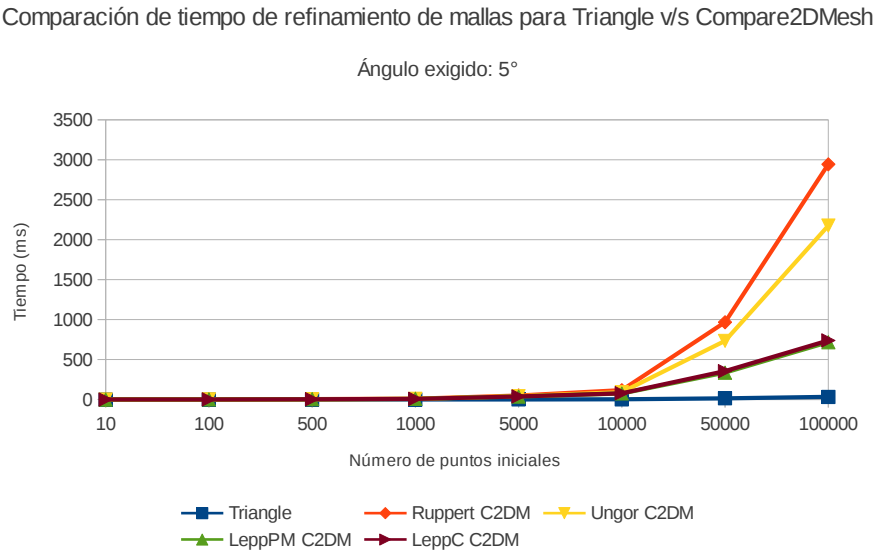

Figura 6.3: Comparación de tiempo de refinamiento entre algoritmos de Ruppert, Üngör, Lepp-Delaunay centroide, Lepp-Delaunay punto medio y algoritmo de Paul Chew (*Triangle*); para ángulo de calidad exigido de 5°.

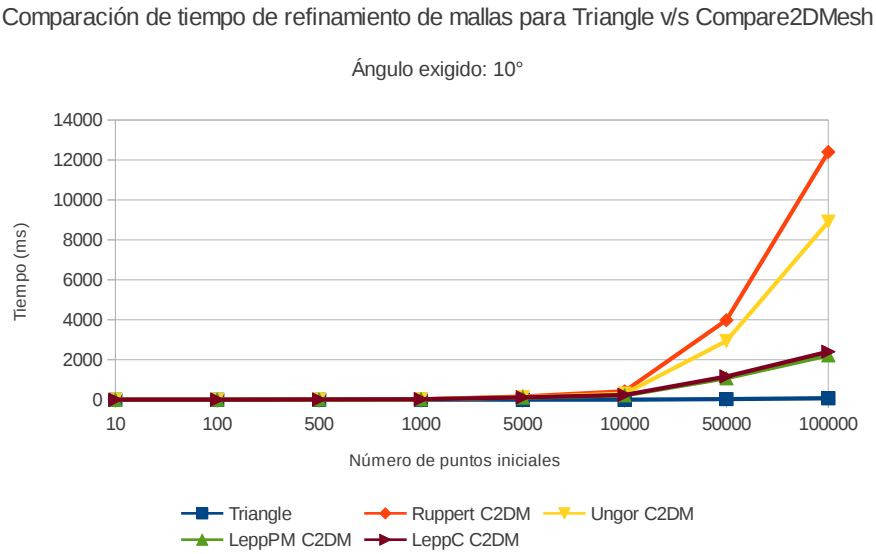

Figura 6.4: Comparación de tiempo de refinamiento entre algoritmos de Ruppert, Üngör, Lepp-Delaunay centroide, Lepp-Delaunay punto medio y algoritmo de Paul Chew (*Triangle*); para ángulo de calidad exigido de 10°.

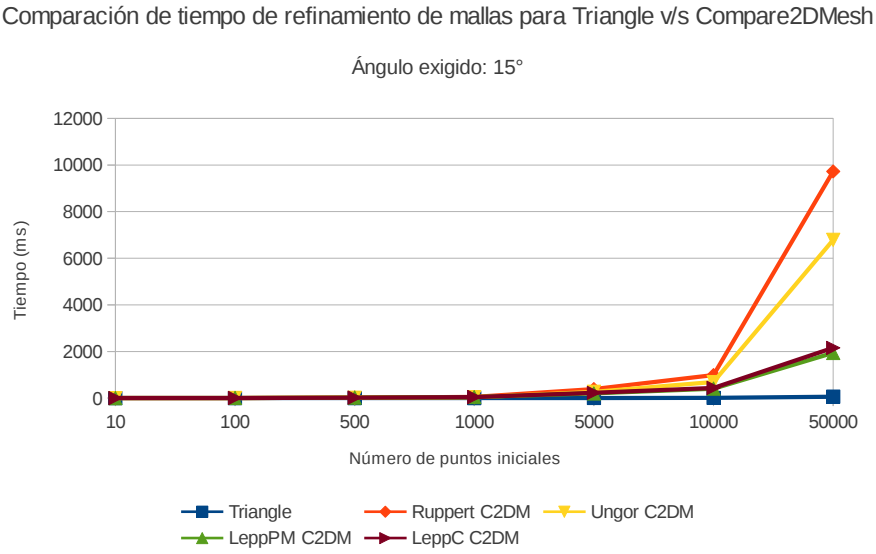

Figura 6.5: Comparación de tiempo de refinamiento entre algoritmos de Ruppert, Üngör, Lepp-Delaunay centroide, Lepp-Delaunay punto medio y algoritmo de Paul Chew (*Triangle*); para ángulo de calidad exigido de 15°.

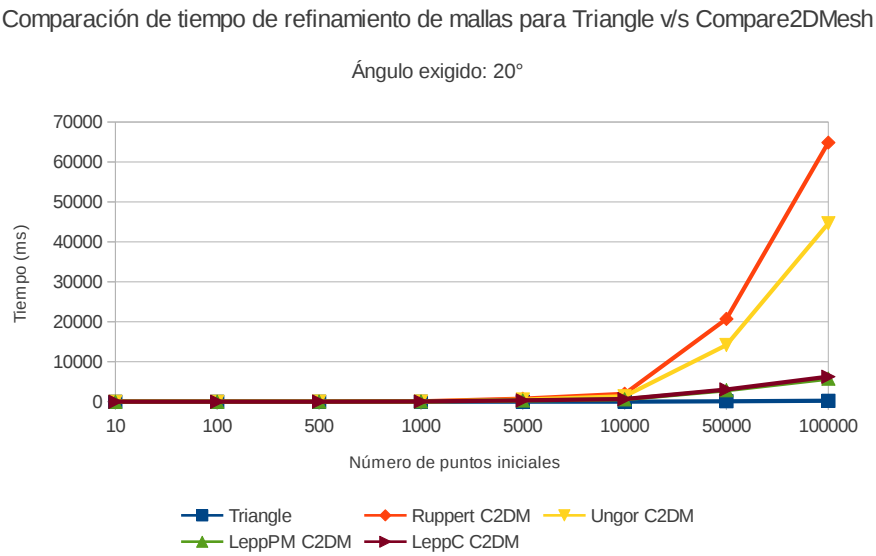

Figura 6.6: Comparación de tiempo de refinamiento entre algoritmos de Ruppert, Üngör, Lepp-Delaunay centroide, Lepp-Delaunay punto medio y algoritmo de Paul Chew (*Triangle*); para ángulo de calidad exigido de 20°.

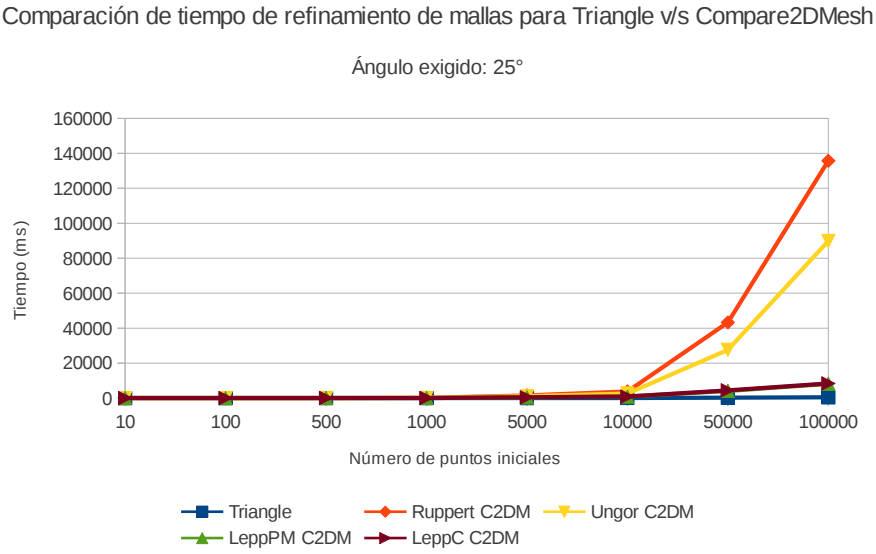

Figura 6.7: Comparación de tiempo de refinamiento entre algoritmos de Ruppert, Üngör, Lepp-Delaunay centroide, Lepp-Delaunay punto medio y algoritmo de Paul Chew (*Triangle*); para ángulo de calidad exigido de 25°.

Los datos muestran que a partir de los diez mil puntos, la diferencia de tiempo de ejecución entre el algoritmo implementado en *Triangle* y los algoritmos de Ruppert y Üngör implementados en *Compare2DMesh* aumenta drásticamente, siendo el tiempo de *Triangle* el que crece más lentamente con respecto al número de puntos de la malla y también con respecto

al ángulo exigido. En el caso de los algoritmos Lepp-Delaunay, los tiempos son cercanos al de *Triangle*.

Los datos muestran que a partir de los diez mil puntos, la diferencia de tiempo de ejecución entre el algoritmo implementado en *Triangle* y los algoritmos de Ruppert y Üngör implementados en *Compare2DMesh* aumenta drásticamente, siendo el tiempo Triangle el que crece más lentamente con respecto al número de puntos de la malla y también con respecto al ángulo exigido. En el caso de los algoritmos Lepp-Delaunay, los tiempos son cercanos al de *Triangle*.

6.3.2. Comparación de número vértices insertados por los algoritmos

Los gráficos de las Figuras 6.8 a 6.12 presentan la relación de número de vértices resultantes luego del proceso de refinamiento versus el número inicial de vértices de la malla, para mallas de diferente tamaño y diferente ángulo exigido. Las pruebas fueron realizadas para los algoritmos de Ruppert, Üngör, Lepp-Delaunay punto medio y Lepp-Delaunay centroide implementados en *Compare2DMesh*; y el algoritmo de refinamiento Delaunay de Paul Chew implementado en *Triangle*.

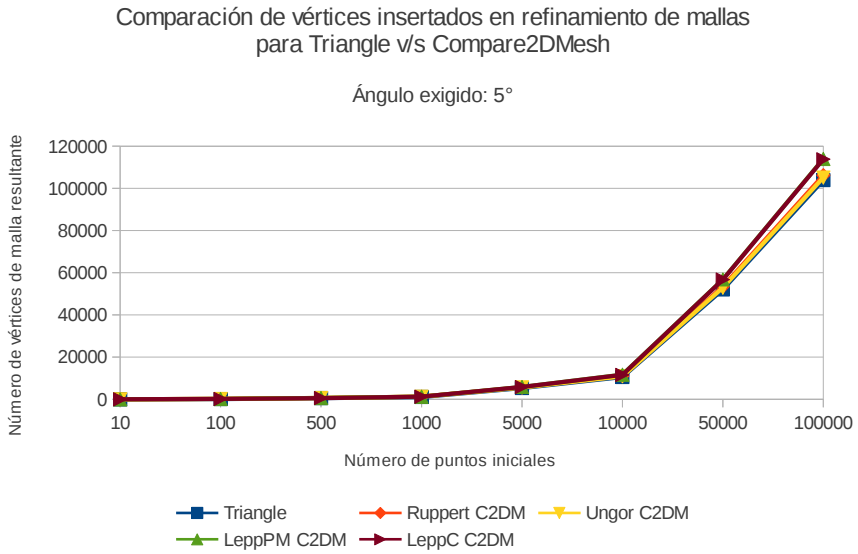

Figura 6.8: Comparación de número de vértices resultantes entre algoritmos de Ruppert, Üngör, Lepp-Delaunay centroide, Lepp-Delaunay punto medio y algoritmo de Paul Chew (*Triangle*); para ángulo de calidad exigido de 5°.

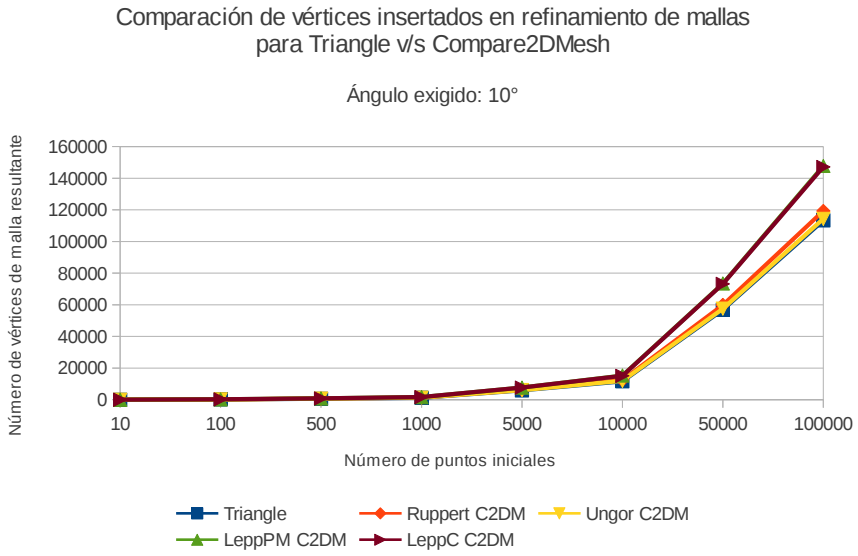

Figura 6.9: Comparación de número de vértices resultantes entre algoritmos de Ruppert, Üngör, Lepp-Delaunay centroide, Lepp-Delaunay punto medio y algoritmo de Paul Chew (*Triangle*); para ángulo de calidad exigido de 10°.

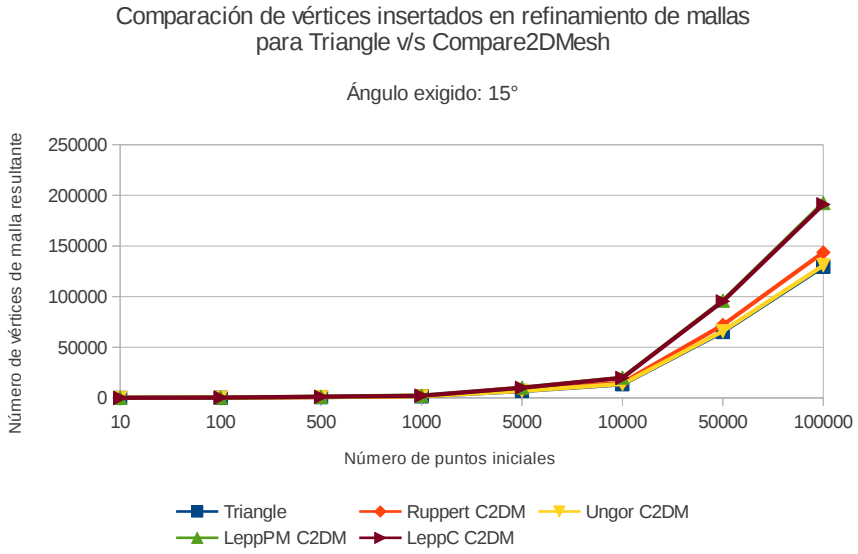

Figura 6.10: Comparación de número de vértices resultantes entre algoritmos de Ruppert, Üngör, Lepp-Delaunay centroide, Lepp-Delaunay punto medio y algoritmo de Paul Chew (*Triangle*); para ángulo de calidad exigido de 15°.

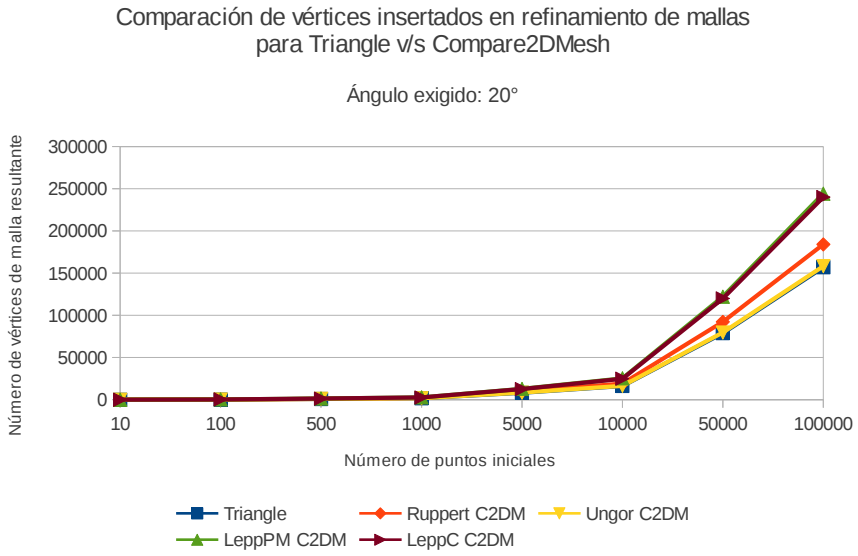

Figura 6.11: Comparación de número de vértices resultantes entre algoritmos de Ruppert, Üngör, Lepp-Delaunay centroide, Lepp-Delaunay punto medio y algoritmo de Paul Chew (*Triangle*); para ángulo de calidad exigido de 20°.

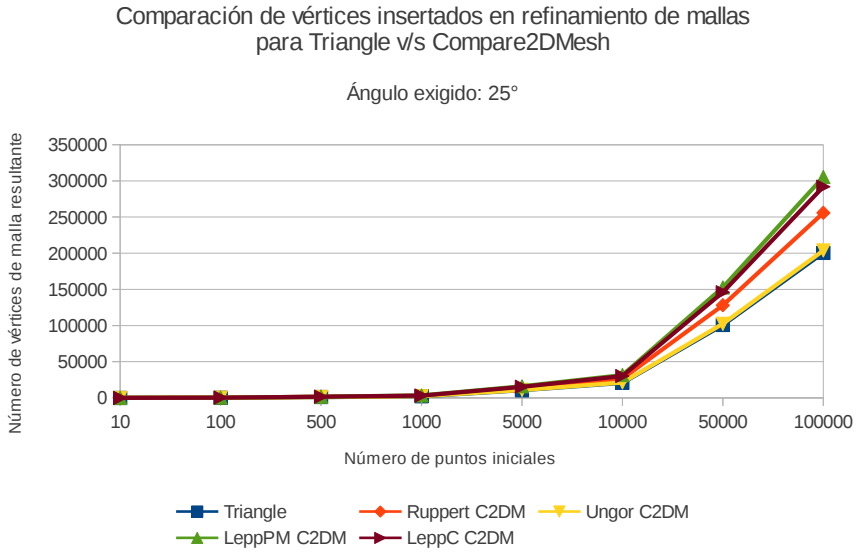

Figura 6.12: Comparación de número de vértices resultantes entre algoritmos de Ruppert, Üngör, Lepp-Delaunay centroide, Lepp-Delaunay punto medio y algoritmo de Paul Chew (*Triangle*); para ángulo de calidad exigido de 25°.

Al igual que en los resultados de la comparación de tiempos de ejecución, las diferencias entre los algoritmos comienzan a ser evidentes a partir de los diez mil puntos. Sin embargo, la diferencia de vértices insertados para los diferentes algoritmos no es tan grande como la diferencia en tiempo de ejecución. Por ejemplo: el algoritmo de Ruppert de *Compare2DMesh* se demora 340 veces más que *Triangle* en refinar una malla de cien mil puntos con umbral de

calidad de 25º. En cambio, la cantidad de vértices resultantes tras el proceso de refinamiento es solo 1.3 veces mayor para la misma instancia (valores obtenidos a partir de los datos de la tabla del anexo A).

Otro punto interesante a analizar es que, al contrario de lo que sucede con los tiempos de refinamiento, en la implementación de *Compare2DMesh* los algoritmos de Ruppert y Üngör se comportan mejor que los algoritmos Lepp-Delaunay Üngör, insertando menos vértices aunque se demoran más tiempo en refinar la malla. Sin embargo, análogamente a lo que ocurre en la comparación de tiempos de ejecución, los algoritmos de *Compare2DMesh* insertan más vértices que el algoritmo implementado en *Triangle*. Excepto el algoritmo de Üngör que, para todas las pruebas realizadas, inserta casi la misma cantidad de vértices que *Triangle*.

6.3.3. Refinamiento de mallas con aristas restringidas y punto interior muy cercano a una arista restringida interior

Cuando dentro de una malla existe algún triángulo T cuya arista más larga A es restringida y posee un vértice muy cercano a ella, se requiere insertar muchos vértices alrededor de la zona para eliminar T. Esto se debe a que no es posible realizar un intercambio de diagonales con la arista A por ser restringida, y a que los ángulos adyacentes a ella son muy pequeños.

En una malla generada de forma aleatoria, como las utilizadas en las subsecciones 6.3.1 y 6.3.2, este caso puede darse con alta probabilidad, por lo que es posible deducir que el algoritmo de refinamiento implementado en el software *Triangle* inserta menos puntos que los algoritmos implementados en *Compare2DMesh* para refinar este tipo de malla.

En la Figura 6.13 se presenta una malla pequeña que contiene un triángulo obtuso cuya arista más larga es restringida. Observar que los ángulos adyacentes a dicha arista son muy pequeños.

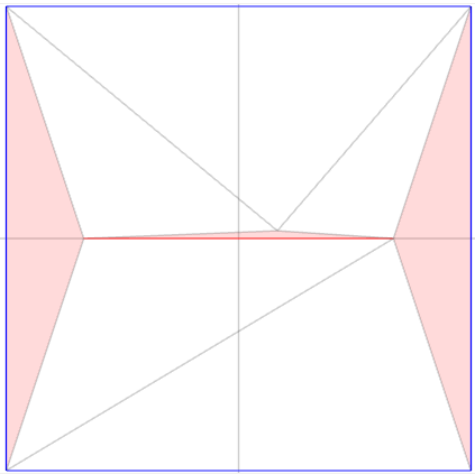

Figura 6.13: Malla inicial con triángulo de ángulo mínimo pequeño y arista más larga restringida

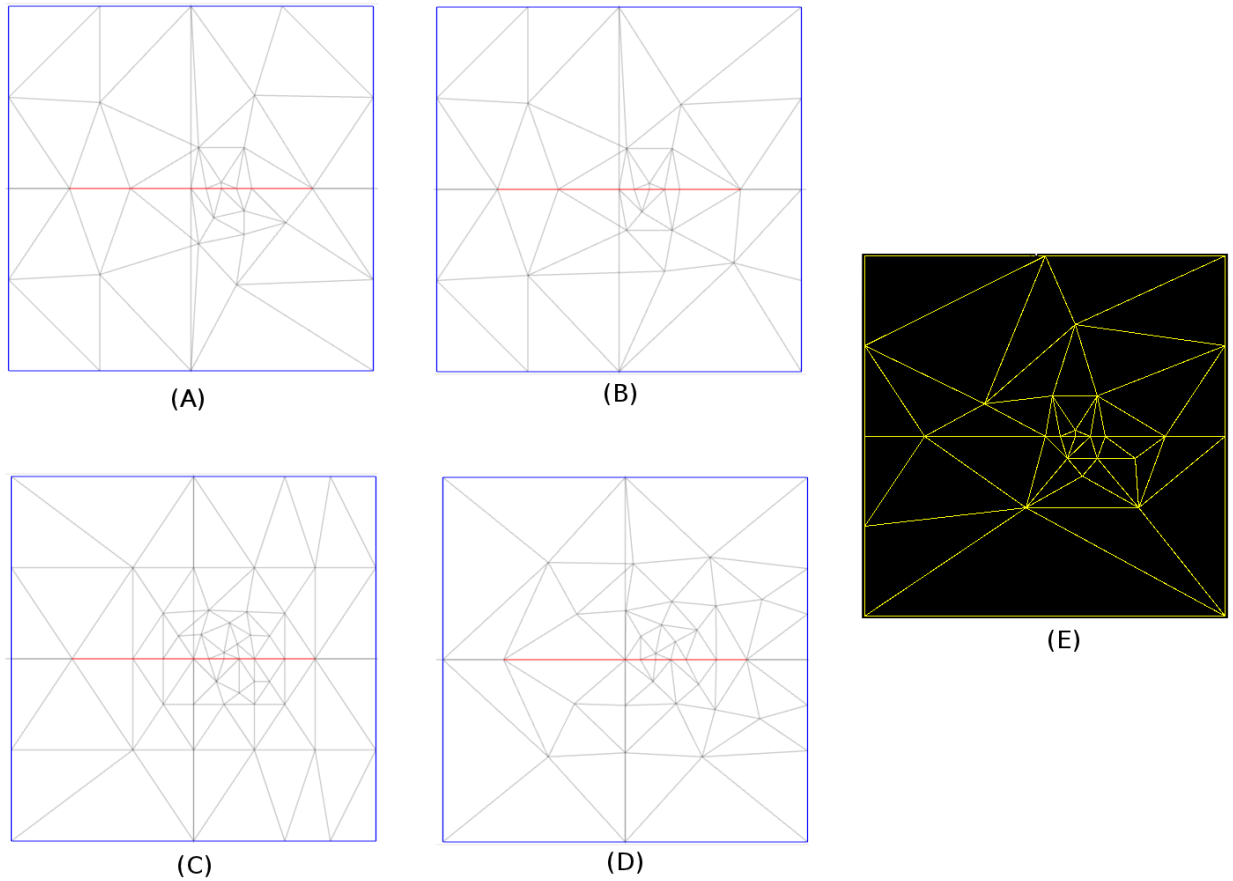

Figura 6.14: Malla resultante después del proceso de refinamiento para un ángulo exigido de 20°. (A) Malla refinada utilizando el algoritmo de Ruppert. (B) Malla refinada utilizando el algoritmo de Üngör. (C) Malla refinada utilizando el algoritmo Lepp-Delaunay punto medio. (D) Malla refinada utilizando el algoritmo Lepp-Delaunay centroide. (E) Malla refinada utilizando el algoritmo implementado en *Triangle*.

En la Figura 6.14 se presenta el resultado de refinar la malla con los diferentes algoritmos de los que se dispone, para un ángulo de 20º. Claramente, y tal como lo indican los resultados anteriores, *Triangle* es el que inserta menos vértices. La pequeña cantidad de vértices insertados por *Triangle* incluso en un caso complejo como este, se debe a que el algoritmo de refinamiento utilizado en *Triangle* inserta menos vértices que dividen las aristas restringidas [13].

6.4. Comparación de algoritmos de refinamiento implementados en Compare2DMesh, con variación del orden de procesamiento de triángulos

Para validar el potencial de comparación del software *Compare2DMesh*, y aprovechando su flexibilidad para combinar diferentes criterios, se realizan pruebas para comparar el número

de vértices insertados por los diferentes algoritmos de refinamiento implementados, cuando se procesan los triángulos ordenados de menor a mayor ángulo mínimo, y cuando se procesan sin ordenar. En estas pruebas se utilizan mallas con vértices generados de forma aleatoria.

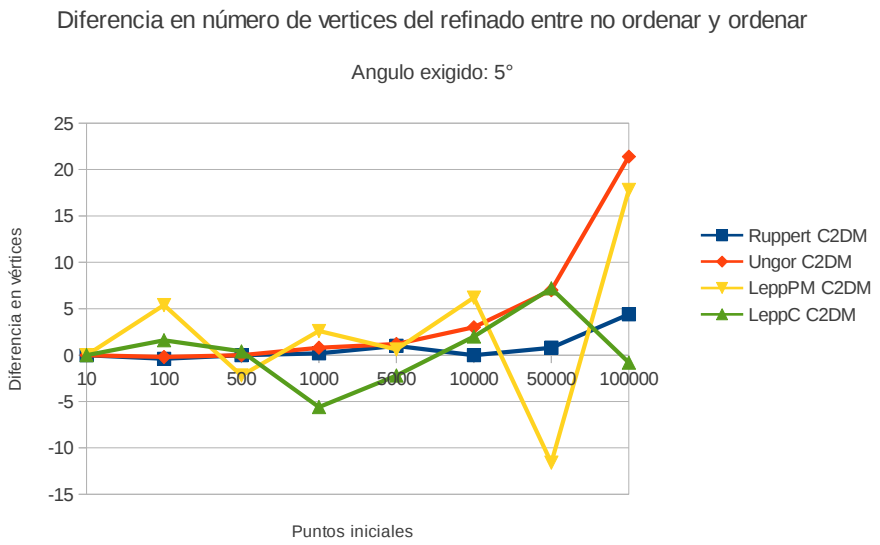

Figura 6.15: Comparación de algoritmos de refinamiento de Ruppert, Üngör, Lepp-Delaunay punto medio y Lepp-Delaunay centroide, con procesamiento de triángulos ordenados según ángulo mínimo y procesamiento de triángulos sin ordenar: para ángulo de calidad exigido de 5°

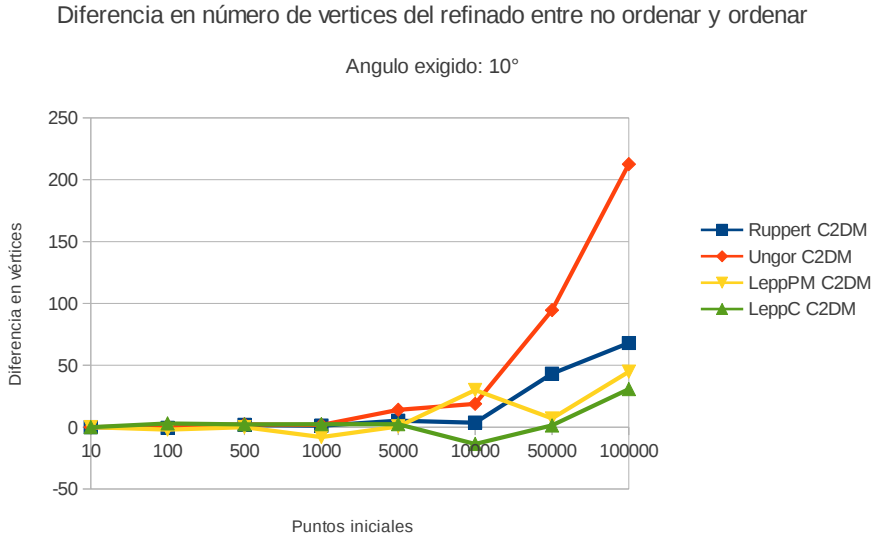

Figura 6.16: Comparación de algoritmos de refinamiento de Ruppert, Üngör, Lepp-Delaunay punto medio y Lepp-Delaunay centroide, con procesamiento de triángulos ordenados según ángulo mínimo y procesamiento de triángulos sin ordenar: para ángulo de calidad exigido de 10°

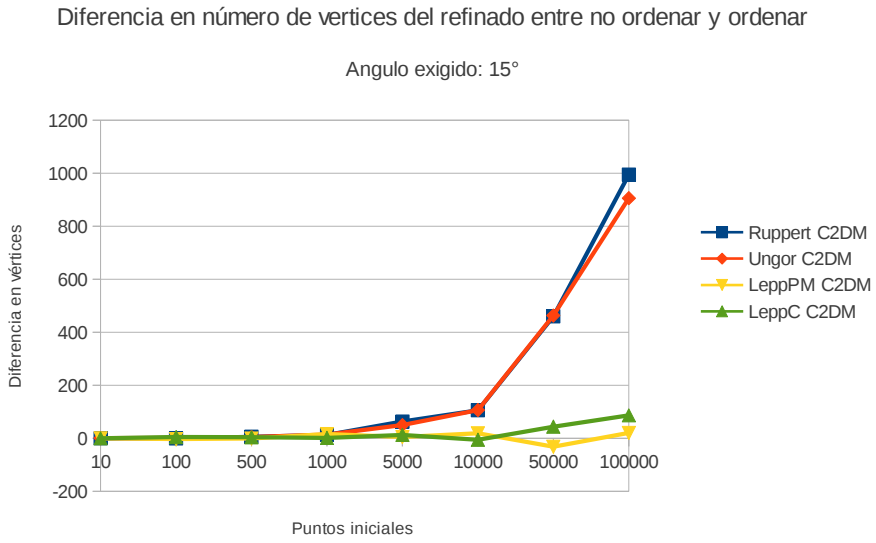

Figura 6.17: Comparación de algoritmos de refinamiento de Ruppert, Üngör, Lepp-Delaunay punto medio y Lepp-Delaunay centroide, con procesamiento de triángulos ordenados según ángulo mínimo y procesamiento de triángulos sin ordenar: para ángulo de calidad exigido de 15°

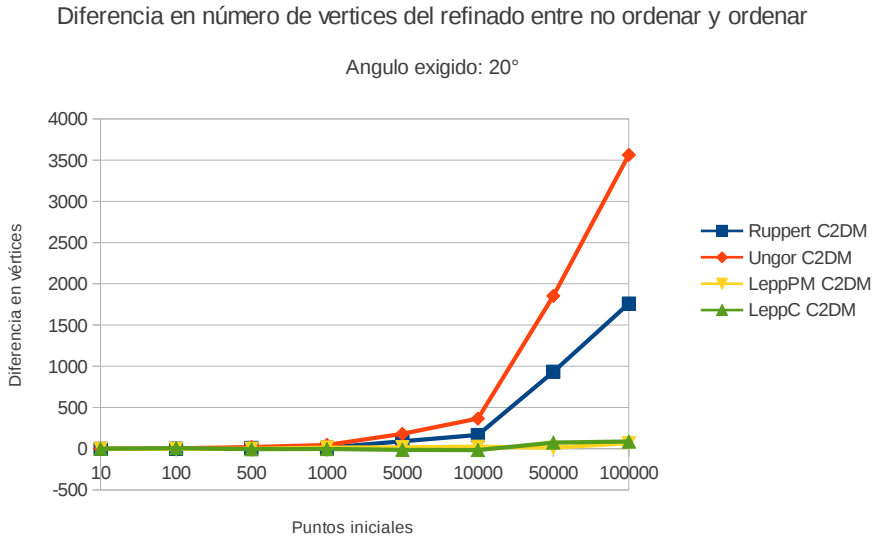

Figura 6.18: Comparación de algoritmos de refinamiento de Ruppert, Üngör, Lepp-Delaunay punto medio y Lepp-Delaunay centroide, con procesamiento de triángulos ordenados según ángulo mínimo y procesamiento de triángulos sin ordenar: para ángulo de calidad exigido de 20°

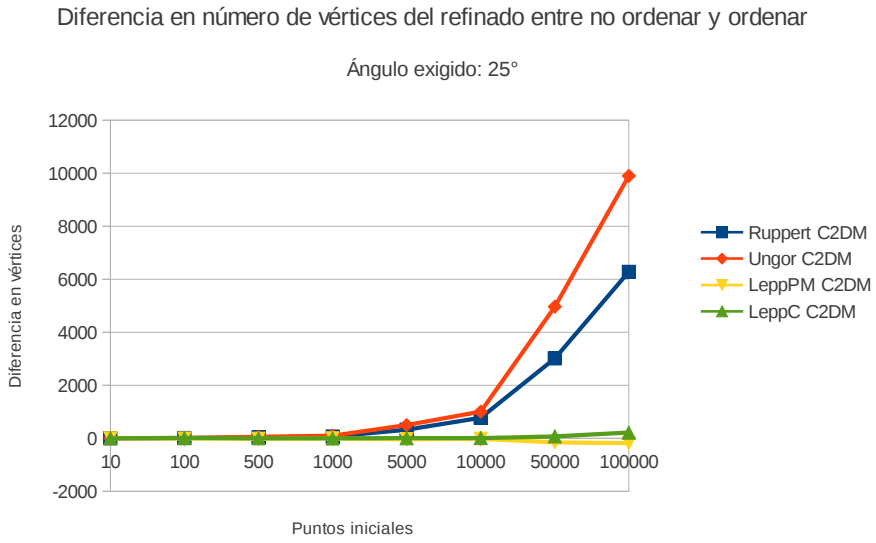

Figura 6.19: Comparación de algoritmos de refinamiento de Ruppert, Üngör, Lepp-Delaunay punto medio y Lepp-Delaunay centroide, con procesamiento de triángulos ordenados según ángulo mínimo y procesamiento de triángulos sin ordenar: para ángulo de calidad exigido de 25°

Los gráficos de las Figuras 6.15 a 6.19 indican que en la implementación de *Compare2DMesh*: a mayor ángulo exigido, más relevancia tiene el ordenamiento de los triángulos en los algoritmos de Ruppert y Üngör, siendo el procesamiento ordenado de los triángulos el que favorece a estos dos algoritmos. En cambio, en el caso de los algoritmos Lepp-Delaunay, el orden en que se procesan los triángulos es irrelevante. Los datos incluso sugieren que el algoritmo Lepp-Delaunay centroide es levemente favorecido si los triángulos se procesan sin ordenar.

Capítulo 7

Conclusiones

7.1. Nueva herramienta de refinamiento de mallas: Compare2DMesh

El objetivo de esta memoria consistió en desarrollar una herramienta de software que permita realizar comparaciones empíricas entre los diferentes algoritmos de refinamiento Delaunay conocidos (Rupper, Off-center, Lepp punto medio, Lepp centroide, etc.).

Existe un prototipo de software de refinamiento de mallas llamado *MeshSuite*. Este prototipo ya tiene implementados los algoritmos a comparar y también muestra al final de cada ejecución algunos datos estadísticos relevantes. En este sentido el software permite comparar algoritmos. Sin embargo, *MeshSuite* presenta algunos serios problemas de ineficiencia en los tiempos de ejecución de los algoritmos de refinamiento. Aunque comparar eficiencia de los algoritmos no es un objetivo primordial, sí lo es el poder refinar mallas de gran tamaño (10 mil, 100 mil, 1 millón de puntos). Es por ello que se decidió tomar como base a *MeshSuite* para construir un nuevo software. El nuevo software, que recibe el nombre de ***Compare2DMesh***, conserva la GUI y diseño del código de *MeshSuite*, pero los algoritmos son reimplementados de forma eficiente.

En términos resumidos, *Compare2DMesh* corrige los siguientes problemas de eficiencia de *MeshSuite*:

- **Selección de triángulo a procesar**: en lugar de recorrer la malla completa en cada iteración, para buscar el triángulo que cumple con el criterio de selección escogido por el usuario, se crea una cola que es inicializada con los triángulos a ser procesados, los cuales son ordenados por prioridad según el criterio de selección que corresponde. Una vez comenzado el proceso de refinamiento, en cada iteración se obtiene el primer elemento de la cola y se procesa.
- **Preproceso de aristas *encroached***: presenta problemas similares a los del caso

anterior, recorriendo la malla completa por cada arista *encroached* en la triangulación, hasta eliminarlas todas. La solución es también similar al del caso de selección de triángulos, se crea una cola que es inicializada con las aristas a ser procesadas. Luego, para eliminarlas basta con ir sacándolas una a una de la cola.

- **Búsqueda de triángulo que contiene un determinado punto**: para insertar un nuevo punto en la triangulación es necesario determinar en qué triángulo de la malla está contenido. Recorrer todos los triángulos hasta encontrar el que contiene el punto es poco eficiente, por lo que se implementa el algoritmo "*Straight Walk*" que recorre solo una porción de la malla.

- **Construcción de estructuras de datos a partir de archivo de entrada .mesh**: en *MeshSuite*, las relaciones de vecindad entre los triángulos de la malla se calculan comparando todos los triángulos con todos. Este procedimiento se mejora aprovechando información del archivo de entrada para construir una tabla que asocia a cada vértice de la malla, los triángulos que lo contienen. Usando esta información se compara cada triángulo solo con aquellos que comparten alguno de sus vértices, lo que reduce el número de comparaciones.

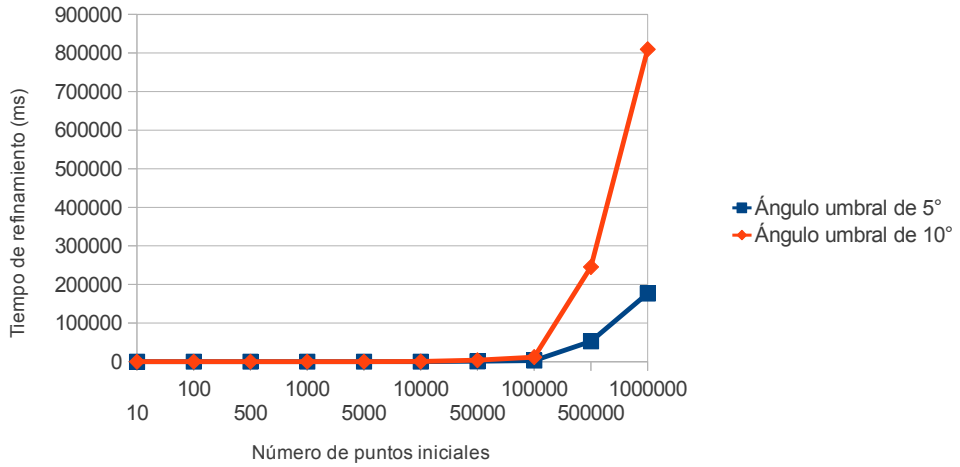

Figura 7.1: Algunas pruebas realizadas en Compare2DMesh con el fin de medir el tiempo que toma refinar mallas de diferentes tamaños. El gráfico muestra los resultados del refinamiento de mallas de hasta un millón de puntos iniciales, utilizando el algoritmo de Ruppert con criterios de calidad de 5° y 10°.

Las mejoras en el proceso de refinamiento incluidas en *Compare2DMesh* permiten refinar mallas mucho más grandes y en menos tiempo que su predecesor *MeshSuite*, en las pruebas realizadas durante el desarrollo del software se refinaron mallas de hasta un millón de puntos (Figura 7.1). *Compare2DMesh* tiene además la ventaja de poder ejecutarse con y sin interfaz gráfica de usuario, esto permite eliminar de los algoritmos el costo adicional de dibujar la malla tras cada inserción en el proceso de refinamiento y, por otro lado, mantiene la posibilidad de visualizar de forma gráfica el progreso de dicho proceso, con las ventajas que esto implica para estudiar el comportamiento de los algoritmos. Por último, cabe mencionar

que *Compare2DMesh* mantiene separadas las implementaciones de preprocesamiento de la malla, selección de triángulo a refinar y algoritmo de selección de punto a insertar, lo cual permite ejecutar los algoritmos de refinamiento con diferentes variaciones.

En resumen, como producto del desarrollo de esta memoria, se obtiene un software que provee implementaciones operativas de los algoritmos sin limitaciones en cuanto al tamaño de mallas y geometrías complejas. Estas características permiten que *Compare2DMesh* pueda ser utilizado para los fines de probar, comparar y afinar algoritmos.

7.2. Trabajo futuro

Como ocurre en el desarrollo de todo software, la herramienta de comparación de algoritmos Compare2DMesh puede ser mejorada. La siguiente lista presenta una propuesta de los puntos a trabajar:

- Eliminación de fugas de memoria del software y mejoramiento del manejo de memoria de las estructuras.
- Generalización de algoritmo de carga de mallas desde archivo, de modo de soportar diferentes formatos.
- Extender el software de modo que pueda escribir en un archivo de salida los datos estadísticos de ejecución (los mismos que actualmente se muestran en pantalla).
- Generalizar estructura de colas de preprocesamiento, de modo que se pueda incluir preproceso de triángulos e incluir diferentes criterios de priorización para el preproceso.
- Parametrizar el margen de error considerado en el test de orientación, test del círculo y otros cálculos que puedan llevar a problemas de precisión, de modo que para cambiar su valor baste cambiar una única variable (incluso podría ser definible a través de la interfaz de usuario).
- Agregar al software un algoritmo para construir la triangulación de Delaunay inicial de un conjunto de puntos, la cual puede o no ser posteriormente refinada. Se debe incluir en la interfaz gráfica una opción que permita al usuario elegir este algoritmo.
- Incluir criterio de término para el algoritmo Lepp-Bisección, ya que actualmente, al igual que los demás algoritmos, está configurado para terminar cuando se eliminan todos los triángulos de mala calidad, lo cual no tiene sentido para este algoritmo pues no está diseñado para mejorar la calidad de la malla, por lo que podría nunca alcanzarse el umbral deseado y por ende, el algoritmo entraría en un bucle infinito. Opciones posibles de criterio de término para Lepp-Bisección serían definir un número máximo de puntos insertados, o definir el valor mínimo de área de triangulo a alcanzar en alguna zona especifica).

- Incluir un preprocesamiento de la malla en el algoritmo Lepp-Delaunay punto medio que consiste en insertar el Off-center de los triángulos de mala calidad cuya arista más pequeña sea $\frac{1}{4}$ de la arista más larga. Este preprocesamiento debe realizarse sólo antes de la primera inserción. Una vez implementado esto, deben realizarse pruebas para comprobar que se comporta de la forma esperada.
- Implementar nuevos criterios de inserción de puntos en el borde y aristas restringidas para obtener comparaciones más precisas sobre los algoritmos.
- Incluir manejo de mallas que poseen algún triángulo con ángulo de mala calidad encerrado entre dos aristas restringidas, identificando el triángulo en el proceso y marcándolo como no refinable, de modo que quede fuera de la cola de triángulos a procesar y así el refinamiento siempre termine.
- Continuar mejorando la eficiencia de tiempo de ejecución de los algoritmos.
- Agregar en la interfaz gráfica un comando/botón que permita pausar el refinamiento en curso.

Capítulo 8

Bibliografía

[1] Mark de Berg, Otfried Cheong, Marc van Kreve, Mark Overmars. Computational Geometry, Algorithms and Applications. Third Edition. Editorial Springer. 2008.

[2] Herbert Edelsbrunner. Cambridge Monographs on Applied And Computational Mathematics, Geometry and Topology for Mesh Generation. Editorial Cambridge University Press. 2001.

[3] María Cecilia Rivara, Mauricio Palma. New Lepp-Algorithms For Quality Polygon and Volume Triangulation: Implementation Issues and Practical Behavior. A. Canann. Saigal (Eds.). 1997.

[4] Jim Ruppert. A Delaunay Refinement Algorithm for Quality 2-Dimensional Mesh Generation. The Computer Journal. 1995.

[5] María Cecilia Rivara, C. Calderón. Lepp terminal centroid method for quality triangulation. Computer-Aided Design. 2010.

[6] María Cecilia Rivara. New longest-edge algorithms for the refinement and/or improvement of unstructured triangulations. International Journal for Numerical Methods in Engineering. 1997.

[7] Üngör, A. Off-centers: A New Type of Steiner Points for Computing Size-Optimal Quality-Guaranteed Delaunay Triangulations. Lecture Notes in Computer Science. 2004.

[8] Álvaro Faúndez. Marco de Experimentación para Algoritmos de Refinamiento de Triangulaciones en 2D. Departamento de Ciencias de la Computación, Universidad de Chile. 2010.

[9] Jonathan Richard Shewchuk. Delaunay Refinement Algorithms for Triangular Mesh Generation. Computational Geometry: Theory and Applications. 2002.

[10] Gilbert Strang and George J. Fix. An Analysis of the Finite Element Method. Prentice-Hall. 1973.

[11] Charles L. Lawson. Transforming triangulations. Discrete Mathematics. 1972.

[12] Rivara MC, Hitschfeld N, Simpson RB. Terminal edges Delaunay (small angle based) algorithm for the quality triangulation problem. Computer-Aided Design. 2001.

[13] L. Paul Chew. Guaranteed-Quality Mesh Generation for Curved Surfaces. Proceedings of the Ninth Annual Symposium on Computational Geometry (San Diego, California). Association for Computing Machinery. May 1993.

[14] Gary L. Miller, Steven E. Pav, and Noel J. Walkington. When and Why Ruppert's Algorithm Works. Twelfth International Meshing Roundtable. Sandia National Laboratories. September 2003.

[15] Olivier Devillers, Sylvain Pion, Monique Teillaud. Walking in a Triangulation. Internat. J. Found. Comput. Sci. 2002.