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

REFINAMIENTO DE MALLAS POLIGONALES POR INSERCIÓN DE PUNTOS

MEMORIA PARA OPTAR AL TÍTULO DE
INGENIERO CIVIL EN COMPUTACIÓN

MARIO ANDRÉS PINEDA MEZA

PROFESORA GUÍA:
NANCY HITSCHFELD KAHLER

PROFESOR CO-GUÍA:
SERGIO SALINAS FERNÁNDEZ

MIEMBROS DE LA COMISIÓN:
VALENTIN MUÑOZ APABLAZA
MATÍAS TORO IPINZA

Este trabajo ha sido parcialmente financiado por FONDECYT N° 1211484

SANTIAGO DE CHILE
2023

Resumen

La generación de mallas poligonales es un área ampliamente estudiada con aplicaciones en muchas disciplinas de la ingeniería. Su capacidad para descomponer dominios complejos en partes más pequeñas y fáciles de manejar la convierten en una herramienta de especial interés en simulaciones físicas. Métodos numéricos clásicos para resolver ecuaciones diferenciales trabajan exclusivamente sobre polígonos convexos, pero nuevos métodos como VEM (elementos virtuales) pueden trabajar sobre polígonos arbitrarios.

Este trabajo se enmarca en un nuevo tipo de mallador llamado Polylla, que genera mallas de polígonos arbitrarios a partir de una triangulación inicial. Esto las vuelve útiles para métodos como VEM, pero las inutiliza para los métodos clásicos como FEM (elementos finitos). Al ser un algoritmo nuevo, aún no cuenta con procedimientos que permitan refinar este tipo de mallas, lo cual es una característica deseable en problemas de refinamiento adaptativo de mallas (o mesh adaptive refinement), donde las mallas son refinadas de manera local sobre áreas que requieran mejorar la precisión. Dado esto, se busca implementar un método de refinamiento mediante inserción de puntos que trabaje sobre la triangulación que el algoritmo recibe como input, y se busca implementar un algoritmo que transforme los polígonos cóncavos de una malla Polylla en convexos, para poder hacer uso de estas mallas en los problemas que tengan el requisito de convexidad.

Para refinar mediante inserción de vértices, se implementaron dos algoritmos de refinamiento de Lepp (Longest Edge Propagation Path), uno basado en la inserción de puntos en el centroide de un cuadrilátero y otro en el punto medio de una arista. Estos algoritmos tienen la capacidad de refinar sobre regiones arbitrarias definidas por el usuario, aunque también permiten realizar refinamientos de manera global. Para transformar las mallas Polylla a mallas puramente convexas, se utiliza un método de inserción de aristas basado en la triangulación que Polylla recibe como input, tratando de obtener el mínimo número de polígonos finales en la malla.

Con esto se logra refinar de manera local y global cualquier tipo de triangulación, con las triangulaciones de Delaunay siendo el caso más interesante e importante. El criterio de refinamiento establecido es el área de los triángulos, pero la solución puede ser fácilmente ampliable a otras métricas.

Por último, se logra refinar los polígonos cóncavos en sus partes convexas, manteniendo un buen número de polígonos finales respecto al entregado por las mallas Polylla.

Tabla de Contenido

Índice de Tablas

Índice de Ilustraciones

Capítulo 1

Introducción

La generación de mallas para representar figuras es un tema ampliamente estudiado con aplicaciones en diversos campos, pasando desde la computación gráfica hasta la modelación de complejos fenómenos físicos. Es justamente en esta última área donde las mallas tienen un papel fundamental, principalmente para la aplicación del método de elementos finitos (o FEM por sus siglas en inglés), que es un método numérico para aproximar soluciones de ecuaciones diferenciales usado en múltiples áreas de la ingeniería, como mecánica de sólidos, mecánica de fluidos, electromagnetismo, entre otras. Para esto, se necesita descomponer objetos en partes más pequeñas, lo cual implica discretizar el dominio para poder ser procesado. Un ejemplo se muestra en la Figura 1.1.

Las mallas utilizadas en la aplicación de estos métodos por lo general deben cumplir criterios de forma, por ejemplo que todos los polígonos sean convexos, valor mínimo o máximo de los ángulos, largo de las aristas o área de los polígonos. Es en este contexto donde el refinamiento de mallas poligonales (entendiéndose como el proceso de modificar una malla para cumplir estos criterios) cobra relevancia, ya que influye directamente en el desempeño y tiempo de la modelación. Por ejemplo, una malla más fina probablemente otorgue mayor precisión pero el tiempo de procesamiento puede aumentar significativamente.

El algoritmo Polylla [21] genera mallas con polígonos convexos y no convexos (ver Figura 1.2), siendo una alternativa a las mallas de Voronoi comúnmente usadas en estos problemas. Este algoritmo se basa en el concepto de terminal-edge region [2], y ha mostrado tener buenas propiedades respecto a las mallas de Voronoi, por lo que más investigación y experimentación es requerida.

Una de las restricciones más fuertes que se imponen sobre las mallas poligonales en muchas aplicaciones es que estén compuestas solo de polígonos convexos, como por ejemplo en la generación de terrenos, computer vision, la aplicación de FEM, etc, por lo que las mallas Polylla en estas situaciones no pueden ser utilizadas. Nuevos métodos numéricos como el método del elemento virtual (o VEM por sus siglas en inglés) utiliza mallas con menos restricciones, particularmente relajando esta última, es decir, admite mallas de polígonos cóncavos y convexos, por lo que las mallas Polylla se pueden usar para la resolución de estos problemas. No obstante, en muchos contextos, el uso de polígonos cóncavos sigue siendo problemático, por lo que se busca tener la opción de poder descomponerlos en sus partes

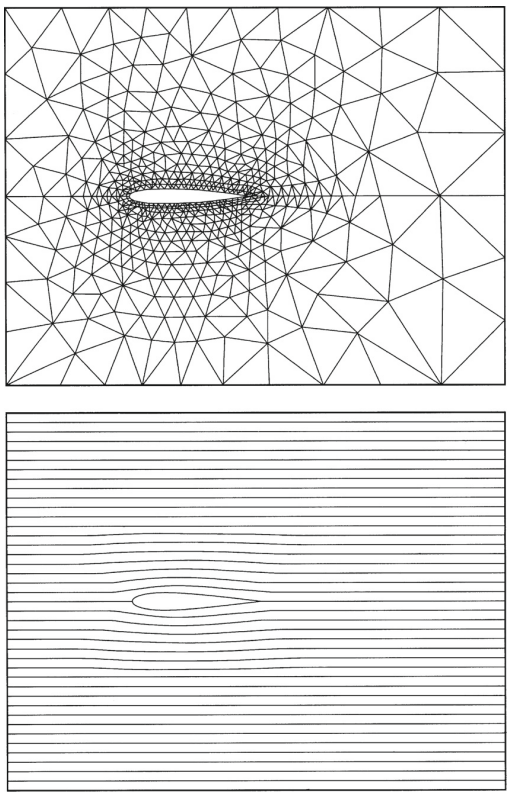
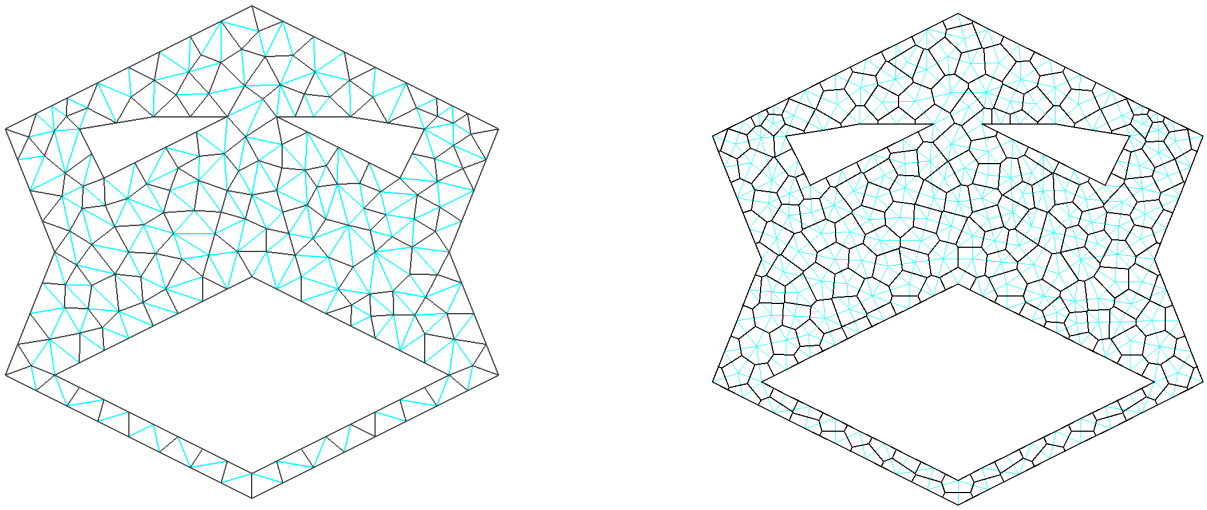


Figura 1.1: Solución del flujo potencial alrededor de un objeto utilizando FEM. Se muestra la malla que subdivide el dominio y las líneas de corriente resultantes. De *The Finite Element Method for Fluid Dynamics (Seventh Edition, p. 24)*, O.C. Zienkiewicz, R.L. Taylor, P. Nithiarasu, 2014, Butterworth-Heinemann.

convexas.

Muchos de los métodos numéricos para la resolución de este tipo de problemas utilizan el método de *adaptive mesh refinement* [15], que consiste en refinar solo ciertas partes de una malla en la que se requiera más precisión, por lo que contar con un método para refinar las mallas Polylla sería de mucha utilidad.

El trabajo de esta memoria busca extender el algoritmo Polylla, implementando un método de refinamiento mediante inserción de puntos, con el objetivo de ampliar la investigación respecto al comportamiento de este tipo de mallas. Específicamente, lo que se busca es tener de input una triangulación cualquiera y, utilizando una inserción de puntos basada en el concepto de terminal-edge region, poder mejorar algún tipo de criterio mencionado con anterioridad, para luego aplicar el algoritmo polylla y ver sus propiedades.



(a) (b)

Figura 1.2: Comparación entre una malla Polylla y una de Voronoi a partir de la misma triangulación inicial. (a) Malla Polylla. En negro las aristas que conforman los polígonos de la malla final. Las aristas negras y cyan constituyen la triangulación inicial. (b) Malla de Voronoi, donde las aristas color cyan corresponden a la triangulación inicial.

1.1. Objetivos

1.1.1. Objetivo General

Diseñar e implementar un algoritmo de refinamiento de mallas de polígonos arbitrarios, es decir, convexos y no convexos, para incluirlo en la actual implementación del mallador Polylla.

1.1.2. Objetivos Específicos

1. Desarrollar e implementar un algoritmo de partición de polígonos cóncavos en sus partes convexas, cuidando la eficiencia y el número final de polígonos.

2. Implementar un algoritmo de refinamiento mediante la inserción de puntos basado en el longest edge propagation path.

3. Definir métricas que controlen el refinamiento de los polígonos, tales como: área de los polígonos, largo de los arcos o ángulos mínimos.

4. Implementar un algoritmo de refinamiento local, es decir, en ciertas partes de la triangulación.

5. Evaluar el desempeño de los algoritmos en tiempo, comparándolo con algoritmos utilizados en la actualidad.

1.2. Descripción General

Para llevar a cabo el refinamiento mediante inserción de puntos, se implementa un algoritmo basado en el *Longest Edge Propagation Path* [4], que funciona realizando inserciones en el punto medio de las aristas, o bien en el centroide de los cuadriláteros. La métrica utilizada como criterio para el refinamiento es él área.

Este algoritmo trabaja con triangulaciones de cualquier tipo y permite el refinamiento global o local. Para esta última, se pueden definir regiones a partir de polígonos arbitrarios, y a través de un algoritmo que determina si un punto se encuentra dentro de una región, se pueden obtener los triángulos objetivo.

Por otro lado, para resolver el problema de las mallas Polylla con polígonos cóncavos, se implementó un algoritmo basado en la inserción de aristas a partir de una triangulación del polígono, tratando de obtener la mínima cantidad de polígonos finales y, en segundo lugar, dividir el ángulo de la manera más uniforme posible.

1.3. Contenido

El presente documento se organiza de la siguiente manera: El capítulo 2 define algunos conceptos geométricos básicos, muestra algunos resultados teóricos importantes para el desarrollo de esta memoria y describe algunas de las principales soluciones con que se cuenta para distintos problemas de interés que conciernen al presente trabajo. El capítulo 3 plantea el problema a resolver, junto con una serie de requisitos que deben ser satisfechos por la solución. También describe la metodología empleada, además de esbozar a grandes rasgos como se estructura la solución. El capítulo 4 describe en detalle la implementación de la solución, explicando en detalle el cálculo de algunas propiedades geométricas, los algoritmos involucrados, las estructuras de datos utilizadas y finalmente la forma de usar lo implementado. En el capítulo 5 se muestran algunos resultados del trabajo junto con ciertos experimentos realizados, validando y comparando la solución obtenida. Finalmente en el capítulo 6 se señalan las conclusiones obtenidas y se señalan áreas en que se puede continuar el trabajo de esta memoria.

Capítulo 2

Marco Teórico

En este capítulo se presentan algunas definiciones básicas que permiten entender el problema abordado en esta memoria, así como los tipos de mallas más utilizados y sus respectivas características. También se explica en detalle el funcionamiento del algoritmo Polylla y algunas propiedades de sus mallas, que van a ser la base del trabajo realizado. Finalmente, se introducen algunos algoritmos de refinamiento de triangulaciones.

2.1. Conceptos básicos

A continuación se presentan algunas definiciones básicas de geometría que serán la base para todo este trabajo, las cuales fueron extraídas de *Computational Geometry in C* [14].

Definición 2.1 *Un segmento $\overline{ab}$ es el subconjunto cerrado de una recta contenida entre dos puntos a y b.*

Definición 2.2 *Un polígono es una región del plano delimitada por una cantidad finita de segmentos que se unen en sus extremos. Estos puntos de unión corresponden a los vértices del polígono.*

Definición 2.3 *Se dice que un polígono es simple si no hay ninguna intersección entre segmentos no adyacentes.*

Definición 2.4 *Un polígono se denomina convexo si para todo par de puntos dentro de este, el segmento que forman está completamente contenido dentro del polígono. Un polígono que no cumple esta condición se denomina cóncavo.*

En la Figura 2.1 se pueden ver ejemplos de estos tipos de polígonos. Una propiedad importante de los polígonos convexos es que ninguno de sus ángulos interiores es mayor a 180°.

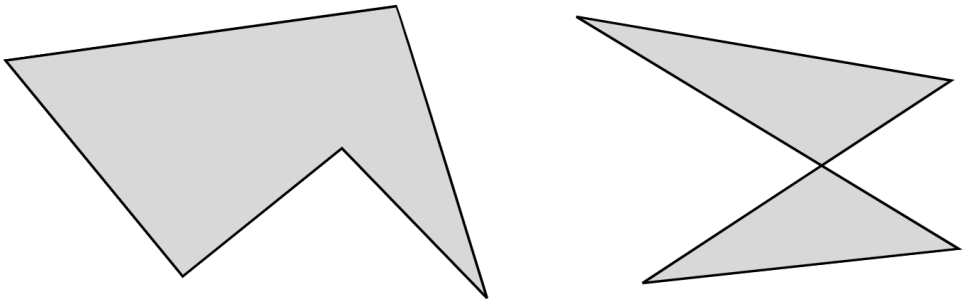


Figura 2.1: Ejemplo de un polígono cóncavo (izquierda) y uno no simple (derecha).

2.2. Generación de mallas poligonales

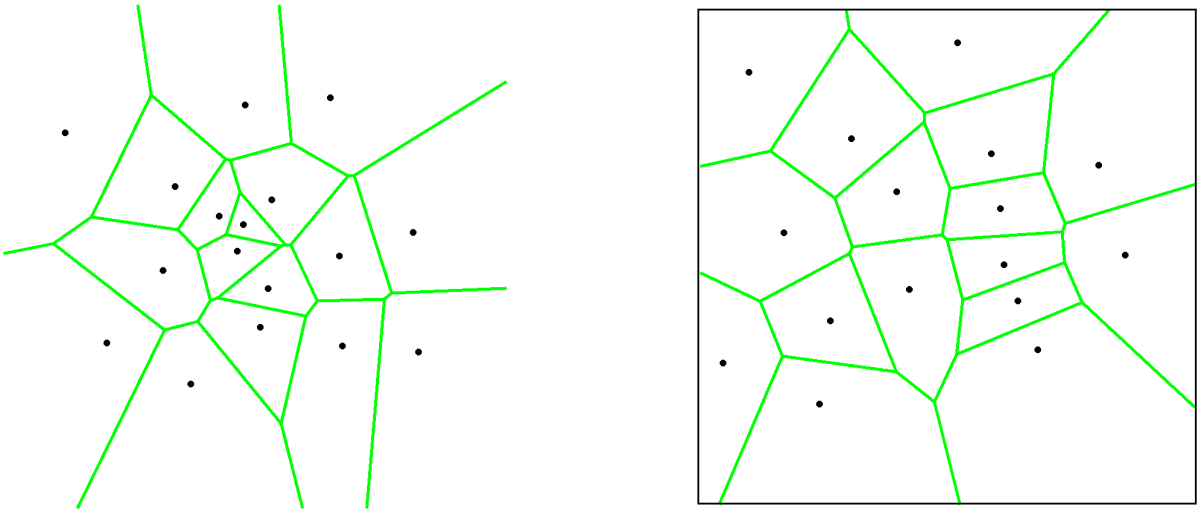
Una malla poligonal corresponde a la descomposición de una región del plano en polígonos que no se intersectan, cubriendo la superficie en su totalidad.

Existen diversos algoritmos para generar mallas poligonales, los cuales se pueden dividir en dos grandes grupos: Algoritmos directos y algoritmos indirectos. Los primeros corresponden a algoritmos que reciben como input la geometría inicial, a diferencia de los segundos que reciben una malla, generalmente una triangulación.

Se han desarrollado diversos algoritmos para generar mallas que satisfacen distintos criterios de calidad, por ejemplo, garantizando los ángulos mínimos o máximos o la relación entre el largo de las aristas. Estos algoritmos se basan en distintos métodos, por ejemplo, haciendo uso de quadtrees [3], usando la técnica de advancing front [12] o métodos basados en triangulaciones de Delaunay [10]. Sin embargo, la mayoría de estos métodos generan solo triangulaciones y no mallas con polígonos arbitrarios.

2.2.1. Diagramas de Voronoi

El método más usado para generar mallas poligonales dado un conjunto de vértices se basa en el uso de los diagramas de Voronoi, que corresponden a una partición del plano en regiones. Por cada vértice inicial se define una región, que consiste en los puntos del plano más cercanos a ese vértice que a cualquier otro (ver Figura 2.2). Las mallas de Voronoi están formadas exclusivamente de polígonos convexos, propiedad requerida en muchas aplicaciones, que es justamente una de las características que hacen que su uso sea tan extendido. Este problema ha sido estudiado por varias décadas y se ha demostrado que toma tiempo $O(n \log n)$ [1], con n siendo el número de vértices.



(a) (b)

Figura 2.2: Diagramas de Voronoi. En (a) se muestra el diagrama de Voronoi para un conjunto de puntos, donde las celdas exteriores son infinitas. En (b) se muestra un diagrama de Voronoi restringido por un cuadrilátero, para un conjunto de puntos. En este caso las todas las celdas son finitas.

2.2.2. Triangulaciones de Delaunay

Un tipo especial de mallas poligonales son las triangulaciones, que consisten en mallas compuestas exclusivamente por triángulos. Un tipo de triangulación muy importante debido a sus propiedades corresponde a la triangulación de Delaunay (ver Figura 2.3), que se define a continuación.

Definición 2.5 *Una triangulación de Delaunay consiste en la triangulación de un conjunto de puntos tal que la circunferencia circunscrita de un triángulo no contiene en su interior ningún vértice de otro triángulo.*

Dado un conjunto de puntos $\mathcal{S}$, tal triangulación siempre existe, y es única cuando $\mathcal{S}$ está en posición general, es decir, no existen cuatro puntos que estén situados en una misma circunferencia.

Para una triangulación $\mathcal{T}$ de un conjunto $\mathcal{P}$ de puntos con m triángulos, se define el vector de ángulos α_i de $\mathcal{T}$ como $A(\mathcal{T}) = (\alpha_1, \alpha_2, ..., \alpha_{3m})$, donde el ángulo $\alpha_i \leq \alpha_j$, para todo $i < j$ [6]. En otras palabras, es la secuencia ordenada en orden ascendente de todos los ángulos de la triangulación.

Ahora, sea $\mathcal{T}'$ otra triangulación del mismo conjunto de puntos, se dice que $A(\mathcal{T}) > A(\mathcal{T}')$ si existe un índice i con $1 < i < 3m$ tal que

$$\alpha_j = \alpha'_j, \ \forall i < j, \quad \text{y } \alpha_i > \alpha'_i$$

esto es, el vector de ángulos de $\mathcal{T}$ es lexicográficamente mayor a $\mathcal{T}'$.

Una triangulación de Delaunay $\mathcal{DT}$ tiene la propiedad de

$$A(\mathcal{DT}) \geq A(\mathcal{T}),\ \forall \text{ triangulación } \mathcal{T} \text{de } \mathcal{P},$$

es decir, la triangulación de Delaunay es la que maximiza los ángulos, entre todas las posibles triangulaciones de $\mathcal{P}$. Esta es una propiedad deseable en muchas aplicaciones, de ahí su importancia.

Estas triangulaciones tienen una estrecha relación con los diagramas de Voronoi, específicamente, se tiene que una triangulación de Delaunay corresponde al grafo dual de este último [10].

Existen diversos algoritmos que se encargan de encontrar una triangulación de Delaunay dado un conjunto de puntos, también con orden $O(n \log n)$, con lo cual se puede encontrar el diagrama de Voronoi en $O(n)$ (dada la propiedad anterior de dualidad).

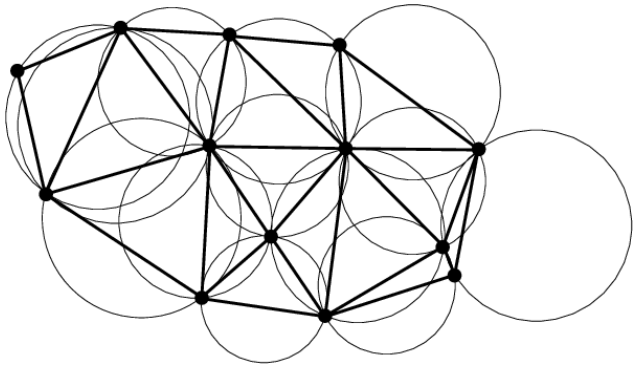


Figura 2.3: Triangulación de Delaunay. De *Delaunay Mesh Generation, Cheng, Siu-Wing et al, 2012, Chapman and Hall / CRC computer and information science series.*

Una operación importante que se utiliza bastante en el contexto de las triangulaciones de Delaunay es la que se detalla a continuación:

Edge flip

Sea $\overline{p_i p_j}$ una arista compartida por dos triángulos $p_i p_j p_l$ y $p_i p_j p_k$, los cuales forman un cuadrilátero convexo. Se dice que $\overline{p_i p_j}$ es ilegal si los triángulos que la comparten no cumplen la condición de Delaunay, o, equivalentemente, si se cumple la condición 2.1:

$$\angle p_i p_l p_j + \angle p_i p_k p_j > \pi \tag{2.1}$$

donde el símbolo $\angle$ representa el ángulo entre tres vértices.

La operación edge flip consiste en eliminar la arista $\overline{p_i p_j}$ y añadir la arista $\overline{p_l p_k}$, es decir, eliminar la diagonal y trazar la otra. Esto se muestra en la Figura 2.4.

Teorema 2.6 *Sea $\mathcal{T}$ una triangulación con una arista ilegal e. Sea $\mathcal{T}'$ la triangulación al hacer edge flip de la arista e en $\mathcal{T}$. Se tiene que*

$$A(\mathcal{T}') > A(\mathcal{T})$$

Con esto, se demuestra que se puede construir una triangulación de Delaunay a partir de cualquier triangulación inspeccionando cada una de las aristas y haciendo edge flip de las ilegales. Esto se deriva del hecho de que en cada operación de edge flip el vector de ángulos aumenta, y al haber un número finito de posibles aristas para un conjunto de puntos, en algún momento el algoritmo no podrá seguir aumentando el vector de ángulos, es decir, terminará. Esto también muestra que la operación edge flip se puede aplicar máximo una vez a una misma arista.

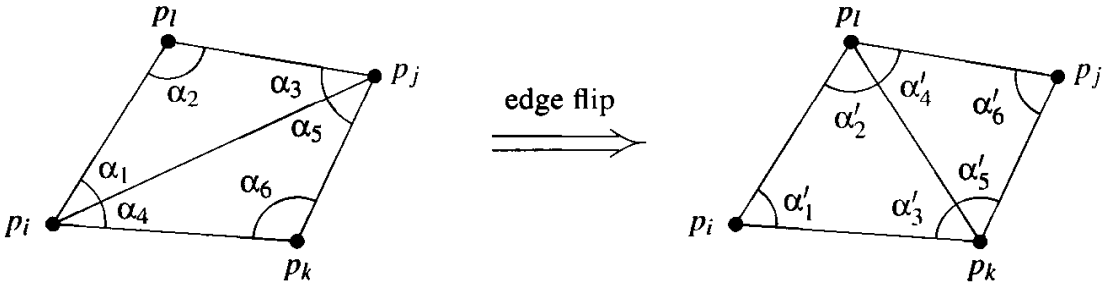


Figura 2.4: Operación de Edge Flip. Se tiene que $\min \alpha_i < \min \alpha'_i$, para $1 \leq i \leq 6$. De Computational Geometry Algorithms and Applications (2nd ed.), de Berg, van Kreveld, Overmars & Schwarzkopf, Springer.

2.2.3. Algoritmo Polylla

Un nuevo método de obtención de mallas poligonales es el algoritmo Polylla, algoritmo en el cual se enmarca esta memoria (aunque no se restringe solo a esto). Este corresponde a un método indirecto de generación de mallas[1], recibiendo una triangulación de cualquier tipo y generando una malla compuesta por polígonos convexos y no convexos. Esta es una de las principales diferencias respecto a las mallas de Voronoi mostradas con anterioridad

Esta restricción de recibir como input una triangulación puede parecer una desventaja frente a métodos directos, sin embargo hay varias aplicaciones disponibles que se encargan de generar triangulaciones de manera robusta y eficiente, como por ejemplo Triangle [22], con lo cual la utilización de métodos indirectos se vuelve una opción muy interesante para explorar nuevos tipos de mallas.

Para explicar como funciona el algoritmo Polylla, primero hay que definir el concepto de Lepp (Longest Edge Propagation Path) [4, 17] de un triángulo, el cual es la base de todo el algoritmo.

[1]Un método indirecto corresponde a aquel que recibe una triangulación inicial y a partir de esta construye una malla, a diferencia de los directos que reciben solo el conjunto de vértices y aristas.

Definición 2.7 *Dado un triángulo t_0 perteneciente a una triangulación Ω, el Lepp(t_0) corresponde a la secuencia ordenada de triángulos adyacentes t_0, t_1, ..., t_{l-1}, $t_l \in \Omega$, donde la arista que comparten los triángulos t_i y t_{i-1} es la más larga de t_i, para $i = 0, 1, 2, ..., l$.*

La secuencia termina cuando se encuentran los triángulos t_{l-1} y t_l, cuya arista compartida es la más larga en los dos (se ahondará más en esto más adelante). Esta se denomina terminal-edge y los triángulos t_{l-1} y t_l se denominan terminal-triangles. Un ejemplo de esto se puede ver en la Figura 2.5, con la arista AB siendo la terminal-edge (más larga de t_2 y t_3).

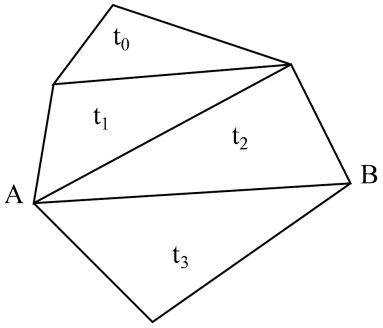


Figura 2.5: El longest edge propagation path del triángulo t_0 corresponde a la secuencia Lepp(t_0) = $\{t_0, t_1, t_2, t_3\}$. De Longest-edge algorithms for size-optimal refinement of triangulations [4].

Con esto, se puede definir el concepto de Terminal-edge region [2], que consiste en una región R formada por la unión de todos los triángulos t_i que tienen la misma terminal-edge. En la Figura 2.6 se puede ver un ejemplo de esto, con la terminal-edge region formada por el Lepp de 4 triángulos.

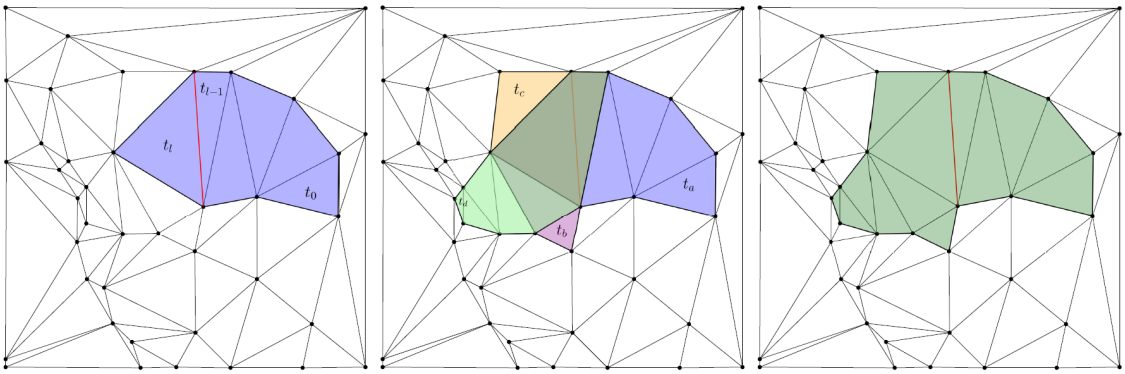


Figura 2.6: Región terminal-edge. (a) Lepp(t_0), con la terminal-edge marcada en rojo. (b) Lepp(t_0), Lepp(t_1), Lepp(t_2), Lepp(t_3), todos con el mismo terminal-edge. (c) Región terminal-edge formada por el Lepp de t_0, t_1, t_2 y t_3. De POLYLLA: polygonal meshing algorithm based on terminal-edge regions [21].

Además, una arista e puede ser clasificada, dependiendo del largo respecto a las demás aristas de los triángulos t_1 y t_2 que comparten e, de la siguiente manera:

- Terminal-edge: e corresponde a la arista más larga tanto del triángulo t_1 como de t_2, que como se mencionó, marca el fin de la secuencia del Lepp.
- Frontier-edge: e no es la arista más larga para ninguno de los triángulos.
- Internal-edge: La arista e es la más larga solo para uno de los dos triángulos t_1, t_2 que la comparten.
- Boundary-edge: e forma parte de un solo triángulo, lo que para efectos del algoritmo Polylla es considerado un frontier-edge.

Con esto, el algoritmo Polylla se puede dividir en tres fases:

Fase de clasificación

A partir de la triangulación que se recibe como input, el primer paso es clasificar cada una de las aristas según el criterio mencionado con anterioridad, es decir, determinando si es terminal, frontier o internal edge. Para esto, se recorre cada uno de los triángulos (lo cual se realiza utilizando la estructura de datos half-edge) identificando cual es la arista más larga, y luego se recorre nuevamente cada arista marcándola según su tipo. Además, se almacena una arista semilla para cada una de las terminal-edge region encontradas (ver Figura 2.7).

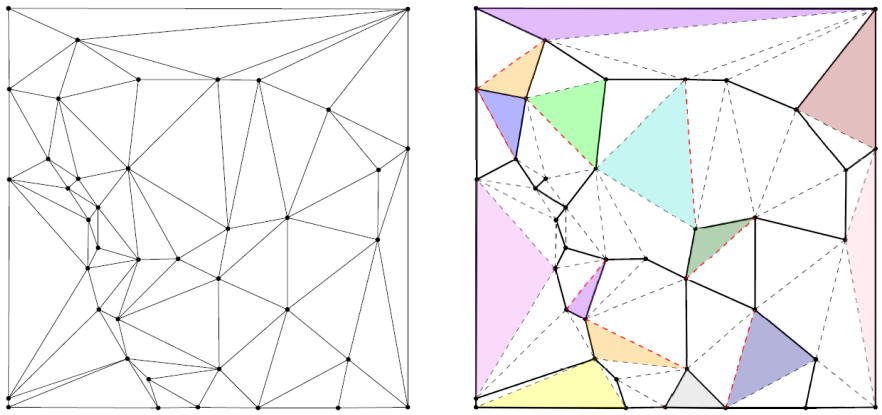


Figura 2.7: Fase de clasificación. (a) Triangulación que se recibe como input. (b) Resultado después de clasificar cada una de las aristas: Las líneas punteadas son internal-edge, las sólidas frontier-edge y las rojas terminal-edge. De POLYLLA: polygonal meshing algorithm based on terminal-edge regions [21].

Fase de viaje

Una vez etiquetada cada una de las aristas de la triangulación, se construyen realmente cada uno de los polígonos, ya que hasta el momento cada arista sigue asociada a su respectivo triángulo de la triangulación inicial. Para lograr esto, se utiliza cada una de las aristas semilla obtenida en la fase anterior, y se recorren los triángulos en busca de las frontier-edge, para luego poder modificar las adyacencias de la estructura half-edge, con lo cual se generan finalmente los polígonos. En la Figura 2.8 se puede ver el recorrido realizado para construir el polígono dado por la terminal-edge region.

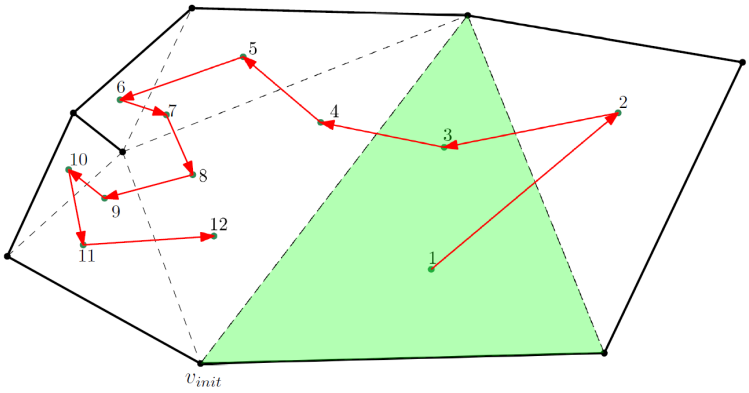
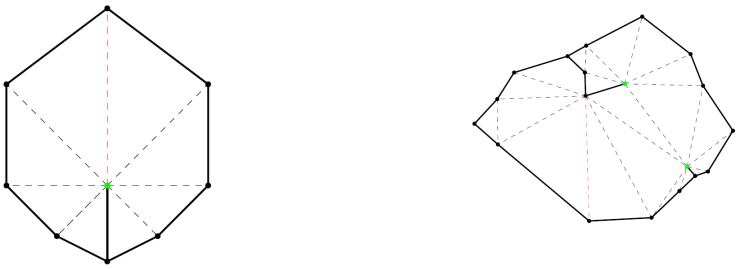


Figura 2.8: Fase de viaje. El triángulo verde es el que contiene la arista semilla, por lo que se empieza el viaje desde ahí. De POLYLLA: polygonal meshing algorithm based on terminal-edge regions [21].

Fase de reparación

En la fase de viaje los polígonos generados pueden ser no simples, específicamente, pueden contener vértices que son el extremo de solo una arista (Figura 2.9). Para solucionar esto, la fase de reparación recorre todos los polígonos haciendo lo siguiente:



(a) (b)

Figura 2.9: Ejemplos de polígonos no simples que pueden ser generados en la fase de viaje. De POLYLLA: polygonal meshing algorithm based on terminal-edge regions [21].

1. Determina si el polígono es no simple (es decir, si hay un vértice que tiene solo una arista incidente).
2. Determina el grado del vértice en la triangulación inicial y escoge la arista incidente del medio. Por ejemplo, en la Figura 2.9 (a), la arista del medio sería la marcada en rojo. En caso de que haya un número par de aristas incidentes, se escoge cualquiera de las dos.
3. Se agrega esta arista como nueva frontier-edge, formando un nuevo polígono.

En la Figura 2.10 se puede ver el proceso de la fase de reparación, en que un polígono no simple se divide en 4 polígonos simples.

Las mallas Polylla han sido comparadas con las de Voronoi en cuanto a la forma, número de polígonos y número de puntos, mostrando que para la misma triangulación inicial, la malla

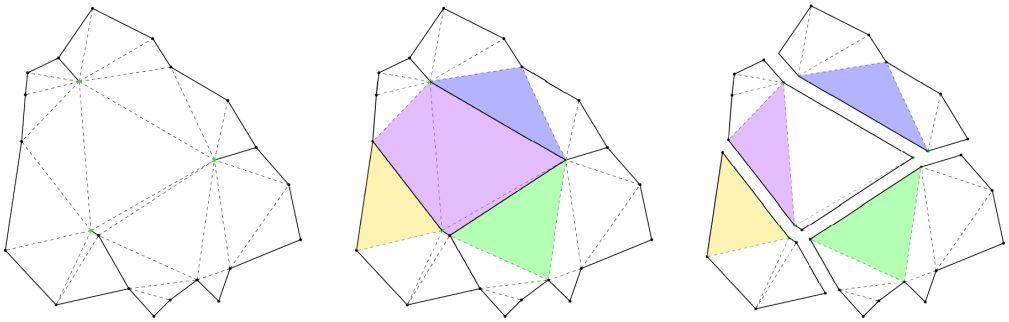


Figura 2.10: Reparación de un polígono no simple. (a) Polígono resultante de la fase anterior (con los vértices a reparar en verde). (b) Se insertan las aristas del medio de cada vértice. (c) Se guardan los nuevos polígonos como polígonos independientes. De POLYLLA: polygonal meshing algorithm based on terminal-edge regions [21].

Polylla contenía menos polígonos que una de Voronoi. Además, el algoritmo Polylla es más simple y rápido que los algoritmos para generar mallas de Voronoi, lo cual es significativo en modelaciones físicas donde el tiempo de proceso es un factor a considerar. En esta línea, el desempeño numérico de ambas al utilizar el método del elemento virtual es similar, lo cual fue probado resolviendo la ecuación de Laplace en un dominio con forma de L.

2.3. Refinamiento de mallas

Para asegurar que las mallas generadas cumplan con las características deseadas en relación al uso que se le darán, se han desarrollado métodos que se encargan de, dada una malla inicial (generalmente una triangulación), modificarla para mejorar alguna de estas características. Este proceso, conocido como refinamiento, se puede realizar de manera iterativa hasta que todos los polígonos cumplan algún criterio definido.

Existen varios métodos para cumplir con este objetivo, donde los algoritmos de refinamiento de Delaunay [8, 18] han sido ampliamente estudiados y utilizados. Como una alternativa que ha mostrado resultados prácticos similares a estos, están los algoritmos de refinamiento de Lepp, que también pueden mantener una triangulación de Delaunay a lo largo del proceso y es el tema de estudio de esta memoria.

El funcionamiento, a grandes rasgos, de los algoritmos de Lepp se muestra en el Algoritmo 1.

Los algoritmos de Lepp que mantienen una triangulación de Delaunay a lo largo del proceso se basan en las *inserciones de Delaunay*, que consisten básicamente en insertar un punto y hacer edge flip de las aristas adyacentes que se hayan vuelto ilegales, lo cual se va propagando hasta que no se encuentre ninguna más. Notar que mediante el Teorema 2.6 se puede demostrar el término de este algoritmo. Un pseudocódigo de este procedimiento se muestra en el Algoritmo 2.

Algoritmo 1 Algoritmo genérico de refinamiento de Lepp

Input: Triangulación τ, conjunto S de triángulos a refinar
Output: Triangulación τ' refinada

```
for t in S do
    while t no ha sido refinado do
        t_1, t_2 ← triángulos terminales de Lepp(t)
        refinar t_1, t_2
    end while
end for
```

Algoritmo 2 Inserción de Delaunay

Input: Triangulación de Delaunay τ, punto p a insertar
Output: Triangulación de Delaunay τ' con el nuevo punto p

```
Insertar p
S ← aristas que ahora pueden ser ilegales
while S no es vacío do
    se extrae una arista edge de S
    if edge es ilegal then
        edgeflip(edge)
        actualizar S con las nuevas posibles aristas ilegales
    end if
end while
```

Para el trabajo de esta memoria, hay dos algoritmos de Lepp importantes, cuya principal diferencia es el método utilizado para insertar puntos. Estos algoritmos pueden funcionar con aristas restringidas o no restringidas, donde las primeras corresponden a segmentos que deben estar en la triangulación final.

2.3.1. Lepp centroid Delaunay

El algoritmo Lepp centroid Delaunay [16] funciona con el concepto de Lepp y las inserciones de Delaunay [6], teniendo tres operaciones que se aplican en el paso 2 mencionado con anterioridad:

- Operación 1: Se calcula el centroide del cuadrilátero formado por dos terminal-triangles no restringidos (es decir, sus aristas no necesariamente tienen que estar en la triangulación final) y se realiza una inserción de Delaunay en aquel punto. En la Figura 2.11 se muestra esta operación para los triángulos t_1 y t_2.
- Operación 2: Dados dos terminal-triangles, donde uno de ellos tiene su segunda arista más larga E restringida, se realiza una inserción de Delaunay restringida de su punto medio.
- Operación 3: Para una terminal-edge restringida, se realiza una Lepp bisection de los terminal-triangles (explicada en la sección 2.3.2)

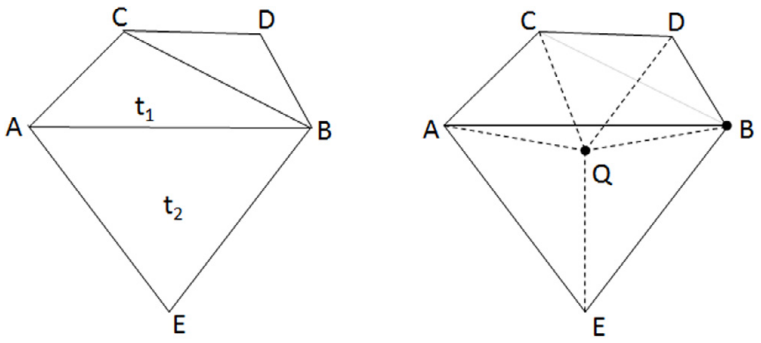


Figura 2.11: Inserción de Delaunay del centroide de los triángulos terminales.

2.3.2. Lepp bisection

Este algoritmo es muy parecido al anterior, solo que se utiliza la operación de Lepp bisection para insertar puntos. Esta operación es mucho más simple que las que se aplican en el algoritmo anterior ya que no es necesario calcular el centroide y la terminal-edge no se destruye (solo se añade un vértice en su punto medio), por lo que no importa si es una arista restringida o no. El método consiste en lo siguiente: Dado un triángulo ABC con arista más larga AB, la Lepp bisection de ABC se lleva a cabo agregando una arista que une el punto medio de AB con el vértice C (es decir, agregando la mediana de AB). Con esto, Lepp bisection realiza esta operación en los dos terminal-triangles (o en uno, en caso que la terminal-edge sea de borde). En la Figura 2.12 se puede ver un ejemplo de esto. Al igual que en el método anterior, se puede hacer una inserción de Delaunay (en la Figura 2.12 se muestra la inserción simple).

2.4. Descomposición de polígonos cóncavos en partes convexas

Otro problema en que se busca particionar un dominio en polígonos corresponde a la descomposición de polígonos cóncavos en convexos, problema que lleva décadas siendo estudiado [14]. Una triángulación es una partición de un polígono arbitrario en partes convexas, sin embargo, es una división en el máximo número de partes posibles. La descomposición de polígonos cóncavos en partes convexas, en cambio, busca los contrario: encontrar la partición con el menor número de polígonos.

Este problema ha demostrado ser bastante complicado, donde el tiempo de los algoritmos y su capacidad de acercarse al óptimo son dos propiedades que van en desmedro de la otra. Más aun, los algoritmos que se acercan a la solución óptima suelen ser tan enrevesados que en la práctica no son una buena opción [9].

Para resolver este problema se han utilizado dos métodos:

- Descomponer añadiendo puntos al polígono (llamados puntos de Steiner) (ver Figura

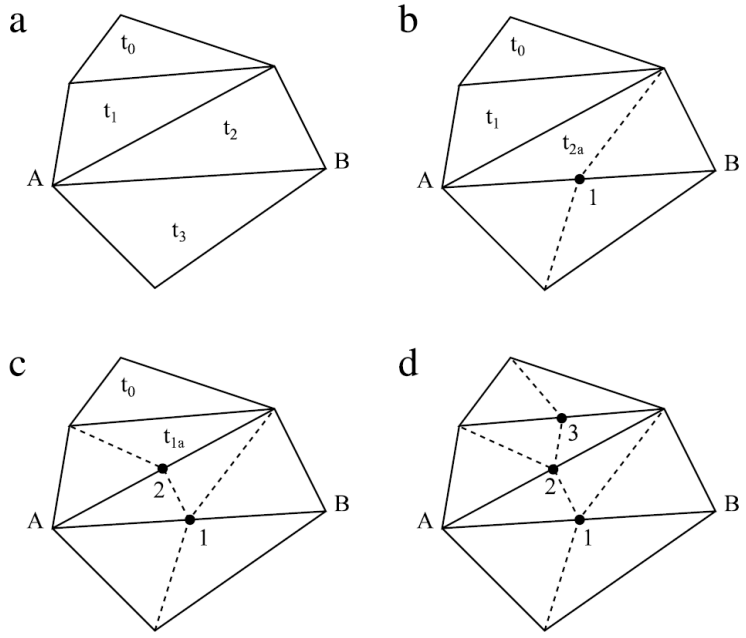


Figura 2.12: Refinamiento del triángulo t_0 mediante Lepp bisection. En b) se añade el vértice y las medianas de los triángulos terminales t_2 y t_3. En c) se añade el vértice de los triángulos t_{2a} y t_1 y finalmente en d) se añade el vértice que refina el triángulo objetivo, dando como resultado la malla final.

2.13).

- Encontrar una partición solo a partir de diagonales (sin añadir ningún punto).

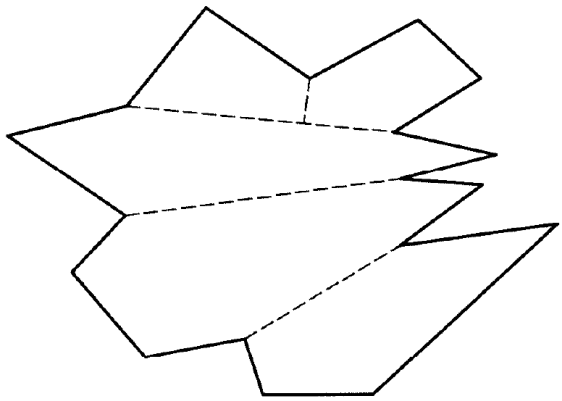


Figura 2.13: Descomposición de un polígono cóncavo añadiendo puntos de Steiner. De Computational Geometry in C, Joseph O'Rourke, Cambridge University Press, 2 edition, 1998.

Chazelle y Dobkin [7] diseñaron un algoritmo que, siguiendo la primera estrategia, descompone un polígono de n vértices con el mínimo número de partes convexas en $O(n^3)$ pero, como se mencionó, este algoritmo es muy complicado de implementar.

Por otro lado, uno de los primeros algoritmos sencillos y rápidos fue propuesto por Hertel y Mehlhorn [11], que consiste en, a partir de una triangulación, eliminar todas las aristas que no sean esenciales. Una arista se considera no esencial si al eliminarla no genera ningún polígono cóncavo. Este algoritmo construye una partición de máximo cuatro veces el óptimo y toma un tiempo de $O(n)$.

Capítulo 3

Problema y Metodología

En este capítulo se describe el problema a resolver y algunos requisitos que debe satisfacer la solución implementada,además de algunas alternativas de solución que fueron consideradas pero finalmente descartadas. Finalmente se muestra la metodología de trabajo seguida durante el desarrollo de esta memoria, junto a la arquitectura de la solución implementada.

3.1. Planteamiento

En esta memoria se trabajan dos problemas principales que se enmarcan dentro de Polylla. Este algoritmo genera mallas compuestas por polígonos convexos y no convexos (Figura 3.1), siendo justamente estos últimos los que representan un problema en la aplicación de muchos métodos numéricos clásicos, por lo que se necesita desarrollar algún método que permita refinar estos polígonos para obtener solo partes convexas. Como Polylla trabaja a partir de una triangulación inicial eliminando aristas, lo más natural es desarrollar un método de refinamiento mediante inserción de aristas, siendo este el primer problema a resolver.

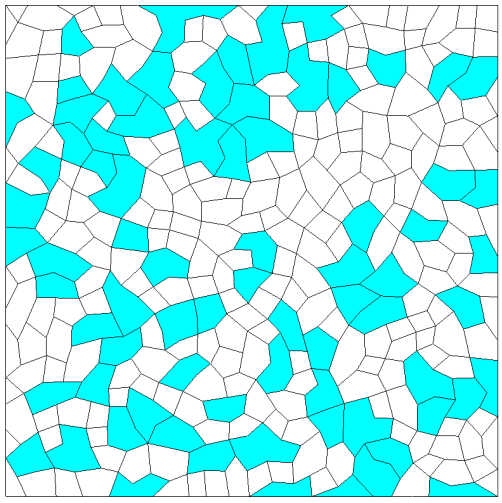


Figura 3.1: Malla Polylla. En celeste los polígonos cóncavos que deben ser refinados.

El segundo problema abordado consiste en poder refinar mallas Polylla mediante inserción de puntos. Los métodos de refinamiento de mallas generalmente trabajan exclusivamente sobre triangulaciones, por lo que este problema se reduce a poder refinar la triangulación que se utiliza como input de Polylla. Como este mallador está basado en el concepto de Lepp, los métodos de refinamiento que se necesitan implementar también se basan en este. Junto a esto, se necesitan desarrollar métricas que controlen este proceso.

3.2. Requisitos

La solución desarrollada debe cumplir con una serie de requisitos funcionales y no funcionales que se detallan a continuación:

Requisitos funcionales

- Debe funcionar dentro de la implementación actual del algoritmo, por lo que todas las funcionalidades actuales se deben mantener.
- Se debe poder señalar regiones específicas a ser refinadas.
- Se debe poder obtener información cualitativa que caracterice las mallas generadas (como número de polígonos, ángulos mínimos y máximos, etc).
- Se debe poder especificar alguna métrica de calidad que las mallas generadas deban cumplir.
- Se debe contar con una interfaz de línea de comandos que permita utilizar el programa de manera sencilla.

Requisitos no funcionales

- La solución debe ser implementada en el lenguaje C++.
- Debe seguir buenas prácticas de programación que permitan ampliar el trabajo de forma fácil.
- La adición de nuevas funcionalidades debe modificar lo menos posible la implementación actual.

3.3. Propuestas

Para el refinamiento de mallas con polígonos cóncavos, se estudiaron otros algoritmos que se encargan de resolver el problema de la descomposición convexa, como el sugerido por Fernández [9] o Wijeweera [23], que pueden obtener un menor número de polígonos convexos. Sin embargo, estos algoritmos pueden trazar diagonales que no están presentes

en la triangulación inicial, con lo cual la estructura general de la malla se vería modificada y un posible refinamiento mediante inserción de aristas sería aún más complejo. Por otra parte, estos algoritmos solo buscan minimizar el número de polígonos finales, lo cual le quita versatilidad a la solución de poder refinar en función de, por ejemplo, los ángulos.

Sumado a lo anterior, se tiene que estos algoritmos son más complicados que una solución basada en la triangulación subyacente de la malla Polylla, lo cual habría significado más tiempo de implementación. Debido a estas razones, estas soluciones se descartaron.

En este capítulo se presenta la metodología de trabajo seguida durante el desarrollo de esta memoria, junto a la arquitectura de la solución implementada.

3.4. Metodología

La metodología utilizada para llevar a cabo la memoria se basó principalmente en dos modelos: En la primera fase se utilizó un método de cascada y en la segunda parte, ya casi llegando al final, se pasó a un método incremental.

Dada la naturaleza del problema a resolver, el método de cascada fue casi una demanda, ya que mucho de lo implementado se basa en una serie de funcionalidades básicas, es decir, que de no haber resuelto estas primero, simplemente no había posibilidad de continuar.

Ya hacia el final, cuando las funcionalidades esenciales fueron implementadas, se pasó a un enfoque incremental, en el cual se fueron añadiendo de manera progresiva más características a la solución e incluso se fueron modificando partes ya desarrolladas, ya sea por encontrar directamente fallas en su funcionamiento o bien por encontrar métodos que aumentaban la eficiencia.

En cuanto a las herramientas utilizadas, como la implementación actual del algoritmo Polylla está en C++, este fue el lenguaje de programación utilizado en la totalidad de la memoria, con excepción de los experimentos y gráficos, los cuales fueron realizados en Python.

Un aspecto importante a mencionar respecto a la metodología de trabajo hace referencia a la manera en que se testeaba lo implementado. Esta es una tarea bastante difícil de desarrollar en muchos problemas de geometría computacional, ya que, por ejemplo, no es fácil establecer casos de prueba para seguir un enfoque de desarrollo guiado por pruebas (o test-driven development). Por lo tanto, para probar lo desarrollado, la mayoría de las veces se probaba la solución sobre casos muy pequeños (por ejemplo, una malla con menos de 10 triángulos) y luego se trataba de ampliar a casos más grandes.

3.5. Arquitectura

La solución planteada sigue el paradigma de programación orientada a objetos, en el cual se definieron una serie de clases que buscan encapsular de mejor manera la funcionalidad,

además de permitir una fácil ampliación con nuevas funcionalidades.

El esquema general hereda dos clases de la implementación anterior de Polylla [19], que son las clases `Triangulation` y `Polylla`. Esta última era la responsable de todo el funcionamiento del algoritmo, tomando siempre como insumo una instancia de la primera.

Lo primero que hay que destacar es que las clases implementadas en esta memoria trabajan exclusivamente con la clase `Triangulation`, ya que toda la funcionalidad referida al refinamiento de mallas Polylla fue añadida como métodos dentro de esta. En la Figura 3.2 se puede ver el diagrama completo de la solución.

Primero se tiene la clase `Criterion`, que corresponde a una interfaz que establece los métodos necesarios para determinar cuando un triángulo es de mala calidad, es decir, que debe ser refinado. La propiedad más importante que proveen estas clases corresponde a la métrica. Para este trabajo la única métrica considerada fue el área, sin embargo, hay muchos otros criterios de calidad que se pueden necesitar dependiendo de la aplicación, motivo por el cual esta interfaz es muy útil para una fácil incorporación de estos.

Luego se tiene la clase `Region`, que se encarga de representar una región del plano, determinando que triángulos pertenecen al área que requiere refinamiento. Hay cuatro clases que heredan de esta: `Rectangle`, `Circle` y `Polygon`. Si bien la clase `Rectangle` parece redundante ante la presencia de la clase `Polygon`, esta fue implementada por simplicidad para el usuario.

`BaseQueue` corresponde a una interfaz que establece el comportamiento de las colas de prioridad utilizadas para el refinamiento, lo cual será visto en más detalle en el siguiente capítulo.

Por último, la clase `Refiner` provee todos los métodos involucrados en los algoritmos de refinamiento, como la inserción de puntos, el cálculo del Lepp o la operación edge flip. Cabe destacar que para el desarrollo de la interfaz de línea de comandos, se creó una clase denominada `parser`, la cual se encarga de corroborar el correcto uso de cada una de las opciones del programa.

A estas nuevas clases, se suman distintos métodos añadidos a las clases `Triangulation` y `Polylla`, siendo principalmente métricas en el caso de la primera y el refinamiento a través de aristas en el caso de la segunda.

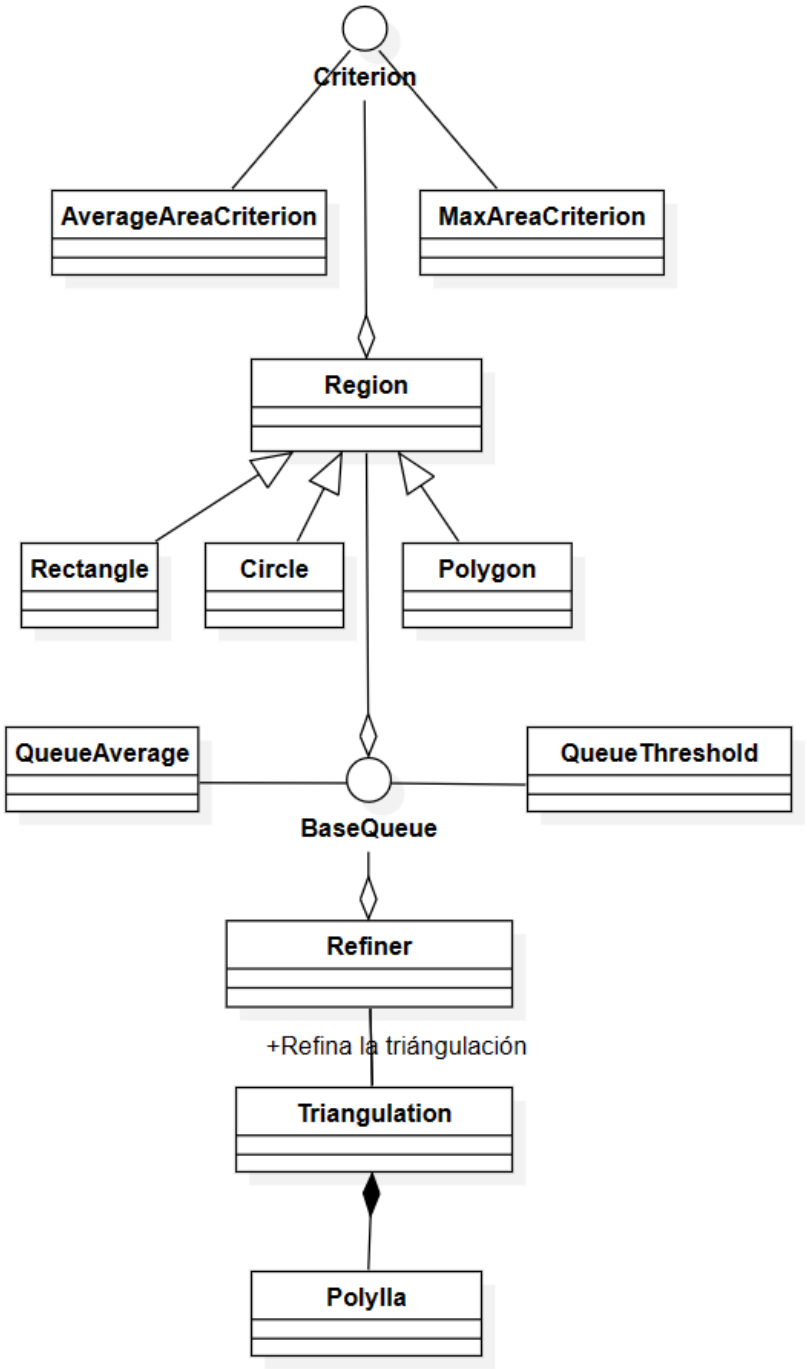


Figura 3.2: Diagrama de clases.

Capítulo 4

Diseño e implementación

En este capítulo se describe detalladamente el funcionamiento de la solución implementada. En primer lugar se muestran las estructuras de datos utilizadas. En segundo lugar se especifican las fórmulas utilizadas para el cálculo de algunas propiedades geométricas. En tercer lugar se da una descripción detallada de cada uno de los algoritmos implementados. Finalmente se muestra como usar el programa, especificando cada una de las opciones disponibles.

4.1. Estructuras de Datos

A continuación se presentan las distintas estructuras de datos que son utilizadas en la implementación de los algoritmos, algunas de las cuales vienen de la implementación anterior y otras implementadas para el actual trabajo.

4.1.1. Lista de aristas doblemente conectadas

Una lista de aristas doblemente conectada [13] (o DCEL por sus siglas en inglés, o estructura half-edge) es una de las estructuras que se utilizaban desde la implementación anterior, y es crucial para la implementación de los algoritmos de manera eficiente y sencilla. Permite modelar cualquier tipo de malla poligonal, permitiendo una fácil navegación entre aristas, vértices y caras (entiéndase una cara como un polígono de la malla), esto es, dada una cara, vértice o arista, se puede acceder de forma rápida a cualquier elemento incidente en el. El concepto de incidencia aquí hace referencia a cualquier elemento que esté "unido" a otro. Por ejemplo, dada una cara, una arista es incidente a ella si esta forma parte del polígono que forma esa cara.

Para esto, las aristas se modelan como dos aristas dirigidas en sentidos opuestos (denominadas half-edges o semiaristas), así cada una puede pertenecer al respectivo polígono que comparte esa misma arista (ver Figura 4.1). Se dice que estas dos semiaristas son gemelas (o twin en inglés). Una DCEL modela una semiarista guardando la información de su vértice de

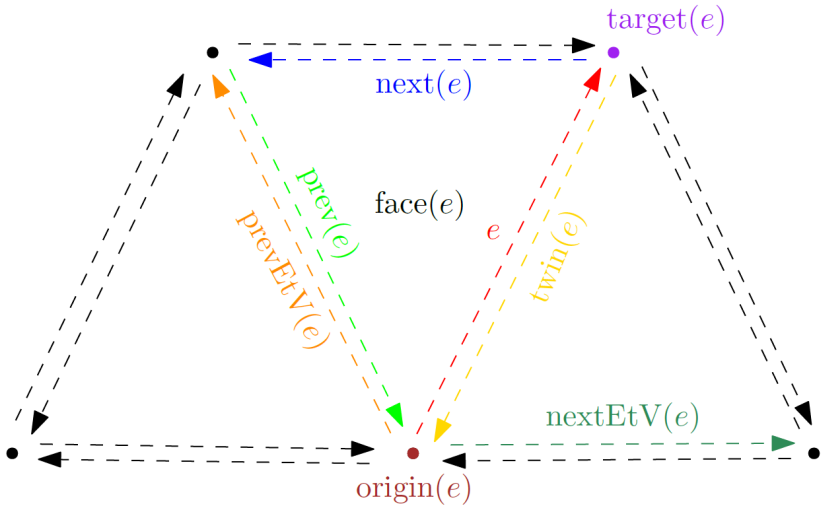


Figura 4.1: Visualización de una DCEL. La operación `nextEtV(e)` corresponde a la siguiente arista que tiene como origen a `origin(e)` en el sentido de las manecillas del reloj. Esta operación se puede implementar a partir de la información básica contenida en la estructura DCEL. De Generation of polygonal meshes in compact space [20].

origen, su arista gemela, la cara a la que pertenece, la siguiente semiarista dentro de la cara a la que pertenece (recordar que son dirigidas) y la semiarista previa. También se guarda si la semiarista es de borde, lo cual permite navegar a través del borde de toda la región modelada. Los vértices, por su parte, se modelan con una estructura, en que se guardan sus coordenadas, una referencia a cualquier semiarista incidente a él y si es de borde o no.

Las semiaristas son almacenadas en la clase `vector` de la librería estándar de C++, que se puede entender como un arreglo que puede modificar su tamaño, lo que permite un acceso en tiempo constante a cada valor dado su índice. Para almacenar los vértices se utiliza esta misma estructura.

En este caso particular, esta estructura debe permitir la inserción de puntos y, por ende, de aristas y caras, por lo cual estas últimas también se almacenan en una estructura aparte para poder iterar sobre cada triángulo[1].

Cada cara corresponde a un puntero a alguna semiarista del triángulo que representa. Para almacenarlas se utilizó la clase `set` de la librería estándar de C++, la cual garantiza un tiempo logarítmico en el número de elementos para las búsquedas. Este requisito se debe principalmente a la operación de edge flip, lo cual será explicado más adelante.

4.1.2. Cola de prioridad

En los algoritmos de refinamiento para saber cual es el siguiente triángulo que corresponde refinar, se necesita guardar los triángulos considerados malos junto con su métrica, y poder

[1]Si la DCEL no va a ser modificada, se pueden ordenar las semiaristas de forma que se pueda iterar por cada triángulo sin necesidad de mantener una lista de caras

saber cuál es el triángulo con peor métrica (este es el próximo a refinar). Esto requiere una estructura de datos capaz de entregar el elemento con mayor prioridad de manera rápida.

Además de esto, durante el refinamiento, muchos triángulos se ven afectados, ya sea al refinar los triángulos del Lepp del triángulo objetivo, o por los sucesivos edge flips pueden llegar a realizarse. Esto genera que se deba actualizar la métrica de muchos triángulos, lo cual modifica la prioridad de estos.

Estos requisitos implican que la estructura, la cual llamaremos PQ para distinguirla de las colas clásicas, debe cumplir con estos dos requisitos:

- Extraer eficientemente el triángulo con peor métrica
- Buscar eficientemente un triángulo

Para cumplir con el primer requisito, se puede implementar alguna cola de prioridad que tenga como llaves las métricas y como valor su respectivo triángulo (notar que pueden haber llaves duplicadas, ya que dos triángulos pueden tener la misma métrica). A esta estructura la denominaremos M. Para el segundo requisito se consideró utilizar hashing o un árbol binario balanceado, teniendo como llaves el triángulo y como valor un puntero al nodo que contiene la métrica de ese triángulo en la cola de métricas. El requisito de que el árbol sea balanceado es que en general es probable que las llaves sean insertadas en orden ascendente, con lo cual el costo de búsqueda podría ser cercano a lineal. A esta estructura la denotaremos como T.

En otras palabras, PQ está formada por la estructura M y la estructura T. Para M se utilizó la clase `multimap` de la librería estándar de C++, la cual tiene la propiedad de guardar sus llaves en orden, permitiendo obtener su elemento de mayor o menor prioridad en tiempo constante, además de soportar llaves duplicadas. La estructura T se implementó con la clase de C++ `unordered_map`, el cual funciona mediante hashing. En el siguiente capítulo se hablará un poco más acerca de la diferencia en tiempo de estas estructuras.

Las operaciones implementadas en la estructura PQ son:

- Inserción: La inserción recibe un triángulo y una métrica y realiza una inserción en M y luego una inserción en T. Esto toma tiempo $O(log(n))$.
- Eliminación: Recibe un triángulo, lo busca y elimina de T y luego lo elimina de H en $O(log(n))$.
- Buscar el elemento de mayor prioridad: Lo busca en H en $O(1)$.

Se implementó una última operación `update`, que recibe una lista de pares de triángulos a actualizar, es decir, que se debe calcular nuevamente sus métricas y ver si siguen perteneciendo a la región objetivo. Estos pares pueden contener dos triángulos distintos, lo cual indica que el primero "se convirtió" en el segundo triángulo, caso que se puede dar con la operación edge flip, que se explicará en la sección 4.3.3. Está toma tiempo $O(log(n))$ para cada par, por lo tanto toma $O(mlog(n))$, con m siendo el número de pares entregados.

Estas son las dos mayores componentes de PQ, sin embargo, estas estructuras (implementadas como clases) también guardan información y clases extra que otorgan cierta lógica para el control del refinamiento.

4.2. Cálculo de propiedades geométricas

Los algoritmos implementados se basan en el cálculo de algunas propiedades geométricas básicas, que se detallan a continuación con sus fórmulas.

4.2.1. Área de un triángulo

El área de un triángulo abc se obtiene con la siguiente fórmula:

$$\acute{A}rea(abc) = \frac{1}{2}\left|(b_x - a_x)(c_y - a_y) - (c_x - a_x)(b_y - a_y)\right| \tag{4.1}$$

donde V_x y V_y representan las coordenadas x e y del vértice V, respectivamente.

4.2.2. Centroide de un triángulo

El centroide [5] de un triángulo abc corresponde al punto de intersección de sus tres medianas, dado por la siguiente expresión:

$$(\overline{x}, \overline{y}) = \left(\frac{a_x + b_x + c_x}{3}, \frac{b_x + b_y + c_y}{3}\right) \tag{4.2}$$

4.2.3. Ángulo formado por dos aristas

Para calcular el ángulo formado por dos aristas se utiliza el producto punto entre dos vectores a y b, dado por la siguiente fórmula:

$$a \cdot b = |a||b| \cos\theta$$

donde θ corresponde al ángulo. A partir de esto, el ángulo que forman dos aristas AB y BC se obtiene de:

$$\theta = \arccos\left(\frac{\vec{AB} \cdot \vec{BC}}{|\vec{AB}||\vec{BC}|}\right)$$

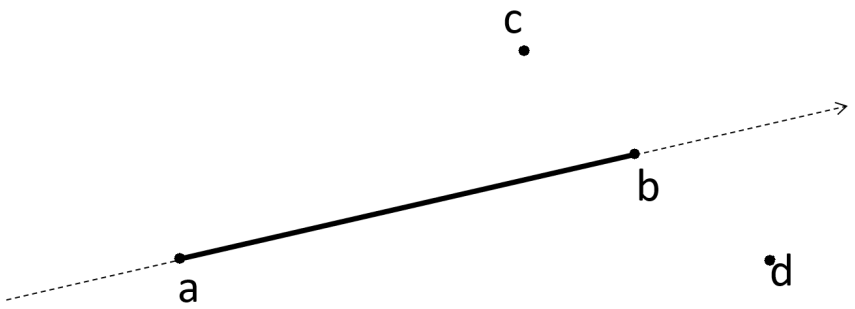


Figura 4.2: Los puntos (a, b, c) están en sentido antihorario, es decir, el punto c está a la izquierda a la recta que contiene ab. Por el contrario, el punto d está en sentido horario, por lo tanto está a la derecha.

Finalmente, la fórmula directa que se ocupa para calcular el ángulo entre dos vectores es:

$$\theta = \arccos\left(\frac{(A_x - B_x)(C_x - B_x) + (A_y - B_y)(C_y - B_y)}{\sqrt{(A_x - B_x)^2 + (A_y - B_y)^2}\sqrt{(C_x - B_x)^2 + (C_y - B_y)^2}}\right) \tag{4.3}$$

4.2.4. Orientación de tres puntos

Sea (a, b, c) una tripleta de tres puntos. Se busca encontrar en que sentido están ordenados, es decir, si los tres puntos forman un circuito en sentido antihorario o no. Para esto, se puede utilizar el determinante de los vectores formados por estos puntos (en orden). Esto corresponde justamente al doble del área del triángulo calculado en la ecuación 4.1, pero sin tomar el valor absoluto, con lo que se tiene que (a, b, c) están en sentido antihorario [14] si la expresión

$$(b_x - a_x)(c_y - a_y) - (c_x - a_x)(b_y - a_y) \tag{4.4}$$

es mayor a 0. Si es menor a 0, quiere decir que están orientados en sentido horario. En caso que sea 0, implica que los tres puntos son colineales.

El que (a, b, c) esté orientado en sentido antihorario es equivalente a decir que el punto c está "a la izquierda" del segmento ab, considerando que el segmento es dirigido (es decir, va de a hacia b). En la Figura 4.2 se ve un ejemplo de esto.

4.3. Algoritmos

Para poder describir en detalle los algoritmos implementados, primero se presenta el funcionamiento general del refinamiento, desglosando sus partes constituyentes a continuación.

El primer paso es insertar en la cola de prioridad, la cual denominaremos como Q, los triángulos que correspondan ser refinados, por lo que al finalizar este paso estarán en Q todos los triángulos que pertenezcan a la región definida y que no cumplan con el criterio de calidad establecido.

Ahora, se extrae un triángulo de Q y se realiza el refinamiento de Lepp. Es importante notar que en este paso muchos triángulos se pueden ver modificados, los cuales deben ser correctamente actualizados en Q.

Luego de esto, se verifica si se ha cumplido con el criterio de refinamiento impuesto, en cuyo caso el algoritmo termina. En caso contrario, se extrae un nuevo triángulo de Q y se repite el proceso. Esto se muestra como pseudocódigo en el Algoritmo 3.

Algoritmo 3 Refinamiento de triangulación

Input: Triangulación τ
Output: Triangulación τ refinada

```
1: Q ← ∅
2: for t in τ do
3:     if t tiene que ser refinado then
4:         Insertar t en Q
5:     end if
6: end for
7: while τ no está refinada do
8:     t ← primer elemento de Q
9:     Refinar t
10:    Actualizar Q
11: end while
```

4.3.1. Cálculo de triángulos a refinar

El primer paso del algoritmo de refinamiento es determinar los potenciales triángulos a refinar (ya se explicará porque se dice potenciales). Para esto, hay dos aspectos que se deben tomar en cuenta: la zona de la triangulación definida y el criterio que se encarga de determinar si un triángulo debe ser refinado respecto al área.

Primero, un triángulo se considera dentro de la región definida por el usuario si cualquiera de sus vértices está dentro de esta región. Esto quiere decir que pueden haber triángulos contenidos parcialmente dentro de la región a refinar, sin embargo no serán parte de los triángulos objetivo a refinar (ver Figura 4.3). Esta decisión de implementación tiene como fundamento la simplicidad y velocidad del algoritmo, ya que que generar de manera precisa la intersección de un polígono con cada triángulo habría sido muy costoso.

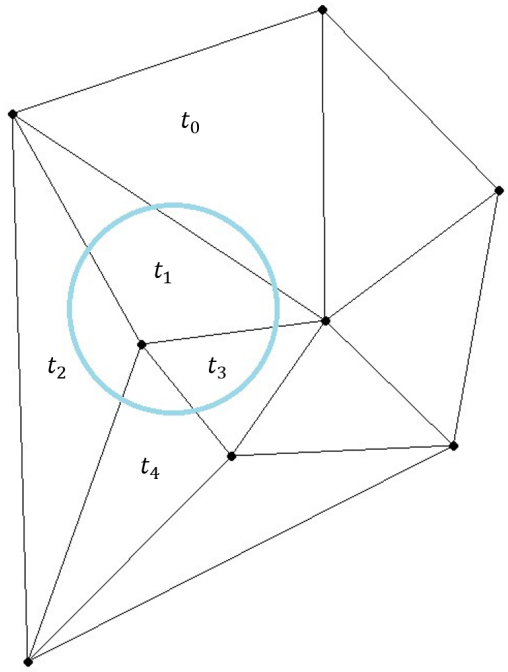


Figura 4.3: Determinación de los triángulos a refinar a partir de una región. La región a refinar corresponde al círculo celeste, y los triángulos a refinar son t_1, t_2, t_3 y t_4. Si bien el círculo intersecta al triángulo t_0, ninguno de los vértices de este último están dentro de la región, por lo cual no se considera para el refinamiento.

Si el triángulo resulta estar dentro de la región de refinamiento, se calcula su área utilizando la fórmula 4.1. Ahora, hay dos criterios que se pueden utilizar respecto a esta:

- Área máxima que puede tener un triángulo
- Refinar los triángulos en orden hasta que el área promedio se haya reducido en un factor f.

En el caso del primero, el triángulo se debe refinar si su área es mayor a este valor, en el segundo caso en cambio, todos los triángulos son considerados como objetivos a refinar, ya que no se sabe de antemano como evolucionarán las áreas de los triángulos refinados. Por esta razón, todos son potenciales triángulos a refinar, ya que puede que muchos de ellos no sean finalmente refinados.

Si se cumplen estas condiciones, el triángulo se inserta en Q y se repite el proceso con otro triángulo de la triangulación. Esto se describe en el Algoritmo 4.

Ray crossings

La línea 2 del algoritmo 4 corresponde a determinar si alguno de los vértices del triángulo τ está dentro de la región R, para lo cual se implementó el algoritmo *ray crossings* [14].

Algoritmo 4 Cálculo de triángulos a refinar

Input: Triangulación τ, region R, criterio CR
Output: Cola Q con todos los triángulos que deben ser refinados

```
for t in τ do
    if t in R then
        if t no cumple con CR then
            insert t in Q
        end if
    end if
end for
```

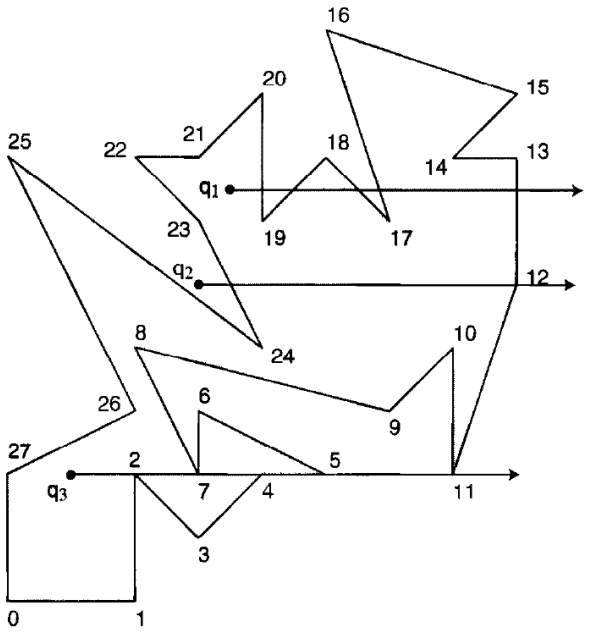


Figura 4.4: Aplicación del algoritmo ray crossings sobre un polígono arbitrario. De Computational Geometry in C, Joseph O'Rourke, Cambridge University Press, 2 edition, 1998.

Este se basa en una idea sencilla: trazar un rayo horizontal r desde q y contar el número de veces que intersecta el polígono. Cada intersección se puede entender como el punto saliendo o entrando en el polígono (en ese orden). Entonces, si después de contar todas las intersecciones r sale del polígono, quiere decir que q estaba dentro. Es decir, si el número de intersecciones es impar q está dentro, en caso contrario, está fuera.

Un simple método que se puede implementar para descartar fácilmente algunos puntos consiste en encontrar el mínimo rectángulo que lo contiene, es decir, encontrar sus coordenadas máximas y mínimas $x_{máx}$, $y_{máx}$, $x_{mín}$ y $y_{mín}$, respectivamente. Entonces, mediante 4 comparaciones menos costosas, se pueden descartar de inmediato ciertos puntos. En caso de que el punto se encuentre dentro de este rectángulo, ahí se aplica el algoritmo ray crossings. Por otra parte, al determinar la coordenada $x_{máx}$ se puede establecer el segmento que va a actuar como el rayo r, el cual fue establecido como $(q_x + x_{máx}, q_y)$. Con esto se asegura que todas las aristas puedan ser intersectadas por r.

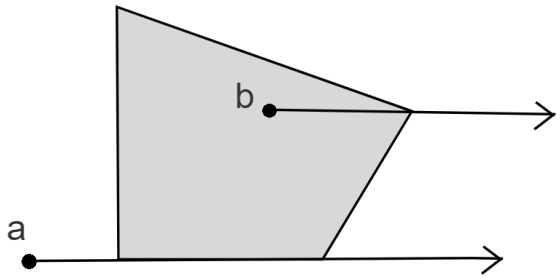


Figura 4.5: Casos en los que la cuenta de intersecciones haría fallar al algoritmo. a tiene 3 intersecciones (por lo que estaría dentro) y b tiene 2 (por lo que estaría fuera).

En la Figura 4.4 se puede ver un ejemplo de este método, donde se tiene que el rayo que parte de q_1 tiene 5 intersecciones, por lo que está dentro del polígono y el rayo de q_2 tiene 2 intersecciones por lo que está fuera del polígono. Algunos casos especiales que hay que tener en cuenta se ven en el caso del punto q_3, donde hay intersecciones con vértices o con aristas horizontales. En la Figura 4.5 se puede ver cuando estos tipos de casos fallan.

Para solucionar estos casos, O'Rourke [14] propone contar como intersección solo las aristas que tengan un extremo completamente sobre la recta, y el otro puede estar en la recta o por debajo. Con esto se solucionan los problemas de los puntos de la Figura 4.5 (a intersecta dos veces y b una vez), pero siguen habiendo problemas con los puntos que están justo sobre el borde.

Para solucionar el problema anterior, O'Rourke propone otra caracterización para contar aristas, sin embargo, la implementación del algoritmo de intersección de segmentos también indica si el punto en cuestión está sobre el segmento o no, con lo cual se puede resolver el problema de los puntos que caen justo en el borde del polígono sin tener que recurrir a esto.

Ahora, dado dos segmentos ab y cd, el algoritmo *intersección(a,b,c,d)* utiliza el resultado entregado en la sección 4.2.4 para determinar si los puntos c y d están "a distintos lados" respecto a la recta que contiene a y b, es decir, si tienen distintas orientaciones. Esto mismo se verifica para el segmento cd, determinando si a y b tienen distintas orientaciones. En el código implementado, la orientación puede tener tres valores:

- 1: Si la expresión 4.4 es mayor a 0 (sentido antihorario.)
- -1: Si la expresión 4.4 es menor a 0 (sentido horario).
- 0: Si la expresión 4.4 es igual a 0 (colineales).

El único caso en que el criterio de que tengan orientaciones distintas falla es cuando todos los puntos son colineales, ya que en este caso las rectas si pueden intersectarse. Para resolver este caso borde, se verifica que los cuatro puntos no sean colineales y se aplica el criterio anterior. Si los cuatro puntos son colineales, se verifica si a o b está en cd, o si c o b está en ab.

Algoritmo 5 Intersección

Input: Segmentos ab y cd
Output: B1, B2: Par indicando si ab y cd se intersectan (B1) y si a está en cd (B2).

```
1: if orientación(a, b, c)=orientación(a, b, d) then
2:     if orientación(a, b, c) = 0 then
3:         B1 ← a en cd || b en cd || c en ab || d en ab
4:         B2 ← a en cd
5:     else
6:         B1 ← False
7:         B2 ← False
8:     end if
9:     return B1, B2
10: end if
11: B1 ← orientación(a, b, c)=orientación(a, b, d)
12: B2 ← orientación(a, c, d) = 0
13: return B1, B2
```

Es importante notar que con todos los cálculos hechos también se puede determinar si el punto a está en cd, lo cual nos ayuda a resolver el caso problemático del algoritmo ray crossing mencionado con anterioridad. En efecto, si los cuatro puntos no son colineales y se intersectan, el punto a está dentro de cd si a, c y d son colineales. En caso que los cuatro puntos sean colineales, se verifica directamente si a esta en cd. Con esto, la implementación del algoritmo *intersección* retorna un par, cuyo primer componente indica si los segmentos se intersectan, y su segunda componente indica si el punto a está dentro de cd.

En el Algoritmo 5 se muestra el funcionamiento completo de *intersección*, con todos los detalles mencionados con anterioridad.

Con todo esto, finalmente se tiene la implementación completa del algoritmo *ray crossings*, la cual se puede ver en el Algoritmo 7.

4.3.2. Longest edge propagation path

Como se mencionó en la sección 2.2.3, el Lepp de un triángulo t corresponde a la secuencia ordenada de triángulos donde cada triángulo comparte una arista que es la más larga del anterior. Si dos triángulos t_{n-1} y t_n comparten la arista más larga para los dos, la secuencia termina. t_{n-1} y t_n se denominan terminal-triangles. Esto da de manera directa un algoritmo para obtener Lepp(t), lo cual se muestra en el Algoritmo 6. Notar que para efectos prácticos, lo que se determina es una secuencia de aristas más largas en vez de la secuencia de triángulos, ya que a partir de estas los triángulos quedan totalmente definidos.

Algoritmo 6 Lepp(t)

Input: Triangulación τ, triángulo t
Output: L: Secuencia de aristas (a_1,a_1,...,a_{n-1})

$a_i \leftarrow$ arista más larga de t
$L \leftarrow (a_i)$
$t_i \leftarrow$ triángulo que comparte a_i
$a_{i+1} \leftarrow$ arista más larga de t_i
while a_i no es de borde y $a_{i+1} \neq a_i$ **do**
    Añadir a_{i+1} a L
    $a_i \leftarrow a_{i+1}$
    $a_{i+1} \leftarrow$ arista más larga de triángulo que comparte a_{i+1}
end while
return L

Algoritmo 7 Ray crossings[2]

Input: Punto q y polígono P
Output: Si q está dentro de P.

$R \leftarrow$ rectángulo más pequeño que contiene a P
if q no está dentro de R **then**
    return q no está en P
else
    $q' \leftarrow (q_x + P_x^{máx}, q_y)$
    $n \leftarrow 0$
    for arista ab en P **do**
        if $Intersección(q, q', a, b).second$ **then**
            return q en P
        else if $Intersección(q, q', a, b).first$ **then**
            $n \leftarrow n + 1$
        end if
    end for
    if n es impar **then**
        return q en P
    else
        return q no está en P
    end if
end if

4.3.3. Edge flip

En la sección 2.2.2 se habló de la operación básica de edge flip que, dado un cuadrilátero convexo de una triangulación, intercambia su diagonal por la otra. Como en este trabajo se utiliza una estructura DCEL para representar una triangulación, la operación de edge flip implica solo modificar una serie de punteros de las semiaristas del cuadrilátero, es decir, no se crean semiaristas ni caras, solo se modifican las que ya estaban. Si bien esto puede parecer

[2]La notación *first* y *second* denota al primer y segundo elemento de un par, respectivamente.

sencillo, modificar todos estos campos es un procedimiento bastante engorroso y propenso a errores, ya que es muy fácil generar inconsistencias en la estructura que no son tan fáciles de detectar.

Como se mencionó en la sección 4.1.1, cada cara corresponde a un puntero a alguna semiarista que pertenece a ella, por lo que al hacer la operación edge flip se puede dar el caso de que una semiarista que representa una cara cambie de cara, por lo que habría que actualizar ese valor en la estructura. Esta es la razón por la cual se decidió almacenar cada una de las caras en un árbol en vez de una lista, ya que el tiempo de buscarla y eliminarla de la lista era lineal y aumentaba considerablemente el tiempo de ejecución.

Cálculo del centroide de un cuadrilátero

Si bien el cálculo del centroide [5] pudo haberse considerado en la sección 4.2 de cálculo de propiedades geométricas, se muestra acá ya que se necesitaban algunos de los resultados mostrados con anterioridad, en especial la intersección de segmentos.

Como se mostró en la sección 2.3, la inserción de puntos se puede realizar en el punto medio de la terminal-edge o en el centroide del cuadrilátero formado por los terminal-triangles. Para el cálculo del centroide, se ocupa la propiedad mostrada en [5], la cual afirma que se puede calcular el centroide de un área más compleja a partir de los centroides de las más pequeñas y sencillas que la componen. De esta forma se llega a que el centroide de un cuadrilátero $abcd$ corresponde a la intersección de los segmentos pq y rs, con p, q, r y s siendo los centroides de los triángulos abc, cda, dab y bcd, respectivamente, dados por la expresión 4.2. Como ya se tiene implementada la función para determinar si dos segmentos se intersectan, simplemente hay que resolver la ecuación de la recta para estos cuatro puntos. Aquí hay que precisar que estos segmentos siempre deberían tener un punto de intersección, no obstante, se implementó de esta forma para darle más generalidad a la función.

4.3.4. Inserción de puntos

La inserción de puntos realizada en la estructura DCEL conlleva un proceso engorroso similar al edge flip mostrado en la sección 4.3.3, en el que hay que modificar campos de varias estructuras y crear algunas nuevas. Las inserciones consisten en añadir un nuevo vértice en un cuadrilátero $abcd$ (formado por terminal-triangles) de diagonal ac. El vértice se puede insertar en el punto medio de ac o en el centroide de $abcd$. Esto añade dos nuevas aristas y dos nuevas caras. En caso que haya que añadir un vértice en una arista de borde este se inserta en el punto medio, lo cual crea solo una arista y una cara.

En este caso, simplemente se modifican las semiaristas involucradas y se añaden caras nuevas, por lo que no es necesario hacer búsquedas en la estructura que almacena a estas últimas.

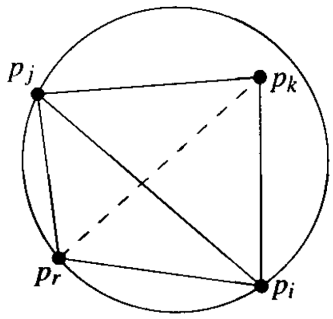


Figura 4.6: Edge flip de la arista ilegal p_jp_i al insertar el vértice p_r.

Inserción de Delaunay

Una vez insertado un punto en un cuadrilátero, se debe corroborar que las aristas del cuadrilátero sigan siendo legales, de lo contrario se aplica la operación edge flip de manera sucesiva hasta que la triangulación sea nuevamente de Delaunay. Este algoritmo fue mostrado a grandes rasgos en la sección 2.3. Sea τ una triangulación de Delaunay, y sea τ' la triangulación creada al insertar el vértice q en un cuadrilátero $p_ip_jp_hp_g$. Se debe corroborar la condición de Delaunay dada por la expresión 2.1 para los cuatro cuadriláteros que comparten el vértice q. Se denomina *legalizar arista* a la operación de corroborar la condición de Delaunay para p_jp_i, y en caso de no cumplirla, hacer edge flip y corroborar la condición para las nuevas aristas p_jp_k y p_kp_i, como se muestra en la Figura 4.6. Esto da naturalmente una solución recursiva [6], sin embargo, el procedimiento implementado sigue una solución iterativa. Esto se muestra en el Algoritmo 8.

Algoritmo 8 Inserción de Delaunay

Input: Triangulación τ, Cola Q, punto q insertado en cuadrilátero $p_ip_jp_hp_g$
Output: Triangulación τ de Delaunay

```
S ← (p_i, p_j, p_h, p_g)
for e en S do
    CS ← (e)
    while CS no es vacío do
        Extraer una arista ab de CS
        if ab es ilegal then
            edgeflip(ab)
            Actualizar los triángulos modificados en Q
            c ← extremo de la nueva arista, que no es q
            Insertar ac y cb en CS
        end if
    end while
end for
```

4.3.5. Refinamiento de Lepp

Con todos los resultados anteriores, se puede enunciar detalladamente el algoritmo de refinamiento para un triángulo, que corresponde a la línea 9 del Algoritmo 3. Como se mostró con anterioridad, al refinar un triángulo t, primero hay que encontrar su Lepp e insertar un punto en el cuadrilátero formado por sus triángulos terminales, y repetir esto hasta que $\text{Lepp}(t) = (t, t_1)$, que será el momento en que finalmente se "destruya" el triángulo t. Sin embargo, se puede dar el caso en que al insertar un punto en los triángulos terminales de $\text{Lepp}(t)$ (en que ninguno de los dos sea t), la propagación de los edge flips destruya t, lo cual es considerado como que este fue refinado. Hay que notar que esto no significa que t ahora cumpla con el criterio impuesto, ya que puede seguir en la cola y ser refinado nuevamente. Esto se muestra en el Algoritmo 9.

Algoritmo 9 Refinar triángulo mediante una inserción de Delaunay

Input: Triangulación de delaunay τ, triángulo t
Output: Triangulación τ con t refinado

```
while t no ha sido refinado do
    t1, t2 ← triángulos terminales de Lepp(t)
    Efectuar una inserción de Delaunay en el cuadrilátero formado por t1, t2
    if t fue modificado en el paso anterior then
        t fue refinado
    end if
end while
```

4.3.6. Descomposición de polígonos cóncavos en convexos

Este problema también puede ser entendido como un refinamiento de polígonos mediante inserción de aristas (si se relaja la condición de que hayan solo convexos), donde el caso extremo sería llegar a una triangulación. En este caso, se busca tener el menor número de polígonos finales y se utiliza la estrategia expuesta en el algoritmo de Mehlhorn, mencionada en la sección 2.4. Esta decisión se debe a que ya se cuenta con una triangulación para cada uno de los polígonos formados por el algoritmo Polylla, por lo que esta estrategia es la más sencilla de implementar.

Primero, se dan algunas notaciones para lo que sigue:

- Un ángulo mayor a 180° se denomina reflex.
- A un vértice de un polígono que subtiende un ángulo reflex también lo llamaremos reflex.
- Para un vértice q de un polígono, denotaremos como q_{-1} y q_1 al vértice anterior y siguiente, respectivamente. También, se denomina orientación de q a la orientación de estos tres puntos.
- Diremos que una arista "repara" un ángulo α reflex (o vértice reflex) si al añadirla, α se divide en dos ángulos no reflex.

El primer paso es determinar si un polígono es cóncavo o convexo, lo cual se realiza recorriendo cada vértice (a través de la estructura DCEL) determinando si es reflex. Para esto se determina si el polígono está ordenado en sentido horario o antihorario, es decir, si el orden de los vértices forman un circuito horario o antihorario. Esto se utiliza para determinar si un ángulo es reflex sin calcular el ángulo [23], y luego también se utiliza para poder saber en que sentido recorrer las aristas de la triángulación.

En este paso se utiliza el método expuesto por Wijeweera [23], que busca el punto q cuya coordenada y sea más baja, y en caso de haber más de un punto que cumpla esta condición, el que tenga la coordenada x más baja. q necesariamente no es reflex, y la orientación de q corresponde a la orientación de todo el polígono. Con esto, se tiene la propiedad de que si la orientación de un vértice es distinta a la del polígono, ese vértice es reflex. Esto se muestra en el Algoritmo 10.

Algoritmo 10 Determinar ángulos mayores a 180°

Input: Polígono P
Output: Vértices reflex L

```
r ← algún vértice de P
for vértice v en P do
    if v_y < r_y then
        r = v
    else if v_y = r_y y v_x < r_x then
        r = v
    end if
end for
L ← ∅
o ← orientación de r
for vértice v en P do
    if orientación de v ≠ o then
        Insertar v en L
    end if
end for
return L
```

Ahora, para q vértice reflex, se deben añadir aristas hasta que todos los ángulos subtendidos por este no sean reflex. En este punto, se utiliza un método distinto al dado por el algoritmo de Mehlhorn: Básicamente, se barre el ángulo de q viajando a través de sus aristas incidentes en la triangulación, determinando cuales son las mejores aristas a insertar siguiendo los siguientes dos criterios, en orden:

1. Si una arista qa repara el ángulo de q y repara el ángulo de a, esta se inserta.
2. Las mejores aristas a insertar son las que lo dividen en partes iguales, o lo más cercano a esto.

Para el segundo criterio, se utiliza la propiedad de que la máxima cantidad de aristas que puede necesitar un ángulo para ser reparado es 2 [11].

Con esto, el algoritmo funciona barriendo el ángulo, almacenando en todo momento la mejor arista a insertar, o las mejores dos aristas a insertar. En caso que se encuentre que se cumpla el primer criterio, el algoritmo termina. Esto se muestra en el algoritmo 11.

Es importante notar que la aplicación del primer criterio puede aumentar el tiempo de ejecución, ya que en el peor caso se haría el doble del trabajo para reparar un polígono. Esto se debe a que si, estando en el vértice a, se quiere verificar que la arista ab repara el vértice b, se debe recorrer todas las aristas incidentes a este último. Como cada arista tiene dos extremos, el tiempo puede llegar a ser el doble. En el próximo capítulo se muestran algunos resultados que muestran la diferencia entre ocupar este criterio o no.

Por otro lado, en algunas aplicaciones puede que un criterio más importante que el número mínimo de polígonos sea el valor de los ángulos, por lo que quizá se querría utilizar de todas formas el segundo criterio por sobre el primero.

Algoritmo 11 Reparar ángulo

Input: Polígono P, triangulación τ de P, vértice reflex q
Output: Aristas a insertar para descomponer P en partes convexas.

$\alpha \leftarrow$ ángulo de q
$a_1 \leftarrow qx$
$a_2 \leftarrow a_1$
$a_3 \leftarrow a_1$
for arista qb **do**
  if qb repara q y repara b **then**
    return qb
  end if
  if $\angle(q_{-1}q, qb)$ más cercano a $\frac{\alpha}{2}$ que $\angle(q_{-1}q, a_1)$ **then**
    $a_1 \leftarrow qb$
  end if
  if $\angle(q_{-1}q, qb)$ más cercano a $\frac{\alpha}{3}$ que $\angle(q_{-1}q, a_2)$ **then**
    $a_2 \leftarrow qb$
  end if
  if $\angle(q_{-1}q, qb)$ más cercano a $\frac{2\alpha}{3}$ que $\angle(q_{-1}q, a_3)$ **then**
    $a_3 \leftarrow qb$
  end if
  if a_1 repara a q **then**
    return a_1
  else
    return (a_2, a_3)
  end if
end for

4.4. Modo de uso

Para ejecutar el programa hay dos formas, dependiendo del formato de archivo que se quiera pasar como argumento. El primero corresponde al formato *.off*[3], el cual es un único archivo conteniendo la información de vértices y polígonos. La segunda forma es utilizando tres archivos con formato *.node*[4], *.ele*[5] y *.neigh*[6]. Así se tiene que el programa puede ser ejecutado de las siguientes dos formas:

```
$ polylla-refinement <file.off> [options]
$ polylla-refinement <file.node> <file.ele> <file.neigh> [options]
```

Las opciones del programa se muestran en la Tabla 4.1.

[3] https://segeval.cs.princeton.edu/public/off_format.html
[4] https://www.cs.cmu.edu/~quake/triangle.node.html
[5] https://www.cs.cmu.edu/~quake/triangle.ele.html
[6] https://www.cs.cmu.edu/~quake/triangle.neigh.html

Tabla 4.1: Opciones para correr el programa.

Argumento	**Descripción**
`-output`	Especifica el directorio en el cual serán generados los archivos
`-factor`	Especifica un factor al cual debe llegar el área promedio al refinar. Debe ser un número mayor a 0 y menor a 1
`-max_area`	Especifica el área máxima que puede tener un triángulo.
`-circle`	Define un círculo como región de refinamiento. Se utiliza como `-circle <x> <y> <r>`, con (x,y) siendo el centro y r el radio.
`-rectangle`	Define un rectángulo como región de refinamiento. Se utiliza como `-rectángulo <x><y><rx><ry>`, con (x,y) siendo el centro, rx la mitad del ancho y ry la mitad de la altura.
`-polygon`	Define un polígono arbitrario como región de refinamiento. Se ocupa como `-polygon <archivo.off>` (único formato soportado).
`-filename`	Especifica un nombre de archivo de salida.
`-print_iter`	Con esta opción, se escriben todas las iteraciones intermedias del refinamiento. Modo de uso: `-print_iter <directorio>`
`-print_region`	Escribe un archivo con la región definida para el refinamiento.
`-dont_write`	No escribe ningún archivo de salida.
`-polylla`	Genera la malla Polylla original y la refinada.
`-centroid`	Especifica que el punto de inserción sea en el centroide. Por defecto es en el punto medio.
`-check_delaunay`	Indica que se corrobore si la triangulación entregada es de Delaunay
`-not_delaunay`	Indica que el refinamiento no sea de Delaunay.

Capítulo 5

Experimentación y validación

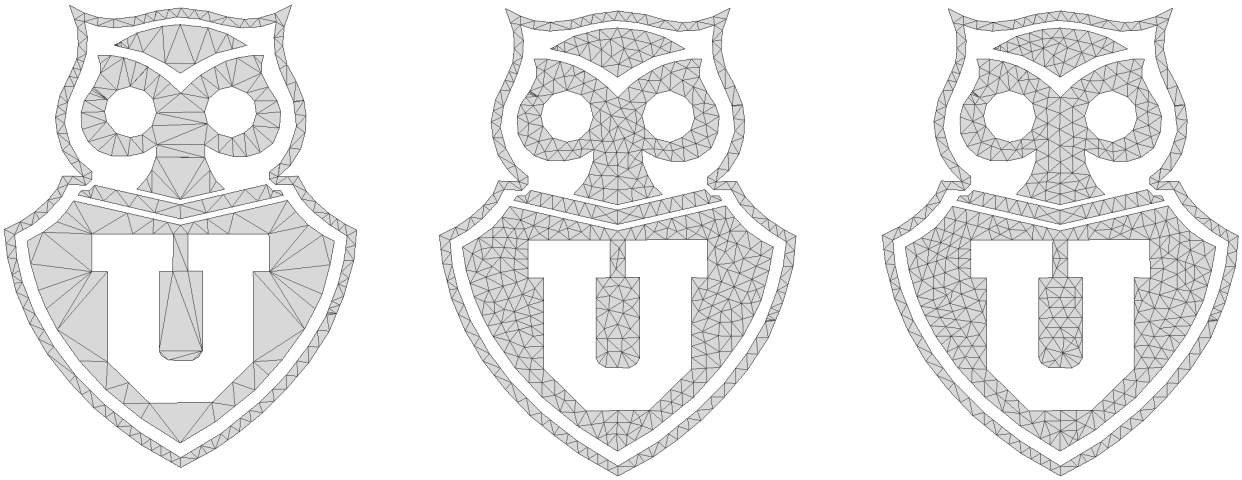
En este capítulo se mostrarán algunos resultados del trabajo realizado, probando las funcionalidades implementadas más importantes. En esta línea, se realizan algunos experimentos que permiten justificar algunas de las decisiones de implementación mostradas en el capítulo anterior. Por otro lado, se compara el refinamiento implementado con la solución entregada por *Triangle* (bajo los mismos parámetros), que es uno de las herramientas más populares y eficientes disponibles. Finalmente, se analizan los resultados del refinamiento de mallas Polylla en partes convexas.

5.1. Resultados refinamiento de Lepp

En la Figura 5.1 se pueden ver dos triangulaciones refinadas bajo los mismos parámetros, pero una utilizando inserción en el punto medio y la otra inserción en el centroide. Las métricas de la triangulación final, como el ángulo mínimo y máximo o el número de polígonos finales son casi análogos, viendo en general un ángulo mínimo un poco mayor en el caso de la inserción en el centroide.

En la Figura 5.2 se puede ver el resultado del refinamiento local, dada una región definida por un polígono arbitrario, y en la Figura 5.3 se puede ver el refinamiento para un círculo. Es importante notar como las zonas circundantes a las regiones definidas también se ven afectadas, ya sea por la propagación del refinamiento de Lepp, o por la propagación de los edge flip.

En la sección 4.1.2 se mencionó el uso de la estructura `unordered_map` de la librería estándar de C++. Está estructura funciona mediante hashing, que se sabe proporciona un tiempo promedio constante para búsquedas e inserciones, sin embargo, dependiendo del número de elementos que se inserten (y de la función de hashing utilizada), este tiempo puede ser lineal en el peor caso. Esto se puede volver problemático en casos en que se estén refinando mallas de gran tamaño, por lo que se exploró la opción de utilizar la estructura `map` de la librería estándar, que garantiza tiempos logarítmicos en el número de elementos en el peor caso (esta estructura está implementada a partir de un árbol balanceado). En la Figura



(a) (b) (c)

Figura 5.1: Comparación entre los dos métodos de inserción para el refinamiento. En (a) se muestra la triangulación inicial, en (b) la refinada por inserción en en el centroide y en (c) la refinada por inserción en el punto medio. Los dos refinamientos fueron realizados con el mismo parámetro de área.

5.4 se muestran resultados de un refinamiento donde se alcanzaron más de 20 millones de triángulos, realizado bajo los mismos parámetros, y se puede observar que el desempeño de `unordered_map` es considerablemente mejor en casi todo el dominio (a excepción de los puntos con un área máxima más grande). Con esto se logra validar el uso de esta estructura para la clase que implementa la cola de prioridad.

5.2. Refinamiento de polígonos cóncavos

Para los experimentos de esta sección se utilizan tres tipos de distribuciones de puntos, para los cuales se construyen triangulaciones que son finalmente pasadas como input al algoritmo Polylla. La primera es una distribución de puntos al azar. La segunda es una triangulación generada con Triangle a partir de un cuadrado, el cual se refina imponiendo un área máxima para alcanzar el número de vértices pedido. A esta distribución la denotaremos como semiuniforme. La tercera corresponde a la distribución de Poisson, que es una técnica iterativa de generación de puntos en la cual se generan puntos al azar que no contengan otros puntos dentro de un radio dado.

En la sección 4.3.6 se mencionó que el algoritmo de refinamiento de polígonos cóncavos en convexos tiene como primer criterio el generar la menor cantidad de polígonos, para lo cual el tiempo de ejecución podría llegar a ser el doble. En la Tabla 5.1 se muestran algunos resultados respecto al número de polígonos añadidos con y sin este criterio, para distintas distribuciones de puntos y distinto número de puntos.

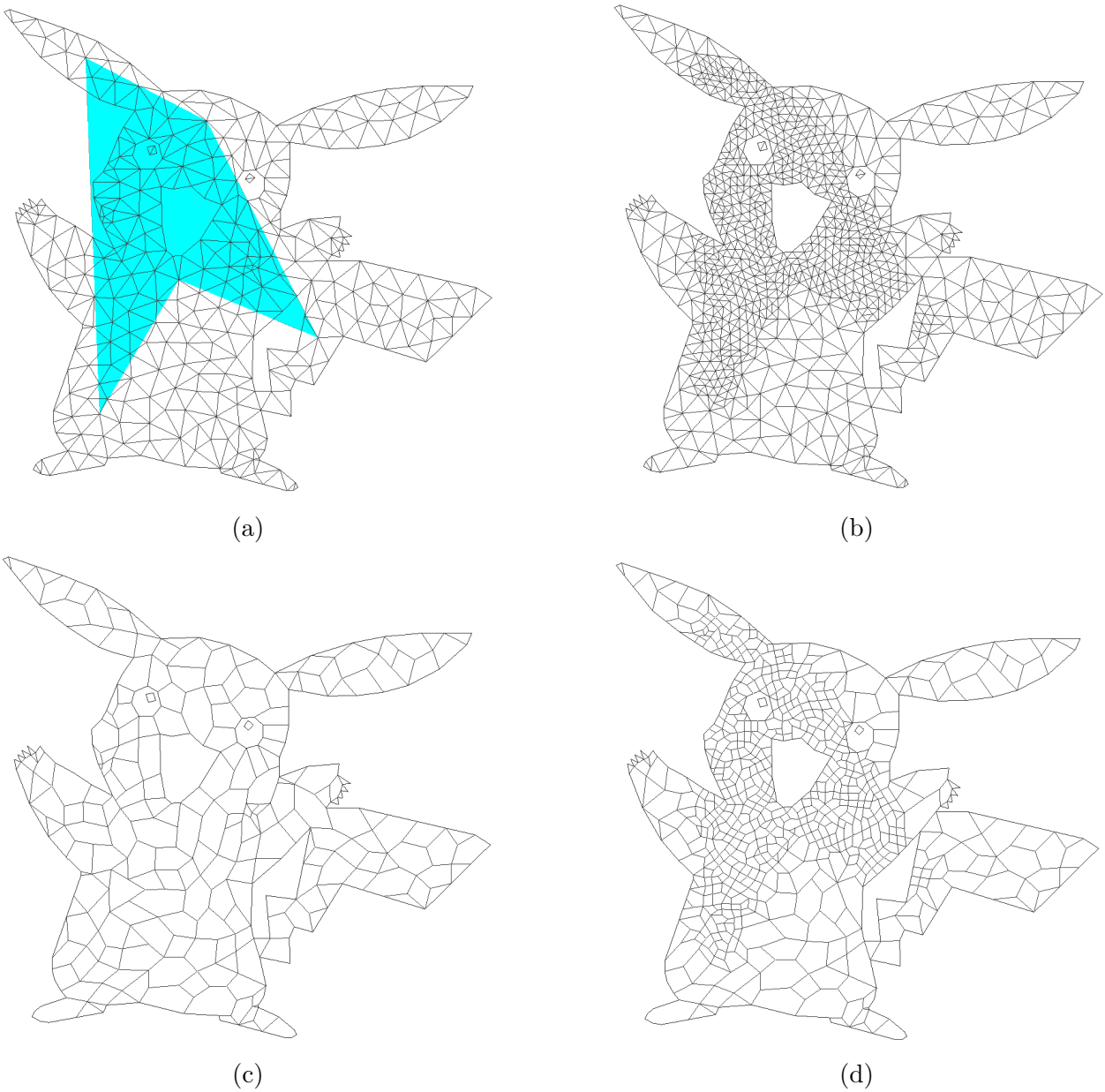


Figura 5.2: Refinamiento local. (a) Triangulación original con el polígono que define la región a ser refinada. (b) Triangulación refinada. (c) Malla Polylla a partir de la triangulación original. (d) Malla Polylla a partir de la triangulación refinada.

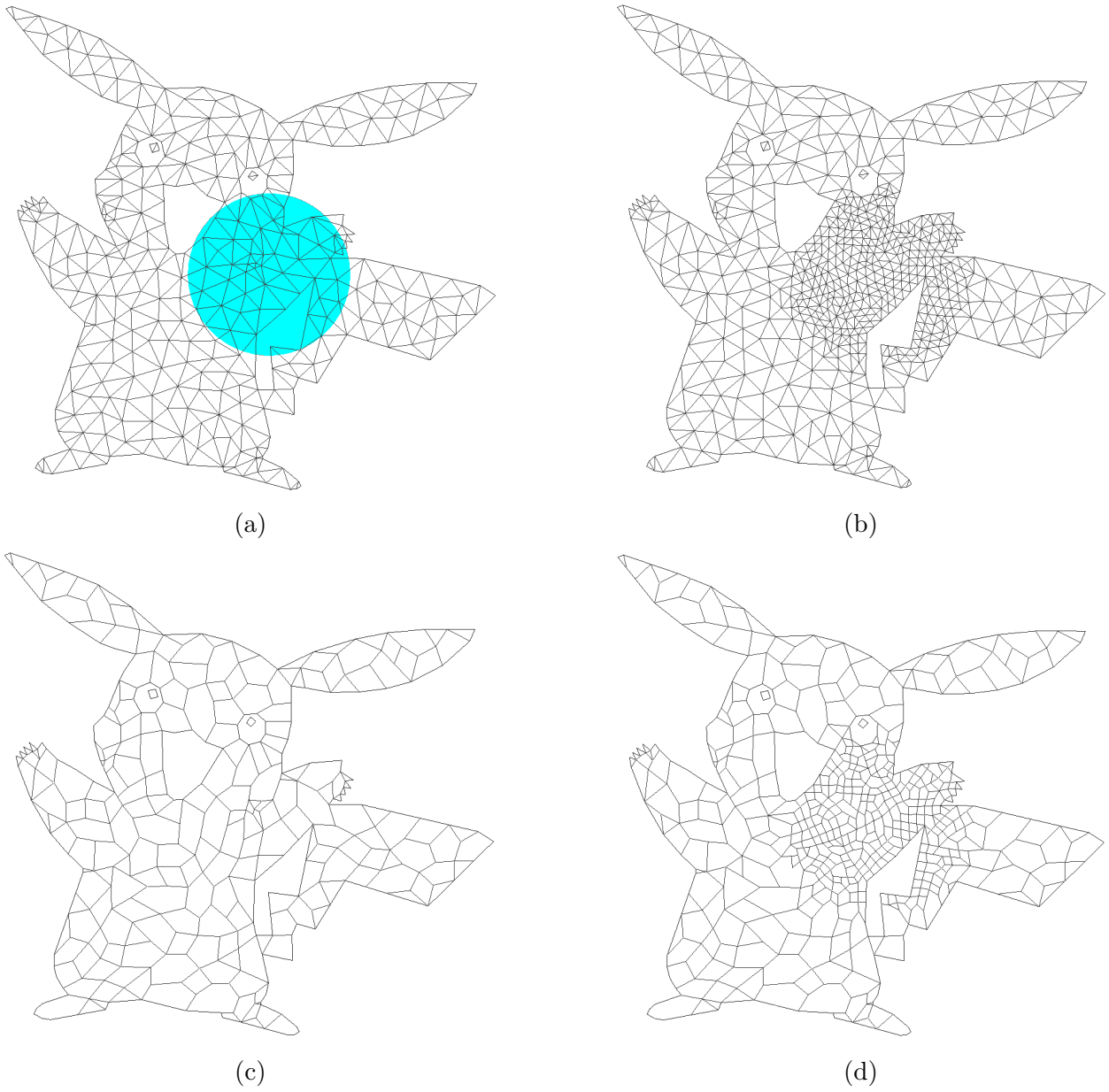


Figura 5.3: Refinamiento local. (a) Triangulación original con el círculo que define la región a ser refinada. (b) Triangulación refinada. (c) Malla Polylla a partir de la triangulación original. (d) Malla Polylla a partir de la triangulación refinada.

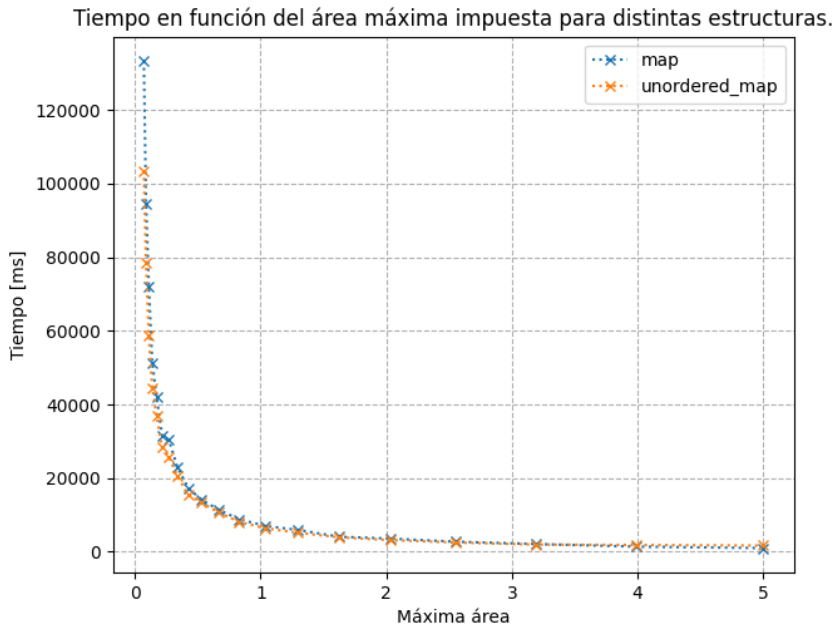


Figura 5.4: Comparación entre el tiempo que toman las implementaciones para las estructuras `map` y `unordered_map`, en función del área máxima impuesta.

Tabla 5.1: Resultados para el refinamiento de polígonos cóncavos con distintas distribuciones de puntos. La columna "sin opt" corresponde a los resultados sin aplicar el primer criterio mostrado en la sección 4.3.6. Aquí la palabra "Insertados" hace referencia al número de polígonos que se añaden para transformar los polígonos cóncavos en convexos.

Distribución	**Vértices**	**Polígonos**	**Cóncavos**	**Insertados**	**Insertados sin opt**
Azar	500	173	104	174	196
	1500	472	283	523	605
	2500	783	473	930	1048
	5000	1633	972	1761	2000
	10000	3172	1953	3621	4132
	30000	9336	5751	10699	12244
Semiuniforme	500	298	65	77	85
	1500	907	222	266	290
	2500	1465	372	434	479
Poisson	500	306	77	93	100
	1500	852	235	254	284
	2500	1481	370	425	458

Se puede observar en la tabla que al aplicar el primer criterio de refinamiento, el número de polígonos finales mejora de manera significativa, siendo en promedio un 10 % menor que la solución que optimiza el ángulo de las aristas insertadas (el segundo criterio mencionado en la sección 4.3.6). Por otra parte, respecto al tiempo de ejecución, no se encontraron diferencias significativas entre las dos soluciones comparadas, llegando a ser en muchos casos incluso más rápida la solución con el primer criterio.

Ahora, en cuanto al número de polígonos finales, se puede observar que para una malla con puntos que siguen una distribución al azar, en promedio se añaden aproximadamente 1,8 polígonos convexos por cada polígono cóncavo. En cambio, para una distribución semiuniforme y de Poisson se generan en promedio 1,2 y 1,1 polígonos, respectivamente. Estas razones parecen mantenerse relativamente constantes respecto al número de vértices de la malla.

Otro parámetro que parece mantenerse constante en relación al número de vértices es el porcentaje de polígonos cóncavos que presentan las mallas Polylla, siendo en promedio un 61 %, 24 % y 26 % para la distribución al azar, semiuniforme y de Poisson, respectivamente.

En las Figuras 5.5, 5.7 y 5.9 se puede ver el resultado del refinamiento para mallas con 500 vértices, al aplicar el primer criterio y al aplicar solo el segundo. En las Figuras 5.6, 5.8 y 5.10 se puede ver cada una de estas mallas por separado, ya que en las figuras que grafican las tres mallas juntas hay aristas de un color que en realidad forman parte de las dos soluciones.

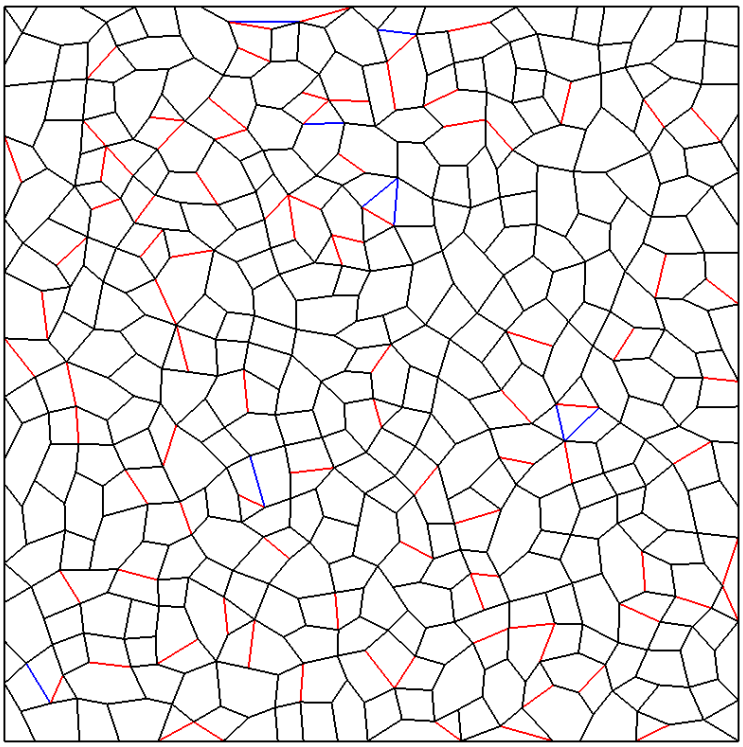
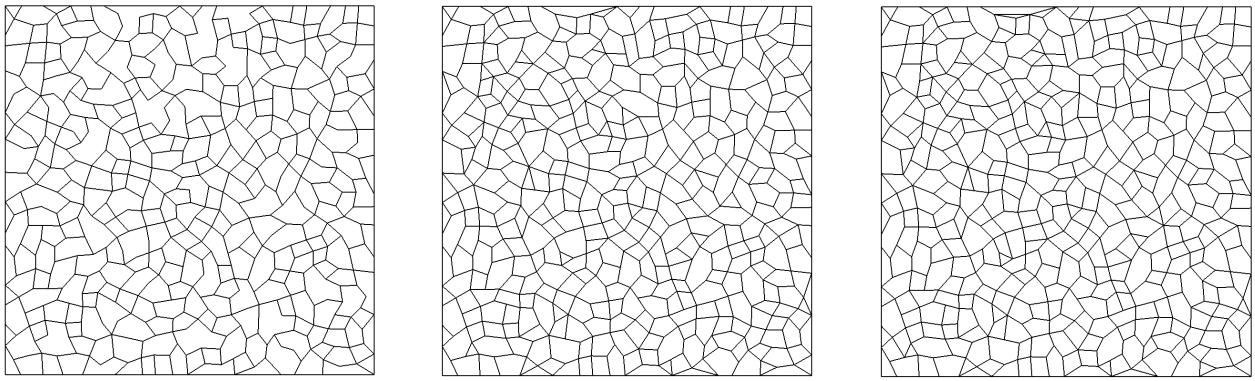


Figura 5.5: Malla Polylla a partir de puntos con una distribución de Poisson. Las aristas añadidas para convertir los polígonos cóncavos en convexos se muestran en rojo. En azul las aristas adicionales que se añaden si no se aplica el primer criterio de refinamiento.



(a) (b) (c)

Figura 5.6: Refinamiento de mallas Polylla. En (a) se muestra la malla Polylla original. En (b) se muestra la malla luego de descomponer los polígonos cóncavos en convexos, aplicando el primer criterio. En (c) se muestra el resultado sin aplicar el primer criterio.

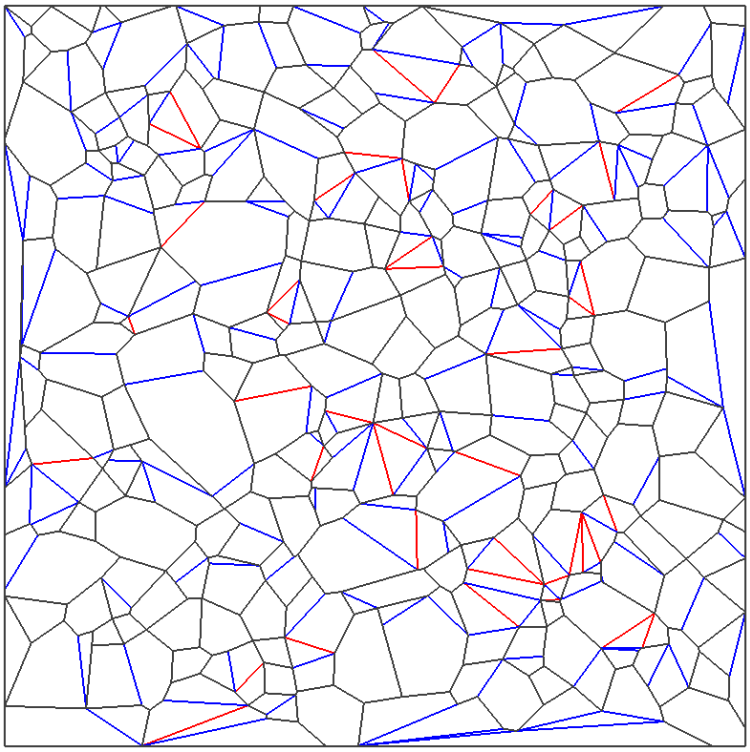
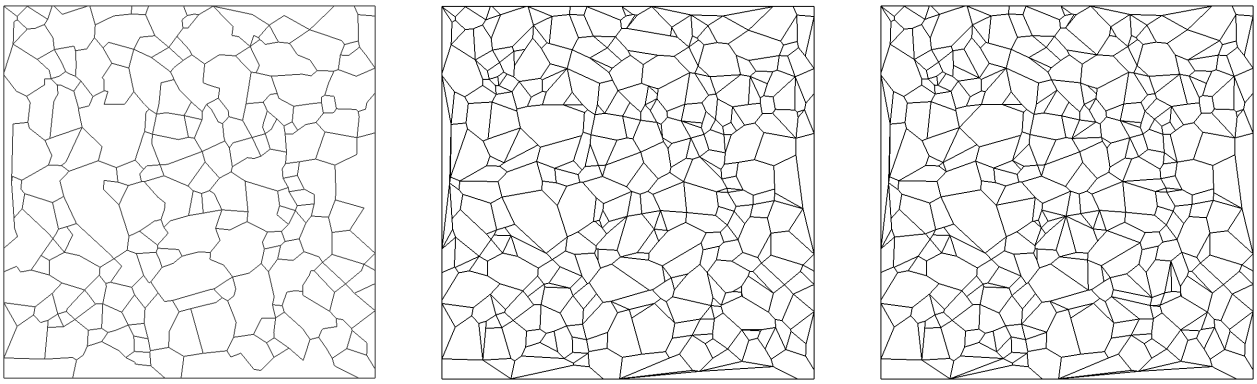


Figura 5.7: Malla Polylla a partir de puntos con una distribución al azar. Las aristas añadidas para convertir los polígonos cóncavos en convexos se muestran en rojo. En azul las aristas adicionales que se añaden si no se aplica el primer criterio de refinamiento.



(a) (b) (c)

Figura 5.8: Refinamiento de mallas Polylla con una distribución de puntos al azar. En (a) se muestra la malla Polylla original. En (b) se muestra la malla luego de descomponer los polígonos cóncavos en convexos, aplicando el primer criterio. En (c) se muestra el resultado sin aplicar el primer criterio.

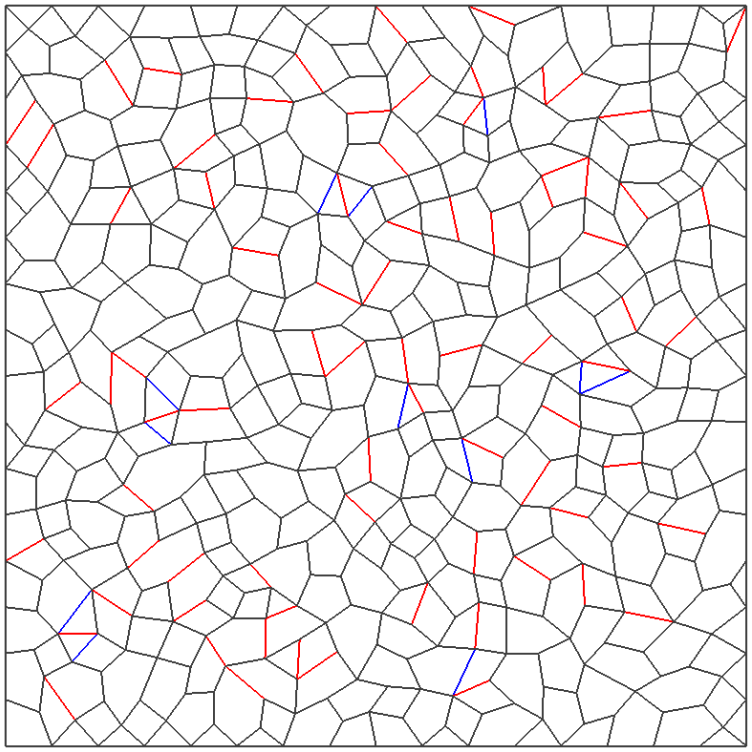
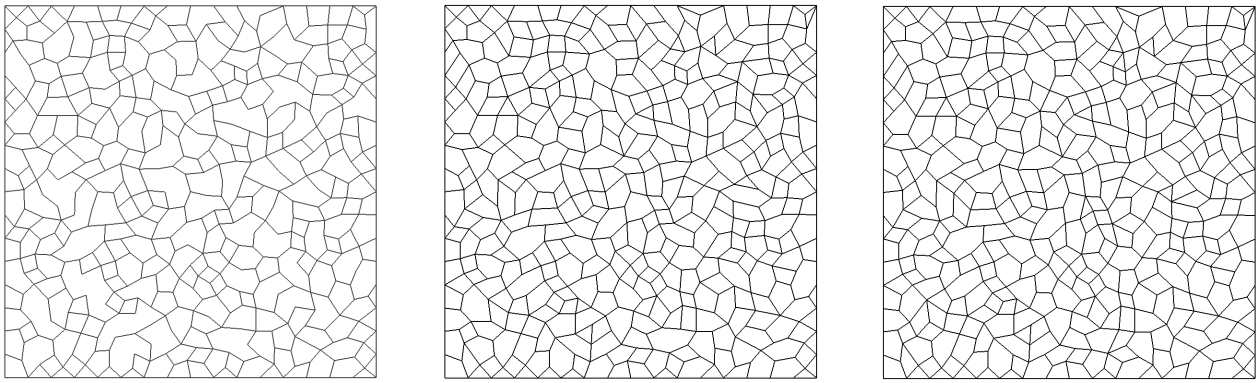


Figura 5.9: Malla Polylla a partir de puntos con una distribución semiuniforme. Las aristas añadidas para convertir los polígonos cóncavos en convexos se muestran en rojo. En azul las aristas adicionales que se añaden si no se aplica el primer criterio de refinamiento.



(a) (b) (c)

Figura 5.10: Refinamiento de mallas Polylla con una distribución de puntos semiuniforme. En (a) se muestra la malla Polylla original. En (b) se muestra la malla luego de descomponer los polígonos cóncavos en convexos, aplicando el primer criterio. En (c) se muestra el resultado sin aplicar el primer criterio.

5.3. Comparación con Triangle

Ahora se compara el desempeño en tiempo y tamaño de malla para el algoritmo implementado con respecto al obtenido con *Triangle*. Estos experimentos fueron realizados utilizando la misma triangulación inicial y el mismo parámetro de refinamiento, el cual corresponde al área máxima de un triángulo.

En la Figura 5.11 se puede observar que el software *Triangle* es más rápido que el programa implementado, en todo el dominio, siendo la diferencia particularmente alta para mallas de gran tamaño.

Por otro lado, los resultados respecto al tamaño de la malla (Figura 5.12) favorecen al algoritmo implementado, llegando a tener un 6 % menos de triángulos que la triangulación de Triangle. Esto comprueba algunos resultados que muestran la optimalidad en tamaño de las triangulaciones refinadas con Lepp. En la Figura 5.13 se puede comparar el refinamiento entregado por *Triangle* con el entregado por nuestra solución, dada la misma triangulación inicial (Figura 5.1 (a)) y los mismos parámetros.

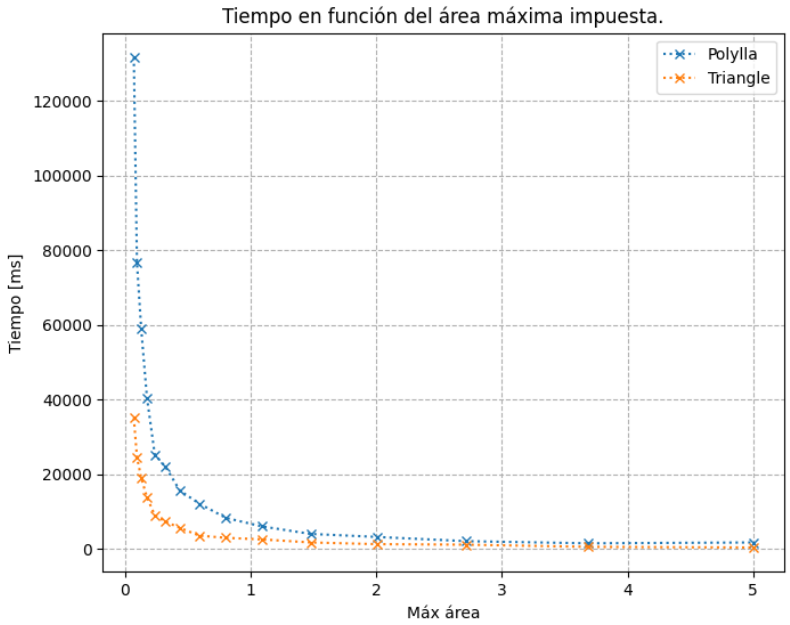


Figura 5.11: Comparación del tiempo de ejecución con el software *Triangle*, para la misma área máxima impuesta como criterio de refinamiento y la misma triangulación inicial.

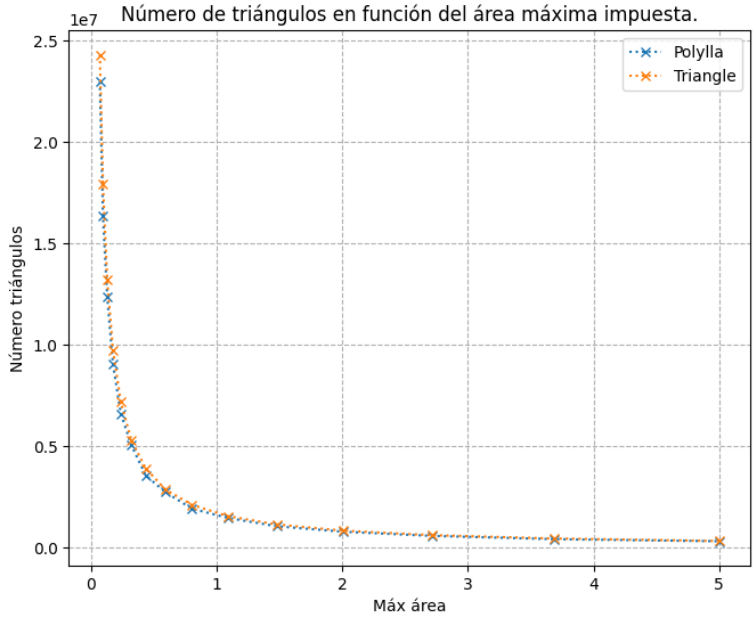
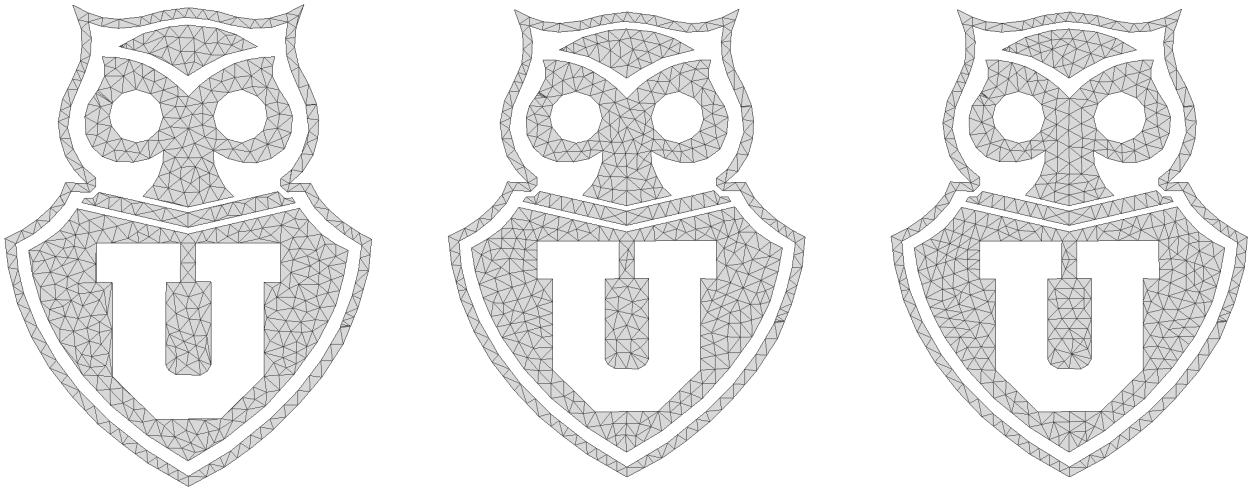


Figura 5.12: Comparación del número de triángulos finales con el software Triangle, para la misma área máxima impuesta como criterio de refinamiento y la misma triangulación inicial.



(a) (b) (c)

Figura 5.13: Comparación entre el refinamiento de *Triangle* con el algoritmo implementado. En (a) se muestra la triangulación entregada por *Triangle*, en (b) la refinada por inserción en en el centroide (misma de la Figura 5.1 (b)) y en (c) la refinada por inserción en el punto medio (mismo de 5.1 (c)).

Capítulo 6

Conclusión

Se desarrolló un algoritmo de refinamiento mediante inserción de puntos basado en el concepto de Lepp, el cual funciona de manera global o local sobre regiones arbitrarias definidas por el usuario. Este algoritmo cuenta con dos métodos de inserción de puntos, los cuales permiten trabajar con triangulaciones de Delaunay, aunque no se limitan solo a estas. Por otra parte, se implementó un método que, mediante la inserción de aristas, transforma una malla Polylla de polígonos arbitrarios en una compuesta exclusivamente por polígonos convexos.

Con esto se concluye que se cumplieron los objetivos planteados al comienzo de la memoria, pudiendo validar su correcto funcionamiento y evaluando su desempeño respecto a herramientas que abordan el mismo problema. Si bien el desempeño en tiempo respecto a *Triangle* es considerablemente peor para mallas muy grandes (millones de triángulos), nuestra solución es mejor respecto al número final de triángulos, además de contar con un método de refinamiento local, el cual *Triangle* carece.

Uno de los objetivos específicos que no se cumplió a cabalidad, corresponde a la implementación de más métricas que controlen el refinamiento, sin embargo, lo implementado provee una arquitectura modular con ciertas interfaces que deberían proporcionar un alto grado de escalabilidad.

Hay varias áreas en que el trabajo realizado en esta memoria puede ser extendido o mejorado como: 1) Trabajar en el objetivo específico mencionado con anterioridad, es decir, añadir más métricas, 2) discutir otras posibles soluciones a los problemas abordados o 3) implementar nuevas funcionalidades.

En cuanto al primer punto, se pueden agregar más métricas y criterios para el refinamiento por inserción de puntos, en el caso de las triangulaciones, y para el refinamiento por inserción de aristas, en el caso de las mallas Polylla.

En cuanto al segundo punto, se pueden explorar y realizar mejoras respecto a las estructuras de datos utilizadas, pudiendo adecuarse mejor al problema y disminuyendo el tiempo de ejecución del programa. Por ejemplo, uno de los componentes más importantes del algoritmo de refinamiento es la cola utilizada, por lo que buscar otras alternativas que mejoren el desempeño de esta tendrá un impacto inmediato en todo el algoritmo.

Junto con las estructuras de datos, se pueden estudiar otros algoritmos para resolver los problemas planteados, por ejemplo en el refinamiento de polígonos cóncavos, donde se han mostrado buenos resultados con algoritmos que toman un tiempo parecido.

Bibliografía

[1] Alok Aggarwal, Leonidas J. Guibas, James Saxe, and Peter W. Shor. A linear-time algorithm for computing the voronoi diagram of a convex polygon. *Discrete Comput. Geom.*, 4(6):591–604, dec 1989.

[2] R. Alonso, J. Ojeda, N. Hitschfeld, C. Hervías, and L.E. Campusano. Delaunay based algorithm for finding polygonal voids in planar point sets. *Astronomy and Computing*, 22:48–62, 2018.

[3] Grosse Eric Rafferty Conor S. Baker, B.S. Nonobtuse triangulation of polygons. *Discrete computational geometry*, 3(1-2):147–168, 1988.

[4] Carlos Bedregal and Maria-Cecilia Rivara. Longest-edge algorithms for size-optimal refinement of triangulations. *Computer-Aided Design*, 46:246–251, 2014. 2013 SIAM Conference on Geometric and Physical Modeling.

[5] Ferdinand P. Beer. *Vector Mechanics for Engineers: Statics and Dynamics*. McGraw-Hill Science/Engineering/Math, 2003.

[6] Mark de Berg, Otfried Cheong, Marc van Kreveld, and Mark Overmars. *Computational Geometry: Algorithms and Applications*. Springer-Verlag TELOS, Santa Clara, CA, USA, 3rd ed. edition, 2008.

[7] Bernard Chazelle and David P. Dobkin. Optimal convex decompositions. In Godfried T. TOUSSAINT, editor, *Computational Geometry*, volume 2 of *Machine Intelligence and Pattern Recognition*, pages 63–133. North-Holland, 1985.

[8] L. Paul Chew. Guaranteed-quality mesh generation for curved surfaces. In *Proceedings of the Ninth Annual Symposium on Computational Geometry*, SCG '93, page 274–280, New York, NY, USA, 1993. Association for Computing Machinery.

[9] J Fernández, L Cánovas, and B Pelegrın. Algorithms for the decomposition of a polygon into convex polygons. *European Journal of Operational Research*, 121(2):330–342, 2000.

[10] Steven Fortune. Voronoi diagrams and delaunay triangulations. In *Handbook of Discrete and Computational Geometry, 2nd Ed.*, 2004.

[11] Stefan Hertel and Kurt Mehlhorn. Fast triangulation of the plane with respect to simple polygons. *Information and Control*, 64(1):52–76, 1985. International Conference on Foundations of Computation Theory.

[12] Rainald Löhner. Progress in grid generation via the advancing front technique. *Engineering with Computers*, 12:186–210, 2005.

[13] D.E. Muller and F.P. Preparata. Finding the intersection of two convex polyhedra. *Theoretical Computer Science*, 7(2):217–236, 1978.

[14] Joseph O'Rourke. *Computational Geometry in C*. Cambridge University Press, 2 edition, 1998.

[15] A. Ortiz-Bernardin, R. Silva-Valenzuela, S. Salinas-Fernández, N. Hitschfeld-Kahler, S. Luza, and B. Rebolledo. A node-based uniform strain virtual element method for compressible and nearly incompressible elasticity. *International Journal for Numerical Methods in Engineering*, 124(8):1818–1855, 2023.

[16] Maria-Cecilia Rivara and Javier Diaz. Terminal triangles centroid algorithms for quality delaunay triangulation. *Computer-Aided Design*, 125:102870, 2020.

[17] María-Cecilia Rivara. New longest-edge algorithms for the refinement and/or improvement of unstructured triangulations. *International Journal for Numerical Methods in Engineering*, 40(18):3313–3324, 1997.

[18] J. Ruppert. A delaunay refinement algorithm for quality 2-dimensional mesh generation. *Journal of Algorithms*, 18(3):548–585, 1995.

[19] Sergio Salinas-Fernández. Polylla: Polygonal meshing algorithm based on terminal-edge regions. `https://github.com/ssalinasfe/Polylla-Mesh-DCEL`, 2022.

[20] Sergio Salinas-Fernández, José Fuentes-Sepúlveda, and Nancy Hitschfeld-Kahler. Generation of polygonal meshes in compact space. In *International Meshing Roundtable Workshop (IMR)*, Amsterdam, Netherlands, March 6–9 2023.

[21] Sergio Salinas-Fernández, Nancy Hitschfeld-Kahler, Alejandro Ortiz-Bernardin, and Hang Si. POLYLLA: polygonal meshing algorithm based on terminal-edge regions. *Engineering with Computers*, pages 1–23, 2022.

[22] Jonathan Richard Shewchuk. Triangle: Engineering a 2D Quality Mesh Generator and Delaunay Triangulator. In Ming C. Lin and Dinesh Manocha, editors, *Applied Computational Geometry: Towards Geometric Engineering*, volume 1148 of *Lecture Notes in Computer Science*, pages 203–222. Springer-Verlag, May 1996. From the First ACM Workshop on Applied Computational Geometry.

[23] K. Wijeweera and Saluka Kodituwakku. Convex partitioning of a polygon into smaller number of pieces with lowest memory consumption. *Ceylon Journal of Science*, 46:55, 03 2017.