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

SOFTWARE PARA ESTUDIO EMPÍRICO Y REPORTES SOBRE EL COMPORTAMIENTO DE ALGORITMOS DE REFINAMIENTO PARA TRIANGULACIONES

MEMORIA PARA OPTAR AL TÍTULO DE INGENIERO CIVIL EN COMPUTACIÓN

MAXIMILIAN IGNACIO WILLEMBRINCK SANTANDER

PROFESOR GUÍA:
MARÍA CECILIA RIVARA ZÚÑIGA

MIEMBROS DE LA COMISIÓN:
JOSÉ A. PINO URTUBIA
ROMAIN ROBBES

SANTIAGO DE CHILE
2016

Resumen

El método de elementos finitos, que permite resolver numéricamente problemas modelados por ecuaciones diferenciales parciales para el análisis de complejos problemas físicos sobre geometrías complejas, es una de las más importantes aplicaciones del ámbito científico. En estos problemas las geometrías son modeladas utilizando mallas geométricas, cuya calidad determina la precisión de los cálculos y de los resultados obtenidos por el método.

En una malla geométrica definida por una triangulación, se entiende por calidad que el ángulo mínimo de cada triángulo esté acotado inferiormente por un valor acorde a la necesidad del problema. Dado un conjunto de puntos para construir una triangulación, las triangulaciones de tipo «Delaunay» se caracterizan por maximizar el ángulo mínimo interior de sus triángulos, y en consecuencia producen la triangulación de mejor calidad basada en tales puntos. Sin embargo, en caso de requerirse una mejor calidad, ésta puede mejorarse insertando nuevos puntos estratégicamente.

En este contexto, los algoritmos tipo «Delaunay» de refinamiento de mallas juegan un rol de importancia ya que proveen métodos de mejoramiento de la calidad de una triangulación por medio de la inserción de puntos adicionales. Cada inserción se realiza mediante un proceso que mantiene la propiedad de «Delaunay» de la malla, e incrementan su calidad.

Los distintos de algoritmos refinamiento tipo «Delaunay» presentan diferencias en cuanto a su desempeño y a la malla resultante. Ciertos algoritmos incorporan una etapa de «preprocesamiento», en la cual se prepara la malla para prevenir complicaciones durante el desempeño del resto del algoritmo. Adicionalmente, algunos se benefician de procesar los triángulos siguiendo un orden en particular. Omitir el preprocesamiento u ordenamiento en estos casos, altera el resultado obtenido. Es posible estudiar el comportamiento de los algoritmos al establecer comparaciones entre sus condiciones de procesamiento respecto a los resultados obtenidos.

Existen programas computacionales que permiten la aplicación de un algoritmo de refinamiento sobre una malla y que como resultado presentan la malla mejorada y estadísticas respecto al desempeño del algoritmo durante la ejecución. Estas aplicaciones requieren ser ejecutadas una vez por cada refinamiento. Para el caso en que el número de algoritmos y/o mallas es considerable, lo anterior impone una dificultad. En particular cuando se busca establecer comparaciones relativas al comportamiento de los algoritmos, esta situación obliga al investigador a realizar un trabajo posterior de manejo de datos y visualización para los resultados obtenidos. Así, el tiempo de trabajo y esfuerzos adicionales, en casos de grandes volúmenes de pruebas y datos, pueden llegar a ser imprácticos fácilmente.

En esta memoria se pretende elaborar una herramienta inteligente para comparar algoritmos de refinamiento tipo «Delaunay», realizar «tuning» de algunos algoritmos, proponer nuevos, etcétera, considerando diversas condiciones de procesamiento. Adicionalmente, se espera que la herramienta permita definir esquemas de ejecuciones de procesamiento, a través de los cuales sea posible analizar y obtener grandes cantidades de resultados de manera simple y práctica. Finalmente, la aplicación deberá generar «Reportes de Visualización de Resultados» en archivos «html», que incluyen cada detalle de la configuración y de los resultados, además de visualizaciones de gráficos para las tablas comparativas producidas.

En las siguientes páginas se aborda en detalle el proceso de diseño e implementación de esta herramienta, cuyo desarrollo se ejecuta a partir de la elaboración de una biblioteca basada en la aplicación «Compare2DMesh», que fue producida en trabajos anteriores, la cual permite la ejecución individual y obtención de resultados de un refinamiento tipo «Delaunay» sobre una malla. Esta nueva biblioteca, llamada «C2M», supera a la aplicación original en múltiples aspectos. Además, se construye una nueva aplicación «BatchRefinement» que utiliza «C2M», y que provee una interfaz gráfica a través de la cual se proveen funcionalidades que satisfacen todos los requerimientos relacionado al trabajo con volúmenes de experimentos y generación de reportes con los análisis de los resultados. Estos reportes son creados en unos pocos segundos, de manera automática y flexible, y presentan información que habría tomado horas procesar sin esta herramienta. Las pruebas realizadas confirman la versatilidad de la aplicación, e indican una mejoría respecto a «Compare2DMesh» en muchos aspectos.

A lo largo de este documento, se discuten temas relevantes y consideraciones relacionadas con la implementación actual y, en determinados casos, con los trabajos anteriores.

Agradecimientos

A cada una de las personas dentro de mis círculos de interacción más cercanos, cuya influencia forma parte de mí, y finalmente de este trabajo.

A Carolina Hernández, por su disposición, aportes y apoyo incondicional.

A mi profesora guía, cuya orientación, atención y asistencia, fue vital e inmensamente apreciada.

Finalmente, a mi familia, a quienes les debo todo.

Tabla de contenido

I Introducción 1

1. Motivación 1

1.1. Contexto - La importancia de tener mallas geométricas de buena calidad. . . 1

1.2. Modificaciones de algoritmos y sus efectos 2

1.3. Procesamiento por lotes 3

1.4. Propuesta del trabajo de memoria 3

1.5. Estructura de la memoria 4

2. Conceptos generales 5

2.1. Malla geométrica y triangulación. 5

2.2. Criterio de calidad. 5

2.3. Refinamiento. 6

2.4. Distinción de métodos de refinamiento. 6

2.4.1. Algoritmos «Lepp» y «No Lepp» 7

3. Objetivo del trabajo 8

3.1. Objetivo general 8

3.2. Objetivos específicos 8

II Desarrollo 10

4. Terminología y algoritmos 10

4.1. Definiciones 10

4.1.1. Punto 11

4.1.2. Vértice 11

4.1.3. Arista 11

4.1.4. Aristas restringidas 12

4.1.5. Aristas «encroached» 12

4.1.6. Triángulo 13

4.1.7. Círcuncírculo 13

4.1.8. Malla Geométrica 13
4.1.9. Grafo (o grafo no dirigido) 14
4.1.10. Grafo planar de línea recta 14
4.1.11. Triangulación 15
4.1.12. Triangulación válida 17
4.1.13. Triangulación Delaunay 17
4.1.14. Procedimiento de intercambio de diagonales 18
4.1.15. Triangulación Delaunay restringida 19
4.1.16. Triangulación Delaunay conforme 19
4.1.17. Inserción Delaunay 20
4.1.18. Calidad de una triangulación 20
4.1.19. Steiner point 20
4.1.20. Lepp 21
4.1.21. Arista terminal 22
4.1.22. Colas 22
4.2. Algoritmos de refinamiento. 24
4.2.1. Algoritmos de refinamientos no Delaunay 25
4.2.2. Algoritmos de refinamiento Delaunay 25
4.2.2.1. Algoritmos Lepp 25
4.2.2.2. Algoritmos «No Lepp» 25
4.3. Algoritmos mixtos y personalizados 26
4.4. Métricas para comparación de resultados 30
4.5. Concepto de «mejor algoritmo» 31

5. Implementación 33
5.1. Descripción de alto nivel 33
5.1.1. Trabajo realizado y descripción de la aplicación 33
5.1.2. Entradas y salidas 34
5.1.2.1. Descripción de parámetros iniciales de refinamiento 35
5.1.2.2. Archivos y elementos de salida 37
5.2. Descripción temas técnicos 41
5.2.1. Marco de desarrollo 41

5.2.1.1. Computador de trabajo 41
5.2.1.2. Entorno de programación. 41
5.2.2. Estado inicial y evolución de funcionalidades generales de trabajos anteriores 42
5.2.3. Formatos de archivos de entrada 44
5.2.3.1. Mejoras sobre inicialización desde archivos de entradas. . . . 44
5.2.3.2. Formatos de archivos de triangulaciones de entrada soportados 47
5.2.3.3. Otros formatos y extensibilidad. 49
5.2.4. Utilidades y validaciones 49
5.2.5. Problemas heredados y soluciones 52
5.2.6. Separación de proyectos 63
5.2.7. Paralelismo 68
5.2.8. Precisión y resolución de mediciones de intervalos de tiempo 69

III Resultados **70**

6. Pruebas y resultados **70**
6.1. Objetivos de las pruebas 70
6.2. Primeros resultados 70
6.2.1. Pruebas simples 71
6.2.2. Grandes triangulaciones 72
6.2.3. Casos conocidos 74
6.3. Comprobando la hipótesis 75
6.4. Afinamiento de visualización y comparaciones «uno a uno» 79

7. Conclusión **82**
7.1. Aplicación implementada y resultados 82
7.2. Trabajo futuro 82

8. Bibliografía **83**

Referencias **83**

9. Anexo **85**

A. Código fuente **85**

B. Descripción de formatos de archivos binarios **85**

B.1. Binmesh 85

B.2. Bitf 86

C. Muestra de «Reporte de número de vértices variante» generado. **89**

D. Muestra de «Reporte de ángulo exigido variante» generado. **104**

D.1. Página de resumen de resultados sobre reporte de ángulo variante para triangulación de 1000 vértices 104

D.2. Página de detalles de resultados para una sola serie, sobre reporte de ángulo variante para triangulación de 1000 vértices 112

Índice de figuras

1.1. Ejemplos de mallas geométricas. Una aproximación de un terreno a la izquierda, y un delfín modelado con pocos polígonos a la derecha. 1

1.2. Mejor aproximación debido a polígonos más pequeños y regulares del modelado de una esfera tridimensional. 2

2.1. Triangulación Delaunay obtenida a partir de un conjunto de puntos. 5

2.2. Muestra de proceso de triangulación y refinamiento. 7

3.1. La funcionalidad principal de «BatchRefinement»: «Generar reportes de visualización de resultados html a partir de conjuntos de experimentos de algoritmos de refinamiento sobre triangulaciones Delaunay». 10

4.1. Procedimiento de corrección de aristas **«encroached»** por medio de bisecciones sucesivas. 13

4.2. Circuncírculo y circuncentro «O» de un triángulo ABC. 14

4.3. Mallas geométricas con caras triangulares y cuadrilaterales bidimensionales. . 14

4.4. Utilización de mallas geométricas en análisis de elementos finitos. 15

4.5. Ejemplos de grafos no dirigidos. 15

4.6. Ejemplo de grafo planar de línea recta. 16

4.7. Triangulación Delaunay restringida obtenida a partir de un grafo planar. . . 16

4.8. Triangulación Delaunay con todos los circuncírculos y sus respectivos circuncentros. 18

4.9. Procedimiento de intercambio de diagonales. 18

4.10. Diferencia entre triangulación Delaunay «restringida» respecto a «conforme» de un PSLG. 19

4.11. Malla geométrica de mala calidad (a la izquierda) al lado de la versión mejorada (a la derecha). 21

4.12. Posibles «Steiner Points»: «O» (a la izquierda) o «P»(a la derecha), los cuales son candidatos a inserción dependiendo del algoritmo y condiciones locales. . 21

4.13. Lepp de un triángulo t_0, compuesto por $\{t_0, t_1, t_2, t_3, t_4\}$. 22

4.14. «Arista terminal» (rojo) y «triángulos vecinos terminales» (amarillos) de un Lepp$\{t_0, t_1, t_2, t_3, t_4\}$. 22

4.15. Jerarquía propuesta de algoritmos de refinamiento. 24

4.16. Diagrama básico y simbología. 28

4.17. Detalles de diagrama. 29

4.18. Detalle de iteración de refinamiento. 30

5.1. Interfaz de generación de reportes html, «ReportCreator». 33

5.2. Diagrama simple y de alto nivel del flujo de datos de Compare2DMesh. . . . 34
5.3. Diagrama simple y de alto nivel del flujo de datos de C2Mcmd. 34
5.4. Diagrama simple y de alto nivel del flujo de datos de BatchRefinement para producción «xml». 35
5.5. Diagrama simple y de alto nivel del flujo de datos de BatchRefinement para producción de Reportes «html» de Visualización de Resultados. 35
5.6. Tabla de índice de vértices finales vs iniciales para una triangulación de 1000 puntos. 37
5.7. Muestra de archivo de resultados «xml» producido por la aplicación. Los «...» fueron incluidos sólo para efectos de resumir la ilustración, e indican la presencia de otros datos análogos a los ya descritos. 38
5.8. Miniatura de muestra de distribución en 2 columnas de los reportes «html». 40
5.9. Ejemplo de tabla obtenida de resultados y reportes tipo **«ángulo variante»**. 40
5.10. Ejemplo de tabla obtenida de resultados y reportes tipo **«número de vértices variante»**. 41
5.11. Interfaz gráfica de «MeshSuite». 43
5.12. Evolución del «código principal». 45
5.13. Muestras de distintos tamaños de las imágenes que produce la aplicación al rasterizar. 51
5.14. Efecto de dimensiones de la imagen resultante del proceso de «rasterización». 52
5.15. Efecto del ajuste de opacidad en rasterizaciones de un mismo tamaño. 52
5.16. Diagrama de jerarquía antiguo de clases involucradas en el proceso de elección del nuevo «Steiner Point» a insertar. 54
5.17. Diagrama de la nueva jerarquía simplificada de clases involucradas en el proceso de elección del nuevo «Steiner Point» a insertar. Finalmente, se redujo a la utilización de sólo una. 54
5.18. Simplificación de jerarquía de clases relacionadas a «InsideInsertion». 55
5.19. Simplificación de jerarquía de clases de colas. 55
5.20. Tabla comparativa de tiempos de ejecución de Compare2DMesh vs. C2M. . . 57
5.21. Código de método con fugas de memoria. 58
5.22. Clase «MemoryDebug» para detectar fugas de memoria, implementada utilizando «C Run-Time Libraries (CRT) debug heap functions». 61
5.23. «Test» implementado para detección de fugas de memoria que abarca todos los pasos del refinamiento. 61
5.24. Diagrama de características de las aplicaciones desarrolladas en torno a «C2M» de la solución final. 64

6.1. Triangulación de 7 vértices utilizada para pruebas. 71
6.2. Resultados de distintas configuraciones de algoritmos para un mismo ángulo de exigencia sobre triangulación de 7 vértices. 72
6.3. Imágenes poco claras para triangulaciones grandes. 73
6.4. Resultados de tiempo total para una triangulación de 1.000.000 vértices. . . 73
6.5. Resultados de número final de vértices para una triangulación de 1.000.000 vértices. 74
6.6. Triangulación de 17 vértices formada principalmente por triángulos malos, utilizada en distintas publicaciones. 74
6.7. Resultados de distintas configuraciones de algoritmos para un mismo ángulo de exigencia sobre triangulación conocida. 75
6.8. Triangulación inicial con 1.000 vértices. 76
6.9. Gráfico de resultado: Tiempo total de procesamiento para series múltiples, con y sin ordenamiento de triángulos. 77
6.10. Gráfico de resultado: Número de vértices resultantes para series múltiples, con y sin ordenamiento de triángulos. 78
6.11. Elección de series para producción de «Reporte de Visualización de Resultados». 79
6.12. Resultados comparativos de algoritmo de Üngor, con y sin prioridad de procesamiento. 80
6.13. Resultados comparativos de algoritmo Lepp-Centroide, con y sin prioridad de procesamiento. 80
6.14. Resultados comparativos de algoritmo de Üngor con ordenamiento vs Lepp-Centroide sin ordenamiento. 81
6.15. Resultados comparativos sobre otras triangulaciones (100 vértices a la izquierda, 1000 vértices a la derecha), de algoritmo de Üngor con ordenamiento vs Lepp-Centroide sin ordenamiento. 81

Parte I

Introducción

1. Motivación

1.1. Contexto - La importancia de tener mallas geométricas de buena calidad.

Muchas aplicaciones del ámbito científico y artístico, hacen uso intensivo de mallas geométricas. Simulaciones, aplicaciones CAD, modelado de sólidos, gráficos computacionales, visualización científica, son ejemplos clásicos. La idea central es la representación de superficies, funciones, u objetos continuos contando con elementos discretos, o la resolución por aproximación de problemas de similares características. La mayor parte de éstas requieren, o se benefician, de que la estructura geométrica de la malla sea de buena calidad, dado que existe una relación inversa de la calidad de los elementos de la malla, y el error asociado a interpolaciones en ésta. En los casos de problemas de elementos finitos, los triángulos delgados afectan negativamente la convergencia y tiempos de ejecución de soluciones[1], incluso propician errores de cálculo, degradando la calidad de la solución numérica[2]. En otras palabras, la exactitud de las soluciones se ven favorecidas a medida que se tenga una mayor calidad de los elementos.

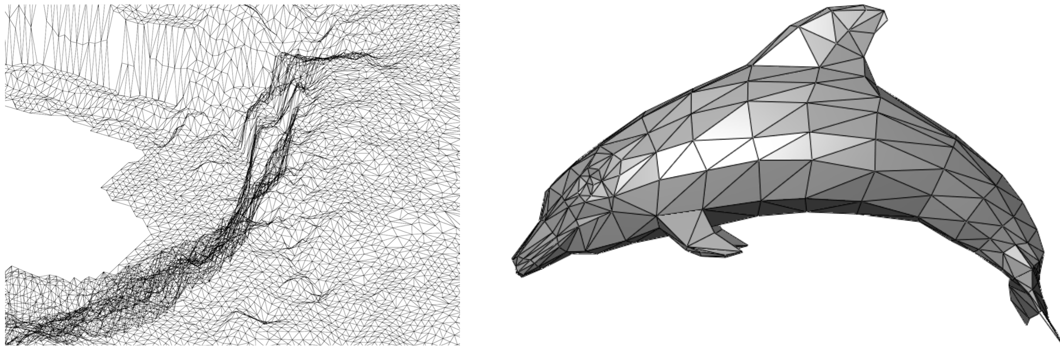


Figura 1.1: Ejemplos de mallas geométricas. Una aproximación de un terreno a la izquierda, y un delfín modelado con pocos polígonos a la derecha.

En el contexto de este estudio, una malla geométrica es una triangulación bidimensional de un conjunto de puntos. Esto incluye las triangulaciones Delaunay (definidas en 4.1.13), las cuales contienen muchas propiedades deseables y son las utilizadas en aplicaciones como las descritas.

Debido a que naturalmente no todas las triangulaciones presentan una buena calidad dado cierto criterio, existen algoritmos que permiten mejorarlas manteniendo las propiedades relevantes (los vértices), y así producir mejores resultados en la aplicación que hace uso de la

malla, conocidos como «algoritmos de refinamiento». Estos tienen características inherentes: ventajas y desventajas, y al buscar la mejor alternativa, es posible establecer ciertas medidas objetivas de en cuáles casos uno es conveniente respecto al resto.

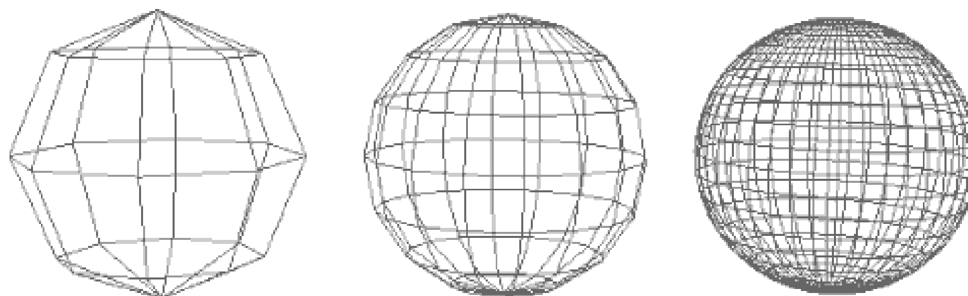


Figura 1.2: Mejor aproximación debido a polígonos más pequeños y regulares del modelado de una esfera tridimensional.

Los algoritmos de refinamiento existentes, que agregando puntos a la malla, garantizan cierta calidad en sus resultados y explicitan un nivel de complejidad de operación. Estas características nos permiten elegir convenientemente entre uno u otro. Si bien, para funcionar no requieren trabajos implícitos o no considerados dentro de los algoritmos, para lograr una ejecución impecable y obtener resultados óptimos requieren de ciertas preparaciones del conjunto de datos de entrada, y otras consideraciones adicionales. Son estos detalles los que en la práctica distancian una implementación final de la teoría original, generando una pequeña, pero aun así significativa, brecha de incertidumbre.

Ocurre que aunque un algoritmo lidere, por ejemplo, en complejidad de operaciones, pueda presentar a la vez un problema considerable de uso de memoria, tiempo en pre-procesos, costo de preparación de datos, etc. Uno de los objetivos de este trabajo es identificar estos ámbitos, y comparar algoritmos considerando cada una de estas implicancias. Aparte de los más populares del rubro, también se considerarán algoritmos mixtos experimentales.

Como concepto general, se considera «óptimo» a aquel algoritmo que produce un mejoramiento de la malla sobre un ángulo de calidad, y cuyo número de triángulos resultantes es el menor posible, sin embargo, estas consideraciones no tienen relación bajo ciertas necesidades prácticas, donde se requieren triangulaciones iniciales más grandes.

1.2. Modificaciones de algoritmos y sus efectos

Ciertas consideraciones al implementar un algoritmo de refinamiento afectan no sólo al resultado, sino que al costo de ejecución del mismo.

El ejemplo más evidente de esto se da en los algoritmos de Üngor y Ruppert, donde es necesario procesar los elementos en un orden en particular para lograr resultados óptimos. Al

dar importancia a estos detalles puede que una «simple» implementación de un algoritmo tradicional caiga en un problema de impracticabilidad o de conseguir algo menos que óptimo, donde se introducen «más triángulos que los necesarios», incluso hasta en dificultades de desarrollo debido al incremento en la complejidad. Existen otros algoritmos que no requieren consideraciones extras para generar resultados competitivos, y el presente documento a continuación expondrá las consecuencias de tener o no en cuenta el trabajo adicional asociado a estas consideraciones.

Los algoritmos en general no son intrínsecamente «flexibles», de manera de poder cambiarles parámetros que modifican la estructura del flujo en sí. No es tarea trivial obtener algo así. El primer paso fue definirlos de manera que esto sea posible, y esto se realizó basado en dos trabajos anteriores, tomando implementaciones de Álvaro Faúndez[3] y Francisca Gallardo[4], permitiendo obtener comparaciones de las métricas de interés. Se tiene ahora la capacidad de elegir características del flujo de procesamiento más allá de las usuales: si los elementos del algoritmo se procesan en orden o no, el ordenamiento en sí, si se ejecuta o no algún pre-proceso, y así, indistintamente del algoritmo base escogido. Adicionalmente es posible obtener resultados de acciones particulares: tiempo utilizado sólo en pre-procesos, triángulos generados por pre-proceso, largo del Lepp más largo, etc. Con estos micro-resultados se puede establecer relaciones más puntuales sobre el comportamiento de los algoritmos, y por lo tanto, verlos y evaluarlos de manera más detallada en contraste a sólo mirar el resultado final.

1.3. Procesamiento por lotes

Para establecer diferencias estadísticas entre ejecuciones de refinamiento es necesario contar con un buen volumen de datos. Para esto es de gran utilidad contar con un sistema automatizado de pruebas. El procesamiento por lotes, o «batch processing», permite ejecutar grandes cantidades de instancias de refinamiento, obteniendo resultados individuales, para posteriormente procesarlos y generar informes sobre las propiedades de interés. Sin la herramienta adecuada, obtener cantidades significativas de datos puede incurrir en grandes demoras, obstaculizando el estudio. Por lo tanto, sería de gran utilidad tener un sistema integrado de procesamiento por lotes de algoritmos de refinamiento, y generación automática de resultados.

1.4. Propuesta del trabajo de memoria

La temática de los trabajos de memoria de título antes mencionados coinciden en parte con el actual. Para establecer de manera más clara la identidad de este trabajo, se facilita un breve recuento de los aportes relevantes relacionados a los trabajos anteriores: El software Meshsuite, de A. Faúndez provee de una buena base de implementación de algoritmos y variaciones[3] . Compare2DMesh de F. Gallardo construye sobre eso, mejorando drásticamente los tiempos de ejecución, complementando el repertorio disponible, y produciendo

finalmente implementaciones automatizadas competitivas de los algoritmos[4].

La presente memoria, retoma el producto previo, profundizando en temas técnicos, y desglosando el impacto de cada parte de los algoritmos, para así observar experimentalmente el comportamiento de éstos y establecer la mejor forma de refinamiento, o un acercamiento al óptimo, y ofrecer una perspectiva no accesible a partir las aplicaciones precedentes, a partir de volúmenes de resultados experimentos obtenidos de manera automatizada y de la generación de reportes de visualización de éstos.

En este trabajo de memoria se pretende elaborar una herramienta inteligente para comparar algoritmos de refinamiento tipo «Delaunay», realizar «tuning» de algunos algoritmos, proponer nuevos, etcétera, considerando diversas condiciones de procesamiento y capacidades de configuración de estos. Adicionalmente, se espera que la herramienta permita definir esquemas de ejecuciones de procesamiento, a través de los cuales sea posible analizar y obtener grandes cantidades de resultados de manera simple y práctica. Finalmente, la aplicación deberá generar reportes visuales estructurados, que incluyen cada detalle de la configuración y de los resultados, además de visualizaciones de gráficos para las tablas comparativas producidas.

A lo largo de este documento, se discuten temas relevantes y consideraciones relacionadas con la implementación actual y, en determinados casos, con los trabajos anteriores.

1.5. Estructura de la memoria

La memoria consiste en 3 partes.

Introducción: Se explica el trasfondo de la memoria y del trabajo realizado, declarando motivos y entregando explicaciones no formales pero intuitivas de los tópicos.

Desarrollo: Contiene una descripción formal, más técnica y detallada de los conceptos utilizados y del trabajo realizado.

Conclusión: Entrega de resultados y discusión sobre estos.

2. Conceptos generales

Se ofrece a continuación explicaciones breves e introductorias a los principales temas de la memoria.

2.1. Malla geométrica y triangulación.

Es una colección de vértices, aristas y caras, que define la forma de un objeto complejo en base a polígonos. En este trabajo, se considerarán solamente mallas bidimensionales, compuestas únicamente por triángulos, los cuales se conectan formando una superficie orientable, y se denominará indistintamente como una triangulación.

Dado un conjunto de vértices, es posible construir una triangulación que los contenga. Las triangulaciones tipo Delaunay(definida en 4.1.13) son las principalmente utilizadas en los procesos de refinamientos.

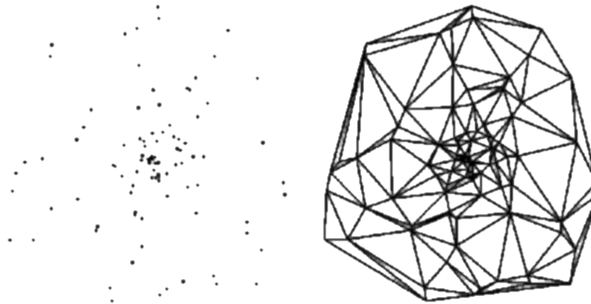


Figura 2.1: Triangulación Delaunay obtenida a partir de un conjunto de puntos.

2.2. Criterio de calidad.

Es una función que mide la calidad de las triangulaciones.

Existen varias medidas para definir la calidad de una malla o de un triángulo en 2D. Se ha demostrado matemáticamente equivalencias entre estas, y en particular, una medida general, usualmente utilizado para saber si un triángulo es de buena o mala calidad, es el del ángulo interior mínimo [1], cuyo valor umbral usual puede variar entre 20° o 30°, puede que más o menos, según la necesidad. Este mismo es el utilizado en el actual trabajo.

Desde el punto de vista teórico es deseable obtener ángulos mínimos de 30 grados, aunque para las mayoria de las aplicaciones son aceptables triángulos con ángulos de 15 a 20 grados.

Luego de establecer un «criterio de calidad», se dice que una malla es de calidad (o de «buena calidad») si no contiene «triángulos de mala calidad».

2.3. Refinamiento.

«Refinar: Perfeccionar algo adecuándolo a un fin determinado.» *[RAE, segunda acepción]*

Dada una triangulación, el refinamiento de ésta consiste en la inserción de vértices y en la consecuente generación de triángulos más pequeños. Puede que uno o varios triángulos sean afectados por cada inserción.

Dependiendo del objetivo, el refinamiento considera un grupo determinado de triángulos de mala calidad para ser «refinados». Cuando el objetivo es mejorar una triangulacion se seleccionan todos los triángulos que tienen un ángulo mínimo menor que un parámetro umbral («thershold»), definido por el usuario, para ser refinados. A modo de ejemplo, en el caso del metodo adaptivo de elementos finitos se seleccionan los triángulos cuyo error de la solución numérica de elementos finitos es más grande que el promedio de los errores.

Si se establece un criterio de calidad para una triangulación, se distinguen los triángulos «malos» de los «buenos», siendo los primeros aquellos que no cumplen el criterio de calidad, y los segundos aquellos que sí.

Mejoramiento de una triangulación: Se define mejorar una triangulación como: «Refinar una triangulación de manera de que ya no contenga triángulos malos». Para mejorar un triángulo se inserta un nuevo vértice a través de un proceso que garantiza un incremento en la calidad de la triangulación.

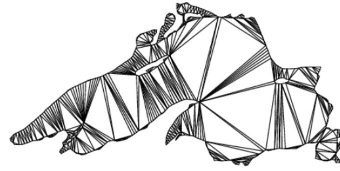
En este documento y de ahora en adelante, se utilizará la palabra «refinar» para referirse a «mejorar».

Para mejorar un triángulo, distintos algoritmos de refinamiento sugieren diferentes ubicaciones para los puntos a ser insertados. El nombre técnico de estos puntos es **«Steiner Points»**[5].

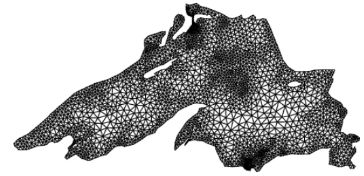
2.4. Distinción de métodos de refinamiento.

Para efectos de este estudio, se considerará sólo algoritmos de refinamiento Delaunay. Se presenta a continuación una descripción general. Más detalles de cada uno pueden ser revisados en la sección de definiciones.

Bordes de una figura que produce un PSLG (Grafo planar de líneas rectas)



Triangulación Delaunay restringida del PSLG, con triángulos de mala calidad



Malla mejorada

Figura 2.2: Muestra de proceso de triangulación y refinamiento.

2.4.1. Algoritmos «Lepp» y «No Lepp»

Los algoritmos Lepp, son los basados en el concepto del «Camino de propagación por la arista más larga», o «Lepp» por sus siglas en inglés - Longest edge propagation path[6]. Al seleccionar un triángulo para mejorar, se calcula su Lepp, el cual, a grandes rasgos, es una secuencia de triángulos vecinos, que parte en el seleccionado, y termina en otro llamado «triángulo terminal» (Ver definición en 4.1.20). En estos algoritmos, el refinamiento de un triángulo de mala calidad parte en el extremo del Lepp asociado al triángulo terminal. Cada «Steiner point» insertado acorta el Lepp, por lo que se espera que dicho triángulo seleccionado se mejore en cierta cantidad de pasos (inserciones). En esta categoría se encuentran los algoritmos Lepp-Delaunay centroide, y Lepp-Delaunay punto medio de la arista terminal, ambos considerados para este estudio.

Por otro lado, los algoritmo «No Lepp» consisten en el mejoramiento directo de triángulos malos a través de la inserción de un «Steiner Point», e inmediatamente luego de la inserción se espera que el triángulo malo haya sido mejorado. Dentro de esta última categoría, los considerados para este estudio son: El algoritmo de refinamiento del circuncentro de Ruppert[1], y el algoritmo de refinamiento off-center de Üngor[7].

3. Objetivo del trabajo

3.1. Objetivo general

Estudiar empíricamente el comportamiento de distintos algoritmos de refinamiento, bajo un ambiente de experimentos automatizado y controlado en detalle, construyendo a partir de memorias anteriores y profundizando para lograr una herramienta definitiva.

3.2. Objetivos específicos

- Medir costos y efectos de considerar explícitamente cada uno de los trabajos adicionales implícitos en los algoritmos actuales de refinamiento.
  - Comparar detalladamente el comportamiento de algoritmos Lepp-Delaunay respecto al de Üngor, y Ruppert, con y sin ordenamiento de la cola de procesamiento.
- Definir un conjunto de métricas para la comparación de resultados.
- Establecer una comparación global, entre las ventajas y desventajas de cada algoritmo y modificaciones de éstos.
- Implementar una aplicación de experimentación, para la extracción de dichos datos.
- Re-escribir el código de «Compare2DMesh» a una versión mejorada: corregir problemas, desvincular QT y reemplazar por bibliotecas más estándar, y código más eficiente.
- Concluir parte del trabajo propuesto de las memorias anteriores[4], lo que incluye:
  - Corregir completamente fugas de memoria.
  - Extender el soporte de formatos de archivos de entrada.
- Se espera que la nueva aplicación permita:
  - Cuantificar costo en tiempo, de cada una de las partes de un algoritmo de manera individual:
    - Pre-procesos.
    - Inserción de «Steiner Points».
    - Generación de cola de procesamiento.
  - Extraer otros indicadores interesantes: vértices generados vs. originales, máximo largo de Lepp, largo promedio de Lepp, etc.
  - Producir y ejecutar baterías de refinamientos con distintas configuraciones.
  - Generar reportes de ejecución, incluyendo tablas y gráficos.

- Que sea multiplataforma (Objetivo deseable)

- Comprobar que el comportamiento de los algoritmos basados en «Steiner Points» del tipo Lepp-Centroide no se ve particularmente favorecido por el ordenamiento de los datos de entrada, como en el caso de los algoritmos de Ruppert y Üngor, en donde sí se requiere para obtener resultados óptimos.

Parte II

Desarrollo

Resumen del desarrollo:

El trabajo realizado, el cual gira en torno a la aplicación desarrollada, «BatchRefinement», puede ser organizado en 2 áreas principales

Teórico: Incluye conceptos investigados y utilizados, además de las definiciones de conceptos y algoritmos. Adicionalmente, se modela el problema apoyándose en diagramas de flujo a partir de lo cual se implementó la aplicación principal de este trabajo. La sección 4 está dedicada a extender el tema.

Técnico: Incluye el desarrollo de la aplicación, temas de más bajo nivel como paralelismo, soporte de formatos de archivos, optimizaciones, refactoring de código, y otros temas de trabajo como la visualización, compatibilidad, experimentación, generación de resultados por lotes y formatos de salidas. Estos tópicos se revisaran en la sección 5.

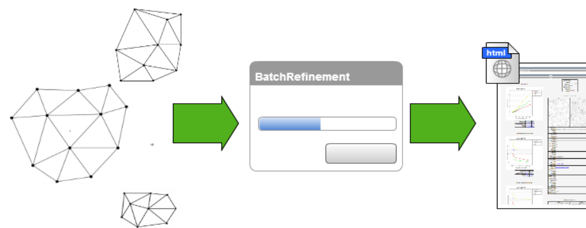


Figura 3.1: La funcionalidad principal de «BatchRefinement»: «Generar reportes de visualización de resultados html a partir de conjuntos de experimentos de algoritmos de refinamiento sobre triangulaciones Delaunay».

4. Terminología y algoritmos

4.1. Definiciones

Se proveen definiciones, descripciones y explicaciones, presentados siguiendo un ordenamiento lógico inductivo de los conceptos claves de la memoria, y en algunos casos pertinentes, adjuntando un recuadro de pseudo código para relacionar el concepto con su implementación computacional, la cual fue escrita en C++11, como se describe en profundidad en la sección 5.

4.1.1. Punto

Elemento geométrico denotado por un par ordenado $(x, y) \epsilon \mathbb{R}^2$ que define una coordenada en un espacio bidimensional. En la implementación, cada coordenada se ve representada por un número de punto flotante de doble precisión, de 64 bits, conocido típicamente como «double».

Pseudo-código:

```
class Point{
    double x;
    double y;
}
```

4.1.2. Vértice

Elemento topológico basado en un **punto**, que adicionalmente contiene un identificador dentro de la triangulación a la que pertenece. Dentro de la implementación, el identificador corresponde al índice dentro del arreglo que contiene todos los triángulos de la triangulación.

Pseudo-código:

```
class Vertex, inherits Point{
    //double x;
    //double y;
    int id;
}
```

4.1.3. Arista

Línea que une dos **vértices** de un **triángulo**. Se denota como un par ordenado de índices de vértices $a_{(i)} = (v_{((i+1)\,\%3)}, v_{((i+2)\,\%3)})$, donde $\%$ es el operador «módulo». Explícitamente, si se utiliza índices base cero, las aristas de un triángulo son:

- $a_0 = (v_1, v_2)$
- $a_1 = (v_2, v_0)$
- $a_2 = (v_0, v_1)$

En la implementación no existe una estructura para manejar aristas ya que estas son referenciadas sólo por un índice, a partir del cual se deducen los puntos que la conforman en el contexto de un triángulo.

4.1.4. Aristas restringidas

Conjunto de aristas que se presentan en una triangulación con la condición de que deben persistir al proceso de refinamiento.

Un factor de preocupación en los algoritmos Delaunay consiste en que es posible que se formen nuevas aristas o descarten debido al procedimiento de «intercambio de diagonales». Este no es un comportamiento deseable en algunos casos, por lo cual se definen inicialmente un conjunto de aristas que no deben ser removidas durante el refinamiento. A éstas se le conoce como «aristas restringidas» y tienen un tratamiento especial al momento de trabajar con ellas. En general, y bajo ciertas consideraciones, lo importante es que la arista restringida original aun exista, incluso como la unión de varias de las resultantes. En este trabajo, subdividir una arista restringida está permitido, lo cual producirá dos nuevas «**aristas restringidas**» que reemplazarán la original.

En los algoritmos implementados, el objeto principal de la triangulación posee una lista de las aristas restringidas, las cuales se definen como pares ordenados con los identificadores de los vértices que la conforman. Estas deben ser especificadas al inicio del proceso de refinamiento.

4.1.5. Aristas «encroached»

Se dice que una arista restringida es **«encroached»** si algún vértice de la triangulación yace en el interior de su **«círculo diametral»**, el cual es el más pequeño(y único) círculo que contiene dicha arista. Estas son de interés, debido a que dada su distribución de vértices, es posible que los **«Steiner Points»** a insertar en posteriores iteraciones de refinamiento se posicionen fuera del dominio de la triangulación. Para manejar este contratiempo, existen procedimientos de «corrección» de aristas «encroached».

El procedimiento considerado en este trabajo es el de bisecciones sucesivas: Si se detecta alguna arista «encroached», ésta se corrige inmediatamente por medio de la subdivisión Delaunay de la arista restringida en su punto medio. Las dos aristas restringidas resultantes tienen círculos diametrales más pequeños que el original, y es posible que estas mismas sean «encroached». En este último caso, el proceso se repite sucesivamente hasta que no queden más aristas «encroached».

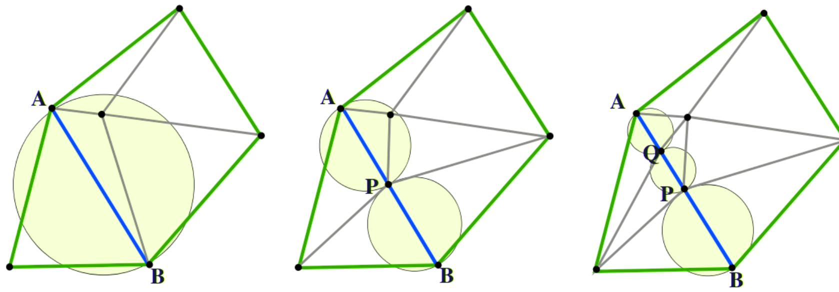


Figura 4.1: Procedimiento de corrección de aristas **«encroached»** por medio de bisecciones sucesivas.

4.1.6. Triángulo

3-tupla de vértices distintos no colineales. Como consecuencia de esto último, su área es siempre mayor a cero. Debido a motivos de consistencia en cálculos, se define un «sentido de orientación correcto». En este trabajo en particular, su orientación es correcta si el orden de los vértices está definido en sentido antihorario. En la implementación, un triángulo tiene adicionalmente un identificador dentro de la triangulación, y guarda propiedades adicionales en variables de instancia, como el ángulo interior menor, largos de sus aristas, etc.

Pseudo-código:

```
class Triangle{
    Vertex v0;
    Vertex v1;
    Vertex v2;
    int id;
}
```

4.1.7. Círcuncírculo

En geometría, la circunferencia circunscrita es la circunferencia que pasa por todos los vértices de una figura plana y contiene completamente a dicha figura en su interior. La figura en este caso es un triángulo. El centro de la circunferencia circunscrita se llama **circuncentro** y su radio **circunradio**. (Figura 4.2)

4.1.8. Malla Geométrica

Colección de vértices, aristas y caras, que aproxima la forma de una superficie u objeto complejo en base a polígonos. Otros usos incluyen la distribución espacial de valores puntuales asociados a vértices, como se utiliza en el análisis de elementos finitos . En este trabajo sólo se considerará como una superficie bidimensional, y las caras que la componen serán solamente

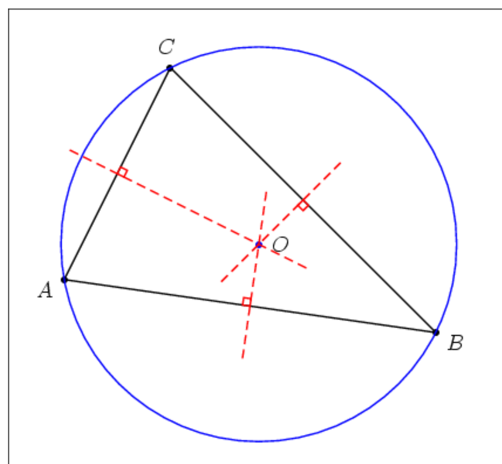


Figura 4.2: Circuncírculo y circuncentro «O» de un triángulo ABC.

triángulos. (Ver definición de Triangulación 4.1.11)

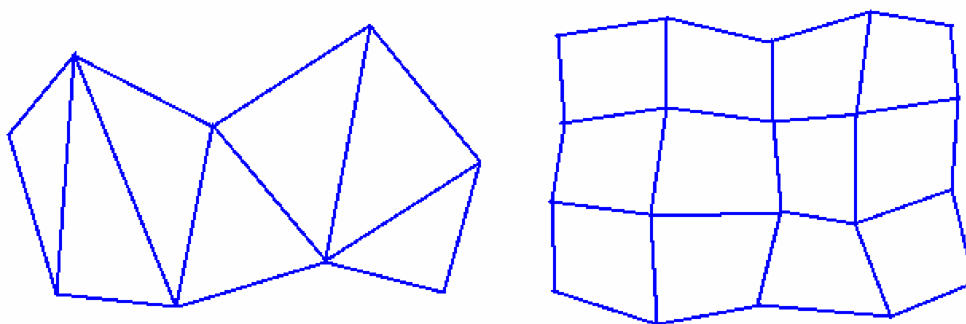


Figura 4.3: Mallas geométricas con caras triangulares y cuadrilaterales bidimensionales.

4.1.9. Grafo (o grafo no dirigido)

Es un conjunto de objetos llamados vértices (o nodos) unidos por enlaces llamados aristas(o arcos), que permiten representar relaciones binarias entre elementos de un conjunto. Son típicamente representados de manera gráfica como un conjunto de puntos (vértices o nodos) unidos por líneas (aristas). (Fig. 4.5)

4.1.10. Grafo planar de línea recta

Un grafo planar, es una colección de vértices y segmentos que puede ser dibujado en el plano sin que ninguna arista se cruce. Un caso especial es el Grafo Planar de Línea Recta, o **PSLG** por sus siglas en inglés (Planar straight-line graph) en el cual, como su nombre lo indica, sus

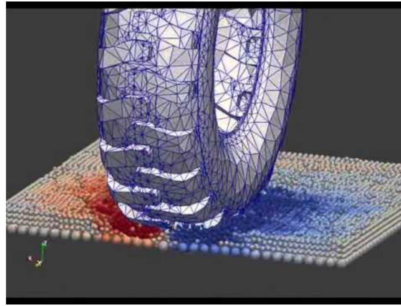


Figura 4.4: Utilización de mallas geométricas en análisis de elementos finitos.

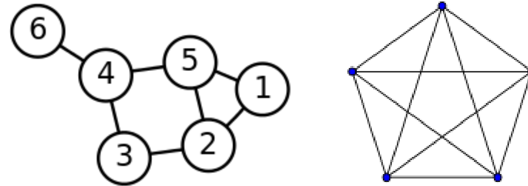


Figura 4.5: Ejemplos de grafos no dirigidos.

arcos están representados solamente por líneas rectas. (Fig. 4.6)

Los bordes de objetos pueden ser representados por PSLG fijando los vértices en coordenadas espaciales.

Es usual formar triangulaciones a partir de los vértices de un PSLG. Más específicamente, se generan triangulaciones Delaunay restringidas a partir de éstos, considerando los nodos del PSLG como vértices, y sus arcos como «aristas restringidas» (Fig.4.7).

Se define **«segmento»** como un arco, o arista del grafo planar, y se extiende esta notación para referirse a las aristas de la triangulación que deben persistir luego del refinamiento. Es decir, un **segmento**, es una **arista restringida**.

4.1.11. Triangulación

En el ámbito de la geometría, una triangulación de un objeto planar es una subdivisión en triángulos. Distintos tipos de triangulaciones se pueden definir dependiendo en cuál objeto geométrico es subdividido y de qué manera es determinada la subdivisión. Algunas definiciones son:

- Considerando un espacio bidimensional, una **«triangulación de un conjunto discreto de puntos»** $P \epsilon \mathbb{R}^2$, es la subdivisión, de la envoltura convexa de los puntos, en triángulos cuya intersección con un vecino es una arista o un vértice, y tal que los vértices coinciden con P. Una de las triangulaciones de éste tipo más estudiada es la Triangulación de Delaunay.

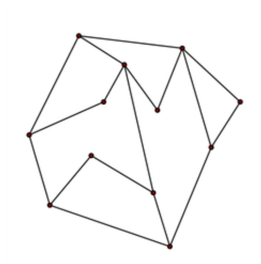
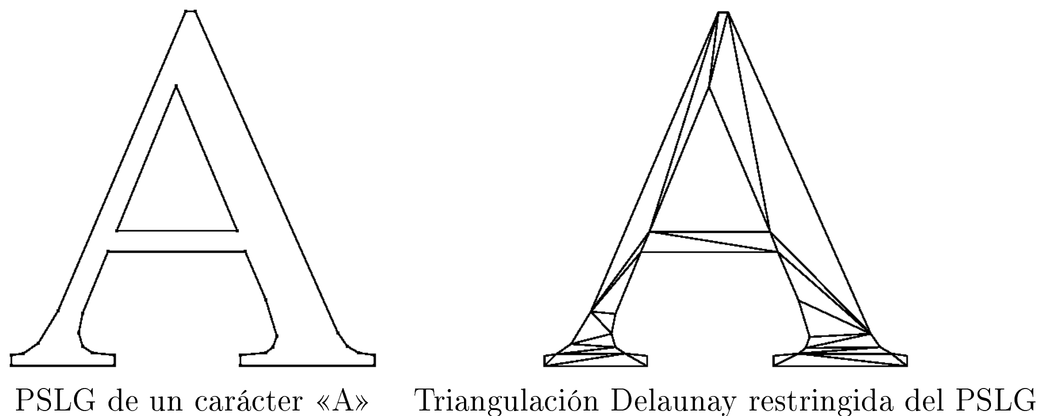


Figura 4.6: Ejemplo de grafo planar de línea recta.



PSLG de un carácter «A» Triangulación Delaunay restringida del PSLG

Figura 4.7: Triangulación Delaunay restringida obtenida a partir de un grafo planar.

- En cartografía, una **«red irregular de triángulos»** es una triangulación de un conjunto de puntos (bidimensionales) que incluyen adicionalmente información de elevación. Al levantar cada punto del plano a su altura de elevación, se producen superficies tridimensionales, las generan una representación aproximada de un terreno.

- Una **«triangulación de un polígono»** es una subdivisión de éste en triángulos que se conectan por sus aristas, con la propiedad de que el conjunto de los vértices de los triángulos coinciden con el del polígono. Pueden ser calculadas en tiempo «casi linear» $O(n * log(n))$ [8] y forman la base de varios algoritmos geométricos importantes. Una triangulación Delaunay restringida es una adaptación de las triangulaciones Delaunay de un conjunto de puntos, y que produce la triangulación de un polígono, o dicho de manera más general, de un grafo planar (PSLG).

Las triangulaciones, consideradas en este trabajo, son casos particulares de mallas geométricas cuyas caras son triangulares.

En el análisis de elementos finitos, es usual que se utilicen triangulaciones como la malla subyacentes en los cómputos. En este caso, los triángulos forman una subdivisión del dominio a ser simulado, y en algunos casos se permite agregar puntos adicionales «Steiner Points» como vértices a través de un proceso de refinamiento, y así minimizar errores de aproximación que se producen producto de la discretización y muestreo del objeto «continuo» original. A este

proceso se le denomina «refinamiento».

4.1.12. Triangulación válida

Dada una triangulación, existen formas de comprobar su validez. Se defina **«triangulación válida»**, al conjunto de triángulos que cumple las siguientes condiciones [3]:

1. La intersección del interior de dos triángulos distintos siempre será vacía.

    a) La intersección de dos triángulos vecinos puede ser sólo una de las dos opciones siguientes:

        1) Un vértice.
        2) Dos vértices y el segmento de recta que une ambos vértices.

    b) La unión de todos los triángulos forma un conjunto conexo.

4.1.13. Triangulación Delaunay

Triangulación que cumple la condición de Delaunay. Esta condición dice que el interior de la circunferencia circunscrita de cada triángulo de la red no debe contener ningún vértice de otro triángulo. A esta condición se le llamará como **«prueba del circuncentro»**. (Figura 4.8)

En éste tipo de triangulaciones, se cumplen las siguientes propiedades:

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

    a) La frontera de la triangulación es la envoltura convexa de los puntos.

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

    En el refinamiento de mallas de triángulos, la propiedad de Delaunay cumple un rol fundamental, debido a que garantiza que no hay otra configuración que produzca mejores ángulos mínimos interiores.

    Existen algunas distribuciones de puntos donde es posible obtener más de una triangulación Delaunay. En particular, los casos donde existen cuatro (o más) puntos que comparten el mismo circuncírculo, en cuya instancia se puede realizar un intercambio de diagonales, manteniendo la propiedad de Delaunay, conservando el ángulo interior mínimo, pero formando una triangulación distinta.

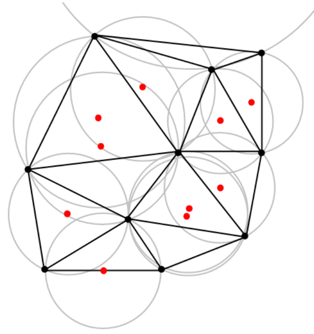


Figura 4.8: Triangulación Delaunay con todos los circuncírculos y sus respectivos circuncentros.

Hay formas para obtener una triangulación Delaunay a partir de una que no cumpla la condición. Un procedimiento de fundamental importancia que merece ser nombrado es el «intercambio de diagonales», que es utilizado constantemente en las iteraciones de refinamiento para mantener la propiedad Delaunay y así maximizar el ángulo interior mínimo.

En este trabajo de memoria, se utilizan inserciones tipo Delaunay basadas en secuencias de intercambio de diagonales, al añadir nuevos puntos a las triangulaciones.

4.1.14. Procedimiento de intercambio de diagonales

Dados 2 triángulos vecinos, es decir, que comparten exactamente dos vértices, existen 2 posibles triangulaciones. Si la triangulación de los 4 vértices considerados no cumplen con el test del circuncírculo, se puede realizar un intercambio de diagonales para así obtener una triangulación Delaunay. (Figura 4.9)

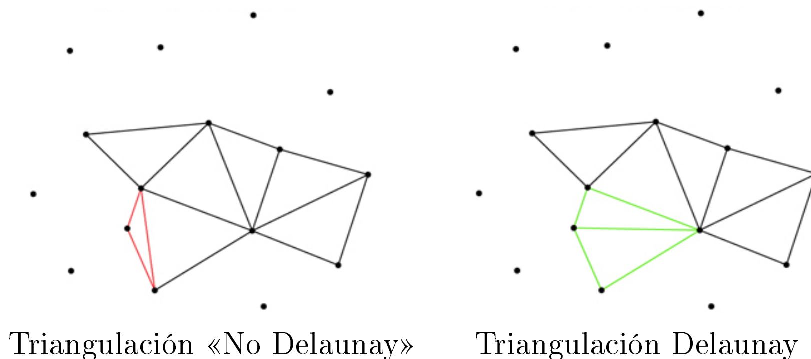


Figura 4.9: Procedimiento de intercambio de diagonales.

4.1.15. Triangulación Delaunay restringida

Al producir una triangulación de un grafo «PSLG», se dice que es «Delaunay restringida» si todos los segmentos del grafo original aparecen en la triangulación resultante [9]. Debido a que la condición de mantener aristas intactas puede impedir el mejoramiento de ciertos triángulos, el resultado puede no ser una triangulación completamente Delaunay.

A pesar de que este tipo de triangulaciones tiene propósitos particulares, no son de tanto interés en la práctica debido a que, dado lo estricto de la restricción, posee una resistencia al mejoramiento de ciertos triángulos de mala calidad. Una alternativa a este tipo, es la «triangulación Delaunay conforme», cuya definición algo menos estricta permite obtener triángulos con mayores ángulos interiores.

4.1.16. Triangulación Delaunay conforme

Dada una triangulación de un grafo PSLG, se dice que es Delaunay conforme si ésta cumple con la condición de Delaunay, y todos los segmentos del grafo original aparecen en la triangulación, considerando incluso uniones de aristas [9].

Su definición es un poco más flexible que la «Triangulación Delaunay restringida» gracias a lo cual es más simple de manejar, y debido a que permite mantener las mismas propiedades que las «restringidas» es la más utilizada en los refinamientos.

Para ilustrar más claramente la diferencia, se puede observar la siguiente figura (4.10), obtenida directamente del sitio de «Triangle» [9].

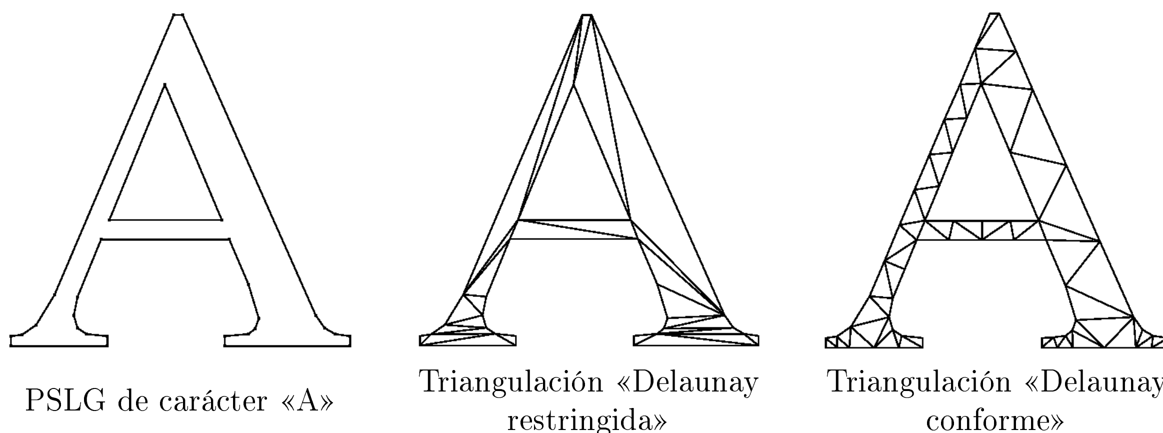


Figura 4.10: Diferencia entre triangulación Delaunay «restringida» respecto a «conforme» de un PSLG.

Los arcos (aristas) del PSLG son considerados como aristas restringidas durante el proceso de refinamiento. En la implementación, todos los refinamientos producen «Triangulaciones

Delaunay conformes». Es relevante notar que el tratamiento de las aristas restringidas es el mismo que las de borde, debido a que estas últimas, son básicamente casos particulares de las primeras.

4.1.17. Inserción Delaunay

Procedimiento que inserta un punto produciendo una triangulación Delaunay. Para esto, las alternativas de procedimiento son:

Inserción con intercambio de diagonales: Se agrega el nuevo punto al o los triángulos que lo contienen y se realiza el número necesario de intercambio de diagonales para restablecer la triangulación Delaunay. Dicho intercambio es necesario, ya que es común que al insertar un nuevo vértice, ya sea generando un nuevo triángulo o dividiendo alguno existente, se pierda localmente la calidad de Delaunay (Puede que él, o los triángulos nuevos, incluso sus vecinos, no pasen la prueba del circuncentro).

Alternativa de la cavidad: Se sacan todos los triángulos que no cumplen el «test del círculo» respecto al nuevo punto «p» a insertar. Luego se une "p" con los vértices del borde del polígono resultante, obteniendo de esta forma una triangulación Delaunay.

En las pruebas realizadas en este trabajo, sólo se utilizó la «inserción con intercambio de diagonales».

4.1.18. Calidad de una triangulación

Según el criterio del mínimo ángulo interior, un triángulo se dice de «buena calidad» si la medida del menor ángulo interior es mayor o igual a cierto umbral. Entonces, se puede decir inequívocamente que una malla o triangulación es de buena calidad si está compuesta sólo por triángulos de buena calidad., o en otras palabras, cada uno de los ángulos interiores de los triángulos que la componen, son mayores al valor umbral. La presencia de triángulos cuya calidad es inferior al umbral, implica posibles errores de aproximaciones o cálculos, debido a lo cual, es deseable el mejoramiento de todos los que sea necesario[1].

4.1.19. Steiner point

En el contexto de triangulaciones Delaunay, los puntos insertados al ejecutar un refinamiento tipo Delaunay son llamados **«Steiner Points»** [5]. No forman parte del conjunto de puntos iniciales de la triangulación. Cada algoritmo de refinamiento define una ubicación distinta para la inserción de cada uno de estos puntos. Ubicaciones usuales, según el algoritmo, incluyen: el circuncentro del triángulo, el «off-center», el punto medio de la arista terminal, etc, algunos de los cuales se encuentran descritos en los algoritmos de esta sección.

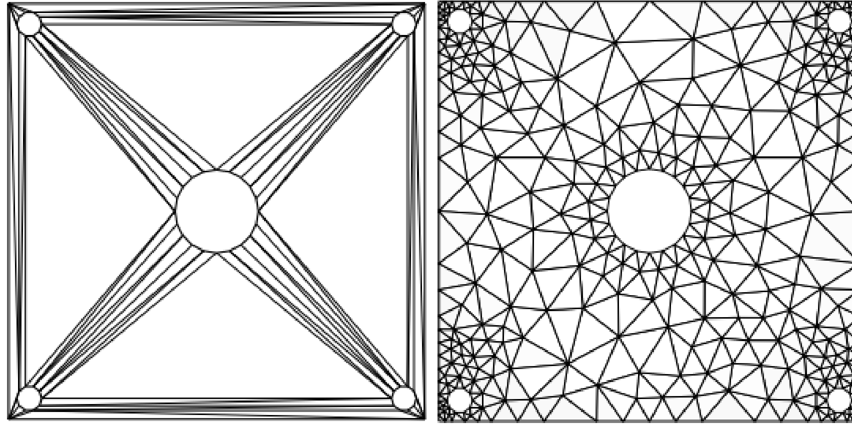


Figura 4.11: Malla geométrica de mala calidad (a la izquierda) al lado de la versión mejorada (a la derecha).

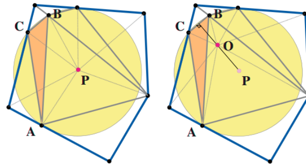


Figura 4.12: Posibles «Steiner Points»: «O» (a la izquierda) o «P»(a la derecha), los cuales son candidatos a inserción dependiendo del algoritmo y condiciones locales.

4.1.20. Lepp

El camino de propagación por la arista más larga (o LEPP por sus siglas en inglés para Longest «Edge Propagation Path») asociado a un triángulo t_0 corresponde a una secuencia de triángulos $(t_0, ..., t_n)$ de la malla que cumplen que para todo $i \epsilon (0, ..., n-1)$, el triángulo $t_i + 1$ es vecino del triángulo de t_i a través de la arista más larga de este último [6]. El camino termina cuando, el vecino de un triángulo, calculado a través de su arista más larga, no existe, o corresponde a su predecesor en el Lepp. La arista más larga del «triángulo terminal» t_n se conoce como **«arista terminal»**.

Cabe notar que, dado un Lepp, para cada triángulo miembro de éste se puede calcular un Lepp, que es subconjunto del original. En la figura 4.13, partiendo desde t_0 se tiene el Lepp $\{t_0, t_1, t_2, t_3, t_4\}$, y si el triángulo inicial fuese t_1, se tendría un Lepp$\{t_1, t_2, t_3, t_4\}$, y así. Esto es relevante debido a que permite ciertas optimizaciones en los cálculos de LEPP siempre y cuando se cuente con la capacidad de almacenarlos en memoria. Por ejemplo, si un algoritmo calculara primero el LeppL_2 de t_2, y más adelante, calculara el Lepp L_1 de t_1, bastaría con insertar t_1 al principio de L_2.

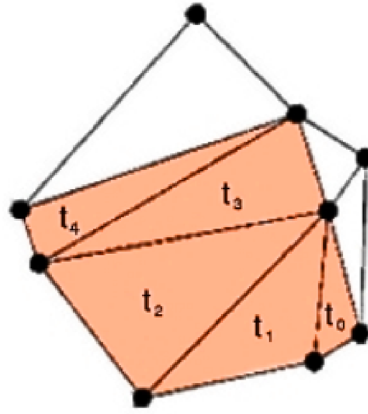


Figura 4.13: Lepp de un triángulo t_0, compuesto por $\{t_0, t_1, t_2, t_3, t_4\}$.

4.1.21. Arista terminal

Dado un **Lepp** definido por los triángulos $(t_0, ..., t_n)$, se define **«arista terminal»**, a la arista más larga del triángulo t_n

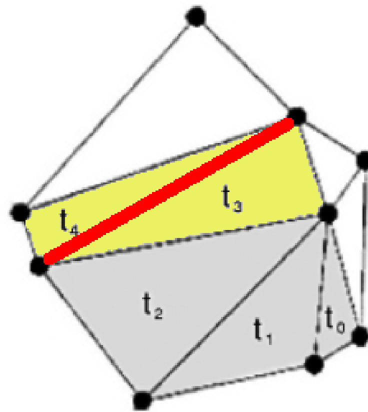


Figura 4.14: «Arista terminal» (rojo) y «triángulos vecinos terminales» (amarillos) de un Lepp$\{t_0, t_1, t_2, t_3, t_4\}$.

La arista terminal y los triángulos que forman el cuadrilátero terminal son de gran importancia en los algoritmos Lepp, debido a que los «Steiner Points» son calculados de acuerdo a las coordenadas de éstos, dependiendo del algoritmo.

4.1.22. Colas

En todos los algoritmos de refinamiento se calcula un conjunto inicial de triángulos que deben ser mejorados. Luego, estos son procesados uno a uno hasta que ya no queden más. El orden en el cual son escogidos define una «cola de procesamiento».

El orden en el cual los triángulos son procesados influye en el resultado obtenido. Por ejemplo, dado una triangulación y un ángulo de exigencia, el algoritmo de Üngor produce distintas triangulaciones dependiendo si los triángulos fueron procesados ordenados(de acuerdo al ángulo interior más pequeño, de manera creciente) o sin ordenamiento. Estas triangulaciones resultantes presentan diferencias en el número de vértices finales, donde se producen menos triángulos en el caso ordenado respecto al caso sin ordenamiento.

Durante las iteraciones de refinamiento, no es inusual que se produzcan nuevos triángulos malos, los cuales deben ser ingresados a la cola. De acuerdo a la configuración de ejecución del refinamiento, se pueden escoger distintos ordenamientos. Por ejemplo, se puede escoger ordenarlos de manera creciente según el valor del ángulo interior mínimo, según el largo de la arista más pequeña, o simplemente de acuerdo al mismo orden de aparición obtenido al leer la entrada.

Se puede distinguir 2 tipos de colas de procesamiento de acuerdo al ordenamiento.

1. Colas «Primero en entrar, primero en salir» (o FIFO, por sus siglas en inglés «First In, First Out»), en las cuales el orden de procesamiento está dado por el de llegada. Si se asume que las entradas no presentan un orden en particular, es seguro decir que estas colas equivalen a procesar los triángulos de manera desordenada (o aleatoria). Estas colas tienen la ventaja de ser simples, en cuanto a implementación y costo de operación.

2. Colas de prioridad: En estas colas, los elementos están siempre ordenados de acuerdo a una llave, la cual corresponde, en el caso de un refinamiento, al valor de la propiedad del triángulo asociada al criterio de calidad, como el ángulo mínimo.

   Las estructuras de datos que soportan este tipo de cola tienen un costo de ejecución generalmente mayor, debido a que necesitan mantener un orden de los elementos. A modo explicativo, es importante notar que una cola FIFO funciona de manera análoga, aunque más lenta, a una cola de prioridad cuya llave de ordenamiento es el «puesto» en el que el elemento fue insertado.

La complejidad computacional de las principales funcionalidades de las colas descritas, se lista en la siguiente tabla.

Tipo de cola	**Características**	**Costo de inicialización (inserción de cada elemento)**	**Costo total de ejecución (extraer todos los elementos)**
FIFO	Más simple y rápida. Elementos sin orden en particular	$O(n)$	$O(n)$
Cola de prioridad	Mantiene siempre elementos ordenados. Más lenta que la FIFO	$O(n * log(n))$	$O(n * log(n))$

En los experimentos, el impacto de rendimiento al usar colas de prioridad en vez de FIFO no fue significativo a pesar de las diferencias de costos.

4.2. Algoritmos de refinamiento.

En esta sección se hará una revisión de los conceptos y abstracciones a partir de las cuales se construyó el trabajo, compuesta por las definiciones abstractas de cada término considerando levemente el tema de la implementación. Para detalles de ésta última, se debe ir a la sección 5.

Existen varios algoritmos de refinamientos de triangulaciones, incluso algunos que no mejoran la calidad de las mallas y cuyo objetivo es simplemente obtener una geometría con triángulos más pequeños, los cuales no serán revisados en este trabajo debido que no pertenecen a la temática relevante. Sin embargo, sí forman parte de la aplicación, por herencia de los trabajos anteriores.

En este documento, los algoritmos de refinamiento se organizarán de acuerdo a características en común siguiendo la jerarquía mostrada en la figura 4.15, y se dará énfasis a los «Algoritmos de Refinamientos Delaunay».

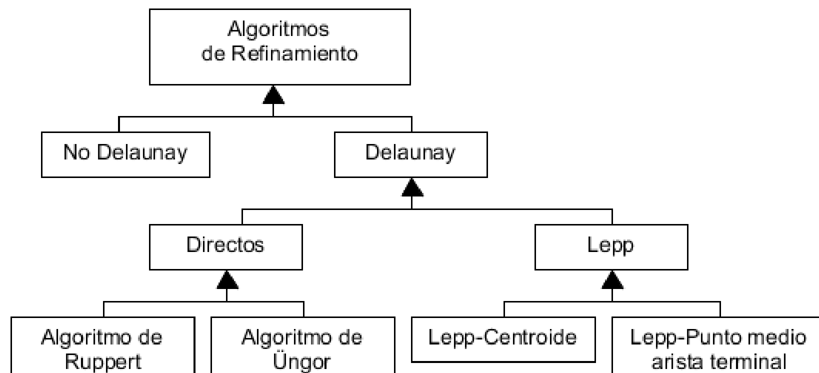


Figura 4.15: Jerarquía propuesta de algoritmos de refinamiento.

En general, estos algoritmos comparten una estructura en común, la cual consiste en un ciclo de iteraciones, donde cada «iteración» que se compone de algunos «pasos» genera una inserción que cambia la geometría, y cada uno se distingue de los demás principalmente en base a las decisiones que toma al calcular las coordenadas de cada nueva inserción de vértices. En la implementación, todos los pasos a realizar son elegibles a partir de opciones iniciales, por lo que se puede experimentar con modificaciones de los algoritmos bases. Este tema se verá en más detalle en el apartado dedicado a la aplicación.

4.2.1. Algoritmos de refinamientos no Delaunay

Dependiendo del fin, se puede requerir simplemente obtener una malla con triángulos de menor tamaño, sin necesidad de producir un mejoramiento de ángulos interiores ni mantener la propiedad de Delaunay. Para este propósito, existen algunos algoritmos de subdivisión de triángulos que cumplen con el objetivo, y que son más simples que los de refinamiento Delaunay. Entre estos se puede mencionar el algoritmo Lepp-Bisección, el cual mantiene la calidad de la malla inicial [10]. En la implementación, se consideró sólo este último, basado en la subdivisión de los triángulos objetivos, por medio de la bisectriz de la arista más larga.

4.2.2. Algoritmos de refinamiento Delaunay

Haciendo uso de la propiedad de maximización del menor ángulo interior de las triangulaciones Delaunay, se pueden obtener mallas de mejor calidad al elegir apropiadamente nuevos puntos a insertar. Todos los algoritmos de esta categoría se basarán en inserciones seguidas por intercambios de diagonales cuando el agregado de un nuevo vértice no cumpla la condición de Delaunay. Los documentos de memoria de título anteriores y sus aplicaciones relacionadas incluían implementaciones de cada uno, los cuales fueron portados para este trabajo. Los detalles pueden ser consultados en las publicaciones respectivas ([4, 3]).

Serán agrupados en esta sección según su definición esté basada en el concepto de «Lepp» o no.

4.2.2.1. Algoritmos Lepp Son todos los algoritmos cuyo procesamiento depende del cálculo del «Lepp» asociado a cada triángulo de mala calidad. En la aplicación, los algoritmos disponibles dentro de esta categoría son:

- Lepp-Centroide: Caracterizado por la elección principal de nuevos «Steiner Points» basados en el centroide del cuadrilátero o triángulo terminal, según sea el caso [11]. (Llamado «Lepp-Centroid» en la implementación)
- Lepp-PuntoMedio: Caracterizado por la elección principal de nuevos «Steiner Points» basados en el punto medio de la arista terminal del Lepp [10]. (Llamado «Lepp-MidPoint» en la implementación)

4.2.2.2. Algoritmos «No Lepp» Los algoritmos en los cuales no se calculan «Lepps». En la aplicación, los algoritmos disponibles dentro de esta categoría son:

- Ruppert: Caracterizado por la elección principal de nuevos «Steiner Points» basados en el circuncentro del triángulo de mala calidad [1]. (Llamado «Ruppert» en la implementación)

- Üngor: Caracterizado por la elección principal de nuevos «Steiner Points» basados en el «off-center de Üngor» [7]. (Llamado «Lepp-MidPoint» en la implementación)

Implementaciones de algoritmos de refinamiento.

Es importante mencionar que las implementaciones de los algoritmos previamente mencionados están basadas en las posibles configuraciones de algoritmos personalizados(ver4.3). Como consecuencia de esto, algunos pasos pueden presentar variaciones respecto a los algoritmos originales debido a que aun no existen opciones suficientes en la aplicación para representarlos con exactitud. Uno de los casos más notorios es el proceso de corrección de aristas «encroached», cuya implementación actual sólo permite la inserción de puntos bisectores de la arista restringida, y no es dependiente del caso como describe, por ejemplo, el algoritmo de Üngor original[7], lo que puede perjudicar la calidad resultante .

4.3. Algoritmos mixtos y personalizados

Gracias a que los programas anteriores, en los cuales está basada la aplicación actual, soportan distintas opciones para ejecutar refinamientos, se logró identificar partes relevantes de los procedimientos, en las cuales se pueden realizar cambios paramétricamente y así producir algoritmos personalizados. Entonces, es posible crear por ejemplo, un algoritmo con preprocesamiento de corrección de aristas «encroached», siguiendo un esquema de procesamiento Lepp, con «Steiner Points» basados en el Off-center de Üngor, y similares. Dada tal libertad de experimentación, es posible también generar algoritmos que no convergen, o de comportamiento poco óptimos, por lo cual es importante tener en cuenta lo que se requiere y así evitar pérdidas de tiempo innecesarias.

Como medida de claridad se hará distinción de los algoritmos mixtos y personalizados de acuerdo a las siguientes definiciones:

Algoritmos personalizados: Son todos los algoritmos que se pueden realizar utilizando la aplicación. Se les llama personalizados porque se puede controlar cada una de las fases en base a opciones iniciales individuales.

Algoritmos mixtos: Por un lado, se pueden reproducir los algoritmos más conocidos utilizando algoritmos personalizados al seleccionar las opciones iniciales apropiadas. Se le llamará «algoritmos mixtos» a todos los algoritmos personalizados que no corresponden a los cánones, o que suponen una leve modificación a estos. Esta distinción no tiene mucho efecto práctico, y se dicta con propósitos de desambiguación.

Configuración de algoritmos

Para efectos de este estudio y guiado por las memorias anteriores, se definen los siguientes conceptos que componen todas las configuraciones para generar algoritmos personalizados.

Archivo de entrada: Es el archivo que define la triangulación original. Los formatos soportados son descritos en la sección de implementación.

Priorización de triángulos: Define la priorización de la cola de triángulos para procesar. Es lo que determina el orden de procesamiento de los triángulos «malos».

Si se utiliza la «cola de prioridad», entonces los «triángulos malos» creados durante etapas intermedias de los algoritmos se integrarán a la cola según su «llave de prioridad»

Los valores disponibles pueden ser cualquiera de los siguientes:

- Sin prioridad: Los triángulos se procesan de manera «desordenada». Es decir, como una cola FIFO, siguiendo el orden de lectura sin considerar ninguna de sus características.
- Menor ángulo interno: Los triángulos se procesan ordenadamente, donde los primeros son los con menor ángulo interior, los últimos los con mayor.
- Menor circunradio: Los triángulos se procesan ordenadamente, donde los primeros son los con menor circunradio, los últimos los con mayor. El circunradio es el radio de la circunferencia circunscrita.
- Otros: Se puede incluir cualquier criterio local a un triángulo, incluyendo los casos inversos de los ya listado.

Priorización de aristas «encroached»: De manera similar a los triángulos, las aristas «encroached» que deben ser corregidas pueden ser escogidas en un orden particular, de acuerdo a características de ésta misma o de alguno de los triángulos que la poseen.

Ángulo mínimo exigido: Es el ángulo mínimo a partir del cual se definen los triángulos malos inicialmente, en otras palabras, a partir de lo que se llena la cola de triángulos para ser procesados mediante el refinamiento.

Elección de Steiner Point: Representa la elección de las coordenadas del nuevo punto a insertar según el algoritmo de refinamiento.

Tipo de inserción: Existen distintas formas implementadas [3] para insertar un nuevo vértice a una triangulación, las cuales son:

- Partición simple del triángulo que contiene al nuevo punto (No Delaunay)
- Método de inserción de la «cavidad» (Conserva propiedad Delaunay)
- Inserción seguida por subsecuentes cambios de diagonales para mantener propiedad Delaunay.

Para los métodos de mejoramiento del ángulo interior mínimo, sólo se considerará esta última, ya que es la más simple y eficiente que optimiza el menor ángulo interior de una triangulación.

Preproceso de corrección de aristas «encroached»: A través de esta variable se define el tratamiento de las aristas «encroached».

Se debe elegir cada una de las variables anteriores para producir un conjunto de «Opciones» iniciales que afectan al «algoritmo mixto» o «personalizado». Los siguientes diagramas ilustran el flujo a través del cual se puede ejecutar un ciclo de refinamiento dadas determinadas opciones iniciales.

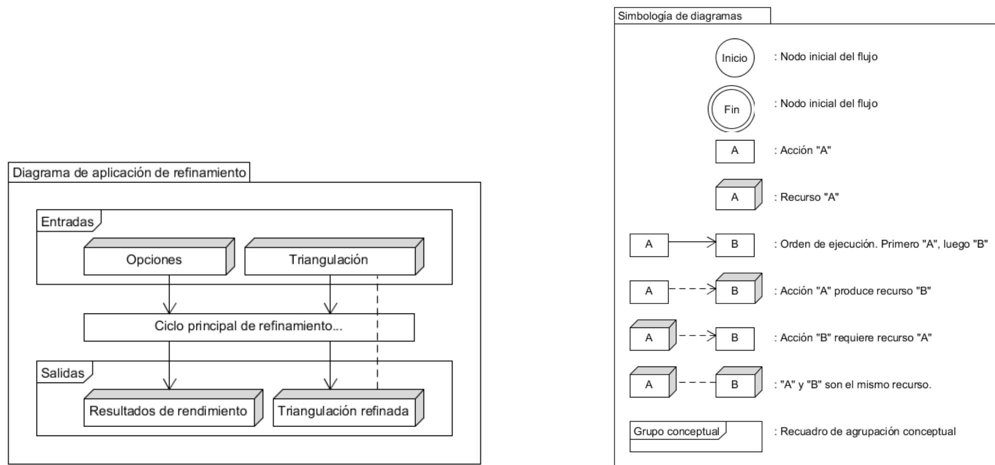


Diagrama básico del proceso de refinamiento de los algoritmos personalizados.

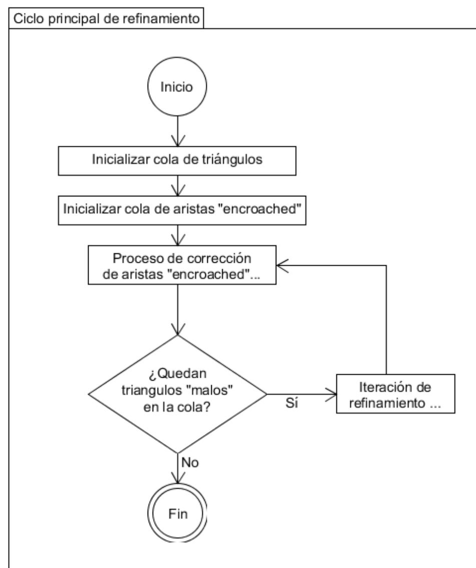
Simbología utilizada por los diagramas de flujo del documento

Figura 4.16: Diagrama básico y simbología.

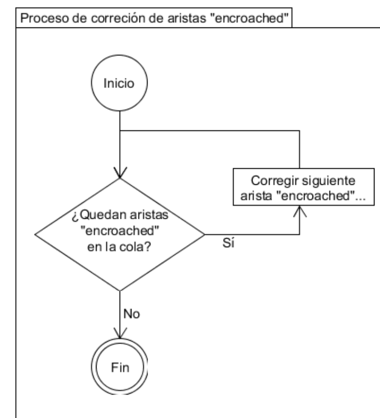
Para mantener cierta simplicidad, se listarán algunos diagramas partiendo de una figura básica, para luego ir profundizando en algunos pasos para obtener un acercamiento modularizado de las partes relevantes del algoritmo personalizable.

Como indica el diagrama básico del «proceso de refinamiento», los recursos de entrada son una triangulación, y un conjunto de opciones iniciales que definen el algoritmo personalizado. Luego, se produce el refinamiento y como salida se obtiene la triangulación refinada y un objeto con los resultados e indicadores de ejecución del algoritmo.

Profundizando en el «ciclo principal» de la figura 4.16, se puede observar el siguiente diagrama de flujo:



Ciclo principal de los algoritmos personalizados



Proceso de corrección de aristas «encroached»

Figura 4.17: Detalles de diagrama.

En el ciclo de refinamiento, existe una fase de corrección de aristas «encroached», compuesta por otro ciclo interior, como define la figura 4.17. Como se observa esta última, el refinamiento es básicamente un ciclo compuesto por un subciclo de corrección de aristas «encroached», y una iteración de refinamiento de triángulos «malos». El diseño está pensado considerando que cada inserción puede producir aun más triángulos malos o aristas «encroached». El flujo indicado cubre aquellos casos.

Posteriormente, la iteración de refinamiento se detalla como muestra la figura 4.18:

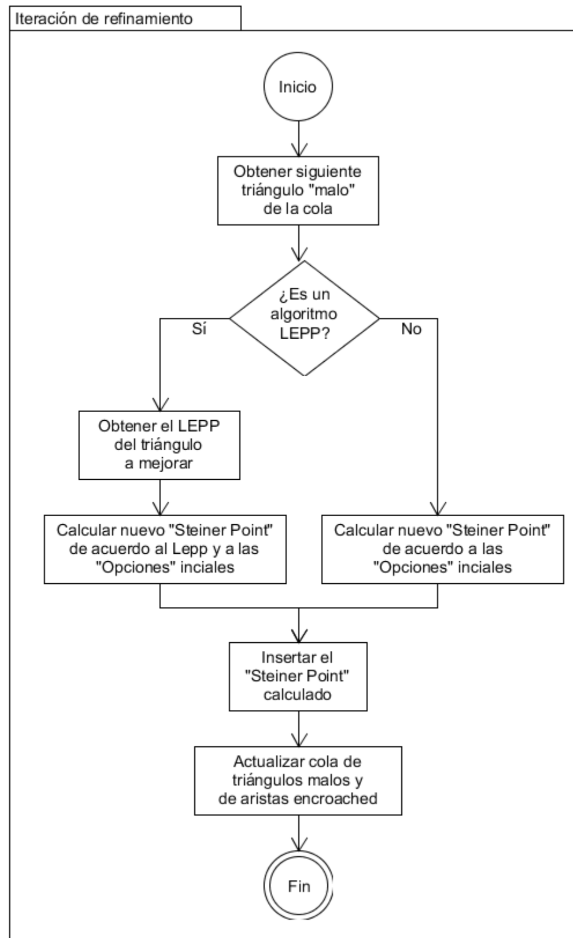


Figura 4.18: Detalle de iteración de refinamiento.

Es importante notar que estos flujos son abstracciones y no dependen de alguna implementación en particular. Para ver consideraciones relacionadas a la implementación, se pueden encontrar con más detalles en la siguiente sección.

4.4. Métricas para comparación de resultados

Para poder comparar el comportamiento de los algoritmos, es necesario definir qué tópicos requieren ser observados y mesurados. Se definieron los siguientes:

- Tiempo efectivo: Es el intervalo de tiempo medido desde el inicio hasta el final del refinamiento. Se mide en milisegundos [ms].
- Número de iteraciones de refinamiento: Es el número de iteraciones de mejoramiento de triángulos. Equivale a contar el número de inserciones realizadas, aunque no incluye los casos de corrección de aristas «encroached».

- Número de iteraciones de correcciones de aristas encroached: Es el número de iteraciones de corrección de aristas «encroached». Equivale a contar el número de inserciones realizadas solo en los casos de corrección de aristas «encroached».

- Lepp más largo: Cada vez que se calcula un Lepp, se guarda el valor del Lepp más largo.

- Ángulo más pequeño inicial: El ángulo más pequeño inicial para una misma malla no varía ni depende del refinamiento aplicado. Sin embargo, se listó ya que puede ser importante cuando se requiere comprar muchas ejecuciones sobre distintas triangulaciones. Esta razón es válida para todos los casos de métricas «iniciales».

- Ángulo más pequeño resultante: Es el valor del ángulo más pequeño de la triangulación posterior al refinamiento.

- Triángulos iniciales: Número de triángulos iniciales. Se listó por las mismas razones que el ángulo más pequeño inicial y que todos las métricas iniciales.

- Triángulos finales: Número de triángulos resultantes posterior al refinamiento.

- Aristas restringidas iniciales

- Aristas restringidas finales: Número de aristas restringidas resultantes posterior al refinamiento.

- Vértices iniciales

- Vértices finales: Número de vértices resultantes posterior al refinamiento.

- Índice de vértices finales vs. iniciales: Es el cociente del número de vértices finales dividido por el número inicial. Indica un índice de crecimiento de la triangulación posterior a la aplicación del refinamiento.

- Índice de triángulos finales vs. iniciales: Análogo al punto anterior, solo que con el número de triángulos en vez de vértices.

La utilidad de generación de reportes(ver en sección 5.1.2.2) permite elegir cuáles de estas métricas se desea visualizar y comparar.

4.5. Concepto de «mejor algoritmo»

Decidir si un algoritmo es mejor que otro(s) es dependiente de la necesidad, y no es aplicable como una medida global. En este trabajo se pueden identificar 3 instancias comparativas:

1. Dada una malla, se busca establecer qué algoritmo puede producir una malla resultante con el mayor ángulo interior mínimo. Es decir, se busca comprobar hasta qué angulo exigido es capaz de responder dicho algoritmo.

2. Dada una malla y un ángulo de exigencia de calidad, si el algoritmo efectivamente mejoró la triangulación de acuerdo a dicho criterio, se busca establecer qué algoritmo llegó al resultado en menos pasos, o equivalentemente, insertando el menor número de triángulos posibles.

3. Dada una malla y un ángulo de exigencia de calidad, se puede elegir el «mejor» a partir de la comparación de sus tiempos (demoras) de procesamiento.

5. Implementación

La aplicación desarrollada es el cimiento de este trabajo, la cual fue diseñada según las siguientes guías sobre:

- Compatibilidad: Soporte multiplataforma.
- Eficiencia: Tanto en tiempo de procesamiento, como de uso de memoria
- Usabilidad: Que permita obtener resultados de manera práctica e intuitiva.

El producto del trabajo de desarrollo de esta memoria, es un conjunto de herramientas, divididas en distintos «módulos» como archivos ejecutables o bibliotecas, con capacidades de ejecución por línea de comando e interfaz gráfica. Este conjunto define la aplicación principal actual «BatchRefinement». Cabe mencionar que finalmente existen varios lenguajes de programación que forman parte de la versión actual. (Ver sección 5.2).

5.1. Descripción de alto nivel

5.1.1. Trabajo realizado y descripción de la aplicación

Explicado de manera breve, del trabajo relacionado a las memorias anteriores se extrajo el código relevante e implementó una biblioteca estática la cual es la base de la nueva aplicación «BatchRefinement». Esta cuenta con una interfaz gráfica y permite refinamientos de diversos tipos de formatos de archivos de triangulaciones, produciendo finalmente Reportes Visuales de Resultados de experimentos, en formato «html», a través de la ventana «ReportCreator». Estos incluyen imágenes de los estados, previos y posteriores al refinamiento, de las triangulaciones, tablas y gráficos, junto a las configuraciones escogidas para cada experimento. Pueden ser abiertos con cualquier explorador web, y son interactivos en el sentido de que las tablas poseen vínculos a paginas con detalles de cada uno de los experimentos realizados.

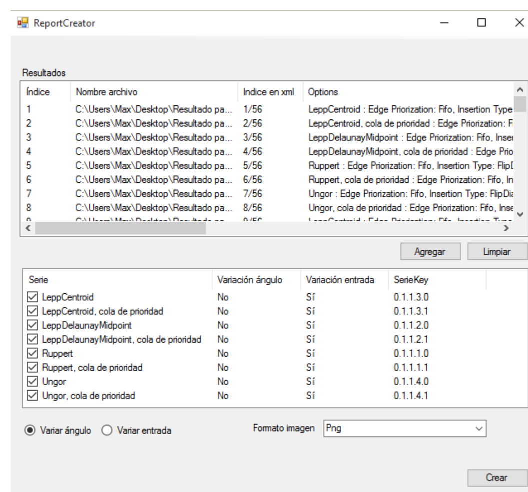


Figura 5.1: Interfaz de generación de reportes html, «ReportCreator».

5.1.2. Entradas y salidas

Para que ocurra un proceso de refinamiento, es necesario entregar a la aplicación archivos de triangulaciones. El producto, o resultado de esto depende de lo que desea obtener el usuario.

Originalmente, en las aplicaciones anteriores, existía un sólo formato de entrada. En la aplicación final, existen 2 módulos o programas ejecutables que permiten realizar refinamientos, pero presentan algunas diferencias de usabilidad. Ambos tienen definidos ciertos tipos de entradas y salidas, que se detallan como indican las figuras.

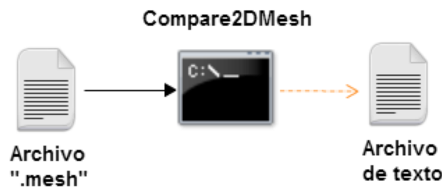


Figura 5.2: Diagrama simple y de alto nivel del flujo de datos de Compare2DMesh.

Rememorando Compare2DMesh, la aplicación de línea de comando, al igual que su interfaz gráfica opcional, recibía una triangulación «.mesh» y escribía los resultados utilizando la salida estándar del programa. Entonces, la forma de obtener los datos era a través de la redirección de dicha salida, volcando lo producido a algún archivo de texto. En la nueva aplicación de línea de comando «C2Mcmd», desarrollada en este trabajo de memoria de título, se extendió el número de formatos de entrada soportado, y en cuanto a las salidas se quiso mantener algo similar, sin embargo, ahora la salida es un archivo «xml» que contiene los resultados de manera estructurada.

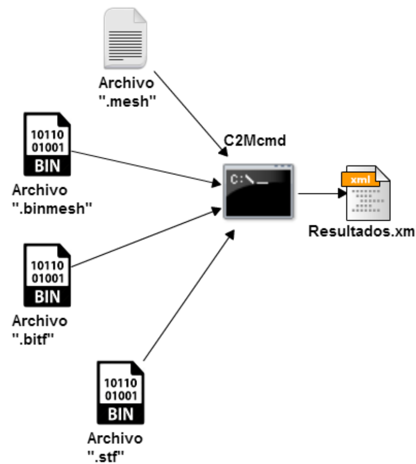


Figura 5.3: Diagrama simple y de alto nivel del flujo de datos de C2Mcmd.

El formato «xml» es básicamente soportado de manera universal. Existen numerosas aplicaciones que permiten visualizarlos de manera más adecuada que un simple editor de textos. Adicionalmente, para los desarrolladores, también existe un gran numero de herramientas

que permiten trabajar con éstos. Estas fueron las principales razones de la elección de dicho formato para las salidas.

Una de las nuevas aplicaciones implementadas «BatchRefinement», la cual posee interfaz gráfica, es capaz de realizar los mismo que «C2Mcmd», es decir, de producir un xml en base a las mismas entradas.

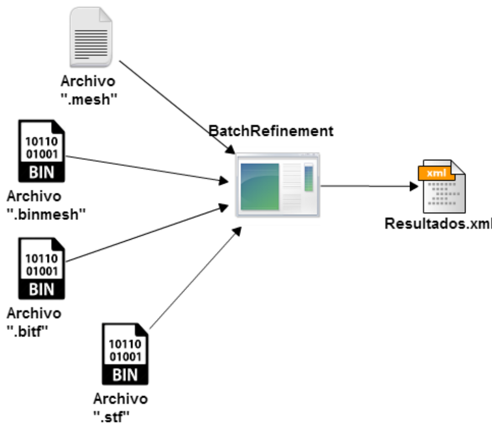


Figura 5.4: Diagrama simple y de alto nivel del flujo de datos de BatchRefinement para producción «xml».

Adicionalmente, ese mismo «xml» puede ser utilizado para la generación de Reportes de Visualización de Resultados.

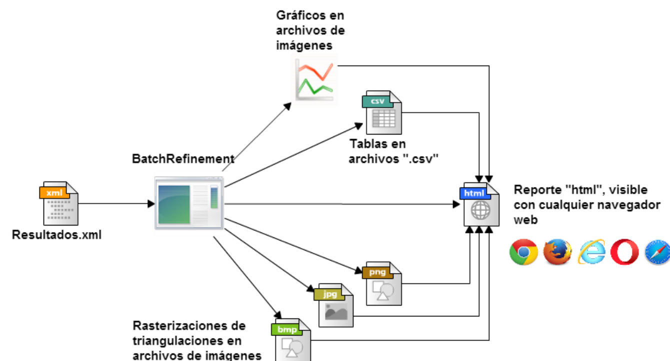


Figura 5.5: Diagrama simple y de alto nivel del flujo de datos de BatchRefinement para producción de Reportes «html» de Visualización de Resultados.

BatchRefinement posee las funcionalidades de producir un «xml» y luego reportes,cada uno en pasos separados, o adicionalmente producir reportes directamente desde los archivos de triangulaciones de entradas.

5.1.2.1. Descripción de parámetros iniciales de refinamiento Para iniciar un refinamiento, es necesario establecer qué tipo de algoritmo se quiere ejecutar, y los archivos de

triangulaciones sobre los cuales se trabajará. En la sección 4.3 se detallan las opciones para generar cualquier tipo de algoritmo, las cuales son:

- Priorización de triángulos.
- Priorización de aristas «encroached».
- Ángulo mínimo exigido.
- Elección de Steiner Point.
- Tipo de inserción.
- Preproceso de corrección de aristas «encroached».

Como medida de seguridad, adicionalmente se añadieron 2 nuevos parámetros opcionales para detectar casos de divergencia.

Timeout Es el tiempo límite, en segundos, en el cual se debe completar un algoritmo de refinamiento. Si la demora sobrepasa dicha espera, entonces el refinamiento se detiene, y en los resultados se indica que hubo un fallo debido a que el «timeout» fue alcanzado.

Índice de divergencia Es cociente entre el número de vértices actuales dividido por el número de vértices iniciales. Indica el crecimiento de la triangulación respecto a la original. Experimentalmente se observa que en los resultados exitosos de refinamientos se pueden encontrar dentro de un número determinado de vértices finales dado un número inicial. Incluso con exigencias cercanas a los ángulos límites, en los cuales la función de tiempo y/o vértices finales ya demuestra un ascenso divergente, los vértices obtenidos no tienden a sobrepasar cierto número de veces los iniciales.

Ambas medidas son complementarias y principalmente al ejecutar un gran número de refinamientos. El «timeout» puede ser ajustado de acuerdo al tiempo disponible para el experimento, o incluso según dicte la paciencia del usuario, y es particularmente útil en los esquemas de ejecución de múltiples refinamientos con ángulo exigido creciente. El índice de divergencia provee forma empírica de detección de algoritmos que no convergen, y adicionalmente pueden proteger la ejecución de posibles desbordes de memoria, lo cual es útil en ejecuciones de múltiples refinamientos evitando que la aplicación se caiga debido a memoria insuficiente y se pierdan los resultados.

Para los experimentos realizados, los valores de seguridad utilizados son los mostrados a continuación, y fueron determinados experimentalmente.

- Valor para «timeout» de experimentos: 60 segundos para uso general, 300 segundos para «pruebas de estrés y capacidad»(ángulos exigidos sobre 34°, o triangulaciones de 1.000.000 o más).
- Valor para «Índice de divergencia» de experimentos: 100.

Tabla de índice de vértices finales vs iniciales

Serie\Ángulo	5	10	15	20	25	28	29	30	31	32	33	34	35
LeppCentroid	1.0	1.3	1.8	2.2	2.8	3.1	3.3	3.5	3.6	4.0	4.3	4.9	5.9
LeppCentroid, cola prioridad	1.1	1.4	1.8	2.3	2.8	3.2	3.3	3.5	3.7	4.0	4.4	4.9	5.9
LeppDelaunayMidpoint	1.1	1.4	1.8	2.3	2.9	3.3	3.6	4.1					
LeppDelaunayMidpoint, cola prioridad	1.1	1.4	1.8	2.3	2.9	3.3	3.6	4.1					
Ruppert	1.1	1.2	1.4	1.8	2.5	3.3	3.7	4.4	5.5	7.9	14.7		
Ruppert, cola prioridad	1.1	1.2	1.4	1.8	2.4	3.2	3.6	4.0	4.8	6.2	9.2		
Ungor	1.1	1.2	1.3	1.6	2.1	2.6	2.9	3.3	3.9	4.8	7.7		
Ungor, cola prioridad	1.1	1.2	1.3	1.6	2.0	2.4	2.6	2.9	3.2	3.8	4.8	7.6	

Figura 5.6: Tabla de índice de vértices finales vs iniciales para una triangulación de 1000 puntos.

Es decir, los algoritmos fueron ejecutados durante a lo más 60 segundos cada uno, y mientras el número de vértices actuales no superara cien veces el número inicial. Cuando el refinamiento se completa dentro de los límites, es considerado como «exitoso». Adicionalmente, y como medida de seguridad extra para minimizar posibles errores humanos en la programación de los algoritmos, una vez que se completa cada uno, para que un resultado sea considerado como exitoso, se verifica que todos los ángulos interiores resultantes sean mayores al exigido.

5.1.2.2. Archivos y elementos de salida

Las aplicaciones actuales producen distintos archivos de salida al ejecutar algunas funcionalidades. No sólo en el caso de los algoritmos de refinamiento y generación de reportes relacionados, si no que también si el usuario hace uso de las utilidades, como la conversión de formato de archivo de triangulación, o en el caso de la «rasterización» a archivo de imagen. En este apartado, se lista los posibles elementos de salida de la aplicación.

Resultados de refinamientos estructurados «xml»

Para cuantificar el rendimiento de cada ejecución de los algoritmos se monitorea el estado de la triangulación y los cronómetros asociados durante cada ciclo de refinamiento. Al finalizar el proceso, todos los resultados son almacenados en un archivo de texto, en formato «xml». El guardado es directo, sin necesidad de tener que recurrir a la redirección de la salida «standard» de la aplicación a un archivo, como era el caso de las aplicaciones anteriores, favoreciendo la simplicidad de utilización.

```xml
<?xml version="1.0" encoding="utf-8"?>
<ArrayOfRefinementResult xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <RefinementResult>
    <OriginalTriangulationFile>
        C:\Max\b_100.mesh.bitf
    </OriginalTriangulationFile>
    <FinalSimpleTriangulationFile>
        b_100.mesh_final_0.1.1.1.0_20.stf
    </FinalSimpleTriangulationFile>
    <InitialSimpleTriangulationFile>
        b_100.mesh_initial.stf
    </InitialSimpleTriangulationFile>
    <Options>
      <DisplayName>Ruppert</DisplayName>
      <TimeoutInSeconds>120</TimeoutInSeconds>
      <Preprocess>FixEncroachedVertex</Preprocess>
      <InsideInsertion>FlipDiagonal</InsideInsertion>
      <SteinerPointMethod>Ruppert</SteinerPointMethod>
      <MinimalAngleTarget>20</MinimalAngleTarget>
      <EncroachedEdgePriorization>FirstInFirstOut</EncroachedEdgePriorization>
      <TrianglePriorization>FirstInFirstOut</TrianglePriorization>
    </Options>
    <success>true</success>
    <divergent>false</divergent>
    <timeout_reached>false</timeout_reached>
    <ending_vertices>235</ending_vertices>
    <starting_vertices>100</starting_vertices>
    <ending_restricted_edges>72</ending_restricted_edges>
    <starting_restricted_edges>12</starting_restricted_edges>
    <ending_triangles>396</ending_triangles>
    <starting_triangles>186</starting_triangles>
    <ending_smallest_angle>20.315083507223164</ending_smallest_angle>
    <starting_smallest_angle>0.602323728240875</starting_smallest_angle>
    <longest_lepp>0</longest_lepp>
    <refinement_iterations>91</refinement_iterations>
    <rest_deencroach_iterations>0</rest_deencroach_iterations>
    <initial_deencroach_iterations>44</initial_deencroach_iterations>
    <refinement_milliseconds>15.628199999999998</refinement_milliseconds>
    <rest_deencroach_milliseconds>0</rest_deencroach_milliseconds>
    <initial_deencroach_milliseconds>0</initial_deencroach_milliseconds>
  </RefinementResult>
  <RefinementResult>
   ...
  </RefinementResult>
  ...
</ArrayOfRefinementResult>
```

Figura 5.7: Muestra de archivo de resultados «xml» producido por la aplicación. Los «...» fueron incluidos sólo para efectos de resumir la ilustración, e indican la presencia de otros datos análogos a los ya descritos.

Estos archivos xml contienen información sobre cada uno de los experimentos realizados, los cuales adicionalmente contienen sólo el nombre de los archivos originales de entrada, así como los archivos de triangulación simple, «.stf» producidos durante el proceso de refinamiento, los cuales contienen, a grandes rasgos, el dibujo vectorial del estado inicial y final de la triangulación.

Archivos de imágenes

Parte del objetivo de la aplicación era atender necesidades de visualización. Se generan finalmente imágenes sobre gráficos y estados determinados de las triangulaciones (antes o despúes del refinamiento). El usuario, al momento de generar un reporte de visualización, tiene la op-

ción de elegir el formato de las imágenes producidas. Los formatos de archivo de imágenes soportados actualmente son:

1. BMP: Formato básico de imágenes, sin compresión, y por lo tanto, sin pérdida.
2. PNG: Formato de imagen con compresión sin pérdida.
3. JPG: Formato de imagen, de compresión con pérdida.Tiene la ventaja de poder utilizar poco espacio de almacenamiento.

Los reportes generados pueden ser bastante extensos debido a la gran cantidad de opciones y detalles que se pretenden ilustrar. Si las imágenes generadas son grandes, entonces el espacio final puede llegar a números que se escapan a lo práctico, cientos de MB, dependiendo del formato. Es por esto que la facilidad para elegir el formato de imagen tiene valor.

El formato predeterminado para las imágenes producidas en el programa es el «PNG», debido a que presenta un buen balance de calidad vs. tamaño.

Para ver consideraciones respecto a la producción de imágenes, ver el párrafo sobre la utilidad de «rasterización» de triangulaciones en 5.2.4.

Tablas en formato «csv»

De los archivo de resultados «xml» se puede obtener toda la información relevante a los experimentos. Sin embargo, no está directamente organizada ni lista para producir gráficos u otro tipo de visualizaciones. Durante el proceso de generación de reportes de visualización, la aplicación lee el archivo de resultados «xml», y para cada propiedad medible seleccionada por el usuario, genera una tabla de datos, a partir de las cuales se dibujan posteriormente los gráficos. Estas tablas son guardadas individualmente en archivos de texto de valores separados por comas, o «csv». Así, el usuario tiene acceso directo al conjunto de datos ya procesados.

Reportes detallados de visualización de resultados

La generación de reportes automática es una de las características claves de la nueva aplicación. Estos incluyen tablas y gráficos sobre cada una de las métricas(definidas en la sección 4.4) observables de los procesos de refinamiento. Dado un conjunto de resultados de ejecución, obtenidos de un archivo «xml de resultados», el usuario dispone de la posibilidad de elegir qué propiedad medida desea visualizar, a partir de lo cual la aplicación genera un «html», que incluye características de navegación, como hipervínculos desde cada resultado de las tablas a su correspondiente página de detalles.

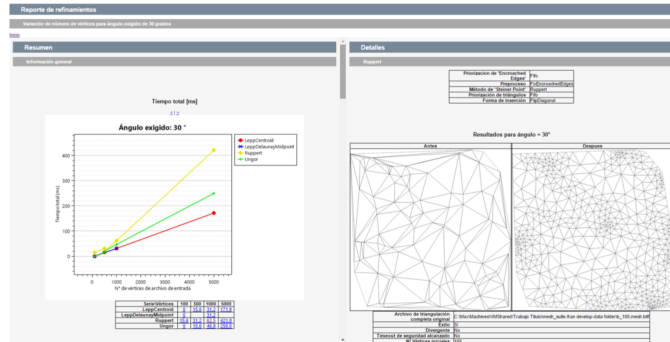


Figura 5.8: Miniatura de muestra de distribución en 2 columnas de los reportes «html».

Esquemas de resultados y reportes

La aplicación permite 2 esquemas de resultados y de reportes, de acuerdo a lo que se requiere observar.

1. **Reportes de ángulo variante**

   Consiste en obtener y representar datos considerando una lista de ángulos mínimos exigidos, aplicados a determinados refinamientos sobre una misma triangulación. En las tablas, las abscisas corresponden a los ángulos exigidos, mientras que las ordenadas corresponden a cada configuración de algoritmos que se requiere estudiar. (Ver Fig. 5.9)

Serie\Ángulo	5	10	15	20	25	28	29	30	31	32	33	34	35
LeppCentroid	1082	1378	1806	2276	2816	3180	3343	3502	3699	4017	4397	4909	5922
LeppCentroid, cola prioridad	1176	1439	1841	2301	2854	3221	3381	3546	3758	4062	4402	4952	5975
LeppDelaunayMidpoint	1181	1436	1885	2340	2972	3366	3623	4102					
LeppDelaunayMidpoint, cola prioridad	1181	1428	1877	2332	2975	3373	3613	4115					
Ruppert	1141	1247	1479	1846	2538	3354	3735	4416	5556	7973	14796		
Ruppert, cola prioridad	1141	1246	1477	1838	2483	3215	3615	4074	4894	6221	9283		
Ungor	1138	1222	1385	1655	2168	2656	2981	3377	3947	4869	7724		
Ungor, cola prioridad	1138	1219	1380	1637	2079	2467	2687	2945	3279	3835	4875	7605	

Figura 5.9: Ejemplo de tabla obtenida de resultados y reportes tipo **«ángulo variante»**.

2. **Reportes de número de vértices variante**

   En este caso los resultados son obtenidos ejecutando determinados refinamientos, aplicados sobre distintas triangulaciones (con distinto número de vértices), para un único ángulo de exigencia fijo. En las tablas, las abscisas corresponden al número de vértice de cada archivo de triangulación de la lista, y las ordenadas, al igual que en caso anterior, corresponden a cada configuración de algoritmos de estudio.

   También son llamados «reportes de entrada o triangulación variante», debido a que como las triangulaciones tienen un número fijo de vértices, es necesario variar los archivos(elegir otro archivo distinto) para obtener las distintas columnas asociadas al número de vértices inicial. (Ver Fig. 5.10)

Serie\Vértices	10	100	500	1000	5000	10000	50000
LeppCentroid	40	484	2231	4194	19116	37983	181015
LeppCentroid, cola de prioridad	40	484	2218	4197	19129	38049	181101
Ruppert	43	505	2995	5320	25233	50880	242148
Ruppert, cola de prioridad	35	513	2632	5020	23261	46707	220379
Ungor	43	451	2159	3959	18327	36118	168102
Ungor, cola de prioridad	35	400	1897	3552	16235	31872	148739

Figura 5.10: Ejemplo de tabla obtenida de resultados y reportes tipo **«número de vértices variante»**.

Los resultados que pueden ser mostrados en la tabla corresponden a las métricas definidas para comprar los algoritmos. (Definiciones en la sección 4.4)

5.2. Descripción temas técnicos

5.2.1. Marco de desarrollo

En este apartado se describirá el computador y los recursos con los que llevaron a cabo la implementación, las ejecuciones y las pruebas de la aplicación.

5.2.1.1. Computador de trabajo

Debido a que los tiempos de ejecución no sólo obedecen a temas de software, es importante tener presente las características del computador de desarrollo y pruebas, las cuales se listan a continuación:

Modelo: Notebook Samsung Series 5 550 NP550P5C-T01CL, modificado.

CPU: Intel Core i7 3610QM (2300 MHz - 3300 MHz)

RAM: 16 GB DDR3 (1333 MHz)

Almacenamiento: SSD Samsung 850 EVO 1TB.

5.2.1.2. Entorno de programación.

Se trabajó en Windows 8 Pro de 64bits. El IDE a elección fue Visual Studio 2013 Community, con el cual se implementaron los 3 proyectos que componen la aplicación final. Se utilizó el framework .Net (4.0) en partes de la solución.

Los lenguajes utilizados fueron Visual Basic .Net (o simplemente VB.Net) para la interfaz gráfica, C++11 - segunda iteración de C++, aprobado por la ISO en 2011, para el cual existe buen soporte del compilador de Visual C++ a la fecha - para los procesos de refinamiento, y C++/CLI como lenguaje intermediario entre el código «administrado» (managed) y el código nativo de C++11.

La interfaz gráfica permite la visualización de las triangulaciones. Para esto se utilizó Opengl 2.0 a través de los bindings proporcionados por la biblioteca de código abierto «OpenTK» para .Net.

Es relevante mencionar que a pesar de que el entorno descrito esté muy vinculado a un ambiente «Windows», en teoría puede ser portado a otros sistemas operativos de manera directa. Mono es una implementación de código abierto de .Net, y el código es compatible. Mono es compatible con los principales sistemas operativos: Linux, Windows, Mac OS X, iOS, entre otros. A su vez, OpenTK también es compatible con Mono y prácticamente las mismas plataformas.

Los resultados (salidas del programa) corresponden a archivos escritos en xml, y los reportes son carpetas con archivos «html» con figuras, tablas y gráficos en formato de imágenes «png».

Los gráficos son generados por una biblioteca para .Net, y no tiene más requerimientos.

Sumario de lenguajes de programación utilizados Se utilizaron diversos lenguajes de programación en el desarrollo de la aplicación. Detalles respecto al «por qué» y «en qué» se utilizó cada uno, pueden ser revisados en 5.2.6.

- **C++11**

  Antes conocido como C++0x, es el nombre de la segunda iteración más reciente del lenguaje de programación C++, reemplazando al C++03 y reemplazado por el C++14, aprobado por la ISO el día 12 de agosto de 2011.1 El nombre es derivado de la tradición de nombrar las versiones de lenguaje por la fecha de la publicación de la especificación.

- **C++/CLI**

  C++/CLI desarrollado por Microsoft y posteriormente estandarizado por ECMA (como ECMA-372), es un «superconjunto» o extensión del lenguaje de programación C++, y en simples palabras, es un lenguaje de primera clase diseñado para interoperar entre el código o aplicaciones de compilaciones C++ clásicas («nativas») con el Framework.Net.

- **HTML**

  Es el lenguaje de marcado para la elaboración de páginas web. La implementación utiliza HTML5 junto a CSS3 para la elaboración de los reportes de ejecución. CSS3, es un lenguaje usado para definir y crear la presentación de un documento estructurado escrito en HTML o XML.

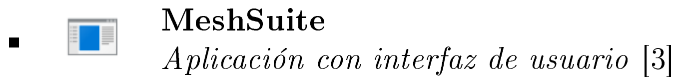
- **VisualBasic.Net**

  Es un lenguaje de programación de alto nivel, orientado a objetos, implementado por Microsoft sobre el framework .NET.

5.2.2. Estado inicial y evolución de funcionalidades generales de trabajos anteriores

A continuación se explica el proceso evolutivo del código que ha sido modificado en consecuentes temas de memoria. El objetivo es esclarecer las diferencias relevantes entre versiones

y el valor que aporta cada entrega respecto a las anteriores, de manera resumida. Los detalles de cada aplicación pueden ser revisados en sus respectivos trabajos.

- 
  MeshSuite
  Aplicación con interfaz de usuario [3]

  Archivo ejecutable con interfaz gráfica, desarrollado como parte del trabajo de memoria de Álvaro Faúndez. Implementa un entorno de ventanas basado en QT y OpenGL, y permite ejecutar refinamientos de triangulaciones en formato de archivos .mesh, eligiendo cada uno de los pasos del algoritmo. Con objetivo en la docencia, el programa puede mostrar paso a paso cada iteración, resaltando los elementos relevantes, y mostrando finalmente los resultados en pantalla.

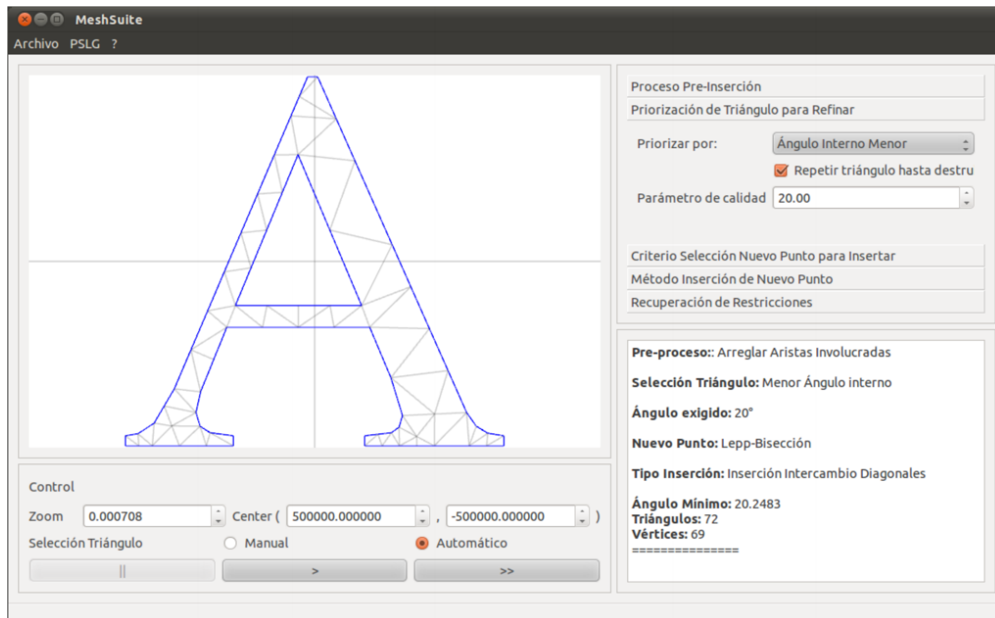
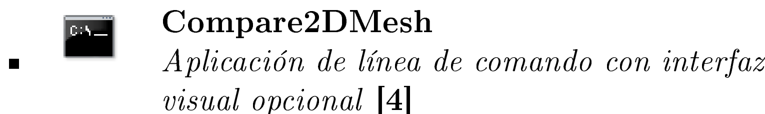


Figura 5.11: Interfaz gráfica de «MeshSuite».

- 
  Compare2DMesh
  Aplicación de línea de comando con interfaz visual opcional **[4]**

  A partir del código de MeshSuite, Francisca Gallardo desarrolla una solución de línea de comando, con interfaz gráfica opcional, para la ejecución de los algoritmos configurables del trabajo anterior. Mejorando muchos aspectos comprometidos por el enfoque didáctico, la aplicación final presenta optimizaciones y mejoras que permiten la ejecución de mallas geométricas más grandes, y con velocidades de ejecución mucho mayores. Al desligar el código de visualización del de los algoritmos, implementó una opción de ejecución de refinamiento sin interfaz gráfica, que mejora aun más los tiempos de procesamiento, y permite automatizar pruebas por medio de scripts «bash».

  La interfaz gráfica opcional de Compare2DMesh, es prácticamente idéntica a la de MeshSuite (Fig. 5.11).

La última aplicación presenta el estado inicial del trabajo actual. Desde lo cual se procuró mejorar y consolidar el código antiguo en una biblioteca estática, a partir de la cual se implementa adicionalmente una aplicación práctica, que asista de manera eficiente en la investigación de efectos sobre las modificaciones de algoritmos.

-
  C2M
  Trabajo correspondiente a memoria actual, biblioteca estática

  C2M es una biblioteca estática - requerida solamente en tiempo de compilación - desarrollada y nombrada a partir de Compare2DMesh, y posee las nuevas versiones de los algoritmos y estructuras de datos desarrolladas. Si bien Compare2DMesh posee características de rendimiento competitivas, aun deja un espacio disponible para posibles mejoras. No sólo en términos de utilización de recursos y velocidad, si no que también en cuando a compatibilidad, dependencias y requerimientos.

  Haciendo uso de la revisión 2011 de C++, también conocida como C++11, y considerando prácticas de manejo explícito de memoria entre otras cosas, se produjo finalmente código que mejora muchos aspectos al predecesor, incluyendo a la vez aspectos de compatibilidad. La ventaja que ofrece una biblioteca estática consiste en que puede ser incluida en otros proyectos, sin afectar decisiones internas de éste último, y mantiene siempre el rendimiento. En el caso de este trabajo, se adjuntó posteriormente a distintos proyectos, incluido uno con interfaz gráfica, que hace uso del framework .NET. La descripción de cada uno de los módulos que componen la aplicación resultante se encuentra en la subsección 5.2.6.

5.2.3. Formatos de archivos de entrada

La aplicación recibe como entrada archivos de triangulaciones. El formato original, utilizado por MeshSuite y Compare2DMesh, consiste en un archivo de texto, con la descripción de cada uno de los vértices y los triángulos que componen la triangulación. En la nueva iteración, se elaboraron 2 nuevos formatos de archivos binarios. En esta sección, se describen los formatos actuales, sus características, y finalmente se discute extensibilidad.

5.2.3.1. Mejoras sobre inicialización desde archivos de entradas.

En este apartado, se explicarán e ilustrarán los principales problemas encontrados respecto al código antiguo relacionado específicamente con la lectura de triangulaciones desde archivos «.mesh», y se explicará la forma en que se solventó.

Dependencia de formato de archivo de triangulación y clase. Las triangulaciones son manejadas en la aplicación a partir de una estructura de datos que las representa. Para esto, un archivo de malla (de algún formato compatible, originalmente sólo «.mesh») es leído, y luego se instancia un objeto de la clase «Triangulation» con información de vecindad, y con todas sus propiedades.

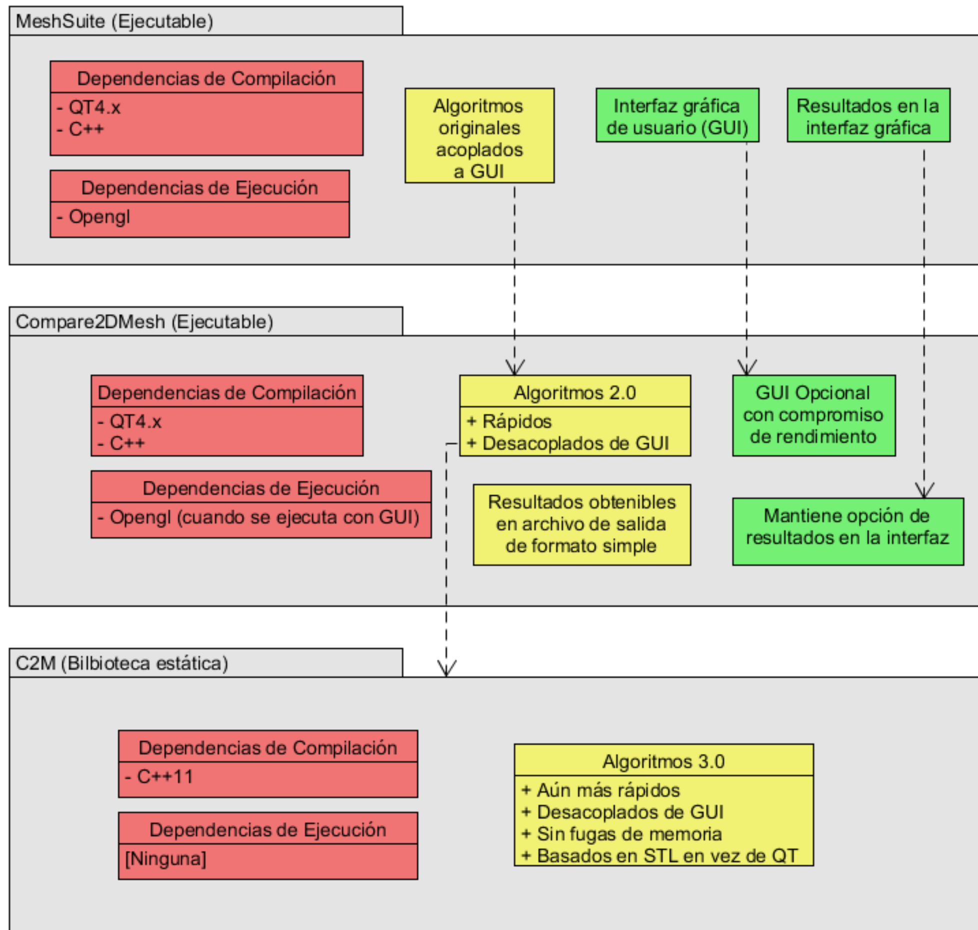


Figura 5.12: Evolución del «código principal».

En las aplicaciones anteriores, donde el objeto de la triangulación era llamado «Mesh» en vez del nombre actual «Triangulation», el formato del archivo estaba muy interiorizado en la inicialización de la triangulación. El objeto «Mesh» se inicializaba recibiendo un string con la ruta del archivo .mesh, y por cada línea leída, la triangulación se iba actualizando.

```
class Mesh{
    //único constructor
        Mesh(string filename){
        ... //código de inicialización de miembros de instancia
        F = abrir_archivo_para_lectura(filename);
        while( ( L = siguiente_linea_de_texto(F)) != NULL) {
            ... //parsear línea L y agregar elemento a triangulación
        }
        cerrar(F);
        ... //calcular información de vecindad de la triangulacion
    }
    ... //definiciones de otros miembros de la clase Mesh
}
```

Esto produce una dependencia de formato. Para extender ingenuamente, de manera directa, y permitir aceptar más formatos de entrada, se tendría que modificar el método de inicialización para que ahora distinguiera otro nuevo formato, e inicializara la triangulación acordemente. Con más de un formato disponible, la clase «Triangulation» se transforma en un conjunto de líneas de código dedicadas a «streams» de archivos, conversión de datos de entradas de acuerdo a la semántica de lo leído, y así, enredando y perdiendo el propósito de lo que se esperaría de una buena clase de objeto.

```
class Mesh{
    //único constructor
    Mesh(string filename){
        ... //código de inicialización de miembros de instancia
        F = abrir_archivo_para_lectura(filename);
        if ( Tipo de archivo de entrada(F) == Tipo1){ //forma original
            while( ( L = siguiente linea de texto(F)) != NULL) {
                ... //parsear línea L y agregar elemento a triangulación
            }
        }
        else if ( Tipo de archivo de entrada(F) == Tipo2){
            ... //leer el archivo de alguna otra forma y obtener elementos
        }
        else ... //resto de las opciones soportadas
        cerrar(F);
        ... //calcular información de vecindad de la triangulacion
    }
    ... //definiciones de otros miembros de la clase Mesh
}
```

La solución natural es desligar el código de lectura del archivo de ésta. Para esto se creó un objeto simple a partir del cual se puede inicializar una triangulación. Si se implementara la lectura de un nuevo formato de archivo, bastaría llenar la estructura e inicializar la triangulación a partir de ésta. Esta estructura «básica», llama «MeshFileData», es equivalente a tener un objeto con exactamente los mismos datos que provee un archivo «.mesh».

```
class MeshFileData{
    public:
    MeshFileData(); //Constructor
    void initialize(int n_vertices, int n_faces, int n_restrictions); //Inicializador
    int NVertices;
    int NFaces;
    int NRestrictions;
    //arreglo de vertices (un vertice es 2 doubles)
    std::vector<std::vector<double>> Vertices;
    //arreglo de triangulos (un triangulo es 3 integers)
    std::vector<std::vector<int>> Faces;
    //arreglo de restricciones (una restriccion es 2 integers)
    std::vector<std::vector<int>> Restrictions;
}
```

Gracias a esto, ahora es más simple extender el soporte de formatos siempre y cuando el programador pueda generar un objeto «MeshFileData» desde el archivo deseado.

Adicionalmente. Para esto, se provee una manera de inicializar el objeto «Triangulation» sin datos, a partir de la cual el desarrollador tiene la libertad, y responsabilidad, de acceder directamente a los miembros de instancia de la triangulación. Es decir, puede establecer explícitamente cada uno de los vértices, caras, información de vecindad, etc, desde código ajeno la clase. Esta funcionalidad permite la extensión del número de formatos de archivos de triangulación soportados que cuentan con información de vecindad.

Aprovechando el nuevo acceso a los componentes, se implementó una utilidad que permite inicializar la triangulación e información de vecindad de manera directa, es decir, a partir de un archivo con todos los cálculos computados y almacenados. Esto es útil en los casos donde se cuenta con archivos que contienen dicha información, como el último formato «bitf» descrito en el documento, y ahorra una cantidad considerable de tiempo de inicialización.

Finalmente, se optó por renombrar la clase «Mesh» como «Triangulation», para hacer más explícito que su elemento básico son triángulos, quedando definitivamente con un constructor sin parámetros, y dos inicializadores que cubren la mayoría de las necesidades.

```
class Triangulation{
    Triangulation();//constructor sin argumentos.
    void initialize_from_meshfiledata(MeshFileData& mesh_file_data);
    void initialize_empty();
    ... //definiciones de otros miembros de la clase Triangulation
}
```

5.2.3.2. Formatos de archivos de triangulaciones de entrada soportados

Actualmente se cuenta con soporte para 3 tipos de formatos de archivos de triangulaciones de entrada. A continuación serán descritos, y se listarán ventajas y desventajas de cada uno.

Archivo de texto .mesh

Es el formato original. Corresponde a un archivo de texto, el cual describe en cada línea un vértice, un triángulo o una arista restringida. Se utiliza el carácter «.» (punto) como separador de miles.

Una línea que comienza con el carácter «v», corresponde a un vértice, y contiene los números de sus coordenadas x, y, z, separados por un espacio.

Las líneas que comienzan con el carácter «f» (de «face»), corresponde a un triángulo, y contiene los índices de los vértices que los conforman, separados por un espacio.

Finalmente, las que comienzan con «r» describen una arista restringida, y contienen los índices de los vértices que lo definen, separados por un espacio.

(*) Como nota general, todos los índices, tanto de vértices, como triángulos, y restricciones de la clase de triangulación, se cuentan a partir del número 1.

Ejemplo de archivo:

```
v −285.000000 −285.000000 0.000000
v 285.000000 −285.000000 0.000000
v 285.000000 285.000000 0.000000
(... hasta listar todos los vertices)
f 43 42 10
f 44 46 48
f 43 41 12
(... hasta listar todos los triángulos)
r 1 2
r 4 3
(... hasta listar todas las restricciones)
```

Ventajas:

- Es fácilmente legible e inspeccionable a simple vista con cualquier editor de texto.
- La precisión de los números es arbitraria. (Aunque la aplicación que se implementó es de precisión fija)

Desventajas:

- La lectura de archivos de texto y posterior conversión a estructuras en memoria (objetos) es relativamente lenta.
- Lectura programática es poco directa. (Involucra cierto análisis y proceso sobre cada línea leída para saber qué hacer con ella)
- Se utiliza espacio en disco innecesario considerando la precisión manejada por la aplicación.
- No almacena ninguna información de vecindad, y en consecuencia requiere de cálculos adicionales para recuperar esta información.

Archivo binario «.binmesh»

Es la versión binaria del archivo .mesh. Almacena la misma información, pero estructurada con el propósito de una lectura más fluida por un «stream» de datos.

La descripción exacta del formato binario se encuentra en el anexo.

Ventajas:

- Tamaño fijo respecto a la precisión soportada por la aplicación.
- Gran precisión gracias al uso de números flotante de 64 bits (doubles).
- De fácil lectura programática. Una vez definido el formato, se sabe perfectamente qué se hace con cada elemento leído.
- Menor tiempo de lectura respecto al formato en texto.

Desventajas:

- Ilegible a simple vista, y requiere de un visor hexadecimal para inspeccionarlo.
- No almacena ninguna información de vecindad, y en consecuencia requiere de cálculos adicionales para recuperar esta información.

Archivo binario «.bitf»

Es la última versión de archivo binario desarrollada pensando exclusivamente en mejorar los tiempos de carga. Su extensión se basa en el nombre en inglés «Binary Initialized Triangulation File». Es recomendado utilizar este formato en vez de los anteriores, ya que permite una inicialización más rápida del objeto de la triangulación.

En los formatos anteriores, al no contar con información de vecindad, cada vez que se leía un archivo de triangulación se incurría en un costo al recalcular la información de vecindad. Los triángulos contienen referencias a sus vecinos, las cuales son necesarias para el proceso de refinamiento. Al guardar esta información dentro del archivo, ésta puede ser recuperada después de manera inmediata, acortando aun más el tiempo de carga.

De manera similar al formato binario anterior, la descripción exacta de éste se encuentra en el anexo.

Ventajas y desventajas: Mantiene los puntos fuertes del formato binmesh, y adicionalmente soluciona problema sobre la vecindad, siendo aun más rápido.

5.2.3.3. Otros formatos y extensibilidad.

No hay soporte directo para otros formatos de archivo. Sin embargo, gracias a haber separado la inicialización de la triangulación, ahora es mucho más fácil implementar nuevas opciones que acepte la aplicación. Solo es necesario que el implementador tenga conocimiento de la estructura del archivo deseado, que la lea y llene la estructura. No es requerimiento conocer internamente el proceso de la clase Triangle, ni como calcular la información de vecindad.

Independientemente del archivo de entrada, se recomienda convertir dicho formato a «bitf», que es con el que mejor se comporta la aplicación.

5.2.4. Utilidades y validaciones

Conversión de formatos de archivos de entrada Como se describió previamente, existen distintos formatos de archivo de entrada. Cada archivo describe una triangulación, sin embargo, hay formatos más eficientes que otros. Lo ideal para la ejecución más expedita de los procesos de refinamiento es que los archivos se encuentren en el formato «bitf». Para esto, se creó una utilidad de conversión de archivos, la cual puede leer de cualquiera de los formatos soportados, y escribir un nuevo archivo bitf a partir de este.

Validación de triangulaciones Debido a que los archivos de triangulaciones provienen de diversas fuentes, no existe garantía de que siempre se tendrá una triangulación Delaunay, o siquiera válida. Un requisito para la correcta ejecución de los algoritmos es contar con una triangulación apropiada, que cumpla con ciertas condiciones. Para esto, se desarrollaron pruebas que indican si las triangulaciones las cumplen o no. Estas comprobaciones también son útiles al momento de implementar algoritmos, gracias a que es posible comprobar después

de cada iteración de que los invariantes persisten. Se realizan desde comprobaciones básicas, por ejemplo, verificando que los punteros no sean NULL, hasta comprobaciones de vecindad y de propiedades Delaunay.

Las pruebas implementadas, junto a su complejidad, fueron:

1. `static bool non_null_vertices_test(Triangulation& triangulation):` Recorre todos los punteros a vértices de la triangulación, verificando que no hayan valores NULL. $O(n)$

2. `static bool non_null_triangles_test(Triangulation& triangulation):` Recorre todos los punteros a triángulos de la triangulación, verificando que no hayan valores NULL. $O(n)$

3. `static bool non_null_triangle_vertices_test(Triangulation& triangulation):` Recorre todos los punteros a vértices de cada triángulo de la triangulación, verificando que no hayan valores NULL. $O(n)$

4. `static bool vertex_unicity_test(Triangulation& triangulation):` Recorre todos los vértices de la triangulación, y comprueba que no existan 2 o más con coordenadas coincidentes. $O(n^2)$

5. `static bool triangle_unicity_test(Triangulation& triangulation):` Recorre todos los triángulos de la triangulación, y comprueba que no existan 2 triángulos definidos con los mismos vértices. $O(n^2)$

6. `static bool vertices_in_triangles_test(Triangulation& triangulation):` Comprueba que cada uno de los vértices de la triangulación, pertenezca al menos a un triángulo. Es decir, que no hayan vértices aislados. $O(n^2)$

7. `static bool triangles_connection_test(Triangulation & triangulation):` Comprueba que para cada par de triángulos, existe un camino de triángulos que los une. $O(n^3)$

8. `static bool triangles_border_intersection_test(Triangulation & triangulation):` Comprueba que la única intersección posible entre cada par de aristas, sea inexistente, o un solo vértice. $O(n^2)$

9. `static bool triangles_orientation(Triangulation & triangulation):` Recorre todos los triángulos, y comprueba que esté bien orientado (en sentido antihorario).$O(n)$

10. `static bool minimal_angle_test(Triangulation & triangulation, double min_angle):` Recorre todos los triángulos, y comprueba que su menor ángulo interno sea mayor al especificado. Esta comprobación se usa para saber si efectivamente un algoritmo de refinamiento produce el resultado buscado. $O(n)$

11. `static bool weak_delaunay_test(Triangulation& triangulation):` Recorre todos los triángulos, y comprueba que todos pasen la prueba del circuncírculo, considerando sólo los vértices de los triángulos próximos. $O(n)$

Pueden existir redundancias entre validaciones, lo cual sólo perjudica el tiempo de ejecución. Si la función retorna el valor «true», la triangulación pasa la prueba.

Las comprobaciones están implementadas de manera directa, sin muchas consideraciones de eficiencia para mantenerlas lo más explícitamente posible, y así disminuir la probabilidad de cometer equivocaciones que finalmente retrasan el tiempo de desarrollo. Debido a esto, se incurre en un costo muy significativo al ejecutarlas. Recordando que estas son sólo herramientas de apoyo, una vez que se comprueba que los algoritmos se comportaban de acuerdo a lo esperado, conservando el invariante, las comprobaciones son desactivadas. Actualmente no influyen en la ejecución, pero están definidas en el código y pueden ser utilizadas si se necesitaran.

«Rasterización» de triangulaciones Al generar informes se incluye una etapa de «rasterización» de triangulaciones para así producir una imagen, de alguno de los formatos soportados, que forma parte del html de salida. Adicionalmente, la aplicación cuenta con una utilidad independiente, que puede generar un archivo de imagen en formato a elección a partir de un archivo «stf». Este último es el formato que guarda configuraciones estáticas y sin información de vecindad de una triangulación, cuyo propósito es básicamente para ser dibujado y leído o escrito de manera fácil.

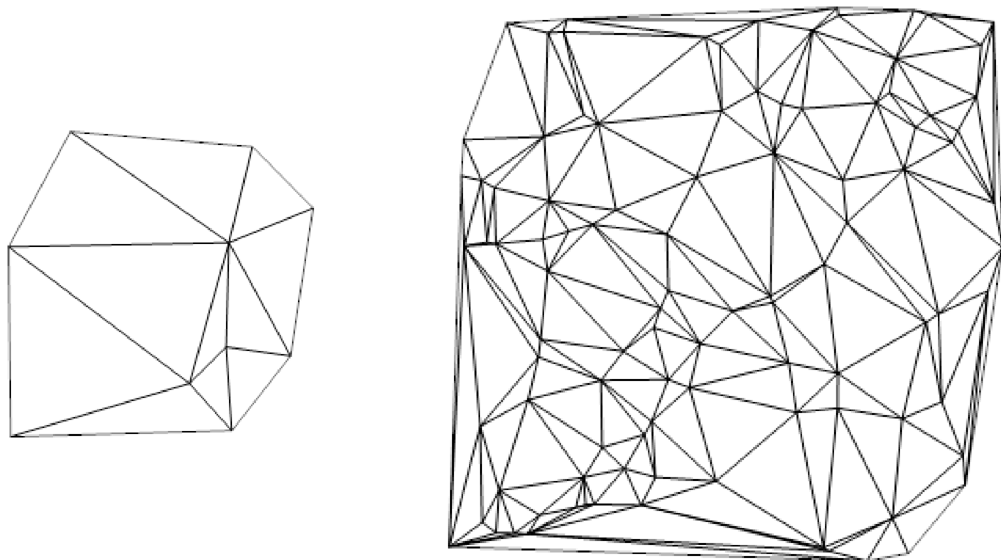
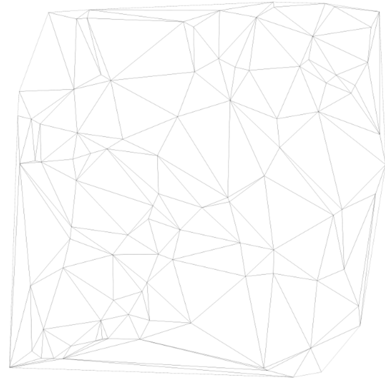


Imagen pequeña de triangulación de pocos vértices

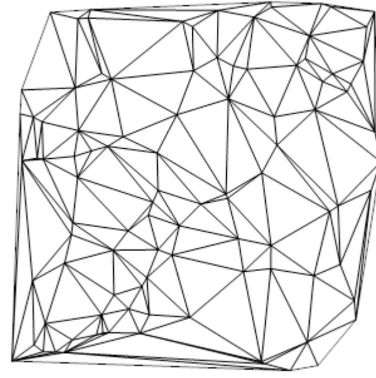
Imagen más grande de triangulación con más vértices

Figura 5.13: Muestras de distintos tamaños de las imágenes que produce la aplicación al rasterizar.

Debido a que actualmente no es posible obtener de manera automática una medida de calidad visual del dibujo de una triangulación, las dimensiones de la imagen producida deben ser especificadas a mano por el usuario. La regla general es, a grandes rasgos, que «a mayor número de vértices a dibujar, mayor deben ser las dimensiones de la imagen». Si el tamaño escogido es inadecuado, es posible conseguir imágenes difusas, con muy poco uso práctico.



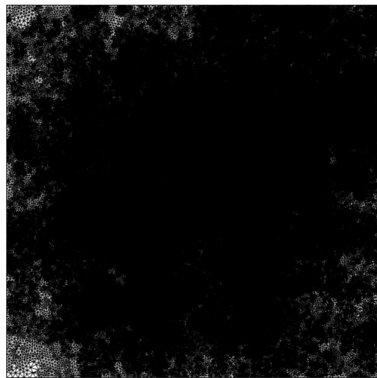
Dimensiones exageradamente grandes produce líneas poco claras



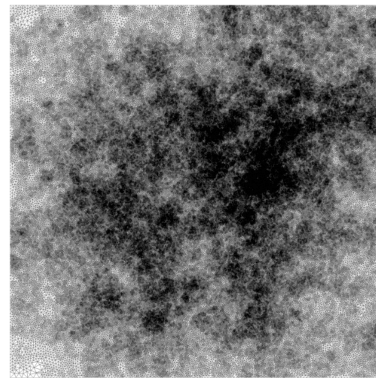
Dimensiones de tamaño apropiado

Figura 5.14: Efecto de dimensiones de la imagen resultante del proceso de «rasterización».

De manera similar, la opacidad de los trazos puede ser ajustada para poder observar detalles que de otra forma son indistinguibles, como muestra la figura 5.15.



Opacidad al 100 %



Opacidad al 20 %

Figura 5.15: Efecto del ajuste de opacidad en rasterizaciones de un mismo tamaño.

El visualizador posee la funcionalidad de poder ajustar la opacidad y zoom de la imagen mostrada, todo en tiempo real, y esta utilidad puede ser usada con el propósito de previsualizar y probar combinaciones de parámetros para producir un archivo de imagen que cumpla con las expectativas del usuario.

5.2.5. Problemas heredados y soluciones

Las aplicaciones exitosas son implementadas a partir de algún diseño que intenta resolver un problema particular. Incluso en los casos más informales, donde el mismo diseño o requerimiento va cambiando a medida que avanza el desarrollo, se puede observar un eje central sobre las necesidades que cubre el programa. Cuando es necesario variar un poco el objetivo, a veces los desarrolladores se encuentran en posiciones donde el código previo no les favorece para producir el cambio. Es por ello, que a pesar de que los trabajos de memoria cumplieron

sus cometidos, al retomar dicho código y darle un rumbo incluso aledaño, como este caso, se produjeron algunos contratiempos que debieron ser resueltos de manera de proporcionar finalmente un producto satisfactorio según los nuevos objetivos. A continuación, se describen los principales problemas que emergieron durante el proceso de desarrollo.

Refactorización

Legibilidad de código

Al momento de trabajar a partir de código ajeno, es particularmente notorio la legibilidad de éste. Estandarización de código y nombramientos descriptivos ayudan al entendimiento e incluso pueden reducir la incidencia de algunos errores. Existen incluso trabajos y teorías sobre mejores prácticas al respecto. Sin profundizar mucho en el tema, se tomó la decisión de mejorar al menos la parte de nombramientos respecto de las aplicaciones anteriores. Esto incluye nombres de clases, de miembros y variables locales a lo largo de todo el código del proyecto anterior.

Algunos de los cambios más relevantes en nombramientos representativos de los conceptos involucrados:

- Clase «Mesh», ahora llamada «Triangulation»: Es la clase que define la estructura de datos principal con los datos de la triangulación sobre la cual se ejecutan los refinamientos.
- «NewPointMethod», ahora «SteinerPoint»: Enumeración y término utilizado para describir la forma de elección de las coordenadas de los nuevos puntos a insertar.

Adicionalmente, hubo numerosas otras instancias de renombramiento de miembros de instancia, para asistir al entendimiento del desarrollador. Para esto, se reemplazaron todos los nombres tipo «abreviados» por expresiones breves más apropiadas y explícitas. Esto incluye miembros de instancia que seguían la línea de: «p» para referirse a puntos, «v» a vértices, «nv» para número de vértices, y variables de métodos denominadas por el estilo.

Simplificación de jerarquía de polimorfismos

Las aplicaciones previas, obedeciendo a la necesidad de cumplir con el objetivo de fácil extensibilidad, siguen patrones de diseño que permiten agregar de manera simple nuevas capacidades: nuevos algoritmos de selección de puntos, nuevos métodos de inserción. Esta flexibilidad venía al costo de una relativamente complicada jerarquía de clases, que dificulta la mantenibilidad. Particularmente, para agregar un nuevo método de selección de «Steiner Points», bastaba con crear una nueva clase que hereda de «NewPointMethod», y adicionalmente agregar una nueva enumeración que identifique al nuevo elemento.

Los problemas surgen cuando se requiere modificar el flujo de la aplicación o la base misma de una jerarquía. Siguiendo el ejemplo, si se quisiera modificar la clase base NewPointMethod, puede ocurrir que sea necesario cambiar cada una de las subclases hijas. Esto ya crea una dificultad de mantenimiento. Suponiendo

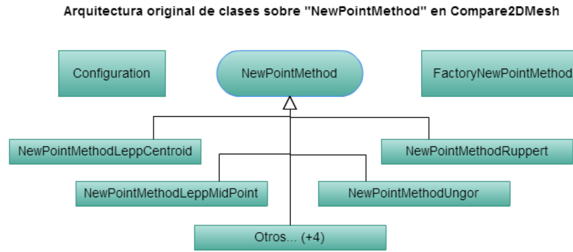


Figura 5.16: Diagrama de jerarquía antiguo de clases involucradas en el proceso de elección del nuevo «Steiner Point» a insertar.

que existen 4 formas de seleccionar un «Steiner Point», y dado que cada clase está definida por 2 archivos, finalmente, se deben revisar y corregir 8 archivos adicionales a los que definen a la clase padre.

El patrón de diseño «FactoryMethod» fue bastante utilizado en las aplicaciones anteriores, gracias a que es fácil de entender, y oculta detalles de la creación de objetos, entregando sólo una «interfaz» que contiene lo relevante.

Una clase «Factory» es utilizada para crear objetos del tipo «NewPointMethod». Al modificar «NewPointMethod», fue necesario cambiar también la clase Factory asociada. Finalmente es inevitable tener que revisitar mucho código, lo que es usualmente poco deseable.

Este problema de mantenimiento jerárquico se repite varias veces en ambas aplicaciones antiguas.

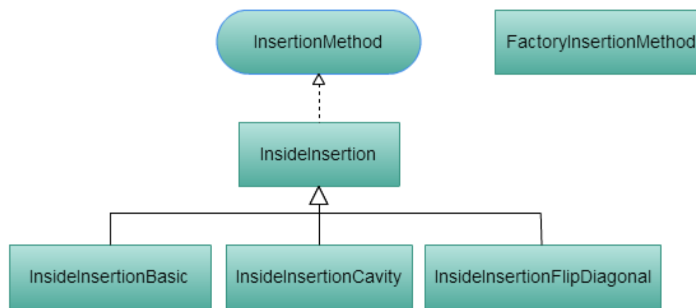
Sacando provecho de que C++11 provee soporte para funciones anónimas, también conocidas como funciones «lambdas», se pudo simplificar las jerarquías de este tipo. En efecto, se observó que los hijos finales de estas jerarquías solamente definían un único método. La solución final fue dejar sólo una clase que cumple el rol de «Factory», por jerarquía, la cual ahora produce funciones lambda que respetan una determinada firma, en vez de trabajar en subclases.

Nueva arquitectura de clases sobre "NewPointMethod" en C2M

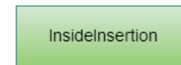
Figura 5.17: Diagrama de la nueva jerarquía simplificada de clases involucradas en el proceso de elección del nuevo «Steiner Point» a insertar. Finalmente, se redujo a la utilización de sólo una.

Otras jerarquías fueron completamente removidas siguiendo acciones similares. Esto incluye, «TriangleSelection», la cual era utilizada para definir las funciones de comparación entre triángulos que definen los posibles ordenamientos en las colas de prioridad, y «PreProcess», que de manera análoga a «NewPointMethod», define la manera de elegir las coordenadas de los nuevos «SteinerPoints» a insertar en el caso de encontrar aristas «encroached», «InsideInsertion» con la cual se

especificaba la forma de inserciones (por cavidad, intercambio de diagonales, etc), y adicionalmente, la definición de colas.



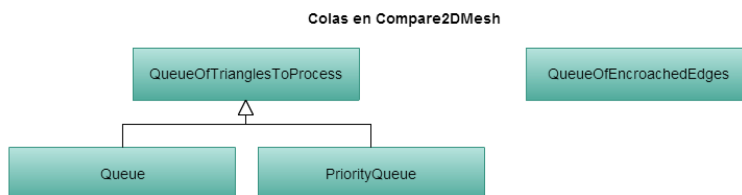
Arquitectura anterior en Compare2DMesh sobre «InsideInsertion»



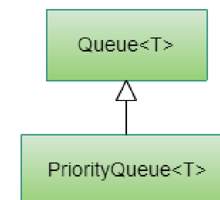
Nueva arquitectura simplificada en C2M sobre «InsideInsertion»

Figura 5.18: Simplificación de jerarquía de clases relacionadas a «InsideInsertion».

En el caso de las colas, la simplificación se logró unificando las distintas clases (tanto para triángulos, o para aristas «encroached») por medio de la utilización de los «templates» de STL. Lo que permitió producir finalmente 2 tipos de estructuras básicas de colas, que sirven para todas las necesidades de la aplicación.



Arquitectura anterior en Compare2DMesh sobre colas



Nueva arquitectura simplificada en C2M sobre colas

Figura 5.19: Simplificación de jerarquía de clases de colas.

El beneficio de no tener clases tan numerosas - con números por sobre lo necesario - consiste en que se mejora significativamente la mantenibilidad del código.

Desvinculamiento de QT

Las aplicaciones anteriores, MeshSuite y Compare2DMesh, fueron implementadas utilizando C++ utilizando las bibliotecas de QT. No sólo la interfaz gráfica dependía completamente de ésta, si no que también muchas líneas del código. Parte importante del esfuerzo de desarrollo consistió en eliminar por completo estas dependencias. La

idea era crear una biblioteca estática que tuviera el menor número de requerimientos posibles, y que utilizara implementaciones conocidamente «optimizadas». La decisión final fue re-implementar todo el código, nuevamente, en «C++11» quitando todo lo relacionado a la interfaz gráfica, y «traduciendo» lo relacionado a estructuras y bloques propios de QT, por sus versiones equivalentes C++11 - STL. La estructura general se mantuvo bastante similar, sin embargo, fue necesario hacer un trabajo «línea a línea», debido a que el código no es directamente portable.

Por ejemplo, en el caso de las estructura u objetos, QHash, QHashIterator, QString, QVector, entre otros, debieron ser reemplazados por sus análogos STL, como «std::map<T1,T2»> std::map<T1,T2>::iterator, std::string, std::vector, correspondientemente y teniendo consideraciones respecto al nivel de referencia (si eran punteros, valores, u otra variedad), etc, y como ejemplo de bloques de código, se cambiaron los bloques «foreach» propios de QT, por sus «for» equivalentes de C++11.

«Namespaces» y conflicto de nombres

Al convertir el código basado en QT a su equivalente C++11 - STL, surgieron algunos conflictos de nombres y «namespaces», los cuales prevenían que el programa modificado compilara. Enumeraciones definidas en el «scope» global formaban la principal fuente de problemas. Se optó por incluir cada elemento antiguo dentro de algún namespace, lo que finalmente tuvo un efecto organizacional positivo, a parte de solucionar los conflictos.

Eficiencia y optimizaciones

«Compare2DMesh» presentaba mejoras respecto a su predecesor, mejorando el código e implementando rutinas más optimizadas. Uno de los casos más influyentes fue la inclusión de la estrategia para «encontrar el triángulo que contiene un punto» «Straight Walk» [12], con lo cual redujo enormemente el costo computacional. En el trabajo actual «C2M», a parte del refactoring de código, hubieron otros tratamientos cuyo objetivo fue mejorar el rendimiento general de la aplicación.

- **Creaciones redundantes de objetos.**

  Existían casos de objetos que eran creados en cada iteración de refinamiento, incluso si en cada una de estas se generaba un nuevo objeto idéntico. Se identificaron tales casos, y se mejoró el código al «subir» la creación de dichos objetos al «scope» adecuado, que permitiera reutilizarlo en todas las iteraciones necesarias. Esto ocurría principalmente durante las creaciones de los objetos de métodos de inserción de nuevos vértices (Inserción «Delaunay» o «No Delaunay»), y de elección de coordenadas de nuevos «Steiner Points». Ahora los objetos son reutilizados apropiadamente en vez de ser instanciados en cada paso, variando sólo los parámetros que necesitan cambiar según la iteración.

- **Valores comparativos «precalculados».**

  En la implementación de una cola de prioridad, es posible indicar explícitamente una función comparadora, la cual se ejecuta cada vez que es necesario establecer el ordenamiento. Es importante minimizar la complejidad de las comparaciones.

Si cada comparación necesita, por ejemplo, calcular el circunradio, el proceso puede ser agilizado precalculando el valor y guardándolo como una propiedad del triángulo. En este caso, el tiempo total de procesamiento no se ve prácticamente afectado, sin embargo, gracias a esta consideración, es posible identificar de manera más exacta las posibles demoras del resto de las fases de los algoritmos.

Con esto, junto a otros cambios y un trabajo más minucioso - línea a línea - la aplicación presenta finalmente mejorías tanto en utilización de recursos, como tiempos de ejecución. Se puede apreciar en la siguiente tabla comparativa, que muestra resultados del tiempo de demora, en milisegundos, para ejecuciones del algoritmo de Ruppert sobre triangulaciones de distintos números de vértices, y con un ángulo de exigencia de 20°.

N° Vértices	**Compare2DMesh**	**C2M**
100	5	0
500	40	15,6
1000	70	31,2
5000	800	93,7
10000	2000	187,5
50000	18000	1890,7
100000	No indicado	4515,8

Figura 5.20: Tabla comparativa de tiempos de ejecución de Compare2DMesh vs. C2M.

Cabe señalar que las mediciones cuentan con un error asociado, el cual se describe en 5.2.8, explicando tiempo «anormales» (como 0 ms), y otras situaciones. Independiente de esto, el orden de magnitud del tiempo de ejecución de la nueva versión del código es inequívocamente menor.

Administración manual de memoria

La decisión de utilizar el lenguaje de programación C++ para implementar las aplicaciones anteriores: «MeshSuite» y «Compare2DMesh», se basó en gran medida en la reconocida eficiencia de C++ en cuanto a velocidad de ejecución [3, 4]. Un tema importante en el desarrollo de programas escrito en dicho lenguaje, es la administración manual (o explícita) de memoria, debido a la inexistencia de un recolector de basura en tiempo de ejecución.

Fugas de memoria en Compare2DMesh

La aplicación contaba con reconocidos problemas con el manejo de memoria. Uno de los puntos descrito como trabajo futuro es explícitamente: «Eliminación de fugas de memoria del software y mejoramiento del manejo de memoria de las estructuras». Durante el trabajo actual, no se pudo ejecutar herramientas para detectar dichas fugas,

principalmente debido a problemas de incompatibilidades de dependencias y compilación del proyecto «Compare2DMesh» con el ambiente de desarrollo. Sin embargo, sí se comprobó la existencia de fugas, aunque de forma analítica. El código a continuación muestra un claro ejemplo.

```
bool RefineProcess::refine(Options *options){
   Triangle* targetTriangle;
   Triangle* selectedTriangle;
   Configuration* conf;
   InsertionMethod* im;
   /* ... */
   if (this->meshp->isVirgin() ){
     this->encroachedEdges = new QueueOfEncroachedEdges();
     /* ... */
   }
   if (this->meshp->isVirgin() || this->lastTriangleSelection != options->triangleSelection()){
     this->trianglesToProcess = FactoryQueueOfTrianglesToProcess::build(options->triangleSelection(),
          this->meshp, options->triangleSelectionValue());
   }
   if( !options->onlyFirstPreProcess()|| (options->onlyFirstPreProcess() && this->meshp->isVirgin()) ){
     FactoryPreProcess::build((Constant::PreProcess)(options->preProcess()),
          this->meshp, this->trianglesToProcess, this->encroachedEdges, options)->execute();
   }
   if( selectedTriangle != 0){
     /* ... */
     conf = FactoryNewPointMethod::create(this->meshp, targetTriangle, options->newPointMethod()
          options->triangleSelectionValue());
     if(conf->triangle() != 0){
        /* ... */
        im = FactoryInsertionMethod::create(conf, this->trianglesToProcess, this->encroachedEdges, options);
        /* ... */
     }
     /* ... */
   }
   /* ... */
}
```

Figura 5.21: Código de método con fugas de memoria.

Para comprender bien lo que sucede en la figura 5.21, es necesario considerar lo siguiente:

1. El método descrito: RefineProcess.refine(...), es llamado dentro de un ciclo «while», numerosas veces. Prácticamente una vez por cada vértice insertado.
2. Varias líneas de códigos crean objetos en la memoria dinámica, los cuales no son liberados posteriormente.
3. Las líneas que contienen «/* ... */» indican la presencia de código que es irrelevante para este ejemplo.

En el código, hay un llamado explícito a la palabra clave «new», que reserva memoria e instancia un nuevo objeto del tipo «QueueOfEncroachedEdges», el cual no es eliminado (con «delete») en ninguna parte del proyecto. Adicionalmente, en la misma figura existen cuatro llamadas a funciones tipo «Factory», las cuales también instancian objetos con «new» y retornan un puntero. Ninguno de estos cuenta con su correspondiente eliminación en todo el proyecto.

Este desbalance entre adquisición y liberación de recursos tiene como resultado fugas, las cuales debieron ser corregidas al implementar la nueva biblioteca estática «C2M».

Las probabilidades de implementar código con fugas de memoria aumenta al instanciar dinámicamente objetos de C++ a través de la palabra clave «new» y al desconocer efectivamente cuando este objeto instanciado deja de ser útil, y así poder ser liberado. Para sobreponerse a estos problemas, se reescribieron numerosas partes del código teniendo en cuenta tres conceptos:

1. RAII

Siglas de «Resource Acquisition Is Initialization», o en español «Adquirir Recursos es Inicializar», es un patrón de diseño utilizado en distintos lenguajes de programación, cuyo fin es controlar y limitar recursos a los bloques de código donde son requeridos. La idea básica es ligar los recursos con instancias de objetos locales de manera que al salir del «scope» del código donde se declaró, su destructor libere dichos recursos [13]. Si se sigue el patrón de diseño, no existe la posibilidad de que el programador olvide liberarlos, debido a que se hace de manera automática al finalizar el bloque, incluso en caso de «excepciones». Para que esta estrategia funcione, se debe tener presente el concepto de «propiedad de objetos», u «object ownership», la cual define una relación entre la «vida» de un objeto y otro.

2. Propiedad de objetos

A veces llamado simplemente «ownership», en el contexto de objetos de programación es, a grandes rasgos, el concepto que establece cuál objeto es dueño de otro. En consecuencia, cuando el dueño es destruido, los objetos que dependen de él deben ser destruidos también, ya que dejan de ser relevantes. Existe más de un tipo de relación de «propiedad»: «única» y «compartida», siendo primera la más común.

3 Punteros inteligentes

C++11 incluye en sus bibliotecas los tipos «smart pointers», herramientas con las cuales es posible implementar de manera explícita los esquemas de propiedad que pertenecen a un diseño de aplicación. En este caso, se utilizó solamente «unique_ptr», el cual es un «smart pointer» que tiene las siguientes propiedades:

1. Es el único propietario de un objeto, el cual es destruido cuando el «unique_ptr» sale del «scope».
2. Dos «unique_ptr» no pueden ser propietarios del mismo objeto (y en consecuencia, los «unique_ptr» no pueden ser copiados).
3. La propiedad del objeto administrado es transferible de un «unique_ptr» a otro, garantizando que siempre exista sólo un «unique_ptr» propietario.

El funcionamiento de «unique_ptr» se basa principalmente en los conceptos de «rvalue references» y «move semantics»[13], evitando técnicas como el seguimiento o conteo de referencias utilizadas por otros tipos de punteros inteligentes, como «shared_ptr».

Se utilizaron punteros inteligentes del tipo «unique_ptr» en el desarrollo de «BatchRefinement», en los lugares donde no era posible declarar e inicializar objetos dentro del «scope» actual.

El código fue reescrito considerando estas ideas. Debido a la nueva estructura de las clases y métodos no es fácil incluir de manera concisa el código resultante que corresponde al nuevo proyecto, ya que las nuevas líneas equivalentes a las descritas en la figura5.21 están distribuidas en distintos métodos y clases.

Detección de fugas

El compilador de VisualC++, provee herramientas para depurar problemas relacionados al manejo de memoria. Unas de las principales herramientas para la detección de fugas son «las funciones de depuración de la pila, de las bibliotecas de tiempo de ejecución de C», en inglés «C Run-Time Libraries (CRT) debug heap functions»[14]. Explicado de manera simplificada, las utilidades permiten reemplazar las funciones del lenguaje «C»: «malloc», «free» , y el operadores «new» de «C++», por versiones «debug», con las cuales se hacen seguimientos de las asignación de memoria («memory allocations») y liberaciones («deallocations»). Gracias a esto, es posible obtener el estado de uso de la memoria en cualquier momento. La técnica sugerida en la documentación consiste en obtener «snapshots» del estado de la memoria antes y después del código que se pretende observar. Posteriormente los estados se comparan. En el caso donde no hay fugas de memoria, todos los recursos adquiridos deberían haber sido liberados, por lo que los números de bloques de memoria utilizados debiesen ser los mismos en ambos casos.

Para comprobar que efectivamente se corrigió el problema en C2M, haciendo uso de las funciones de depuración provistas por VisualC++ se implementó una clase **«MemoryDebug»** que al ser instanciada guarda el estado de la memoria del proceso («snapshot»), y que al ser destruida calcula las diferencias entre el estado actual respecto al inicial, reportando si se produjeron o no fugas.

```
class MemoryDebug{
  private:
    _CrtMemState _initial_state;
  public:
    MemoryDebug(){
      _CrtMemCheckpoint(&_initial_state);
    }
    ~MemoryDebug(){
      _CrtMemState current_state;
      _CrtMemCheckpoint(&current_state);
      _CrtMemState difference;
      if (_CrtMemDifference(&difference, &_initial_state,
&current_state)){ //_CrtMemDifference retorna 0 si no hay fugas
        std::cout << "Fugas de memoria encontradas!\n";
      }
      else{
        std::cout << "Sin fugas de memoria\n";
      }
      _CrtMemDumpStatistics(&difference);
    }
};
```

Figura 5.22: Clase «MemoryDebug» para detectar fugas de memoria, implementada utilizando «C Run-Time Libraries (CRT) debug heap functions».

Con la clase «MemoryDebug», se escribió un «test» cuyo objetivo es verificar si existen fugas de memoria al ejecutar un proceso completo de refinamiento.

```
void C2M::Utils::execute_memory_test(std::string filename,
      C2M::Options & options,
      bool test_convergence,
      double divergence_ratio){
  C2M::MemoryDebug mem;
  C2M::RefineProcess refinement(test_convergence, divergence_ratio);
  C2M::Utils::load_binary_initialized_triangulation_file(filename,
refinement.get_triangulation());
  C2M::RefinementResult result;
  refinement.execute_all(options, result);
}
```

Figura 5.23: «Test» implementado para detección de fugas de memoria que abarca todos los pasos del refinamiento.

El «test» implementado para la detección de fugas es simple y consta de 4 pasos, cada uno con su correspondiente línea de código en la figura 5.23:

1. Se instancia un objeto de la clase «MemoryDebug», que al salir del «scope» indica si toda la memoria asignada desde su creación fue liberada o hay fugas.

2. Se instancia un objeto de la clase «RefineProcess».
3. Se carga una triangulación desde un archivo al objeto «RefineProcess».
4. Se ejecuta el refinamiento y se guardan los resultados en un objeto.

Como los objetos están definidos en «la pila» («stack») y no fueron creados con el operador «new», se espera que sus destructores sean llamados al finalizar el bloque.

El reporte de utilización de memoria, que indica la diferencia entre el estado final respecto al inicial, obtenido desde el «test» y generado por «MemoryDebug» es el siguiente:

```
0 bytes in 0 Free Blocks.
0 bytes in 0 Normal Blocks. (<== Sin fuga de memoria)
79996 bytes in 102 CRT Blocks.
0 bytes in 0 Ignore Blocks.
0 bytes in 0 Client Blocks.
Largest number used: 1219329 bytes.
Total allocations: 3111329 bytes.
```

De la documentación oficial de «C Run-Time Libraries (CRT) debug heap functions» se esclarece que el valor que se debe observar de la estadística entregada es «Normal blocks»[14], el cual corresponde a asignaciones y liberaciones de memoria realizadas por el usuario. Este resultado indica que se liberaron tantos bloques como fueron asignados, lo que confirma la correcta administración de memoria de la aplicación.

Para comprobar el funcionamiento de la clase «MemoryDebug» junto con «C Run-Time Libraries (CRT) debug heap functions», se ejecutaron pruebas simples que generaban fugas de memoria deliberadas, producidas por la línea:

```
int* arr = new int[20];//4 x 20 = 80 bytes
```

El resultado de estas pruebas, donde sí se esperan fugas, es el siguiente.

```
0 bytes in 0 Free Blocks.
80 bytes in 1 Normal Blocks. (<== Fuga de memoria)
0 bytes in 0 CRT Blocks.
0 bytes in 0 Ignore Blocks.
0 bytes in 0 Client Blocks.
Largest number used: 0 bytes.
Total allocations: 80 bytes.
```

El total de bytes reportados asignados y sin liberar coincide con el número calculado, y en este caso sí se observa una fuga de memoria. Se obtuvo el resultado buscado, y así se comprobó que las herramientas de diagnóstico de memoria funcionan bien y fueron correctamente utilizadas.

En el caso de la ejecución de múltiples procesos de refinamientos, el código es básicamente el mismo, pero dentro de un ciclo que ejecuta una iteración para cada experimento. Ya que cada iteración está libre de fugas, se concluye que el proceso total tampoco lo

tiene, ya que no realiza tareas adicionales. Finalmente, la biblioteca estática «C2M» quedó libre de fugas de memoria.

5.2.6. Separación de proyectos

Debido al alcance del trabajo, el desarrollo de la aplicación final involucró finalmente diversas herramientas, algunas de las cuales son necesarias para su correcto funcionamiento. Los requerimientos actuales, en su globalidad, son los siguientes:

1. Poder ejecutar aplicaciones compiladas desde C++. Es lo principal, y afortunadamente la mayoría de los sistemas operativos tienen una opción tanto para compilarlas a código nativo y ejecutarlas.
2. Runtimes del framework .Net 4.0, necesario principalmente para la interfaz gráficas y otros procesos menores.
3. (Opcional) OpenTK y soporte para OpenGL 2.0. Se requiere para la visualización de la triangulación.

Al aumentar la cantidad de requerimientos necesarios para la aplicación, se produce una solución que puede padecer de una incapacidad de ejecución debido que no existe un determinado programa instalado en el computador. Incluso si sólo se requiriera realizar un refinamiento, sin visualización ni gráficos, la aplicación podría presentar problemas de ejecución debido a la falta de dependencias.

Para obtener mejores resultados, la implementación del programa se dividió finalmente en 6 módulos, los cuales corresponden a aplicaciones ejecutables o, bibliotecas estáticas y dinámicas. La separación de esta forma favorece el aprovechamiento de las ventajas que ofrece cada tipo de compilación. Esta modularización permite minimizar las dependencias para propósitos particulares, por ejemplo, si sólo se requiere ejecutar refinamientos desde la línea de comandos, no es necesario tener instalado el framework .Net. Así además, se hizo uso de la eficiencia de C++ para las partes donde el rendimiento es crítico, y en otras partes, se obtuvo beneficios de las herramientas de desarrollo e interfaz de usuario proveídas por el framework .Net. Todo por separado. Los módulos resultantes son los siguientes.

1. C2M «biblioteca estática»
2. C2Mcmd «ejecutable»
3. C2MInterop «biblioteca dinámica»
4. BatchRefinement «ejecutable»
5. ChartCreator «biblioteca dinámica»

De manera adicional, actualmente se pueden realizar compilaciones para 32 o 64 bits. Para que esto ocurra, es necesario contar con las librerías correspondientes, lo cual es el caso de cada uno de los módulos.

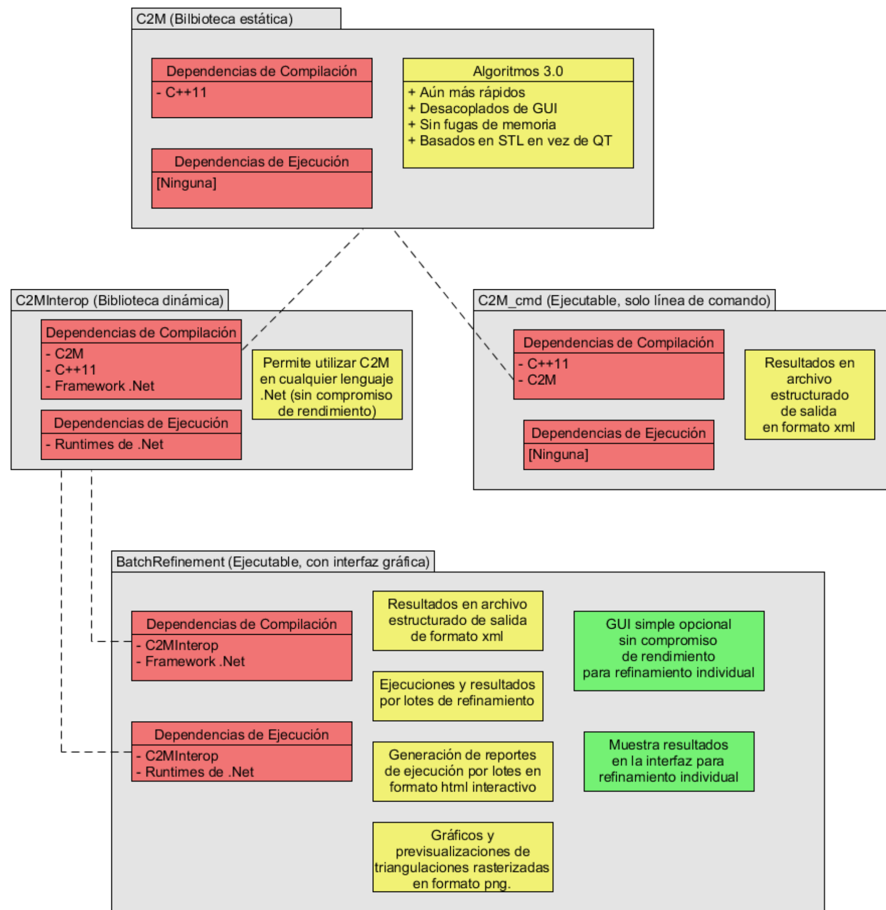


Figura 5.24: Diagrama de características de las aplicaciones desarrolladas en torno a «C2M» de la solución final.

Justificación de la elección de lenguajes

Inicialmente, se consideró que la aplicación estuviera completamente escrita en C++. Esta idea era apoyada por la reputación de eficiencia y relativa portabilidad ofrecida por un programa escrito en dicho lenguaje y que no requiere de bibliotecas no estándar. Un contratiempo asociado a esta elección consiste en dificultades de desarrollo, especialmente cuando el alcance del proyecto y sus objetivos se ven enfrentados a una fecha de entrega. Se decidió que solo se utilizaría C++11 para implementar una biblioteca estática que contuviera sólo las funcionalidades críticas en cuánto a rendimiento.

Para la aplicación con interfaz gráfica, se eligió desarrollar utilizando el framework .Net, con el lenguaje **VisualBasic.Net**. Una de las principales características de este lenguaje que favoreció su utilización, es su enfoque de «Desarrollo Rápido de Aplicaciones», o RAD por sus siglas en inglés. Cabe notar, que a pesar de que VB.Net no tiene buena reputación en el ámbito, es un lenguaje orientado a objetos, y su potencial es prácticamente el mismo que uno de los más populares de Microsoft, como C#. Este lenguaje mantiene una estrecha relación con el IDE Visual Studio, combinación que facilita mucho los quehaceres los programadores. El incremento en la complejidad de la solución, al agregar un nuevo lenguaje de programación,

se vé más que compensada en este caso por los aspectos positivos mencionados.

Existe un único problema con esta elección de lenguajes, y este consiste en que no se puede interactuar fácilmente entre una aplicación desarrollada en VB.Net con una biblioteca estática programada en C++. Para solucionar esto es necesario implementar un biblioteca dinámica en C++/CLI que hace de intermediario entre el código VB.Net y el de C++.

El lenguaje C++/CLI cuenta casi con las mismas capacidades de VB.Net y de C#. Una pregunta natural sería entonces, «¿Por qué no programar todo en C++/CLI (en vez de VB.Net)?» La razón para no hacerlo es similar a la decisión de no usar C++11 para toda la aplicación, la cual se fundamenta en temas de productividad y de abarcar la mayor cantidad de objetivos dentro del plazo.

Detalles de módulos

Siguiendo una categorización de acuerdo al tipo de compilación, se listarán los módulos separados en 3 grupos. El primer grupo, «Código Nativo» también llamado «Código No Administrado» corresponden en este caso a las bibliotecas o programas que fueron implementadas utilizando sólo C++11, cuya compilación genera directamente código de máquina, el cual es ejecutado directamente por el sistema operativo sin necesidad de un «intérprete». Las características principales de este grupo son la eficiencia de ejecución y el manejo explícito de memoria. Otro grupo corresponde al «Código Administrado», cuya característica consiste en que su compilación produce un «código intermediario» que es interpretado en su ejecución, en este caso, por el «runtime» del «framework» .Net. El entorno «administrado» provee un «Garbage Collector», o recolector de basura, por lo que no es necesario el manejo explícito de memoria. La interacción entre código nativo y administrado no es particularmente directa. De hecho, casi todos los lenguajes de .Net no reconocen clases ni métodos nativos, por lo que es necesario tener un intermediario. El tercer grupo, «Código Mixto», es el que hace de puente entre el código nativo y el administrado.

Código Nativo

C2M

Biblioteca estática. Compuesta por las implementaciones de los algoritmos y estructuras de datos necesarias para los refinamientos. Está escrita en C++11, y es compilada como una biblioteca estática. Acá se concentra casi la mitad del trabajo realizado.

Utilizando como base la implementación de «Compare2DMesh» (de ahí el nombre «C2M») de F. Gallardo, se hizo un trabajo de revisión de algoritmos, correcciones a fugas de memoria, optimizaciones de algoritmos,simplificación de arquitectura de clases, y de desvinculamiento de la antigua dependencia a QT, quedando sólo código C++11 con uso de STL «Standard Template Library», lo cual produjo como resultado «C2M», que mejora en simpleza, eficiencia, y compatibilidad a su predecesora.

Requerimientos:

Compilación: Algún compilador que soporte C++11. Existe al menos uno para prácticamente todas las plataformas.

Ejecución: La capacidad de ejecutar el código producido por algún compilador compatible a la plataforma de ejecución. Nuevamente, es una característica que la mayoría soporta.

Aspectos destacados: Eficiencia, principalmente debido al hecho de que el código producido es de bajo nivel.

C2Mcmd

Aplicación ejecutable, sin interfaz gráfica, que permite hacer uso de las funciones de la biblioteca C2M, a través de la línea de comando.

Requerimientos: Mismos que C2M.

Aspectos destacados: Mismos que C2M.

Entradas y salidas: Recibe como entrada el nombre de un archivo de triangulación y opciones de refinamiento. Produce como resultado un archivo xml con los datos obtenidos.

Código Administrado

BatchRefinement

Aplicación ejecutable. Compuesta por gran parte del trabajo desarrollado. Es un programa con interfaz de usuario, con el cual se facilita el ingreso de datos y ejecución de procesamiento por lotes. Hace uso de todas las otras bibliotecas: C2M, C2MInterop, y ChartCreator, para ejecutar secuencias de refinamientos y generar reportes estructurados, en base a tablas, y gráficos. Adicionalmente cuenta con herramientas de conversión de formatos de archivos, y con un visualizador de triangulaciones. Está escrito el lenguaje de programación «VB.Net», y utiliza la biblioteca de interoperabilidad C2MInterop para poder acceder a las funcionalidades de código no administrado de C2M.

- **Requerimiento fundamental:** Framework .Net 4.0.
- **Requerimiento opcional:** Soporte para OpenGL 2.0 para visualización de las triangulación con aceleración gráfica.
- **Aspectos relevantes:** El framework provee herramientas de desarrollo y también facilita la implementación de interfaces gráficas de usuario.
- **Entradas y salidas:**

  Resultados «xml»: Para producir resultados de refinamientos, se puede elegir un conjunto de archivos de triangulaciones y de refinamientos a ejecutar a través de la interfaz gráfica. La salida es un archivo xml de resultados, al igual que C2M.

  Reportes: Para crear un reporte, se selecciona un archivo de resultados xml, y se elige el tipo de reporte a generar (De variación de ángulos o de vértices de entrada). El resultado es un reporte «Html» interactivo y navegable, e imprimible, con estadísticas de las ejecuciones.

ChartCreator

Biblioteca dinámica. Provee solamente de una función para generar una imagen con un gráfico. Al ser dinámica, se carga sólo si se utiliza. Es decir, si la ejecución de la aplicación «BatchRefinement» no necesitara producir gráficos, esta biblioteca no se cargaría, y por lo tanto, tampoco se necesitarían sus dependencias.Este módulo fue separado del proyecto principal con el propósito de aislar las posibles dependencias, y proveer la oportunidad de ésta pueda ser implementada sin necesidad de recompilar el resto de la aplicación, siempre que se mantenga la interfaz programática. En este trabajo en particular, ChartCreator crea imágenes de gráficos utilizando la biblioteca «OxyPlot»[15], la cual permite el ploteo de gráficos en aplicaciones .Net.

- **Requerimientos de compilación y ejecución:** Framework .Net 4.0
- **Entradas y salidas:** Genera imágenes a partir una tabla de datos entregada como una arreglo bidimensional.

Código Mixto

C2MInterop

Biblioteca dinámica. Existe un tipo de compilación que soporta tanto código administrado y nativo a través del cual es posible conectar la biblioteca desarrollada en C++ al resto de la aplicación implementada en «.Net». A este tipo de proyecto se le conoce como «modo mixto», y permite escribir programas compuestos por dos sintaxis: C++ para el código nativo, y C++/CLI para el código administrado. Como toda alternativa, esta tiene sus ventajas y desventajas, pero finalmente muestra su valía al permitir la interacción entre distintos componentes. Se usa principalmente para escribir «Wrappers» o «Adapters» de bibliotecas nativas para poder ser accedidas por otros lenguajes de «.Net», lo cual corresponde al caso de este trabajo.

- **Requerimientos:** Compilador C++/CLI, y Framework .Net 4.0

Binarios de 32 y 64 bits

La aplicación y todas las bibliotecas relacionadas ofrecen compilaciones para máquinas con arquitectura x86, y x64. Cada una tiene características propias, ventajas y desventajas, que deben ser consideradas a la hora de elegir la opción adecuada. Las aplicaciones x86 tienen como limitación principal el hecho de que el número máximo de direcciones de memoria permite como tope una utilización de aproximadamente 4GB de RAM. Dependiendo de la plataforma, este número es menor. Este límite es debido a que los números enteros de 32 bits desde los cuales se definen los punteros en memoria, pueden producir un máximo de $2^{32} = 4.294.967.296$ direcciones, cada una indicando un único byte. Los sistemas de 64 bits, tienen un techo mucho mayor, llegando a un máximo teórico de $2^{64} = 18.446.744.073.709.551.616$ bytes, número que sobrepasa por lejos a las capacidades de equipos de uso común actuales, cuya memoria oscila en torno a números de 2 dígitos de Gigabytes.

En el computador de desarrollo, se observaron importantes diferencias en el uso de memoria y tiempo de procesamiento de una compilación respecto a la otra. Las mediciones de los costos de memoria, observados vía el «Administrador de Tareas», los tiempos de ejecución, son los siguientes:

- x86:
  - GUI: Solo la ventana de interfaz gráfica abierta: 5 MB
  - GUI + Triangulación sin procesar de 1000 vértices: 17MB
  - Tiempo para refinamiento con exigencia de 30 grados para algoritmo Lepp-Centroide: 30 [ms]
- x64:
  - GUI: Solo la ventana de interfaz gráfica abierta: 30MB
  - GUI + Triangulación sin procesar de 1000 vértices: 55MB
  - Tiempo para refinamiento con exigencia de 30 grados para algoritmo Lepp-Centroide: 201 [ms]

Las mediciones muestran que el se obtienen mejor uso de recursos utilizando la versión de 32 bits. Una posible causa para dicho aumento en recursos y tiempos puede ser atribuida al framework .Net.

Si se desconocen las características del equipo objetivo, o simplemente para uso general, la compilación x86 es la alternativa segura recomendada.

5.2.7. Paralelismo

Actualmente es común que los computadores cuenten con más de un núcleo de procesamiento. Pueden ejecutar más de un proceso de manera simultánea, uno en cada núcleo, disminuyendo los tiempos de respuesta en aplicaciones que hacen uso de esta ventaja.

En ***«BatchRefinement»***, es posible ejecutar algoritmos de refinamiento en distintos «threads», acortando el tiempo de espera en la obtención de resultados de un conjunto de experimentos. Existen, incluso, algoritmos que pueden beneficiarse de un entorno de núcleos múltiples [16], los cuales no forman parte de la aplicación, y quedan propuestos como adiciones futuras.

Cualquiera sea el caso, es importante notar que al ejecutar elementos de manera paralela, siempre existe una carga adicional de recursos o trabajo que debe ser satisfecha. En este caso, como la aplicación puede ejecutar 2 o más refinamientos simultáneos, todos deben ser cargados en memoria, lo cual puede producir problemas de memoria para las mallas más grandes. El control de cuantos «threads» se ejecutan a la vez, queda a elección del usuario.

Esto se implementó utilizando un «ThreadPool», el cual consiste en una «cola» de «N» tareas (algoritmos) y un número fijo «M» de «hilos de ejecución»(«threads»). A grandes rasgos, cada uno de los M hilos ejecuta una de las tareas de la cola. Cuando termina una, sigue con otra, removiéndola de la cola, hasta que no quedan más. Cuando la cola de tareas se vacía, y la última completó su ejecución, la aplicación continúa y se presentan y exportan los resultados.

5.2.8. Precisión y resolución de mediciones de intervalos de tiempo

Una «mejora» implementada a partir del programa anterior «Compare2DMesh», fue utilizar bibliotecas estándar para la medición de lapsos de tiempos transcurridos. En particular, se utilizó «chrono», que forma parte de C++11. Lamentablemente, esta decisión no quedó libre de desventajas. A nivel de «instrucciones de cpu», la frecuencia con que las que se ejecutan las interrupciones para consultar el tiempo actual del sistema, en esta implementación, pueden ser indeseablemente elevadas. Estas dependen del sistema operativo que las ejecuta, y también de características del Hardware. Más aun, las implementaciones de C++11 varían de compilador en compilador. En ciertas versiones de Microsoft Visual Studio (anteriores a las 2015), en particular del compilador de VC++, las interrupciones que generan los intervalos medibles de tiempo que utiliza la aplicación corresponden a una espera de aproximadamente 15 milisegundos. La precisión del tiempo del sistema suele ser al nivel de nanosegundos, pero debido a la separación entre intervalos no es posible medir eventos de corta duración de esta manera. Existen alternativas que permiten consultar lapsos con mayor resolución, sin embargo, dependen de llamadas al API de Windows, lo cual es un tema que se pretende evitar. Otra opción utilizada frecuentemente, es ejecutar los experimentos de corta duración un gran número de veces, y finalmente dividir el tiempo total por el número de ejecuciones. Esto es debido a que en grandes lapsos, 15 milisegundos pasan a ser menos significativos.

En conclusión, y para efectos de este informe, las mediciones pequeñas cercanas o menores a 15 milisegundos deben ser consideradas como una aproximación, y en general, para cada una de los tiempos totales, es seguro suponer un margen de error de $\pm 15[ms]$.

Parte III

Resultados

6. Pruebas y resultados

6.1. Objetivos de las pruebas

Durante el trabajo, se realizaron numerosas ejecuciones de refinamientos considerando una variedad de configuraciones iniciales y resultados obtenidos. Los principales objetivos de las pruebas fueron lo siguientes:

1. Corroborar el buen funcionamiento de la aplicación, en particular, de refinamientos y la generación de reportes.
2. Obtener resultados visuales de casos particulares que ilustren situaciones icónicas de los temas revisados en este informe, y así además mostrar las funcionalidades ofrecidas por la aplicación.
3. Comprobar que hubo una mejoría respecto al trabajo de memoria anterior.
4. Obtener resultados de distintos refinamientos en triangulaciones conocidas, de otras publicaciones.
5. Reproducir algunos de los resultados y conclusiones planteados en las memorias anteriores.

Los primeros tres objetivos fueron abordados inicialmente, y gracias a esto se pudo desarrollar este documento. Los detalles y resultados están distribuidos en las secciones donde eran requeridos. Para las siguientes pruebas, los resultados constituyen las subsecciones a continuación.

6.2. Primeros resultados

En primera instancia se realizaron pruebas para triangulaciones de tamaño modesto. Luego, para tener una idea sobre los límites prácticos de la aplicación, se experimentó con triangulaciones grandes. Adicionalmente, para tener una visión más global de los resultados, de manera de relacionarlos con otras publicaciones, se utilizaron mallas «conocidas», las cuales suelen aparecen publicaciones.

Para las muestras de este apartado, los algoritmos fueron configurados con selección de «Steiner Points» basados en «Lepp-Centroide», «Lepp-PuntoMedio», «Ruppert», y «Üngor» ofrecidas por la aplicación. Otros detalles se pueden encontrar en los anexos correspondientes.

6.2.1. Pruebas simples

El primer paso es comprobar y validar las funcionalidades en pequeños conjuntos de datos, y así facilitar el control sobre detalles. La triangulación escogida es una malla de 7 vértices, y presenta un evidente triángulo de mala calidad en su zona central, incluso para criterios relajados, como muestra la figura 6.1.

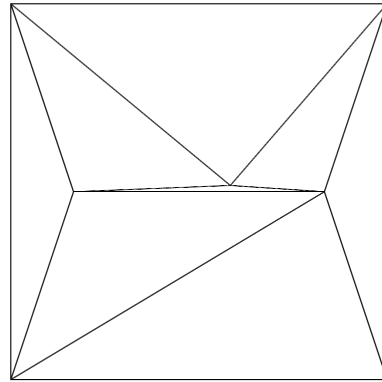
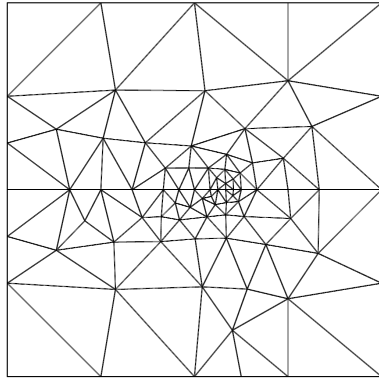


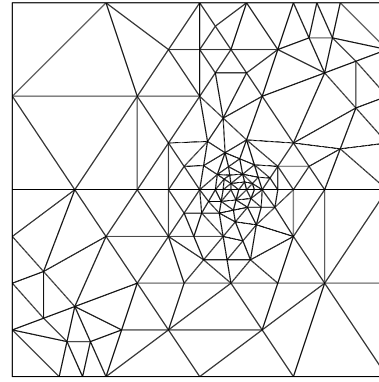
Figura 6.1: Triangulación de 7 vértices utilizada para pruebas.

Para dicha triangulación, se ejecutó una serie de refinamientos siguiendo el esquema de «ángulo variante». Los distintos algoritmos seleccionan coordenadas de inserción de «Steiner Points» diferentes, por lo que el resultado esperado contiene triangulaciones apreciablemente distintas. En esta etapa, la validación es visual.

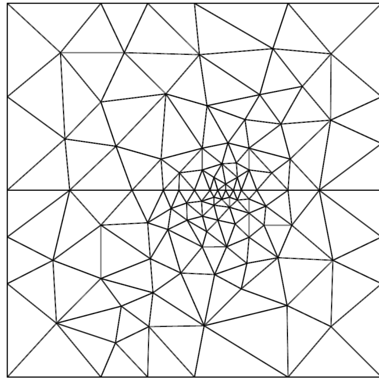
Los resultados obtenidos para 30° de exigencia de ángulo mínimo interior, de cada uno de los algoritmos, son los que muestra la figura 6.2.



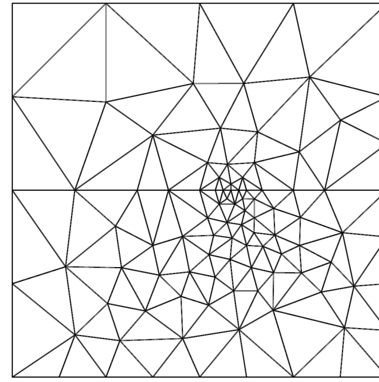
30° exigencia, Algoritmo «Lepp-Centroide»



30° exigencia, Algoritmo «Lepp-PuntoMedio»



30° exigencia, Algoritmo «Ruppert»



30° exigencia, Algoritmo «Üngor»

Figura 6.2: Resultados de distintas configuraciones de algoritmos para un mismo ángulo de exigencia sobre triangulación de 7 vértices.

Los resultados numéricos obtenidos para el refinamiento de la malla de 7 vértices iniciales, en cuanto al número final de vértices es el siguiente:

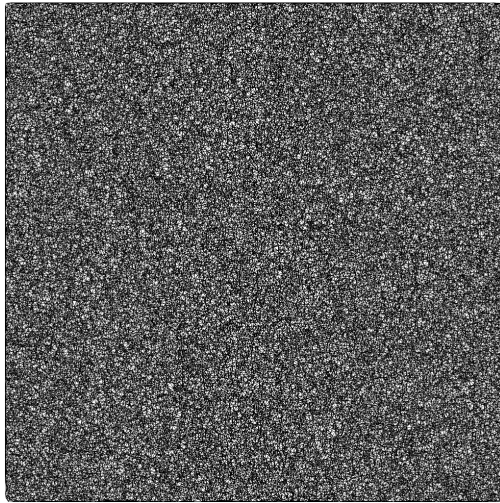
- 30° exigencia, Algoritmo «Lepp-Centroide»: 81 vértices finales.
- 30° exigencia, Algoritmo «Lepp-PuntoMedio»: 112 vértices finales.
- 30° exigencia, Algoritmo «Ruppert»»: 105 vértices finales.
- 30° exigencia, Algoritmo «Üngor»: 99 vértices finales.

En este caso, el algoritmo que presenta mejores resultados es el algoritmo «Lepp-Centroide» debido a que la triangulación producida contiene menos vértices que el resto.

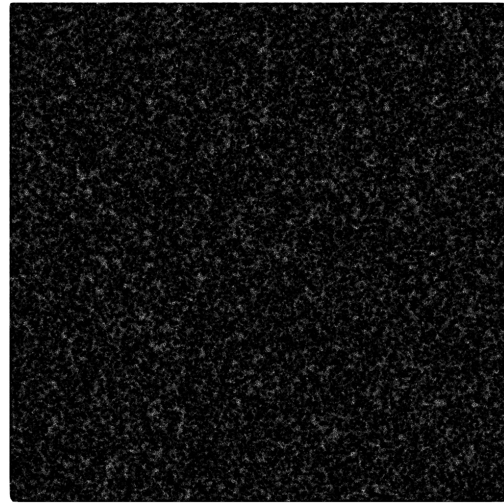
6.2.2. Grandes triangulaciones

Intentando buscar el límite sobre el cual la aplicación responde de manera práctica se ejecutaron pruebas con mallas cuyo número de vértice superan los 100.000.

Debido al gran número de vértices no es posible producir una imagen, a partir de las triangulaciones iniciales y finales, que quepa apropiadamente en algún reporte y que sea distinguible, sin embargo, estas se incluyen a continuación para ilustrar el problema.



«Rasterización» de una triangulación de 100.000 vértices «indistinguible»

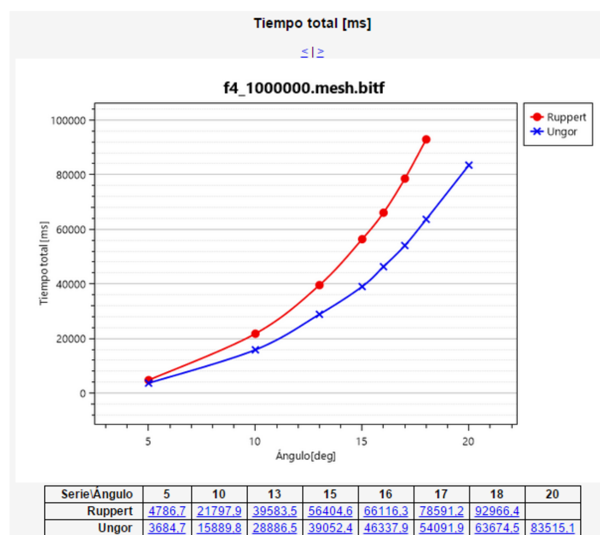


«Rasterización» del mejoramiento de una triangulación de 100.000 vértices iniciales, aun más «indistinguible»

Figura 6.3: Imágenes poco claras para triangulaciones grandes.

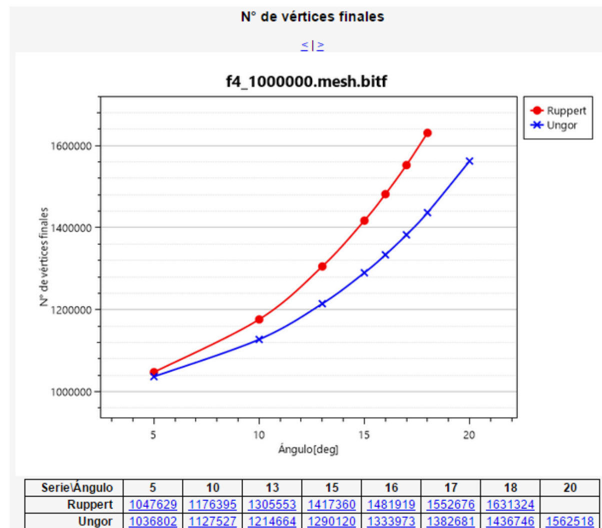
En los experimentos realizados para la «compilación x86» de la aplicación, fue posible trabajar con mallas de hasta un millón de vértices, donde comenzaron a ocurrir problemas, principalmente relacionados a la memoria disponible y utilizada.

Los resultados son, considerando sólo algoritmos de Ruppert y Üngor, para una triangulación de 1.000.000 (un millón) de vértices son los que muestran las figuras 6.4 y 6.5 respectivamente.



Serie\Ángulo	5	10	13	15	16	17	18	20
Ruppert	4786,7	21797,9	39583,5	56404,6	66116,3	78591,2	92966,4	
Ungor	3684,7	15889,8	28886,5	39052,4	46337,9	54091,9	63674,5	83515,1

Figura 6.4: Resultados de tiempo total para una triangulación de 1.000.000 vértices.



Serie\Ángulo	5	10	13	15	16	17	18	20
Ruppert	1047629	1176395	1305553	1417360	1481919	1552676	1631324	
Ungor	1036802	1127527	1214664	1290120	1333973	1382681	1436746	1562518

Figura 6.5: Resultados de número final de vértices para una triangulación de 1.000.000 vértices.

No se incluyeron imágenes de las mallas debido al problema descrito sobre la claridad de «imágenes para triangulaciones grandes».

En determinados casos, la aplicación presentaba un comportamiento poco estable, presentando caídas ocasionales. Se identificaron algunos trozos de código posiblemente problemáticos dentro de los algoritmos heredados, los cuales requieren de un trabajo con especial dedicación para ser representaciones fieles y aun más robustas de algoritmos en los que están basados.

6.2.3. Casos conocidos

Existen algunas triangulaciones que son citadas frecuentemente, como casos singulares de mallas de mala calidad. Son utilizadas para ilustrar el comportamiento de los algoritmos. Uno en particular, es el formado a partir círculos dentro de un cuadrado, que incluye muchos triángulos de mala calidad, en el cual las aristas restringidas son fácilmente observables, y es el que se muestra a continuación.

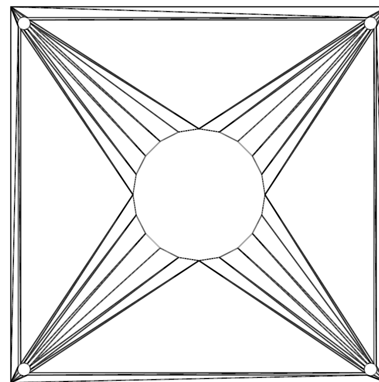
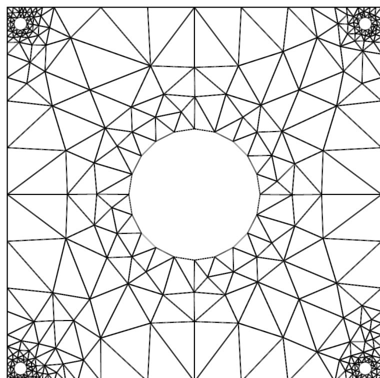


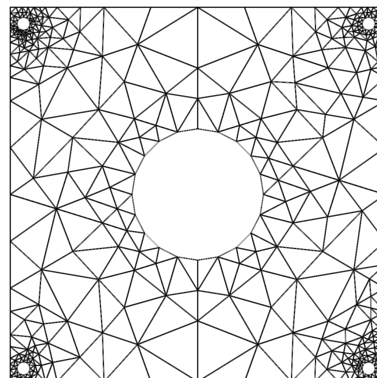
Figura 6.6: Triangulación de 17 vértices formada principalmente por triángulos malos, utilizada en distintas publicaciones.

Al igual que en el caso de prueba sobre una triangulación simple, se ejecutó una serie de refinamientos siguiendo el esquema de «ángulo variante», cuyos resultados detallados se encuentran en el anexo.

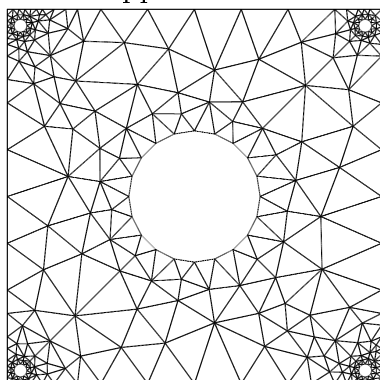
Para observar las diferencias en las mallas producidas, se adjuntan a continuación resultados para 30° de exigencia de ángulo mínimo interior, de cada uno de los algoritmos.



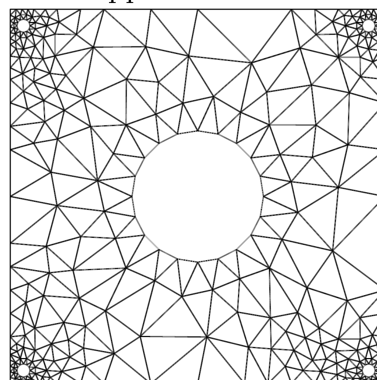
30° exigencia, Algoritmo «Lepp-Centroide»



30° exigencia, Algoritmo «Lepp-PuntoMedio»



30° exigencia, Algoritmo «Ruppert»



30° exigencia, Algoritmo «Üngor»

Figura 6.7: Resultados de distintas configuraciones de algoritmos para un mismo ángulo de exigencia sobre triangulación conocida.

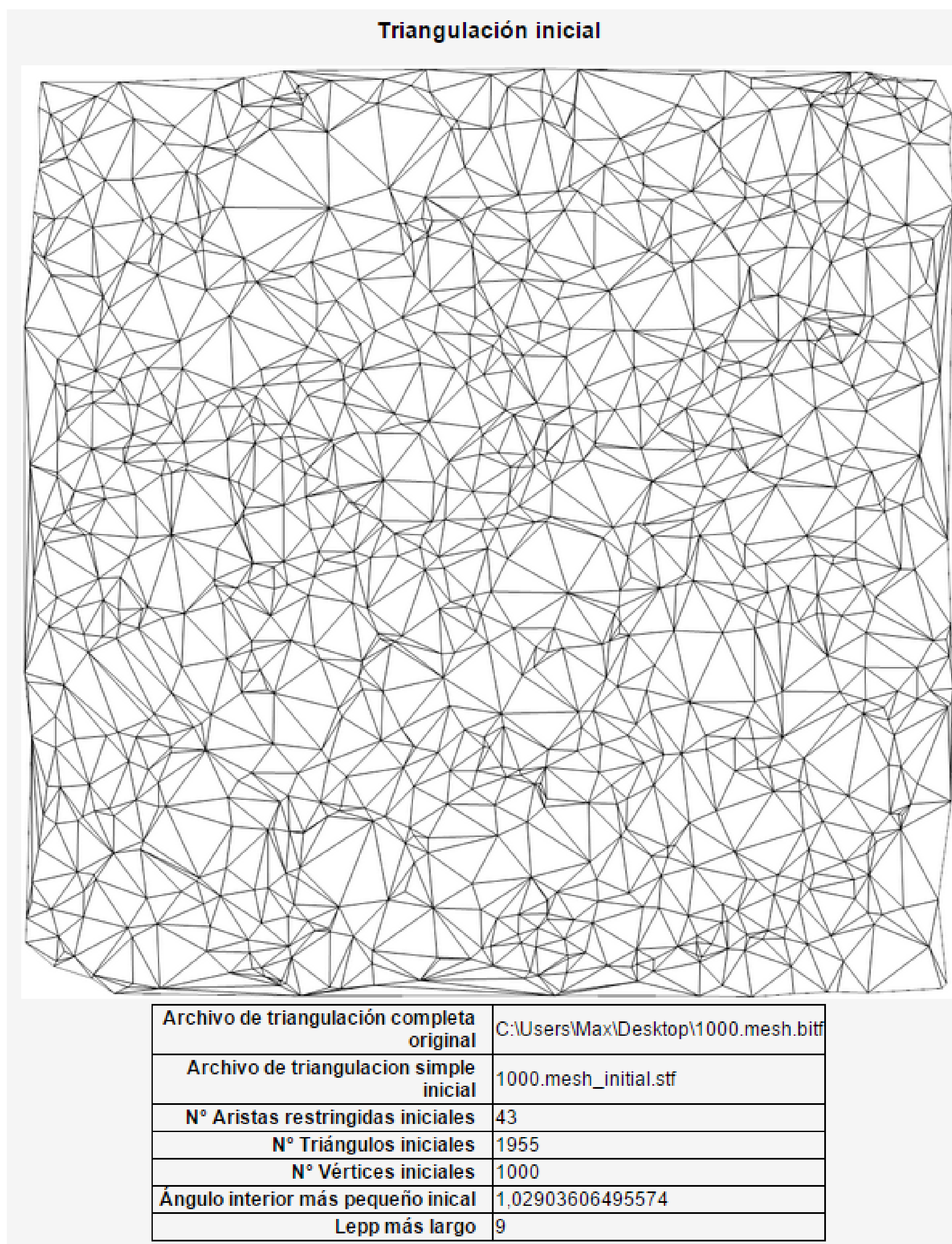
Las aristas restringidas (que conforman las circunferencias) son conservadas, y visualmente no se observan triángulos de mala calidad, hecho que se comprueba en los resultados numéricos de los reportes.

6.3. Comprobando la hipótesis

Una de las hipótesis iniciales, era que «el comportamiento de los algoritmos basados en «Steiner Points» del tipo Lepp-Centroide no se ve particularmente favorecido por el ordenamiento de los datos de entrada, como en el caso de los algoritmos de Ruppert y Üngor, en donde sí se requiere para obtener resultados óptimos». El ordenamiento de elementos siempre tiene un costo asociado, siendo $O(n * log(n))$ la complejidad representativa. Como en cada caso de

optimización, si se pudieran mejorar los tiempos y disminuir la carga de trabajo para obtener resultados similares o mejores, entonces se obtiene un beneficio.

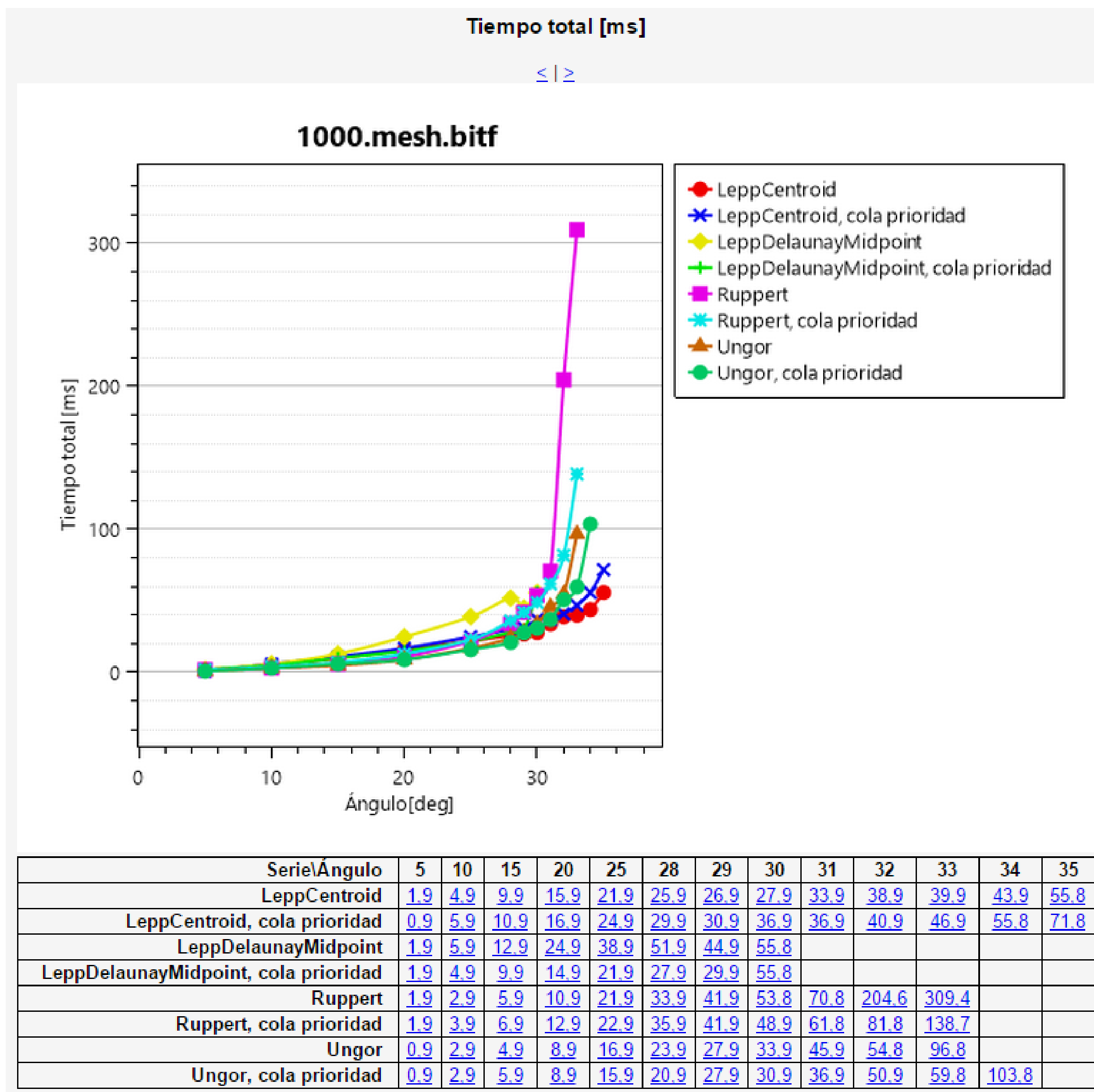
Para comprobar esto, se utilizó la aplicación BatchRefinement, la cual, de manera automática, obtuvo los resultados relevantes y generó un reporte html.



Archivo de triangulación completa original	C:\Users\Max\Desktop\1000.mesh.bitf
Archivo de triangulacion simple inicial	1000.mesh_initial.stf
N° Aristas restringidas iniciales	43
N° Triángulos iniciales	1955
N° Vértices iniciales	1000
Ángulo interior más pequeño inical	1,02903606495574
Lepp más largo	9

Figura 6.8: Triangulación inicial con 1.000 vértices.

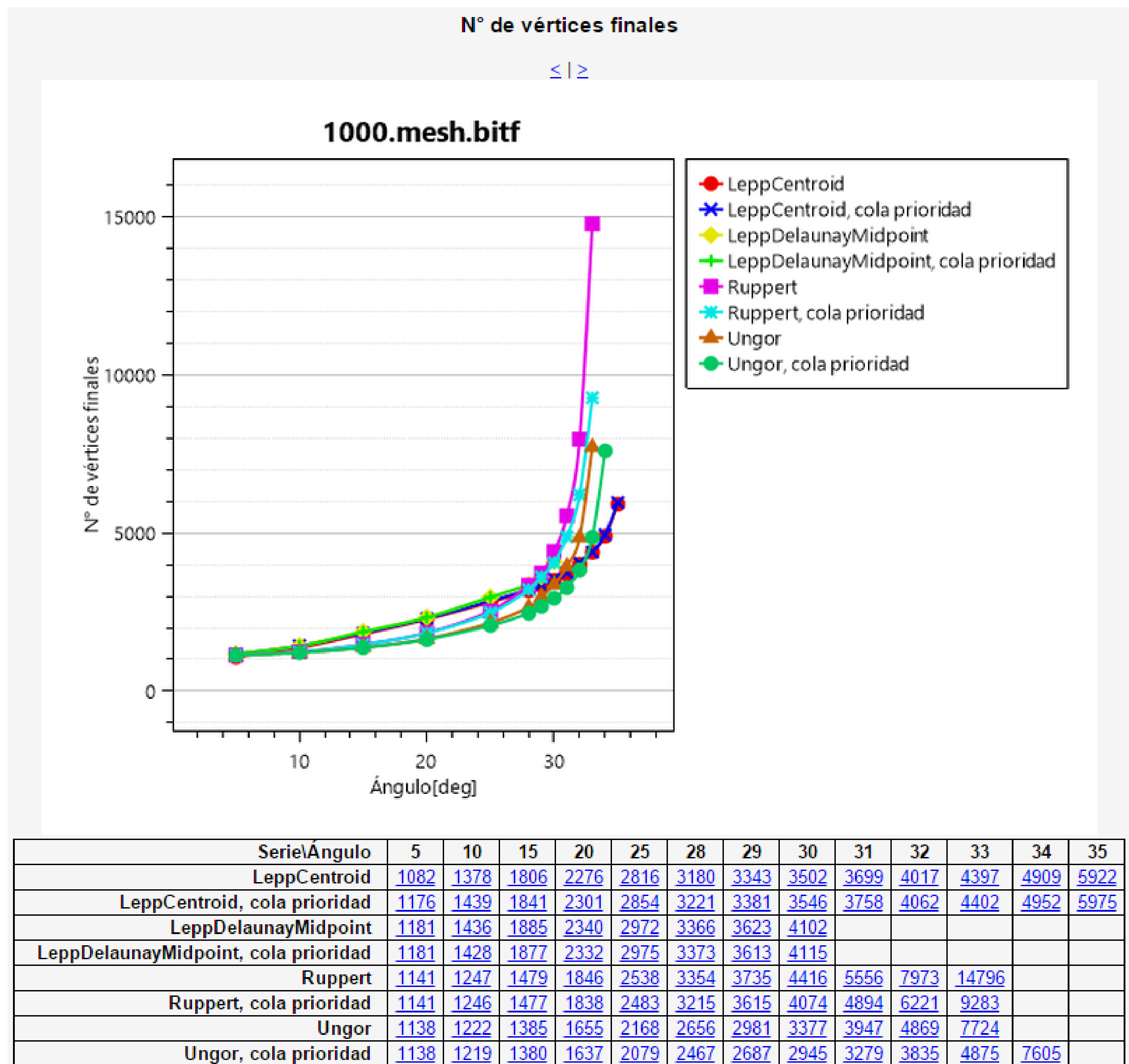
La triangulación de entrada se determinó a partir de un archivo .bitf de 1000 vértices. Las opciones de ejecución consistieron en las versiones con y sin ordenamiento de los triángulos a procesar de los algoritmos «Lepp-Centroide», «Lepp-Punto medio arista terminal», «Ruppert» y «Üngor». El orden de procesamiento de los triángulos de mala calidad (o priorización), se lleva a cabo utilizando «Colas de Prioridad», cuya llave de ordenamiento corresponde al ángulo interior mínimo de los triángulos, en lugar de «Colas Fifo». Algunos resultados importantes obtenidos son los mostrados en las siguientes figuras (6.9 y 6.10), que contienen gráficos y sus correspondientes tablas, y en las cuales una celda en blanco implica la «no convergencia del algoritmo» en esas condiciones de exigencia.



Serie\Ángulo	5	10	15	20	25	28	29	30	31	32	33	34	35
LeppCentroid	1.9	4.9	9.9	15.9	21.9	25.9	26.9	27.9	33.9	38.9	39.9	43.9	55.8
LeppCentroid, cola prioridad	0.9	5.9	10.9	16.9	24.9	29.9	30.9	36.9	36.9	40.9	46.9	55.8	71.8
LeppDelaunayMidpoint	1.9	5.9	12.9	24.9	38.9	51.9	44.9	55.8					
LeppDelaunayMidpoint, cola prioridad	1.9	4.9	9.9	14.9	21.9	27.9	29.9	55.8					
Ruppert	1.9	2.9	5.9	10.9	21.9	33.9	41.9	53.8	70.8	204.6	309.4		
Ruppert, cola prioridad	1.9	3.9	6.9	12.9	22.9	35.9	41.9	48.9	61.8	81.8	138.7		
Ungor	0.9	2.9	4.9	8.9	16.9	23.9	27.9	33.9	45.9	54.8	96.8		
Ungor, cola prioridad	0.9	2.9	5.9	8.9	15.9	20.9	27.9	30.9	36.9	50.9	59.8	103.8	

Figura 6.9: Gráfico de resultado: Tiempo total de procesamiento para series múltiples, con y sin ordenamiento de triángulos.

Debido al número de series de datos consideradas, la visualización de datos (Fig.6.9) se ve un tanto desfavorecida y presenta un nuevo problema. Las curvas se interponen unas a otras, aunque en definitiva, sí se pueden observar comportamientos sobresalientes. En la siguiente subsección, se aborda el problema de sobreposición, sin embargo, por última vez y para seguir con la visión más amplia de comparaciones, se listan todas en la siguiente figura de resultados.



Serie\Ángulo	5	10	15	20	25	28	29	30	31	32	33	34	35
LeppCentroid	1082	1378	1806	2276	2816	3180	3343	3502	3699	4017	4397	4909	5922
LeppCentroid, cola prioridad	1176	1439	1841	2301	2854	3221	3381	3546	3758	4062	4402	4952	5975
LeppDelaunayMidpoint	1181	1436	1885	2340	2972	3366	3623	4102					
LeppDelaunayMidpoint, cola prioridad	1181	1428	1877	2332	2975	3373	3613	4115					
Ruppert	1141	1247	1479	1846	2538	3354	3735	4416	5556	7973	14796		
Ruppert, cola prioridad	1141	1246	1477	1838	2483	3215	3615	4074	4894	6221	9283		
Ungor	1138	1222	1385	1655	2168	2656	2981	3377	3947	4869	7724		
Ungor, cola prioridad	1138	1219	1380	1637	2079	2467	2687	2945	3279	3835	4875	7605	

Figura 6.10: Gráfico de resultado: Número de vértices resultantes para series múltiples, con y sin ordenamiento de triángulos.

A partir de estas figuras se pueden establecer algunas conclusiones. Comenzando por la hipótesis, se puede corroborar el enunciado al observar que el comportamiento del algoritmo «Lepp-Centroide» efectivamente no se ve favorecido por el orden de procesamiento de los triángulos, y siendo aun más estricto, perjudica levemente los resultados al producir una

triangulación resultante con un mayor número de vértices respecto a su versión «FIFO». Por otro lado, y como se esperaba, las series basadas en Üngor y Ruppert sí se ven a afectadas significativamente por el tipo de cola utilizada, como se verá más detalladamente en 6.4 a continuación.

6.4. Afinamiento de visualización y comparaciones «uno a uno»

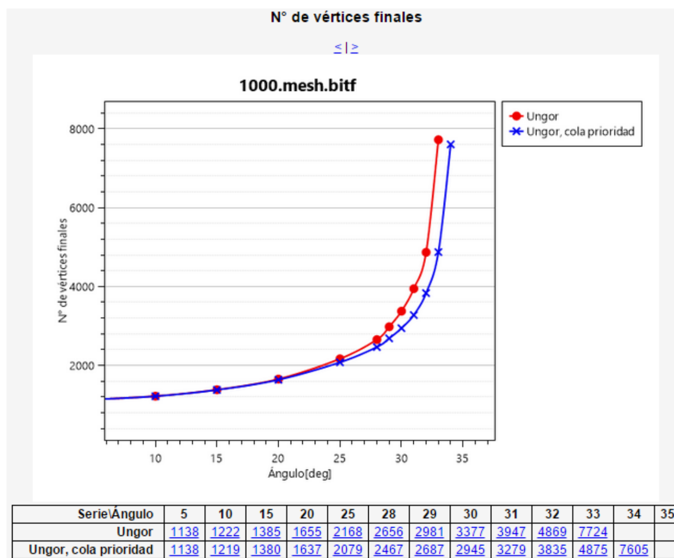
Al contar con una herramienta de producción de resultados masivos, se manifiestan nuevamente problemas relacionados al tópico «sobre la correcta visualización de volúmenes de datos». En el apartado 5.2.4, sobre «rasterización» de triangulaciones se propuso, como mejora opcional de visualización, cambiar las dimensiones de la imagen resultante y opacidad de las líneas dibujadas para poder ver triangulaciones con gran número de triángulos de manera más apropiada. El problema de fondo consiste en que no hay espacio suficiente para mostrar todo lo que se pretende. La solución para este caso de amontonamiento de curvas, fue ofrecer al usuario la posibilidad de generar reportes considerando series de datos a elección.

Los archivos de resultados «xml» ya poseen todos los datos de ejecuciones de refinamientos, que incluyen todas las series. El usuario puede elegir directamente qué series deben ser consideradas al generar el reporte de visualización.

Serie	Variación ángulo	Variación entrada	SerieKey
☑ LeppCentroid	Sí	No	0.1.0.3.0
☑ LeppCentroid, cola prioridad	Sí	No	0.1.1.3.1
☑ LeppDelaunayMidpoint	Sí	No	0.1.1.2.0
☑ LeppDelaunayMidpoint, cola prioridad	Sí	No	0.1.1.2.1
☑ Ruppert	Sí	No	0.1.1.1.0
☑ Ruppert, cola prioridad	Sí	No	0.1.1.1.1
☑ Ungor	Sí	No	0.1.1.4.0
☑ Ungor, cola prioridad	Sí	No	0.1.1.4.1

Figura 6.11: Elección de series para producción de «Reporte de Visualización de Resultados».

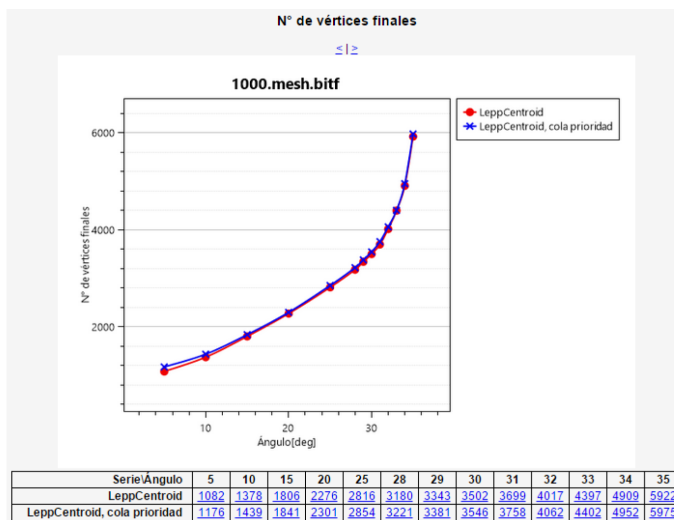
De esta manera es posible crear distintas visualizaciones donde lo que se desea comparar es claramente distinguible. A modo de muestra, se adjuntan gráficos y tablas comparativas producidos al elegir combinaciones de solamente 2 series.



Serie\Ángulo	5	10	15	20	25	28	29	30	31	32	33	34	35
Ungor	1138	1222	1385	1655	2168	2656	2981	3377	3947	4869	7724		
Ungor, cola prioridad	1138	1219	1380	1637	2079	2467	2687	2945	3279	3835	4875	7605	

Figura 6.12: Resultados comparativos de algoritmo de Üngor, con y sin prioridad de procesamiento.

En este caso, se hace manifiesto que el algoritmo de Üngor se ve favorecido por el ordenamiento de los triángulos a procesar. De manera similar, para el algoritmo Lepp-Centroide, se obtiene la siguiente figura comparativa:

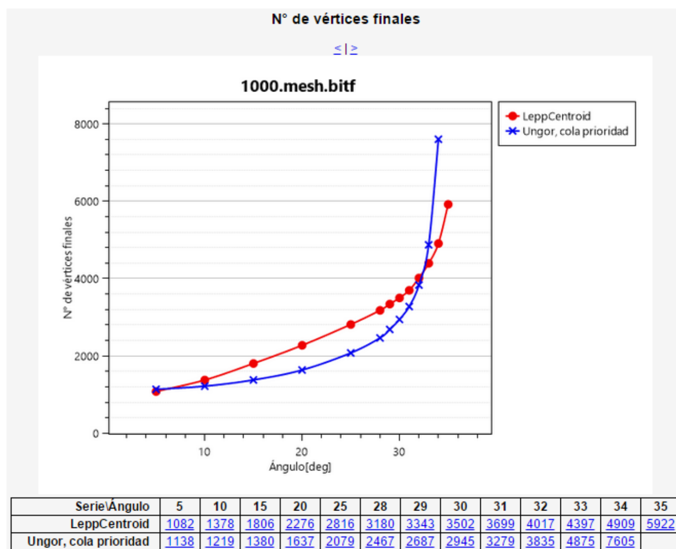


Serie\Ángulo	5	10	15	20	25	28	29	30	31	32	33	34	35
LeppCentroid	1082	1378	1806	2276	2816	3180	3343	3502	3699	4017	4397	4909	5922
LeppCentroid, cola prioridad	1176	1439	1841	2301	2854	3221	3381	3546	3758	4062	4402	4952	5975

Figura 6.13: Resultados comparativos de algoritmo Lepp-Centroide, con y sin prioridad de procesamiento.

De acuerdo a este gráfico, no hay una diferencia significativa debido al impacto de ordenar los elementos según calidad, versus procesarlos sin ordenamiento en el algoritmo Lepp-Centroide.

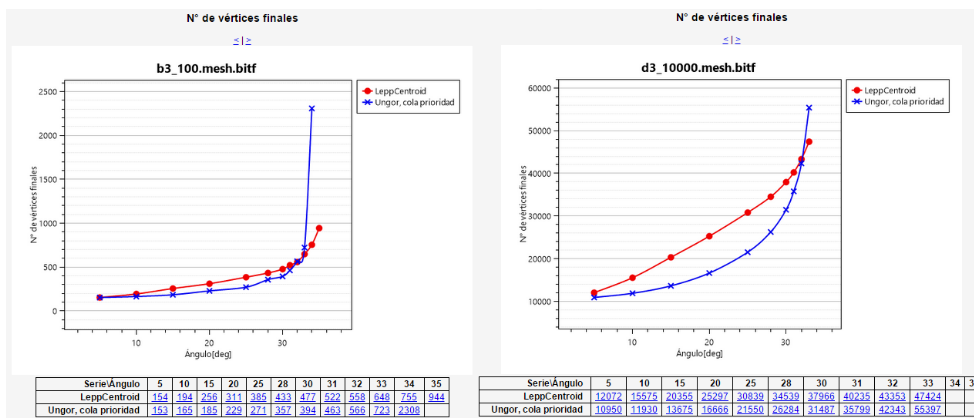
Retomando la intención de establecer una comparación entre algoritmos para comprobar la hipótesis, y considerando sólo los casos relevantes a esta, se obtiene:



Serie\Ángulo	5	10	15	20	25	28	29	30	31	32	33	34	35
LeppCentroid	1082	1378	1806	2276	2816	3180	3343	3502	3699	4017	4397	4909	5922
Ungor, cola prioridad	1138	1219	1380	1637	2079	2467	2687	2945	3279	3835	4875	7605	

Figura 6.14: Resultados comparativos de algoritmo de Üngor con ordenamiento vs Lepp-Centroide sin ordenamiento.

De acuerdo a esta última figura, existe un valor para el ángulo exigido, en el cual los algoritmos producen resultados prácticamente similares. Otros refinamientos fueron aplicados a distintas triangulaciones (Fig. 6.15), y por inspección, se observa que dicho valor del ángulo está consistentemente entre 32° y 33°. Si la exigencia de ángulo es menor a este valor, la indicación sería «preferir algoritmo de Üngor», y para exigencias mayores, se obtienen mejores resultados utilizando «Lepp-Centroide».



Serie\Ángulo	5	10	15	20	25	28	30	31	32	33	34	35
LeppCentroid	154	194	256	311	385	433	477	522	558	648	755	944
Ungor, cola prioridad	153	165	185	229	271	357	394	463	566	723	2308	

Serie\Ángulo	5	10	15	20	25	28	30	31	32	33	34	35
LeppCentroid	12072	15575	20355	25297	30839	34539	37966	40235	43353	47424		
Ungor, cola prioridad	10950	11930	13675	16666	21550	26284	31487	35799	42343	55397		

Figura 6.15: Resultados comparativos sobre otras triangulaciones (100 vértices a la izquierda, 1000 vértices a la derecha), de algoritmo de Üngor con ordenamiento vs Lepp-Centroide sin ordenamiento.

Cabe mencionar que estos valores obtenidos experimentalmente sólo ilustran el comportamiento para un determinado tipo de triangulación, donde los puntos están distribuidos de manera aleatoria. Para casos de mallas específicas el comportamiento varía.

7. Conclusión

7.1. Aplicación implementada y resultados

La aplicación desarrollada, junto a las bibliotecas y módulos relacionados, cumplen los objetivos anunciados tanto en el preámbulo de este documento, como los considerados posteriormente en los experimentos.

En primera instancia, las mediciones de rendimiento reflejan una mejora significativa, y un manejo de memoria que corrige efectivamente los problemas de las implementaciones anteriores.

BatchRefinement ofrece una amplia gama de utilidades, todas enfocadas en facilitar el acceso del usuario a realizar observaciones del proceso y resultado de refinamientos de triangulaciones Delaunay de manera intuitiva. La capacidad de ejecuciones por lotes y de reportes de visualización de resultados html, presentan una potente herramienta de análisis y estudio del comportamiento de los algoritmos de refinamiento. Los reportes html son portables, y pueden ser compartidos o publicados directamente en internet, siempre que se suban junto a sus imágenes, incentivando a compartir no sólo resultados, si no que también configuraciones de los experimentos.

De manera adicional, funcionalidades individuales como la «rasterización» y de triangulaciones, y conversión de formatos de triangulaciones complementan el entorno de trabajo.

Las implementaciones de los algoritmos estudiados no son un reflejo perfecto de los originales, sin embargo sí son un acercamiento que mantiene las ideas centrales de éstos.

Sobre los resultados obtenidos de los distintos algoritmos implementados:

1. El algoritmo de Üngor es considerado como un buen exponente en cuanto a que obtiene buenos resultados con relativamente pocos vértices adicionales.

2. Si el requerimiento es realizar refinamientos de mayores exigencias, entonces es preferible utilizar el algoritmo «Lepp-Centroide», el cual no sólo presenta un mejor comportamiento asintótico, si no que incluso presenta un «ángulo de convergencia máximo de exigencia» mayor. Si bien estos resultados ya no son novedad dentro de los trabajos de memoria de esta línea [3, 4], la herramienta actual permite reproducir fácilmente dichos experimentos incluyendo variaciones, y tener de manera inmediata comparaciones según se necesiten.

3. Las implementaciones actuales del algoritmo de Ruppert y «Lepp-PuntoMedio» no presentaron comportamientos sobresalientes en los resultados de las pruebas.

7.2. Trabajo futuro

Durante el transcurso del trabajo surgieron nuevas ideas y necesidades. Debido a lo acotado del tiempo de desarrollo, y a un proceso de priorización de lo relevante para presentar en esta ocasión, quedaron algunos temas pendientes, los cuales son buenos candidatos para esfuerzos futuros.

1. Incrementar el número de opciones para configuraciones iniciales de algoritmos, principalmente incluyendo nuevos métodos de corrección de aristas «encroached».
2. Extender la lista de métricas y resultados relevantes mostrados en los informes. Esto incluye, mostrar largo promedio de Lepps, dimensiones de aristas más pequeñas post refinamientos, etc, y otros más estrechamente relacionados con las implementaciones, como por ejemplo, el uso de memoria.
3. Agregar una opción para incluir una imagen que resalte los triángulos de mala calidad iniciales, a los reportes de visualización de resultados html.
4. Mejorar el formato de los reportes incluyendo «saltos de páginas», o equivalente, dentro del html, para producir versiones imprimibles más estéticas y ordenadas.
5. Agregar soporte para producción de imágenes tipo SVG, que pueden resultar muy apropiadas en algunos casos.
6. Incluir algoritmos de refinamiento paralelizados.
7. Agregar funcionalidad para realizar cada configuracion de los algoritmos más de una vez.
8. Continuando el punto anterior, agregar una nueva funcionalidad que permita extraer datos estadísticos útiles, como el promedio, desviación estándar, varianza etc, para cada métrica.
9. Desvincular la creación de reportes html del framework .Net, utilizando sólo alternativas C++11 para crear una nueva biblioteca estática más independiente, similar a «C2M».

8. Bibliografía

Referencias

[1] J. Ruppert, "A new and simple algorithm for quality 2-dimensional mesh generation," in *SODA*, vol. 93, 1993, pp. 83–92.

[2] J. R. Shewchuk, "Delaunay refinement algorithms for triangular mesh generation," *Computational geometry*, vol. 22, no. 1, pp. 21–74, 2002.

[3] Á. M. Faúndez Reyes, "Marco de experimentación para algoritmos de refinamiento de triangulaciones en 2d," 2010, disponible en http://www.repositorio.uchile.cl/handle/2250/103862.

[4] F. D. Gallardo Palacios, "Software de comparación de algoritmos delaunay de refinamiento de triangulaciones," 2012, disponible en http://www.repositorio.uchile.cl/handle/2250/111303.

[5] P. Fleischmann, *Mesh generation for technology CAD in three dimensions.* na, 1999.

[6] M.-C. Rivara, "New longest-edge algorithms for the refinement and/or improvement of unstructured triangulations," *International journal for numerical methods in Engineering*, vol. 40, no. 18, pp. 3313–3324, 1997.

[7] A. Üngör, "Off-centers: A new type of steiner points for computing size-optimal quality-guaranteed delaunay triangulations," in *LATIN 2004: Theoretical Informatics.* Springer, 2004, pp. 152–161.

[8] R. Seidel, "A simple and fast incremental randomized algorithm for computing trapezoidal decompositions and for triangulating polygons," *Computational Geometry*, vol. 1, no. 1, pp. 51–64, 1991.

[9] J. R. Shewchuk, "A two-dimensional quality mesh generator and delaunay triangulators," *Computer Science Division University of California at Berkeley, Berkeley, California, http://www.cs.cmu.edu/quake/triangle.html*, pp. 94 720–1776, 1997.

[10] M.-C. Rivara, "Lepp-bisection algorithms, applications and mathematical properties," *Applied Numerical Mathematics*, vol. 59, no. 9, pp. 2218–2235, 2009.

[11] M.-C. Rivara and C. Calderon, "Lepp terminal centroid method for quality triangulation," *Computer-Aided Design*, vol. 42, pp. 58–66, 2010, disponible en http://www.repositorio.uchile.cl/handle/2250/125434.

[12] O. Devillers, S. Pion, and M. Teillaud, "Walking in a triangulation," *International Journal of Foundations of Computer Science*, vol. 13, no. 02, pp. 181–199, 2002.

[13] B. Stroustrup, *The C++ programming language.* Pearson Education, 2013.

[14] "Finding memory leaks using the crt library," [posted 2015-12-09]. [Online]. Available: https://msdn.microsoft.com/en-us/library/x98tx3cf(v=vs.120).aspx

[15] "Oxyplot," [posted 2015-10-20]. [Online]. Available: http://oxyplot.org/

[16] M.-C. Rivara, P. Rodriguez, R. Montenegro, and G. Jorquera, "Multithread parallelization of lepp-bisection algorithms," *Applied Numerical Mathematics*, vol. 62, no. 4, pp. 473–488, 2012.

9. Anexo

A. Código fuente

El código fuente será publicado en un repositorio de «Github», y estará disponible en 2016 en el siguiente enlace:

https://github.com/maxwills/batch-refinement

B. Descripción de formatos de archivos binarios

A continuación se detallan los formatos binarios utilizados, cuyos nombres listados proceden de la extensión de los archivos. Cabe señalar que la implementación está desarrollada en .Net, incluidos los procesos de almacenamiento y lectura de archivos en formatos binarios.

Para especificaciones y consideración de más bajo nivel («endianness», especificaciones de punto flotante, etc) , se debe consultar la documentación del framework.

B.1. Binmesh

Descripción simple del formato binario «Binmesh», o «Binary Mesh File».

Los bloques de bytes están listados en orden, y son continuos. Considera un encabezado para facilitar la distinción del formato, y adicionalmente tiene soporte de versiones. La versión actual del formato es la «1».

- 7 bytes: <char x 7> caracteres individuales y continuos ASCII, con el encabezado del archivo, y que se leen como «BINMESH»
- 4 bytes: <int> número entero de 32 bits que indica la versión del formato binmesh del archivo.

  La única versión soportada actualmente es la 1.
- 4 bytes: <int> número entero de 32 bits que indica el número de vértices "V".
- 4 bytes: <int> número entero de 32 bits que indica el número de triángulos "T".
- 4 bytes: <int> número entero de 32 bits que indica el número de restricciones "R".
- V x 16 bytes: bloque continuo que contiene la información de los "V" vértices {

  Cada vértice se compone de la siguiente distribución:

- 8 bytes: <double> número punto flotante de 64 bits, que contiene la coordenada x.
- 8 bytes: <double> número punto flotante de 64 bits, que contiene la coordenada y.

}

- T x 12 bytes: bloque continuo que contiene la información de los "T" triángulos{

  Cada triángulo se compone de la siguiente distribución:

  - 4 bytes: <int> número entero de 32 bits, que contiene el id del primer vértice del triángulo.
  - 4 bytes: <int> número entero de 32 bits, que contiene el id del segundo vértice del triángulo.
  - 4 bytes: <int> número entero de 32 bits, que contiene el id del tercer vértice del triángulo.

  }

- R x 8 bytes: bloque continuo que contiene la información de las "R" restricciones{

  Cada restricción, indica una "arista restringida", y se compone de la siguiente distribución.

  - 4 bytes: <int> número entero de 32 bits, que contiene el id del primer vértice de la arista.
  - 4 bytes: <int> número entero de 32 bits, que contiene el id del segundo vértice de la arista.

  }

El tamaño esperado de un archivo «binmesh» es de $21 + 16V + 12T + 8R$ bytes, donde V,T,R corresponden al número de vértices, triángulos, y aristas restringidas respectivamente.

B.2. Bitf

Descripción simple del formato binario «Bitf», o «Binary Initialized Triangulation File».

Los bloques de bytes están listados en orden, y son continuos. Considera un encabezado para facilitar la distinción del formato, y adicionalmente tiene soporte de versiones. La versión actual del formato es la «2».

- 7 bytes: <char x 7> caracteres individuales y continuos ASCII, con el encabezado del archivo, y que se leen como «BITRIAF»
- 4 bytes: <int> número entero de 32 bits que indica la versión del formato bitf del archivo.

  La versión descrita acá es la «2».
- 1 byte: <bool> valor booleano que indica si el archivo incluye explícitamente los «IDs» de los triángulos, o estos son deducidos según el orden (enumeración parte de 1).

  Este valor booleano es importante, y si es «true» o «false» influye en el tamaño del resto de los bloques.

  Si el valor es verdadero, entonces los triángulos y vértices incluyen un ID, y en cuyo caso, se debe considerar la lectura y escritura de este para mantener la integridad de los bloques.

  Si el valor es falso, entonces no se incluye dicha información y no se debe intentar leer o escribir, y en su lugar, se debe leer/escribir inmediatamente el próximo elemento.

  Los elementos a continuación, cuya presencia en el archivo dependen de este booleano, serán indicados encerrándolos con «[ ]», y es importante recordar que el bloque aporta o no al tamaño listado dependiendo del valor.
- 4 bytes: <int> número entero de 32 bits que indica el número de vértices "V".
- 4 bytes: <int> número entero de 32 bits que indica el número de triángulos "T".
- 4 bytes: <int> número entero de 32 bits que indica el número de restricciones "R".
- 8 bytes: <double> número punto flotante de 64 bits, que contiene la mínima coordenada x.
- 8 bytes: <double> número punto flotante de 64 bits, que contiene la mínima coordenada y.
- 8 bytes: <double> número punto flotante de 64 bits, que contiene la máxima coordenada x.
- 8 bytes: <double> número punto flotante de 64 bits, que contiene la máxima coordenada y.
- 8 bytes: <double> número punto flotante de 64 bits, que contiene la coordenada x del centro de la triangulación.
- 8 bytes: <double> número punto flotante de 64 bits, que contiene la coordenada y del centro de la triangulación.

- V x 16 bytes [ó V x 20 bytes]: bloque continuo que contiene la información de los "V" vértices {

  Cada vértice se compone de la siguiente distribución:

  - [4 byte: <int> indica el ID (enumeración parte de 1) del vértice.]
  - 8 bytes: <double> número punto flotante de 64 bits, que contiene la coordenada x.
  - 8 bytes: <double> número punto flotante de 64 bits, que contiene la coordenada y.

  }

- T x 36 bytes [ó T x 40 bytes]: bloque continuo que contiene la información de los "T" triángulos{

  Cada triángulo se compone de la siguiente distribución:

  - [4 byte: <int> indica el ID (enumeración parte de 1) del triángulo.]
  - 4 bytes: <int> número entero de 32 bits, que contiene el ID del primer vértice del triángulo.
  - 4 bytes: <int> número entero de 32 bits, que contiene el ID del segundo vértice del triángulo.
  - 4 bytes: <int> número entero de 32 bits, que contiene el ID del tercer vértice del triángulo.
  - 4 bytes: <int> número entero de 32 bits, que contiene el índice(dentro del triángulo) de la arista de la primera arista restringida, ó -1.
  - 4 bytes: <int> número entero de 32 bits, que contiene el índice(dentro del triángulo) de la arista de la segunda arista restringida, ó -1.
  - 4 bytes: <int> número entero de 32 bits, que contiene el índice(dentro del triángulo) de la arista de la primera arista restringida, ó -1.
  - 4 bytes: <int> número entero de 32 bits, que contiene el ID del triángulo vecino, por la arista del lado opuesto al primer vértice. Es -1 si no tiene vecino por ese lado.
  - 4 bytes: <int> número entero de 32 bits, que contiene el ID del triángulo vecino, por la arista del lado opuesto al segundo vértice. Es -1 si no tiene vecino por ese lado.
  - 4 bytes: <int> número entero de 32 bits, que contiene el ID del triángulo vecino, por la arista del lado opuesto al tercer vértice. Es -1 si no tiene vecino por ese lado.

  }

- R x 16 bytes: bloque continuo que contiene la información de las "R" restricciones{

  Cada restricción, indica una "arista restringida", y se compone de la siguiente distribución.

  - 4 bytes: <int> número entero de 32 bits, que contiene el id del primer vértice de la arista.
  - 4 bytes: <int> número entero de 32 bits, que contiene el id del segundo vértice de la arista.
  - 4 bytes: <int> número entero de 32 bits, que contiene el ID del primer triángulo adyacente. Es -1 si no tiene vecino por ese lado.
  - 4 bytes: <int> número entero de 32 bits, que contiene el ID del segundo triángulo adyacente. Es -1 si no tiene vecino por ese lado.

  }

El tamaño esperado de un archivo «bitf» es de:

- $72 + 20 \cdot V + 40 \cdot T + 16 \cdot R$ bytes, si el archivo incluye IDs. Donde V,T,R corresponden al número de vértices, triángulos, y aristas restringidas respectivamente.
- [$72+16 \cdot V+36 \cdot T+16 \cdot R$] bytes, si el archivo no incluye IDs. Donde V,T,R corresponden al número de vértices, triángulos, y aristas restringidas respectivamente]

C. Muestra de «Reporte de número de vértices variante» generado.

Se adjuntan a continuación las páginas del resumen, y los resultados para una sola serie. Para todos los detalles de ejecuciones, visitar el documento completo en:

http://users.dcc.uchile.cl/~mwillemb/memoria/inp/index.html

Página de resumen de resultados sobre reporte de número de vértices variantes con ángulo exigido de 30°

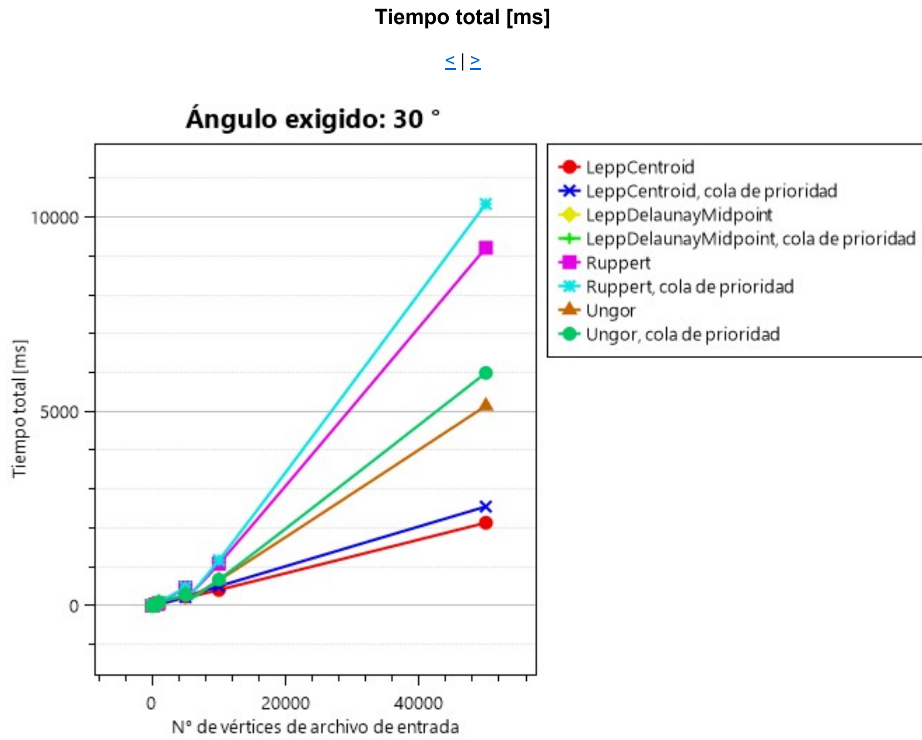
(Comienza en la siguiente página)

Resumen

Información general

Tiempo total [ms]

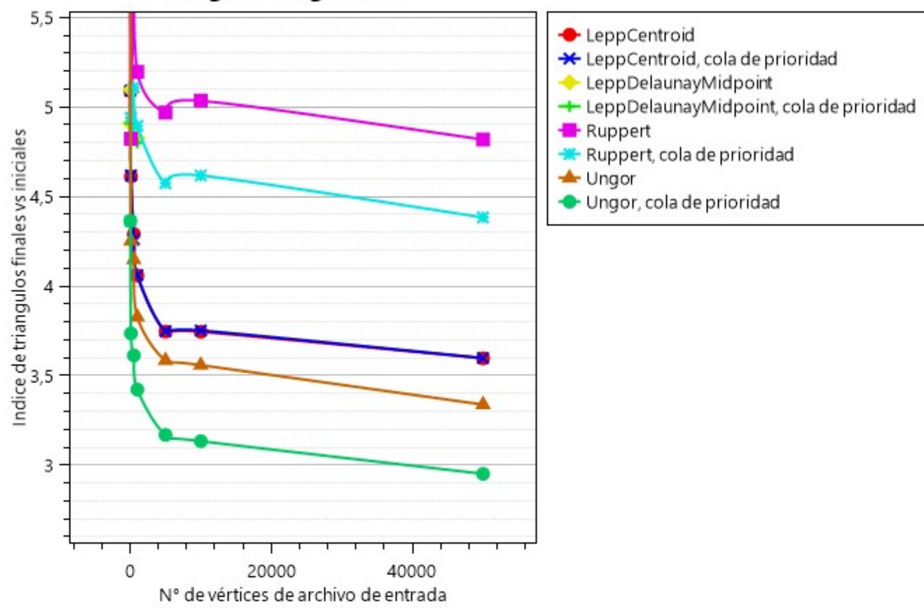
< | >



Serie\Vértices	10	100	500	1000	5000	10000	50000
LeppCentroid	15,6	15,6	15,6	46,8	234,3	406,2	2134,6
LeppCentroid, cola de prioridad	0	0	15,6	31,2	218,7	500,0	2551,7
LeppDelaunayMidpoint	0	0		31,2			
LeppDelaunayMidpoint, cola de prioridad	0	0		46,8			
Ruppert	0	0	31,2	62,5	468,7	1078,1	9215,1
Ruppert, cola de prioridad	15,6	0	25,3	93,7	484,3	1171,9	10342,8
Ungor	0	15,6	15,6	78,1	281,2	656,2	5143,4
Ungor, cola de prioridad	0	0	15,6	93,7	281,2	671,9	5992,9

Indice de triangulos finales vs iniciales

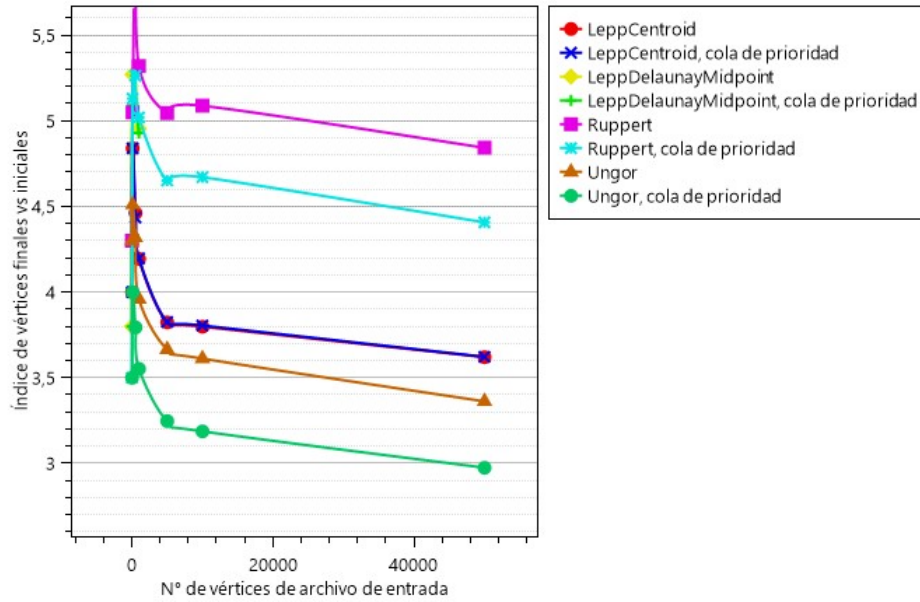
< | >



Serie\Vértices	10	100	500	1000	5000	10000	50000
LeppCentroid	5,0	4,6	4,2	4,0	3,7	3,7	3,5
LeppCentroid, cola de prioridad	5,0	4,6	4,2	4,0	3,7	3,7	3,5
LeppDelaunayMidpoint	4,9	5,0		4,8			
LeppDelaunayMidpoint, cola de prioridad	4,9	4,9		4,8			
Ruppert	5,5	4,8	5,8	5,1	4,9	5,0	4,8
Ruppert, cola de prioridad	4,3	4,9	5,1	4,8	4,5	4,6	4,3
Ungor	5,5	4,2	4,1	3,8	3,5	3,5	3,3
Ungor, cola de prioridad	4,3	3,7	3,6	3,4	3,1	3,1	2,9

Índice de vértices finales vs iniciales

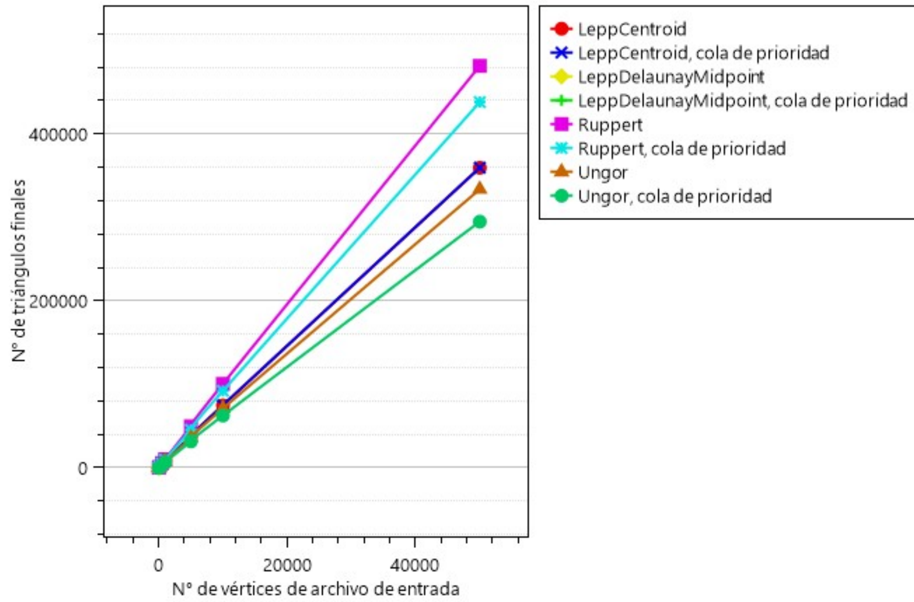
< | >



Serie\Vértices	10	100	500	1000	5000	10000	50000
LeppCentroid	4	4,8	4,4	4,1	3,8	3,7	3,6
LeppCentroid, cola de prioridad	4	4,8	4,4	4,1	3,8	3,8	3,6
LeppDelaunayMidpoint	3,8	5,2		4,9			
LeppDelaunayMidpoint, cola de prioridad	3,8	5,0		4,9			
Ruppert	4,3	5,0	5,9	5,3	5,0	5,0	4,8
Ruppert, cola de prioridad	3,5	5,1	5,2	5,0	4,6	4,6	4,4
Ungor	4,3	4,5	4,3	3,9	3,6	3,6	3,3
Ungor, cola de prioridad	3,5	4	3,7	3,5	3,2	3,1	2,9

N° de triángulos finales

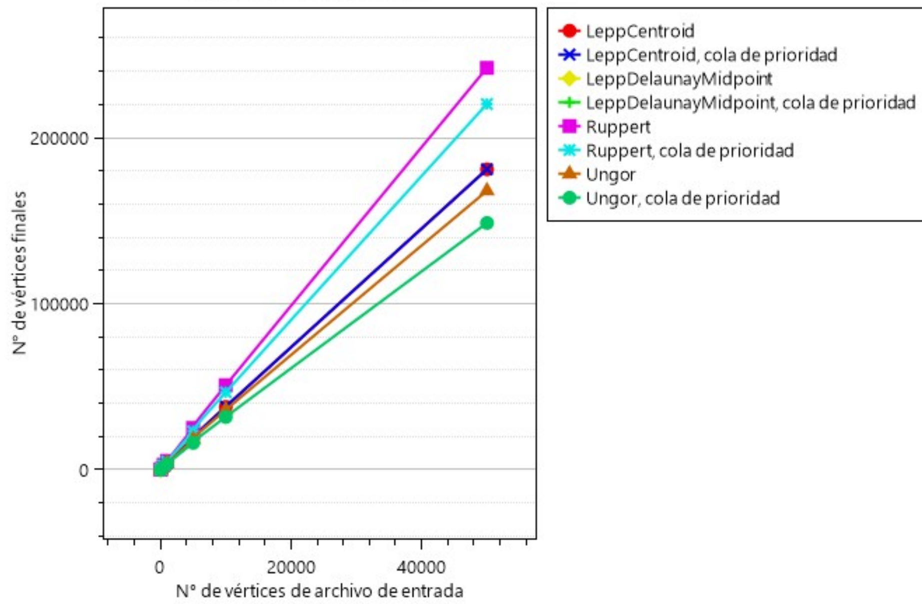
< | >



Serie\Vértices	10	100	500	1000	5000	10000	50000
LeppCentroid	56	858	4210	8035	37367	74805	359579
LeppCentroid, cola de prioridad	56	859	4180	8037	37386	74941	359741
LeppDelaunayMidpoint	54	948		9553			
LeppDelaunayMidpoint, cola de prioridad	54	913		9502			
Ruppert	61	897	5732	10284	49550	100551	481715
Ruppert, cola de prioridad	48	919	5012	9688	45623	92225	438237
Ungor	61	791	4070	7575	35759	71076	333755
Ungor, cola de prioridad	48	695	3545	6774	31610	62600	295101

N° de vértices finales

< | >



Serie\Vértices	10	100	500	1000	5000	10000	50000
LeppCentroid	40	484	2231	4194	19116	37983	181015
LeppCentroid, cola de prioridad	40	484	2218	4197	19129	38049	181101
LeppDelaunayMidpoint	38	527		4954			
LeppDelaunayMidpoint, cola de prioridad	38	509		4929			
Ruppert	43	505	2995	5320	25233	50880	242148
Ruppert, cola de prioridad	35	513	2632	5020	23261	46707	220379
Ungor	43	451	2159	3959	18327	36118	168102
Ungor, cola de prioridad	35	400	1897	3552	16235	31872	148739

Lepp más largo

< | >

Ángulo exigido: 30 °

LeppCentroid
LeppCentroid, cola de prioridad
LeppDelaunayMidpoint
LeppDelaunayMidpoint, cola de prioridad
Ruppert
Ruppert, cola de prioridad
Ungor
Ungor, cola de prioridad

Lepp más largo

N° de vértices de archivo de entrada

Serie\Vértices	10	100	500	1000	5000	10000	50000
LeppCentroid	4	10	9	9	16	18	19
LeppCentroid, cola de prioridad	4	9	11	12	18	17	20
LeppDelaunayMidpoint	5	9		14			
LeppDelaunayMidpoint, cola de prioridad	4	8		13			
Ruppert	0	0	0	0	0	0	0
Ruppert, cola de prioridad	0	0	0	0	0	0	0
Ungor	0	0	0	0	0	0	0
Ungor, cola de prioridad	0	0	0	0	0	0	0

Página de detalles de resultados para una sola serie, sobre reporte de número de vértices variantes con ángulo exigido de 30°

(Comienza en la siguiente página)

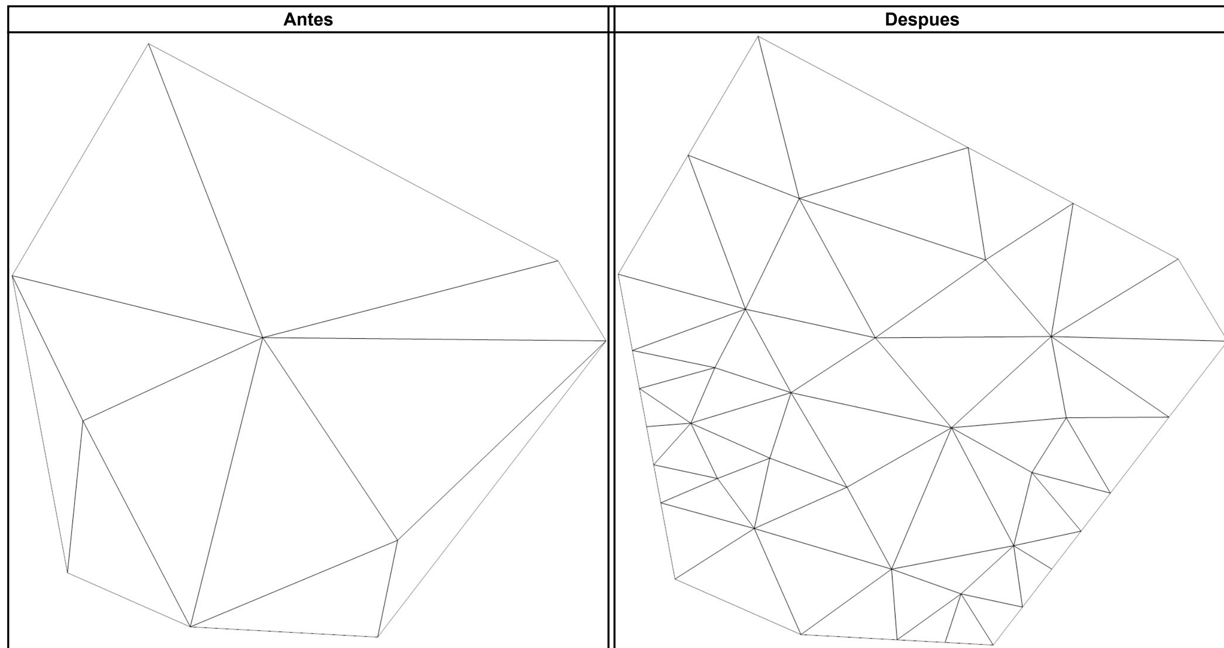
Detalles

LeppCentroid

Priorizacion de 'Encroached Edges'	Fifo
Preproceso	FixEncroachedEdges
Método de 'Steiner Point'	LeppCentroid
Priorización de triángulos	Fifo
Forma de inserción	FlipDiagonal

Resultados para ángulo = 30°

Antes	Despues

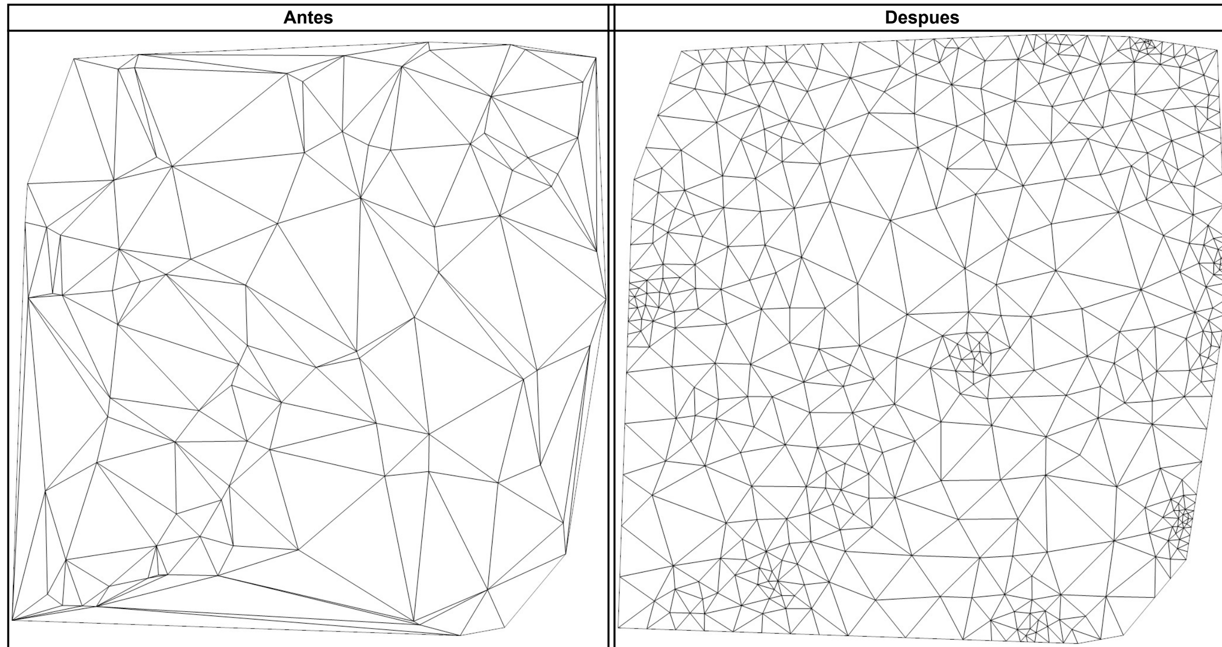


Archivo de triangulación completa original	C:\Max\Machines\VMShared\Trabajo Titulo\mesh_suite-fran develop-data folder\a4_10.mesh.bitf
Éxito	Sí
Divergente	No
Timeout de seguridad alcanzado	No
N° Vértices iniciales	10
N° Vértices finales	40
N° Triángulos iniciales	11
N° Triángulos finales	56
N° Aristas restringidas iniciales	7
N° Aristas restringidas finales	22
Ángulo interior más pequeño inical	8,42176320461515
Ángulo interior más pequeño final	31,3012841357159
Archivo de triangulacion simple inicial	a4_10.mesh_initial.stf
Archivo de triangulación simple final	a4_10.mesh_final_0.1.1.3.0_30.stf
Iteraciones de corrección de aristas encroached inicial	4
Iteraciones de corrección de AE post inicio	1
Iteraciones de refinamiento	25
Tiempo de corrección de aristas encroached inicial [ms]	0
Tiempo de corrección de AE post inicio[ms]	0

Tiempo de refinamiento [ms]	15,6259
Lepp más largo	4

Resultados para ángulo = 30°

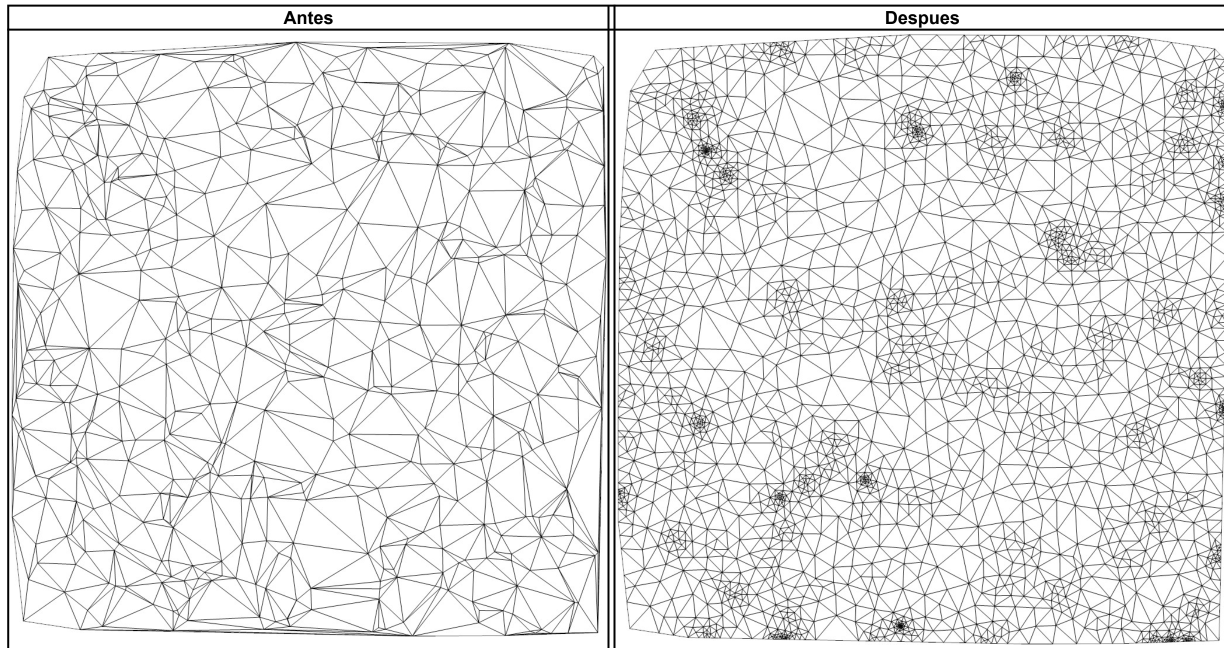
Antes	Despues



Archivo de triangulación completa original	C:\Max\Machines\VMShared\Trabajo Titulo\mesh_suite-fran develop-data folder\b_100.mesh.bitf
Éxito	Sí
Divergente	No
Timeout de seguridad alcanzado	No
N° Vértices iniciales	100
N° Vértices finales	484
N° Triángulos iniciales	186
N° Triángulos finales	858
N° Aristas restringidas iniciales	12
N° Aristas restringidas finales	108
Ángulo interior más pequeño inical	0,602323728240875
Ángulo interior más pequeño final	30,026442588548
Archivo de triangulacion simple inicial	b_100.mesh_initial.stf
Archivo de triangulación simple final	b_100.mesh_final_0.1.1.3.0_30.stf
Iteraciones de corrección de aristas encroached inicial	44
Iteraciones de corrección de AE post inicio	21
Iteraciones de refinamiento	319
Tiempo de corrección de aristas encroached inicial [ms]	0
Tiempo de corrección de AE post inicio[ms]	0
Tiempo de refinamiento [ms]	15,6268
Lepp más largo	10

Resultados para ángulo = 30°

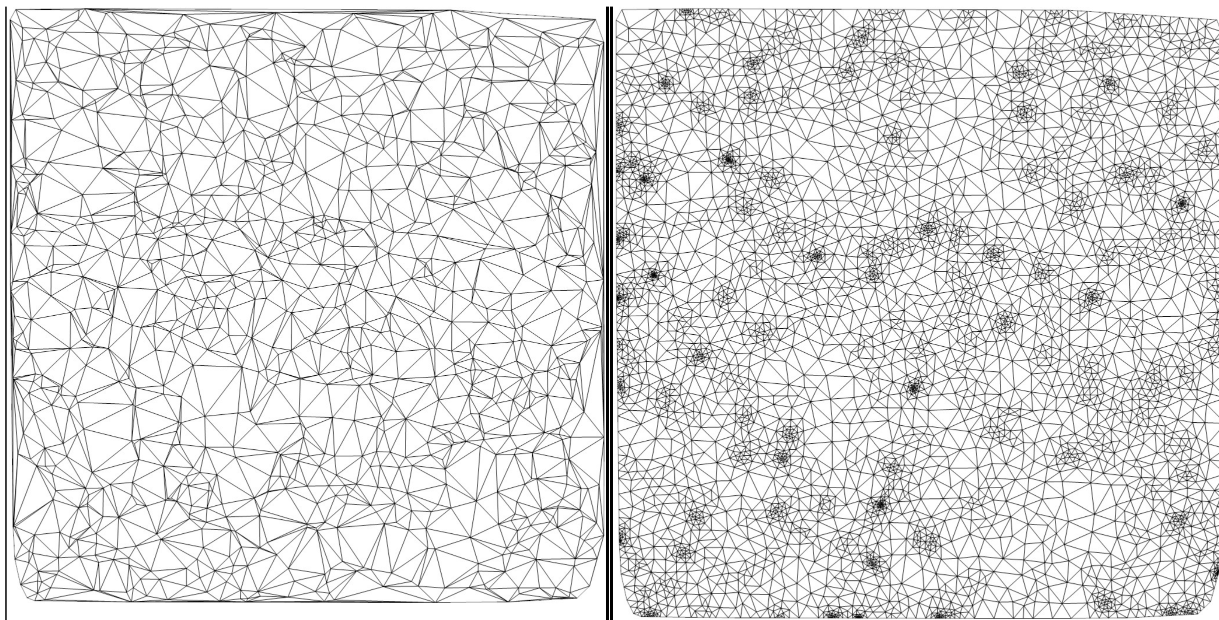
Antes	Despues



Archivo de triangulación completa original	C:\Max\Machines\VMShared\Trabajo Titulo\mesh_suite-fran develop-data folder\b_500.mesh.bitf
Éxito	Sí
Divergente	No
Timeout de seguridad alcanzado	No
N° Vértices iniciales	500
N° Vértices finales	2231
N° Triángulos iniciales	981
N° Triángulos finales	4210
N° Aristas restringidas iniciales	17
N° Aristas restringidas finales	250
Ángulo interior más pequeño inical	0,199042127522343
Ángulo interior más pequeño final	30,0065313955497
Archivo de triangulacion simple inicial	b_500.mesh_initial.stf
Archivo de triangulación simple final	b_500.mesh_final_0.1.1.3.0_30.stf
Iteraciones de corrección de aristas encroached inicial	134
Iteraciones de corrección de AE post inicio	37
Iteraciones de refinamiento	1560
Tiempo de corrección de aristas encroached inicial [ms]	0
Tiempo de corrección de AE post inicio[ms]	0
Tiempo de refinamiento [ms]	15,6251
Lepp más largo	9

Resultados para ángulo = 30°

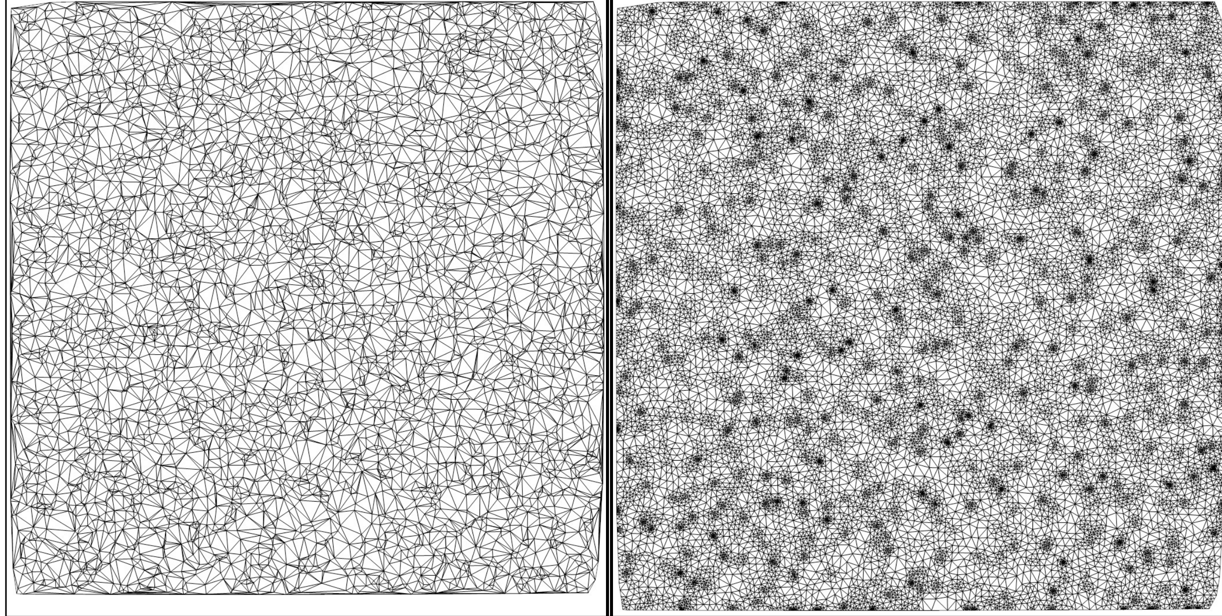
Antes	Despues



Archivo de triangulación completa original	C:\Max\Machines\VMShared\Trabajo Titulo\mesh_suite-fran develop-data folder\c_1000.mesh.bitf
Éxito	Sí
Divergente	No
Timeout de seguridad alcanzado	No
N° Vértices iniciales	1000
N° Vértices finales	4194
N° Triángulos iniciales	1979
N° Triángulos finales	8035
N° Aristas restringidas iniciales	19
N° Aristas restringidas finales	351
Ángulo interior más pequeño inical	0,102541083046465
Ángulo interior más pequeño final	30,0150735330707
Archivo de triangulacion simple inicial	c_1000.mesh_initial.stf
Archivo de triangulación simple final	c_1000.mesh_final_0.1.1.3.0_30.stf
Iteraciones de corrección de aristas encroached inicial	194
Iteraciones de corrección de AE post inicio	50
Iteraciones de refinamiento	2950
Tiempo de corrección de aristas encroached inicial [ms]	0
Tiempo de corrección de AE post inicio[ms]	0
Tiempo de refinamiento [ms]	46,8765
Lepp más largo	9

Resultados para ángulo = 30°

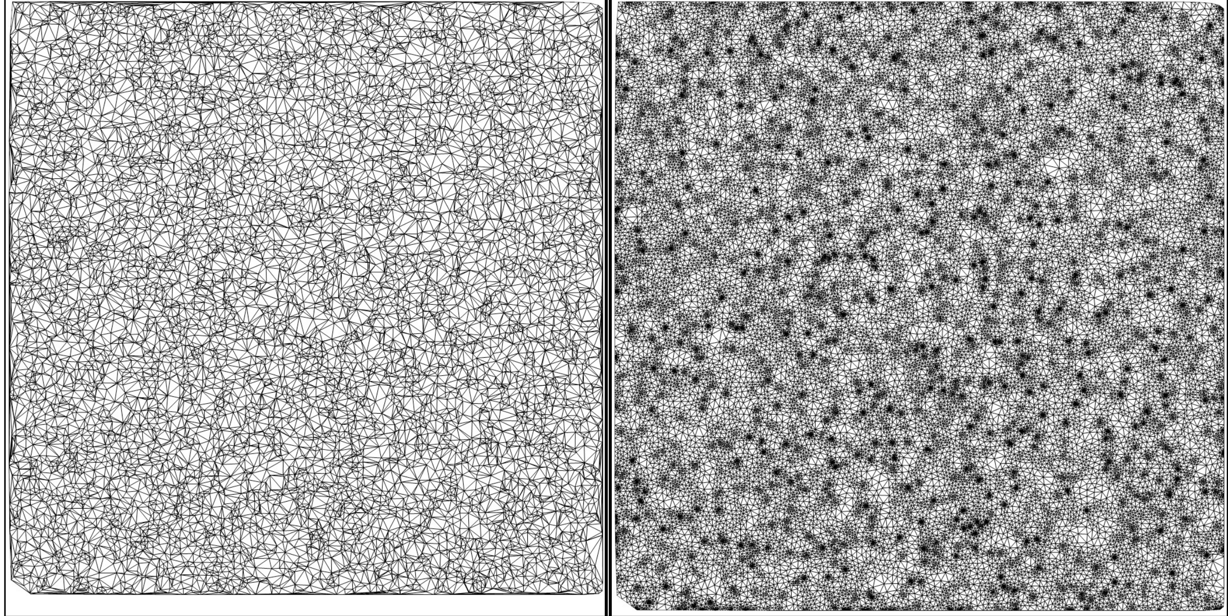
Antes	Despues



Archivo de triangulación completa original	C:\Max\Machines\VMShared\Trabajo Titulo\mesh_suite-fran develop-data folder\c5_5000.mesh.bitf
Éxito	Sí
Divergente	No
Timeout de seguridad alcanzado	No
N° Vértices iniciales	5000
N° Vértices finales	19116
N° Triángulos iniciales	9973
N° Triángulos finales	37367
N° Aristas restringidas iniciales	25
N° Aristas restringidas finales	863
Ángulo interior más pequeño inical	0,0130176850373396
Ángulo interior más pequeño final	30,0014758646553
Archivo de triangulacion simple inicial	c5_5000.mesh_initial.stf
Archivo de triangulación simple final	c5_5000.mesh_final_0.1.1.3.0_30.stf
Iteraciones de corrección de aristas encroached inicial	471
Iteraciones de corrección de AE post inicio	128
Iteraciones de refinamiento	13517
Tiempo de corrección de aristas encroached inicial [ms]	15,6233
Tiempo de corrección de AE post inicio[ms]	0
Tiempo de refinamiento [ms]	218,7598
Lepp más largo	16

Resultados para ángulo = 30°

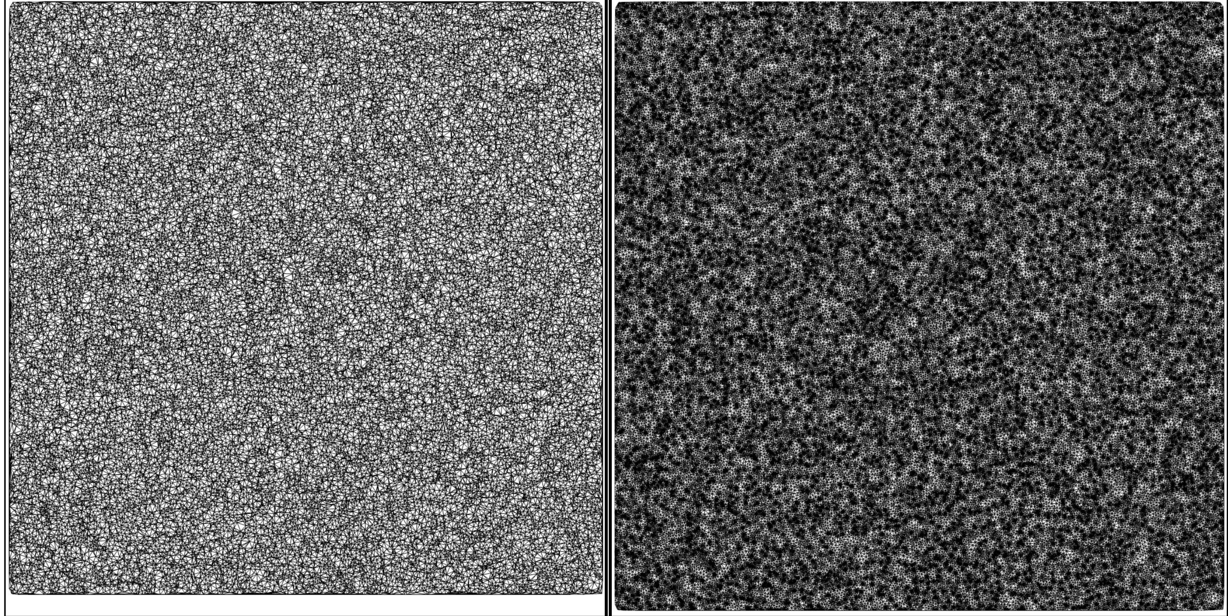
Antes	Despues



Archivo de triangulación completa original	C:\Max\Machines\VMShared\Trabajo Titulo\mesh_suite-fran develop-data folder\d2_10000.mesh.bitf
Éxito	Sí
Divergente	No
Timeout de seguridad alcanzado	No
N° Vértices iniciales	10000
N° Vértices finales	37983
N° Triángulos iniciales	19971
N° Triángulos finales	74805
N° Aristas restringidas iniciales	27
N° Aristas restringidas finales	1159
Ángulo interior más pequeño inical	0,00107866216588603
Ángulo interior más pequeño final	30,001698661704
Archivo de triangulacion simple inicial	d2_10000.mesh_initial.stf
Archivo de triangulación simple final	d2_10000.mesh_final_0.1.1.3.0_30.stf
Iteraciones de corrección de aristas encroached inicial	612
Iteraciones de corrección de AE post inicio	183
Iteraciones de refinamiento	27188
Tiempo de corrección de aristas encroached inicial [ms]	0
Tiempo de corrección de AE post inicio[ms]	0
Tiempo de refinamiento [ms]	406,2704
Lepp más largo	18

Resultados para ángulo = 30°

Antes	Despues



Archivo de triangulación completa original	C:\Max\Machines\VMShared\Trabajo Titulo\mesh_suite-fran develop-data folder\d3_50000.mesh.bitf
Éxito	Sí
Divergente	No
Timeout de seguridad alcanzado	No
N° Vértices iniciales	50000
N° Vértices finales	181015
N° Triángulos iniciales	99960
N° Triángulos finales	359579
N° Aristas restringidas iniciales	38
N° Aristas restringidas finales	2449
Ángulo interior más pequeño inical	0,0013946700227621
Ángulo interior más pequeño final	30,0003095345433
Archivo de triangulacion simple inicial	d3_50000.mesh_initial.stf
Archivo de triangulación simple final	d3_50000.mesh_final_0.1.1.3.0_30.stf
Iteraciones de corrección de aristas encroached inicial	1371
Iteraciones de corrección de AE post inicio	358
Iteraciones de refinamiento	129286
Tiempo de corrección de aristas encroached inicial [ms]	15,6238
Tiempo de corrección de AE post inicio[ms]	0
Tiempo de refinamiento [ms]	2119,0679
Lepp más largo	19

D. Muestra de «Reporte de ángulo exigido variante» generado.

Se adjuntan a continuación las páginas del resumen, y los resultados para una sola serie. Para todos los detalles de ejecuciones, visitar el documento completo en:

http://users.dcc.uchile.cl/∼mwillemb/memoria/ang/index.html

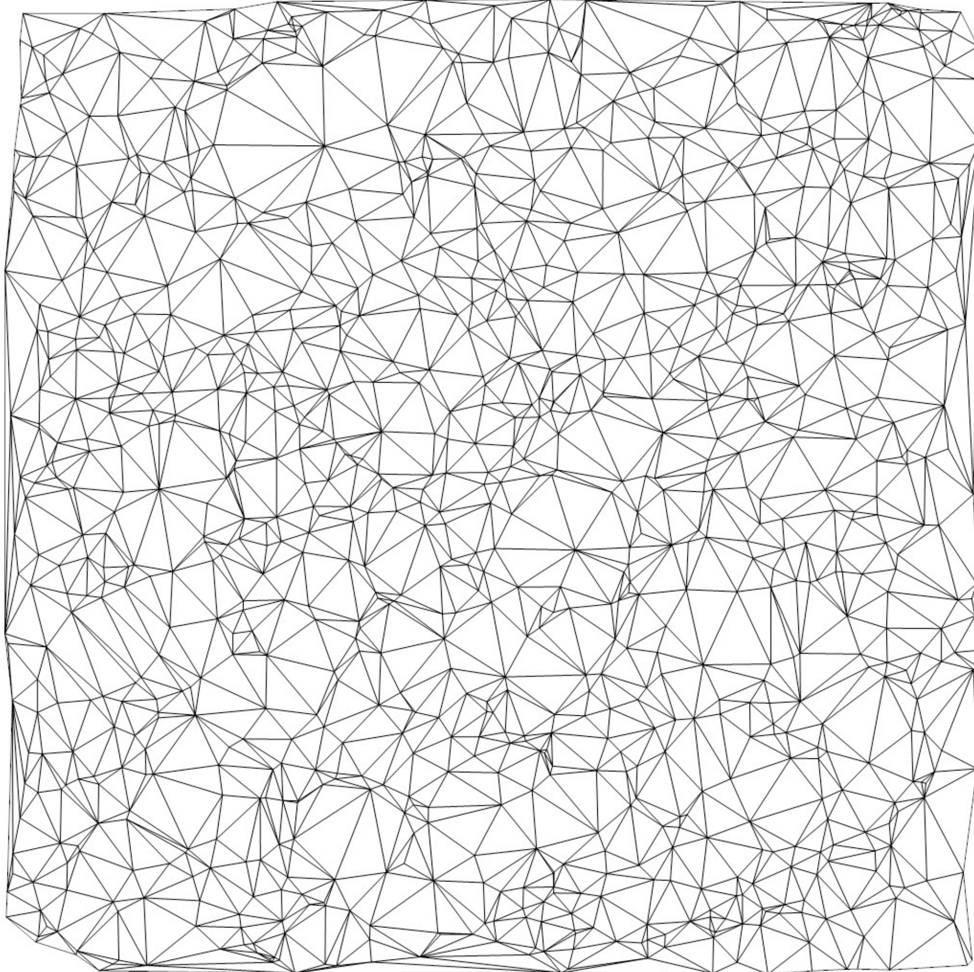
D.1. Página de resumen de resultados sobre reporte de ángulo variante para triangulación de 1000 vértices

(Comienza en la siguiente página)

Resumen

Información inicial

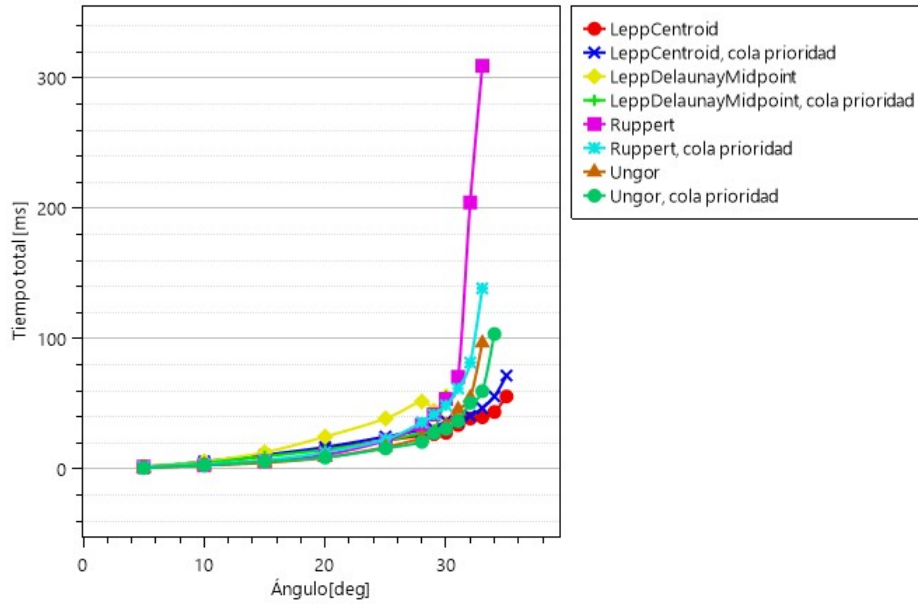
Triangulación inicial



Archivo de triangulación completa original	C:\Users\Max\Desktop\1000.mesh.bitf
Archivo de triangulacion simple inicial	1000.mesh_initial.stf
N° Aristas restringidas iniciales	43
N° Triángulos iniciales	1955
N° Vértices iniciales	1000
Ángulo interior más pequeño inical	1,02903606495574
Lepp más largo	9

Tiempo total [ms]

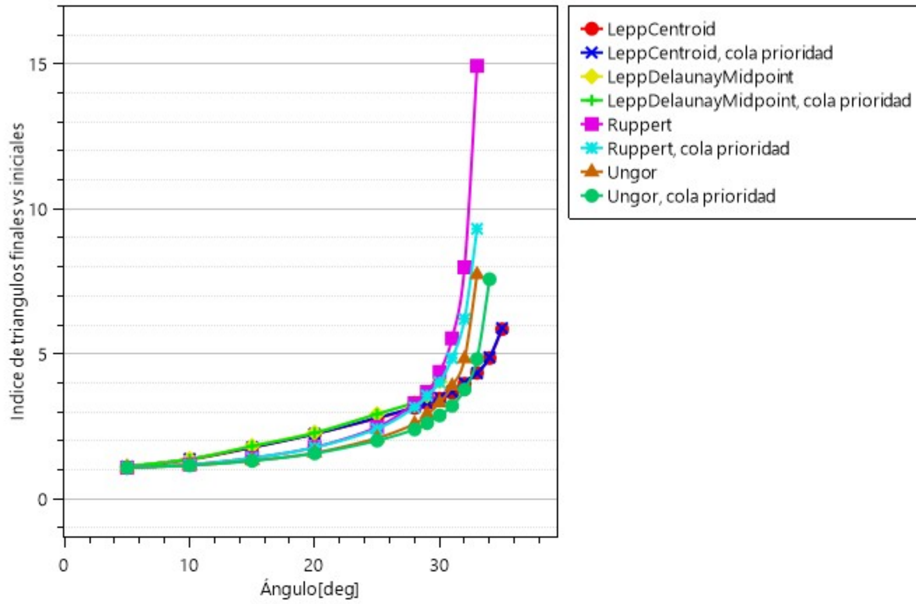
< | >



Serie\Ángulo	5	10	15	20	25	28	29	30	31	32	33	34	35
LeppCentroid	1,9	4,9	9,9	15,9	21,9	25,9	26,9	27,9	33,9	38,9	39,9	43,9	55,8
LeppCentroid, cola prioridad	0,9	5,9	10,9	16,9	24,9	29,9	30,9	36,9	36,9	40,9	46,9	55,8	71,8
LeppDelaunayMidpoint	1,9	5,9	12,9	24,9	38,9	51,9	44,9	55,8					
LeppDelaunayMidpoint, cola prioridad	1,9	4,9	9,9	14,9	21,9	27,9	29,9	55,8					
Ruppert	1,9	2,9	5,9	10,9	21,9	33,9	41,9	53,8	70,8	204,6	309,4		
Ruppert, cola prioridad	1,9	3,9	6,9	12,9	22,9	35,9	41,9	48,9	61,8	81,8	138,7		
Ungor	0,9	2,9	4,9	8,9	16,9	23,9	27,9	33,9	45,9	54,8	96,8		
Ungor, cola prioridad	0,9	2,9	5,9	8,9	15,9	20,9	27,9	30,9	36,9	50,9	59,8	103,8	

Indice de triangulos finales vs iniciales

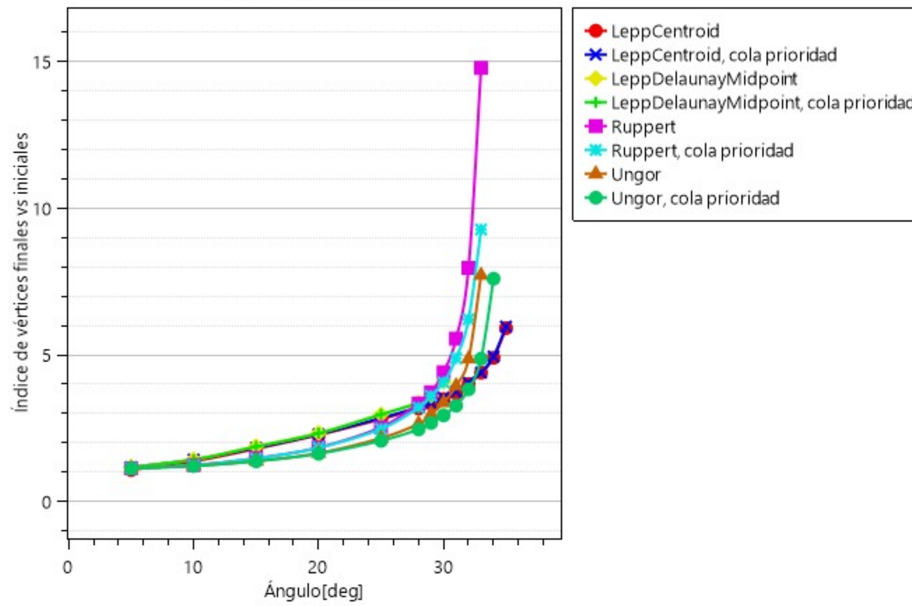
< | >



Serie\Ángulo	5	10	15	20	25	28	29	30	31	32	33	34	35
LeppCentroid	1,0	1,3	1,7	2,2	2,7	3,1	3,3	3,4	3,6	3,9	4,3	4,8	5,8
LeppCentroid, cola prioridad	1,1	1,3	1,7	2,2	2,8	3,1	3,3	3,4	3,7	4,0	4,3	4,8	5,8
LeppDelaunayMidpoint	1,1	1,3	1,8	2,2	2,9	3,3	3,5	4,0					
LeppDelaunayMidpoint, cola prioridad	1,1	1,3	1,8	2,2	2,9	3,3	3,5	4,0					
Ruppert	1,0	1,1	1,4	1,7	2,4	3,2	3,6	4,3	5,5	7,9	14,9		
Ruppert, cola prioridad	1,0	1,1	1,4	1,7	2,4	3,1	3,5	4,0	4,8	6,2	9,3		
Ungor	1,0	1,1	1,3	1,5	2,1	2,5	2,9	3,3	3,8	4,8	7,7		
Ungor, cola prioridad	1,0	1,1	1,3	1,5	2,0	2,4	2,6	2,8	3,2	3,7	4,8	7,5	

Índice de vértices finales vs iniciales

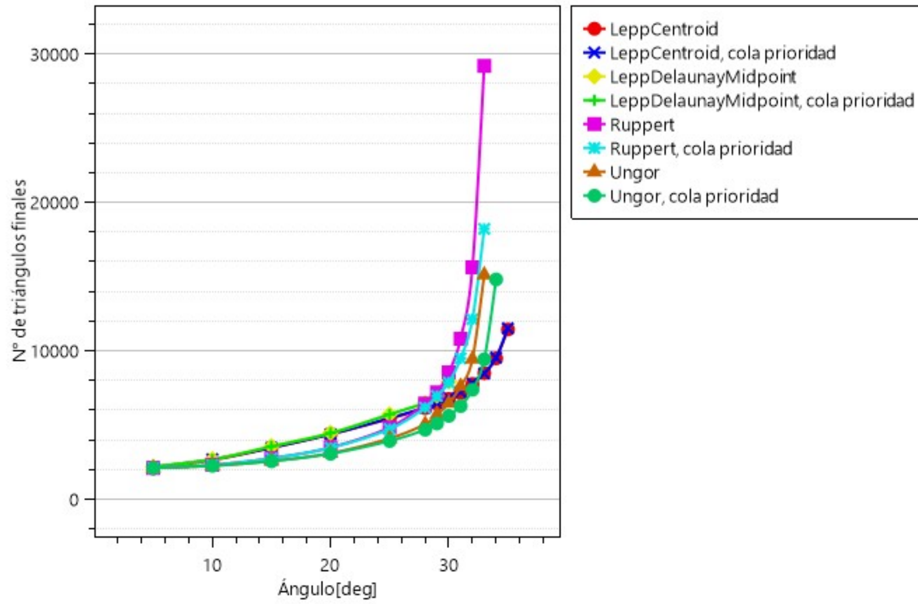
< | >



Serie\Ángulo	5	10	15	20	25	28	29	30	31	32	33	34	35
LeppCentroid	1,0	1,3	1,8	2,2	2,8	3,1	3,3	3,5	3,6	4,0	4,3	4,9	5,9
LeppCentroid, cola prioridad	1,1	1,4	1,8	2,3	2,8	3,2	3,3	3,5	3,7	4,0	4,4	4,9	5,9
LeppDelaunayMidpoint	1,1	1,4	1,8	2,3	2,9	3,3	3,6	4,1					
LeppDelaunayMidpoint, cola prioridad	1,1	1,4	1,8	2,3	2,9	3,3	3,6	4,1					
Ruppert	1,1	1,2	1,4	1,8	2,5	3,3	3,7	4,4	5,5	7,9	14,7		
Ruppert, cola prioridad	1,1	1,2	1,4	1,8	2,4	3,2	3,6	4,0	4,8	6,2	9,2		
Ungor	1,1	1,2	1,3	1,6	2,1	2,6	2,9	3,3	3,9	4,8	7,7		
Ungor, cola prioridad	1,1	1,2	1,3	1,6	2,0	2,4	2,6	2,9	3,2	3,8	4,8	7,6	

N° de triángulos finales

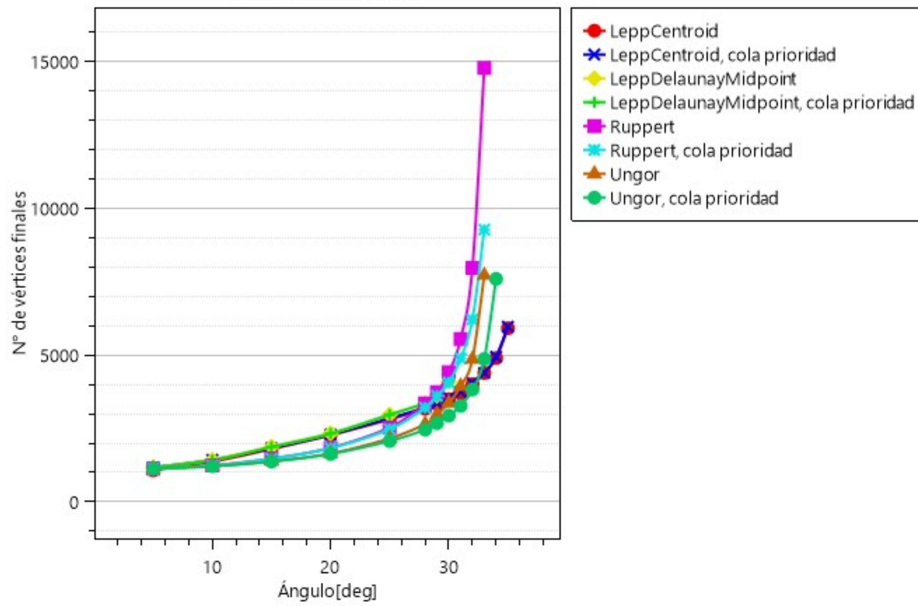
< | >



Serie\Ángulo	5	10	15	20	25	28	29	30	31	32	33	34	35
LeppCentroid	2093	2644	3468	4384	5436	6145	6462	6774	7161	7774	8507	9504	11455
LeppCentroid, cola prioridad	2187	2702	3489	4392	5475	6190	6502	6824	7240	7828	8484	9556	11525
LeppDelaunayMidpoint	2198	2696	3577	4476	5720	6490	6993	7933					
LeppDelaunayMidpoint, cola prioridad	2198	2680	3563	4459	5726	6504	6975	7962					
Ruppert	2118	2319	2770	3490	4842	6451	7206	8558	10823	15631	29209		
Ruppert, cola prioridad	2118	2317	2766	3473	4734	6179	6968	7875	9500	12139	18223		
Ungor	2112	2270	2583	3109	4108	5065	5706	6488	7618	9451	15129		
Ungor, cola prioridad	2112	2264	2573	3074	3936	4694	5120	5630	6290	7387	9441	14821	

N° de vértices finales

< | >



Serie\Ángulo	5	10	15	20	25	28	29	30	31	32	33	34	35
LeppCentroid	1082	1378	1806	2276	2816	3180	3343	3502	3699	4017	4397	4909	5922
LeppCentroid, cola prioridad	1176	1439	1841	2301	2854	3221	3381	3546	3758	4062	4402	4952	5975
LeppDelaunayMidpoint	1181	1436	1885	2340	2972	3366	3623	4102					
LeppDelaunayMidpoint, cola prioridad	1181	1428	1877	2332	2975	3373	3613	4115					
Ruppert	1141	1247	1479	1846	2538	3354	3735	4416	5556	7973	14796		
Ruppert, cola prioridad	1141	1246	1477	1838	2483	3215	3615	4074	4894	6221	9283		
Ungor	1138	1222	1385	1655	2168	2656	2981	3377	3947	4869	7724		
Ungor, cola prioridad	1138	1219	1380	1637	2079	2467	2687	2945	3279	3835	4875	7605	

Lepp más largo

< | >

1000.mesh.bitf

Serie\Ángulo	5	10	15	20	25	28	29	30	31	32	33	34	35
LeppCentroid	9	10	10	9	9	9	9	9	9	9	9	9	12
LeppCentroid, cola prioridad	9	9	9	9	9	9	9	9	9	9	9	9	9
LeppDelaunayMidpoint	8	8	10	10	10	10	10	13					
LeppDelaunayMidpoint, cola prioridad	8	8	10	10	10	10	10	10					
Ruppert	0	0	0	0	0	0	0	0	0	0	0		
Ruppert, cola prioridad	0	0	0	0	0	0	0	0	0	0	0		
Ungor	0	0	0	0	0	0	0	0	0	0	0		
Ungor, cola prioridad	0	0	0	0	0	0	0	0	0	0	0	0	

D.2. Página de detalles de resultados para una sola serie, sobre reporte de ángulo variante para triangulación de 1000 vértices

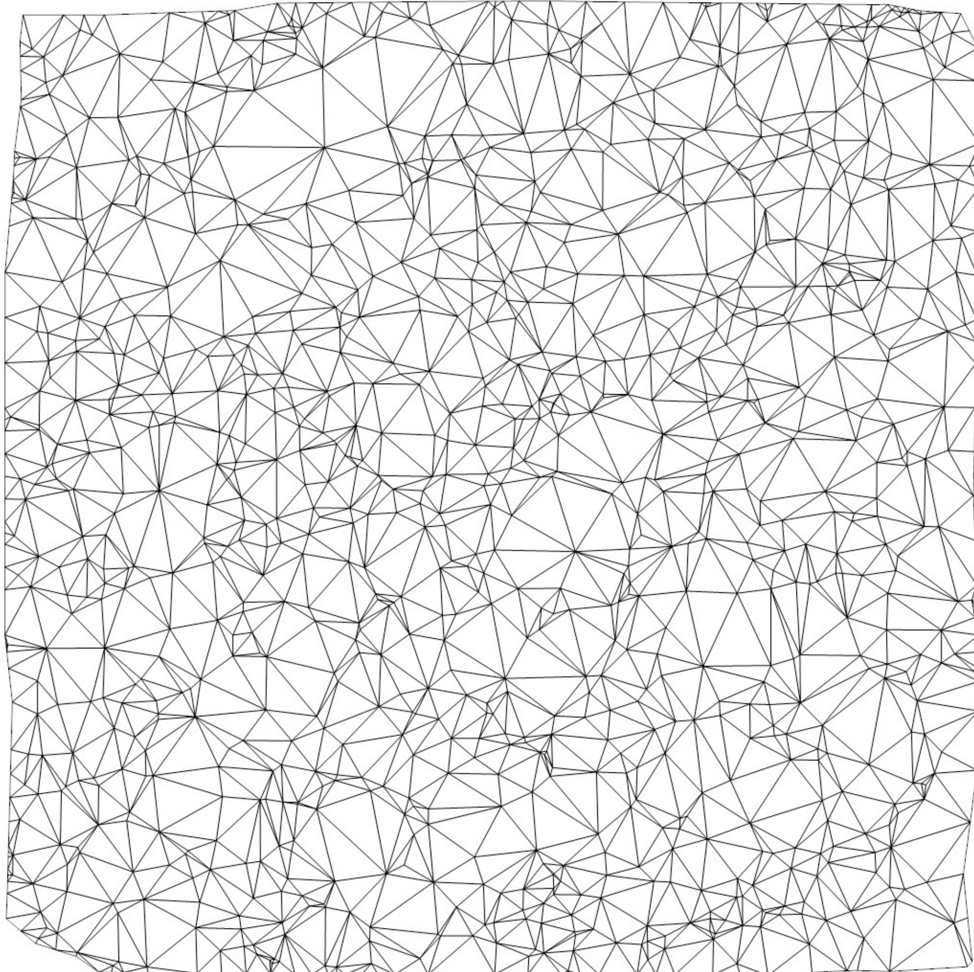
(Comienza en la siguiente página)

Detalles

Ungor

Priorizacion de 'Encroached Edges'	Fifo
Preproceso	FixEncroachedEdges
Método de 'Steiner Point'	Ungor
Priorización de triángulos	Fifo
Forma de inserción	FlipDiagonal

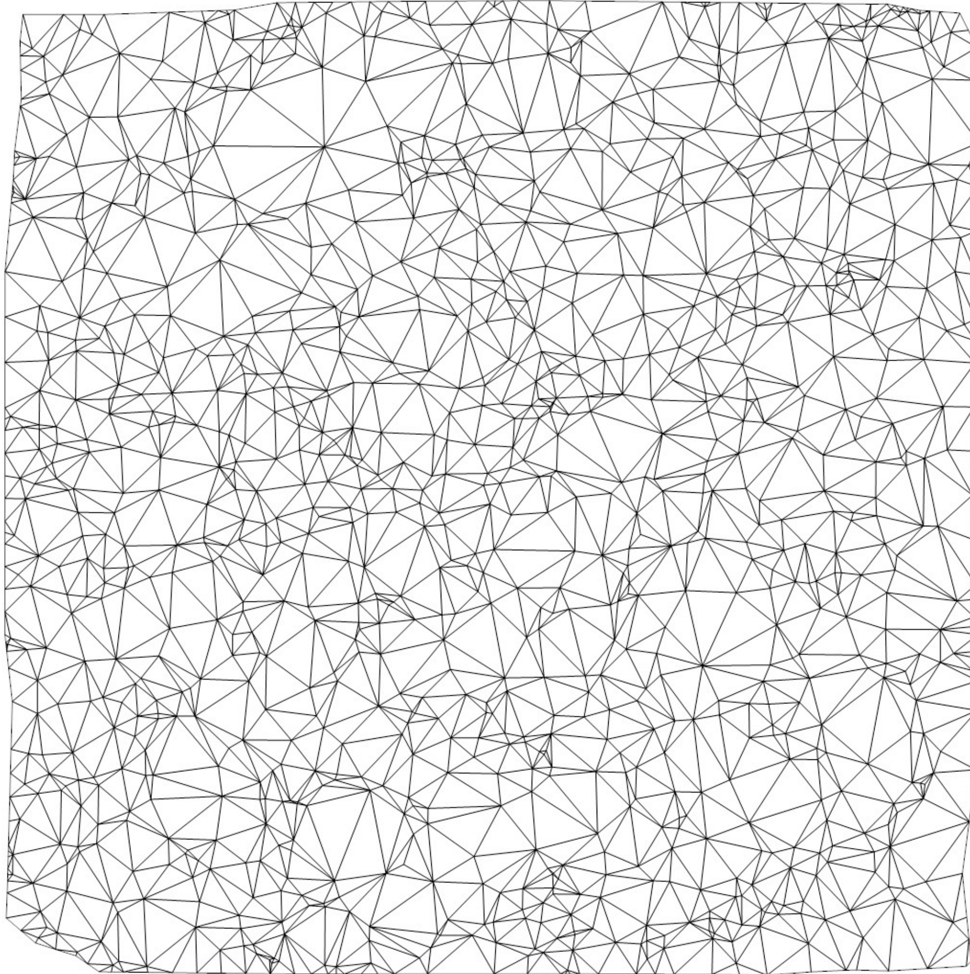
Resultados para ángulo = 5°



Archivo de triangulación completa original	C:\Users\Max\Desktop\1000.mesh.bitf
Éxito	Sí
Divergente	No
Timeout de seguridad alcanzado	No
N° Vértices iniciales	1000
N° Vértices finales	1138
N° Triángulos iniciales	1955
N° Triángulos finales	2112
N° Aristas restringidas iniciales	43
N° Aristas restringidas finales	162
	1,02903606495574

Ángulo interior más pequeño inical	
Ángulo interior más pequeño final	5,04198130564351
Archivo de triangulacion simple inicial	1000.mesh_initial.stf
Archivo de triangulación simple final	1000.mesh_final_0.1.1.4.0_5.stf
Iteraciones de corrección de aristas encroached inicial	117
Iteraciones de corrección de AE post inicio	0
Iteraciones de refinamiento	21
Tiempo de corrección de aristas encroached inicial [ms]	0,9977
Tiempo de corrección de AE post inicio[ms]	0
Tiempo de refinamiento [ms]	0
Lepp más largo	0

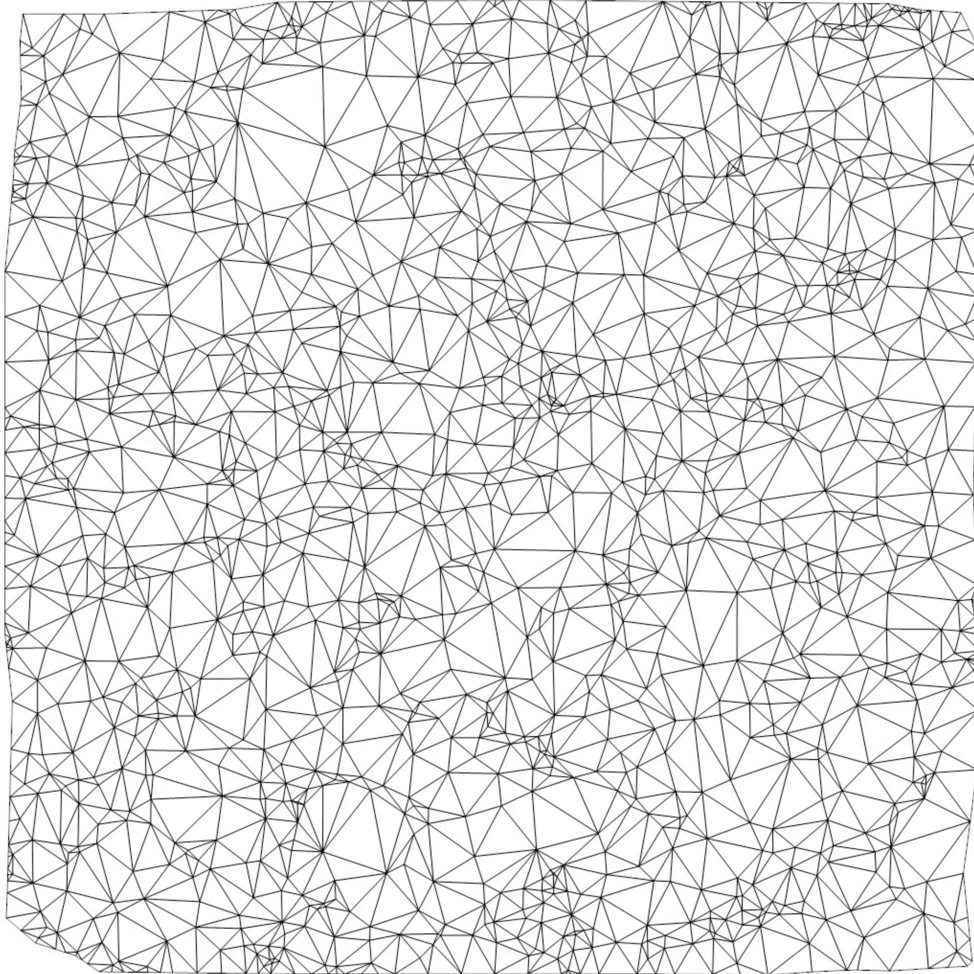
Resultados para ángulo = 10°



Archivo de triangulación completa original	C:\Users\Max\Desktop\1000.mesh.bitf
Éxito	Sí
Divergente	No
Timeout de seguridad alcanzado	No

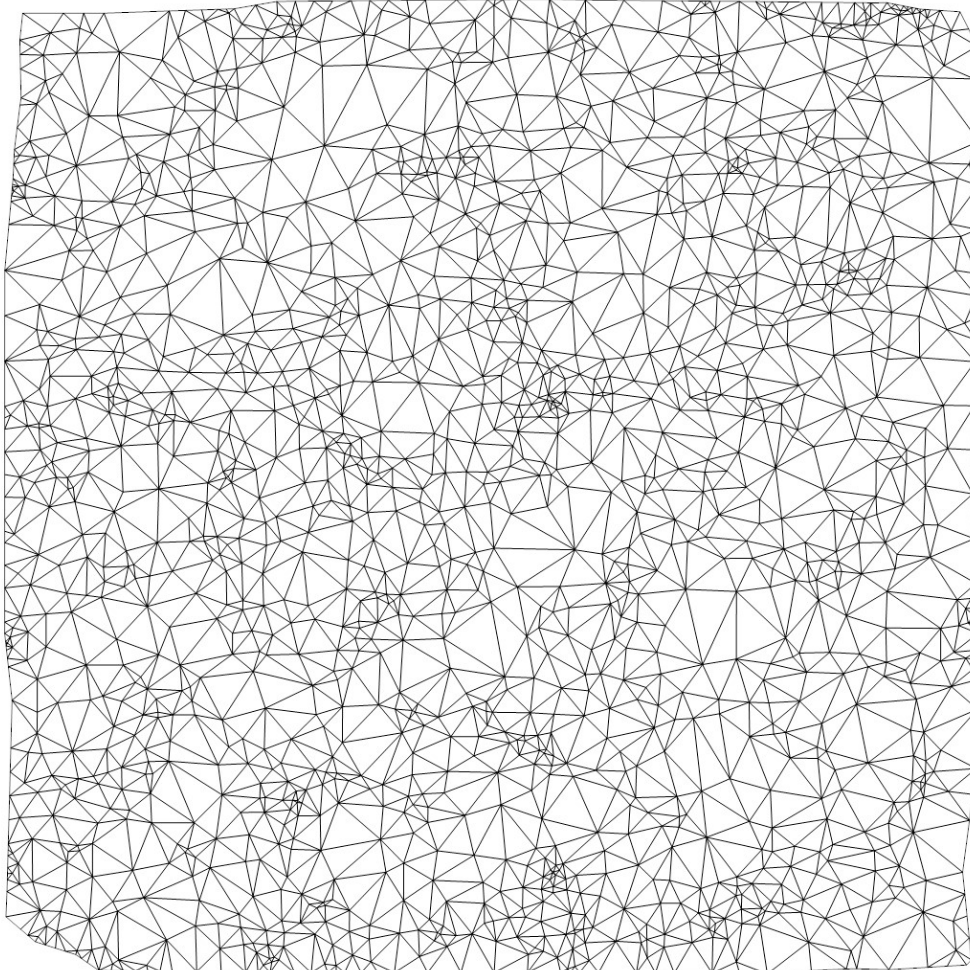
N° Vértices iniciales	1000
N° Vértices finales	1222
N° Triángulos iniciales	1955
N° Triángulos finales	2270
N° Aristas restringidas iniciales	43
N° Aristas restringidas finales	172
Ángulo interior más pequeño inical	1,02903606495574
Ángulo interior más pequeño final	10,0103352660882
Archivo de triangulacion simple inicial	1000.mesh_initial.stf
Archivo de triangulación simple final	1000.mesh_final_0.1.1.4.0_10.stf
Iteraciones de corrección de aristas encroached inicial	117
Iteraciones de corrección de AE post inicio	0
Iteraciones de refinamiento	105
Tiempo de corrección de aristas encroached inicial [ms]	0,9991
Tiempo de corrección de AE post inicio[ms]	0
Tiempo de refinamiento [ms]	1,9959
Lepp más largo	0

Resultados para ángulo = 15°



Archivo de triangulación completa original	C:\Users\Max\Desktop\1000.mesh.bitf
Éxito	Sí
Divergente	No
Timeout de seguridad alcanzado	No
N° Vértices iniciales	1000
N° Vértices finales	1385
N° Triángulos iniciales	1955
N° Triángulos finales	2583
N° Aristas restringidas iniciales	43
N° Aristas restringidas finales	185
Ángulo interior más pequeño inical	1,02903606495574
Ángulo interior más pequeño final	15,0115320871686
Archivo de triangulacion simple inicial	1000.mesh_initial.stf
Archivo de triangulación simple final	1000.mesh_final_0.1.1.4.0_15.stf
Iteraciones de corrección de aristas encroached inicial	117
Iteraciones de corrección de AE post inicio	0
Iteraciones de refinamiento	268
Tiempo de corrección de aristas encroached inicial [ms]	0,9978
Tiempo de corrección de AE post inicio[ms]	0
Tiempo de refinamiento [ms]	3,9932
Lepp más largo	0

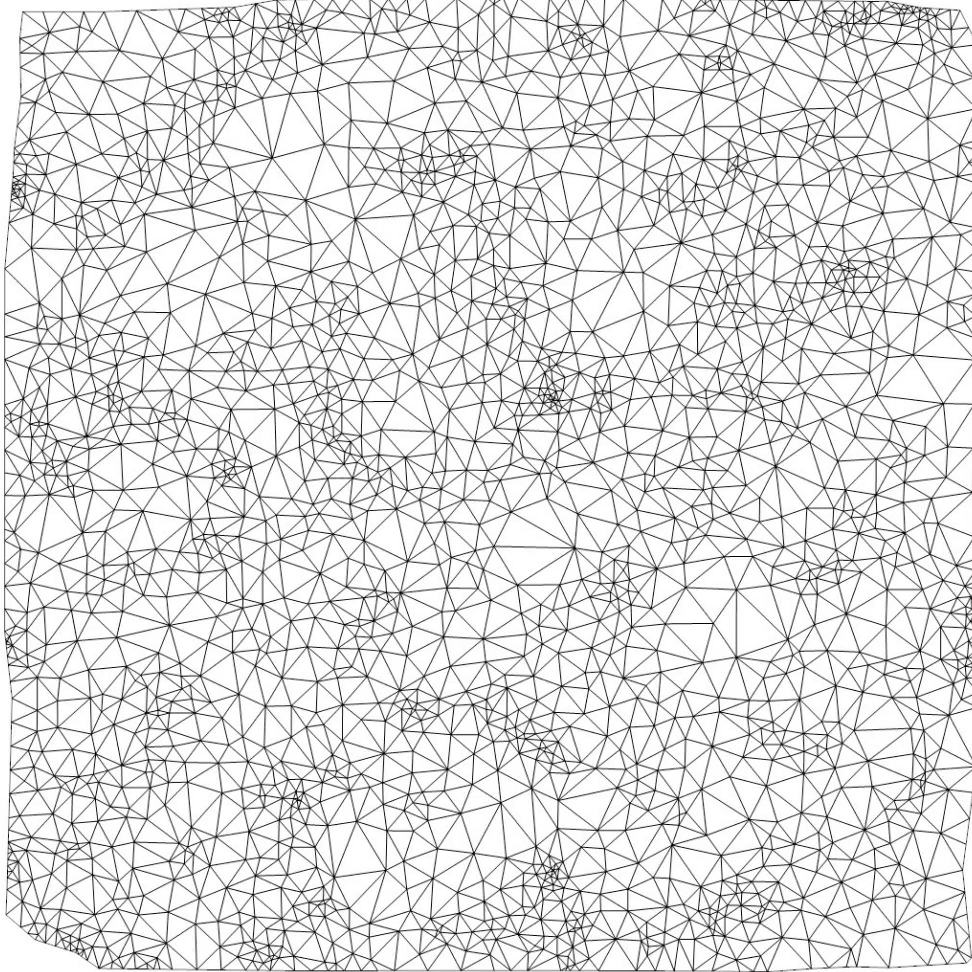
Resultados para ángulo = 20°



Archivo de triangulación completa original	C:\Users\Max\Desktop\1000.mesh.bitf
Éxito	Sí
Divergente	No
Timeout de seguridad alcanzado	No
N° Vértices iniciales	1000
N° Vértices finales	1655
N° Triángulos iniciales	1955
N° Triángulos finales	3109
N° Aristas restringidas iniciales	43
N° Aristas restringidas finales	199
Ángulo interior más pequeño inical	1,02903606495574
Ángulo interior más pequeño final	20,030283860687
Archivo de triangulacion simple inicial	1000.mesh_initial.stf
Archivo de triangulación simple final	1000.mesh_final_0.1.1.4.0_20.stf
Iteraciones de corrección de aristas encroached inicial	117
Iteraciones de corrección de AE post inicio	0

Iteraciones de refinamiento	538
Tiempo de corrección de aristas encroached inicial [ms]	0,9978
Tiempo de corrección de AE post inicio[ms]	0
Tiempo de refinamiento [ms]	7,9848
Lepp más largo	0

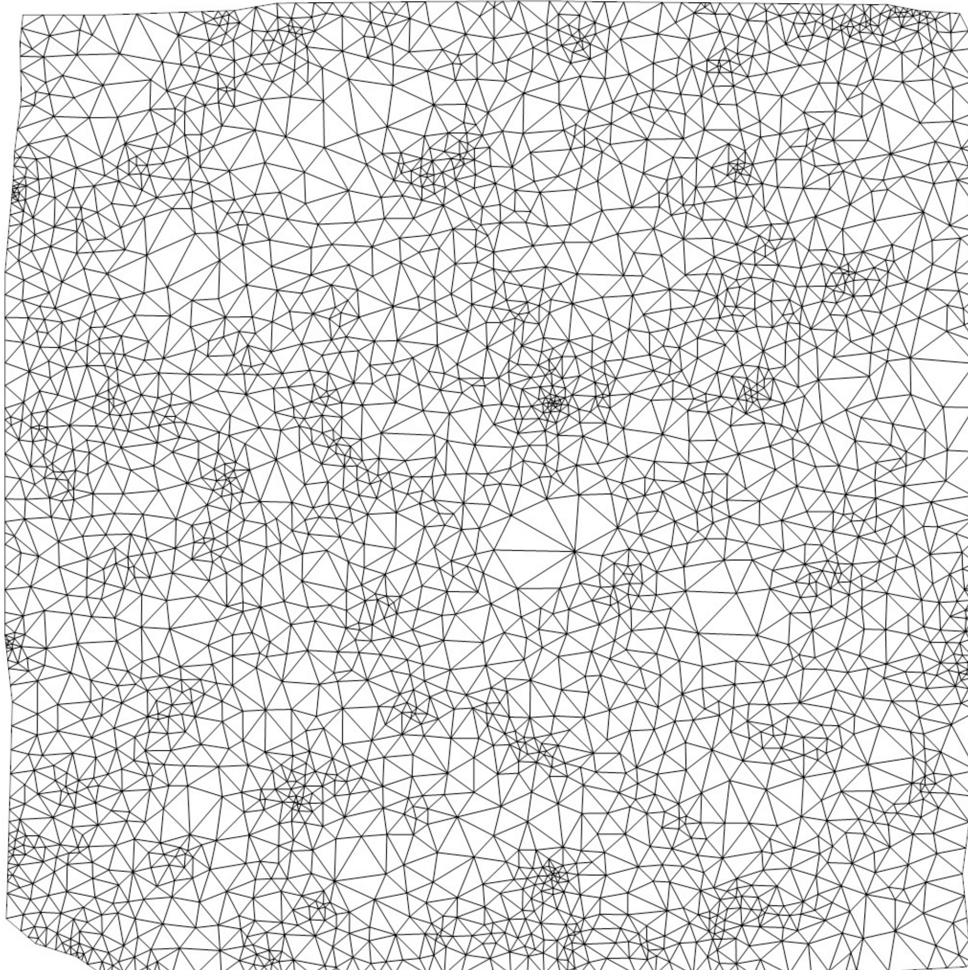
Resultados para ángulo = 25°



Archivo de triangulación completa original	C:\Users\Max\Desktop\1000.mesh.bitf
Éxito	Sí
Divergente	No
Timeout de seguridad alcanzado	No
N° Vértices iniciales	1000
N° Vértices finales	2168
N° Triángulos iniciales	1955
N° Triángulos finales	4108
N° Aristas restringidas iniciales	43
N° Aristas restringidas finales	226
Ángulo interior más pequeño inical	1,02903606495574
Ángulo interior más pequeño final	25,0140227801888
	1000.mesh_initial.stf

Archivo de triangulacion simple inicial	
Archivo de triangulación simple final	1000.mesh_final_0.1.1.4.0_25.stf
Iteraciones de corrección de aristas encroached inicial	117
Iteraciones de corrección de AE post inicio	0
Iteraciones de refinamiento	1051
Tiempo de corrección de aristas encroached inicial [ms]	0,9982
Tiempo de corrección de AE post inicio[ms]	0
Tiempo de refinamiento [ms]	15,9696
Lepp más largo	0

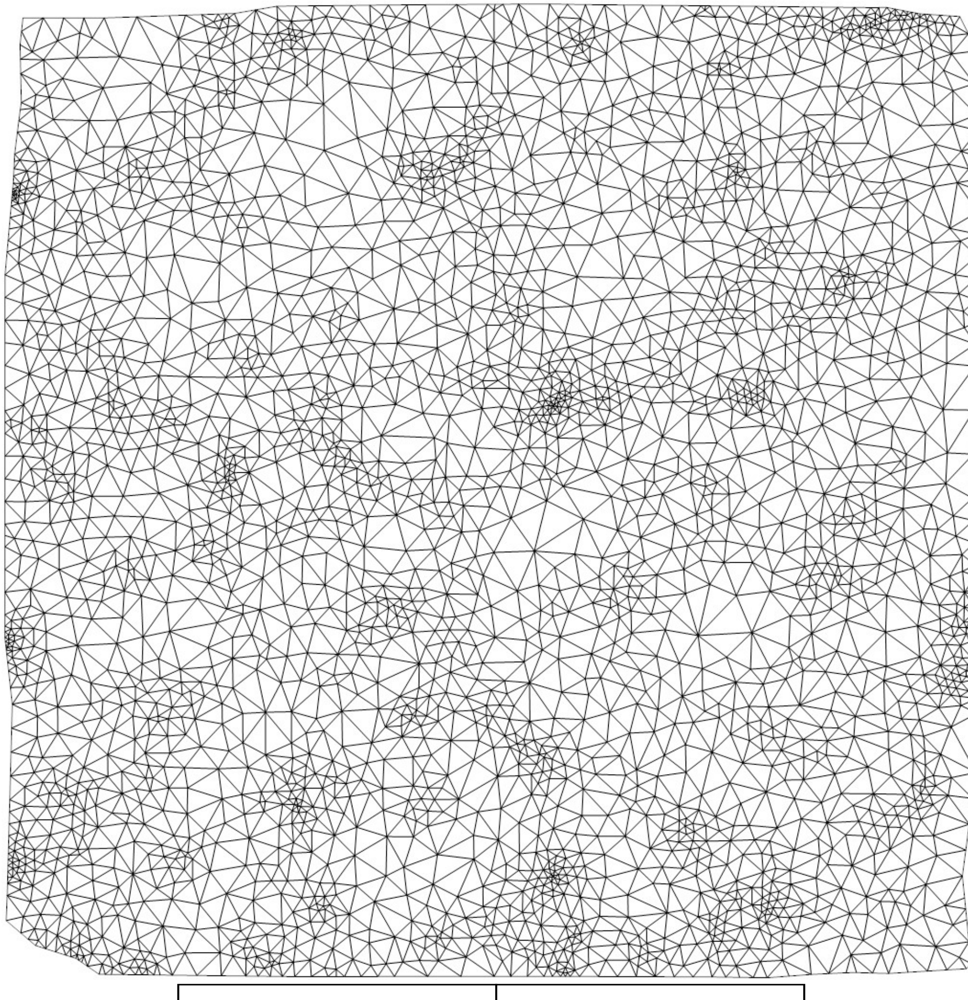
Resultados para ángulo = 28°



Archivo de triangulación completa original	C:\Users\Max\Desktop\1000.mesh.bitf
Éxito	Sí
Divergente	No
Timeout de seguridad alcanzado	No
N° Vértices iniciales	1000
N° Vértices finales	2656

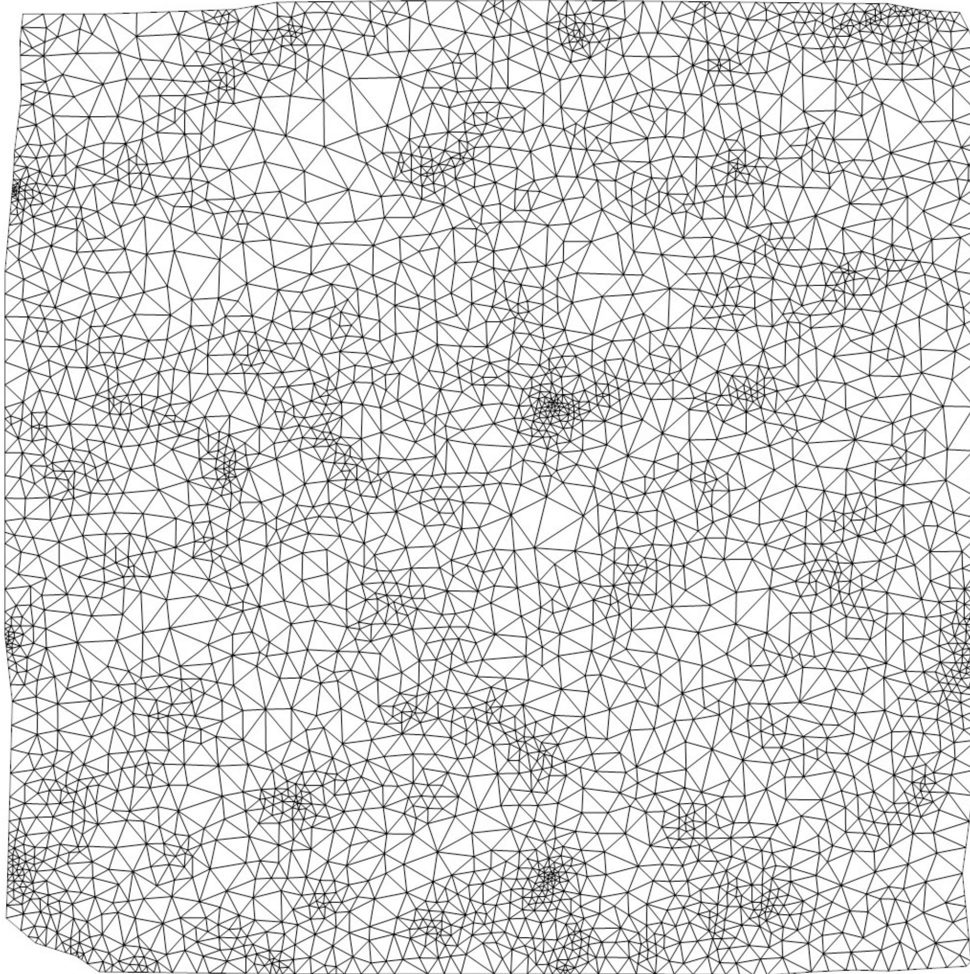
N° Triángulos iniciales	1955
N° Triángulos finales	5065
N° Aristas restringidas iniciales	43
N° Aristas restringidas finales	245
Ángulo interior más pequeño inical	1,02903606495574
Ángulo interior más pequeño final	28,0106347983363
Archivo de triangulacion simple inicial	1000.mesh_initial.stf
Archivo de triangulación simple final	1000.mesh_final_0.1.1.4.0_28.stf
Iteraciones de corrección de aristas encroached inicial	117
Iteraciones de corrección de AE post inicio	0
Iteraciones de refinamiento	1539
Tiempo de corrección de aristas encroached inicial [ms]	0,9982
Tiempo de corrección de AE post inicio[ms]	0
Tiempo de refinamiento [ms]	22,9561
Lepp más largo	0

Resultados para ángulo = 29°



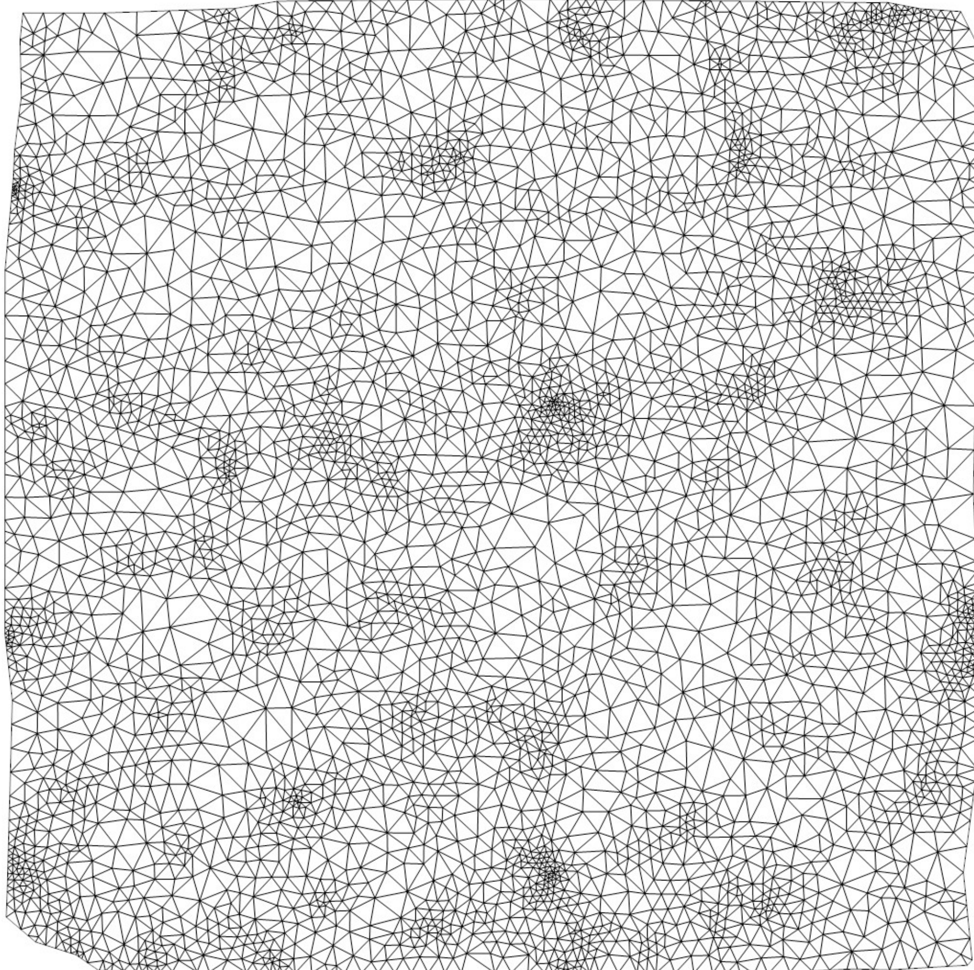
Archivo de triangulación completa original	C:\Users\Max\Desktop\1000.mesh.bitf
Éxito	Sí
Divergente	No
Timeout de seguridad alcanzado	No
N° Vértices iniciales	1000
N° Vértices finales	2981
N° Triángulos iniciales	1955
N° Triángulos finales	5706
N° Aristas restringidas iniciales	43
N° Aristas restringidas finales	254
Ángulo interior más pequeño inical	1,02903606495574
Ángulo interior más pequeño final	29,0026931269709
Archivo de triangulacion simple inicial	1000.mesh_initial.stf
Archivo de triangulación simple final	1000.mesh_final_0.1.1.4.0_29.stf
Iteraciones de corrección de aristas encroached inicial	117
Iteraciones de corrección de AE post inicio	0
Iteraciones de refinamiento	1864
Tiempo de corrección de aristas encroached inicial [ms]	0,9978
Tiempo de corrección de AE post inicio[ms]	0
Tiempo de refinamiento [ms]	26,9485
Lepp más largo	0

Resultados para ángulo = 30°



Archivo de triangulación completa original	C:\Users\Max\Desktop\1000.mesh.bitf
Éxito	Sí
Divergente	No
Timeout de seguridad alcanzado	No
N° Vértices iniciales	1000
N° Vértices finales	3377
N° Triángulos iniciales	1955
N° Triángulos finales	6488
N° Aristas restringidas iniciales	43
N° Aristas restringidas finales	264
Ángulo interior más pequeño inical	1,02903606495574
Ángulo interior más pequeño final	30,0047645616812
Archivo de triangulacion simple inicial	1000.mesh_initial.stf
Archivo de triangulación simple final	1000.mesh_final_0.1.1.4.0_30.stf
Iteraciones de corrección de aristas encroached inicial	117
Iteraciones de corrección de AE post inicio	0
Iteraciones de refinamiento	2260
Tiempo de corrección de aristas encroached inicial [ms]	0,9982
Tiempo de corrección de AE post inicio[ms]	0
Tiempo de refinamiento [ms]	32,9368
Lepp más largo	0

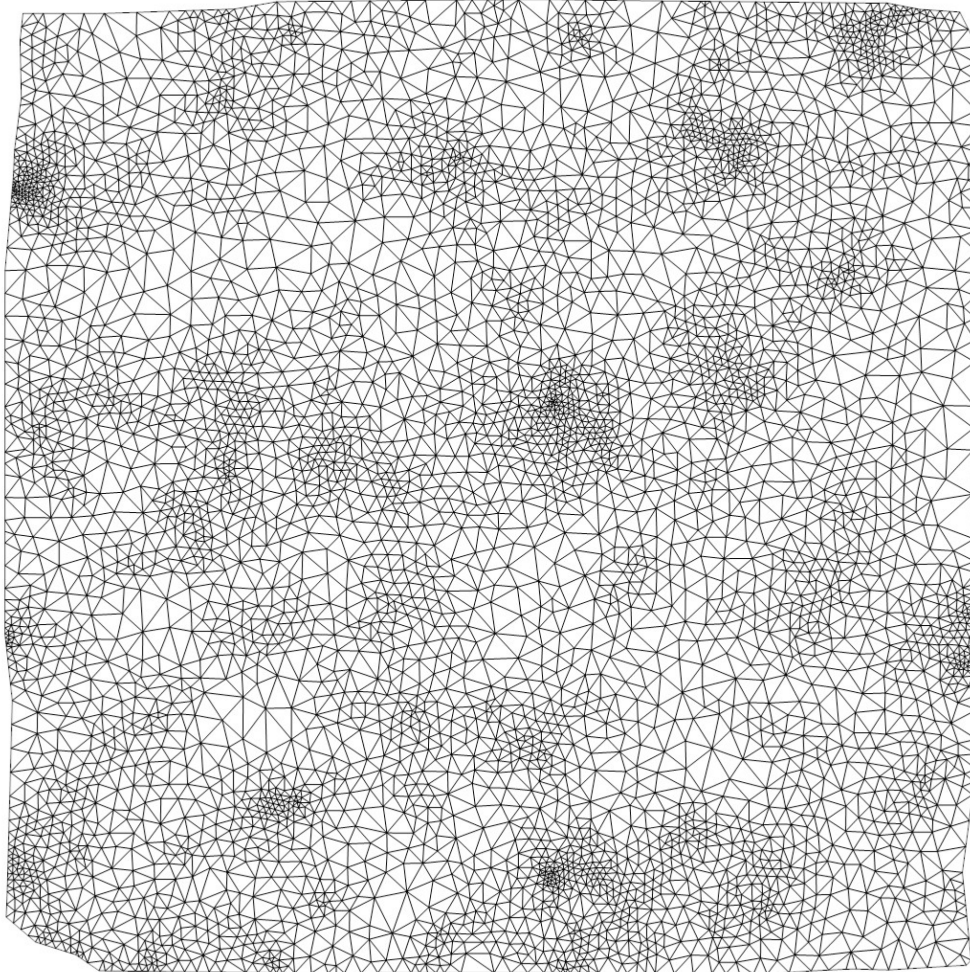
Resultados para ángulo = 31°



Archivo de triangulación completa original	C:\Users\Max\Desktop\1000.mesh.bitf
Éxito	Sí
Divergente	No
Timeout de seguridad alcanzado	No
N° Vértices iniciales	1000
N° Vértices finales	3947
N° Triángulos iniciales	1955
N° Triángulos finales	7618
N° Aristas restringidas iniciales	43
N° Aristas restringidas finales	274
Ángulo interior más pequeño inical	1,02903606495574
Ángulo interior más pequeño final	31,0083006476287
Archivo de triangulacion simple inicial	1000.mesh_initial.stf
Archivo de triangulación simple final	1000.mesh_final_0.1.1.4.0_31.stf
Iteraciones de corrección de aristas encroached inicial	117
Iteraciones de corrección de AE post inicio	0

Iteraciones de refinamiento	2830
Tiempo de corrección de aristas encroached inicial [ms]	0,9982
Tiempo de corrección de AE post inicio[ms]	0
Tiempo de refinamiento [ms]	44,914
Lepp más largo	0

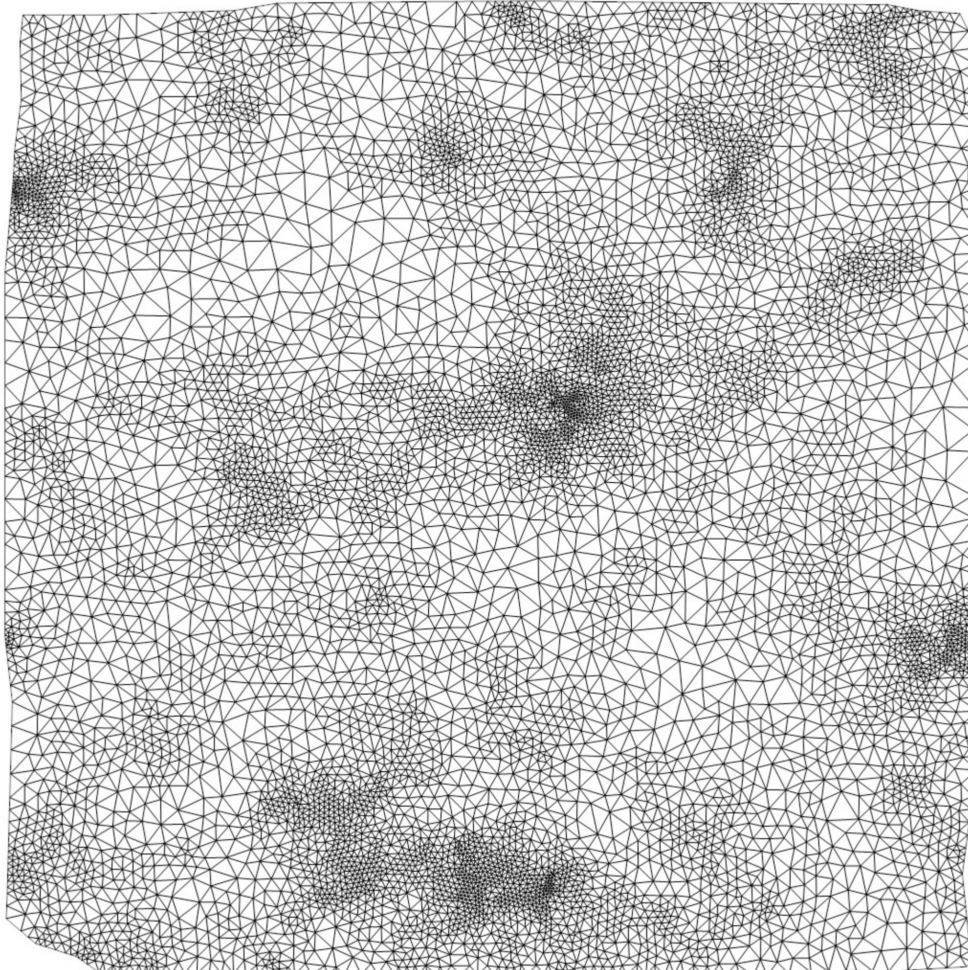
Resultados para ángulo = 32°



Archivo de triangulación completa original	C:\Users\Max\Desktop\1000.mesh.bitf
Éxito	Sí
Divergente	No
Timeout de seguridad alcanzado	No
N° Vértices iniciales	1000
N° Vértices finales	4869
N° Triángulos iniciales	1955
N° Triángulos finales	9451
N° Aristas restringidas iniciales	43
N° Aristas restringidas finales	285
Ángulo interior más pequeño inical	1,02903606495574
Ángulo interior más pequeño final	32,0043794207474
	1000.mesh_initial.stf

Archivo de triangulacion simple inicial	
Archivo de triangulación simple final	1000.mesh_final_0.1.1.4.0_32.stf
Iteraciones de corrección de aristas encroached inicial	117
Iteraciones de corrección de AE post inicio	0
Iteraciones de refinamiento	3752
Tiempo de corrección de aristas encroached inicial [ms]	0,9982
Tiempo de corrección de AE post inicio[ms]	0
Tiempo de refinamiento [ms]	53,897
Lepp más largo	0

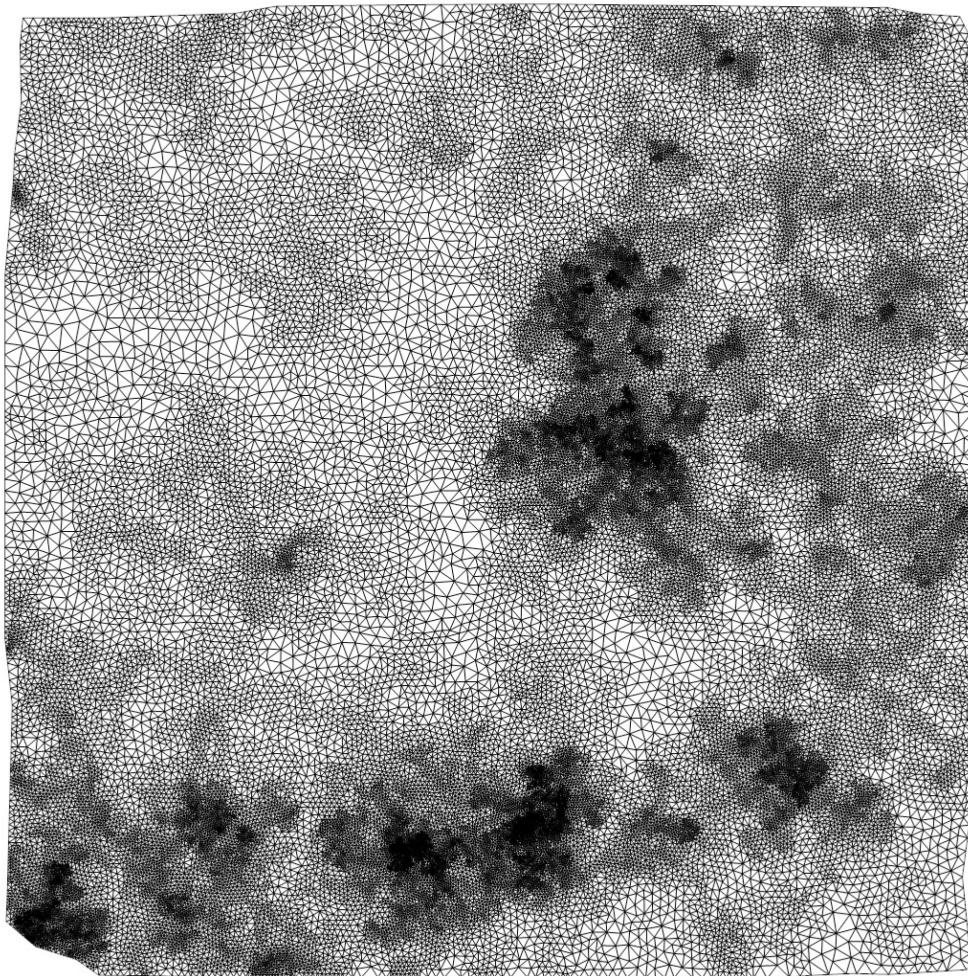
Resultados para ángulo = 33°



Archivo de triangulación completa original	C:\Users\Max\Desktop\1000.mesh.bitf
Éxito	Sí
Divergente	No
Timeout de seguridad alcanzado	No
N° Vértices iniciales	1000
N° Vértices finales	7724

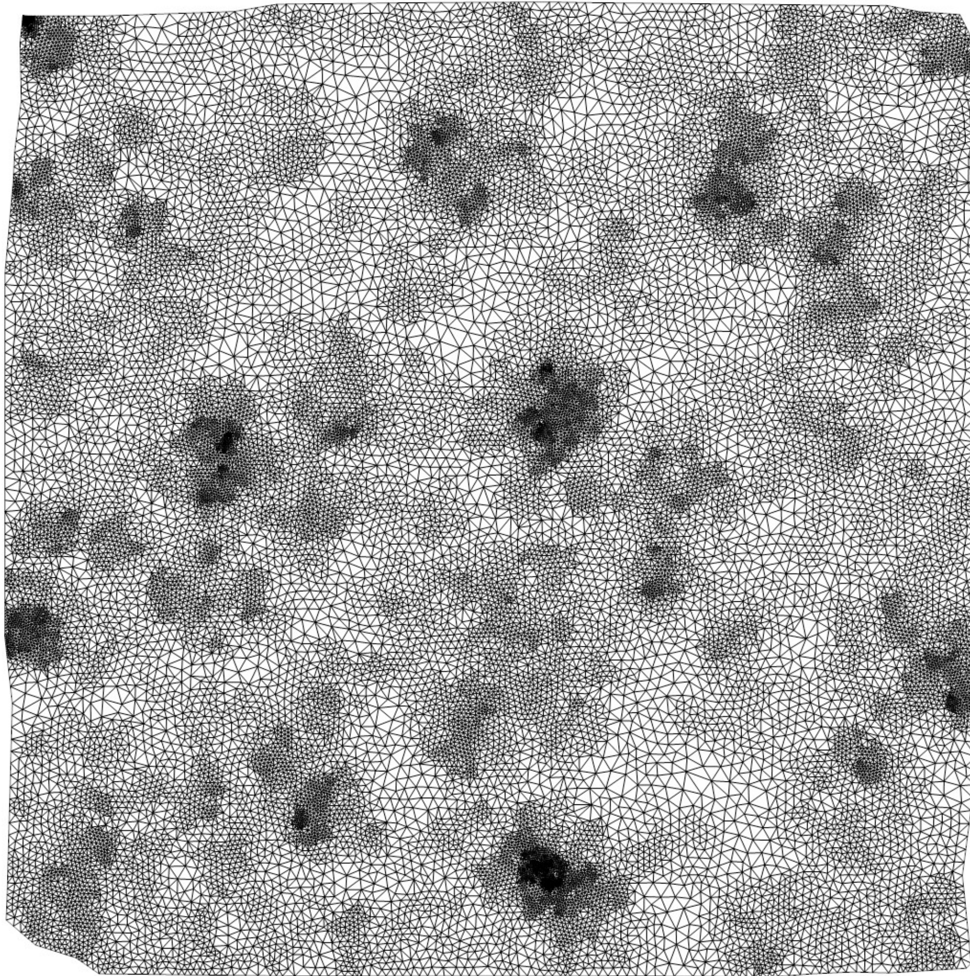
N° Triángulos iniciales	1955
N° Triángulos finales	15129
N° Aristas restringidas iniciales	43
N° Aristas restringidas finales	317
Ángulo interior más pequeño inical	1,02903606495574
Ángulo interior más pequeño final	33,0014601947892
Archivo de triangulacion simple inicial	1000.mesh_initial.stf
Archivo de triangulación simple final	1000.mesh_final_0.1.1.4.0_33.stf
Iteraciones de corrección de aristas encroached inicial	117
Iteraciones de corrección de AE post inicio	0
Iteraciones de refinamiento	6607
Tiempo de corrección de aristas encroached inicial [ms]	0,9978
Tiempo de corrección de AE post inicio[ms]	0
Tiempo de refinamiento [ms]	95,8176999999999
Lepp más largo	0

Resultados para ángulo = 34°



Archivo de triangulación completa original	C:\Users\Max\Desktop\1000.mesh.bitf
Éxito	No
Divergente	Sí
Timeout de seguridad alcanzado	No
N° Vértices iniciales	1000
N° Vértices finales	40776
N° Triángulos iniciales	1955
N° Triángulos finales	81001
N° Aristas restringidas iniciales	43
N° Aristas restringidas finales	549
Ángulo interior más pequeño inical	1,02903606495574
Ángulo interior más pequeño final	16,2590155812028
Archivo de triangulacion simple inicial	1000.mesh_initial.stf
Archivo de triangulación simple final	1000.mesh_final_0.1.1.4.0_34.stf
Iteraciones de corrección de aristas encroached inicial	117
Iteraciones de corrección de AE post inicio	0
Iteraciones de refinamiento	39659
Tiempo de corrección de aristas encroached inicial [ms]	0,9978
Tiempo de corrección de AE post inicio[ms]	0
Tiempo de refinamiento [ms]	714,638199999998
Lepp más largo	0

Resultados para ángulo = 35°



Archivo de triangulación completa original	C:\Users\Max\Desktop\1000.mesh.bitf
Éxito	No
Divergente	No
Timeout de seguridad alcanzado	No
N° Vértices iniciales	1000
N° Vértices finales	0
N° Triángulos iniciales	1955
N° Triángulos finales	0
N° Aristas restringidas iniciales	43
N° Aristas restringidas finales	0
Ángulo interior más pequeño inical	1,02903606495574
Ángulo interior más pequeño final	0
Archivo de triangulacion simple inicial	1000.mesh_initial.stf
Archivo de triangulación simple final	1000.mesh_final_0.1.1.4.0_35.stf
Iteraciones de corrección de aristas encroached inicial	117
Iteraciones de corrección de AE post inicio	0
Iteraciones de refinamiento	23434
Tiempo de corrección de aristas encroached inicial [ms]	0,9982
Tiempo de corrección de AE post inicio[ms]	0
Tiempo de refinamiento [ms]	403,232099999999
Lepp más largo	0