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

DESARROLLO DE UNA HERRAMIENTA QUE GENERA MALLAS DE SUPERFICIE COMPUESTAS DE CUADRILÁTEROS PARA MODELAR EL CRECIMIENTO DE ÁRBOLES

MEMORIA PARA OPTAR AL TÍTULO DE INGENIERO CIVIL EN COMPUTACIÓN

CRISTINA MELO LAGOS

PROFESOR GUÍA:
NANCY HITSCHFELD KAHLER

MIEMBROS DE LA COMISIÓN:
MARÍA CECILIA RIVARA ZÚÑIGA
PATRICIO INOSTROZA FAJARDIN

SANTIAGO DE CHILE
DICIEMBRE 2007

Tabla de Contenidos

Índice de figuras

Resumen

Éste es el resumen

Capítulo 1

Introducción

1.1. Antecedentes Generales

La Computación Gráfica abarca no sólo desplegar figuras en la pantalla, sino que esas figuras sean el resultado del modelado de un problema real. Es así como una aplicación común es tratar geométrica y numéricamente un problema de compleja resolución analítica. Para ello se requieren poderosas herramientas que permitan representar la realidad lo más fielmente posible, con las restricciones de factibilidad y costo computacional existentes.

Un ejemplo de lo anterior es el problema del crecimiento de un árbol. Existen trabajos anteriores [1, 2] en que, motivados por este problema, se construyó una aplicación que dada una malla geométrica triangular como entrada, modela la evolución en el tiempo del tronco del árbol.

Por otra parte, las mallas de cuadriláteros está siendo cada vez más comunes en aplicaciones tales como gráfica de juegos, la industria del cine y visualizaciones médicas y científicas.

Resulta entonces interesante comparar los resultados obtenidos utilizando una malla triangular y de cuadriláteros, en el ejemplo antes descrito.

1.2. Motivación

Se desea modelar el crecimiento de un árbol, en particular de la superficie cilíndrica de su tronco, mediante una malla de cuadriláteros. Este modelamiento permitirá a otros profesionales estudiar esta discretización para luego extrapolar los resultados al modelo real.

Cada punto de la malla sufre distintos desplazamientos, la concentración de la hormona de crecimiento en cada punto de la malla es usada para estimar el desplazamiento del

nodo, por lo que hay que abordar el problema de la deformación de la malla manteniendo la consistencia, aplicando distintas estrategias conocidas para lograrlo.

Además, se requiere que la implementación de la herramienta se adhiera al paradigma de Programación Orientada a Objetos, con el fin de apoyar la extensibilidad y claridad de la aplicación a desarrollar.

Se espera reutilizar parte del trabajo anterior [2] y, a la vez, diseñar nuevas clases y algoritmos y rediseñar algunas componentes que se estimen necesarias.

1.3. Objetivo General

El objetivo general de esta memoria es implementar una herramienta para visualizar, desplazar y trabajar (refinar/desrefinar, suavizar) mallas geométricas de cuadriláteros. Se reutilizará el diseño de las clases y la implementación de la interfaz gráfica del trabajo descrito en [2], en la medida de lo posible. Se rediseñarán clases y parte de la interfaz que necesiten perfeccionarse.

1.4. Objetivos Específicos

- Adaptar, diseñar e implementar algoritmos de refinamiento, desrefinamiento y movimiento de bordes para mallas de cuadriláteros.
- Revisar y mejorar el diseño actual. Por ejemplo, la representación de la malla es una clase que posee métodos que no corresponden al alcance de la malla y que pueden reimplementarse en otra clase.
- Incorporar algoritmos para manejar colisiones de sectores no vecinos.
- Mejorar el tratamiento de los problemas de precisión provocados por la representación en punto flotante de valores reales. Se trabajará en dos enfoques: reescribir las operaciones para disminuir el error y mejorar la elección del rango de error aceptado al hacer comparaciones.

Capítulo 2

Antecedentes

2.1. Marco teórico

En la generación de mallas en 2D (dos dimensiones), los dos enfoques más comunes son utilizar triángulos o cuadriláteros.

Los triángulos son los polígonos más simples y poseen importantes propiedades (por ejemplo, sobre el tamaño de sus lados, medianas y puntos notables) y relaciones con circunferencias (inscritas, circunscritas). Esta simplicidad y, a la vez, completitud, hacen que los triángulos hayan sido la figura geométrica natural para crear mallas geométricas que discretizan figuras reales.

Por otra parte, los cuadriláteros son figuras más complejas que los triángulos y a diferencia de éstos, pueden no estar completamente contenidos en un plano (utilizando la definición genérica de cuadriláteros, que no exige que sean polígonos). Además, su representación es más compleja, pues dependiendo del orden en que se unen los cuatro puntos, se genera un cuadrilátero válido o bien polígonos no simples (con arcos cruzados).

Sin embargo, el uso de mallas compuestas por cuadriláteros es un enfoque que ha ganado adeptos, pues se considera que estas mallas son más precisas y flexibles. Una dificultad de abordar este enfoque es que no existe abundante literatura al respecto como en el caso de las mallas de triángulos.

En particular, enfocándose en el problema a tratar en esta memoria, se cree que las mallas de cuadriláteros podrían representar mejor la estructura de la corteza de un árbol, al ser la forma de las células más similar a esta figura geométrica, lo que permitiría simular la distribución de la hormona de crecimiento en el árbol de manera más natural.

Existe un trade-off entre los métodos de generación automática de mallas geométricas y los que requieren intervención manual. Los primeros entregan una malla de menor calidad (errores en los bordes de la malla y mayor número de nodos irregulares), por lo que a veces se descompone manualmente la geometría en regiones antes de aplicar los métodos automáticos, lo que evidentemente conlleva mayor trabajo. Existen también en-

foques intermedios que intentar aprovechar lo mejor de ambas alternativas: *paving* [7], que construye progresivamente la malla insertando filas de cuadriláteros desde el contorno del objeto hacia el interior.

Por otra parte, independiente de la figura geométrica usada para general la malla, existen propiedades que se deben garantizar, en particular cuando se evaluarán modelos matemáticos en la malla generada, como en el caso de esta memoria. Algunas de las consideraciones necesarias son [3, 6]:

- Representar el dominio de la manera más exacta posible, acotando el error (ϵ). Esto es particularmente importante en los bordes, pues las aplicaciones científicas o ingenieriles suelen ser sensibles a cambios en ellos.
- Considerar las restricciones impuestas por los métodos numéricos usados, principalmente para acelerar la convergencia de éstos. Las restricciones son:
  - Ser conforme.
  - Satisfacer requisitos de densidad de puntos, usando la menor cantidad de puntos posibles. (Métodos de refinamiento).
  - Satisfacer criterios de calidad medidos en función de ángulos, aspecto de radio, altura y área, entre otros.
  - Evitar ángulos muy pequeños. (Métodos para mejorar la calidad).

A continuación se desarrollarán alguno de estos puntos.

2.1.1. Precisión en Computación Geométrica [5]

Los computadores tienen recursos limitados y por lo tanto no pueden representar un conjunto infinito de números, como los enteros o los reales, sino que utilizan representaciones aproximadadas de éstos. En el caso de los enteros, se representa sólo un rango de ellos. En el caso de los reales, además de no representar los valores fuera de un cierto intervalo, tampoco se representan todos los valores de ese intervalo, pues dentro de cualquier intervalo no vacío de reales hay infinitos valores y obviamente no pueden representarse todos ellos. En adelante se limitará la discusión a la representación de reales, también llamada *representación en punto flotante*, pues la Computación Geométrica utiliza aritmética real y no entera.

El conjunto de números de punto flotante $\mathbb{F}(\beta, p, e_{min}, e_{max})$ se define como:

$$\mathbb{F} = \{x = s \cdot \sum_{k=0}^{p-1} d_k \beta^{-k} \cdot \beta^e / s \in \{1, -1\}, d_k \in \mathbb{Z} \cap [0, \beta),$$

$$e \in \mathbb{Z} \cap [e_{min}, e_{max}], \sum |d_k| > 0 \Rightarrow d_0 > 0\}$$

La expresión anterior corresponde a la representación normalizada, es decir, $d_0 > 0$ a menos que $x = 0$. Esta convención se ocupa para disminuir los errores cometidos al realizar operaciones. En los sistemas computacionales, se trabaja con números binarios ($\beta = 2$) normalizados, por lo que d_0 es siempre 1, así que no es necesario almacenarlo.

Para aproximar números reales en este conjunto $\mathbb{F}$, se implementa la función $fl : \mathbb{R} \to \mathbb{F}$, que a cada x en $\mathbb{R}$ le asigna el número en $\mathbb{F}$ más cercano a x y de no ser único, le asigna aquél con último dígito distinto de cero par.

El error relativo de aproximar x por $fl(x)$ está acotado superiormente por ϵ_m, una expresión constante con respecto a x y dependiente sólo de la base β y de la mantiza p características del conjunto $\mathbb{F}$. A esta expresión se le llama *epsilon máquina*.

Los números representables en un sistema de punto flotante no se distribuyen homogéneamente, sino que la distancia entre ellos va aumentando con el exponente e utilizado, por lo tanto aumenta el error absoluto que se puede cometer, pero el error relativo siempre está acotado por epsilon máquina.

Para implementar una función real f en un sistema de números flotantes, se construye una función análoga $\tilde{f}$ que va desde $\mathbb{F}$ a $\mathbb{F}$ y se dice que implementa bien a la original si corresponde a la mejor aproximación posible en $\mathbb{F}$, es decir, si:

$$\tilde{f}(x) = fl(f(x))$$

La mayoría de las funciones no son bien implementadas en los sistemas numéricos. Sólo se garantiza que un número pequeño de ellas lo sea, como las operaciones elementales binarias (suma, resta, multiplicación y división) y algunas unarias (como la raíz cuadrada). Las demás funciones que se derivan de ellas no siempre están bien implementadas, y por lo tanto, los errores relativos en los cálculos pueden ser mayores a epsilon máquina. Además, al realizar sucesivas operaciones, aunque cada una de ellas por separado esté bien implementada, se va acumulando el error y aumenta la cota del error. Las operaciones que se realizan en un sistema de números flotantes se dice que usan aritmética imprecisa.

Un caso que se ha estudiado particularmente es el error numérico en una suma de términos: $S = \sum_i a_i$. La cota al error cometido que se ha encontrado es directamente proporcional a epsilon máquina, al número de términos sumados y a un factor de amplificación A:

$$A = \frac{\sum_{a_i>0} a_i + \sum_{a_i<0} -a_i}{\sum_{a_i>0} a_i - \sum_{a_i<0} -a_i}$$

Si $\sum_{a_i>0} a_i \approx \sum_{a_i<0} -a_i$, entonces $A \to +\infty$, lo que se conoce como la amplificación de errores producida por diferencias de números cercanos.

Lo anterior no significa que necesariamente los errores crezcan al hacer diferencia de números cercanos, pues se trata de una cota superior, pero empíricamente se han observado casos en que se alcanzan valores cercanos a la cota, por lo que es altamente recomendable

evitar usar expresiones donde aparezcan diferencias de números cercanos. Por ejemplo, muchas de esas expresiones pueden reescribirse para evitar esta situación:

- $\sqrt{n+1} - \sqrt{n} = \frac{1}{\sqrt{n+1}+\sqrt{n}}$
- $\frac{1}{n} - \frac{1}{n+1} = \frac{1}{n(n+1)}$

Al no acotar el error acumulado puede tenerse que el valor absoluto del error de aproximación cometido sea mayor al valor absoluto de la expresión, es decir, "que el error se coma la expresión".

El no controlar los errores numéricos cometidos al implementar algoritmos geométricos, puede llevar a programas que se caen, se quedan en ciclos infinitos, o simplemente entregan resultados erróneos.

La parte más crítica en la implementación son las condiciones, pues ellas determinan el flujo de control del programa. La mayoría de ellas involucran desigualdades, las que pueden reformularse como preguntas sobre el signo de una expresión. El enfoque más común es utilizar un valor epsilon para suponer que un número es suficientemente cercano a cero si es menor a epsilon y por lo tanto considerarlo como cero, lo que puede ser incorrecto pues entre cero y epsilon hay valores representables en la aritmética imprecisa cuya aproximación a cero es incorrecta. El principal problema de este enfoque es que por simplicidad el valor de epsilon es elegido por ensayo y error y se toma constante para todos los cálculos, sin considerar el tamaño de los operandos involucrados.

Un mejor enfoque es realizar cálculos exactos, es decir, asegurar que todas las decisiones hechas por el algoritmo son correctas. Para ello, no es necesario que todos los cálculos se realicen con aritmética exacta, sino que basta con que los errores producidos no alteren las decisiones del algoritmo. La técnica de *floating point filter* apunta a esto mismo y consiste en identificar aquellos casos en que el error podría llevar a una mala decisión, y sólo en esos casos, reevaluar la expresión con alguna técnica que permita asegurar la correctitud de la decisión, por ejemplo, esa técnica puede ser usar arimética exacta. Esto se justifica porque realizar aritmética exacta es caro y en ocasiones ni siquiera es posible hacerlo, por ejemplo, no es posible representar exactamente números irracionales en ningún sistema computacional, pues tienen infinitas cifras.

Por ejemplo, la técnica de *floating point filter* se puede usar para determinar si un conjunto de puntos está dentro de una circunferencia de centro (x_0, y_0) y radio r. Para ello hay que revisar para cada punto (x, y) si cumple que $r^2-(x-x_0)^2+(y-y_0)^2 \geq 0$. Para los puntos que no se encuentran "cerca" del borde (tomando como escala el posible error cometido), no se requiere usar aritmética exacta. En cambio los puntos cercanos al borde de la circunferencia se pueden calcular con aritmética exacta, pues utilizando aritmética inexacta podrían clasificarse como dentro o fuera de la circunferencia incorrectamente.

Al implementar un algoritmo geométrico hay que tratar con especial cuidado los *casos degenerados*. Los casos degenerados corresponden una configuración límite de un problema geométrico tal que no es una entrada que los algoritmos que resuelven el problema

están preparados para recibir. En ese sentido, constituyen excepciones del algoritmo pues no se les puedes aplicar el mismo procesamiento que para el resto de las entradas. Desde el punto de vista del cálculo, esto se traduce en que cuando el algoritmo quiere testear el signo de una expresión, se encuentra con que la expresión vale cero y hay que definir qué hacer en ese caso, pues se esperaba clasificar la expresión como positiva o negativa. Los casos degenerados obligan a hacer implementaciones más complejas para tratar estas excepciones. Otro enfoque es convertir un caso degenerado en uno no degenerado, dándole un signo a las expresiones que son cero.

Otro problema existente es cuando se utilizan epsilons y se puede clasificar como un caso degenerado uno que no lo es (un valor muy cercano a cero, pero distinto de cero).

Un ejemplo de caso degenerado es cuando se quiere trabajar con el triángulo formado por tres rectas no paralelas, pero las tres rectas se intersectan en un mismo punto. Al determinar la distancia entre los vértices del triángulo, se espera que el valor sea positivo, pero se obtiene un valor cero.

2.1.2. Generación de mallas geométricas

Todo polígono puede ser particionado en triángulos mediante la sucesiva adición de diagonales. Este simple proceso describe un algoritmo de generación de mallas triangulares en 2D. Sin embargo, el algoritmo ampliamente usado es la triangulación de Delaunay que asegura un grado de calidad de la malla al maximizar el mínimo ángulo entre todas las posibles triangulaciones de un polígono. Esto lo logra exigiendo que cada triángulo cumpla el criterio de Delaunay, esto es, que su circuncírculo (el círculo circunscrito en el triángulo) sea vacío, es decir, sin vértices en su interior. El mismo principio se aplica para triangular superficies en 3D.

En el caso de mallas de cuadriláteros, pueden ser generadas triangulando el dominio y luego dividiendo cada triángulo en tres cuadriláteros. Se puede incluso minimizar el número de cuadriláteros generados si el dominio es convexo. El problema de este enfoque es que genera mallas de mala calidad.

Otro enfoque es el presentado en [4], que apunta a "empaquetar círculos", esto es, llenar el dominio con círculos de modo que el espacio entre ellos esté rodeado de tres o cuatro círculos tangentes y luego unir los centros de los círculos formando cuadriláteros. Este algoritmo busca garantizar la calidad de la malla formada y acotar el número de cuadriláteros usados.

Un algoritmo que ha ganado adeptos es el de *Paving* [7] que permite generar mallas bien formadas (es decir, los elementos son cercanos a cuadrados, perpendiculares a los bordes, etcétera) y geométricamente satisfactorios (es decir, los contornos de la malla tienden a seguir los contornos geométricos de los bordes).

El algoritmo consiste en iterativamente disponer filas de elementos desde el borde de la región hacia el interior. Cuando las filas se superponen o coinciden, cuidadosamente se conectan para formar una malla de cuadriláteros válida.

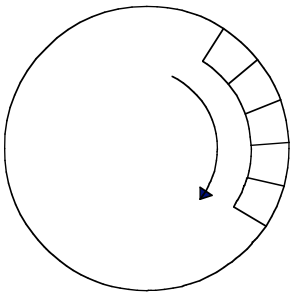

Figura 2.1: Algoritmo Paving

Este algoritmo es un aporte con respecto a otros pues logra generar una malla automáticamente, sin los costos comunes asociados a la automatización: mala calidad (falta de sensibilidad en los bordes y gran presencia de nodos que se conectan con más, o menos, de cuatro nodos).

2.2. Aplicación del legado

La aplicación del legado corresponde al software desarrollado en dos memorias de título anteriores ([1, 2]). A continuación se describirá el estado de la aplicación al inicio del desarrollo de este trabajo de título.

2.2.1. Funcionalidades

La aplicación trabaja con mallas de triángulos, pero tiene un desarrollo incipiente en mallas de cuadriláteros.

En términos generales, permite crear mallas simples, cargar mallas desde distintos formatos de entrada y guardarlas también en varios formatos de salida.

Permite la visualización gráfica de mallas y modificar una malla deformándola, refinándola, desrefinándola o mejorándola. Además, entrega información sobre la composición de la malla.

De forma más detallada, las funcionalidades de la aplicación son:

Crear malla: Se pueden crear mallas a partir de una médula y crear mallas cilíndricas. En el caso de la médula, el usuario especifica el nombre del archivo donde se encuentran las coordenadas de la médula. El cilindro, por su parte, corresponde al caso en

que la médula es una vertical, por lo tanto, en vez de ingresarse las coordenadas de la médula, se ingresa la altura del cilindro. En ambos casos el algoritmo consiste en generar anillos horizontales alrededor de la médula. Cada anillo se compone de un número constante de nodos. Al unir los nodos, se genera la malla. El usuario ingresa el radio, número de anillos y número de puntos por anillo que desea generar.

Las mallas cilíndricas puedes estar compuestas o bien de triángulos o bien de cuadriláteros, según lo decida el usuario. Las mallas generadas a partir de una médula sólo son triangulares.

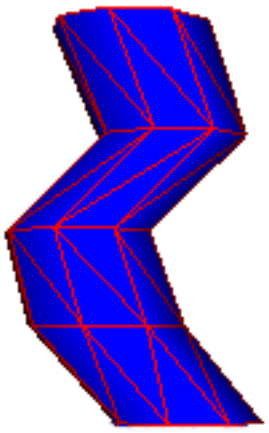

Figura 2.2: Malla triangular generada a partir de una médula

Cargar malla: Se pueden cargar mallas en distintos formatos de entrada: *Comsol*, *Matlab*, *nxyzu* y *Geomview* (los formatos se explican en el Apéndice). Cada formato tiene su propio algoritmo para cargar la malla, pero a grandes rasgos todos ellos leen las coordenadas de los nodos de la malla, leen o deducen el orden en que los nodos forman las caras, crean los nodos y caras correspondientes y además los arcos de las caras. Del mismo modo leen o calculan las normales de los puntos y calculan las normales de las caras. Una vez cargada la malla, ya no se distingue desde qué formato fue cargada, o si fue creada directamente en la aplicación.

Visualizar malla: Al crear o cargar una malla ésta es dibujada por la aplicación. Se puede elegir si se desean ver o no las caras y/o arcos de la malla. Además, se puede mover la cámara de forma de acercar, alejar y mover la malla.

Guardar mallas: Las mallas pueden ser guardadas en distintos formatos de salida: *Comsol*, *nxyzu*, *Off* y *Debug*. Éste último formato es de uso interno y se utiliza para depurar la aplicación, pues consiste en un recorrido exhaustivo de toda la información de la malla.

Deformar malla: La malla puede ser deformada. La deformación consiste en el desplazamiento de cada nodo de la malla según su concentración y en la dirección de la

normal del nodo. El usuario indica el porcentaje de la concentración que se moverá cada nodo en un paso y el número de pasos que se moverá la malla. Los nodos se desplazan un paso por unidad de tiempo y se puede cambiar la velocidad de la animación resultante. El usuario también ingresa el algoritmo de verificación que desea aplicar, es decir, cómo se tratarán las posibles inconsistencias que se producirán al desplazar los nodos de la malla. Uno de las opciones posibles en no realizar ninguna verificación, es decir, deformar la malla aunque ésta quede inconsistente.

Refinar malla: Se puede refinar la malla, es decir, aumentar el número de caras de la malla disminuyendo el área de las caras originales. El usuario elige el algoritmo de refinamiento y el criterio de detención del algoritmo.

Desrefinar malla: Se puede desrefinar la malla, es decir, disminuir el número de caras de la malla aumentando el área de las caras sobrevivientes. El usuario elige el criterio de detención del algoritmo de desrefinamiento.

Mejorar malla: Se puede aplicar un algoritmo de mejoramiento de la calidad de la malla. El algoritmo implementado es el que usa el criterio de Delaunay.

Ver información malla: Se puede desplegar un cuadro con información sobre la composición de la malla (número de caras, área promedio de las caras, etc.). Esta información es de suma importancia para escoger los parámetros adecuados al aplicar algoritmos de refinamiento y desrefinamiento.

2.2.2. Diseño

La aplicación del legado fue diseñada siguiendo el paradigma de *Programación Orientada a Objetos*. En la figura 2.3 se muestra el diagrama de clases, en donde se aprecia el uso de patrones de diseño. A continuación se explican:

Command: En la aplicación, este patrón es utilizado para encapsular los requerimientos que el usuario realiza a través de la interfaz gráfica, de modo de separarlos de la capa lógica que se encarga de ejecutarlos. Las clases que cumplen el rol de comandos son *Generar*, *Guardar*, *Refinar*, *Desrefinar*, *Mejorar*, *Desformar* e *Informar*.

Strategy: Este patrón es utilizado para encapsular una familia de algoritmos de forma que sea fácil intercambiarlos y agregar nuevos algoritmos. En la aplicación, asociado a cada comando, salvo *Informar*, hay una familia de algoritmos encargados de satisfacer el requerimiento asociado al comando.

Además, se observa que en el sistema está bien separada la capa de presentación de la lógica.

Se consideran que hay dos importantes aspectos mejorables del diseño. El primero es la clase *Cara*, que representa una cara de cualquier número de lados, pero que tiene métodos que sólo son válidos para triángulos. El segundo es la clase *Malla*, que tiene más de 50 métodos públicos, sin contar constructores, *setters* ni *getters*. Esto afecta la mantenibilidad de la clase y se debe a que la malla es la única clase que tiene toda la información

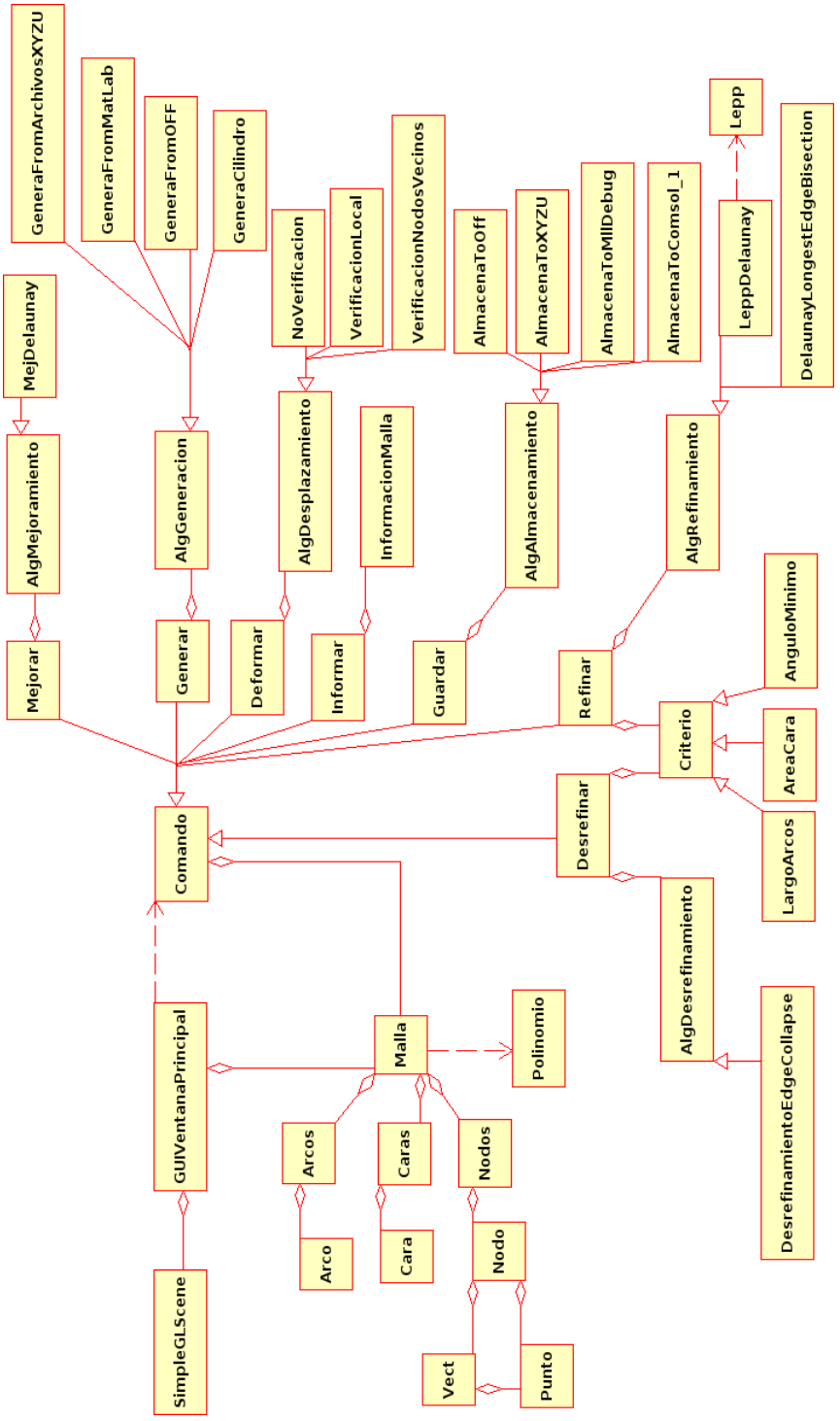

Figura 2.3: Diagrama de clases de la aplicación del legado

necesaria para realizar la mayoría de los cálculos necesarios. Esto se explicará en más detalle en la próxima sección.

2.2.3. Representación de la malla

A continuación se explica la representación de la clase *Malla*, la más importante del sistema y, a su vez, la representación de las clases que la componen.

Malla: Básicamente, se representa por tres contenedores: de nodos, arcos y caras.

Nodo: Se representa mediante un punto, un vector normal y una concentración. Por razones de eficiencia, mantiene además vectores de índices de los arcos y caras de los que forma parte. Los índices, tanto en esta clase como en las siguientes, son con respecto a los contenedores que almacena la malla.

Arco: Se representa mediante índices de los dos nodos que lo conforman y de las dos caras de las que forma parte.

Cara: Se representa mediante vectores de índices de los nodos y arcos que la conforman. Además, almacena su número de lados y color.

La aplicación trabaja con clases contenedoras: *Nodos*, *Arcos* y *Caras*, que proveen métodos para borrar, agregar y obtener elementos, entre otros métodos.

De lo anterior se deduce que sólo la clase *Malla* tiene referencias a los nodos, arcos y caras. Las otras clases almacenan índices, pero no tienen acceso a los contenedores referenciados por los índices. Por lo tanto, un arco no es capaz de calcular su largo, pues no conoce la posición de los nodos que lo forman, sólo los índices de los nodos. Esto hace que las clases *Nodo*, *Arco* y *Cara* estén compuestas principalmente por constructores, *setters* y *getters* y que los comportamientos de estas clases estén implementados en la clase *Malla*.

2.2.4. Algoritmos

La aplicación implementa una serie de algoritmos para modificar la malla. A continuación se explican.

Verificación desplazamiento: Al deformar la malla, cada nodo se desplaza proporcional a su concentración, en la dirección de su normal. Estos desplazamientos pueden producir colisiones, por lo que se implementan tres algoritmos para tratarlas:*No Verificación*, *Verificación Local* y *Verificación Nodos Vecinos*. El primero de ellos no realiza ninguna verificación, por lo que simplemente desplaza la malla.

El algoritmo *Verificación Local* recorre pares de caras vecinas (comparten un arco) coplanares y detecta si hay intersección entre sus arcos y, de haberla, intenta corregir esta inconsistencia haciendo *flipping* de arcos.

El algoritmo *Verificación Nodos Vecinos* recorre los nodos y para cada nodo, encuentra el polígono formado por los arcos de las caras incidentes en el nodo, de forma que el nodo queda en el centro del polígono. Luego, construye un pseudocilindro en que la base inferior es el polígono y la base superior está formada por las posiciones futuras de los vértices del polígono (luego del desplazamiento), de forma tal que las aristas verticales del cilindro son las trayectorias de los vértices. El algoritmo determina si la trayectoria del nodo central colisiona con alguna de las paredes del cilindro. Si es así, corrige la trayectoria del nodo central, haciendo que la dirección de la normal del nodo sea la suma de las normales de los vértices del polígono. Si este proceso no corrige la inconsistencia, entonces borra el nodo central.

Desrefinamiento: El algoritmo *Desrefinamiento Edge Collapse* recorre la malla buscando caras que no cumplan el criterio de detención. Luego detecta el menor arco de esa cara y lo colapsa, eliminando las dos caras adyacentes a él.

Mejora: El algoritmo *Mejora Delaunay* recorre la malla buscando pares de triángulos vecinos que no cumplan el criterio de Dalaunay (es decir, existen vértices al interior del círculo circunscrito en él). Luego reemplaza el arco común de los triángulos por el otro posible, haciendo que ambos triángulos cumplan con el criterio.

Capítulo 3

Diseño

Para incorporar a la aplicación mallas de cuadriláteros se hace necesario modificar la clase *Cara*. En la aplicación del legado, esta clase está representada por vectores de índices de nodos y de arcos y por un entero que indica el número de lados de la cara. Varios de los métodos de la clase son válidos sólo si la cara es triangular. El trabajar con vectores sin preocuparse de su largo, es decir, del número de lados de la cara, hace que los métodos sean más generales y por lo tanto más reusables. Además, en muchos casos acortan la extensión del código. Sin embargo, es incorrecto conceptualmente tener las caras de cualquier número de lados en una misma clase siendo que no tienen siempre el mismo comportamiento, lo que se evidencia en la implementación del legado en que hay métodos de la clase que sólo son válidos para triángulos.

La necesidad de crear una jerarquía se hace más clara aún al querer extender los algoritmos de refinamiento, desrefinamiento y mejoramiento a mallas de cuadriláteros. Estos algoritmos son fuertemente dependientes de la forma de las caras de la malla y por lo tanto no es cierto que funcionen para caras de cualquier número de lados. Por ejemplo, el algoritmo de mejoramiento utiliza el *criterio de Delaunay*, que se basa en propiedades geométricas de los triángulos.

Corresponde entonces crear una jerarquía para la clase *Cara* donde ella es la clase base y las clases *Triangulo* y *Cuadrilatero* son derivadas, dejando además abierta la posibilidad a la incorporación de nuevas clases (caras con más de cuatro lados).

El primer enfoque que se tomó para llevar a cabo este rediseño fue hacer que la clase *Cara* fuera una clase abstracta que incluya todos los métodos que representan el comportamiento de una cara, independiente de su número de lados. Los métodos que representan el comportamiento específico de un triángulo o cuadrilátero, deben estar en su correspondiente clase derivada. Esta estrategia implica trabajar en dos ámbitos distintos: revisar todos los métodos de la clase *Cara* e identificar y modificar los que utilizan que se trata de un triángulo pero son generalizables (por ejemplo, no iteran sobre los nodos de la cara, sino que trabajan con tres variables, una para cada nodo). El segundo ámbito a abordar corresponde a detectar la creación de objetos de la clase *Cara* y cambiarla a la creación de objetos de la clase *Triangulo* o *Cuadrilatero*, según corresponda.

Este enfoque finalmente fue modificado pues algunos de los formatos desde los que se carga la malla soportan el uso de caras de más de cuatro lados y los archivos de ejemplo con los que se trabaja incluyen esos casos. Se prefirió entonces no restringir la aplicación y seguir permitiendo la visualización de este tipo de mallas, aunque no exista la implementación de los algoritmos para modificar esas mallas. Por lo tanto, se decidió que la clase *Cara* no sería abstracta, sino que se crearían instancias para representar caras de más de cuatro lados y que los algoritmos que modifican la malla y que son propios de triángulos o cuadriláteros trabajarían con objetos de la clase derivada correspondiente, de forma de mantener la correctitud conceptual de la aplicación.

Capítulo 4

Implementación

4.1. Algoritmos para Mallas de Cuadriláteros

Como se explicó anteriormente, la creación de la jerarquía de la clase *Cara* tiene amplias repercusiones en el código de la aplicación, implica cambiar la implementación existente y agregar nuevos algoritmos A continuación se explica cómo se llevó a cabo en cada una de las partes relevantes de la aplicación.

4.1.1. Generación de malla

La malla puede ser generada desde una médula ingresada por el usuario, o generada en forma cilíndrica. Para el caso de la malla cilíndrica, la aplicación del legado tenía implementada la opción con mallas de cuadriláteros, así es que sólo hubo que cambiar la creación de los objetos de la clase *Cara* por objetos de las clase *Triangulo* o *Cuadrilatero*, según correspondiera.

La generación de la malla a partir de una médula se extendió para el caso de mallas de cuadriláteros. Para esto, primero se modificó la interfaz para permitir la elección entre una malla de triángulos y una de cuadriláteros. Luego se agregó un método que genera la malla de cuadriláteros. El algoritmo consiste en generar los anillos de la malla desde abajo hacia arriba. Al tener un anillo, se itera sobre el número de puntos del anillo. Para cada punto, se reconocen cuatro nodos: el actual, el siguiente en el mismo anillo, el superior al actual y el superior al siguiente del actual, como se muestra en la figura 4.1. Con esos nodos se crea el cuadrilátero, creando también los correspondientes arcos.

4.1.2. Carga de malla

La malla puede ser cargada en cuatro formatos, para todos ellos hay que adaptar el algoritmo de lectura para que trabaje también con mallas de cuadriláteros.

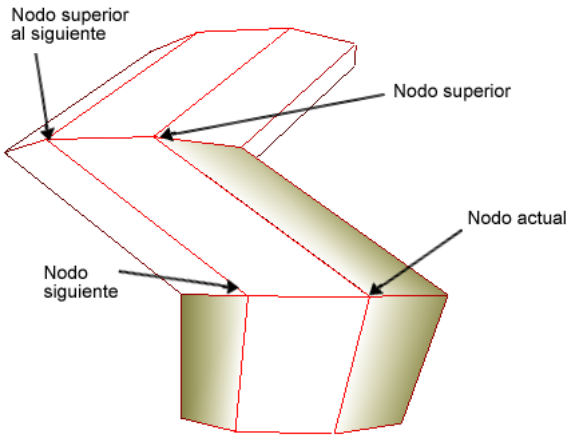

Figura 4.1: Generación de caras cuadriláteras a partir de una médula

En el caso del formato *Matlab* ya existía un método para cargar mallas de cuadriláteros. El formato almacena nodos y su ubicación en anillos alrededor de una médula implícita, por lo que un mismo archivo puede representar una malla de triángulos o de cuadriláteros, dependiendo de cómo se unen los nodos mediante arcos. Es por esto que el usuario elige el tipo de malla a crear a partir de un archivo en este formato. La aplicación del legado ya tenía implementada esta funcionalidad, pero se detectó y solucionó un error que dejaba la malla inconsistente (más adelante se detallará).

En el caso de los otros formatos, la estrategia usada fue generalizar el algoritmo existente, en vez de crear otro para el caso de cuadriláteros. De esta forma, se evita duplicar el código y por lo tanto se ayuda a la mantenibilidad de la aplicación. La generalización consistió en detectar el código común para ambos tipos de cara, dejarlo paramétrizado con respecto al número de lados (tres ó cuatro) y separar las partes diferentes según el tipo de cara, por ejemplo, la creación del objeto *Triangulo* o *Cuadrilatero*.

El formato *Geomview* ya estaba generalizado y ahora se crean objetos *Triangulo*, *Cuadrilatero* o *Cara* según si el número de lados de la cara es tres, cuatro o mayor, respectivamente.

En los formatos *Comsol* y *nxyzu* la generalización es válida para caras triangulares y cuadriláteras. No se consideró necesario extenderlo también a caras de más lados, pues ese caso no se presenta en las mallas de ejemplo que se tienen (a diferencia del formato *Geomview*), pero se considera que ya hecha esta primera generalización, las posteriores no presentan gran dificultad.

Para estos dos últimos formatos se dejó también la opción de que la malla sea mixta, es decir, que esté formada por caras triangulares y cuadriláteras. Sin embargo, al igual que en el caso de caras de más de cuatro lados, la aplicación no provee algoritmos para modificar esas mallas, por lo que sólo pueden ser visualizadas.

4.1.3. Guardado de malla

Para cada uno de los cuatro formatos de salida de la malla fue necesario adaptar los algoritmos de guardado para que iteraran sobre los nodos y arcos sin suponer que su cardinalidad era tres. Además, en el caso de los formatos *Comsol* y *nxyzu* se cubrió el caso en que la malla es mixta.

4.1.4. Algoritmo de eliminación de un nodo

La eliminación de un nodo de la malla es una operación que la aplicación del legado utiliza al deformar la malla, cuando se produce una inconsistencia que no puede ser reparada cambiando el vector normal del nodo (como se explicó en la sección 2.2.4). Además se utiliza para desrefinar la malla.

El algoritmo para triángulos, implementado en el método *VertexDeletion*, puede entenderse como el colapso de un arco en un vértice. Esto se traduce en que se elimina el arco y uno de sus nodos y el otro nodo "hereda" las caras y arcos del nodo borrado. Para esto se eliminan las dos caras que estaban conectadas por el arco borrado y se modifican las otras caras a las que pertenecía el nodo borrado, como se muestra en la figura 4.2

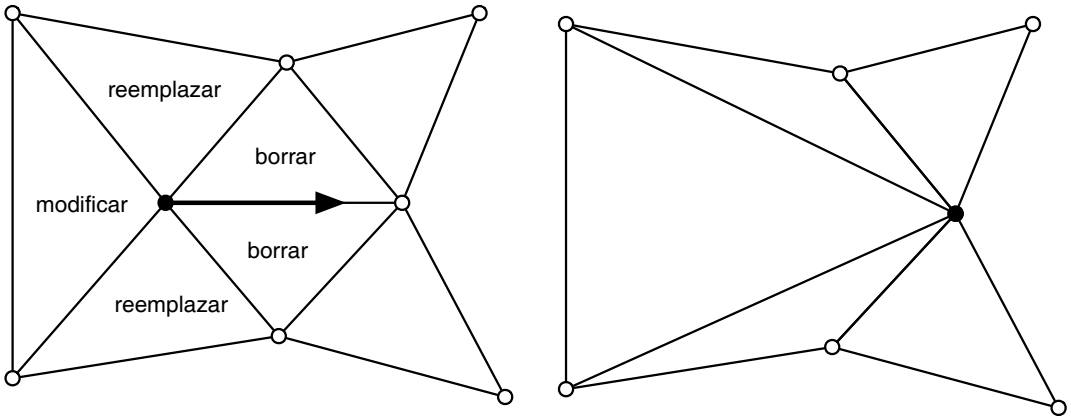

Figura 4.2: VertexDeletion para caras triangulares

El primer enfoque adoptado para adaptar este algoritmo para cuadriláteros fue encontrar la analogía con el caso de triángulos. Se identificaron los mismos actores que en el caso anterior: caras a borrar, caras a modificar, arcos a mantener, etc. y se quiso proceder igual que en el caso triangular, como se explica en la figura 4.3.

Sin embargo, este procedimiento no logra los resultados deseados. Para entender lo que ocurre, se puede volver a la explicación intuitiva sobre el procedimiento en una malla triangular: al colapsar un arco en un nodo, a su vez se colapsa en un arco las dos caras adyacentes al arco borrado. Esto no puede ser replicado en un cuadrilátero, pues al colapsar un arco, se forma un triángulo. Dejar la malla mixta no es una opción pues no permite

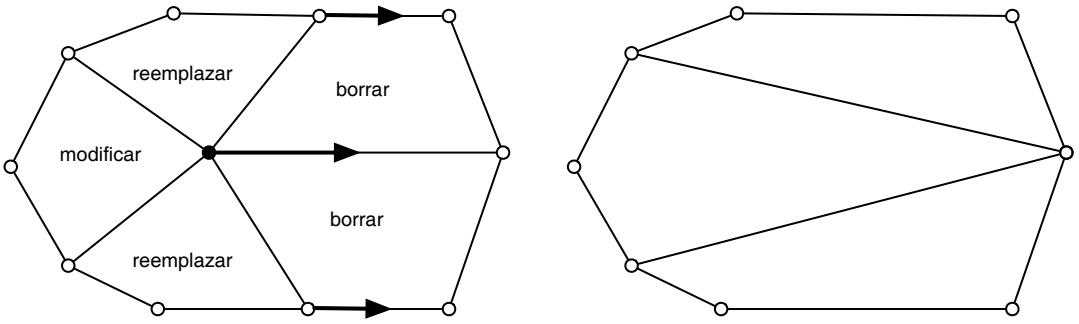

Figura 4.3: Generalización VertexDeletion para cuadriláteros

aplicar ninguno de los algoritmos de modificación de la malla que están implementados. La segunda opción es colapsar dos arcos, en vez de uno, borrando dos de los nodos pertenecientes a las caras a borrar, como se indica en la figura 4.4. En la figura, se muestra una malla inconsistente pues hay un par de arcos en cuya intersección no hay un nodo. Por lo tanto, el borrar estos dos nodos, adicionales con respecto al algoritmo para triángulos, conlleva a modificar también el entorno de esos nodos, lo que hace que el problema deje de ser local y se reproduzca recursivamente, pudiendo no tener solución.

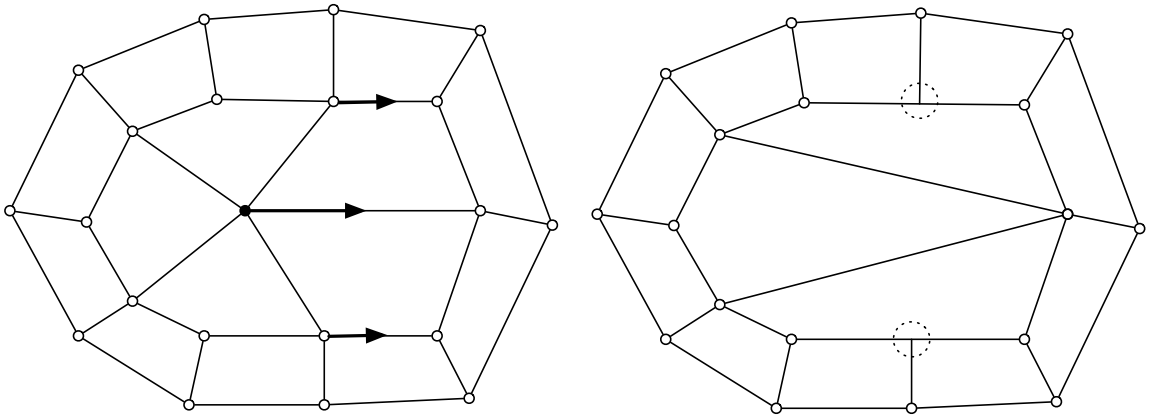

Figura 4.4: Problema al generalizar VertexDeletion para cuadriláteros

Otra forma de entender esto es dándose cuenta de que el algoritmo para triángulos lo que hace es eliminar un nodo y triangular el polígono que rodeaba al nodo (el formado por los arcos no incidentes en el nodo, de las caras incidentes en el nodo). Siempre es posible realizar este procedimiento pues siempre es posible triangular un polígono [1]. En cambio, no siempre es posible generar una malla de cuadriláteros en un polígono. De hecho, no se puede generar una malla de cuadriláteros en un polígono cuyo número de arcos es impar.

Por otra parte, incluso si sólo se trabajase con polígonos cuyo número de lados es par, igualmente surgen problemas cuando hay arcos adyacentes colineales, lo que se tiene, por

[1]El algoritmo realiza una triangulación particular que en casos específicos no puede efectuarse, pero se podría hacer otra triangulación

ejemplo, al generar la malla de un cilindro. En la figura 4.5 se observa que al borrar el nodo central y generar la malla de cuadriláteros, quedan colineales arcos adyacentes y pertenecientes a la misma cara, lo que es inconsistente.

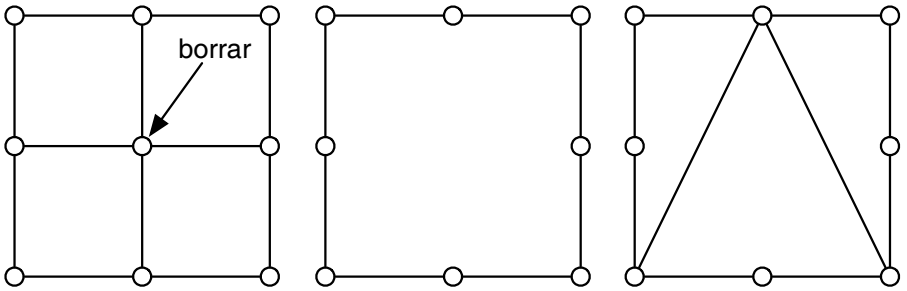

Figura 4.5: Inconsistencia al generar malla de cuadriláteros: hay arcos colineales

Una posible solución a este problema es desplazar el nodo que une los arcos colineales de modo de que dejen de serlo. El desplazamiento no puede ser infinitesimal pues se pueden producir errores numéricos. Este enfoque tiene dos desventajas: puede convertir en cóncava la cara vecina (si las dos caras se encuentran en la misma situación) y puede generar caras que no pertenezcan a un plano, como se muestra en las figuras 4.6 y 4.7.

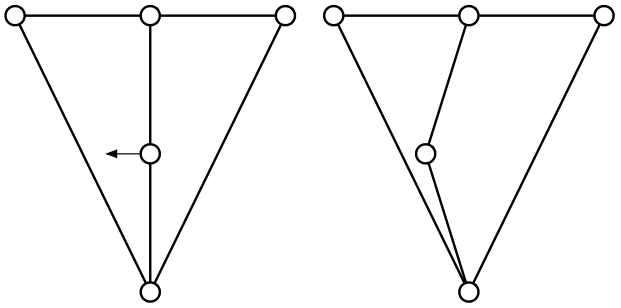

Figura 4.6: Una cara es cóncava y la otra convexa

Considerando todo lo anterior, se concluyó que no es posible utilizar para cuadriláteros un algoritmo análogo al usado para triángulos, por lo que se hizo necesario evaluar las implicancias de no contar con este procedimiento para mallas de cuadriláteros. Para el caso del algoritmo de verificación, se decidió implementar un nuevo algoritmo que en vez de colapsar un arco en un punto, colapsa en un punto todas las caras incidentes en el nodo. Para el caso del algoritmo de desrefinamiento, se decidió adoptar otra estrategia que no requiere de esta operación. Ambas decisiones se discutirán en los capítulos siguientes. En la sección siguiente se explicará el nuevo algoritmo que colapsa una región en un nodo.

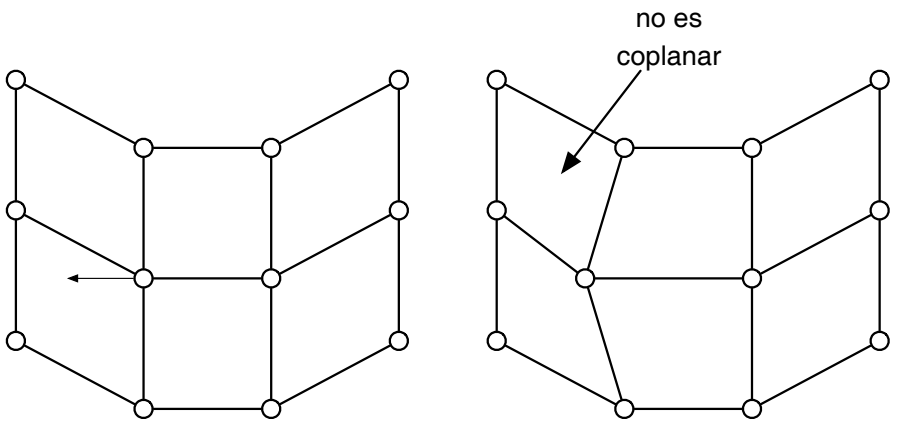

Figura 4.7: Caras dejan de ser coplanares

4.1.5. Algoritmo de colapso de una región en un nodo

La idea del algoritmo es colapsar en un punto las caras aledañas a un nodo. Este procedimiento tiene repercusiones en caras no adyacentes al nodo, por lo que en la práctica el algoritmo colapsa una región que rodea al nodo central.

El algoritmo detecta el polígono que rodea al nodo y desplaza sus vértices hacia la posición de este nodo central, colapsando el arco que los une y eliminando de esta forma las caras que rodeaban al nodo.

En la figura 4.8 aparece una malla triangular en donde el polígono que se muestra destacado será colapsado en el nodo n. En el primer paso, el nodo n_1 se desplaza para superponerse al nodo n, es decir, se colapsa el arco que une ambos nodos. Al hacer esto, desaparecen las caras c_1 y c_2. Pero además, este movimiento modifica las caras c_3 y c_6, vecinas de las caras borradas. Otras dos caras, c_4 y c_5, aumentan su tamaño.

En el siguiente paso, el nodo n_2 se desplazará hacia n. Esto borrará la caras c_3. Repitiendo el procedimiento, se obtendrá el resultado mostrado en la figura 4.9. El algoritmo puede resumirse como la modificación de tres grupos de caras que forman anillos alrededor del nodo: las que lo rodean directamente, las que son vecinas a las anteriores y las que rodean a éstas últimas. Los dos primeros grupos de caras desaparecen, mientras que las caras del tercero aumentan su área, ocupando el espacio que dejaron las caras borradas, lo que se explica en la misma figura. El algoritmo no tiene repercusiones en caras más lejanas pues el polígono formado por las caras del tercer grupo no cambia, por lo que este algoritmo, al igual que el que colapsa un arco en un nodo, modifica localmente la malla y puede entenderse como una nueva triangulación de un polígono que rodea al nodo, salvo que esta vez se considera un polígono más grande.

Para el caso de una malla de cuadriláteros, el procedimiento es similar: se desplazan hacia el nodo algunos vértices del polígono que rodea al nodo. Los vértices del polígono

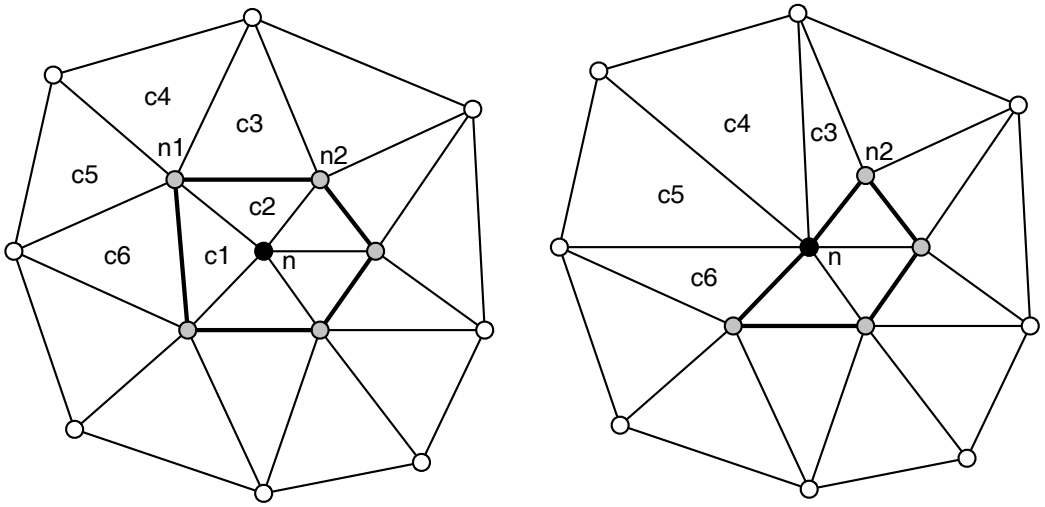

Figura 4.8: Desplazamiento del nodo $n1$ hacia n

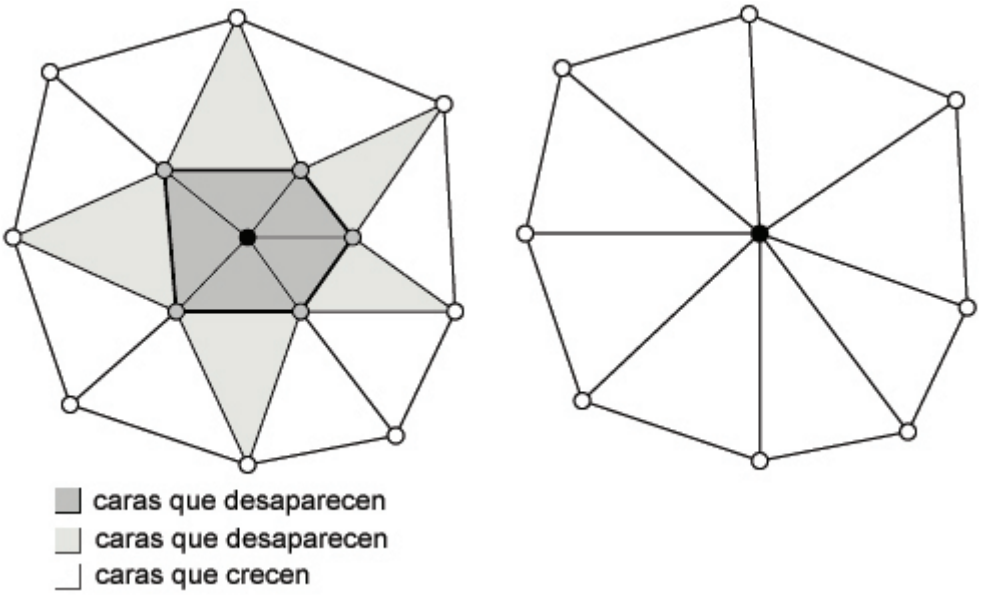

Figura 4.9: Algoritmo de colapso de una región de triángulos

que se desplazan son los que pertenecen a los arcos incidentes en el nodo central (los que en una malla triangular son todos los vértices del polígono). En este caso, a diferencia del caso triangular, se eliminan sólo las caras que rodean directamente al nodo. Algunas de las caras que forman el segundo anillo alrededor del nodo aumentan su área, ocupando el espacio dejado por las caras borradas, como se muestra en la figura 4.10.

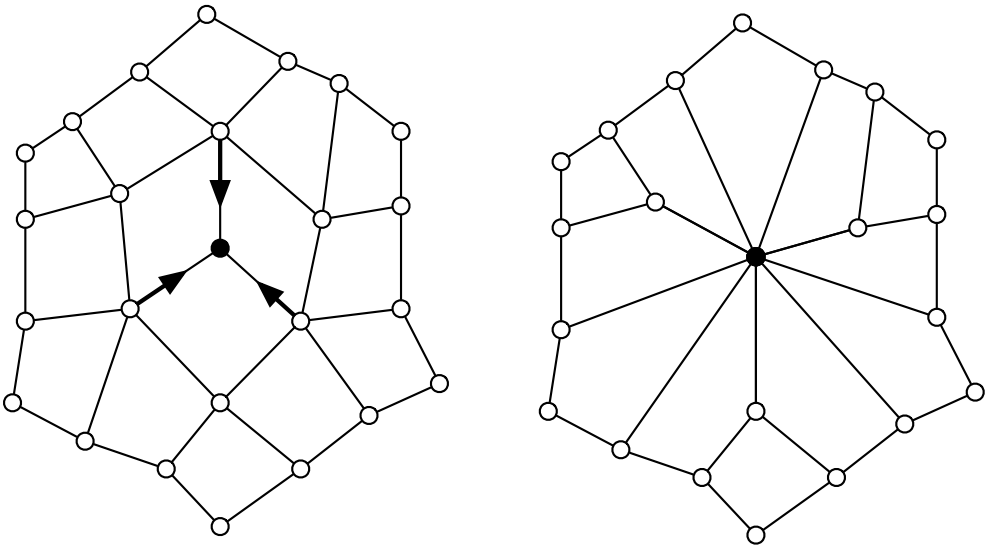

Figura 4.10: Algoritmo de colapso de una región de cuadriláteros

En el algoritmo explicado en la sección anterior, el que una de las caras que se borran tuviera dos caras vecinas que además son vecinas entre ellas produce un caso degenerado. Para evitar que ocurra, si la malla está en esa situación, el procedimiento no se aplica.

En este nuevo algoritmo, este caso no es problemático para el caso de mallas triangulares. Para el caso de mallas de cuadriláteros, el algoritmo puede ser aplicado en este caso, pero genera cuadriláteros cóncavos. En la figura 4.11 se observa que la cara c_1 será borrada y dos de sus caras vecinas, c_2 y c_3, son vecinas entre ellas. Al desplazar los nodos n_1 y n_2 hacia el nodo n, los arcos a_1 y a_2 se juntarán, por lo que las caras c_2 y c_3 compartirán dos arcos, lo que hace que o bien esos dos arcos son colineales, en cuyo caso estamos frente a dos cuadriláteros degenerados (tienen tres lados), o bien uno de los cuadriláteros es convexo y el otro cóncavo (los arcos forman dos ángulos, cada uno perteneciente a una de las caras, si uno de los ángulos mide menos de 180°, el otro debe medir más de 180°), como se muestra también en la figura.

Tener caras cóncavas es una desventaja, pero no hace que se descarte el algoritmo. De hecho, se considera que modificar la malla para que deje de tener caras cóncavas es una forma de mejorar la calidad de la malla, por lo que es un buen trabajo futuro incorporar algoritmos que trabajen en ese sentido.

En particular, se sugieren dos algoritmos de mejoramiento de la malla que abordan el problema. El primero de ellos consiste en reconocer pares de cuadriláteros vecinos, que

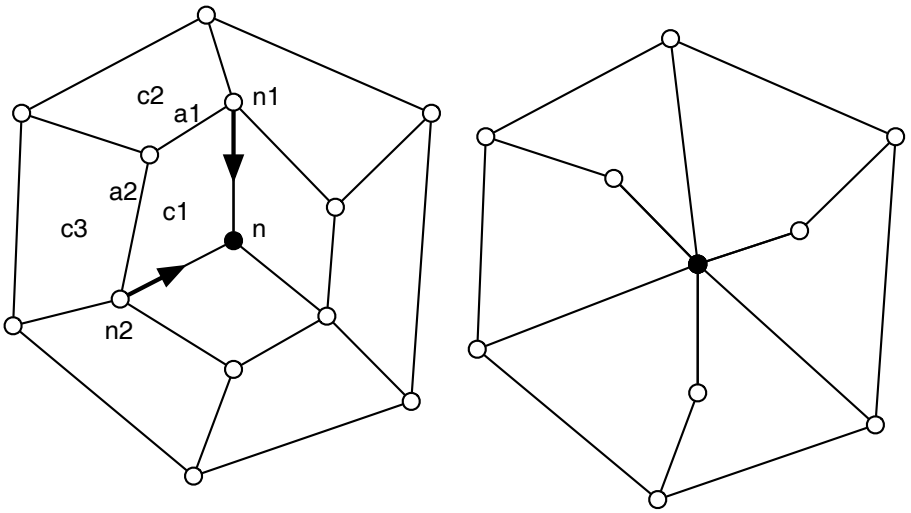

Figura 4.11: Aparición de caras cóncavas al colapsar región

comparten un solo arco, tales que al reemplazar el arco común por otro de los dos posibles (*flip de arcos*), ambos cuadriláteros queden convexos. En la figura 4.12 se muestra el resultado de aplicar dos veces esta idea, a partir de la malla producida al colapsar la región. Uno de los tres casos de concavidad del ejemplo no pudo ser solucionado por el algoritmo.

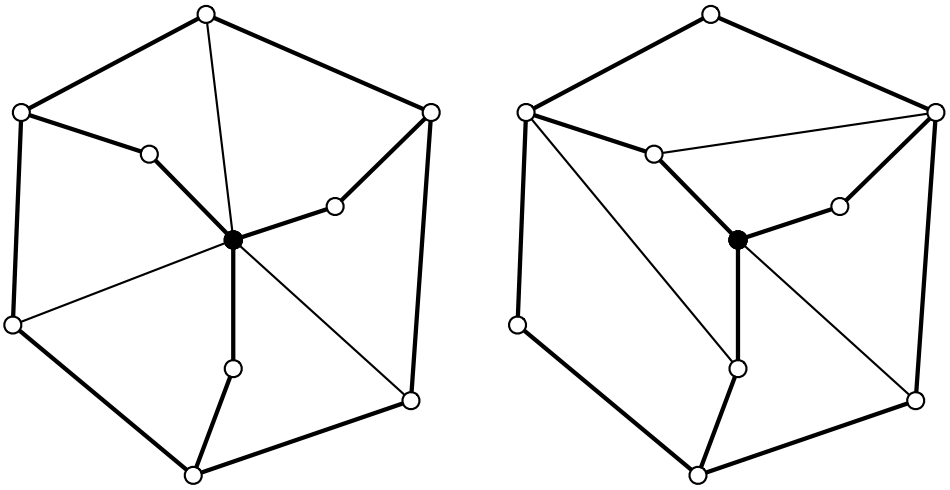

Figura 4.12: *Flipping* de arcos para mejorar calidad de la malla

Otro algoritmo posible consiste en reconocer pares de cuadriláteros que comparten dos arcos y eliminar esos dos arcos y el nodo que los une. Como los cuadriláteros compartían dos arcos, cada uno de ellos tiene otros dos arcos. La unión se los dos pares de arcos sobrevivientes forma también un cuadrilátero. Este enfoque mejora la calidad de la malla y también la desrefina, lo que no siempre es deseable.

Si se tiene un caso como el mostrado en la figura 4.11, en que todas las caras a borrar tienen caras vecinas que son vecinas entre sí, aplicar el algoritmo de colapso de la región y posteriormente aplicar este segundo algoritmo de mejoramiento produce un resultado similar al obtenido aplicando el algoritmo de colapso en una malla de triángulos.

4.1.6. Algoritmos verificación

Se extendió a mallas de cuadriláteros de algoritmo *Verificación de Nodos Vecinos*.

La primera parte de la adaptación consistió en generalizar la forma en que el algoritmo detecta que existirá una colisión al desplazar un nodo. El algoritmo crea un pseudo-cilindro en el que la base inferior es el polígono que rodea a un nodo central y la base superior se forma por las posiciones futuras de los vértices del polígono. Es decir, los arcos verticales del cilindro son las trayactorias de los nodos que forman el polígono. Luego detecta si la trayectoria del nodo central chocará con alguna de las caras del cilindro. Se generalizó la forma en que el algoritmo genera este cilindro, pues originalmente la implementación era válida sólo para el caso de mallas triangulares.

La segunda parte de la adaptación se relaciona con cómo el algoritmo soluciona una futura colisión. El primer intento consiste en modificar la trayectoria del nodo central. Si esto falla, entonces elimina el nodo central. Es aquí donde el algoritmo ocupa el método *VertexDeletion* que colapsa un arco en un nodo y que, como se explicó anteriormente, no es posible generalizar para mallas de cuadriláteros.

La estrategia adoptada entonces fue crear un nuevo algoritmo de verificación, válido para mallas de triángulos y para mallas de cuadriláteros. El algoritmo actúa igual que el anterior salvo en el momento en que la colisión no pudo ser solucionada mediante el cambio de la trayectoria del nodo central.

Si no se pudo solucionar la colisión, lo que se hace es colapsar en el nodo central la región que rodea a tal nodo. La intuición que explica esto es que el desplazamiento de los nodos corresponde al crecimiento del tronco del árbol. Si pese a cambiar la trayectoria del nodo sigue produciéndose la colisión, entonces no es responsabilidad del nodo, sino de las trayectorias de los nodos vecinos. Es decir, existe un conjunto de cédulas del árbol (caras de la malla) que interfieren entre sí por lo que se eliminan todas ellas, absorbidas por las células que las rodean y se mantiene el nodo.

Esta solución es más simétrica que la original, que colapsa un arco cualquiera de la región problemática, sobrecargando el nodo sobreviviente del arco colapsado, pues hereda los arcos sobrevivientes del nodo borrado.

Por lo tanto el nuevo algoritmo de verificación y solución de colisiones utiliza el algoritmo de colapso de una región en un nodo, explicado en la sección anterior.

4.1.7. Algoritmos desrefinamiento

El algoritmo de desrefinamiento *Edge Collapse* que está implementado para mallas triangulares en la aplicación del legado, recorre la malla y, para cada cara que no cumple el criterio de detención, colapsa el menor de sus arcos, eliminando las dos caras adyacentes al arco. Para ello utiliza el método *VertexDeletion* que no es extendible a mallas de cuadriláteros.

Un enfoque posible es utilizar el nuevo método que colapsa una región en un nodo. El problema de esto es que el método elimina todas las caras que rodean un nodo, es decir, no se puede asegurar que sólo se eliminan las caras pequeñas. El algoritmo original tiende a eliminar sólo las caras pequeñas pues de cada cara suficientemente pequeña (según el criterio elegido), elimina su arco más pequeño, con lo que además de eliminar la cara que cumple el criterio, elimina la otra cara adyacente al arco, pero como el arco es pequeño (el menor arco de una cara pequeña debe ser pequeño), la segunda cara también debe serlo.

Por lo tanto, si bien es factible hacer un algoritmo de desrefinamiento basado en eliminar regiones de la malla, probablemente su resultado no será satisfactorio pues elimina demasiadas caras, incluso si se tiene la precaución de no aplicar el método de eliminación sobre todos los nodos, por ejemplo, excluyendo los nodos que forman el borde de las regiones que ya han sido desrefinadas.

Entonces, el enfoque elegido es triangular la malla de cuadriláteros, añadiendo diagonales en los cuadriláteros; utilizar el algoritmo de desrefinamiento para mallas triangulares y finalmente hacer que la malla vuelva a ser de cuadriláteros.

4.2. Corrección de errores

Durante el desarrollo del trabajo de esta memoria se encontraron errores heredados de la aplicación del legado. Ellos tenían distintos niveles de impacto en la aplicación, pero sí impactaron de forma importante el desarrollo de ésta, pues en general eran errores que no se detectan en los casos de prueba usuales, por lo que en un principio fueron atribuidos a los cambios introducidos durante este desarrollo y se utilizó una gran cantidad de tiempo en hacer un seguimiento de los cambios para concluir finalmente que eran errores heredados.

Esta situación se produjo porque al cambiar cada parte de la aplicación se probó su funcionamiento fabricando mallas pequeñas de forma de poder garantizar que la malla generada era una instancia válida (revisando sus variables de instancia).

Se considera que fue valioso realizar este chequeo minucioso, pese al tiempo que utilizó, pues permitió detectar errores que no se notaban al visualizar la malla, lo que explica que no hayan sido corregidos cuando se desarrolló la aplicación del legado.

A continuación se explican los errores encontrados y solucionados:

Ejecución en modo *debug*: Uno de los métodos utilizados para graficar la malla chequeaba una precondición que generalmente no se cumplía. Por lo tanto, al ejecutar la aplicación en modo *debug* (con las instrucciones *assert* activadas), el programa se caía. Este error no se detectaba al ejecutar la aplicación en modo *release*.

Visualización la malla: Al cargar una malla en que todos los nodos tenían concentración nula, las caras se graficaban de color negro, en vez del color azul esperado. Esto se debía a que el color de los nodos es proporcional a la concentración del nodo con respecto a la máxima concentración de la malla. No se consideraba el caso en que ése máximo era cero, por lo que se producía una división por cero.

Deformación de la malla: En el cálculo de las normales de las caras faltaba normalizar el vector. Estas normales se utilizan para calcular la normal de los nodos, la que define la dirección del desplazamiento del nodo. Por lo tanto, el error hacía que la deformación de la malla no fuera la deseada.

Además, se encontró otro error al calcular la normal de los nodos. Ésta se calculaba como la suma de las normales incidentes en el nodo. En el ejemplo de la figura 4.13 se muestra una malla de seis caras triangulares, donde todas las diagonales van desde el nodo superior izquierdo al nodo inferior derecho. Cada nodo de la malla forma parte de tres caras. Dada la simetría del ejemplo, se espera que las normales de los nodos sean radiales y que los nodos pertenecientes a un mismo arco vertical tengan las mismas normales. Sin embargo no se obtenía eso, pues se consideraban de igual forma las tres normales de las caras, por lo que un vector se sumaba dos veces. Esto se corrigió considerando que el peso de cada cara en el nodo es proporcional al ángulo que forma la cara en ese nodo.

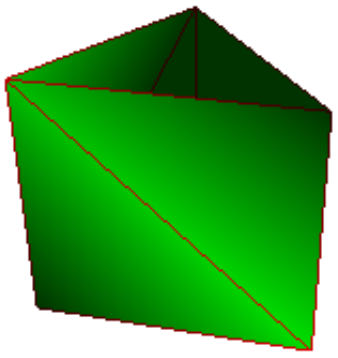

Figura 4.13: Malla de seis caras triangulares

Consistencia de la malla: En el método que estaba implementado para generar una malla de cuadriláteros a partir de un archivo de formato *Matlab*, se calculaba mal uno de los índices de los arcos que forman la cara. Esto hacía que la cara registrara

que estaba formada por un arco que no informaba que fuera parte de esa cara, pues efectivamente no lo era. Este error no se notaba al visualizar la malla, pues ahí se ocupa la información de los nodos que componen la cara y esa información estaba correcta. Sin embargo, en métodos que recorren los arcos de la cara el error se hacía presente. Se solucionó el error y además se modificó un método que chequea la consistencia de la malla, para que revisara también que todas las relaciones entre elementos de la malla sean simétricas, es decir, en ambos sentidos. Por ejemplo, si un arco se relaciona con una cara, la cara también se debe relacionar con el arco.

Detección de colisiones: En el algoritmo *Verificación de Nodos Vecinos* que detecta y soluciona las colisiones al deformar la malla no se cubría el caso en que algunos de los nodos de la malla no se desplazaran (tuvieran una concentración nula), por lo que la aplicación se caía al deformar la malla ocupando ese algortimo.

Capítulo 5

Conclusiones

Éstas son las conclusiones.

Bibliografía

[1] Ricardo Medina Díaz. Modelador de cambios en la geometría de objetos utilizando mallas geométricas. *Memoria para optar al título de Ingeniero Civil en Computación*, 2005.

[2] Nicolás Silva Herrera. Modelamiento del crecimiento de árboles usando mallas de superficie. *Memoria para optar al título de Ingeniero Civil en Computación*, 2007.

[3] Nancy Hitschfeld Kahler. Apunte de curso. *CC60R: Seminario de Geometría Computacional, clase 15. Página web: http://www.dcc.uchile.cl/ cc60q*, 2001.

[4] David Eppstein Marshall Bern. Quadrilateral meshing by circle packing. *Sixth International Meshing Roundtable (Park City, Utah), http://www.cs.berkeley.edu/ jrs/mesh/present.html*, pages 7–19, 1997.

[5] Stefan Schirra. Robustness and precision issues in geometric computation. *Versión preliminar de un capítulo de 'In Handbook of Computational Geometry', J.-R. Sack and J. Urrutia, Eds. Elsevier Science Publishers, B. V. North-Holland, Amsterdam, The Netherlands*, pages 1–26, 1999.

[6] Jonathan Richard Shewchuk. Lecture notes on delaunay mesh generation. *Página web: http://www.cs.berkeley.edu/ jrs/meshpapers/delnotes.ps.gz*, pages 3–5, 1999.

[7] Michael B. Stephenson Ted D.Blacker. Paving: a new approach to automated quadrilateral mesh generation. *International Journal for Numerical Methods in Engineering (ISSN 0029-5981)*, pages 811–812, 1991.

Apéndices

A . Formatos

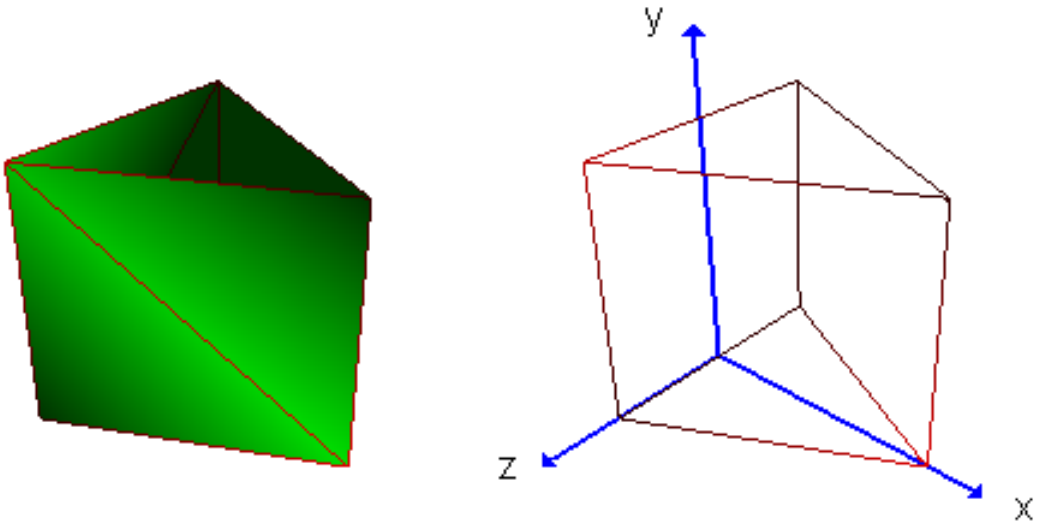

Figura 5.1: Malla que se representará en distintos formatos

La aplicación importa y exporta mallas en distintos formatos. A continuación se explican los formatos utilizados. Esta información se obtuvo analizando el código de la aplicación y los archivos de mallas y se registra aquí pues se considera que será de ayuda para futuras extensiones de la aplicación.

Se ejemplificarán los formatos con la malla mostrada en la figura 5.1.

Comsol - Extensión cms_1: El archivo se divide en dos secciones: coordenadas y elementos. La primera línea indica el inicio de la sección de coordenadas, a partir de la segunda línea, en cada línea se encuentran las coordenadas x,y,z de un nodo de la malla. Luego, hay una línea que indica el inicio de la sección de elementos y posteriormente en cada línea se encuentra la información de una cara. La cara se especifica indicando los índices 1-basados de los nodos que la componen, según el orden en que los nodos fueron especificados en la primera sección.

```
Archivo malla.cms_1
% Coordinates
10 0 0
0 0 5
0 0 -5
10 10 0
0 10 5
0 10 -5
% Elements (triangular)
2 5 1
5 4 1
3 6 2
6 5 2
1 4 3
4 6 3
```

Matlab - Extensión txt: Este formato se utiliza sólo para mallas tubulares (es decir, con anillos alrededor de una médula). Al cargar el archivo, el usuario indica el número de anillos de la malla y la cantidad de puntos por cada anillo. El archivo se divide en cuatro secciones. Primero están las coordenadas x de todos los nodos, luego las z y las y. Finalmente, están las concentraciones de los nodos. No son relevantes los saltos de línea en el archivo, pero para un mejor entendimiento, se tiene la convención de que cada línea del archivo tiene los datos de un anillo de la malla.

En este formato no existe una sección donde se indique qué nodos forman cada cara, sino que esto se deduce de la ubicación de los nodos en los anillos. Es por esto que el mismo archivo puede representar una malla de triángulos o de cuadriláteros, dependiendo de cómo se unen los nodos mediante arcos.

```
Archivo malla.txt
10 0 0
10 0 0
0 5 -5
0 5 -5
0 0 0
10 10 10
1 1 1
0 0 0
```

nxyzu - Extensión txt: La información de la malla se divide en cuatro archivos: *nx*, *ny*, *nz* y *u*. Por convención, al final del nombre del archivo se indica de cuál de los cuatro tipos es el archivo. Cada archivo tiene tres secciones. Las dos primeras secciones son igual al formato *Comsol* y son iguales en los cuatro archivos. La tercera sección tiene las coordenadas del vector normal y la concentración de cada nodo. En el archivo *nx* está la coordenada x de las normales de los nodos, en el mismo orden en que se listan los nodos en la primera sección. Es análogo para *ny* y *nz*. En el archivo *u* está la concentración de hormona en el nodo, es decir, el módulo del

vector desplazamiento del nodo.

Archivo malla_nx.txt	Archivo malla_ny.txt
% Coordinates	% Coordinates
10 0 0	10 0 0
0 0 5	0 0 5
0 0 -5	0 0 -5
10 10 0	10 10 0
0 10 5	0 10 5
0 10 -5	0 10 -5
% Elements (triangular)	% Elements (triangular)
1 5 2	1 5 2
1 4 5	1 4 5
2 6 3	2 6 3
2 5 6	2 5 6
3 4 1	3 4 1
3 6 4	3 6 4
% Data (nx)	% Data (ny)
1.000000	0.00000
-0.525731	0.00000
-0.525731	0.00000
1.000000	0.00000
-0.525731	0.00000
-0.525731	0.00000

Archivo malla_nz.txt	Archivo malla_u.txt
% Coordinates	% Coordinates
10 0 0	10 0 0
0 0 5	0 0 5
0 0 -5	0 0 -5
10 10 0	10 10 0
0 10 5	0 10 5
0 10 -5	0 10 -5
% Elements (triangular)	% Elements (triangular)
1 5 2	1 5 2
1 4 5	1 4 5
2 6 3	2 6 3
2 5 6	2 5 6
3 4 1	3 4 1
3 6 4	3 6 4
% Data (nz)	% Data (u)
0.000000	1.000000
0.850651	1.000000
-0.850651	1.000000
0.000000	0.000000
0.850651	0.000000
-0.850651	0.000000

Geomview - Extensión off: Las primeras dos líneas del archivo son informativas, la primera indica que se trata de un archivo en formato *off* y la segunda indica el número de nodos, caras y arcos de la malla. El número de arcos no se ocupa, sino que se deduce de la topología de la malla. Las siguientes líneas contienen las coordenadas de los nodos y las últimas líneas corresponden a las caras. El primer número de la línea de una cara indica el número de lados de la cara, luego se indican los índices 0-basados de los nodos que componen la cara. Opcionalmente se puede indicar también el color de la cara en formato *RGB*. La aplicación lee y almacena el color de la cara, si es proporcionado, pero no lo utiliza al graficar la malla. En el ejemplo, la mitad de las caras tienen color rojo y la otra mitad verde.

Archivo malla.off
OFF
6 6 12
10 0 0
0 0 5
0 0 -5
10 10 0
0 10 5
0 10 -5
3 0 4 1 1 0 0
3 0 3 4 1 0 0
3 1 5 2 1 0 0
3 1 4 5 0 1 0
3 2 3 0 0 1 0
3 2 5 3 0 1 0

Debug - Extensión mll: Este formato se utiliza para depurar la aplicación. En él se muestra toda la información de la malla. La primera línea indica que se que se trata de un archivo de formato *Debug*. La segunda línea indica el número de nodos, caras y arcos de la malla. La línea siguiente es similar a la anterior, pero se limita a los nodos, caras y arcos válidos (es decir, sin incluir los elementos borrados). A partir de la cuarta línea se listan los nodos, indicando las caras y arcos a los que pertenece el nodo, la normal y la concentración. Luego se listan los arcos, indicando los nodos que lo forman y las caras de las que forma parte. Finalmente se listan las caras de la malla, junto con el número de lados de la cara; los arcos que la conforman, los que a su vez indican los nodos que los conforman; la normal de la cara y la concentración de la cara.

Por razones de espacio, en el ejemplo se cambiaron las palabras *Nodos*, *Nodo*, *Caras*, *Cara*, *Arcos*, *Arco*, *Normal* y *Concentracion* por *N*s, *N*, *Cs*, *C*, *As*, *A*, *Nor* y *Con*, respectivamente.

Archivo malla.mll

```
MLL (Debug file)
6 6 12
6 6 12
N{0}=(10,0,0) Cs{0,1,4} As{0,2,3,10} Nor=(1,0,0) Con={1}
N{1}=(0,0,5) Cs{0,2,3} As{1,2,5,7} Nor=(-0.525,0,0.850) Con={1}
N{2}=(0,0,-5) Cs{2,4,5} As{6,7,9,10} Nor=(-0.525,0,-0.850) Con={1}
N{3}=(10,10,0) Cs{1,4,5} As{3,4,9,11} Nor=(1,0,0) Con={1}
N{4}=(0,10,5) Cs{0,1,3} As{0,1,4,8} Nor=(-0.525,0,0.850) Con={1}
N{5}=(0,10,-5) Cs{2,3,5} As{5,6,8,11} Nor=(-0.525,0,-0.850) Con={1}
A{0} Ns{0,4} Cs{0,1}
A{1} Ns{4,1} Cs{0,3}
A{2} Ns{1,0} Cs{0,-1}
A{3} Ns{0,3} Cs{1,4}
A{4} Ns{3,4} Cs{1,-1}
A{5} Ns{1,5} Cs{2,3}
A{6} Ns{5,2} Cs{2,5}
A{7} Ns{2,1} Cs{2,-1}
A{8} Ns{4,5} Cs{3,-1}
A{9} Ns{2,3} Cs{4,5}
A{10} Ns{0,2} Cs{4,-1}
A{11} Ns{5,3} Cs{5,-1}
C{0} 3 Ns{0,4,1} As{0:0-->4,1:4-->1,2:1-->0} Nor=(0.447,0,0.894) Con=3
C{1} 3 Ns{0,3,4} As{3:0-->3,4:3-->4,0:0-->4} Nor=(0.447,-0,0.894) Con=3
C{2} 3 Ns{1,5,2} As{5:1-->5,6:5-->2,7:2-->1} Nor=(-1,-0,-0) Con=3
C{3} 3 Ns{1,4,5} As{1:4-->1,8:4-->5,5:1-->5} Nor=(-1,0,0) Con=3
C{4} 3 Ns{2,3,0} As{9:2-->3,3:0-->3,10:0-->2} Nor=(0.447,0,-0.894) Con=3
C{5} 3 Ns{2,5,3} As{6:5-->2,11:5-->3,9:2-->3} Nor=(0.447,0,-0.894) Con=3
```