Índice general

1. Introducción **4**

1.1. Antecedentes Generales 4

1.2. Motivación 4

1.3. Objetivo General 6

1.4. Objetivos Específicos 6

1.5. Metodología 6

2. Antecedentes **8**

2.1. Generación de Mallas con Triangulación de Delaunay 8

2.2. GTKmm 9

2.3. OpenGL 10

2.4. Aplicación del Legado 10

2.4.1. Análisis Global 10

2.4.2. Análisis Detallado 13

2.4.3. Estructuras de Datos 18

2.4.3.1. Representación de la Malla 18

2.4.3.2. Otras Estructuras de Datos 21

2.4.4. Transformaciones Utilizadas en una Triangulación 21

2.4.5. Algoritmos de Desplazamiento 23

2.4.5.1. Desplazamiento sin Verificación 23

2.4.5.2. Desplazamiento con Verificación Local 23

2.4.6. Interfaz Gráfica 24

3. Análisis y Rediseño **26**

3.1. Rediseño Global 26

3.2. Rediseño de las Clases Asociadas a los Procesos Refinamiento y Mejoramiento 26

3.3. Rediseño de las Clases Asociadas a los Procesos de Almacenamiento de la Malla 29

3.4. Rediseño de Interfaz Gráfica 29

3.4.1. Rediseño Gráfico y de Usabilidad 30

3.4.2. Rediseño de Clases Asociadas al Proceso de Visualización 34

4. Implementación **36**

4.1. Ambiente de Trabajo 36

4.2. Uso de Bibliotecas en la Aplicación 36

4.3. Algoritmo de Lectura de la Malla Inicial 37

4.3.1. Lectura desde un Archivo de Texto Comsol 37

4.4. Información y Estadísticas de la Malla 40

4.5. Algoritmos de Transformación 41

4.5.1. Transformación Vertex-Deletion 42

4.5.2. Nueva Transformación Edge-Collapse 45

4.6. Algoritmo de Desrefinamiento 47

4.6.1. Desrefinamiento por Edge-Collapse 48

4.7. Algoritmos de Desplazamiento 48

4.7.1. Desplazamiento con Verificación de Nodos Vecinos 50

4.7.2. Ejemplos de Desplazamiento de Mallas 55

4.8. Visualización 57

4.8.1. Visualización OpenGL en Tiempo Real 59

4.8.2. Rotación y Traslación 60

4.8.3. Ajuste Automático de la Cámara 62

4.8.4. Animación 63

4.9. Corrección de Errores y Memory Leaks 64

4.9.1. Chequeo de Problemas de Memoria 64

4.9.2. Métodos Auxiliares para Chequeo de Consistencia 65

5. Discusión y Conclusiones **68**

5.1. Conclusiones 68

5.2. Trabajo Futuro 70

A. Diagrama de Clases de Patrones de Diseño **74**

B. Pseudocodigo Algoritmo de Lectura desde Comsol **76**

Capítulo 1

Introducción

1.1. Antecedentes Generales

Las mallas geométricas son herramientas muy utilizadas en la simulación de fenómenos modelados con ecuaciones diferenciales parciales. Una malla es un conjunto de celdas contiguas que sirven para representar el dominio del problema a modelar y sus variables físicas. En general, las celdas más usadas son triángulos o cuadriláteros en el caso de dos dimensiones (2D), y tetraedros o hexaedros en el caso de tres dimensiones (3D). Cuando se desea modelar el comportamiento de objetos, muchos de éstos presentan cambios o deformaciones en su geometría a través del tiempo. Para modelar el proceso de cambio de la geometría, se recalcula una nueva malla cada cierto tiempo. Al pasar de una malla a otra, muchas veces se pierden propiedades las cuales son necesarias de mantener. En algunos casos, si no se aplican las correcciones necesarias, las geometrías quedan inconsistentes debido a que se producen colisiones entre los elementos no vecinos de la malla. Debido a esto, la generación de una malla consistente y de buena calidad requiere de algoritmos eficientes y robustos que permitan resolver problemas de colisiones globales, locales, de inserción de puntos (Refinamiento) y de eliminación de puntos (Desrefinamiento).

1.2. Motivación

Una de las áreas donde se aplican los conceptos de cambios en la geometría de objetos es el modelamiento de ecosistemas de organismos vegetales como plantas y árboles. En este contexto, se deben verificar tanto interacciones endógenas como exógenas. En el caso de mi trabajo de título, el estudio está centrado en el modelamiento de cambios en la geometría de la superficie de troncos de árboles, en la cual la interacción es sólo endógena. La aplicación de este estudio será utilizada para modelar los efectos de la distribución de una determinada hormona en el crecimiento de un árbol.

Este trabajo consistirá en extender el trabajo de memoria realizada por Ricardo Medina[1]. En esta memoria se comenzó el desarrollo de una aplicación que modela el crecimiento de un

árbol. La aplicación recibe como entrada una malla de superficie cilíndrica generada a través de una figura predefinida o como output de un modelo de Femlab[1]. Esta geometría representa el tronco de un árbol. Una vez dada la geometría inicial y generada la malla que la representa, se recibe como información de entrada al sistema una perturbación de la geometría para cada instante de tiempo, para así poder modelar la evolución del tronco del árbol. Este trabajo, dejó un campo muy abierto para futuros temas de investigación y mejora de esta aplicación.

El propósito en esta memoria, es extender este trabajo haciendo un estudio del diseño de la aplicación, para mejorar el diseño y ajustarlo mejor a buenas prácticas de OOP[2]. Además, se incluirán mejoras y características que no se concretaron en el desarrollo anterior como:

- Refinamiento y mejoramiento de las mallas: Existen varios criterios de calidad que se pueden agregar a la aplicación, así como también otros algoritmos de refinamiento y mejoramiento. También se pueden incluir algoritmos de des-refinamiento de las mallas, es decir, remover puntos donde estos se encuentren muy concentrados.
- Mallas de cuadriláteros: En este momento las operaciones sobre la topología de la malla solo están definidas para mallas de triángulos. Seria interesante extender las operaciones para que soportaran mallas de cuadriláteros, las cuales se asemejan mejor a la forma natural de las células.
- Algoritmos de verificación de consistencia global de la malla: En este momento la verificación de consistencia solo se hace de manera local, lo cual puede provocar inconsistencias a nivel global de la malla, entre elementos que no se encuentren en una misma vecindad. Esto se puede llegar a generalizar para unir dos mallas distintas, manejando así la unión de varias mallas a solo una. Algo que se da en forma natural en los vegetales.
- Algoritmos de verificación local más eficientes: Se desea desarrollar algoritmos de verificación local más eficientes que los actuales, que sean capaces de controlar cada una de las inconsistencias de la malla.
- Visualización de los cambios o deformaciones de la malla en forma On-line: Incluir la capacidad de mostrar la malla a medida que los cambios van ocurriendo. Esto haría más atractivo el modelamiento ya que actualmente la visualización se muestra paso a paso. Es decir, mostrando un solo estado a la vez.
- Solucionar problemas de precisión e inexactitud de los cálculos: Estos problemas en la aplicación actual generan esporádicamente malos resultados y caídas de la aplicación.
- Comunicación directa con software Femlab: Actualmente la aplicación no interactúa en forma directa con Femlab. Solo funciona a través de exportaciones hechas desde Femlab a archivos de texto.

[1] Software de Calculo Numérico que provee el input para la aplicación que modela cambios a la geometría.
[2] Object Oriented Programming (Programación Orientada a Objetos)

1.3. Objetivo General

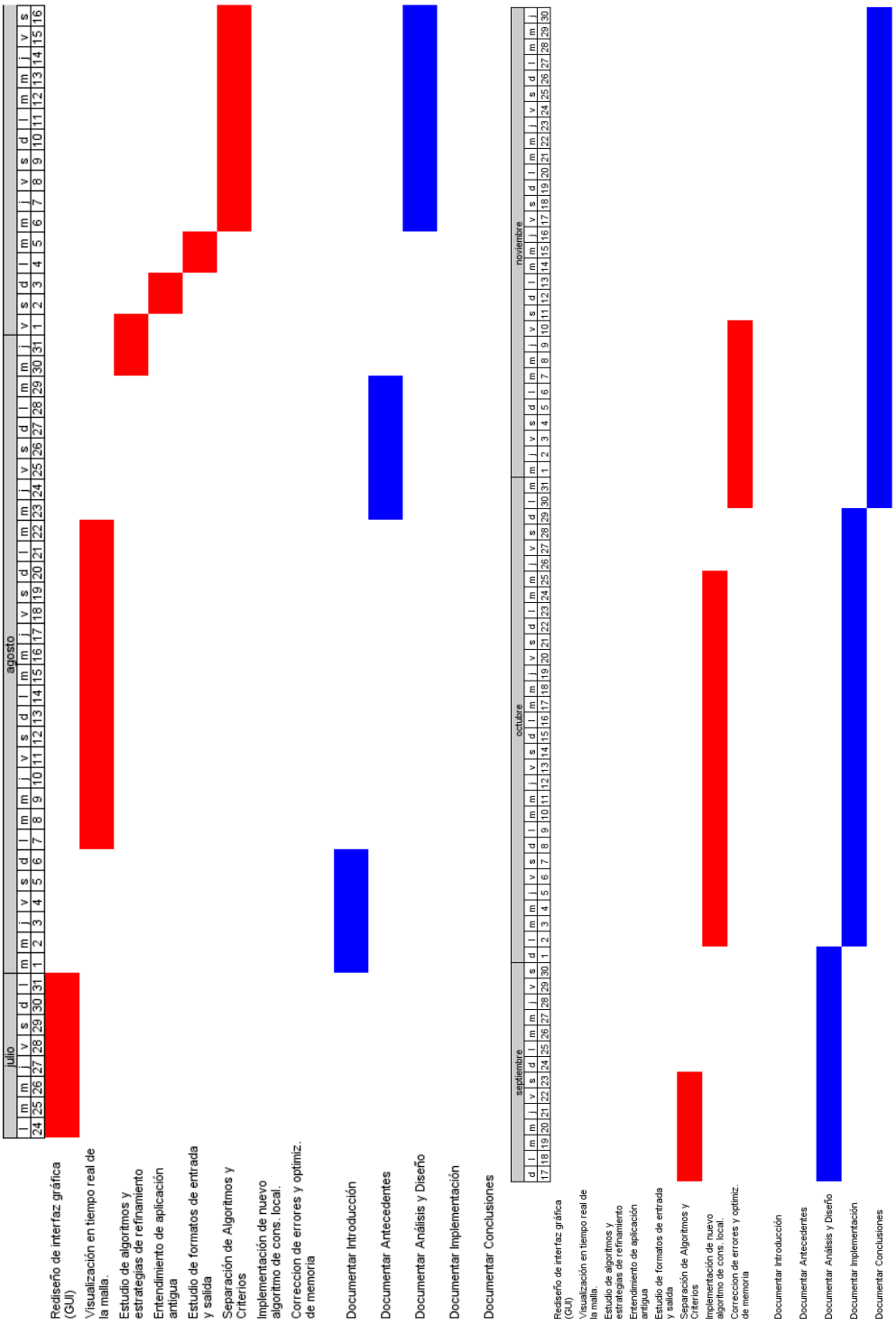
El objetivo general del presente trabajo es la extensión y rediseño de un modelador de cambios de la geometría de un objeto, en particular la de un árbol, que sea capaz de representar en forma real y resolver en forma eficiente colisiones locales en la malla y al mismo tiempo tener un buen diseño orientado a objetos que facilite la mantención y posterior desarrollo en el área.

1.4. Objetivos Específicos

- Revisión y mejoramiento del diseño actual.
- Reorganización y reestructuración de clases.
- Mejoramiento y corrección de algoritmos de transformación y desplazamiento de la malla.
- Implementación de algoritmo de manejo de colisiones a nivel de la vecindad de cada nodo.
- Inclusión de nuevo algoritmo de desrefinamiento.
- Rediseño y reimplementación de la interfaz gráfica de la aplicación.
- Implementación de opción de visualización de crecimiento de arboles, en forma animada.
- Corrección de errores y optimización del uso de memoria en la aplicación.

1.5. Metodología

A continuación se muestra los pasos seguidos en el trabajo para el cumplimiento de los objetivos. Estos pasos están orientados a tener un mejor entendimiento acerca de los temas involucrados en el trabajo y un acercamiento hacia las herramientas que serán utilizadas para el desarrollo de la aplicación. La planificación desarrollada se puede ver en la Tabla 1.1 en la página siguiente.



Cuadro 1.1: Planificación de la Memoria

Capítulo 2

Antecedentes

2.1. Generación de Mallas con Triangulación de Delaunay

Para la generación de una buena malla geométrica es importante formarla de tal manera de tener una triangulación que se ajuste al criterio de Delaunay. Este criterio dice que ningún otro punto debe estar dentro del circumcírculo de cada triangulo de la malla. La Triangulación de Delaunay tiene la gran característica de maximizar el ángulo mínimo de los ángulos de los triángulos de la malla. Esto hace que los triángulos sean más "gordos", obteniendo así una malla de mejor calidad en donde se minimizan los problemas de precisión.

Existen variados métodos para la generación de triangulación de de Delaunay. Entre estos métodos los más conocidos son el algoritmo incremental y el algoritmo de "Dividir y Conquistar".

La forma más intuitiva y directa de calcular la triangulación de Delaunay es utilizando un método incremental. Es decir, agregar un vértice a la vez y luego triangular las partes afectadas de la malla. Cuando se añade un vértice, se hace una revisión de los circumcírculos de los triángulos que contienen el vértice. Si no se satisface la condición de Delaunay, se hace un flip o intercambio de arcos para satisfacer la condición de Delaunay. Implementando este algoritmo en forma directa nos da un tiempo de ejecución de $O(n^2)$. Una manera habitual de hacer más eficiente este método es ordenar los vértices por su primera coordenada y luego agregarlos en ese orden. Utilizando este método el tiempo de ejecución promedio es de $O\left(n^{\frac{3}{2}}\right)$, sin embargo en el peor caso el algoritmo sigue siendo de $O(n^2)$.

El algoritmo de "Dividir y Conquistar" consiste en en recursivamente dividir los vértices de la malla en dos sets. La triangulación de Delaunay es calculada para cada set y luego unir ambos sets en una sola triangulación. Usando una buena implementación la operación de juntar ambos sets se puede hacer en $O(n)$, teniendo un tiempo de ejecución total de $O(n \log n)$.

La aplicación desarrollada en esta memoria permite mejorar mallas para que cumplan los criterios de la Triangulación de Delaunay.

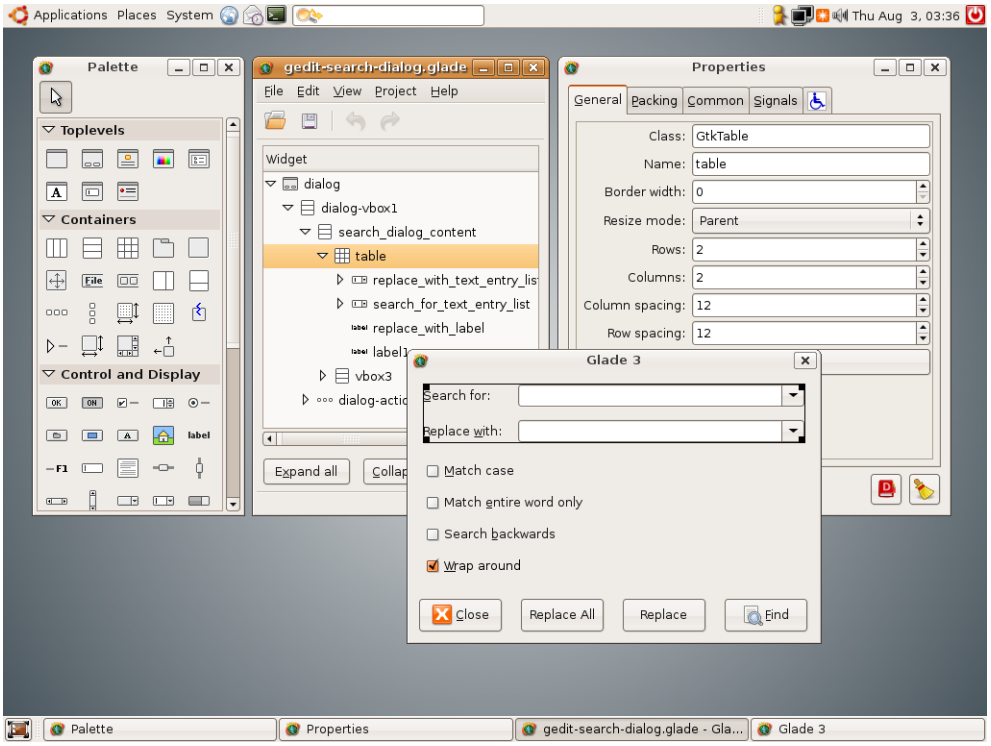


Figura 2.1: Diseñador de Interfaces Glade

2.2. GTKmm

Para la implementación de la interfaz gráfica se usará la librería GTKmm. GTKmm es la interfaz oficial en C++ de la popular librería GTK+[1]. Entre sus características destacan un diseño orientado a objetos, callbacks con verificación de tipos, widgets extensibles por herencia y una gran cantidad de widgets disponibles. Se pueden crear interfaces usando código o con el diseñador de interfaces Glade (Ver figura 2.1). GTKmm es una librería gratuita.

A diferencia de muchos otros toolkits de creación de interfaces, GTK+ no está basado en Xt. La ventaja de esto es que le permite a GTK+ estar disponible en otros sistemas y ser mucho más flexible. La desventaja está en no tener acceso a la base de datos de recursos de X, la cual es la forma tradicional de personalizar aplicaciones para X.

El usuario tiene la posibilidad de configurar el aspecto visual del toolkit, seleccionando entre una gran variedad de temas de visualización. Existen temas que emulan la apariencia de otras importantes plataformas como Windows, Motif, QT o NextStep.

Existen varios entornos gráficos de Unix que usan a GTK+ como base. Entre estos están el popular GNOME, además de XFCE, GPE, Maemo, etc. Estos ambientes gráficos no son requeridos para correr aplicaciones GTK+. Si las librerías que requiere el programa están

[1]GTK+ es conocido como el GIMP Toolkit. GTK+ es una de las dos librerías de interfaz gráfica disponibles para el X Windows System. GTK+ y QT han reemplazado al viejo Motif que previamente era la librería más usada.

instaladas, un programa GTK+ puede correr sobre otros ambientes gráficos de X como por ejemplo KDE con un muy buen rendimiento y velocidad.

Para la implementación del software de esta memoria, se decidió utilizar GTKmm. Se utilizará esta librería gráfica en vez de continuar con QT, ya que se integra mejor y tiene un mejor rendimiento para diferentes ambientes gráficos de Unix, a diferencia de QT donde el rendimiento no es muy bueno si el ambiente gráfico es diferente a KDE. Otra razón importante para la elección de GTKmm es su licencia LGPL la cual es absolutamente gratuita.

2.3. OpenGL

Para la visualización en tiempo real de las mallas se usan las librerías gráficas OpenGL. OpenGL es una especificación estándar que define una API multi-lenguaje multi-plataforma para escribir aplicaciones que producen gráficos 3D, desarrollada originalmente por Silicon Graphics Incorporated (SGI). OpenGL significa Open Graphics Library, cuya traducción es biblioteca de gráficos abierta.

Entre sus características podemos destacar que es multiplataforma (habiendo incluso un OpenGL ES para móviles), y su gestión de la generación de gráficos 2D y 3D por hardware ofreciendo al programador una API sencilla, estable y compacta. Además su escalabilidad ha permitido que no se haya estancado su desarrollo, permitiendo la creación de extensiones, una serie de añadidos sobre las funcionalidades básicas, en aras de aprovechar las crecientes evoluciones tecnológicas. Podemos reseñar la inclusión de los GLSL (un lenguaje de shaders propio) como estándar en la versión 2.0 de OpenGL presentada el 10 de agosto de 2004.

Siendo OpenGL multiplataforma puede encontrarse en una gran cantidad de plataformas (Linux, Unix, Mac OS, Microsoft Windows, etc.). Pero en Linux además encontramos la implementación Mesa.

Para la implementación del software de esta memoria se decidió utilizar OpenGL debido a que es la librería gráfica más popular en Linux, además de tener el mejor rendimiento. Además, tengo una gran experiencia en el manejo de esta librería en cursos de computación gráfica lo que me facilita bastante la implementación de aplicaciones basadas en esta librería.

2.4. Aplicación del Legado

2.4.1. Análisis Global

A continuación, se hará un análisis de lo que ya está implementado en el software llamado Modelador de Cambios. La revisión se llevará a cabo observando el diseño actual de la aplicación obtenido mediante técnicas de Ingeniería en Reversa. Las técnicas utilizadas consisten en generar diagramas de clases de la aplicación a partir del código fuente. Para esto, se utilizo

una útil herramienta llamada Umbrello la cual permite generar diagramas a partir de código escrito en C++.

Comenzaremos analizando el diagrama de clases global del sistema completo mostrado en la Figura 2.2 en la página siguiente. Este diagrama no contiene información de las operaciones, ni de las variables de instancia del sistema, pero nos permite tener una visión global de la implementación del sistema.

Al ver el diagrama simplificado podemos notar la utilización de dos patrones de diseño ampliamente conocidos[3]. Estos patrones son Command y Strategy. El diagrama de clases de estos patrones se puede ver en el Apéndice A.

- Command: Este patrón se utiliza para encapsular requerimientos basados en diferentes acciones a realizar. Un ejemplo clásico en donde se utiliza este patrón es el modelamiento de opciones de un menú. En este caso se tiene algo muy parecido, en donde el usuario envía requerimientos o acciones a realizar a nuestra interfaz gráfica. Este patrón utiliza una interfaz que agrupa mediante herencia cada uno de los comandos concretos. Cada comando provee un método llamado execute(), el cual es el responsable de ejecutar las acciones asociadas al comando. También participa en este patrón una o más clases que implementan las acciones asociadas al comando. Además, se usa una clase cliente que crea el comando especifico y una clase invocadora que llama al comando.
  Los comandos concretos son las clases Refinar, Generar, Deformar, Guardar y Visualizar. En el caso de los tres primeros, las clases que realizan la acción son los algoritmos específicos para cada comando. En el caso de los dos últimos, son ellos mismos los encargado de ejecutar la acción. La clase cliente e invocadora es la misma para todos los comandos y es ModeladorGUI.

- Strategy: Este patrón define y encapsula una familia de algoritmos que resuelven un mismo problema y permite intercambiar fácilmente el algoritmo que un cliente usa. Este patrón se adecúa perfectamente a los requerimientos en los cuales se plantean distintas formas de generar una malla, de refinarla y de desplazar sus nodos. Además del intercambio y la manera limpia de resolver el problema, esto permite agregar otros algoritmos que se quiera utilizar más adelante en la aplicación, permitiéndole ser altamente extensible. Este patrón define una interfaz común para cada clase que implementa un algoritmo, la cual consiste en un método que debe ser implementado en las diferentes clases. Además define una clase contexto la cual mantiene una referencia al algoritmo o estrategia específica y la aplica.
  Este patrón se utiliza para tres tipos de algoritmos distintos, los cuales se representan con las clases AlgRefinamiento, AlgGeneracion y AlgDesplazamiento. De la primera clase se derivan los algoritmos que refinan o mejoran una malla. De la segunda, los algoritmos que generan una malla inicial desde distintos formatos o figuras predefinidas. De la tercera, los algoritmos para modelar los cambios a la geometría del objeto. Las clases de contexto para estas clases son los comandos Refinar, Generar y Desplazar respectivamente. Estas clases mantienen una referencia a un algoritmo específico y se encargan de aplicarlo.

Una importante mejora a implementar es separar la implementación de la interfaz gráfica del resto de la aplicación. Vemos que en este momento el sistema es muy poco extensible

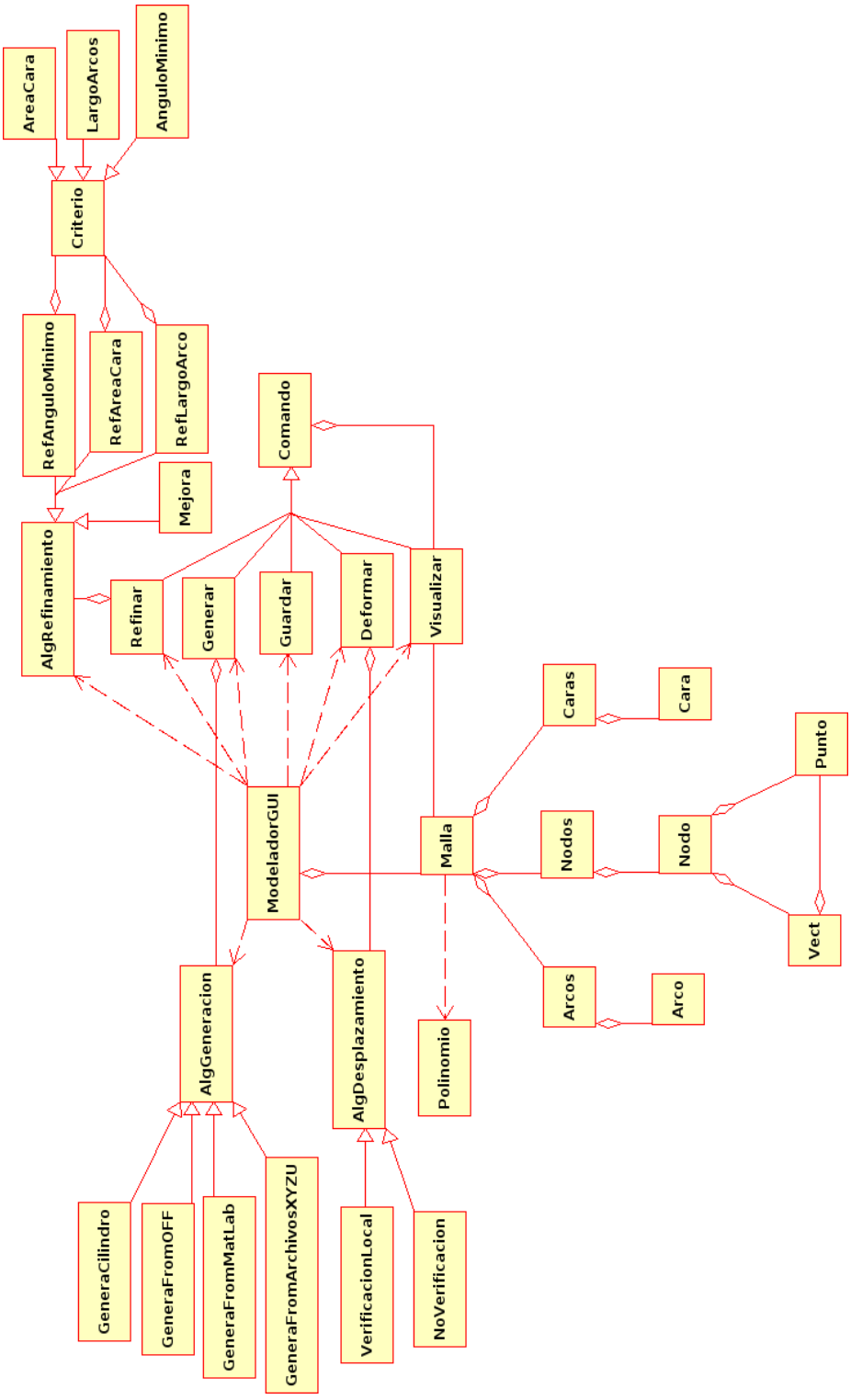


Figura 2.2: Diagrama de Clases Global del Sistema

para poder hacer un cambio a la interfaz gráfica. Vemos que la clase ModeladorGUI usa directamente varias clases aparte de las clases de comandos, como por ejemplo AlgGeneracion, AlgDesplazamiento y AlgRefinamiento. Esto da como resultado que la clase ModeladorGUI tenga que manejar información de muchas otras clases, además de tener responsabilidades que no le competen. ModeladorGUI solamente debería tener como responsabilidad desplegar información y recibir input del usuario. De esta forma lograríamos desligar la interfaz gráfica del resto de la aplicación, y al mismo tiempo, mantener un sistema más extensible para futuros cambios en las formas de desplegar la información.

Otro aspecto importante a incluir, es la implementación de algoritmos para el formato de salida de la malla. Mediante el patrón de diseño Strategy podemos mantener en forma extensible la inclusión de nuevos formatos de salida en los cuales se puedan guardar las mallas mejoradas y refinadas por el modelador de cambios.

Se evaluará también, la posibilidad de implementar nuevas formas de visualizar las mallas. Actualmente la aplicación solo permite visualizar las mallas a través del software Geomview. La aplicación será extensible para incluir nuevas formas de visualización de la información, como puede ser la visualización de las mallas en tiempo real en la misma aplicación mediante la utilización de OpenGL.

Además de los algoritmos de refinamiento y mejoramiento ya implementados, existen otros algoritmos y criterios de calidad que se pueden implementar. En el diseño actual, hay algoritmos asociados a criterios. Estos conceptos debieran estar separados.

La aplicación solamente tiene un algoritmo que verifica la consistencia local. En esta memoria se implementará un nuevo algoritmo mas poderoso para la verificacion de consistencia en la malla.

2.4.2. Análisis Detallado

A continuación, se hará un análisis detallado de las diferentes clases del sistema.

Primero, se revisará el diagrama de clases detallado mostrado en la Figura 2.3 en la página siguiente. Este diagrama muestra la estructura de clases Malla, la cual almacena información topológica y geométrica del objeto modelado.

Las siguientes clases son importantes de detallar en esta parte del diseño:

- Malla: Representa la superficie de los objetos que se desea modelar. Esta compuesta de Nodos, Arcos y Caras. Vemos que en este momento esta clase tiene una gran cantidad de métodos públicos los cuales no necesariamente pertenecen a esta clase. Por ejemplo, hay métodos como getLargoArco que deberían ser parte de la clase Arco en vez de la clase Malla. Esto provoca que sea una clase muy compleja de mantener.
- Polinomio: Representa un polinomio y sus operaciones básicas como suma, resta y multiplicación, además de la obtención de sus raíces. Se utiliza para la resolución de polinomios de tercer y segundo grado que aparecen en el algoritmo de verificaciones locales en el desplazamiento de nodos de una malla.

Figura 2.3: Diagrama de Clases del Sistema - Estructura de la Malla

- Arco: Representa un arco de la malla. Posee referencias a los dos nodos que lo forman y dos índices a las caras a las que pertenece.
- Nodo: Contiene un índice a un punto, la dirección en que será desplazado y la distancia en que se moverá en esa dirección.
- Cara: Representa una cara de la malla, por ejemplo un triangulo. Esta formada por un número determinado de puntos y arcos (3 en el caso de triángulos). Posee referencias a los nodos y arcos que la formas.
- Punto: Representa un punto especifico en el espacio de coordenadas en 3 dimensiones. Posee tres números que representan las coordenadas X,Y,Z.
- Vect: Representa un vector en el espacio en tres dimensiones. Está definido el punto P y el punto en el origen o de coordenadas (0,0,0).

A continuación, se revisará con profundidad el detalle de los algoritmos de generación de mallas (Generar una malla a partir de un input dado) y de los algoritmos de desplazamiento (Generar una malla en la cual los puntos hayan tenido un desplazamiento respecto a un estado anterior). El diseño actual de estos algoritmos se pueden ver en la Figura 2.4 en la página siguiente.

Las siguientes clases son importantes de detallar en esta parte del diseño:

- AlgGeneracion: Interfaz que agrupa a cada uno de los algoritmos de generación de una malla inicial de un objeto a modelar.
- GeneraFromMatlab: Algoritmo de generación de una malla inicial a partir de un archivo de texto obtenido de un modelamiento en Matlab.
- AlgDesplazamiento: Interfaz que agrupa a cada uno de los algoritmos de desplazamiento de cada uno de los nodos que componen la malla.
- VerificacionLocal: Algoritmo de desplazamiento de los nodos de una malla basado en la verificación de la consistencia local de la malla.

Viendo esta parte del diseño, vemos que una mejora importante es la separación del formato de entrada del algoritmo con el algoritmo que genera la malla inicial. La idea sería pasar primero a un formato normalizado y luego de eso calcular la malla.

Finalmente, se describe la estructura de los algoritmos de refinamiento y mejora (En la implementación actual, los algoritmos de refinamiento y mejora están agrupados bajo una misma interfaz). El diseño lo podemos ver en la Figura 2.5 en la página 17.

Las siguientes clases son importantes de detallar en esta parte del diseño:

- AlgRefinamiento: Interfaz para los algoritmos de refinación y mejoramiento de la malla.

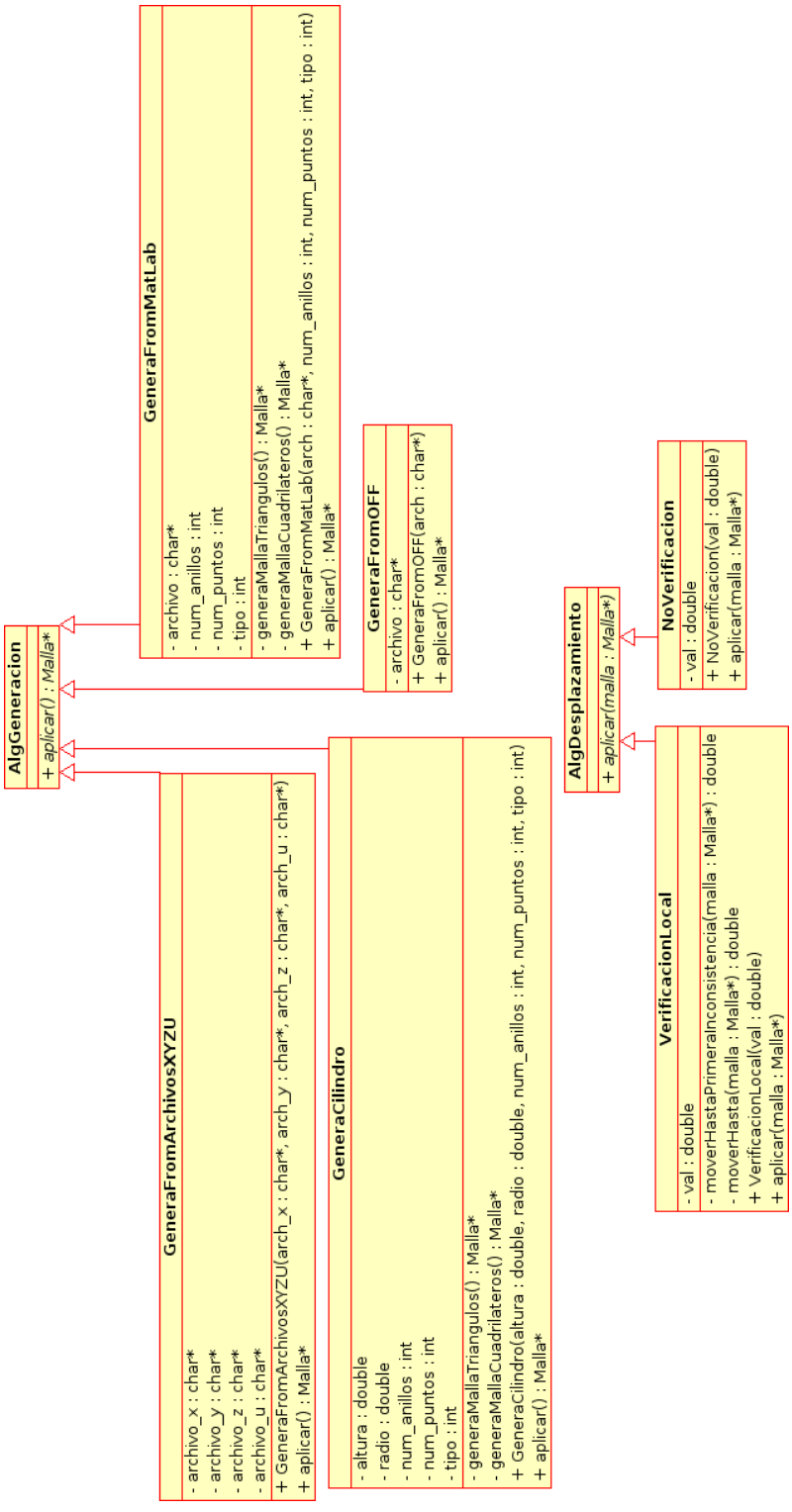


Figura 2.4: Diagrama de Clases del Sistema - Algoritmos de Generación y Desplazamiento

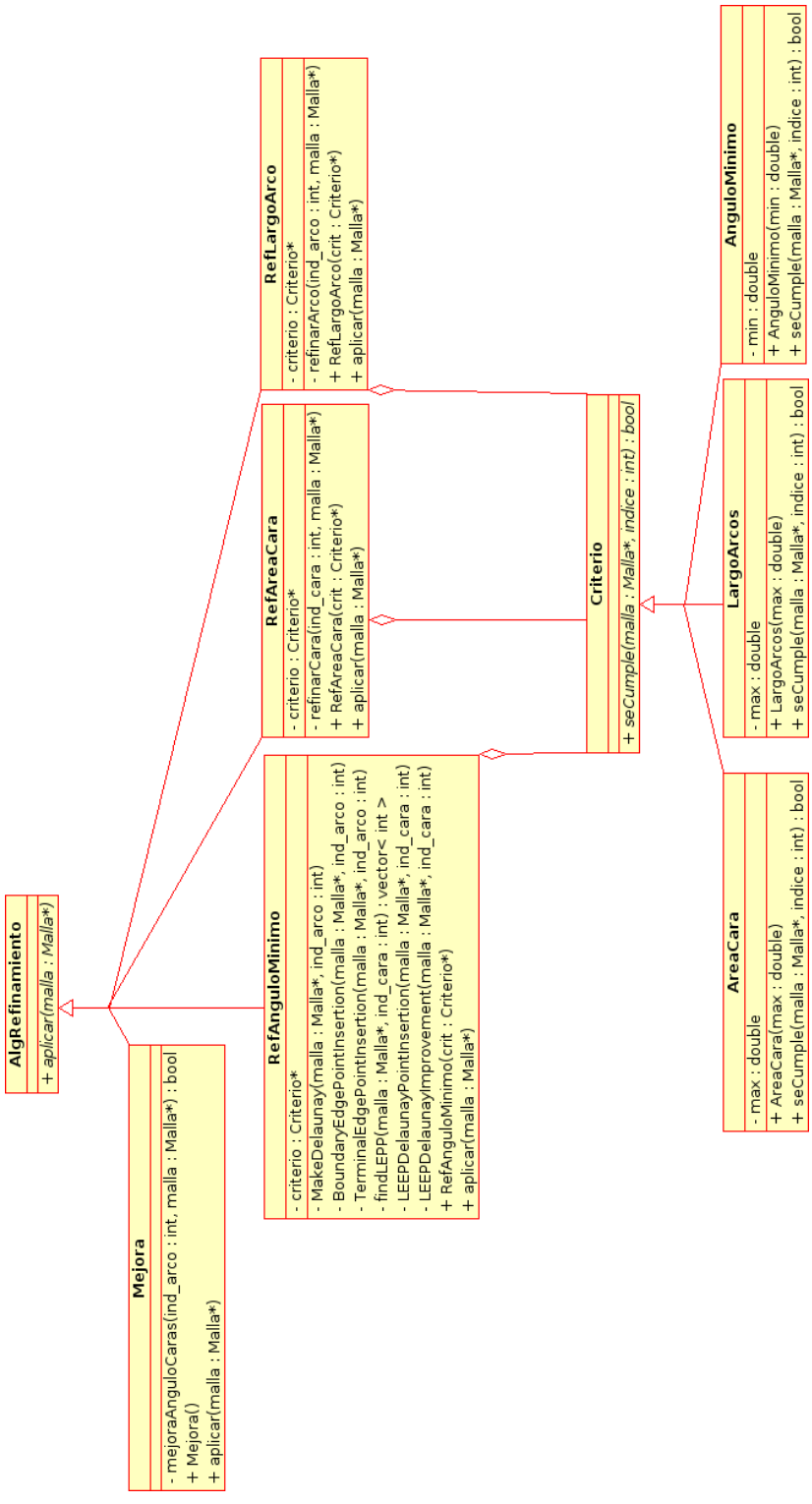


Figura 2.5: Diagrama de Clases del Sistema - Algoritmo de Refinamiento

- Mejora: Algoritmo que mejora la malla maximizando el ángulo mínimo de las caras mediante flips de arcos.
- RefAnguloMinimo: Refinamiento de la malla basado en el ángulo mínimo de las caras.
- Criterio: Interfaz que agrupa a cada uno de los criterios utilizados para refinar la malla.
- AreaCara: Criterio que verifica el área de una cara.

Tal como indiqué en mi análisis general de la aplicación, esta parte del diseño requiere mejoras de manera de separar los criterios de los algoritmos. Actualmente los criterios son creados y definidos dentro del mismo algoritmo, en vez de ser pasados como parámetros. Esto último permitirá usar un mismo algoritmo con distintos criterios de manera natural.

2.4.3. Estructuras de Datos

En esta sección se describen las estructuras de datos que se utilizan para representar las mallas de los objetos a modelar. Además se detallan operaciones o transformaciones sobre la topología de la malla, las cuales serán utilizadas para desplazar los elementos de la malla y para refinarla. La estructura de datos de la malla se puede ver en la Figura 2.6.

2.4.3.1. Representación de la Malla

De manera general se sabe que una malla en tres dimensiones esta formada por puntos, arcos, caras (En general triángulos o cuadriláteros) y volúmenes (En general tetraedros). En este caso se debe representar solo la superficie de los objetos en tres dimensiones con lo cual no se necesita representar volúmenes, sino solo los puntos, arcos y caras.

Se debe considerar que las mallas representadas están sujetas a deformaciones, las que se producen en cada uno de los puntos de la malla, con lo cual se necesita almacenar más información que solo la posición de los puntos en el espacio. Así, para representar a un punto de la malla se utiliza la estructura de datos Nodo, la cual tiene la siguiente información:

- Posición en el espacio: Representada por la estructura de datos Punto, la cual contiene las coordenadas X, Y, Z de su posición.
- Vector unitario que indica dirección de movimiento del nodo: Un vector se representa mediante la estructura de datos Vect, que a su vez contiene un Punto P_0, de manera que el vector parte en el origen y termina en P_0. Se llamo Vect para no confundir con la estructura vector de C++ que representa una lista de elementos[2].
- Concentración: Número de punto flotante que indica el valor o cantidad del desplazamiento del nodo para un tiempo t determinado. En este caso para t=1, de manera que en t=1 cada nodo se desplazará una cantidad igual a su concentración.

[2]Para resolver el duplicado de nombres, también se podrían haber usado namespaces de C++.

Figura 2.6: Estructura de Datos de la Malla

- Dos listas de índices: Una de las caras que contienen el nodo y otra de los arcos que contienen este nodo. Estos índices se almacenan para hacer más eficientes los algoritmos que se utilizaran puesto que para formar una malla solo basta con que las caras posean una lista de los puntos que las forman.

Un arco de la malla se representa en la estructura de datos Arco, la cual contiene la siguiente información:

- Dos índices a los nodos que forman el arco.
- Dos índices a las caras que comparten este arco. Se almacenan solo dos índices puesto que se trabajara con mallas simples, es decir, cada arco separa a lo mas dos caras. Se es un arco borde uno de los índices es -1.

Una cara de la malla se representa en la estructura de datos Cara, la cual contiene la siguiente información:

- Una lista de índices a los nodos que forman la cara.
- Una lista de índices a los arcos que pertenecen a la cara. Obviamente, el número de índices a arcos es igual al número de nodos y depende del tipo de cara que se represente.

Así, una malla está formada por un conjunto de elementos Nodo, un conjunto de elementos Arco y un conjunto de elementos Cara. Cada uno de estos conjuntos se representará mediante estructuras de datos contenedoras de estos elementos. Estas estructuras son: Nodos, Arcos y Caras respectivamente. Cada una de ellas contiene la siguiente información:

- Una lista de elementos que almacena.
- Una lista de índices a elementos borrados del contenedor.
- El número de elementos totales del contenedor o tamaño de la lista.
- El número de elementos válidos o que no han sido borrados. Es igual al número de elementos totales menos el número de elementos borrados.

Las listas se representan mediante la estructura de datos vector provista por C++, puesto que es una estructura fácil de manejar y provee acceso constante a los elementos mediante índices. Esta última característica es muy importante ya que los elementos guardan referencias mediante índices.

Así, cada contenedor tiene una lista de los elementos que almacena, y dentro de esta lista pueden haber elementos válidos o elementos borrados o vacíos. Los contenedores se manejan de esta manera, por tres motivos principales:

1. Los algoritmos utilizados hacen que la topología de la malla esté en constante cambio, es decir, se eliminan varios elementos y se agregan otros.

2. Los índices dentro de los contenedores no pueden cambiar si se elimina o se agrega algún elemento. Es decir, si se agrega un elemento no puede ser sobre un índice ocupado y si se elimina uno, los elementos que quedan no se pueden reacomodar con el índice que queda. Esto porque los elementos de la malla poseen índices que apuntan a otros elementos, si cambian, entonces cambiara de manera inconsistente la topología de la malla.

3. Se tiene una lista de índices borrados para poder aprovechar esos índices cuando se agreguen nuevos elementos. Así, cuando se agrega un nuevo elemento se obtiene un índice de los borrados y allí se inserta el elemento. Si no hay índices borrados, entonces se agrega un nuevo elemento a la lista. Se pensó de esta manera por la razón de que la lista del contenedor no creciera demasiado.

2.4.3.2. Otras Estructuras de Datos

Además de las estructuras de datos para representar una malla, se utiliza la estructura de datos Polinomio. Como su nombre lo indica, representa un polinomio y surge por la necesidad de resolver una ecuación cúbica dentro del algoritmo de deformación con verificación local. Esta estructura almacena una lista con los coeficientes del polinomio y su grado. Posee las funciones básicas de suma, resta y multiplicación de polinomios, además de la obtención de sus raíces mediante la biblioteca GSL.

2.4.4. Transformaciones Utilizadas en una Triangulación

A continuación se describirán las transformaciones utilizadas tanto en los algoritmos de deformación de la malla como en los algoritmos de refinamiento, desrefinamiento y mejoramiento de las mallas. Las transformaciones básicas que se utilizan son tres. Estas son: Edge-Flip, Edge-Split y Edge-Collapse. A continuación se describe en detalle cada una de estas transformaciones.

1. Edge-Flip (Intercambiar Arco): Es la transformación más simple de todas. Consiste en intercambiar el arco uniendo los nodos opuestos de las caras que separa. El nodo opuesto a un arco en una cara dada, es el nodo que no esta asociado a ese arco dentro de la cara. Si el arco no es borde, esta asociado a dos caras, y en cada una de ellas el arco posee un nodo opuesto. El Edge-Flip se realiza borrando el arco original y creando un nuevo arco con los nodos opuestos del arco original. La Figura 2.7 muestra esta transformación tanto en dos dimensiones como en mallas de superficie. En el caso de mallas de superficie, esta transformación se utiliza principalmente para corregir casos degenerados donde caras vecinas se intersectan entre si, como se verá en la Sección 2.4.5.2.

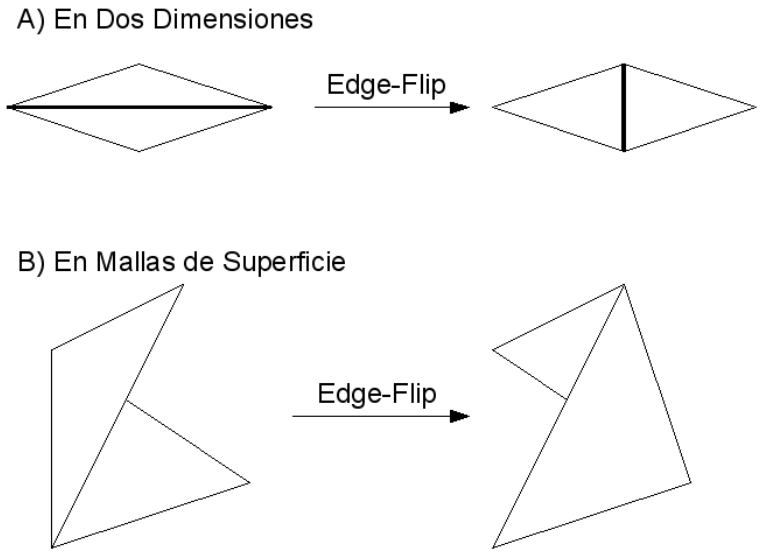


Figura 2.7: Transformación Edge-Flip

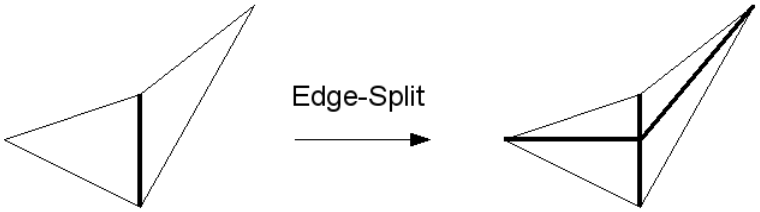


Figura 2.8: Transformación Edge-Split

2. Edge-Split (Dividir Arco): Esta transformación consiste en dividir un arco, insertando un nodo sobre el arco, es decir, entre los dos nodos que forman el arco. Para mantener consistente la malla, se unen los nodos opuestos al arco original al nuevo nodo. La Figura 2.8 muestra un ejemplo de la transformación Edge-Split.

3. Edge-Collape (Reducir Arco): Esta transformación elimina un arco colapsándolo en un nuevo nodo ubicado sobre el arco original, eliminando así también los dos nodos que formaban el arco. En la Figura 2.9 se muestra un ejemplo de Edge-Collapse.

Cabe mencionar que la implementación actual de estas transformaciones carece de un importante grado de robustez en la implementación. Por lo tanto, se deberá analizar una reimplementación de estos métodos, de manera tal que permitan lograr una implementación limpia de las operaciones de refinamiento y deformación de las mallas.

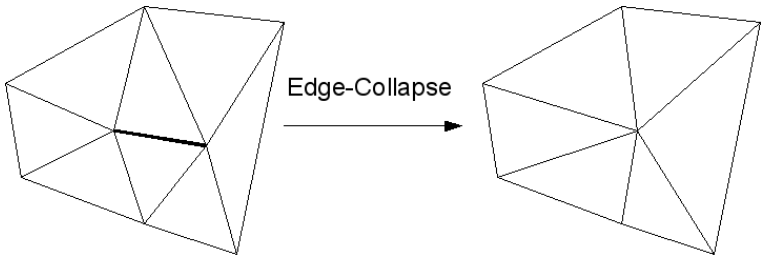


Figura 2.9: Transformación Edge-Collapse

2.4.5. Algoritmos de Desplazamiento

En [1] se proponen e implementan dos algoritmos que manejan la deformación de una malla geométrica. Estos algoritmos son el Algoritmo sin Verificación, y el Algoritmo con Verificación Local.

2.4.5.1. Desplazamiento sin Verificación

Es el algoritmo más simple. En él se generan solo cambios geométricos y no topológicos, es decir, la estructura de la malla no cambia, solo cambia la posición inicial de cada nodo según su información de movimiento.

El algoritmo se aplica fácilmente sobre la malla, moviendo cada nodo de esta según su vector de dirección y su concentración. Los vectores de dirección de cada nodo son vectores unitarios, es decir, indican solo la dirección de movimiento del nodo. La distancia que se debe mover esta dada por la concentración.

2.4.5.2. Desplazamiento con Verificación Local

Este algoritmo realiza verificaciones de consistencia en la malla para cada par de caras unidas por un arco. Es decir, se verifica consistencia a nivel de caras vecinas. De esta manera, se debe cumplir que mientras los nodos se muevan en forma simultanea para ambas caras, ninguno de los arcos de una cara cruce un arco de la otra y que ningún vértice este sobre una cara a la que no pertenezca. De manera sencilla se puede deducir que estas restricciones se dan solamente cuando ambas caras están sobre un mismo plano. En la Figura 2.10 se puede apreciar una inconsistencia provocada por dos caras coplanares donde sus arcos están intersectados.

De esta manera, una condición necesaria pero no suficiente para que exista una inconsistencia local, es que en un momento determinado ambas caras estén sobre un mismo plano. Mientras no se de esta situación se dirá que no habrá una inconsistencia local y los nodos podrán seguir siendo desplazados.

La implementación del algoritmo consiste en términos generales en:

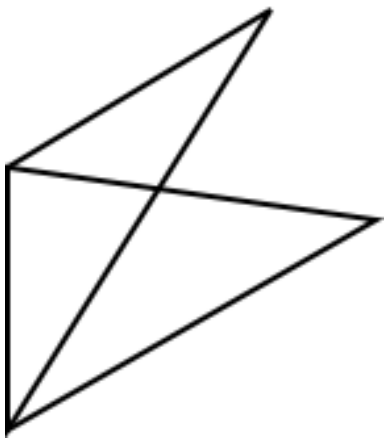


Figura 2.10: Inconsistencia entre Caras Vecinas

1. Detectar si caras vecinas están situadas dentro de un mismo plano.
2. Si caras vecinas son coplanares, detectar si hay intersección de arcos entre ellas.
3. Si hay intersección, aplicar transformaciones vistas en la Sección 2.4.4, dependiendo del tipo de intersección que se encontró (Intersección arco con vértice, arco con arco o vértice con vértice). Para cada tipo de intersección se deben efectuar una serie diferente de transformaciones para volver consistente la malla. Este es uno de los grandes problemas de esta algoritmo ya que se deben manejar una gran cantidad de casos especiales en la implementación. La implementación se vuelve tremendamente compleja por lo que se provocan serios problemas de robustez al algoritmo.

En el desarrollo de la nueva aplicación se verá una nueva estrategia de solución a este problema que involucre una implementación más limpia para detectar inconsistencias a nivel local.

2.4.6. Interfaz Gráfica

En la Figura 2.11 en la página siguiente se puede observar la interfaz gráfica que tiene actualmente la aplicación. Al analizar detenidamente esta interfaz podemos notar su falta de usabilidad. Alguna de estas fallas son:

- Todas las opciones son mostradas en un mismo diálogo.
- Interfaz absolutamente recargada.
- No hay posibilidad de incluir más funcionalidades ya que no caben más opciones dentro del diálogo.

En la implementación actual, los modelos generados para ser visualizados, requieren de el software de visualización Geomview. En la figura 2.12 en la página siguiente se puede observar una figura generada por la aplicación Modelador de Cambios, que es visualizada a través de Geomview.

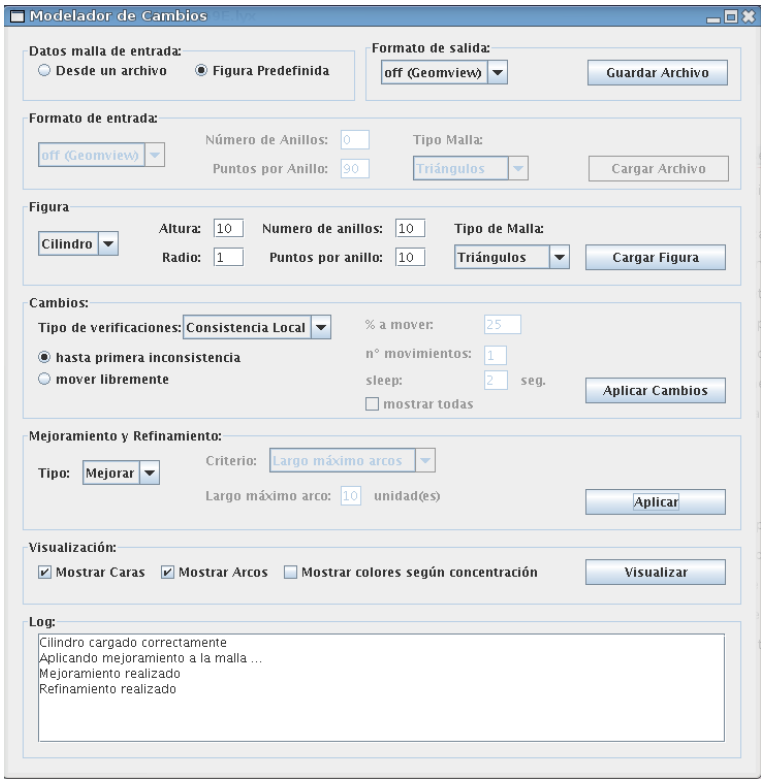


Figura 2.11: Screenshot de la Aplicación Modelador de Cambios

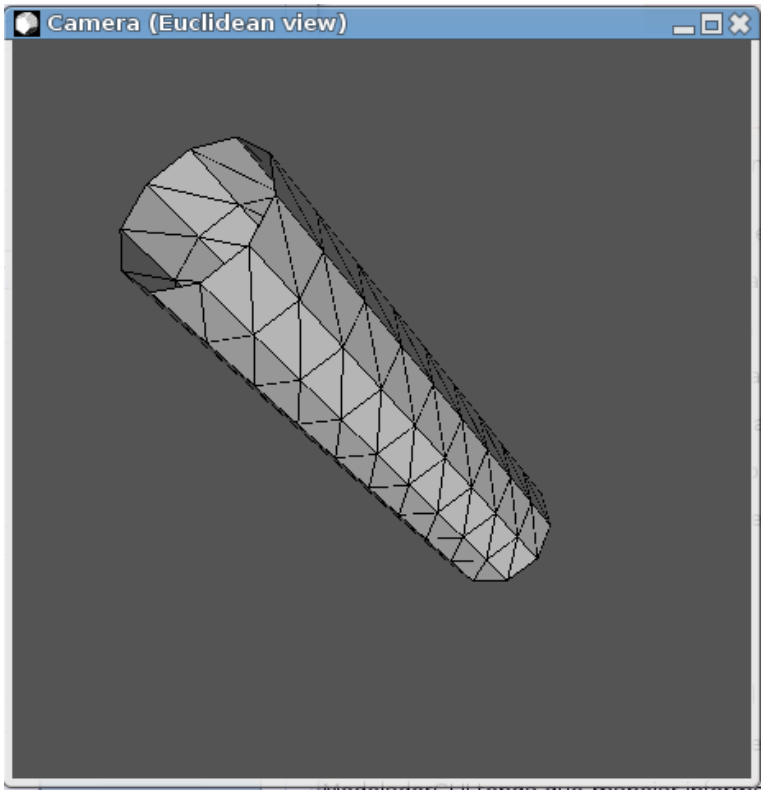


Figura 2.12: Screenshot del Visualizador Externo Geomview

Capítulo 3

Análisis y Rediseño

3.1. Rediseño Global

En la Figura 3.1 se puede ver el diagrama de clases completo de la aplicación ya rediseñada. En las próximas Secciones se describirán en detalle cada uno de los aspectos mejorados

3.2. Rediseño de las Clases Asociadas a los Procesos Refinamiento y Mejoramiento

Tal como se revisó en la Sección 2.4.1 , los procesos de Refinamiento y Mejoramiento requerían de un rediseño importante. En este rediseño se consideró tanto los requerimientos originales de la aplicación como los nuevos requerimientos. A continuación presento algunos de los puntos que se tomaron en cuenta para este rediseño:

- Extensibilidad de los algoritmos de refinamiento de tal forma de poder incluir nuevos algoritmos en forma sencilla.
- Extensibilidad de los criterios para poder incluir nuevos criterios.
- Los algoritmos no deben estar asociados estáticamente a los criterios. Uno debiera poder decidir qué criterio de selección aplicará cada algoritmo en tiempo de ejecución de la aplicación.
- Separación de las clases relacionadas al refinamiento con las relacionadas al mejoramiento. Conceptualmente son cosas distintas que no deberían seguir agrupadas bajo una misma clase interfaz.

Tomando en cuenta todos estos puntos, se diseñó el diagrama de clases mostrado en la Figura 3.2 que consistió en.

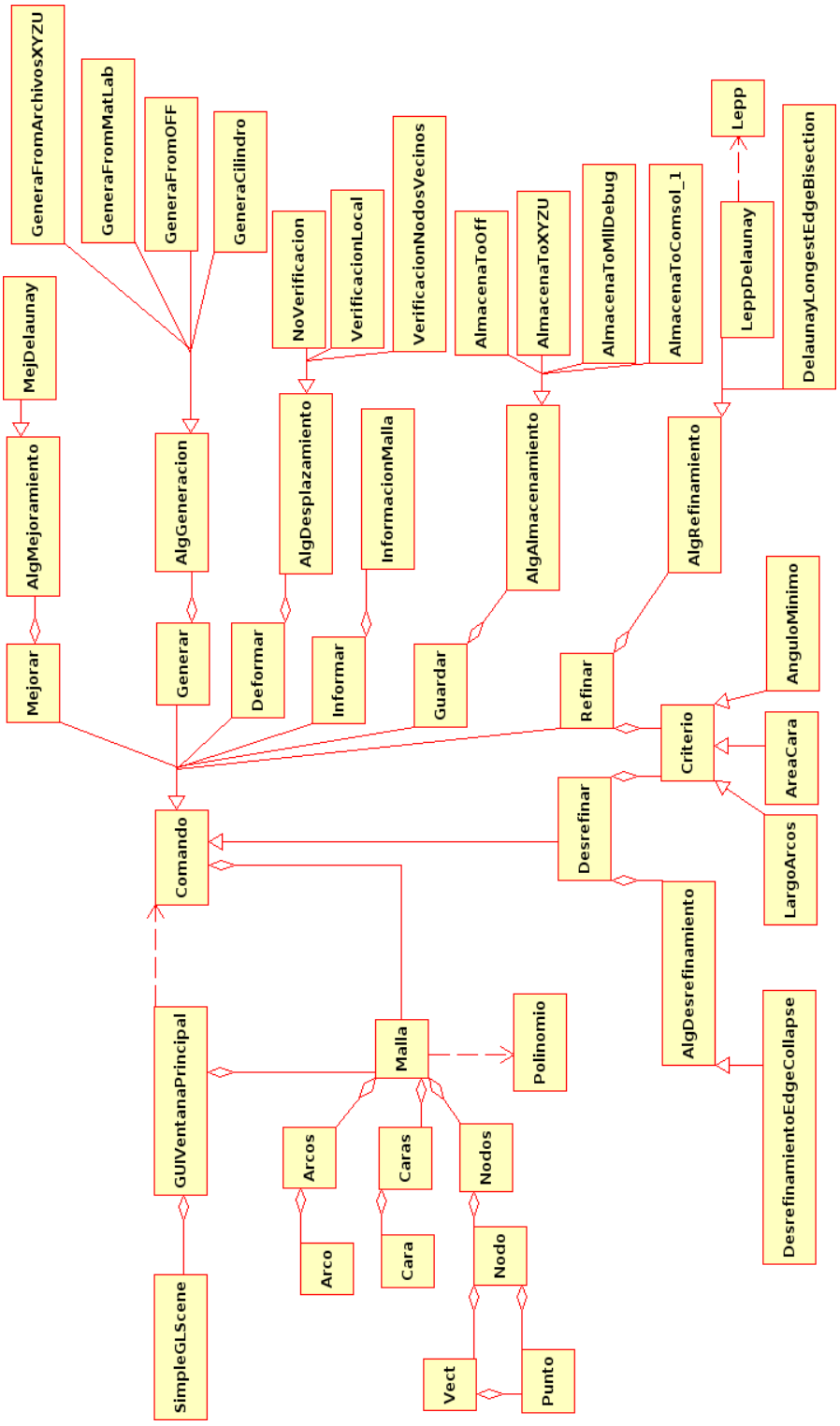


Figura 3.1: Diagrama de Clases Completo Despues del Rediseño de la Aplicación

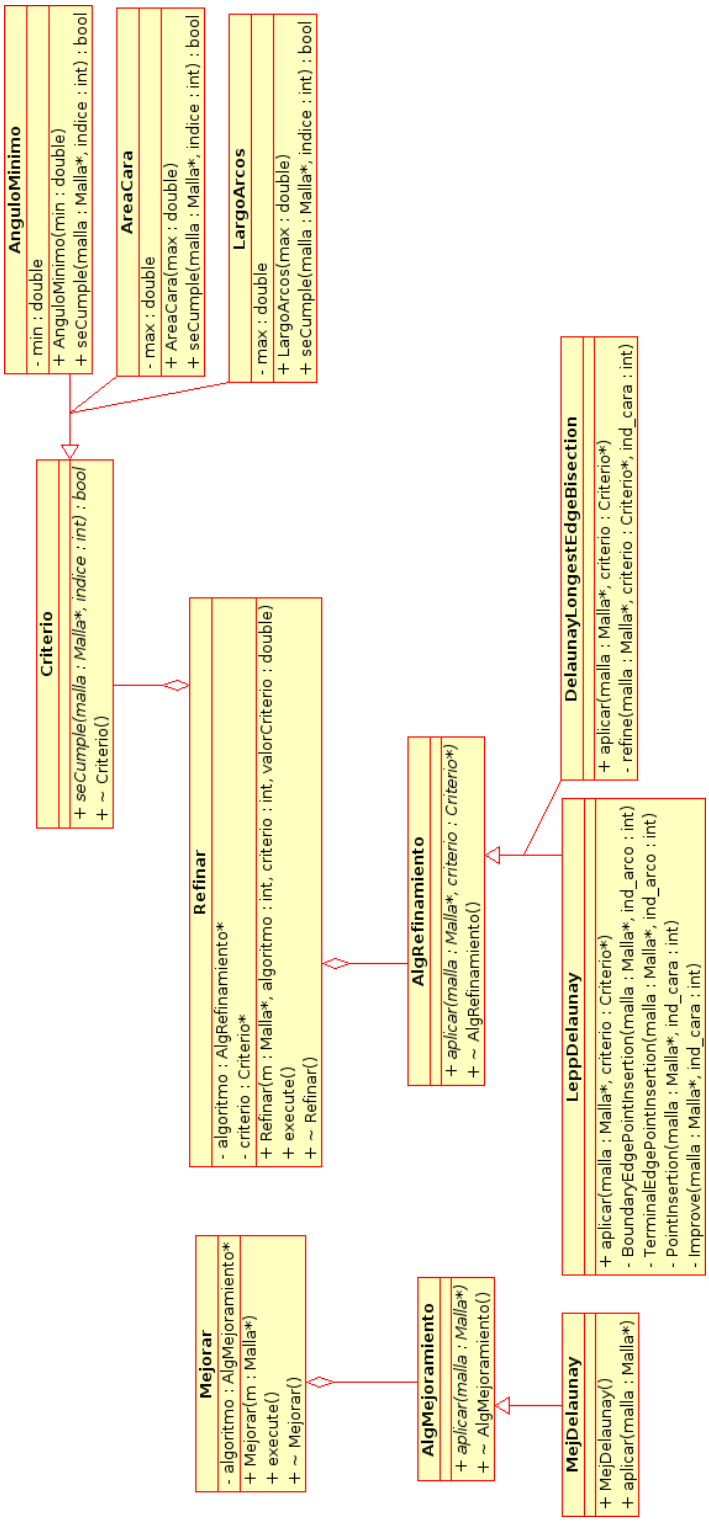


Figura 3.2: Diagrama de Clases del Rediseño de Refinamiento y Mejoramiento

- Las clases de refinamiento se separaron en dos clases distintas. La clase Mejorar agrupa a los algoritmos encargados de mejorar la calidad de la malla sin agregar puntos y la clase Refinar agrupó a los algoritmos que aumentan la densidad de puntos
- Las clases Mejorar y Refinar son clases encargadas de invocar el comando seleccionado por el usuario. Se siguió este esquema considerando el patrón de diseño Command. En este rediseño se separaron los comandos de Mejorar y Refinar ya que son acciones completamente diferentes, las cuales son llamadas con argumentos diferentes.
- Utilizando el patrón de diseño Strategy, se agruparon las distintas estrategias de refinamiento LeppDelaunay[15] y DelaunayLongestEdgeBisection bajo la clase interfaz virtual AlgRefinamiento. De igual forma, se agrupó la estrategia Delaunay bajo la clase interfaz virtual AlgMejoramiento.
- Los criterios dejaron de ser implementados en forma estática dentro de los algoritmos de refinamiento. Ahora los criterios son seleccionados en tiempo de ejecución y son pasados como argumento a AlgRefinamiento mediante el objeto interfaz Criterio. De esta clase interfaz criterio heredan los criterios específicos AreaCara, LargoArcos y AnguloMinimo.

Como resultado, este rediseño permite la extensibilidad tanto de los algoritmos de mejoramiento, como de los algoritmos de refinamiento y de los criterios de selección.

3.3. Rediseño de las Clases Asociadas a los Procesos de Almacenamiento de la Malla

Tal como se vio en la Sección 2.4.1, todos los algoritmos para almacenamiento de la malla estaban implementados en una misma clase. Ese diseño tiene el gran problema de no permitir agregar en forma limpia nuevos algoritmos para el almacenamiento de mallas.

Utilizando el patrón de diseño Command y Strategy se rediseñó esta parte de la aplicación para así permitir agregar nuevos tipos de formatos de almacenamiento en forma rápida y transparente para el resto de la aplicación. En la Figura 3.3 se puede ver el resultado del rediseño.

3.4. Rediseño de Interfaz Gráfica

Esta sección describe el rediseño de la interfaz gráfica de la aplicación, donde se incorporan nuevas características tanto para mejorar la usabilidad del software, como la facilidad para el desarrollador de incorporar nuevas características a la aplicación. Para reflejar una de las nuevas características como es el uso de OpenGL, se ha decidido renombrar la vieja aplicación Modelador de Cambios como Modelador de Cambios GL.

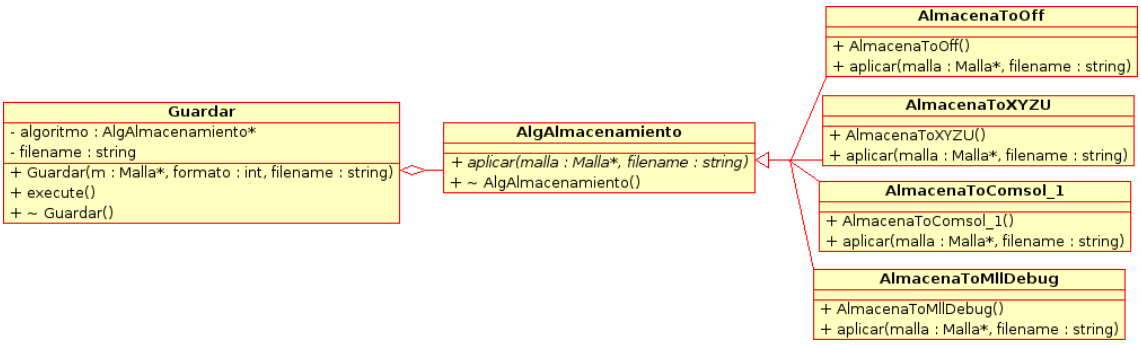


Figura 3.3: Diagrama de Clases del Rediseño de Almacenamiento

3.4.1. Rediseño Gráfico y de Usabilidad

En la Sección 2.4.6 se vieron las deficiencias de la interfaz gráfica antigua. Es por esta razón que se decidió hacer un completo rediseño de la parte gráfica tomando en cuenta la usabilidad y la extensibilidad de ella.

Al tomar en cuenta la usabilidad, debemos diseñar una interfaz que sea fácil de usar y de entender para el usuario final. Una interfaz que incluya menús, iconos, etc. en forma organizada y limpia.

Una de las importantes metas a lograr en esta memoria, era la posibilidad de tener un visualizador de mallas integrado en la interfaz del programa. Esta característica le otorga un alto grado de usabilidad al programa ya que el usuario puede ver en forma instantánea los resultados de cada algoritmo que aplique. Ya no será necesario de que esta pasándose de un programa a otro para aplicar algoritmos y luego para visualizar la malla. En la Figura 3.4 se puede ver un screenshot de la aplicación con el visualizador en tiempo real integrado en la interfaz.

Para el diseño de esta nueva interfaz gráfica, se tomó en cuenta las GNOME Human Interface Guidelines (GNOME HIG) [20]. Estos alineamientos ayudan a escribir interfaces que sean fáciles de usar y que además sean consistentes con el entorno gráfico GNOME. Algunos de los beneficios que nos otorgan son:

- Los usuarios aprenden a usar el programa más rápidamente, ya que los elementos de la interfaz se ven y se comportan de la forma que ellos están acostumbrados.
- Tanto usuarios principiantes como avanzados podrán realizar tareas en forma rápida y fácil, ya que la interfaz no será confusa y no hará las cosas más difíciles.
- La aplicación tendrá una vista atractiva que encajará con el resto del escritorio.
- La aplicación seguirá viéndose bien incluso cuando los usuarios cambien de tema gráfico, tipos de letras y colores.

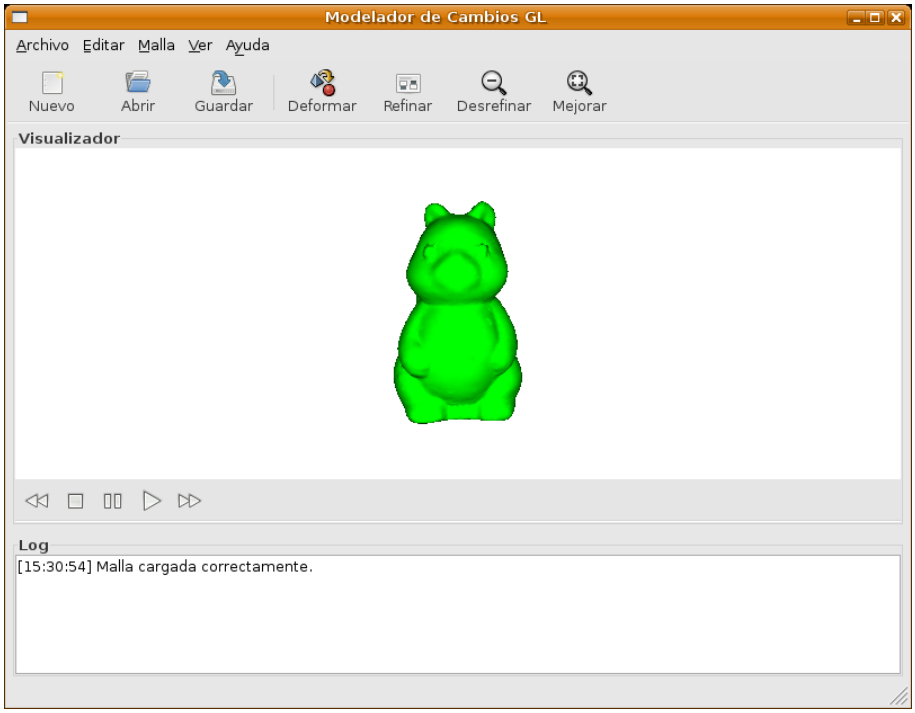


Figura 3.4: Screenshot Modelador de Cambios GL

Como se pudo observar en la Figura 3.4, la ventana principal de la aplicación esta dividida en cuatro secciones las cuales son recomendadas dentro del GNOME HIG para las aplicaciones SDI[1]. La ventana principal esta compuesta por una barra de menús, una barra de herramientas, el área de trabajo y la barra de estado:

1. Barra de Menús: La barra de menús presenta toda la gama de comandos de la aplicación al usuario. En muchas ocasiones también presenta un subconjunto de las preferencias. Es importante tener en cuenta que la ubicación de los comandos en los menús deben aparecer en los mismo lugares que esos comandos suelen aparecer en otras aplicaciones, de manera que sean más fácil de aprender para el usuario. Tomando en cuenta estos aspectos se diseño la barra de menús de la aplicación que se puede ver en la Figura 3.5.

2. Barra de Herramientas: La barra de herramientas es un conjunto de controles que permiten al usuario tener acceso rápido y conveniente a los comandos de la aplicación más usados. Estos controles generalmente son botones gráficos. Un diseño cuidadoso y consistente debe considerar acelerar las tareas del usuario al darle acceso directo a funciones que de otra manera estarían ocultas dentro de un menú. Es importante usar controles solo para las funciones más importantes, ya que el tener muchos controles en la barra de herramientas los hace difícil de encontrar y reducen el espacio en pantalla para el resto de la aplicación. La barra de herramientas diseñada para la aplicación se puede ver en la figura 3.6.

[1]SDI: Single Document Interface. Se refiere a interfaces diseñadas para manejar un solo documento a la vez.

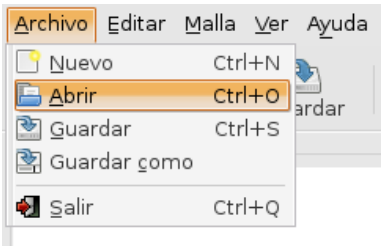


Figura 3.5: Barra de Menús de Modelador de Cambios GL

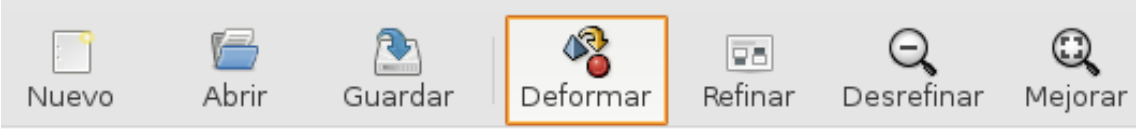


Figura 3.6: Barra de Herramientas de Modelador de Cambios GL

3. Área de Trabajo: Esta es el área principal donde se desarrolla toda la interacción con el usuario. En esta área el usuario puede ver los resultados de los comandos que ha efectuado. El área de trabajo de la aplicación se puede ver en la Figura 3.7. En nuestra aplicación el área de trabajo esta dividida en tres partes. La parte 1 es el área de visualización. En esta parte se pueden visualizar en tiempo real los cambios efectuados en la malla. Incluso tiene la posibilidad de rotación y acercamiento de la malla utilizando los botones del mouse. La parte 2 es el área de animación. Esta área contiene controles para manejar la animación de las deformaciones de la malla con respecto al tiempo. Para una mayor usabilidad, los controles se han diseñado de manera que se asemejan a los de cualquier reproductor multimedia con botones para Play, Stop, Slow, Fast, Pause. La parte 3 es el Log de Cambios. Esta área le indica al usuario los últimos comandos que ha realizado, el resultado que generaron y además la hora en que fueron efectuados.

4. Barra de estado: Es simplemente la barra inferior de la aplicación. Esta barra sirve para darle información de ayuda al usuario cuando el cursor se sitúa sobre algún comando de la aplicación.

Además de la ventana principal, la aplicación cuenta con varias ventanas auxiliares, o en estricto rigor, cuadros de diálogo. El propósito de estos cuadros es obtener información adicional del usuario al momento de efectuar ciertas operaciones. En el momento en que se necesita esta información, se levanta un cuadro de diálogo al usuario preguntándole sobre esta información. En la Figura 3.8 se puede ver un ejemplo de uno de estos cuadros de diálogo en donde se le pregunta al usuario que algoritmo desea utilizar para el refinamiento de la malla.

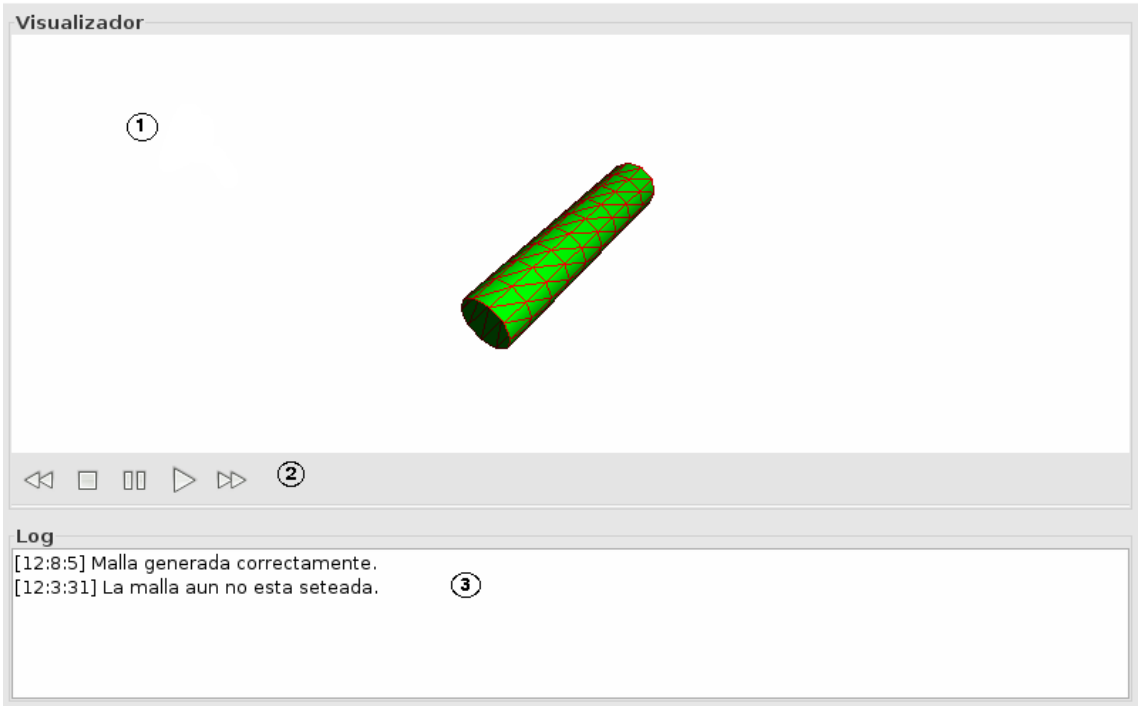


Figura 3.7: Área de Trabajo Modelador de Cambios GL

Figura 3.8: Cuadro de Diálogo Modelador de Cambios GL

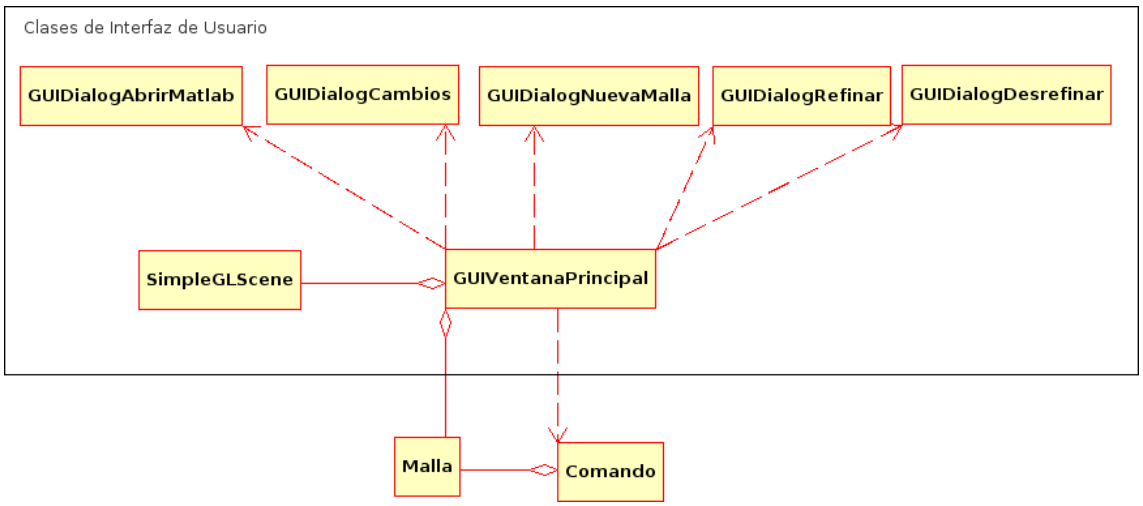


Figura 3.9: Diagrama de Clases del Rediseño de la Visualización

3.4.2. Rediseño de Clases Asociadas al Proceso de Visualización

No solo el rediseño de la parte gráfica fue importante dentro de la nueva interfaz de usuario. También fue importante un rediseño estructural de las clases encargadas de manejar la interfaz. Como se vio en la Sección 2.4.1, el diseño de clases de la interfaz antigua tenía el grave problema de estar fuertemente acoplado al resto de las clases de la aplicación. Esto daba como consecuencia la necesidad de manejar una gran cantidad de objetos diferentes si se quería hacer alguna modificación o cambio de la interfaz de usuario.

Como consecuencia, se rediseño completamente el esquema de clases, desacoplando en gran parte las clases relacionadas con la interfaz de usuario con las relacionadas con el control y manejo de objetos de la aplicación. El resultado obtenido es el diagrama de clases mostrado en la Figura 3.9, cuyas clases son las siguientes:

- GUIVentanaPrincipal: Es la ventana principal de la aplicación.
- GUIDialogAbrirMatlab: Cuadro de diálogo que se despliega al abrir un archivo Matlab. Pregunta al usuario información sobre el despliegue de la malla para el archivo Matlab abierto.
- GUIDialogCambios: Es el cuadro de diálogo donde se seleccionan las deformaciones que se le desean aplicar a la malla y sus parámetros.
- GUIDialogNuevaMalla: Es el cuadro de diálogo donde se selecciona alguna figura geométrica para crear una nueva malla.
- GUIDialogRefinar: Cuadro de diálogo donde se seleccionan algoritmos de refinamiento y sus parámetros.
- GUIDialogDesrefinar: Cuadro de diálogo donde se seleccionan algoritmos de desrefinamiento y sus parámetros.

- SimpleGLScene: Clase encargada de el área de visualización OpenGL de la ventana principal. Esta clase renderiza la malla en memoria y la despliega en el visualizador en tiempo real.

Este diseño cumple el objetivo deseado, ya que las clases de la interfaz gráfica son independientes del resto de la aplicación. La conexión que se mantiene con la aplicación se hace a través de la clase Comando la cual ejecuta acciones a realizar sobre la malla, de acuerdo a los parámetros indicados por el usuario en la interfaz gráfica.

Capítulo 4

Implementación

En este capítulo se describen en detalle las herramientas y procesos utilizados en la construcción de la aplicación. En particular, se describe el ambiente de trabajo en que se desarrolló la aplicación; el uso de bibliotecas complementarias; la corrección o rediseño de antiguos algoritmos; e implementación de nuevos algoritmos y características a la aplicación.

4.1. Ambiente de Trabajo

La aplicación fue desarrollada sobre el sistema operativo Linux, distribución Ubuntu 6.06 Dapper Drake con kernel 2.6.15-27. El lenguaje de programación usado es C++, con la versión 4.0.3 del compilador GNU (gcc).

El Ambiente de Desarrollo Integrado (IDE) utilizado para la generación e implementación de las clases en C++ fue Anjuta versión 1.2.4a. Para el diseño y generación de la interfaz de usuario se utilizo Glade 2.12.1.

Para la corrección y debugging de la aplicación, específicamente los problemas de memoria como Fallos de Segmentación y Memory Leaks, se uso la aplicación Valgrind versión 3.1.0 junto con su entorno gráfico Alleyoop versión 0.9.0.

4.2. Uso de Bibliotecas en la Aplicación

La aplicación usa varios conjuntos de bibliotecas. Estas son GTK+, GTKmm, GTKGlExtmm y GSL. GTK+ y GTKmm se utilizaron para la creación de la interfaz de usuario de la aplicación. GTKGlExtmm se utilizó para la implementación del visualizador OpenGL en tiempo real integrado en la aplicación. Finalmente, GSL surgió por la necesidad de resolver ecuaciones polinomiales de primer, segundo y tercer grado. Después de una búsqueda extensiva, se decidió utilizar estas bibliotecas que fueron las que mas se acomodaban a los requerimientos inmediatos de la aplicación. A continuación se entrega una descripción y una referencia de estas bibliotecas.

- GTK+: Provee un conjunto de bibliotecas para la construcción de interfaces gráficas de usuario (GUI). Esta desarrollado en C. Es una de las bibliotecas mas utilizadas en el desarrollo de aplicaciones gráficas para Linux. En esta implementación se uso la versión 2.8. En [16] se puede obtener mayor información.
- GTKmm: Es la Interfaz oficial escrita en C++ para la biblioteca GTK+. GTKmm me permite utilizar el lenguaje C++ para el desarrollo de la interfaz gráfica. Junto con ello obtengo todos los beneficios de la programación orientada a objetos como lo son la herencia, extensibilidad, programación por componentes, etc. En esta implementación se uso la versión 2.8. En [17] se puede obtener mayor información.
- GTKGlExtmm: Es una extensión OpenGL para GTK+. Esta extensión provee objetos adicionales para GTK+ los cuales tienen la característica de soportar renderizado OpenGL sobre ellos. Tiene una Interfaz en C++ para que se pueda utilizar libremente con ese lenguaje. En esta implementación se usó la versión 1.2. En [18] se puede obtener mayor información.
- GSL: Es una biblioteca numérica para la programación en C y C++. Algunas de las áreas que cubre son: Números complejos, vectores, matrices, raíces polinomiales, diferenciación numérica, estadística, álgebra lineal, etc. En esta implementación se uso la versión 1.7. En [19] se puede obtener mayor información.

4.3. Algoritmo de Lectura de la Malla Inicial

En esta sección se describe y analiza un algoritmo para obtener la malla inicial del objeto a modelar y almacenarla en el objeto correspondiente.

El algoritmo lee desde los archivos cms_1 de Comsol[1]. Una de las metas importantes a lograr en la implementación de este algoritmo es implementarlo en forma integra usando funciones de lectura de C++, y dejar de lado las funciones en C que se usaban para la lectura en la aplicación del legado. Las funciones de lectura en C tienen la desventaja de ser inseguras por el pobre manejo de errores que mantienen.

4.3.1. Lectura desde un Archivo de Texto Comsol

En este algoritmo se recibe un archivo de texto generado a partir de un modelamiento realizado en Comsol. En la generación desde este tipo de archivo se manejan solo caras triangulares, es decir, tres índices por cada cara de la malla. Los índices indican los puntos que la definen.

Un archivo Comsol presenta la siguiente información:

1. Lista con la posición de cada punto (x,y,z).

[1]Software de modelamiento de fenómenos físicos.

2. Línea separadora. Sirve para distinguir cuando se termina la lista de puntos y comienza la lista de caras. La línea es la siguiente: " %Elements (triangular)".

3. Lista de cada una de las caras, indicando los índices de los puntos que conforman la cara.

A partir de esta información se debe generar la malla inicial con su conjunto de Nodos, Arcos y Caras tal como se describió en la Sección 2.4.3.

Notar que en este formato, no se indica ni número de nodos, ni número de caras, ni número de arcos. Estos valores deben ser obtenidos de los datos leídos.

El algoritmo para la lectura y generación de la malla, realiza lo siguiente:

1. Se recibe como entrada el archivo Comsol.

2. Se recorre el archivo, obteniendo la posición de cada uno de los puntos de la malla. Se crean los nodos de la malla con información de este archivo y se agregan al conjunto de nodos. Se continúa la lectura hasta encontrar la primera línea separadora.

3. Se recorre el archivo obteniendo los índices de los nodos para cada una de las caras de la malla.

4. Se obtienen los arcos. Por eficiencia, los índices de los arcos se manejan con la estructura Map[2], la cual maneja una lista que asocia el índice de un arco, con los índices a sus respectivos nodos que lo forman. La estructura de datos Map tiene la ventaja de almacenar la información y una llave para acceder a ella. Luego, la búsqueda de esta información se hace en orden logarítmico, es decir, $\log(n)$, con n el número de elementos almacenados.

5. Se definen las normales para cada uno de los nodos. La normal de cada nodo es calculada como el promedio de las normales de las caras vecinas.

El pseudocódigo del algoritmo se puede ver en el Apéndice B. En la Figura 4.1 se puede ver un ejemplo de una malla generada con archivos obtenidos desde un modelamiento en Comsol.

Análisis de tiempo

Sea N el número de nodos que contiene el archivo Comsol, C el número de caras, cada una formada por 3 puntos y 3 arcos, y A el número de arcos que se forman. Luego, en el algoritmo se realizan las siguientes operaciones:

- Se recorre casa uno de los N nodos. Esto es O(N).

[2]Map: Estructura de la biblioteca estándar de C++

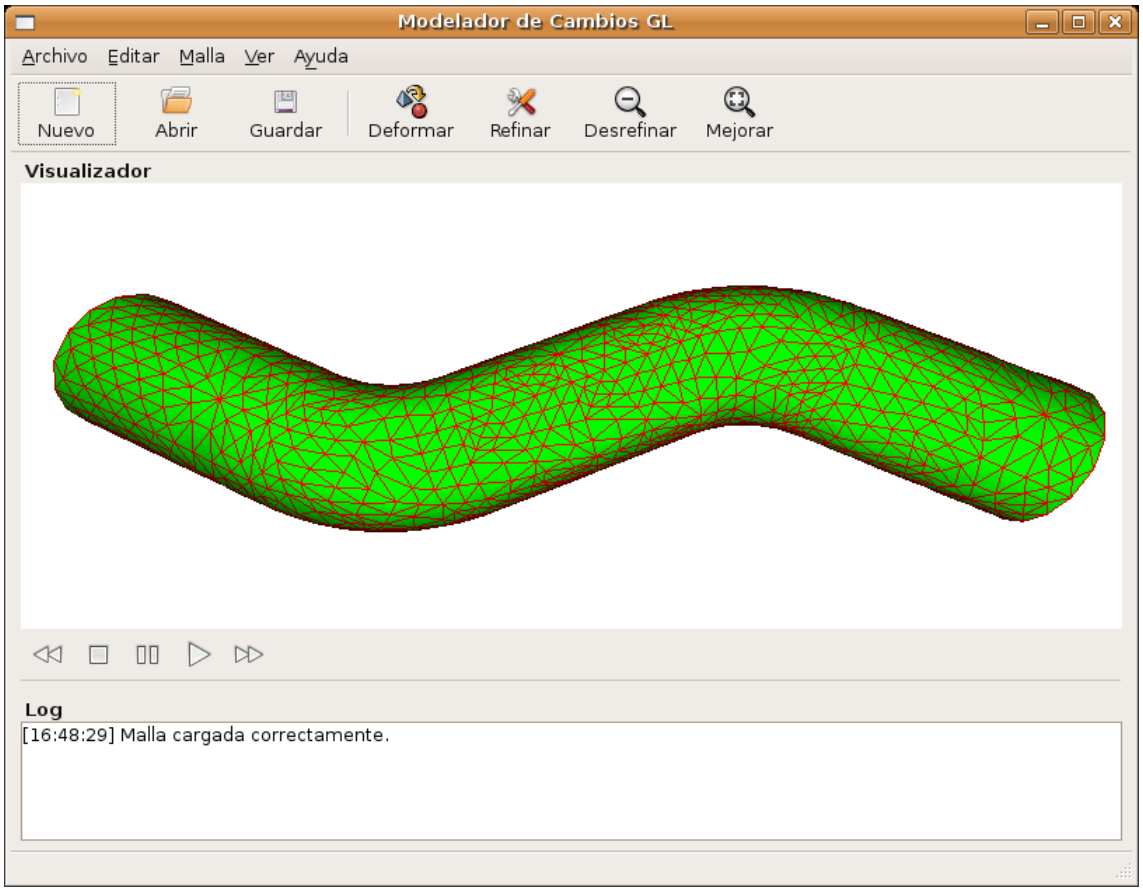


Figura 4.1: Malla Generada desde un Archivo Comsol

- Luego, se recorre cada una de las C caras. Para cada cara se recorren cada uno de sus nodos. Aquí, se busca cada par (no ordenado) de índices a nodos $\left(N_i, N_{i+1mod(K)}\right)$ y el par $\left(N_{i+1mod(K)}, N_i\right)$con $i = 0, 1, 2$ en el conjunto actual de Arcos usando la estructura Map. Esto demora $log(N)$ con N el número de nodos. En el peor caso, cada arco formado por el par indicado, dará origen a un nuevo arco, con lo cual siempre se hace una búsqueda completa sobre el conjunto de Arcos.
- Se lee la concentración de cada uno de los nodos si el archivo incluye esta información. Esto es O(N).

El número máximo que puede tener el conjunto de arcos, es el número total A de arcos, con lo cual en el peor caso se tiene $O(N+3C\cdot 2\log(A)+N)$. Despejando esta ecuación y considerando que siempre se cumple que $A \leq 3C$, tenemos un orden total de $O\left(2N + 6C\log(3C)\right)$. Es decir, el algoritmo nos queda lineal en cuanto al número de nodos, y $C\log\left(C\right)$ en cuanto al número de caras.

4.4. Información y Estadísticas de la Malla

Para un correcto uso de la aplicación, es esencial que el usuario pueda obtener buena información antes de realizar sus operaciones. Para esto se implementó una función para mostrar estadísticas e información de la malla. Esta característica es capaz de mostrar en pantalla una gran cantidad de información importante para el uso de la aplicación. Esta información incluye:

- Numero de caras, arcos y nodos de la malla.
- Área mínima, máxima y promedio de las caras.
- Ángulo mínimo y máximo encontrado en alguna cara.
- Largo de los arcos mínimo, máximo y promedio.
- Histograma de áreas que agrupa cantidad de caras de la malla según un rango de áreas en particular.

Toda esta información es directamente accesible en cualquier punto de la aplicación, desplegándose una ventana como la mostrada en la Figura 4.2.

La principal importancia de una característica como ésta, está en la facilidad que se le da al usuario para tomar decisiones sobre que operaciones realizar. Por ejemplo, al ver estas estadísticas uno puede notar rápidamente si hay un gran porcentaje de caras que tienen un área demasiado pequeña con lo que se necesitaría realizarle un desrefinamiento a la malla. Por el contrario uno pudiera encontrar que hay caras que tienen un área demasiado grande por lo que sería necesario hacer un refinamiento. Más aún, uno puede ver la calidad de la

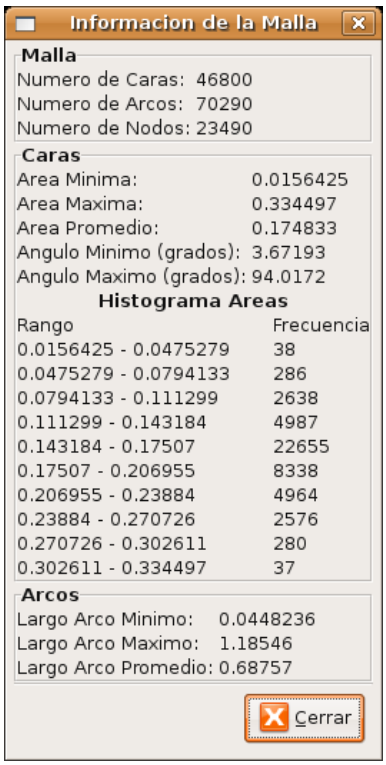


Figura 4.2: Estadísticas e Información de la Malla

malla revisando la calidad de los ángulos, y luego ejecutar algún algoritmo de refinamiento para el mejoramiento de los ángulos como Lepp-Delaunay[15]. En fin, los usos que se le puede dar a esta característica son bastantes y de mucha importancia.

Para la implementación se desarrolló una clase InformacionMalla la cual entrega a la Interfaz Gráfica de la aplicación un arreglo con toda la información para ser desplegada en pantalla. La implementación InformacionMalla se desarrolló con la ayuda de las librerías GSL las cuales proveen funciones muy útiles y eficientes para el manejo de estadísticas como histogramas, medias, etc. Se buscó la forma de implementar estos métodos de la forma mas eficiente posible de tal manera de hacer solamente una pasada por cada cara, arco y nodo de la malla. Finalmente, se obtuvo una implementación para calcular todas las estadísticas de la malla de $O(C + A + N)$ donde C es el número de caras, A es el numero de arcos y N el numero de nodos.

4.5. Algoritmos de Transformación

En esta sección se describen y analizan los nuevos algoritmos de transformaciones locales en la malla. Estas funciones son extremadamente útiles como base para implementar los algoritmos que actúan sobre la malla tales como el refinamiento, desrefinamiento y la deformación de la malla.

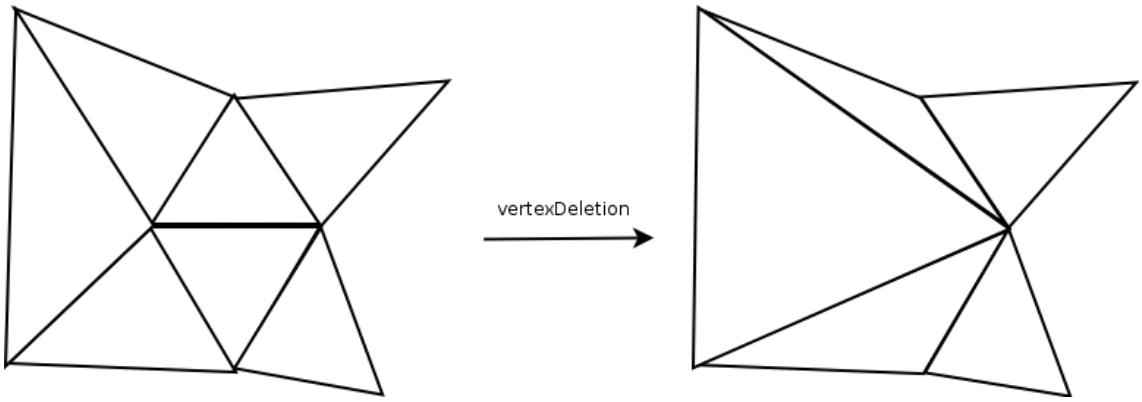


Figura 4.3: Transformación Vertex-Deletion

En la Sección 2.4.4 se pudo ver la implementación de algunos de estos algoritmos en la aplicación legada. Estos eran Edge-Collapse, Edge-Split y Edge-Flip. Con las pruebas realizadas a la aplicación se pudo notar que estas transformaciones básicas no contaban con la robustez necesaria al ejecutar ciertas operaciones en la malla. Las transformaciones Edge-Split y Edge-Flip fueron corregidas para así resolver una serie de problemas de robustez que tenían, tales como chequeos de inconsistencia con errores, problemas de precisión y problemas en el uso de memoria los cuales se detallan más profundamente en la Seccion 4.9. Lamentablemente, Edge-Collapse tuvo que ser reimplementada completamente pues tenía grandes problemas de robustez y a menudo generaba mallas totalmente inconsistentes. Por esta razón, se decidió hacer una completa reimplementación que incluye la programación del método Vertex-Deletion que ayuda bastante a simplificar el problema.

Es importante destacar que para la implementación de cada uno de estos algoritmos es necesario mantener un pequeño valor épsilon en la aplicación como margen de error para evitar posibles problemas de precisión. Mas allá de que los algoritmos funcionen en forma correcta, siempre pueden ocurrir errores de precisión cuando se trabaja con aritmética de punto flotante.

4.5.1. Transformación Vertex-Deletion

En la transformación Vertex-Deletion se decidió abarcar el problema de colapsar el arco con un enfoque más simple. A diferencia de la 1º versión de Edge-Collapse, en este enfoque no se elimina el arco completo junto a todos sus nodos, caras y arcos que lo rodean. Aquí, en cambio se elimina solo un vértice en particular, y las caras alrededor del arco y luego se modifican las caras vecinas para apuntar hacia el vértice opuesto. Una demostración gráfica de esta transformación puede ser visualizada en la Figura 4.3.

Este enfoque tiene dos grandes ventajas con respecto al anterior:

- Al borrar solamente uno de los vértices del arco, la cantidad de elementos a eliminar y modificar de la malla disminuye sustancialmente. Solamente se deben modificar las

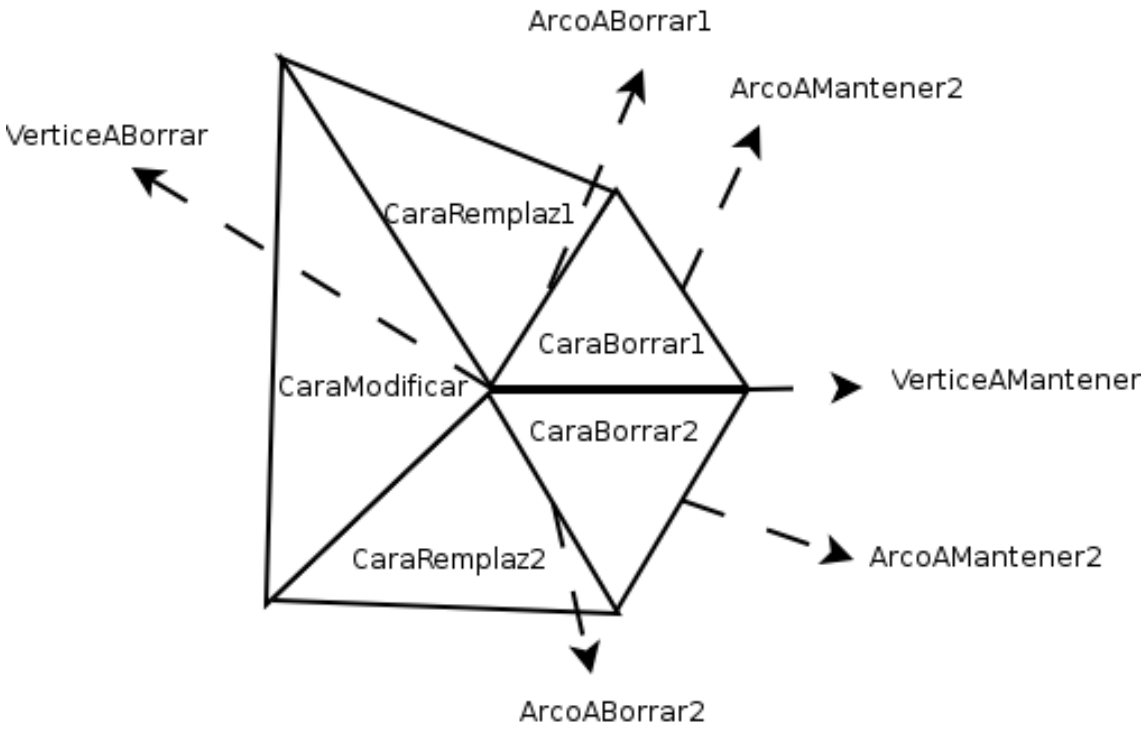


Figura 4.4: Variables usadas en Algoritmo Vertex-Deletion

relaciones de elementos vecinos en uno de los vértices. Esto simplifica bastante el algoritmo, por lo que se puede hacer una implementación más simple, robusta y libre de errores.

- Esta transformación no provoca deformaciones en las orillas de la malla. Si un vértice del arco a colapsar se encuentra en un borde de la malla, el algoritmo no permite aplicar Deletion sobre él. Si se aplica Vertex-Deletion sobre el vértice opuesto, los arcos vecinos apuntarán hacia el vértice de la orilla de la malla sin provocar una deformación.
- El algoritmo es más eficiente ya que se realiza una menor cantidad de operaciones.

El pseudocódigo del algoritmo se puede ver en el Algoritmo 1 y 2. Para un mayor entendimiento del funcionamiento del algoritmo, se incluyen en la Figura 4.4 indicaciones de las variables usadas en la implementación.

Cabe notar que existen casos donde esta transformación no se puede aplicar ya que produciría un caso degenerado tal como se muestra en la Figura 4.5. Este caso ocurre cuando una de las caras que separa el arco a colapsar tiene caras vecinas, que además son vecinas entre ellas. En la Figura 4.5 se observa claramente destacado en gris, como una cara queda volteada y superpuesta a otra compartiendo los mismos nodos. Cuando se detecta que ocurre este caso, el algoritmo no efectúa la transformación de Vertex-Deletion y deja la malla como estaba. En la aplicación antigua y en [1] se propone un método para solucionar este tipo de casos. El método consiste en efectuar la transformación Edge-Flip sobre el arco antes de colapsarlo. En pruebas realizadas al programa se pudo ver que esta estrategia no es muy efectiva ya que en

Algorithm 1 Vertex-Deletion - Parte 1

```
int vertexDeletion(indArcoAColapsar, indVerticeABorrar){
    Obtenemos los indices a todas las variables indicadas en la Figura 4.4;
    //Modificamos las caras.
    while (iteramos por todas las CaraRemplaz y CaraModificar que tengamos){
        caraAModificar=Cara que estamos iterando en este momento;
        //Modificamos sus nodos.
        caraAModificar->changeNodo(VerticeABorrar,VerticeAMantener);
        for (int j=0; j<caraAModificar->getNumNodos(); j++){
            int NodoAModificar=caraAModificar->getNodo(j);
            nodoAModificar->eraseCara(CaraABorrar1);
            nodoAModificar->eraseCara(CaraABorrar2);
            nodoAModificar->eraseArco(ArcoABorrar1);
            nodoAModificar->eraseArco(ArcoABorrar2);
            }
        //Modificamos sus arcos.
        for (int j=0; j<caraAModificar->getNumArcos(); j++){
            int ArcoAModificar=caraAModificar->getArcos(j);
            if (ArcoAModificar==ArcoABorrar1)
                caraAModificar->changeArco(ArcoAModificar,ArcoAMantener1);
            else if (ArcoAModificar==ArcoABorrar2)
                caraAModificar->changeArco(ArcoAModificar,ArcoAMantener2);
            else if (arcoAModificar->getNodo1()==VerticeABorrar)
                arcoAModificar->setNodo1(VerticeAMantener);
            else if (arcoAModificar->getNodo2()==VerticeABorrar)
                arcoAModificar->setNodo2(VerticeAMantener);
            }
        }
        //Modificamos los nodos mantenido.
        verticeAMantener->eraseCara(CaraABorrar1);
        verticeAMantener->eraseCara(CaraABorrar2);
        verticeAMantener->eraseArco(ArcoABorrar1);
        verticeAMantener->eraseArco(ArcoABorrar2);
        verticeAMantener->eraseArco(ArcoAColapsar);
        while (iteramos por todas las CaraRemplaz y CaraModif que tengamos){
           cara=cara que iteramos en este ciclo;
            verticeAMantener->addCara(cara);
            for (int j=0; j<cara->getNumArcos(); j++){
                int arco=cara->getArcos(j);
                if (arco!=ArcoAMantener1 && arco!=ArcoAMantener2 &&
                   (arco->getNodo1()==VerticeAMantener ||
                    arco->getNodo2()==VerticeAMantener))
                        verticeAMantener->addArco(indArco);
                }
        }
        Continua en Parte 2;
```

Algorithm 2 Vertex-Deletion - Parte 2

```
//Modificamos los arcos mantenidos.
if (arcoAMantener1->getCara1()==indCaraABorrar1)
    arcoAMantener1->setCara1(indCaraAReemplazar1);
else if (arcoAMantener1->getCara2()==indCaraABorrar1)
    arcoAMantener1->setCara2(indCaraAReemplazar1);

if (arcoAMantener2->getCara1()==indCaraABorrar2)
    arcoAMantener2->setCara1(indCaraAReemplazar2);
else if (arcoAMantener2->getCara2()==indCaraABorrar2)
    arcoAMantener2->setCara2(indCaraAReemplazar2);

//Borramos
nodos->eraseNodo(indVerticeABorrar);
caras->eraseCara(indCaraABorrar1);
caras->eraseCara(indCaraABorrar2);
arcos->eraseArco(indArcoABorrar1);
arcos->eraseArco(indArcoABorrar2);
arcos->eraseArco(indArcoAColapsar);

return 0;
}
```

bastantes ocasiones el algoritmo se quedaba encerrado en ciclos infinitos tratando de hacer flips a los arcos sin lograr nunca llegar a una configuración adecuada para realizar el Vertex-Deletion. Por esta razón, en la nueva aplicación se decidió implementar el algoritmo de tal manera que detecte este caso y deje la malla intacta para no provocar mayores problemas de robustez al algoritmo.

Otro caso degenerado ocurre cuando el polígono formado por la vecindad de caras del Nodo que va a ser trasladado forman un polígono no convexo. Como vemos en la Figura 4.6, cuando la vecindad es no convexa, una cara se podra dar vuelta. Es decir, su normal quedará apuntando hacia el lado opuesto al que tenía. Para detectar estos casos, se calcula la normal de cada cara antes y después del Vertex-Deletion. Si el ángulo entre ambas normales es mayor a 90^o, significa que la cara se volteó y por lo tanto no se puede realizar la transformación Vertex-Deletion. Para calcular el angulo entre ambas normales basta con utilizar la siguiente formula:

$$\cos\theta = \frac{u \cdot v}{\|u\| \cdot \|v\|}$$

4.5.2. Nueva Transformación Edge-Collapse

Una vez implementado el algoritmo Vertex-Deletion, la creación del algoritmo Edge-Collapse se vuelve bastante más fácil. Esta nueva versión del algoritmo esta pensada para reemplazar

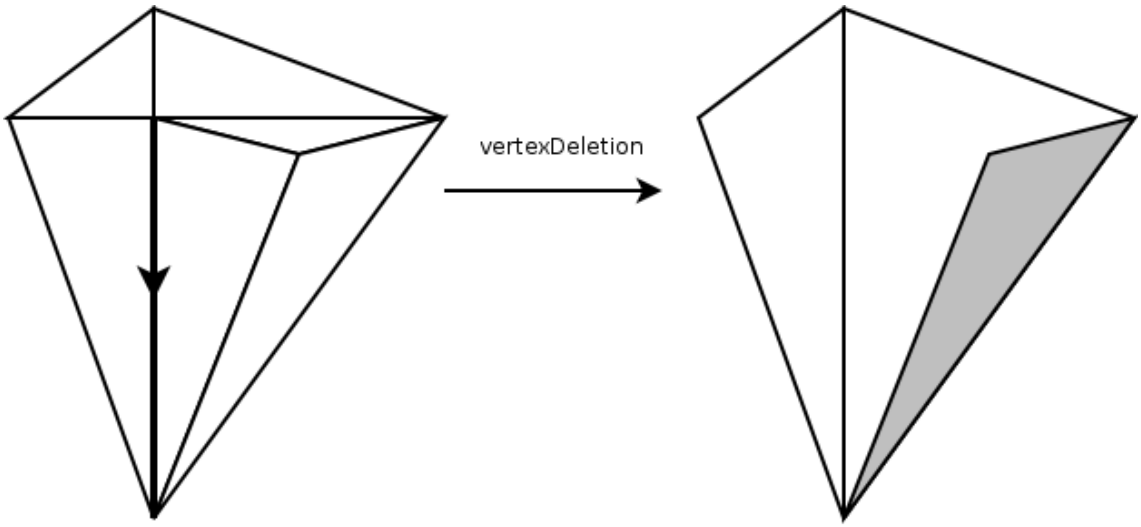


Figura 4.5: Caso Degenerado de Vertex-Deletion en Caras Vecinas

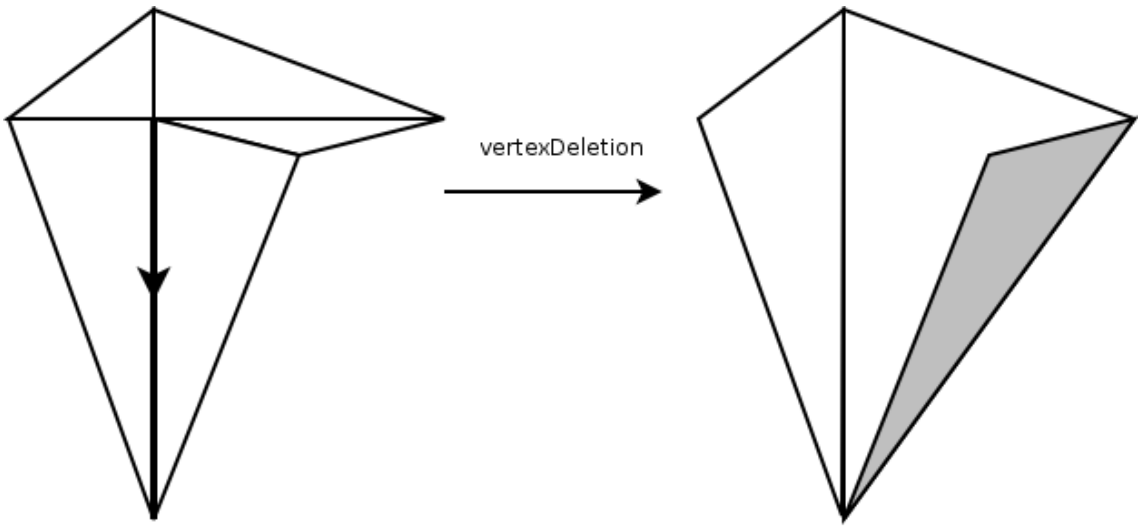


Figura 4.6: Caso Degenerado de Vertex-Deletion en Sectores no Convexos

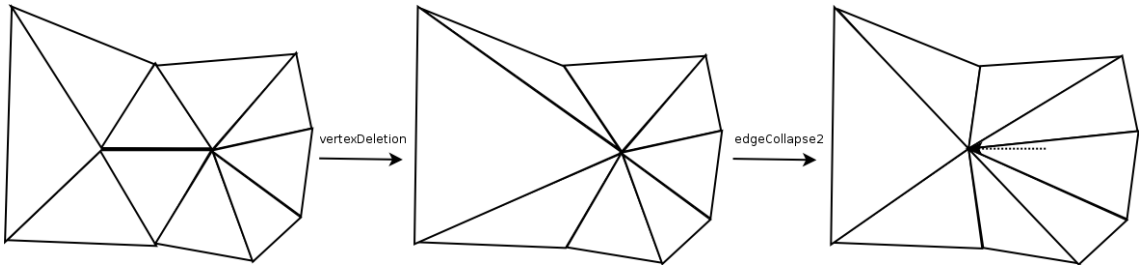


Figura 4.7: Transformación Edge-Collapse

al antiguo Edge-Collapse visto en la Sección 2.4.4. Esta versión deberá ser robusta y además proveer la misma funcionalidad de la versión antigua. Es decir, debe tener la posibilidad de seleccionar en que punto se debe colapsar el arco.

Vertex-Deletion ya hace la labor difícil de colapsar un arco y dejar la malla topológicamente correcta. Lo único que falta, es que el vertice en donde se colapso el arco quede en la posición indicada por el usuario. Para lograr esto, solo basta cambiar las coordenadas del vértice que se mantuvo del arco colapsado durante la transformación Vertex-Deletion. La malla continuará siendo consistente topológicamente ya que solo se le hace un cambio de coordenadas a un nodo. Las coordenadas deben ser actualizadas por un punto calculado a partir de la distancia dada por el usuario con respecto al vértice del arco borrado. En la Figura 4.7 se puede ver gráficamente como funciona el algoritmo.

4.6. Algoritmo de Desrefinamiento

Un importante método para la malla, son los métodos de desrefinamiento. Muchas veces, al aplicarle transformaciones y deformaciones a la malla, podemos provocar que las caras de la malla se vuelvan muy pequeñas. Por ejemplo, esta situación se provoca normalmente en las caras interiores de un tronco curvo de un árbol a medida que esté va creciendo como se puede ver en la Figura 4.8.

Tener un algoritmo de desrefinamiento de la malla es muy importante. Esto debido a que:

- Una buena malla no debería contener caras tan pequeñas ya que a la larga puede causar problemas de precisión.
- Un algoritmo de desrefinamiento puede modelar en forma bastante similar lo que ocurre en la naturaleza. Tal como se menciona en [22], la competición por crecer de las células de un árbol, puede hacer que algunas terminen desapareciendo. Por los estudios realizados, se ha observado que las células desaparecen cuando su área disminuye por debajo de un cierto valor o cuando sus células vecinas toman contacto entre ellas.

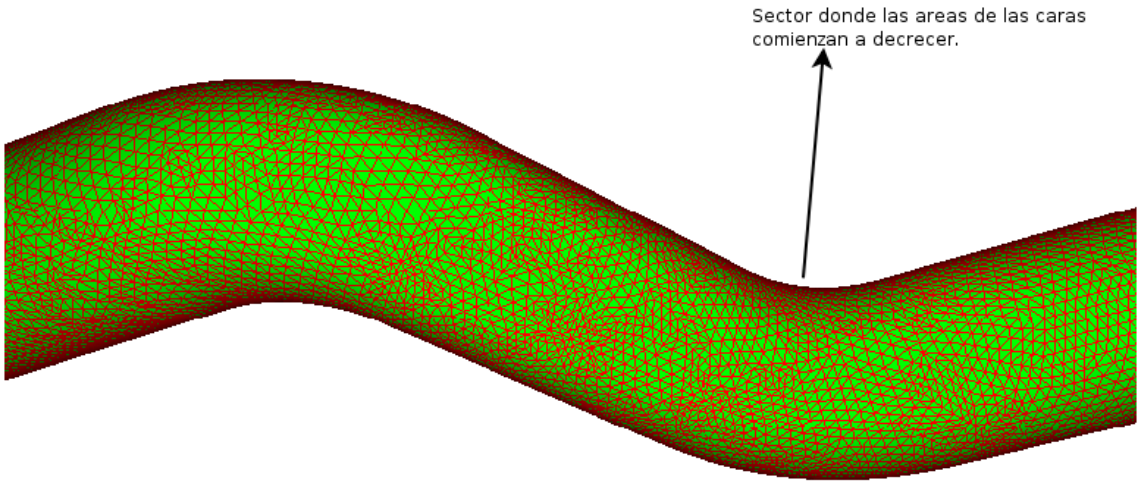


Figura 4.8: Sector donde Caras Disminuyen su Área Durante el Crecimiento del Árbol

4.6.1. Desrefinamiento por Edge-Collapse

La estrategia utilizada para este algoritmo de desrefinamiento está basado en la nueva transformación Edge-Collapse vista en la Sección 4.5.2. Al haber implementado ya la transformación Edge-Collapse, la implementación de un método de desrefinamiento se hace trivial. Solo basta tomar en cuenta que no todas las caras pueden ser desrefinadas ya que hay existen casos especiales. Por ejemplo, las caras de los bordes en muchas ocasiones no pueden ser desrefinadas ya que provocarían un deformamiento de la malla. Por esta razón, en el algoritmo se maneja una lista de caras excluidas las cuales ya se ha visto que no pueden ser desrefinadas. Si no manejamos esta lista, podríamos dejar pegado el algoritmo en un ciclo infinito tratando de desrefinar siempre la misma cara.

El algoritmo realiza el proceso de desrefinar cada una de las caras de la malla que no cumplen el criterio dado como argumento. El algoritmo hace pasadas completas por la malla hasta ver que ya no se puede seguir desrefinando la malla. El pseudocódigo del algoritmo se puede ver en el Algoritmo 3.

Este algoritmo ha demostrado funcionar bastante bien, especialmente para desrefinar las caras pequeñas que se producen en las curvas cóncavas de lo que sería el tronco del árbol. Si estas caras no se desrefinan, producirían a futuro problemas de precisión con el resto de los algoritmos. El desrefinamiento por Edge-Collapse soluciona estos problemas. Un ejemplo de su funcionamiento en un caso como el recién enunciado se puede ver en la Figura 4.9.

4.7. Algoritmos de Desplazamiento

En esta sección se describe el nuevo algoritmo utilizado para el desplazamiento de los nodos que componen la malla. Este algoritmo trata de resolver carencias del algoritmo implementado

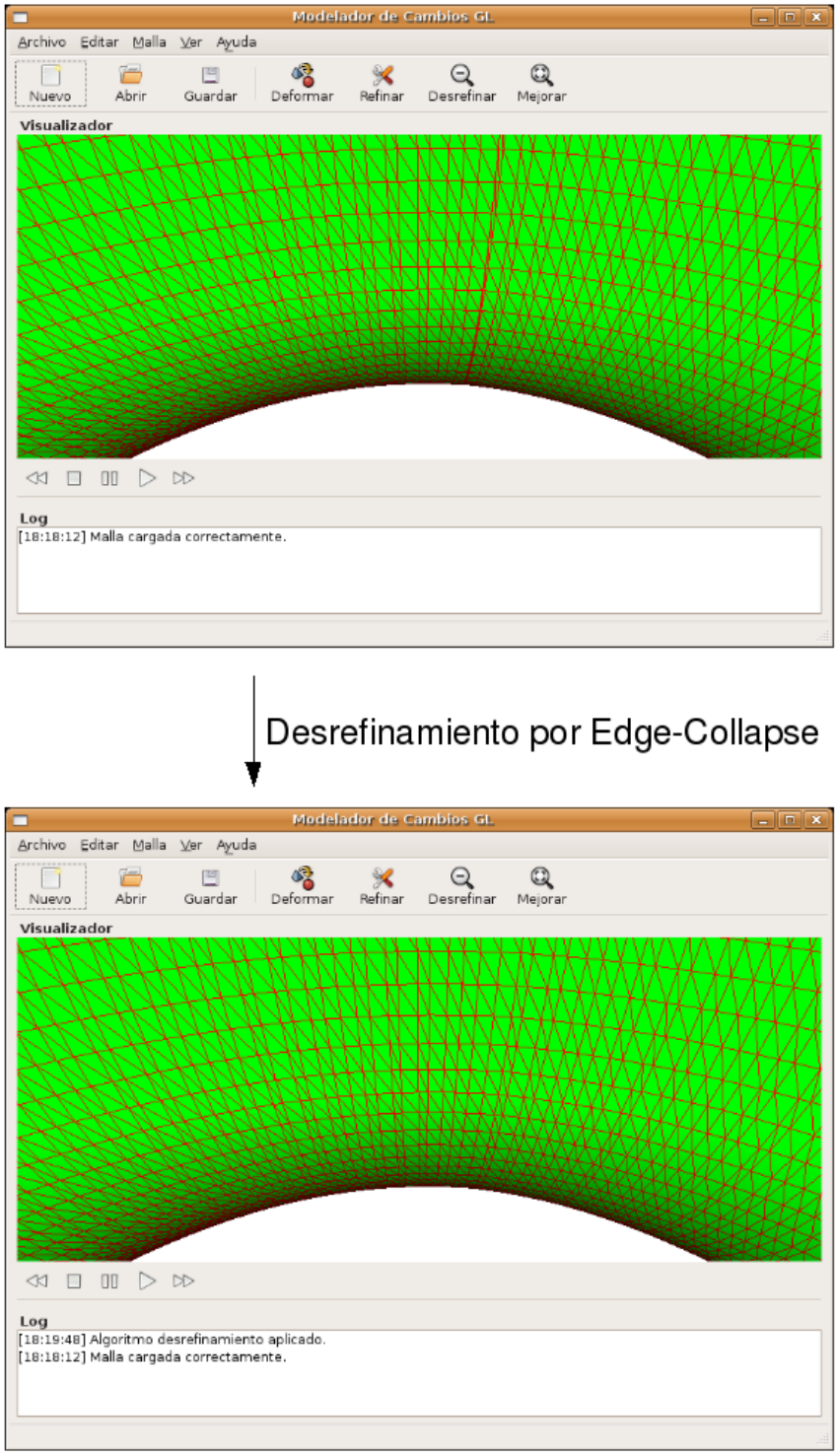


Figura 4.9: Ejemplo de Desrefinamiento

Algorithm 3 DesrefinamientoEdgeCollapse

```
void desrefinamientoEdgeCollapse(Malla *malla, Criterio *criterio){
    Caras *caras = malla->getCaras();
    bool seDesrefinoCara=true;
    while(seDesrefinoCara==true){
        seDesrefinoCara=false;
        for(int i=0; i<caras->getNumCaras(); i++) {
        if(la cara no cumple el criterio &&
           la cara no esta excluida) {
            desrefinar(malla,i);
            seDesrefinoCara=true;
            }
        }
    }
}
int desrefinar(Malla* malla, int indCara){
    int arcomascorto=malla->getArcoMasCorto(indCara);
    int resultado=malla->edgeCollapse2(arcomascorto,0.5);
    if (resultado==-1)
        excluidos.push_back(indCara);
    return resultado;
    }
```

en [1], tomando un nuevo enfoque para resolver los problemas generados por las colisiones en el momento del desplazamiento.

El algoritmo presentado mueve los nodos de la malla según su concentración y su normal. La concentración de cada nodo indica la cantidad de unidades que será movido en el lapso de un tiempo determinado. Es decir, si por ejemplo un nodo tiene una concentración de 5 unidades, quiere decir que en un tiempo de t=1 se desplazará 5 unidades en la dirección indicada por su vector normal. La concentración está dada por los datos de entrada donde se asignan de acuerdo a lo que se quiere modelar. Por ejemplo, si se quiere modelar un tronco con un número determinado de ramas que van creciendo, se genera un cilindro y se asigna una concentración mayor en determinados nodos, para que estos vayan generando las ramas. Las normales de cada nodo son unitarias e indican la dirección en que se desplazará el nodo. En el caso que las normales de los nodos no sean ingresadas como datos de entrada, estas se calculan como el promedio de las normales de las caras vecinas que comparte.

4.7.1. Desplazamiento con Verificación de Nodos Vecinos

El algoritmo de desplazamiento con verificación local implementado en [1] permitía solamente el desplazamiento hasta las primeras colisiones de caras vecinas. Su implementación era compleja y por ello tenía una gran falta de robustez. El algoritmo consideraba una gran

Algorithm 4 Deformación Con Verificación de Nodos Vecinos

```
void deformacionConVerificacionNodoVecinos(Malla *malla){
    repairMalla(malla);
    moverHasta(malla);
    }
void repairMalla(Malla *malla){
    for(int indNodo=0; indNodo<malla->maxNodo(); indNodo++){
        bool seIntersecta=checkInterseccionNodo(malla, indNodo);
        if (seIntersecta)
            corrigeInconsistencia(malla, indNodo);
        }
    }
void moverHasta(Malla* malla){
    malla->getNodos()->moverTodosSegunConcentracion(val);
    }
```

cantidad de casos especiales, lo que hacía muy difícil implementar un algoritmo eficiente y libre de errores.

En esta memoria se decidió tomar otro enfoque e implementar un algoritmo basado en una estrategia que permitiera desplazar la malla mas allá de las primeras inconsistencias. Esta estrategia está basada en las ideas encontradas en [21]. Este algoritmo tiene la ventaja de no solo hacer chequeos a nivel de caras vecinas compartidas por un arco, sino que también chequea consistencia a nivel de caras vecinas que comparten un mismo nodo.

El algoritmo consta de dos etapas. En la primera se hacen todas las reparaciones y chequeos de la malla para verificar trayectorias que por problemas de precisión produzcan choques entre nodos vecinos. Si el algoritmo detecta una inconsistencia a producirse a nivel de los nodos vecinos, llama a un método para corregir la inconsistencia. En la segunda etapa, el algoritmo simplemente mueve la malla según sus normales, tomando en cuenta que las trayectorias ya han sido corregidas. El pseudocódigo de esta secuencia se puede ver en el Algoritmo 4.

La parte más compleja de este algoritmo es sin duda el chequeo y reparación de las trayectorias. Para entender de mejor manera qué significa tener inconsistencias a nivel de los nodos vecinos podemos observar la Figura 4.10. El polígono que se forma en la parte baja de la figura representa la parte de la malla actual. Las flechas indican las normales de cada nodo, y el polígono superior indican las posiciones futuras de los nodos en la malla. En una malla sin inconsistencias los nodos vecinos con sus trayectorias deberían formar un pseudo-cilindro alrededor del nodo central. Si la trayectoria del nodo central intersecta una de las paredes de este cilindro, implica que ha ocurrido una inconsistencia al haber una intersección entre la trayectoria del desplazamiento del nodo con las trayectorias de los desplazamientos de los nodos vecinos. Esto ocurre probablemente por problemas de precisión.

En el Algoritmo 5 se puede ver en pseudocódigo los pasos para realizar la corrección de las trayectorias. Una parte importante y compleja de este algoritmo fue la parte de detección de intersección entre un vector y una cara. La solución de este problema fue dada gracias al

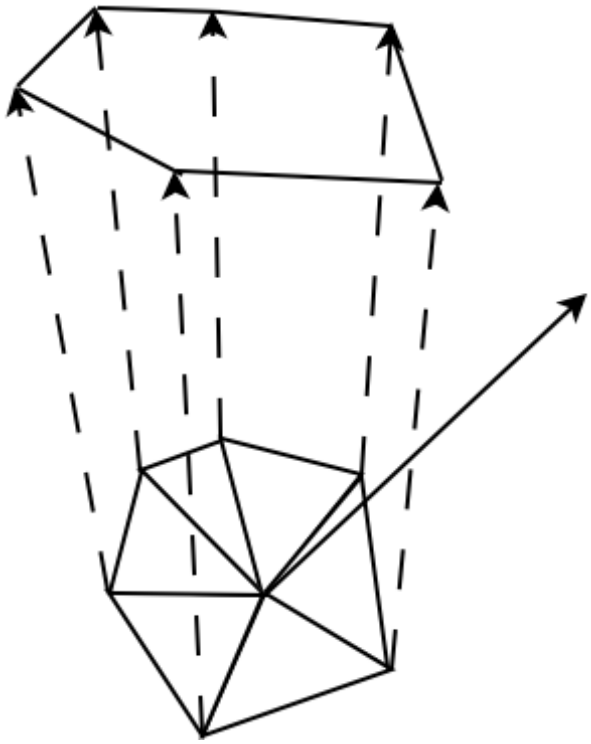


Figura 4.10: Inconsistencia con Nodos Vecinos

algoritmo SegTriInt[3] obtenido de [6]. El algoritmo SegTriInt permite detectar intersecciones entre un segmento y un triangulo, por lo tanto, en nuestro algoritmo se debió calcular qué puntos formarían el segmento y luego dividir cada pared del pseudo-cilindro en dos triángulos. Finalmente se chequea mediante SegTriInt la intersección del segmento con cada uno de estos triángulos.

En caso de encontrar una intersección se modifica la trayectoria del nodo central de tal forma que pase por la parte superior del cilindro como se muestra en la Figura 4.11. Una trayectoria que cumple este requisito es el resultado de una suma de todos los vectores que definen las trayectorias de los nodos vecinos. Esta trayectoria normalizada es la nueva normal del nodo central. El pseudocódigo se puede ver en el Algoritmo 6.

Un caso especial ocurre cuando la nueva trayectoria sigue teniendo inconsistencias. Si se produce esto, se recurre a la última solución disponible para corregir la inconsistencia. Esto es borrar el nodo central mediante el uso de la transformación Vertex-Deletion vista en la Sección 4.5.1. En la Figura 4.12 se puede ver el resultado de esta estrategia. Esta solución produce muy buenos resultados, ya que el caso en donde ocurren este tipo de inconsistencias son análogas a la naturaleza en el caso en que el árbol ha crecido a tal punto que dos células separadas por una célula en el medio no le dan espacio a la célula del medio para crecer. Cuando ocurre esto la célula del medio debería desaparecer, tal como lo hace en el algoritmo.

[3]SegTriInt: Abreviación de Segment Triangle Intersection.

Algorithm 5 Chequeo de Inconsistencias

```
bool checkInterseccionNodo(Malla* malla, int indNodo){
    //Definimos los puntos que componen el segmento de proyeccion del nodo.
    Vect* q=Pos actual de indNodo;
    Vect* r=Pos futura de indNodo
    //Revisamos cada una de las caras vecinas al nodo.
    for(indCaraVecina=0;indCaraVecina<indCarasVecinasMax;indCaraVecina++){
        //Formo una pared del cilindro que rodea la proyeccion.
        Vect* A=caraVecina->getPuntosOpuestos(indNodo)[0];
        Vect* B=caraVecina->getPuntosOpuestos(indNodo)[1];
        Vect* PuntoProyecA=La pos futura de A;
        Vect* PuntoProyecB=La pos futura de B;
        //Chequeamos interseccion con el primer triangulo
        Vect* triangulo1[3]=(A,PuntoProyecA,PuntoProyecB);
        hayInterseccion=segtriint->aplicar(triangulo1,q,r);
        if (resultadoInterseccion==true)
             return true;
        //Chequeamos interseccion con el segundo triangulo
        Vect* triangulo2[3]=(B,A,PuntoProyecB);
        hayInterseccion=segtriint->aplicar(triangulo2,q,r);
        if (resultadoInterseccion==true)
             return true;
        }
    return false;
    }
```

Algorithm 6 Corrección de Inconsistencias en el Desplazamiento

```
bool corrigeInconsistencia (Malla *malla, int indNodo){
    Vect* viejaNormal=indNodo->getNormal();
    for (indArco=0;indArco<indArcosVecinos.size();indArco++){
        Nodo* NodoA=arcoVecino->getNodoDistinto(indNodo);
        Vect* normalVecino=NodoA->getNormal()*val*NodoA->getConcentracion();
        nuevaNormal=nuevaNormalvectorVecino;
        }
    //Normalizamos.
    nuevaNormal = nuevaNormal*(1/nuevaNormal->largo());
    nodo->setNormal(nuevaNormal);
    //Si la inconsistencia no se corrige, borramos los vecinos.
    if (checkInterseccionNodo(malla,indNodo)==true){
        nodo->setNormal(viejaNormal);
        borrarNodoCentral(malla,indNodo);
        return false;
        }
    return true;
    }
```

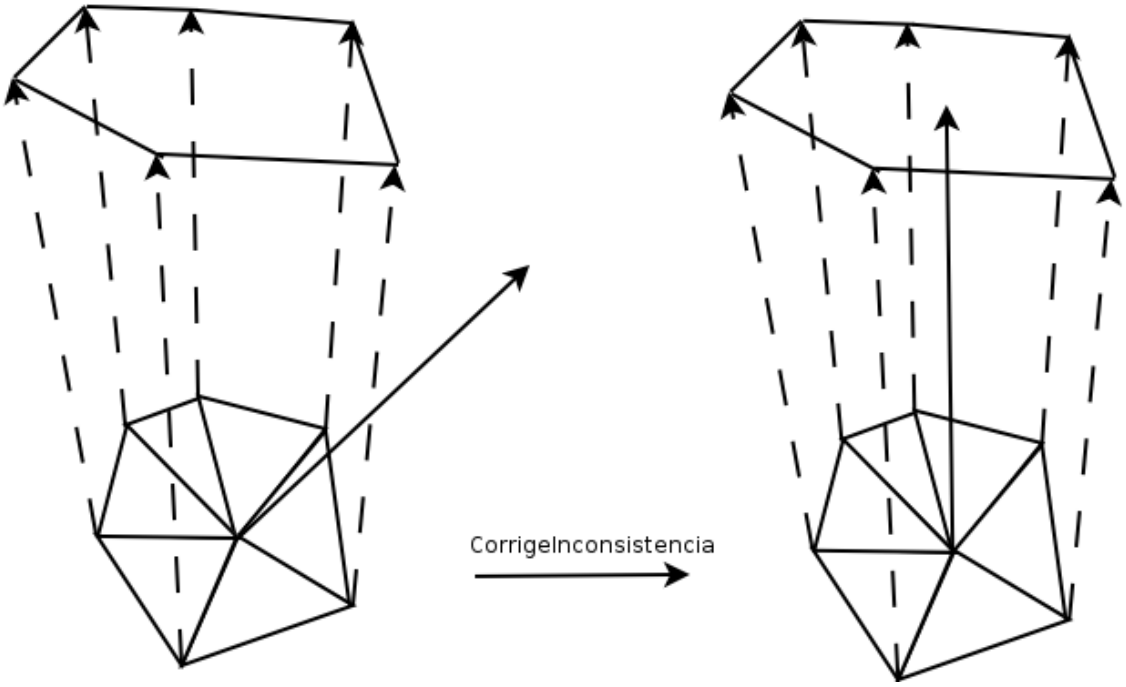


Figura 4.11: Corrección de Inconsistencias en el Desplazamiento

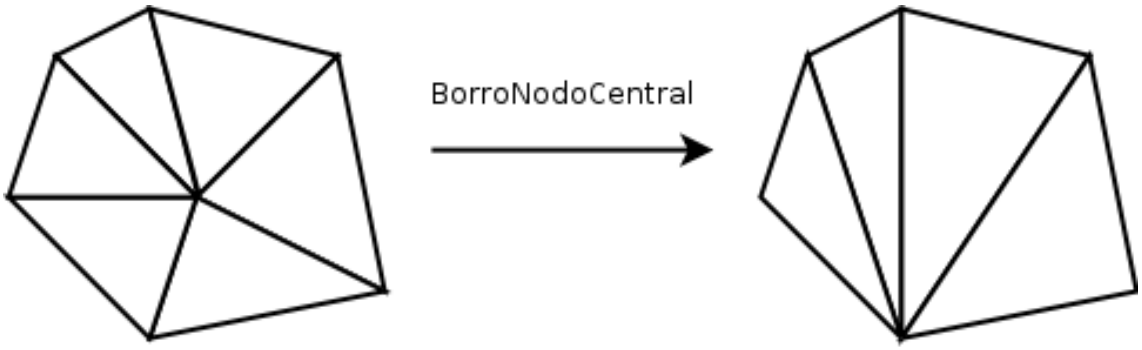


Figura 4.12: Corrección de Inconsistencias Borrando el Nodo Central de la Vecindad

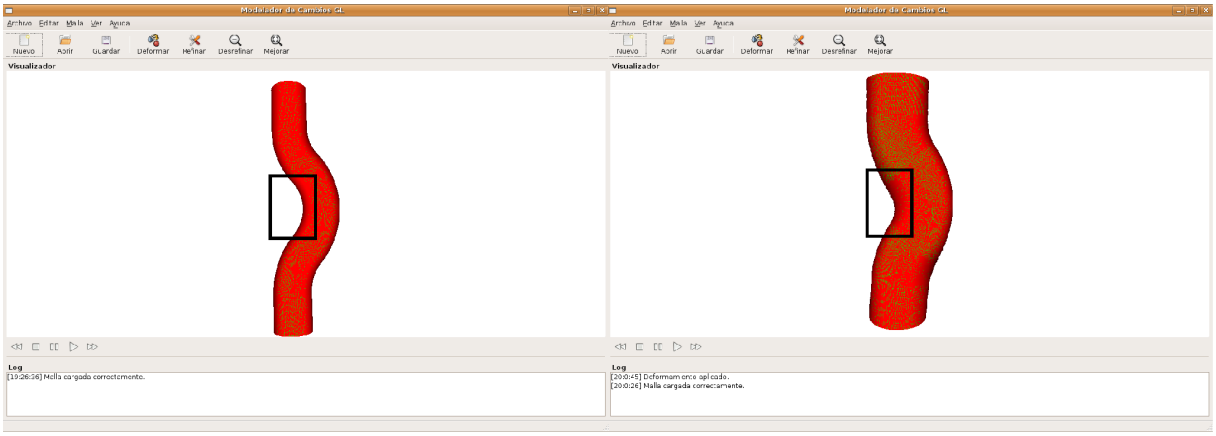


Figura 4.13: Ejemplo de Desplazamiento

Análisis de tiempo

El tiempo de ejecución del algoritmo crece en forma lineal respecto a su número de nodos. El algoritmo realiza un ciclo a través de cada uno de los nodos de la malla para hacer la verificación y si es necesario, corregirla. Es importante destacar que la operación que se hace sobre cada nodo es bastante pesada en tiempo de procesamiento ya que se debe generar el pseudo-cilindro que lo rodea, verificar intersecciones y hacer las correcciones sobre el nodo. El tiempo aproximado de ejecución del algoritmo sobre una malla de 5.000 nodos es de 20 segundos[4].

4.7.2. Ejemplos de Desplazamiento de Mallas

En esta sección se dan ejemplos en que se desplaza la malla de dos maneras distintas:

1. Desplazamiento con el Algoritmo sin verificación de inconsistencias.
2. Desplazamiento con el Algoritmo con verificación de inconsistencias con los nodos vecinos.

Todos los ejemplos se realizaron con la misma malla, la cual fue generada con la información de un archivo de texto de Matlab. Esta malla tiene 23490 nodos, 46800 caras y 70290 arcos.

En la Figura 4.13 se muestra un ejemplo de desplazamiento de una malla hasta t=1, es decir, cada punto se desplaza un valor igual a su concentración en dirección a la normal. El cuadro negro muestra la sección de la malla donde se producen inconsistencias. A esta área le haremos un acercamiento para ver las diferencias entre el desplazamiento sin verificación y el con verificación con los nodos vecinos.

En la Figura podemos ver un acercamiento de la zona a una malla donde se utiliza un desplazamiento sin verificaciones. En la figura se puede ver claramente las inconsistencias en la malla. Se pueden observar caras y arcos incompletos que han sido tapados por otros.

[4]Medición realizada sobre una CPU AMD64 3000+.

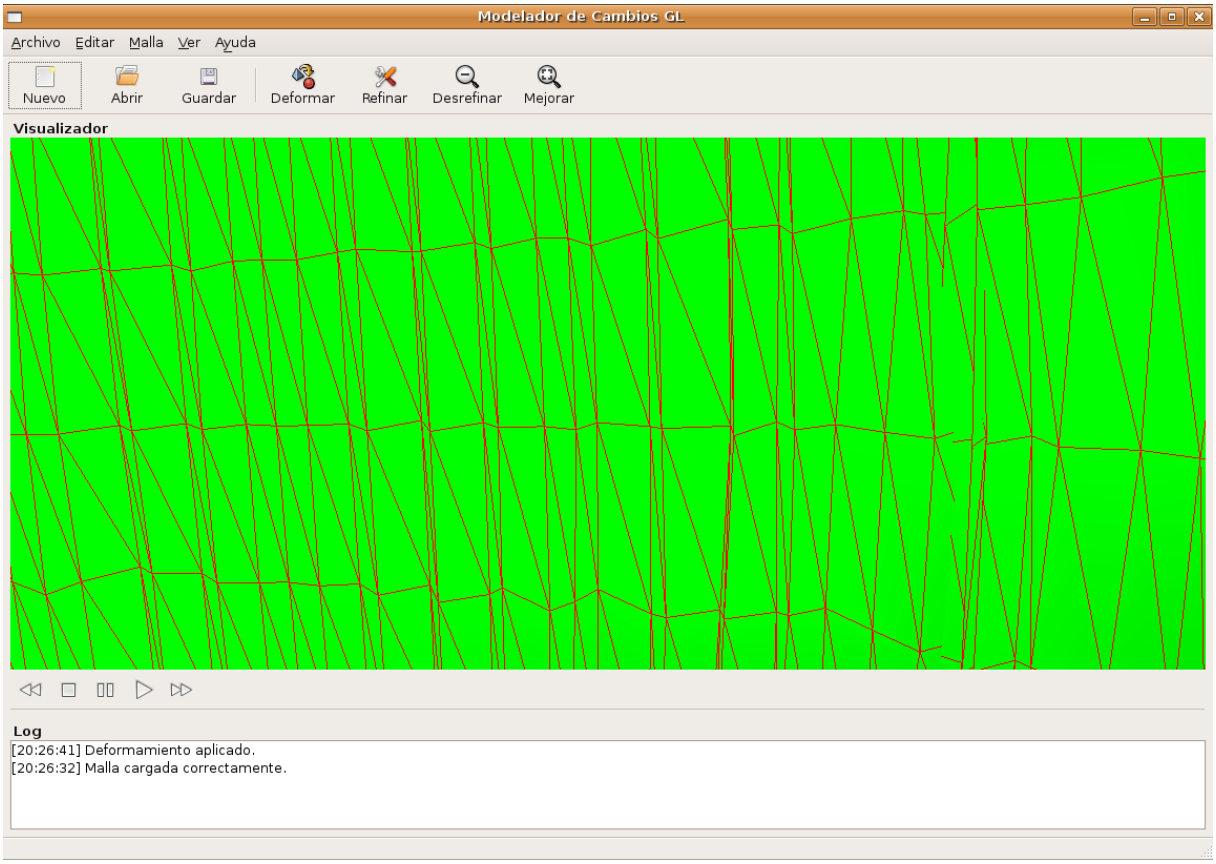


Figura 4.14: Ejemplo de Desplazamiento Sin Verificación

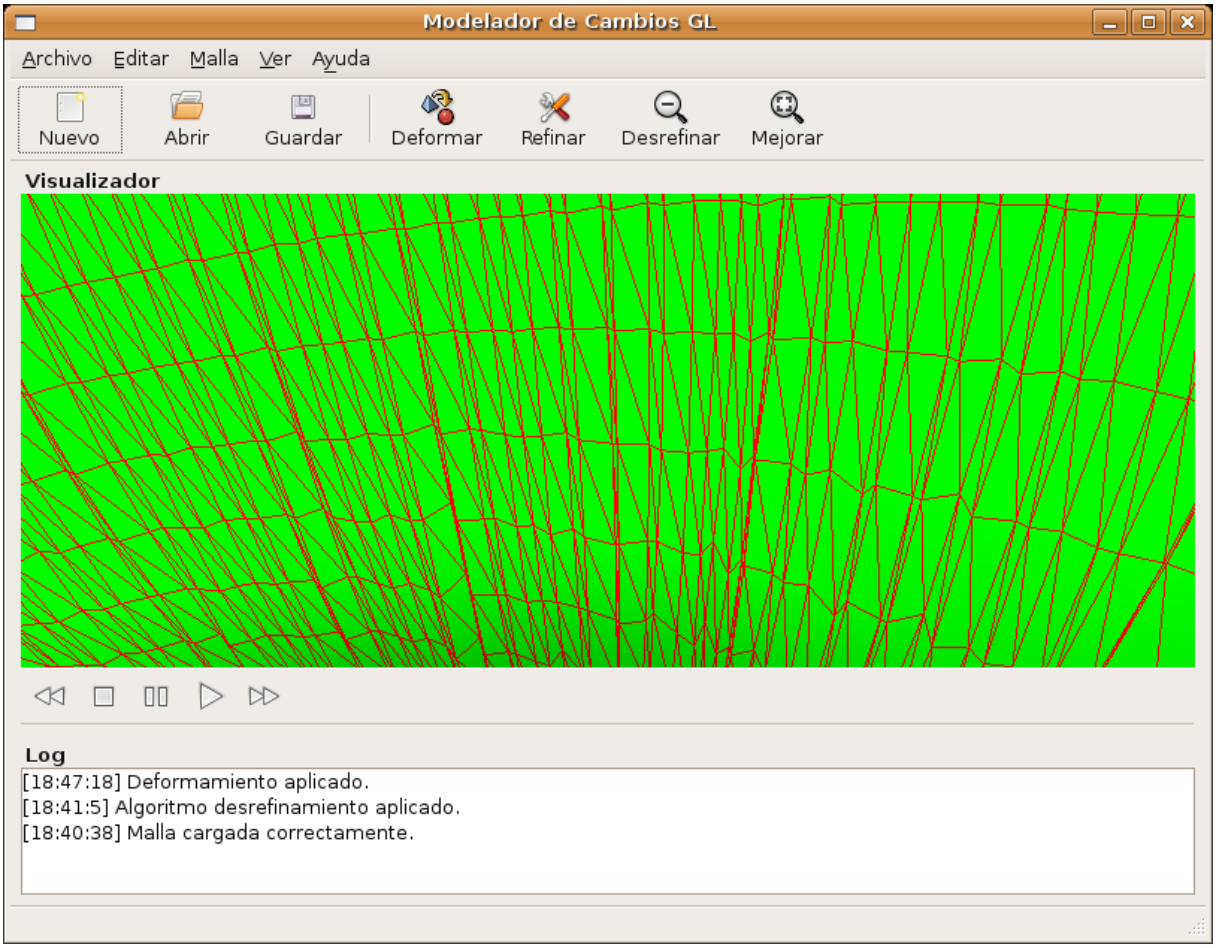


Figura 4.15: Ejemplo de Desplazamiento con Verificación con Nodos Vecinos

Al aplicar en la misma malla original, el algoritmo con verificación por los nodos vecinos, pudimos encontrar un total de 56 inconsistencias las cuales fueron completamente corregidas por el algoritmo. Para corregir las inconsistencias se debió cambiar la trayectoria de 51 de estos nodos, y se eliminó el nodo central en 4 de ellos. La vista del área conflictiva de la malla corregida se puede ver en la Figura 4.15. En la imagen se puede ver claramente que ya no se distinguen inconsistencias como en el caso sin verificación. El único problema que se puede distinguir es que a pesar de la malla ser consistente, muchas caras han disminuido su área a un nivel tan pequeño que podrían causar problemas de precisión en cualquier momento si se sigue trabajando con ella. Para solucionar esto, podemos utilizar el algoritmo de desrefinamiento descrito en la Sección 4.6. Se puede ver en la Figura 4.16 la malla luego de aplicar el algoritmo de desrefinamiento. El resultado es notable ya que se puede apreciar una malla de muy buena calidad para seguir siendo trabajada y procesada.

4.8. Visualización

Una de las características más novedosas de la nueva aplicación es la inclusión de un visualizador de mallas en tiempo real. La antigua aplicación no contaba con un visualizador de

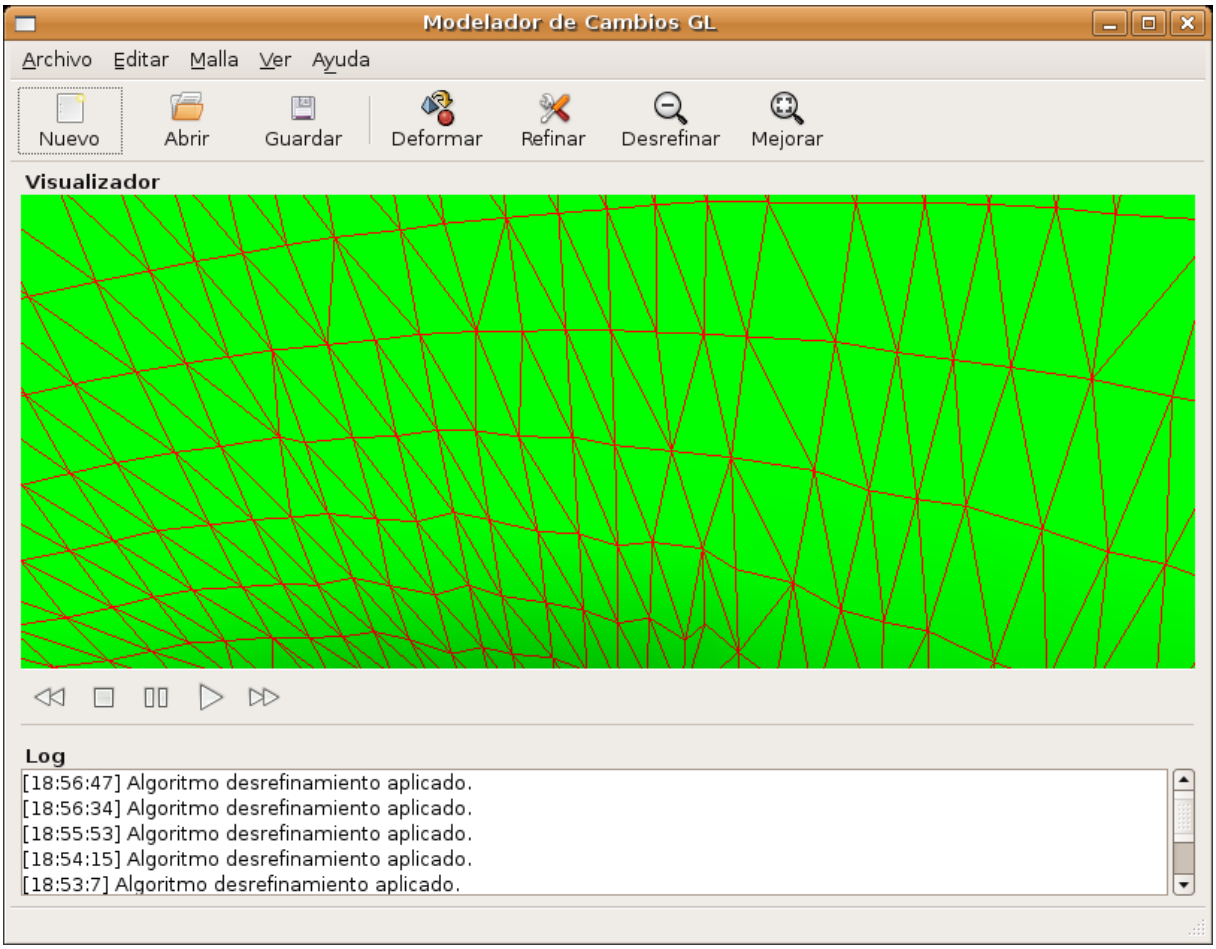


Figura 4.16: Ejemplo de Desrefinamiento luego del Desplazamiento con Verificación

mallas integrado. Esto no permitía notar con facilidad qué cambios estaban ocurriendo en la malla a medida que se aplicaba cada algoritmo. Cada vez que se deseaba visualizar el resultado de la malla se debía recurrir a la aplicación externa Geomview.

En esta Sección especificare los detalles de implementación de el nuevo visualizador de mallas integrado en la aplicación.

4.8.1. Visualización OpenGL en Tiempo Real

El visualizador de mallas fue programado usando las librerías GTKGlExtmm. Estas librerías proveen extensiones a la librería gráfica GTKmm para poder trabajar usando la librería gráfica OpenGL. La librería gráfica OpenGL permite con bastante simplicidad renderizar objetos tridimensionales en pantalla. Además, cuenta con una gran eficiencia lo que la hace una elección clara para el desarrollo de cualquier aplicación que requiera el dibujado de objetos en 3D.

El módulo completo con la implementación del visualizador está encerrado en la clase SimpleGLScene. Con el uso de esta clase cualquier objeto de la aplicación puede configurar que resultados desea visualizar en pantalla. Para manejar la visualización, la clase SimpleGLScene cuenta con los siguientes métodos:

- updateMalla (Malla* malla): Este método muestra en pantalla la malla que se le entregue como argumento. Configura y renderiza tanto la malla solida, como la malla en forma de wireframe[5]. Para la implementación de este método se utilizaron las "Display Lists" que provee el lenguaje OpenGL. Las Display Lists son una forma eficiente de almacenar en memoria mallas que han sido renderizadas con anterioridad. Estas mallas quedan almacenadas bajo un índice. Luego, cuando se desea que ellas sean desplegadas se llama a la Display List bajo su índice, y el malla es visualizada en pantalla. El uso de esta característica permite renderizar de antemano la malla completa y luego poder aplicarle las rotaciones y acercamientos que el usuario desee sin tener que volver a renderizar. Para poder manejar simultáneamente tanto la malla sólida, como la malla wireframe, se indexó la malla actual en la Display List bajo el índice 1 y la malla wireframe bajo el índice -1.

- setMuestraCaras: Este método muestra las caras de la malla. Es decir, una malla sólida. En la Figura 4.17 podemos ver una malla donde solamente se están mostrando sus caras. Este método es simple, ya que solo llama a desplegar en la Display List el índice positivo de la malla deseada, la cual fue renderizada con anterioridad por updateMalla.

- setMuestraArcos: Este método muestra el wireframe de la malla. En la Figura 4.18 podemos ver una malla donde solamente se está mostrando su wireframe. Como en la implementación de updateMalla se almacenaron los wireframes bajo un índice negativo, solo basta saber el índice de la malla que se quiere desplegar, y buscar su número opuesto negativo para visualizarlo.

[5]Wireframe: Malla compuesta solamente por arcos. Se podría decir que es el "esqueleto" de la malla.

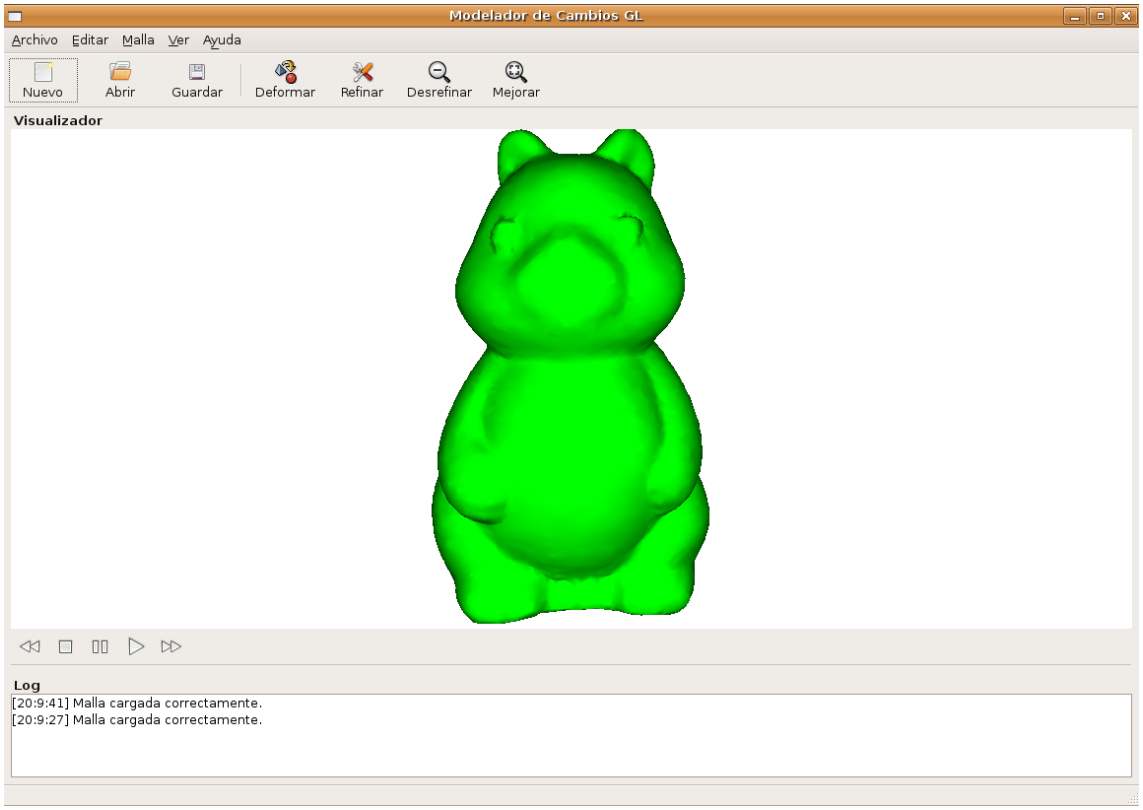


Figura 4.17: Visualizador Mostrando solo Caras de la Malla

- clear: Este método borra todo lo que se este mostrando en pantalla. Para la implementación, simplemente se borra todo el contenido que tenga la Display List.

4.8.2. Rotación y Traslación

Un aspecto importante que debía considerar el visualizador era la posibilidad de rotar, alejar y acercar la malla según los requerimientos del usuario. Para implementar esta funcionalidad se consideró como mejor alternativa el uso de coordenadas esféricas para manejar las posiciones de la malla y de la cámara.

En la Figura 4.19 se puede ver una gráfica indicando la ubicación en el espacio de las coordenadas esféricas respecto a las coordenadas cartesianas. El uso de este tipo de coordenadas facilita bastante la implementación del visualizador ya que podemos dejar al objeto a visualizar centrado en el origen. Luego, la posición de la cámara se puede definir mediante tres parámetros: ángulo θ, ángulo ϕ y radio r. Definir la cámara mediante estos parámetros es tremendamente conveniente ya que son justamente los parámetros que uno intuitivamente desea modificar con el movimiento del mouse. Por ejemplo, al hacer click en el visualizador y arrastrar el mouse horizontalmente, uno estaría modificando el parámetro θ y estaría rotando la cámara alrededor del objeto. Al arrastrar el mouse verticalmente uno estaría modificando el parámetro ϕ y rotando la cámara verticalmente alrededor del objeto. Finalmente, al hacer

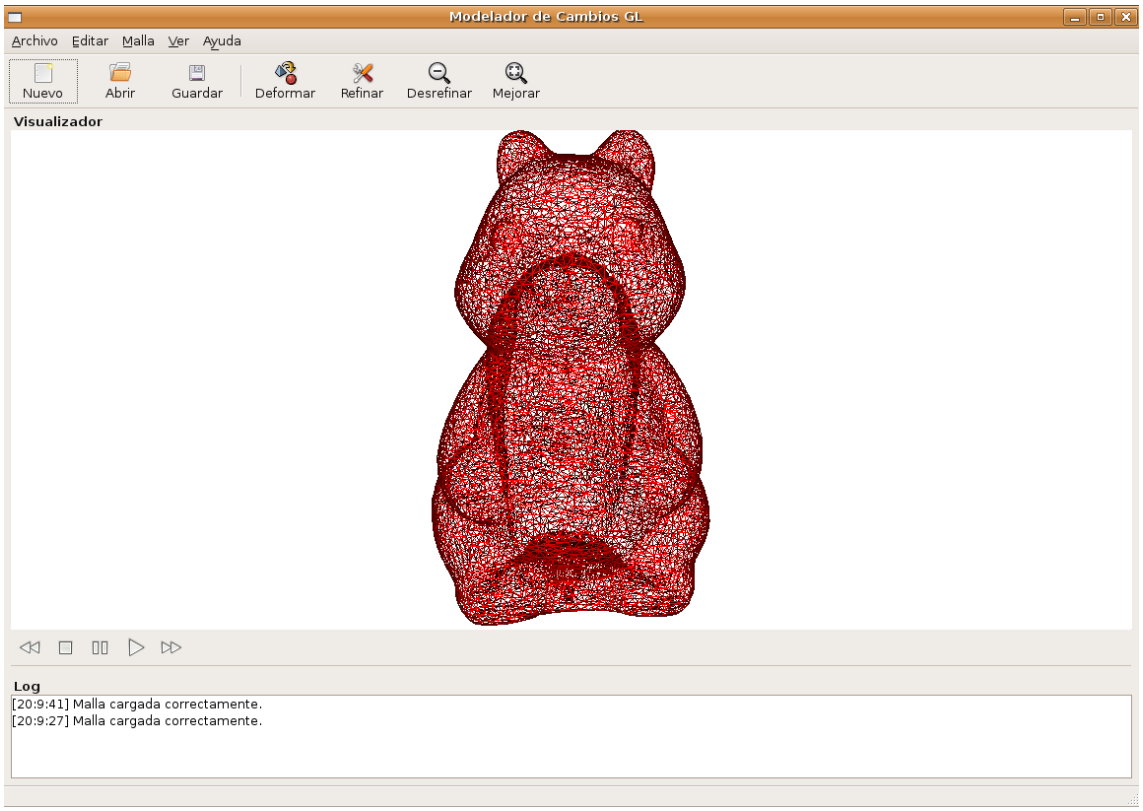


Figura 4.18: Visualizador Mostrando solo Arcos de la Malla

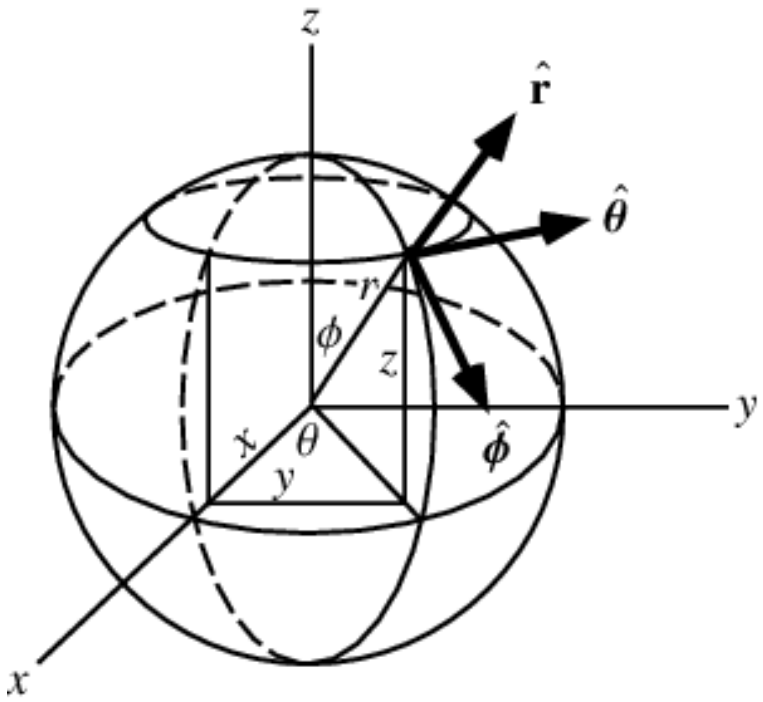


Figura 4.19: Coordenadas Esféricas

click con el botón secundario y mover hacia adelante o atrás el mouse uno estaría modificando el parámetro r y estaría alejando o acercando la cámara al objeto.

Finalmente, para poder visualizar la malla con OpenGL según estas diferentes configuraciones de posición de la cámara se uso el útil método gluLookAt. Este método calcula automáticamente la matriz de transformación de visualización. Para calcularla se le debe dar como parámetro las posiciones de la cámara y el objeto en coordenadas cartesianas. Esto no es difícil ya que el objeto estará siempre centrado en el origen. En el caso de la cámara, convertir su posición a coordenadas cartesianas se hace mediante las siguientes ecuaciones:

$$x = r\cos\theta\sin\phi$$
$$y = r\sin\theta\sin\phi$$
$$z = r\cos\phi$$

4.8.3. Ajuste Automático de la Cámara

La aplicación al trabajar con diferentes tipos de mallas que tienen diferentes escalas se vería enfrentado al problema de que la cámara en su posición inicial no quede ubicada adecuadamente para visualizar la malla. Por ejemplo, si la malla es demasiado pequeña puede que en la posición inicial de la cámara el objeto no se pueda ver, o por el contrario si la malla es muy grande puede que la cámara quede ubicada dentro de la malla misma.

Para solucionar esto se buscó algún método para que la cámara ajuste automáticamente su posición inicial al abrir una nueva malla. La estrategia que se siguió fue la siguiente:

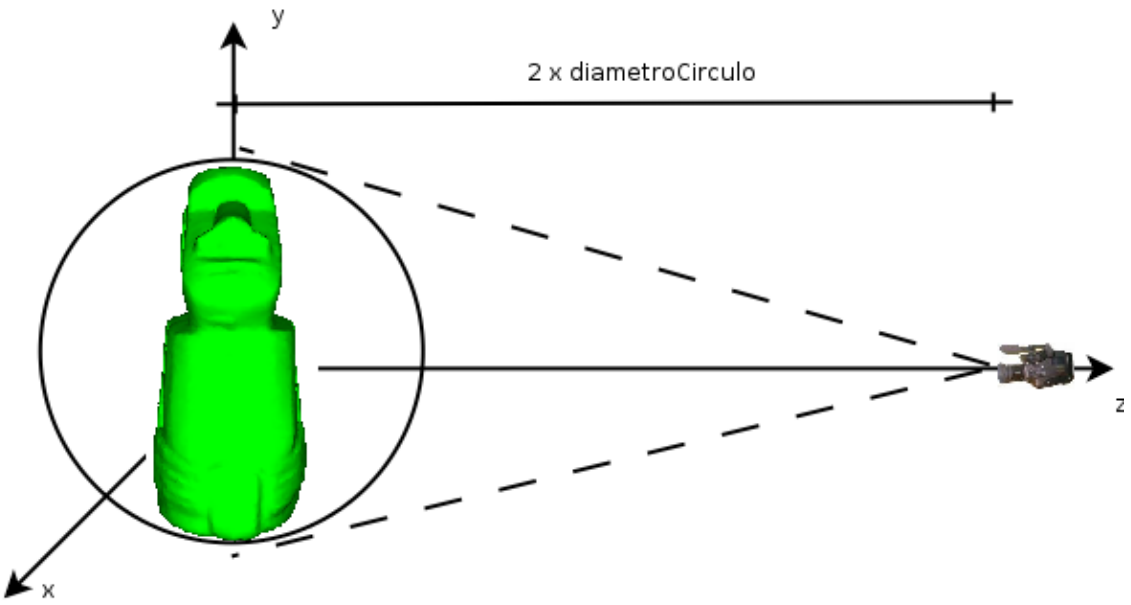


Figura 4.20: Configuración Inicial de la Cámara

1. Se recorre la malla completa para así calcular la esfera que encierre en su interior la malla completa (Bounding Sphere).
2. Calculamos el diámetro de la esfera.
3. Se ubica la posición de la cámara en su parámetro $\theta = 0$, $\phi = \frac{\pi}{2}$ y $r = 2 \cdot diametro$(El multiplicar el diámetro por 2, se coloca la cámara siempre a una distancia en la cual se puede ver el objeto completo de acuerdo al ángulo de visión seleccionado por defecto en OpenGL).

En la Figura 4.20 se puede ver ejemplificado gráficamente la configuración inicial de la cámara al abrir una malla geométrica.

4.8.4. Animación

Una atractiva característica de la nueva aplicación es la posibilidad de ver las deformaciones y desplazamientos de la malla en forma animada. El visualizador cuenta con funciones de reproducción, pausa, detención y velocidad de animación. Mediante estas funciones se pueden lograr resultados notables como ver el crecimiento de tronco de un árbol en forma animada y fluida.

La implementación de esta característica se consiguió mediante el uso de una Display List de OpenGL. Como vimos en la Sección 4.8.1, la malla actual se almacena bajo el índice 1 en su forma solida y -1 en su forma wireframe. Ahora, necesitamos ir almacenando en la Display List cada una de las mallas que componen la secuencia de animación. Cada secuencia de la animación la almacenamos bajo el siguiente esquema: Cada malla sólida se almacena bajo

un entero positivo en la Display List. Su wireframe se almacena bajo el índice opuesto en forma negativa. Por ejemplo, si la forma sólida de la malla se almacena bajo el índice 2, su wireframe se almacenara bajo el índice -2. Para manejar las animaciones se construyó dentro de la misma clase visualizadora SimpleGLScene los siguientes métodos:

- updateAnimacion (Malla* malla, int nframe): En este método se construye en la Display List la malla perteneciente a la animación y se ubica en la posición nframe+1 (Esto debido a que en la posición 1 debe estar solamente la malla actual, no mallas que son parte de la animación).
- startAnimacion(int nmiliseg): Este método configura una señal que se ejecuta cada nmiliseg. Al ejecutarse esta señal se avanza en el Display List una posición en el índice que se está visualizando en pantalla.
- pauseAnimacion: Se detiene la señal, se mantiene la posición en el Display List.
- stopAnimacion: Se detiene la señal y se pone la posición del Display List en 1 para visualizar la malla actual en pantalla.

4.9. Corrección de Errores y Memory Leaks

Una fase importante a la hora de continuar un desarrollo anterior es la de una correcta detección de errores. Este caso no fue la excepción, y se convirtió en una ardua y difícil labor. Las nuevas características que se le incorporaron a la nueva aplicación requirieron de un funcionamiento eficiente y sin errores de los métodos existentes. Mejorar la eficiencia, encontrar errores y corregirlos fue una de las tareas más costosas en tiempo dentro del desarrollo. Por lo tanto, vale la pena revisar que estrategias se utilizaron para solucionar estos problemas. En esta Sección se describirán algunas de las correcciones más importantes efectuadas a la antigua aplicación y los métodos que se utilizaron.

4.9.1. Chequeo de Problemas de Memoria

Para la implementación de las nuevas características de la aplicación se requería solucionar los problemas de memoria que traía la aplicación antigua. La mayor parte de estos problemas eran memory leaks o fugas de memoria. Un memory leak ocurre cuando un bloque de memoria reservada no es liberada. Comúnmente ocurre porque se pierden todas las referencias a esa área de memoria antes de haberse liberado. Esta provocaba problemas al intentar ejecutar métodos más complejos como por ejemplo abrir mallas geométricas de mayor tamaño o efectuar animaciones paso por paso de los desplazamientos en el modelamiento. Cuando se ejecutaban estos métodos la memoria utilizada por la aplicación rápidamente crecía en forma exponencial provocando un cuelgue de la aplicación en unos pocos segundos.

Detectar cuales son los bloques de memoria que no están siendo liberados en una aplicación grande suele ser una tarea bastante tediosa y compleja. Afortunadamente existen herramientas muy útiles para la corrección de este tipo de problemas. Una de las mas populares es

Valgrind. Esta aplicación cuenta con un conjunto de herramientas que ayuda en la depuración de problemas de memoria y rendimiento de programas. Además, la aplicación es totalmente libre y de código abierto por lo cual no hay restricciones para usarla.

A través de Valgrind se debe abrir el archivo ejecutable del programa a depurar, lo que le permite realizar un seguimiento del uso de la memoria y detectar los siguientes problemas:

- Uso de memoria no inicializada.
- Lectura/escritura de memoria que ha sido previamente liberada.
- Lectura/escritura fuera de los límites de bloques de memoria dinámica.
- Fugas de memoria

La utilización de Valgrind mediante la línea de comandos de Linux puede ser un poco difícil de manejar. Para hacer más fácil el uso de esta herramienta es conveniente usar una interfaz gráfica para Valgrind. Una interfaz gráfica recomendada es Alleyoop.

Una vez que se ha lanzado el programa mediante Valgrind y se han ejecutado las operaciones a las cuales se les desea chequear por memory leaks, se debe cerrar la aplicación en ejecución. Luego de ello, Valgrind nos dará un reporte detallado de la cantidad de bytes de memoria que se esta perdiendo, y en que líneas específicamente se esta pidiendo memoria que luego no es liberada. En la Figura 4.21 se puede ver un ejemplo de un reporte de Valgrind.

Usando esta herramienta, se pudo ir resolviendo la gran mayoría de problemas de la aplicación obteniendo finalmente una aplicación robusta y eficiente en el uso de memoria.

4.9.2. Métodos Auxiliares para Chequeo de Consistencia

Una importante ayuda en el desarrollo para la búsqueda de errores (debugging) fue el uso de métodos auxiliares. En esta Sección brevemente describiré los métodos mas importantes:

- Formula de Euler: Este método utiliza la formula de Euler para verificar la topología global de la malla. La fórmula nos dice que una malla de poliedros sin hoyos debe cumplir que $V - E + F = 2$ donde V es el número de nodos, E el numero de arcos, y F el numero de caras.
- Chequeo de punteros: Se implementaron métodos que fueron capaces de analizar la malla completamente y chequear cada uno de sus punteros. Es decir punteros de caras hacia arcos, caras hacia nodos, arcos hacia caras, arcos hacia nodos, nodos hacia caras y nodos hacia arcos. Este método es útil ya que en la gran mayoría de los casos en que se producen inconsistencias se dan situaciones en que elementos de la malla apuntan hacia sectores que han sido eliminados. Por lo tanto, usando este método uno rápidamente puede verificar si un algoritmo esta dejando la malla en un buen estado, o en uno que provocaría un futuro cuelgue en la aplicación.

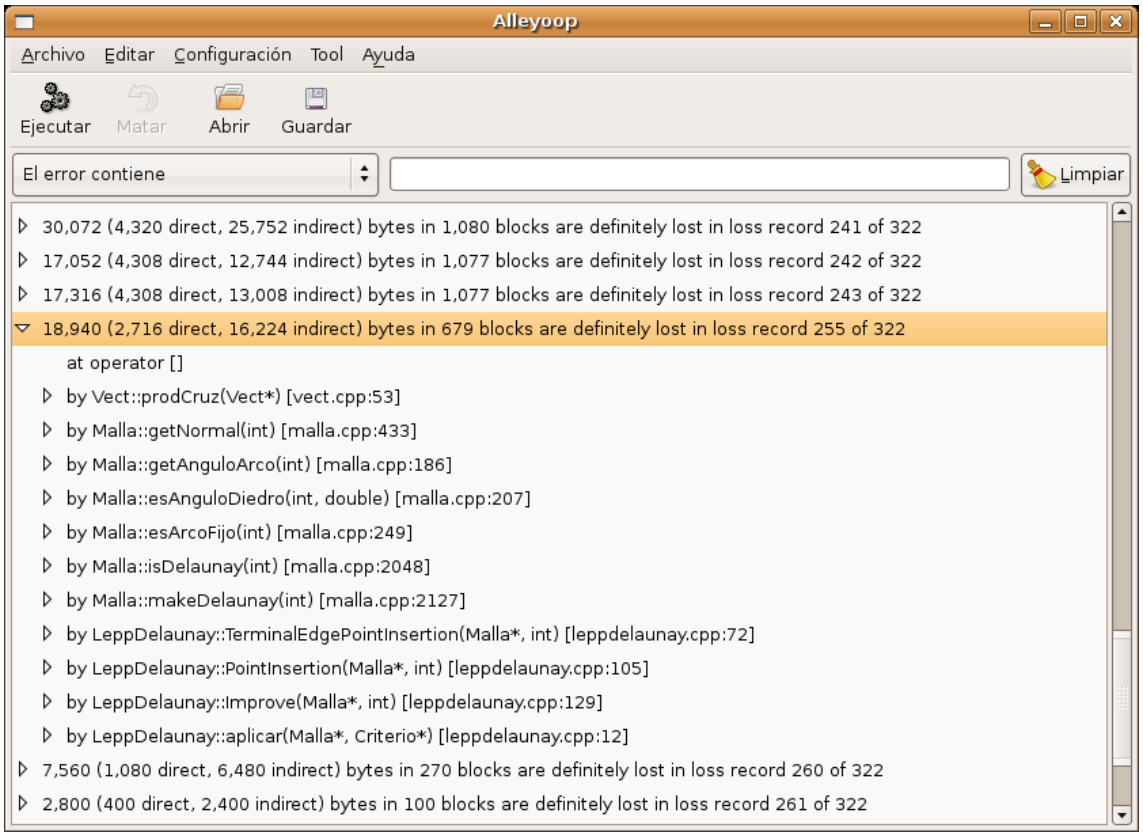


Figura 4.21: Detección de Memory Leaks Usando Valgrind

- Destacar Nodos: Este método es capaz de colorear un nodo en particular de un diferente color al resto en el visualizador de la aplicación. Usando este método uno puede identificar visualmente los nodos que están causando errores en los algoritmos de la aplicación.

Capítulo 5

Discusión y Conclusiones

5.1. Conclusiones

En esta memoria, se abordó el problema de modelar el crecimiento de árboles mediante el uso de mallas geométricas. Para esto, se continuó el desarrollo de la aplicación escrita por R. Medina en [1]. A la aplicación se le incluyeron los siguientes cambios o extensiones:

1. Rediseño de las clases de la aplicación.
2. Mejoramiento de la robustez y desempeño en el uso de memoria de la aplicación.
3. Cambio y mejoramiento de la interfaz gráfica de la aplicación.
4. Incorporación de un visualizador en tiempo real capaz de mostrar las mallas en forma completamente animada.
5. Incorporación de nuevos algoritmos de generación, desrefinamiento, desplazamiento y verificación de las mallas.
6. Corrección de los algoritmos existentes de generación, refinamiento y mejoramiento.

Como conclusión y resultado final de este trabajo, se tiene una aplicación computacional que realiza lo siguiente:

- Genera una malla inicial del objeto a modelar, ya sea desde un archivo OFF de Geomview o desde archivos de texto generados con información de Matlab, Femlab o Comsol. También se puede generar una malla de un cilindro que cumple con ciertos parámetros entregados por el usuario.
- Permite el desplazamiento de los nodos de la malla de acuerdo a un vector dirección que indica el movimiento que tendrá a futuro la célula del árbol representado. Los desplazamientos se pueden efectuar tanto sin verificación como con el uso de distintos algoritmos de chequeos de consistencia.

- Permite hacer operaciones de desrefinamiento de la malla, las cuales son análogas al caso de la naturaleza en donde una célula desaparece por ser comprimida en su tamaño por las células vecinas.
- Permite hacer operaciones de refinamiento las cuales son una buena aproximación a lo que ocurre en la naturaleza con la división celular.
- Se puede mejorar la calidad de las mallas mediante criterios de Delaunay.
- Permite visualizar la malla en forma animada dentro de la aplicación. La malla se puede rotar, acercar en forma simple. Se puede visualizar la secuencia de cambios y desplazamientos que sufrirá el árbol en forma completamente animada.
- Capacidad de mostrar diferentes estadísticas de la malla. Algunas de ellas como la distribución de las áreas de las caras de la malla. tamaño de ángulos, tamaño de arcos, número de nodos y muchos más.
- Permite guardar los datos de la malla en formato OFF, Femlab o Comsol.

El rediseño de la aplicación fue realizado aplicando herramientas UML como diagramas de clases para así obtener un buen diseño orientado a objetos considerando el uso de patrones de diseño en las partes necesarias. El rediseño se concentró principalmente en resolver las falencias que traía la aplicación antigua. Entre estas falencias se resolvió la poca extensibilidad que tenía la aplicación para agregar nuevos algoritmos de almacenamiento, refinamiento y mejoramiento. Otras falencias resueltas fueron el poco encapsulamiento que tenía las clases relacionadas a la interfaz gráfica con el resto de la aplicación. El rediseño de la aplicación también debió considerar un buen diseño de las nuevas características agregadas de tal forma de facilitar la tarea de un futuro desarrollo sobre ésta.

La aplicación fue desarrollada en C++, en conjunto con otras aplicaciones, herramientas y librerías, las cuales fue necesario estudiar. Las librerías utilizadas fueron GTKGLExtmm, GSL y OpenGL. La primera es un potente conjunto de funciones para el desarrollo de interfaces gráficas la cual está desarrollada en C++ y orientada a objetos. GSL es una biblioteca numérica que se utiliza en la aplicación para resolución de ecuaciones polinomiales cúbicas y para el cálculo de estadísticas sobre las mallas. OpenGL es una librería de gráficos en 3D muy poderosa y con un gran rendimiento, siendo considerada el estándar para el desarrollo de cualquier aplicación en 3D sobre Linux.

La visualización de la aplicación tuvo un cambio radical en la nueva aplicación teniendo un visualizador de malla completamente incorporado en la aplicación. Este poderoso visualizador es capaz de mostrar en pantalla a cada momento la forma de la malla justo al instante de haber aplicado algún algoritmo en particular. Además incluye operaciones de rotación y acercamiento. Incluso permite visualizar el crecimiento del árbol modelado en forma animada para ir viendo paso a paso como va creciendo nuestro árbol virtual.

La parte esencial de este trabajo fue dirigido a la implementación de nuevos algoritmos para la aplicación que fuesen más poderosos y tuvieran un funcionamiento más robusto que los antiguos. Dentro de los algoritmos de desplazamiento y deformación de la malla destaca un

nuevo algoritmo que es capaz de detectar y corregir inconsistencias a nivel de sus nodos vecinos. Entre sus ventajas esta el permitir el desplazamiento de la malla mas allá de las primeras inconsistencias, chequeo y corrección en un área mas grande que los algoritmos anteriores y una mayor robustez en general.

Otro importante algoritmo fue el de desrefinamiento el cual es capaz de generar en la malla un proceso análogo al que ocurre en la naturaleza. Este proceso es la desaparición de células de la corteza del árbol en el momento en que su área comienza a disminuir reiterativamente.

Se ha agregado una importante característica a la aplicación como son la visualización de estadísticas de la malla. Esto nos puede mostrar directamente algunos datos importantes sobre la malla como son la calidad de sus ángulos, la distribución de sus áreas y el largo de sus arcos. Usando estas estadísticas el usuario puede entender mejor el estado de la malla para así poder aplicar los algoritmos que sean necesarios en cada estado de la malla.

La aplicación ha mejorado bastante en robustez ya que una gran cantidad de errores tanto geométricos, como de implementación e ineficiencia en el uso de memoria han sido corregidos. Al mismo tiempo se han incorporado funciones que permiten chequear que no haya problemas de inconsistencia que pudieran hacer colgarse a la aplicación. Estos métodos son extremadamente útiles para hacer un buen chequeo de errores en el código.

Finalmente, por lo detallado, la aplicación construida puede ser utilizada como una poderosa herramienta para el modelamiento de crecimiento de arboles.

5.2. Trabajo Futuro

Existen varias direcciones en donde este trabajo puede seguir desarrollándose, tanto en nuevas estrategias para los distintos procesos que realiza la aplicación, como en nuevas funcionalidades. Entre los problemas que se pueden abordar se encuentran:

- Simplificación de la estructura malla. Al observar detenidamente las clases de la aplicación se puede notar que la clase malla tiene una gran cantidad de métodos. Esto provoca que sea muy difícil de mantener y extender. Muchos de los métodos de Malla podrían pasar a formar parte de otras clases que tienen mayor relación con el uso del método. Por ejemplo, métodos como getLargoArco o getAreaCara deberían ser parte de la clase Arco o Cara respectivamente. Actualmente métodos como este se manejan dentro de la clase malla ya que esta clase es la única que tiene punteros hacia los diferentes elementos de la malla. Por ejemplo malla tiene punteros tanto a una cara como a los nodos que la forman. Para calcular el área accede directamente a cada uno de los nodos que la forman para obtener sus coordenadas y hacer el calculo. Dentro de un trabajo futuro podría estar el añadir este tipo de métodos fuera de la clase malla para así tener un diseño mas simple y fácil de mantener en la aplicación.
- Limitar el uso de memoria dinámica en operaciones sobre vectores. Actualmente muchas de las operaciones que se hacen sobre vectores como la suma, resta, multiplicación,

producto cruz, etc., piden memoria dinámica para retornar el resultado. Esto da como resultado que sea haga muy difícil la implementación de otros métodos ya que hay que estar constantemente preocupándose de liberar la memoria que se pidió dinámicamente cada vez que se ejecuta una operación. El hecho de reimplementar estas operaciones y los métodos que las utilizan para utilizar memoria estática ayudarían bastante a evitar memory leaks en el desarrollo a futuro.

- Creación de algoritmos que verifiquen la consistencia global de la malla dentro de las deformaciones: Como se describió, el algoritmo implementado en esta aplicación se basa en verificaciones sobre la vecindad de elementos sobre cada nodo. Por lo tanto, no se verifica que puedan haber inconsistencias mas allá de los elementos que estén en la vecindad. El trabajo a futuro puede estar dirigido a considerar este tipo de inconsistencias donde el problema principal se puede dar por la eficiencia del algoritmo. Para mejorar esto podría ser útil usar estructuras como Octrees que discretizan el espacio en donde se chequean intersecciones.

- Mallas obtenidas a partir de imágenes, ya sea en dos o tres dimensiones. Este es un campo complejo pero muy estudiado en la actualidad, especialmente en aplicaciones medicas, en donde se pueden extraer superficies de objetos mediante la generación aproximada de una malla inicial y la posterior deformación de la malla.

- Evitar problemas de precisión. Muchos de los problemas de robustez de la aplicación se producen por problemas de precisión. Cuando se realizan deformaciones en las mallas, se puede llegar a un estado donde las caras son muy pequeñas y en ese momento los cálculos arrojan resultados erróneos para la posterior caída de la aplicación. Un trabajo a futuro podría revisar esta problemática y buscar formas de efectuar algunos cálculos con mayor precisión, quizás con el uso de bibliotecas especiales que manejen matemática exacta.

- Detección de caídas o ciclos infinitos en la aplicación. Varias operaciones que el usuario puede solicitar pueden ser imposibles de realizar. Por ejemplo se puede desear refinar los ángulos de las caras menor a un cierto valor el cual jamas puede ser alcanzado. Una importante característica a desarrollar sería que los algoritmos sean capaces de detectar que una operación no se va a poder realizar y no se quede encerrada la aplicación en un ciclo infinito.

- Mallas de cuadriláteros. La aplicación puede generar este tipo de mallas desde los formatos definidos, pero las operaciones sobre la topología de la malla se definieron solo para mallas de triángulos. Por esto, un trabajo a futuro sería la actualización de los algoritmos para que funciones sobre mallas de cuadriláteros, las cuales podrían ser bastante útiles para representar una célula de forma mas natural.

Bibliografía

[1] Medina RA. Modelador de Cambios en la Geometría de Objetos Utilizando Mallas Geométricas, Memoria para optar al título de Ingenierio Civil en Computación, 2005.

[2] Bastarrica MC, Hitschfeld-Kahler N. Designing a Product Family of Meshing Tools, Advances in Engineering Software, 37(2006)1-10.

[3] Gamma E, Helm R, Johnson R, Vlissides J. Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995.

[4] Prusinkiewicz P. Simulation Modeling of Plants ans Plant Ecosystems, Communication of the ACM, Vol 43, N°7, Julio 2000.

[5] Lintermann B, Deussen O. Interactive Modeling of Plants, IEEE Computer Graphics and Application, Vol 19, Issue 1, Enero 1999.

[6] O'Rourke J. Computational Geometry in C, Cambridge University Press, 2º Edicion, 1998.

[7] Weber J, Penn J. Creation and Rendering of Realistic Trees, Proceedings of the 22º Annual Conference on Computer Graphics and Interactive Technique, Pages 119-128, 1995.

[8] Kobbelt L, Bareuther T, Seidel HP. Multiresolution Shape Deformations for Meshes with Dynamic Vertex Connectivity, Computer Graphics Forum, Vol 19, Issue 3, Agosto 2000.

[9] Shalloway A, Trott JR. Design Patterns Explained: A New Perspective On Object-Oriented Design, Addison-Wesley, 2000.

[10] Phillips CL. The Level-Set Method. MIT Undergraduate Journal of Mathematics.

[11] Xian-Yang L, Shang-Hua T, Ungor A. Simultaneous Refinenment and Coarsening: Adaptive Meshing with Moving Boundaries, 1998.

[12] Liu A, Horlacher M. Tracking the Front of Moving Boundaries Using Grid Generation Techniques.

[13] Hitschfeld-Kahler N. Apuntes Curso CC60Q - Seminario de Geometría Computacional. Departamento de Ciencias de la Computacion. Universidad de Chile. URL: http://www.dcc.uchile.cl/~cc60q

[14] Kraus M, Ertl T. Simplification of Nonconvex Tetrahedral Meshes. Visualization and Interactive Systems Group. Universitat Stuttgart.

[15] Rivara MC, Hitschfeld-Kahler N. LEPP-Delaunay Algorithm. A Robust Tool For Producing Size-Optimal Quality Triangulations. Departamento de Ciencias de la Computacion. Universidad de Chile.

[16] Sitio Oficial de GTK+. URL: http://www.gtk.org

[17] Sitio Oficial de GTKmm. URL: http://www.gtkmm.org

[18] Sitio Oficial de GTKGlExtmm. URL: http://www.k-3d.org/gtkglext/Main_Page

[19] Sitio Oficial de GSL. URL: http://www.gnu.org/software/gsl

[20] Sitio Oficial de GNOME Human Interface Guidelines. URL: http://developer.gnome.org/projects/gup/hig/

[21] Villablanca L. Mesh Generation Algorithms for Three-Dimensional Semiconductor Process Simulation. 2000.

[22] Forest L, Demongeot J. Cellular Modelling of Secondary Radial Growth in Conifer Trees: Application to Pinus radiata. Bulletin of Mathematical Biology. 2006.

Apéndice A

Diagrama de Clases de Patrones de Diseño

En las Figuras A.1 y A.2 se muestran los diagramas de clases de los patrones de diseño usados en la aplicación Command y Strategy.

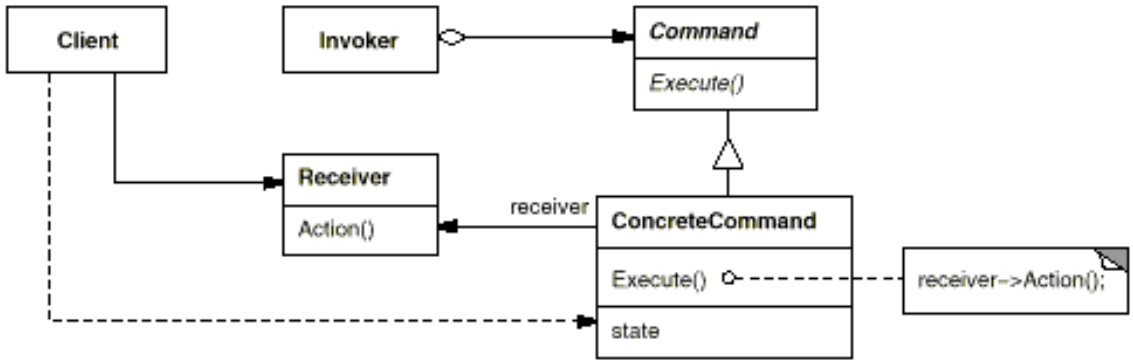


Figura A.1: Diagrama de Clases de Patrón de Diseño Command

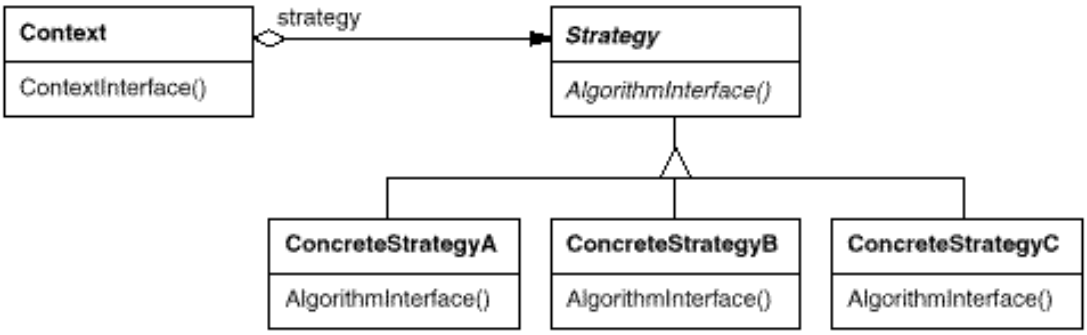


Figura A.2: Diagrama de Clases de Patrón de Diseño Strategy

Apéndice B

Pseudocodigo Algoritmo de Lectura desde Comsol

El pseudocódigo del algoritmo se puede ver en el Algoritmo 7 en la página siguiente

Algorithm 7 GeneraFromComsol

```
GeneraFromComsol (string archivo) {
//Generación de los nodos de la malla.
numero_puntos=0;
Nodos nodos;
while(true){
    LeerLinea(archivo);
    if (esLineaSeparadora)
        break;
    else{
        Se obtiene posicion x,y,z de la linea;
        nodo=new Nodo(new Punto(x,y,z));
        nodos->agregar(nodo);
        numeroPuntos++;
        }
    }
//Generacion de los arcos y caras de la malla.
Arcos arcos; Caras caras;
vector<int> indice_nodos, indice_arcos;
Map<Pair<int,int>,int> pares_de_arcos;
for (i=0; i<numero de caras de la malla; i++){
    LeerLinea();
    //Se agregan los indices de nodos a la lista de nodos de la cara.
    for (j=0; j<3; j++)
        indice_nodos->agregar(getIndiceNodo());
    //Se debe recorrer cada par de nodos consecutivos.
    for (j=0; j<3; j++){
        indice_nodo1=indice_nodos[j];
        indice_nodo2=indice_nodos[(j+1)%3];
        Pair<int,int> par1(indice_nodo1, indice_nodo2);
        Pair<int,int> par2(indice_nodo2, indice_nodo1);
        if(ya se encuentra par1 entre pares_de_arcos)
              indice_arcos->Agregar(el indice donde esta par1);
        else if(ya se encuentra par2 entre pares_de_arcos)
              indice_arcos->Agregar(el indice donde esta par2);
        else { //Hay que crear el nuevo arco
               indice_arco=arcos->AgregarArco(new Arco (indice_nodo1, indice_nodo2));
               indice_arcos->Agregar(indice_arco);
               pares_de_arcos->Insertar(Par(indice_nodo_1, indice_nodo_2));
               nodos->AgregarArcoNodo(indice_nodo1, indice_arco);
               nodos->AgregarArcoNodo(indice_nodo2, indice_arco);
}}}}
```