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

MODELAMIENTO DE UN GENERADOR DE MALLAS BASADO EN OCTREES, USANDO PATRONES DE DISEÑO

MEMORIA PARA OPTAR AL TÍTULO DE INGENIERO CIVIL EN COMPUTACIÓN

NICOLÁS IGNACIO ARANCIBIA ROMÁN

PROFESORA GUÍA:
NANCY HITSCHFELD KAHLER

MIEMBROS DE LA COMISIÓN:
BENJAMIN BUSTOS CARDENAS
MARIA BASTARRICA PIÑEYRO

SANTIAGO DE CHILE
OCTUBRE 2007

RESUMEN DE LA MEMORIA PARA OPTAR AL
TÍTULO DE INGENIERO CIVIL EN COMPUTACIÓN
POR : NICOLÁS ARANCIBIA ROMÁN
FECHA : 22/10/2007
PROF. GUÍA : NANCY HITSCHFELD KAHLER

"MODELAMIENTO DE UN GENERADOR DE MALLAS BASADO EN OCTREES, USANDO PATRONES DE DISEÑO"

El objetivo del presente trabajo de titulación es mejorar el diseño, robustez, extensibilidad y aplicabilidad de un generador de mallas basado en octrees. Para ello se ha propuesto un nuevo diseño, el cual es orientado a objetos, usando patrones de diseño. Dentro del diseño, el cual incluye varios algoritmos que representan las etapas para generar una malla (generación de una malla inicial, refinamiento y mejoramiento de la malla según requerimientos del usuario, y finalmente, generación de la malla final), se ha implementado uno de partición de elementos 1-irregular, que junto a otro anterior que no abarcaba todas las configuraciones, permite entregar una malla particionada en su totalidad sin elementos 1-irregular en su interior. Un elemento 1-irregular es aquel que contiene a lo más un punto adicional en cada una de sus aristas.

En una primera etapa se ha estudiado el diseño anterior, posteriormente se ha modelado el generador de mallas con un enfoque orientado a objetos entregando un diagrama de clases que permitirá en el futuro modificar el generador anterior y llevarlo a un rumbo más actual en relación a las buenas prácticas de ingeniería de software. Dado que existe una gran cantidad de algoritmos para distintos propósitos, se ha seguido la filosofía del patrón de diseño Strategy, el cual permite definir una familia de algoritmos, encapsular uno de ellos y hacerlos intercambiables.

A continuación se abordó el tema del particionamiento de elementos 1-irregular, en particular, el cuboide 1-irregular. El algoritmo diseñado e implementado pertenece a la familia de algoritmos de generación de una malla final, y se suma a otro que entrega particiones de elementos mixtos para algunas configuraciones 1-irregular. El algoritmo en cuestión, recibe un cuboide 1-irregular, genera una malla inicial, la particiona mediante bisección de tetraedros, y posteriormente mejora su calidad aplicando la condición de Delaunay en 3D mediante un algoritmo que recorre las caras interiores y realiza transformaciones locales cuando un tetraedro no satisface la condición de Delaunay. La idea de agregar este nuevo algoritmo es resolver todas las configuraciones irregulares posibles para un cuboide, entre las que se incluyen aristas y caras con un punto de Steiner en su mitad.

Finalmente se ha revisado la robustez del particionamiento Delaunay, y se han probado todas las configuraciones posibles para cuboides 1-irregulares. La implementación se llevó a cabo en lenguaje C++, y la visualización de los resultados se realizó mediante el programa GeomView en Linux usando el formato vectorial. Como resultado se tiene un diseño extensible para el generador de mallas basado en octrees y un algoritmo robusto de particionamiento final que resuelve el problema completamente.

A mis padres

Agradecimientos

Al llegar al final de mi carrera, y repasando lo que ha sido mi extensa vida universitaria, me llegan a la memoria momentos, personas y situaciones, que han permitido que esta meta pueda ser cumplida.

En primer lugar agradezco a mis padres. A mi mamá, por estar siempre presente, por creer en mi, por su amor, preocupación y tremendo esfuerzo que hizo por ayudarme y lograr tener un hijo profesional. A mi papá, por su cariño constante, por estar siempre presente y con su esfuerzo, cumplirme en lo que pudiera darme. También agradezco a mi hermana por sus consejos y cariño, y por último a la nueva integrante de mi familia, mi sobrinita Sofía, quien con sólo su presencia me motivaba para seguir adelante en esto.

También quiero agradecer a las personas que conocí en Santiago y con las que compartí muchos momentos. Primero, a las grandes amistades que forjé en el Hogar Universitario Paulina Starr, donde he vivido desde que estoy en Santiago, Carlos Loyola, Iván Morales, Sergio Aguayo y Marianela Rojas. También agradezco a los miembros de la "comunidad IAE", que gracias a ellos, y a veces en conjunto, se pudo sobrellevar, renacer y vencer las batallas que se venían en cada semestre, finalizando todos victoriosos.

Agradecimiento especial tiene mi profesora guía, Nancy Hitschfeld, primero por su confianza, y segundo por su paciencia, preocupación y ayuda cuando lo he necesitado. Muchas gracias.

Otro agradecimiento importante, es para la dirección de bienestar de la Universidad y de la Facultad, por su ayuda en distintos beneficios, los que sin duda me permitieron mantenerme tranquilo durante mis estudios, ayuda que si no hubiera existido, me habría dificultado bastante la oportunidad de ser profesional, aun teniendo capacidad. Muchas gracias.

Finalmente, me gustaría agradecer la ayuda recibida en el contexto del proyecto Fondecyt No. 1061227, por el apoyo financiero en este trabajo.

Contenido

Capítulo 1

Introducción

1.1 Aspectos Generales

Para el estudio, simulación y resolución de problemas reales es necesario discretizar el dominio de manera adecuada para obtener soluciones confiables y en un tiempo razonable. La discretización del dominio (malla) es una tarea fundamental y ésta se realiza con herramientas conocidas como generadores de mallas. Esta discretización se aplica para representar objetos que en realidad son continuos y así poder simular numéricamente su comportamiento. Estas mallas (que se crean uniendo un conjunto de puntos), pueden ser usadas en diversas áreas de la ingeniería, como por ejemplo en el estudio del comportamiento de estructuras sólidas de formas complejas, tales como los pequeños dispositivos semiconductores hasta incluso el diseño de naves espaciales. El comportamiento de estos sólidos generalmente es modelado utilizando ecuaciones diferenciales parciales, cuya solución numérica implica contar con una representación espacial discreta del sólido, que consiste en una malla de poliedros. La precisión de la malla, depende de los requisitos impuestos por el problema, por ejemplo se espera un mejor refinamiento en las áreas de mayor interés para el estudio.

Dentro del área de la geometría computacional, los algoritmos de generación de mallas son materia de estudio e investigación desde que las aplicaciones motivaron su desarrollo. Con la aparición de nuevas tecnologías cada vez más poderosas, la demanda por generadores de mallas que puedan generar millones de elementos en un tiempo razonable ha crecido en los últimos años. Por lo tanto, se desea disponer de generadores de mallas eficientes, robustos y fáciles de modificar y extender.

Existen distintos tipos de mallas tridimensionales (3D). Las más comunes son las que se componen de hexaedros y de tetraedros. La elección del tipo de elemento y de los algoritmos a usar depende de las restricciones del problema y de los criterios de calidad impuestos sobre la discretización a generar.

Las mallas más comunes son las triangulaciones. Estas triangulaciones se pueden aplicar tanto en 2D como en 3D. Para obtener una malla de mayor calidad que asegure una convergencia en los métodos numéricos, se necesitan triángulos (o tetraedros en 3D) razonablemente buenos. Esto significa que los elementos sean lo más equiláteros posibles. Un criterio que asegura mejorar la calidad de las triangulaciones, y uno de los más utilizados, es el de Delaunay. Este criterio establece que la circunferencia circunscrita a un triángulo (en 3D es la esfera circunscrita a un tetraedro) no debe contener ningún otro punto en su interior.

Otro tipo de generador de mallas es el basado en octrees. Éste se crea encerrando el objeto de estudio en un cubo y subdividiéndolo recursivamente en octantes. Aunque este método da buenos resultados, se revelan algunas desventajas cuando se enfrenta a geometrías complejas. Estas desventajas se refieren a la densidad de puntos, la cual es la misma en las tres dimensiones, aunque sólo interese refinar en una, otro caso son los bordes del objeto, que no siempre coincidirán con los puntos medios de un octante, por lo tanto se realizan refinaciones muy elaboradas, y finalmente se pueden cometer imprecisiones al calcular el volumen de los elementos. Estas desventajas se pueden evitar agregando otros poliedros a la malla, llamados *macroelementos*, los cuales se pueden también subdividir en otros con el objeto de refinar la malla. En esta memoria se presenta un diseño orientado a objetos para este tipo de mallas, llamadas mallas mixtas, y además se implementa un algoritmo de malla final Delaunay que convierte un cuboide o paralelepípedo regular en tetraedros y así evitar algunas de las desventajas nombradas anteriormente.

1.2 Justificación y Motivación

El análisis, diseño y desarrollo de un generador de mallas en tres dimensiones es una tarea bastante compleja y requiere de un acabado estudio y cuidadosas decisiones durante su realización. En el Departamento de Ciencias de la Computación existe un generador de mallas mixtos conocido como *met* (Mixed Elemente Tree) basado en octrees, y se requiere modificarlo y extenderlo con el objetivo de mejorar su rendimiento y diseño para adaptarlo fácilmente a distintas aplicaciones. Su implementación no usa patrones de diseño ni buenas prácticas recomendadas por las métricas más usadas en ingeniería de software.

La motivación principal para desarrollar este tema de memoria es usar buenas prácticas de ingeniería de software para mejorar la aplicación antes descrita. Además es un desafío personal el enfrentar un problema complejo, desarrollado en C++, que tiene variadas aplicaciones en distintas ramas de la tecnología y ciencias y que requiere un cuidadoso estudio.

1.3 Objetivos de la memoria

1.3.1 Objetivos General

El propósito de este trabajo de titulación es mejorar el diseño, robustez, extensibilidad y aplicabilidad de un generador de mallas basado en octrees.

1.3.2 Objetivos Específicos

A continuación se detallan los objetivos específicos en este trabajo de titulación:

1. Rediseño de la aplicación. Diseñar una solución extensible, mantenible y de fácil configuración en los siguientes aspectos:
    - Formatos de entrada/salida.
    - Algoritmos para generar la malla inicial.
    - Algoritmos de refinamiento.
    - Algoritmos de partición.

2. Búsqueda e integración de un visualizador opensource asociado a la visualización de los nuevos algoritmos de particionamiento.

3. Incorporación de nuevos algoritmos para la partición de elementos 1-irregular para algunas configuraciones de puntos no resueltas en la implementación actual del *met*.

4. Comparación del algoritmo actual con el original para particiones 1-irregular.

1.4 Contenido de la memoria

1. Introducción: Se da una mirada general, mostrando la justificación, motivación y objetivos de esta memoria.

2. Antecedentes: Se muestran los antecedentes previos relacionados al área en el cual está inserta esta memoria.

3. Diseño OOP del Generador de Mallas: Se describe el generador de mallas *met*, se muestra el análisis el diseño anterior y se presenta un diseño orientado a objetos.

4. Diseño e Implementación de un Algoritmo de Malla Final Delaunay: Se describe un algoritmo anterior para el particionamiento de un cuboide 1-irregular y se presenta el diseño y la implementación de un nuevo algoritmo que cumple con la condición de Delaunay y cuyo objetivo también es particionar un cuboide 1-irregular.

5. Resultados: Se muestran los resultados obtenidos en las distintas pruebas a la que fueron sometidos los algoritmos para las distintas etapas de la implementación de esta memoria.

6. Conclusiones: Se muestran las conclusiones obtenidos del desarrollo de este trabajo.

7. Bibliografía: Se muestran los trabajos anteriores relacionados con este trabajo y cuya información fue importante en este desarrollo.

8. Anexos: Se presenta información adicional.

Capítulo 2

Antecedentes

En este capítulo se mostrarán los antecedentes previos relacionados al área en la cual está inserta esta memoria.

2.1 Conceptos Geométricos

2.1.1 Geometría Computacional

La geometría computacional estudia problemas geométricos desde el punto de vista de la computación y se ocupa del diseño y análisis de este tipo de algoritmos. Las aplicaciones de esta área de la computación no sólo van dirigidas a representar objetos reales a través del computador, o a generar imágenes de objetos reales o imaginarios usando modelos computacionales, sino además, a ser una herramienta de apoyo para analizar, diseñar y simular problemas reales. Entre algunos aplicaciones de la vida real podemos encontrar: robótica, reconocimiento de voz y de patrones, diseño gráfico, sistemas de información geográfica, biología, localización, estudio del comportamientos de sólidos como semiconductores, automóviles, y en general, cual fenómeno físico.

La simulación de fenómenos reales se puede analizar en 2 o 3 dimensiones. El problema en 2D es normalmente una simplificación del problema en 3D y actualmente se considera resuelto. El trabajo en 3D es bastante más complejo, y es por ello que la investigación en esta área apunta más a este tipo de problemas. El tema de memoria que se presenta en este informe se orienta al problema en 3D.

2.1.2 Mallas Geométricas

Una malla es un conjunto de celdas contiguas que permite representar en forma discreta el dominio de un problema a resolver numéricamente. La confiabilidad de la solución numérica obtenida depende de la calidad de la malla generada. Los criterios de calidad a usar dependen del método numérico escogido y del tipo de problema a resolver.

Las mallas geométricas se pueden clasificar en estructuradas y no estructuradas. Las mallas estructuradas se caracterizan por estar compuestas de celdas de un tamaño similar y del mismo tipo, por ejemplo, triángulos o rectángulos, en dos dimensiones, tetraedros o hexaedros en tres dimensiones. Estas mallas son fáciles de generar pero no permiten modelar eficientemente problemas geométricamente complejos. Por otro lado, las mallas no estructuradas permiten el uso de celdas de distinto tipo y/o de diferente tamaño. Estas mallas requieren de algoritmos y estructuras de datos más complejos que las anteriores, pero permiten modelar geometrías complejas y optimizar el número de celdas usadas de acuerdo a la necesidad de la aplicación.

2.1.3 Elementos en mallas geométricas en 3D

Algunos elementos usados en la literatura se pueden ver en la figura 2.1:

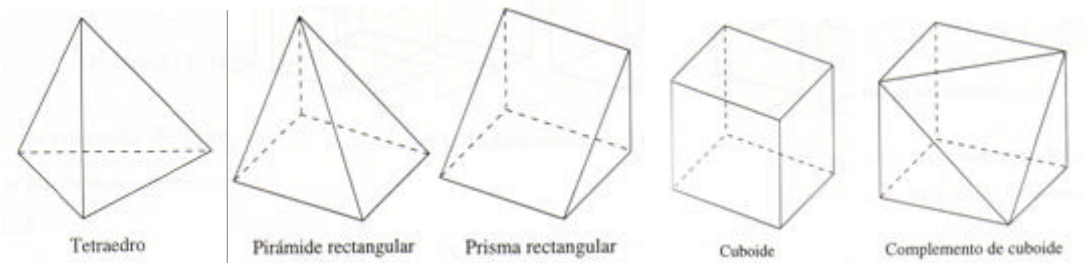


Figura 2.1: Elementos usados en la literatura.

2.1.4 Octree

Un octree es una estructura de datos que permite almacenar información en un árbol. Cada nodo interno tiene ocho hijos. Los octrees son usados frecuentemente para particionar espacios tridimensionales y en forma recursiva al subdividir el espacio en ocho octantes. Algunos usos de los octrees son: indexación espacial, detección de colisiones en tres dimensiones y generación de mallas. En el caso de generación de mallas, las hojas del árbol son cuboides y éstos, en su conjunto, forman la malla.

En la figura 2.2 se pueden apreciar los distintos niveles para un Octree. A la izquierda se ve el elemento y a la derecha su representación, la cuál es en forma de árbol. En el nivel 0 (raíz) se ve sólo un nodo. En el nivel 1, se ve el nodo raíz mas sus ocho hijos, cada hijo representa un octree en forma recursiva. El nivel 2 contiene el nodo raíz más su ocho hijos y además uno de esos hijos se ve representado como otro octree.

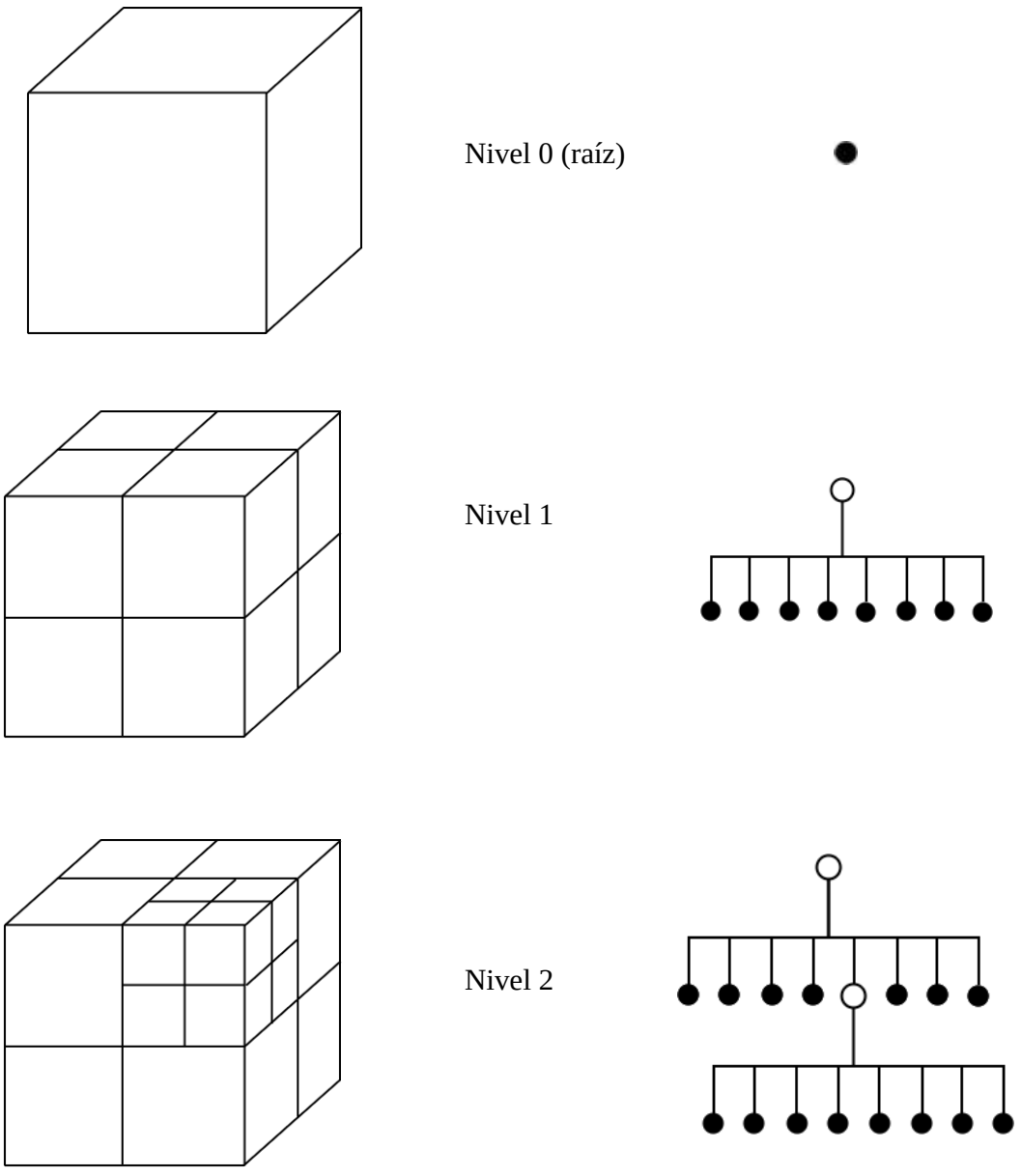
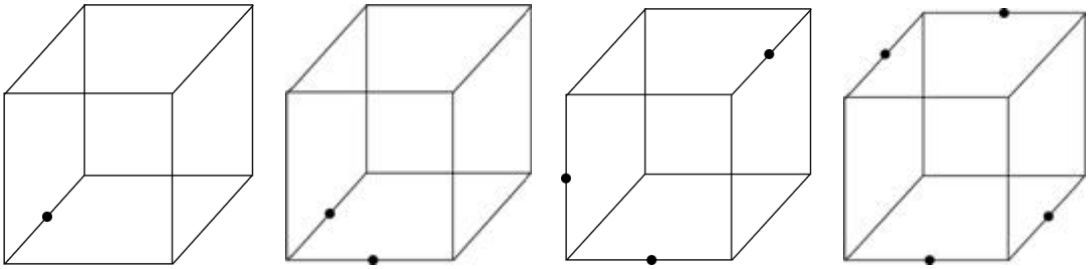


Figura 2.2: Octree en 3 niveles

2.1.5 Elemento 1-irregular

Un elemento es 1-irregular si tiene aristas que contienen a lo más un punto extra (punto de Steiner) en sus arcos. Elementos con puntos de Steiner en sus arcos aparecen después de dividir sus vecinos en elementos más pequeños. Los elementos se hacen 1-irregular insertando puntos aislados en arcos apropiados o dividiendo (refinando) elementos en elementos más pequeños.

La figura 2.3 muestra varios cuboides 1-irregular.



(a) (b) (c) (d)

Figura 2.3: Diferentes configuraciones para cuboides 1-irregular.

2.1.6. Triangulación

Una triangulación es una malla formada por triángulos en 2D y por tetraedros en 3D. La figura 2.4 muestra una triangulación en 2D, mientras que la figura 2.5 muestra una malla de tetraedros válida en 3D.

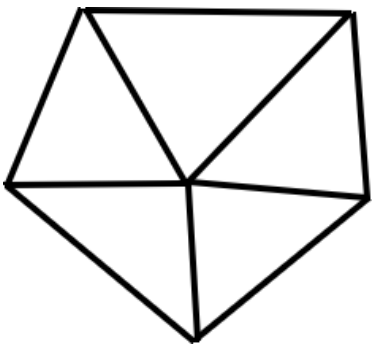


Figura 2.4: Triangulación en 2D.

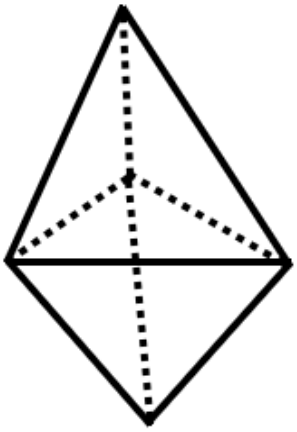


Figura 2.5: Malla de tetraedros.

2.1.7. Test del Círculo

El test del círculo se usa para verificar si un triángulo satisface la condición de Delaunay.

Dados un punto *p* y un triángulo *t* de vértices (*v1*,*v2*,*v3*), sea *C* el círculo que corresponde al círculo circunscrito a *t*. Entonces el test del círculo se usa para calcular la posición del punto *p* con respecto al círculo *C*, de la siguiente forma:

- El Test del círculo retorna *verdadero* si *p* está fuera de *C* (Ver figura 2.6 izquierda). En este caso *t* satisface la condición de Delaunay.
- El Test del círculo retorna *falso* si *p* está dentro de *C* (Ver figura 2.6 derecha). En este caso *t* no satisface la condición de Delaunay.

El test del círculo se utiliza para construir y validar triangulaciones Delaunay en 2D.

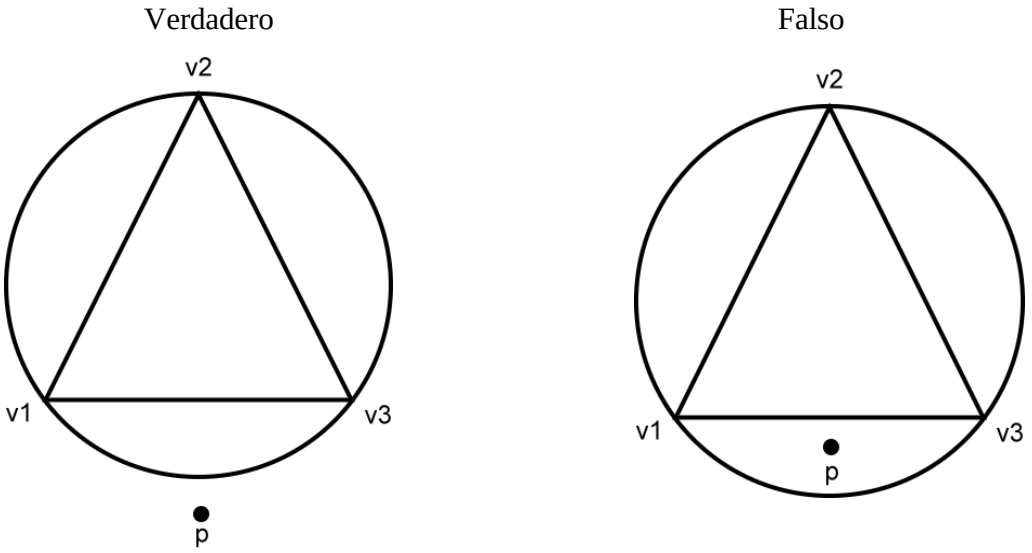


Figura 2.6: Test del Círculo

2.1.8. Test de la Esfera

De manera similar, un tetraedro satisface la condición de Delaunay si su circunsfera no contiene otro punto de la malla en su interior.

Dado un punto *p* y un tetraedro *t* cuyos vértices son *(v1,v2,v3,v4)*, donde *E* corresponde a la esfera circunscrita a *t*, entonces el test de la esfera estudia la posición del punto *p* con respecto a la esfera de la siguiente forma:

- El Test de la esfera retorna *verdadero* si *p* está fuera de *E* (Ver figura 2.7 izquierda).
- El Test de la esfera retorna *falso* si *p* está dentro de *E* (Ver figura 2.7 derecha).

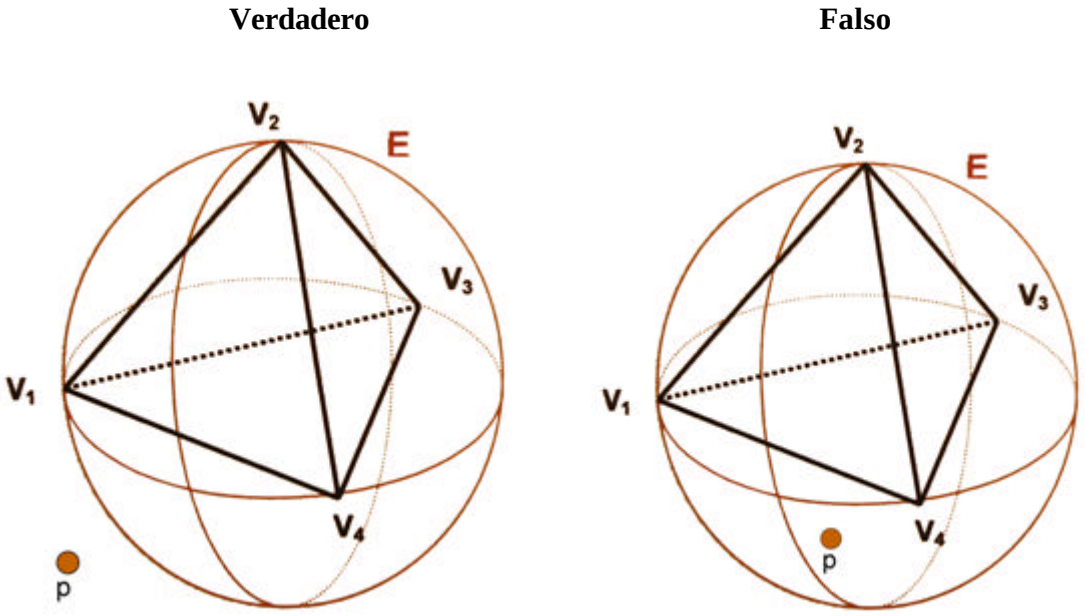


Figura 2.7: Test de la Esfera

2.1.9. Test Relajado de la Esfera

Para el caso particular en que el punto *p* se encuentra muy cercano a la esfera *E*, ya sea por su interior o exterior, y debido a los errores involucrados en la práctica al utilizar aritmética de punto flotante, la decisión del Test se relaja usando un parámetro de tolerancia *epsilon*, de modo de obtener siempre una decisión consistente.

Sean *c* y *r*, radio y centro de *E* respectivamente.

- El Test relajado de la esfera retorna *verdadero* si *distancia(c,p) = r - epsilon.*
- El Test relajado de la esfera retorna falso si *distancia(c,p) < r - epsilon.*

2.1.10. Triangulación Delaunay 2D

Una triangulación T definida por un conjunto de puntos P en el plano, es Delaunay, si para cada triángulo $t \in T$ el Test del Círculo retorna *verdadero* para los restantes puntos de P.

Los algoritmos más utilizados para generar una triangulación Delaunay 2D corresponden a:

- Intercambio de Diagonales: Una triangulación cualquiera se puede transformar en una de Delaunay intercambiando aristas tal como se muestra en la figura 2.8.

- Generación de Cavidad: Si un punto es incluido en el circumcírculo de uno o más triángulos, estos se eliminan, generando una cavidad que debe ser triangulada de nuevo (Ver figura 2.9).

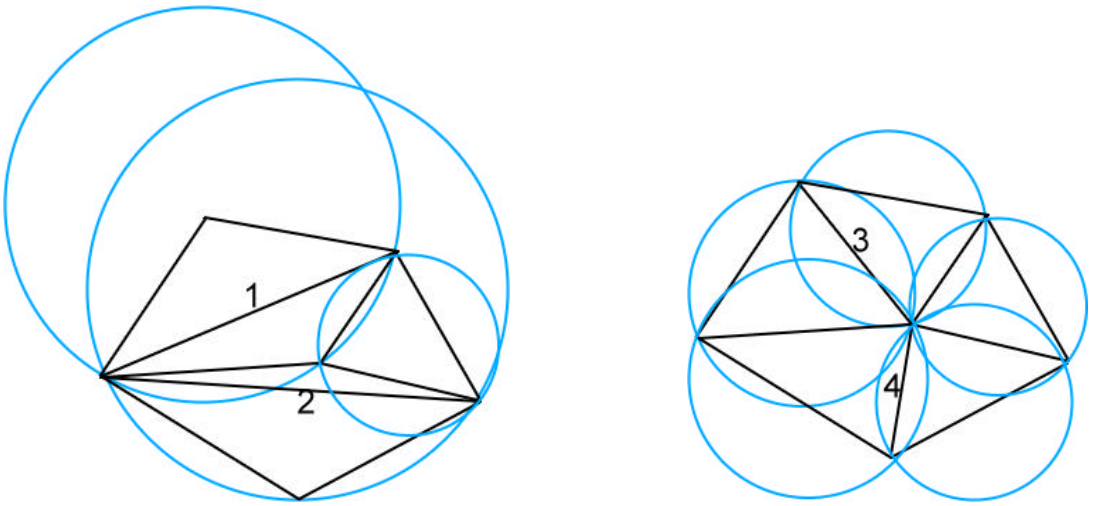


Figura 2.8: Algoritmo de intercambio de diagonales. A la izquierda se muestra una triangulación en la cual el Test de Delaunay es falso. A la derecha se muestra una triangulación con su test de Delaunay verdadero que resulta de intercambiar las diagonales 1 por la 3 y 2 por la 4.

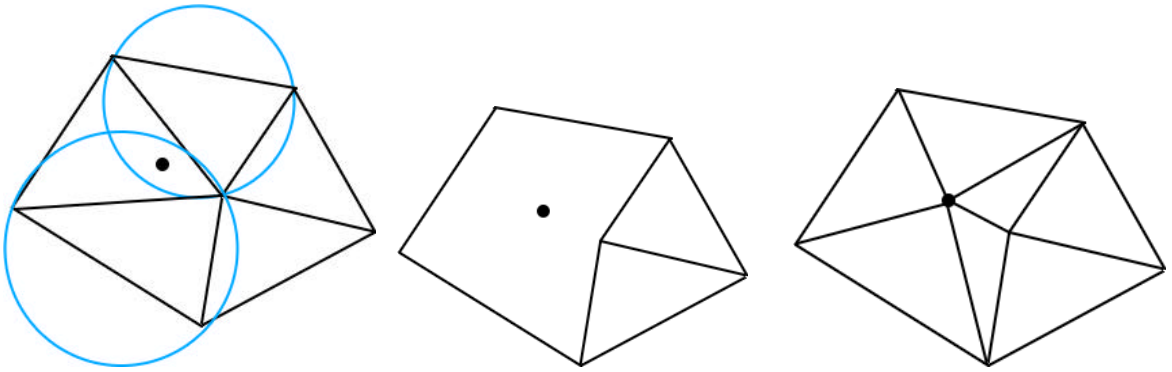


Figura 2.9: Algoritmo de Generación de Cavidad. En la triangulación de la izquierda se inserta un punto que está dentro del circumcírculo de dos triángulos. Estos triángulos se eliminan tal como se muestra en la figura de centro y se genera una cavidad. Esta cavidad es triangulada según la condición de Delaunay tal como se muestra en la figura de la derecha.

2.1.11. Triangulación Delaunay 3D

Una triangulación T definida por un conjunto de puntos P en el espacio, es Delaunay si para cada tetraedro $t \in T$ el Test de la Esfera retorna *verdadero* respecto de los restantes puntos de P.

2.2. Conceptos de Ingeniería de Software

2.2.1 Programación Orientada a Objetos (OOP)

La Programación Orientada a Objetos (OOP) es un paradigma de programación que define los programas en términos de "clases de objetos", objetos que son entidades que combinan estado (es decir, datos), comportamiento (esto es, procedimientos o métodos) e identidad (propiedad del objeto que lo diferencia del resto). La programación orientada a objetos expresa un programa como un conjunto de estos objetos, que colaboran entre ellos para realizar tareas. Esto permite hacer los programas y módulos más fáciles de escribir, mantener y reutilizar. De esta forma, un objeto contiene toda la información, (los denominados atributos) que permite definirlo e identificarlo frente a otros objetos pertenecientes a otras clases (e incluso entre objetos de una misma clase, al poder tener valores bien diferenciados en sus atributos). A su vez, dispone de mecanismos de interacción (los llamados métodos) que favorecen la comunicación entre objetos (de una misma clase o de distintas), y en consecuencia, el cambio de estado en los propios objetos. Esta característica lleva a tratarlos como unidades indivisibles, en las que no se separan (ni deben separarse) información (datos) y procesamiento (métodos).

2.2.2 Patrones de Diseño

Los patrones de diseño describen un problema que ocurre repetidas veces en algún contexto determinado de desarrollo de software, y entregan una buena solución ya probada. Esto ayuda a diseñar correctamente en menos tiempo, ayuda a construir problemas reutilizables y extensibles, y facilita la documentación. Los patrones enseñan a aplicar de manera eficaz qué hacer con la herencia, el polimorfismo, y todas las ventajas que posee la Programación Orientada a Objetos.

Un patrón de diseño es una descripción de clases y objetos comunicándose entre sí, adaptada para resolver un problema de diseño general en un contexto particular.

A continuación se muestra una lista con los patrones de diseño más habituales [10].

2.2.2.1 Patrones de Creación

- **Abstract Factory**: Proporciona una interfaz para crear familias de objetos o que dependen entre sí, sin especificar sus clases concretas.
- **Builder**: Separa la construcción de un objeto complejo de su representación, de forma que el mismo proceso de construcción pueda crear diferentes representaciones.
- **Factory Method:** Define una interfaz para crear un objeto, pero deja que sean las subclases quienes decidan qué clase instanciar. Permite que una clase delegue en sus subclases la creación de objetos.
- **Prototype:** Especifica los tipos de objetos a crear por medio de una instancia prototípica, y crear nuevos objetos copiando este prototipo.
- **Singleton**: Garantiza que una clase sólo tenga una instancia, y proporciona un punto de acceso global a ella.

2.2.2.2 Patrones Estructurales

- **Adapter:** Convierte la interfaz de una clase en otra distinta que es la que esperan los clientes. Permiten que cooperen clases que de otra manera no podrían por tener interfaces incompatibles.
- **Bridge**: Desvincula una abstracción de su implementación, de manera que ambas puedan variar de forma independiente.
- **Composite**: Combina objetos en estructuras de árbol para representar jerarquías de parte-todo. Permite que los clientes traten de manera uniforme a los objetos individuales y a los compuestos.
- **Decorador:** Añade dinámicamente nuevas responsabilidades a un objeto, proporcionando una alternativa flexible a la herencia para extender la funcionalidad.
- **Facade**: Proporciona una interfaz unificada para un conjunto de interfaces de un subsistema. Define una interfaz de alto nivel que hace que el subsistema sea más fácil de usar.

- **Flyweight**: Usa el compartimiento para permitir un gran número de objetos de grano fino de forma eficiente.
- **Proxy**: Proporciona un sustituto o representante de otro objeto para controlar el acceso a éste.

2.2.2.3 Patrones de Comportamiento

- **Chain of Responsibility**: Evita acoplar el emisor de una petición a su receptor, al dar a más de un objeto la posibilidad de responder a la petición. Crea una cadena con los objetos receptores y pasa la petición a través de la cadena hasta que ésta sea tratada por algún objeto.
- **Command**: Encapsula una petición en un objeto, permitiendo así parametrizar a los clientes con distintas peticiones, encolar o llevar un registro de las peticiones y poder deshacer la operaciones.
- **Interpreter**: Dado un lenguaje, define una representación de su gramática junto con un intérprete que usa dicha representación para interpretar las sentencias del lenguaje.
- **Iterator**: Proporciona un modo de acceder secuencialmente a los elementos de un objeto agregado sin exponer su representación interna.
- **Mediator**: Define un objeto que encapsula cómo interactúan un conjunto de objetos. Promueve un bajo acoplamiento al evitar que los objetos se refieran unos a otros explícitamente, y permite variar la interacción entre ellos de forma independiente.
- **Memento**: Representa y externaliza el estado interno de un objeto sin violar la encapsulación, de forma que éste puede volver a dicho estado más tarde.
- **Observer**: Define una dependencia de uno-a-muchos entre objetos, de forma que cuando un objeto cambia de estado se notifica y actualizan automáticamente todos los objetos.
- **State**: Permite que un objeto modifique su comportamiento cada vez que cambia su estado interno. Parecerá que cambia la clase del objeto.
- **Strategy**: Define una familia de algoritmos, encapsula uno de ellos y los hace intercambiables. Permite que un algoritmo varíe independientemente de los clientes que lo usan.

- **Template Method**: Define en una operación el esqueleto de un algoritmo, delegando en las subclases algunos de sus pasos. Permite que las subclases redefinan ciertos pasos del algoritmo sin cambiar su estructura.
- **Visitor**: Representa una operación sobre los elementos de una estructura de objetos. Permite definir una nueva operación sin cambiar las clases de los elementos sobre los que opera.

2.2.3 Patrones de Diseño Convenientes en un Generador de Mallas.

Cada proceso en un generador de mallas puede realizarse con distintas estrategias. Esto va desde la generación de la malla inicial, pasando por los procesos intermedios e incluso en la malla final.

En la figura 2.10 se aprecia la estructura del patrón S*trategy*. La idea básica de este patrón es mantener un conjunto de algoritmos que cumplan una misma función. Por ejemplo si se desea ordenar un arreglo de números, se puede usar quicksort, mergesort, el método de la burbuja, o cual otro, y todos ellos entregarán un arreglo ordenado, aunque su funcionamiento y orden sea distinto. En una malla geométrica ocurre lo mismo. Existen distintos criterios y algoritmos para cada proceso de la malla. Estas estrategias se deben elegir de acuerdo a los requisitos del problema particular en el cual se está trabajando. Es por ello que en el diseño del generador de mallas mixtas resulta conveniente usar este patrón en todas sus fases, así cuando se genere la malla inicial, se puede generar una malla Delaunay u otra, cuando se refine, se puede usar un refinamiento por el método Octree o por Elementos mixtos, y dada la flexibilidad que permite este patrón, es posible aumentar el número de algoritmos que participen en los distintos pasos.

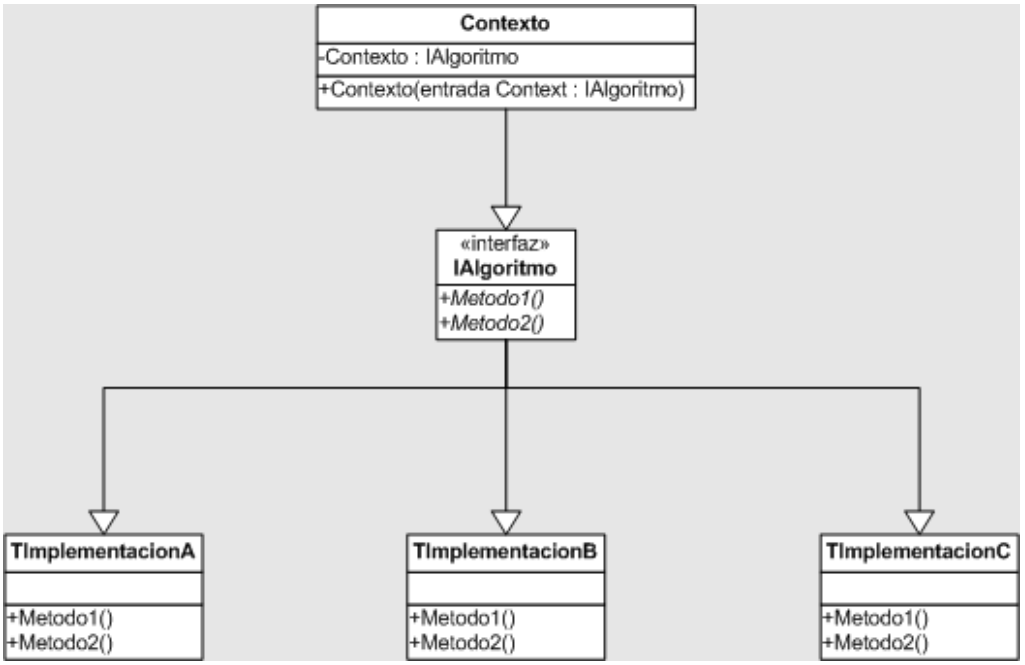


Figura 2.10: Estructura patrón de diseño *Strategy*

Otro patrón de diseño interesante y que resulta muy simple de explicar y entender es el *Singleton.* Este patrón sirve cuando buscamos restringir la creación de instancias de un objeto, obligando a que sólo se pueda crear una única y exclusiva. En el caso de un generador de mallas, debiese existir una única instancia de *Mesh*, clase que almacena toda la información de la malla. En la figura 2.11 se muestra la estructura general del patrón. Este patrón funciona creando una instancia del objeto cuando esta no existe. Si se solicita y ya existe, entonces se devuelve la existente.

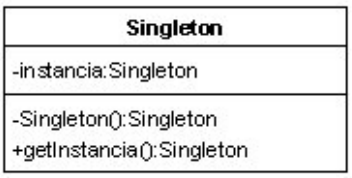


Figura 2.11: Estructura patrón de diseño *Singleton.*

Otros patrones de diseño que son recomendables usar en un generador de mallas [3], pero que no serán abordados en esta memoria, aunque, dada la extensibilidad del diseño que se presentará, podrán ser incluidos sin mucha dificultad, son *Composite,* para las regiones, *Observer*, para la evaluación de la malla y *Command*, para la interfaz.

Capítulo 3

Diseño OOP del Generador de Mallas

Este capítulo describe primero el algoritmo *met*, cuyo diseño no es orientado a objetos. En la segunda parte del capítulo se presenta un nuevo diseño orientado a objetos para un generador de mallas con elementos mixtos.

3.1 Análisis del Diseño Anterior

A continuación se describirá el generador de ma llas mixtas *met*, junto a su diseño como a su implementación.

3.1.1 Descripción General

El generador de mallas *met* ejecuta los siguientes pasos:

- Genera una malla inicial que cubre y representa la geometría del objeto modelado.
- Refina la malla inicial para cumplir la densidad requerida especificada por el usuario.
- Genera una malla apropiada 1-irregular.
- Particiona los elementos 1-irregular en elementos finales válidos: cubos, prismas y pirámides, entre otros.
- Almacena la información requerida por la aplicación.

En su estado actual, el *met* tiene las siguientes características:

- Acepta un formato de entrada para la geometría y especificaciones de refinamiento.
- Tiene un algoritmo para generar la malla inicial.
- Tiene un algoritmo de refinamiento.
- Tiene un algoritmo de generación de malla 1-irregular.
- Genera un formato de salida.

3.1.2 Descripción de la malla de elementos mixtos

Los elementos se representan a través de la estructura `Tree`. La información almacenada para cada nodo del árbol es la siguiente [9]:

```
Struct Tree{
     Integer        type;
     Integer        material;
     Integer        orientation[3];
     Integer        split_axes[3];
     Integer        refinement_levels[3];

     Integer        global_corner_numbers[Corners_per_element];
     Tree*          sons;
     Cut_info*      cut_information;

     Flag           edge_midpoint[Edges_per_element];
     Flag           face_midpoint[Faces_per_element];
};
```

El campo `type` se refiere al tipo de elemento. El arreglo `orientation` almacena una transformación con respecto a su posición normalizada. Normalmente, cada hijo hereda la orientación de su padre, sin embargo, en la partición de una pirámide, un prisma tiene otra orientación. El arreglo `split_axes` identifica los ejes locales a través de los cuales el nodo ha sido dividido. El arreglo `refinement_levels` permite obtener cuántas veces ha sido dividiso en una cierta dirección. La variable `depth_of_the_forest` es usada para evitar la generación de nuevos nodos, los cuales pueden incrementar la profundidad de un árbol haciendo todos los elementos divididles. El arreglo `global_corner_numbers` contiene los índices globales de cada punto que define el elemento. `cut_information` dice si alguna interfaz de borde o material no pudo ser ajustada durante la generación de macro elementos. Para cada nodo que no es hoja, `sons` apunta a nodos sucesores. Mientras se hace los elementos divididles, los arreglos `edge_midpoint` y `face_midpoint` mantienen el rastro de los puntos medios en aristas y puntos centrales en caras.

El proceso de generación de una grilla es como se muestra en el siguiente algoritmo:

```
Read_Device_Geometry(geometry_description);
Read_Refinement_and_Doping_Data(refinement_data, doping_data);
Fit_Device_Geometry(grid, geometry_description);
```
/* Se genera una grilla inicial compuesta por cuboides, prismas rectangulares y pirámides rectangulares. La grilla es manejada como un bosque donde cada macro-elemento es la raíz de un árbol.*/
```
Refine_Grid(grid, refinement_data, doping_data);
```
/* Una grilla irregular se genera refinando cada macro-elemento independientemente de los otros, con el objetivo de ajustarse a parámetros físicos y geométricos.*/
```
Make_Irregular_Leaves_Splittable(grid);
```
/*Una grilla finite de elements se obtiene después de particionar todos los elementos irregulares en tetraedros, pirámides, prismas y cuboides si pueden ser calculas las secciones interiores de Voronoi.*/
```
Assemble_Voronoi_Surfaces(grid);
Compute_Regions(grid, geometry_description, region_information);
```
/*Los elementos de la grilla son separados en regiones de acuerdo a los parámetros de entrada.*/
```
Write_Geometrical_Information(grid, region_information);
```
/*Los puntos de coordenadas, puntos por elemento y superficies de Voronoi son almacenadas en un archive.*/
```
Write_Doping_Information(grid, doping_data);
```
/*Los valores de doping para cada punto en la grilla son almacenados en un archivo.*/

Dado lo anterior, podemos ver que el algoritmo se maneja con la información del árbol (`Tree`) y con los métodos de la grilla (`grid`).

3.2 Nuevo Diseño Orientado a Objetos

Con el análisis realizado en la sección anterior, se puede presentar un diseño del generador de mallas mixtos en 3D basado en conceptos tales como orientación a objetos y patrones de diseño, los cuales promueven flexibilidad y mantenibilidad, entre otras cosas.

3.2.1 Metodología

En [2] se presenta un diseño para una familia de productos de herramientas de generación de mallas en 2D, las cuales pueden ser usados tanto en la generación de mallas de elementos o volúmenes finitos, así como para el procesamiento de imágenes, entre otros. En [3] se presenta un diseño de un framework de generación de mallas en 3D basado en conceptos para producir un software flexible y fácil de modificar. Ambos diseños comparten el uso de orientación a objetos

y patrones de diseño, y es en ellos en el cual se basará el diseño del generador de mallas mixtas que se presenta en esta memoria.

El diseño del generador de mallas mixtas en 3D presentado en esta memoria sigue la siguiente directriz [2]: el uso de herencia para lograr un software que sea fácil de mantener, entender y extender, y la identificación de patrones de diseño que puedan ser útiles en este modelo.

3.2.2 Requerimientos y Análisis

Tal como se menciona en la sección 3.1, la implementación de un generador de mallas mixtas en 3D basada en octrees debe permitir los siguientes procesos:

- Recibir la entrada de la geometría aceptando diferentes formatos.
- Generación de una malla inicial que fije el dominio de la geometría.
- Generación de una malla intermedia que satisface los requerimientos de densidad especificados por el usuario.
- Generación de una malla mejorada que satisface criterios de calidad.
- Generación de una malla final.
- Almacenar la malla final en algún formato de salida.

Es así como según los requerimientos, nuestro modelo debe permitir aspectos como el poder ocupar distintas estrategias para:

- Generar una malla inicial.
- Refinar, mejorar y desrefinar una malla.
- Implementar criterios de refinamiento y mejoramiento de la malla.
- Generar una malla final.
- Implementar regiones en refinamiento y mejoramiento.

La incorporación de nuevas estrategias para refinamiento, mejoramiento, generación de mallas iniciales o finales, etc, no deberían causar un mayor impacto en la modificación del código fuente.

Dada toda esta información, se debe decidir en cómo organizar todos los complejos procesos asociados al generador, en sus distintas etapas. Se ha decidido manejar cada diferente proceso de generación de mallas y cada criterio siguiendo la filosofía del patrón Strategy, esto es porque hay varias formas de implementar los mismos procesos y se desea que sean intercambiables según las decisiones del usuario, además se busca que el software sea extensible y sea capaz de permitir la incorporación de nuevas estrategias y criterios haciendo muy pocas modificaciones en el código.

3.2.3 Diseño

La Figura 3.1 muestra el nuevo diseño de clases propuesto para el generador de mallas mixtas, basado en [2] y [3]. Se puede apreciar que según los requerimientos, cada proceso (malla inicial, refinamiento, mejoramiento, malla final) se representa como una clase abstracta y cada distinta estrategia está implementada en una subclase concreta. Por ejemplo, para la generación de la malla inicial se dispone de la clase `InitialMeshAlgorithm`, la cual es una clase abstracta y está implementada en varias subclases, siendo una de ellas la elegida para la generación de la malla inicial. Cabe destacar que para cada estrategia se agrega una subclase *`Dummy`*`_strategy`, la cual no hace nada. Por ejemplo si la malla ya viene refinada y se desea omitir este paso, se debe elegir la estrategia `DummyRefinementAlgorithm` en la fase de refinamiento.

Cada clase abstracta junto con sus subclases concretas se explican a continuación.

3.2.3.1 Mesh

La clase `Mesh` contiene la información sobre la malla a manejar. Como es un generador mixto en 3D, éste se debe componer de distintos tipos de elementos (Cuboides, pirámides, tetraedros, prismas), y cada elemento está compuesto de puntos, aristas y caras. Los elementos se

modelan con una clase abstracta (`Element`) y con sus subclases implementando a cada elemento específico. Puntos, aristas y caras también son modelados como clases, cada uno debiendo proveerse de sus funcionalidades específicas, como por ejemplo el guardar la información de sus vecinos. La figura 3.2 muestra el diagrama de clases de `Mesh`.

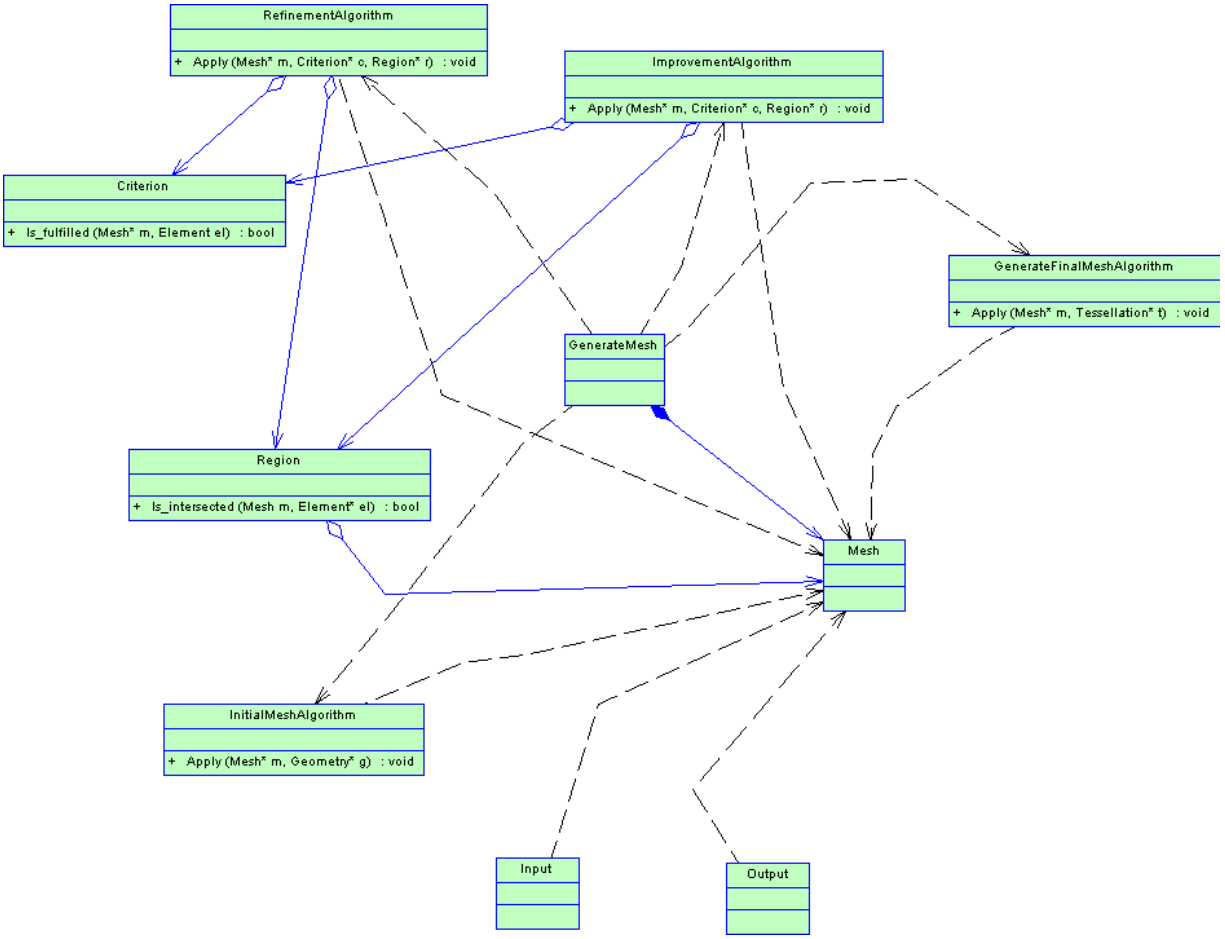


Figura 3.1: Diagrama de clases general

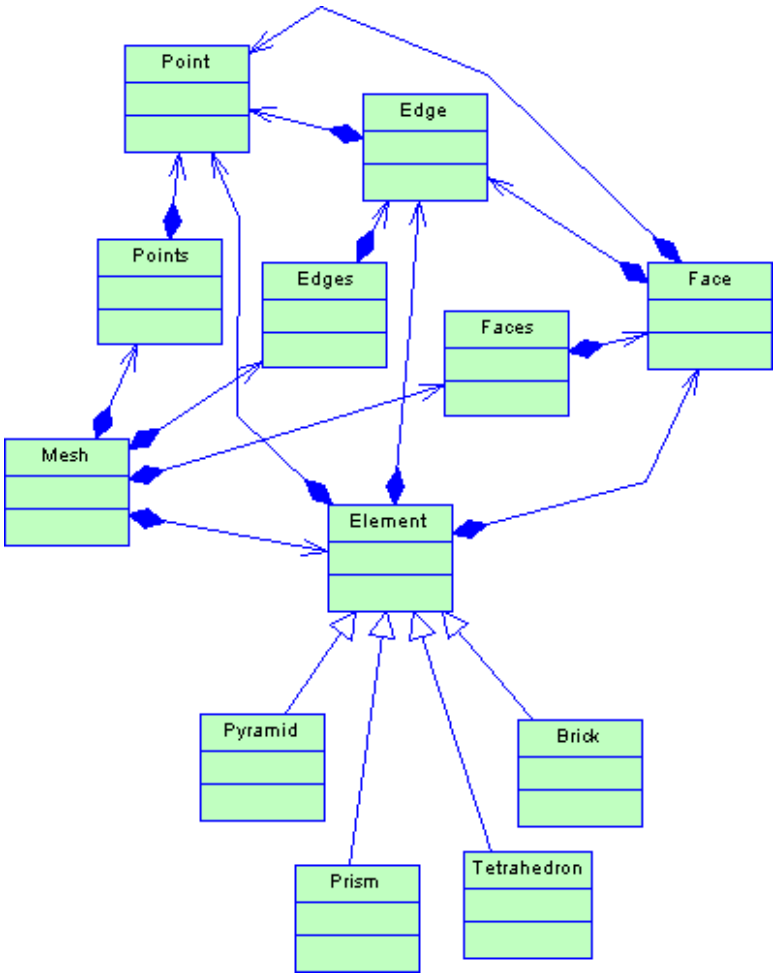


Figura 3.2: Diagrama de clases de `Mesh`

3.2.3.2 InitialMeshAlgorithm

La figura 3.3 muestra el diagrama de clases que representa diferentes algoritmos para generar una malla inicial. Además de la ya explicada subclase *Dummy_* se dispone de una implementación para generar una malla inicial Delaunay, una malla inicial Mixta, y se agrega una subclase que podría representar cualquier otro algoritmo que se desee agregar. Cada subclase debe implementar el método virtual `Apply`, el cual recibe como entrada la geometría del dominio y un objeto vacío `Mesh`, luego construye la correspondiente malla inicial y agrega esta información en el objeto que almacena la malla.

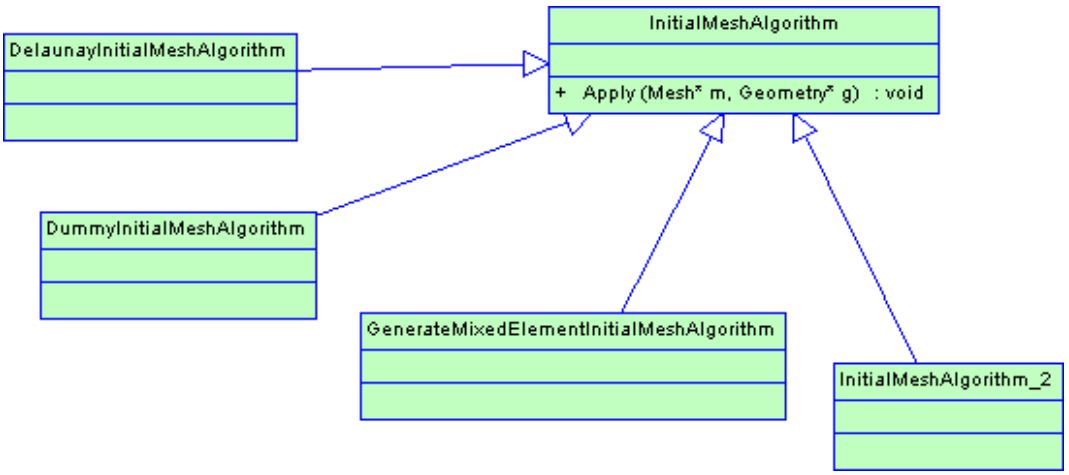


Figura 3.3: Diagrama de clases para los algoritmos de generación de malla inicial.

3.2.3.3 RefinementAlgorithm

En la figura 3.4 se muestra el diagrama de clases que representa los algoritmos de refinamiento del generador de mallas. Se incluyen las clases concretas `OctreeRefinementAlgorithm` y `MixedElementRefinementAlgorithm`. Además la subclase `DummyRefinementAlgorithm` es usada cuando no se requiere refinar la malla. Se incluye una subclase que puede representar cualquier otro algoritmo de refinamiento. Cada subclase debe implementar el método `Apply`, el cual recibe como entrada una malla, un criterio y una región, y refina la malla hasta que el criterio es cumplido y si hay intersección con la región.

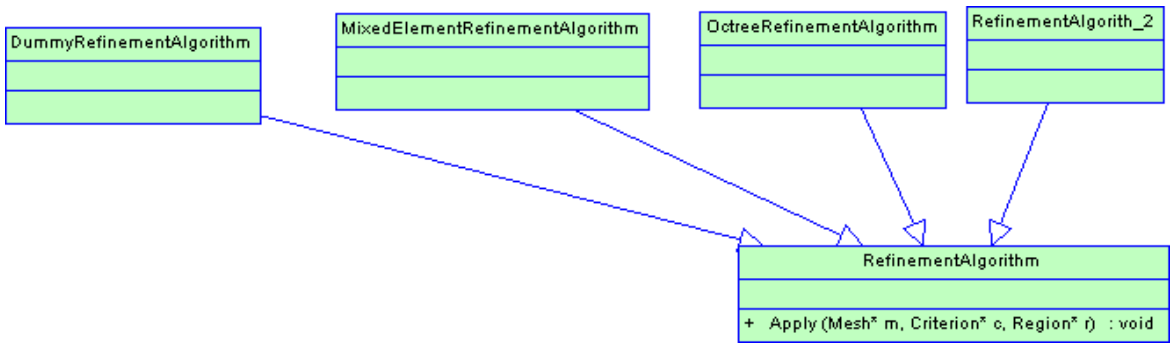


Figura 3.4: Diagrama de clases para los algoritmos de refinamiento

3.2.3.4 ImprovementAlgorithm

En la figura 3.5 se muestra el diagrama de clases que representa los algoritmos de mejoramiento del generador de mallas. Se incluye la subclase concreta `ImprovingMixedElement`. Además la subclase `DummyImprovementAlgorithm` es usada cuando no se requiere mejorar la malla. Se incluye una subclase que puede representar cualquier otro algoritmo de mejoramiento. Cada subclase debe implementar el método `Apply`, el cual recibe como entrada una malla, un criterio y una región, y se mejora la malla hasta que el criterio es cumplido y si hay intersección con la región.

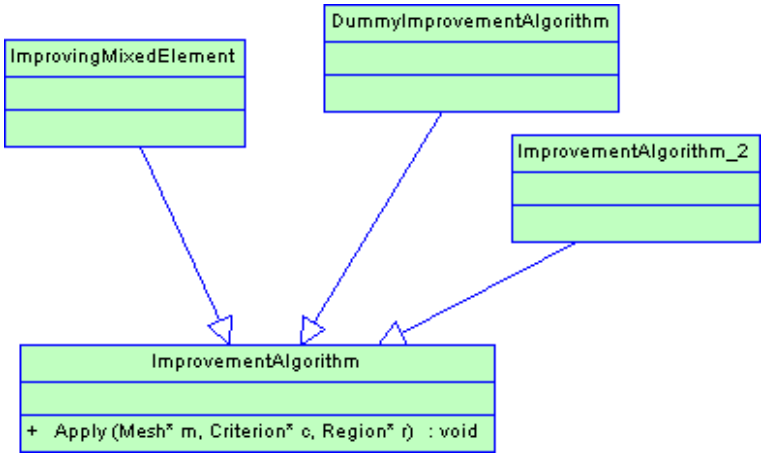


Figura 3.5: Diagrama de clases para los algoritmos de Mejoramiento de la malla

3.2.3.5 Criterion

La figura 3.6 muestra el diagrama de clases para el modelo de criterios. El método virtual de la clase abstracta `Criterion` es `Is_fulfilled`. Este método recibe como entrada la malla y el elemento que se desea chequear, se evalúa y retorna si se cumple o no el criterio específico.

Se han agregado los siguientes criterios como subclases concretas en este generador de mallas: `DopingConcentration`, `MaximunEdgeLength`, `MinimunEdgeLength`, `RadiousEdgeRatio`, `AspectRatio` y `VolumenLongestEdgeRatio`. Dado que se está usando el patrón de diseño *Strategy*, es posible agregar nuevos criterios sin mucha dificultad.

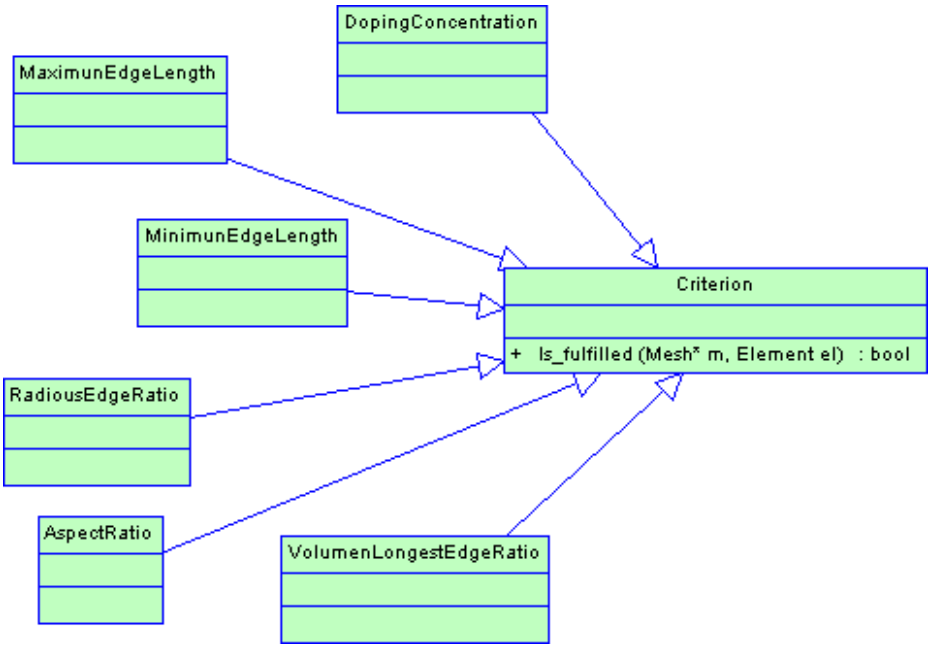


Figura 3.6: Diagrama de clases para los criterios de refinamiento y mejoramiento

3.2.3.6 Region

La figura 3.7 muestra el diagrama de clases para las regiones donde los algoritmos de refinamiento o mejoramiento de la malla son aplicados. La clase abstracta es llamada `Region`, y sus subclases son `Cube`, `Point_`, `Sphere`, `Edge` y `Whole_geometry`. La clase Region provee el método virtual Is_intersected, el cual se debe redefinir e implementar en cada subclase. Este método retorna true o false dependiendo si el elemento que se está evaluando intersecta a la región actual o no. Para el caso de `Whole_geometry`, dado que esta subclase representa a toda la geometría del problema, siempre retornará true.

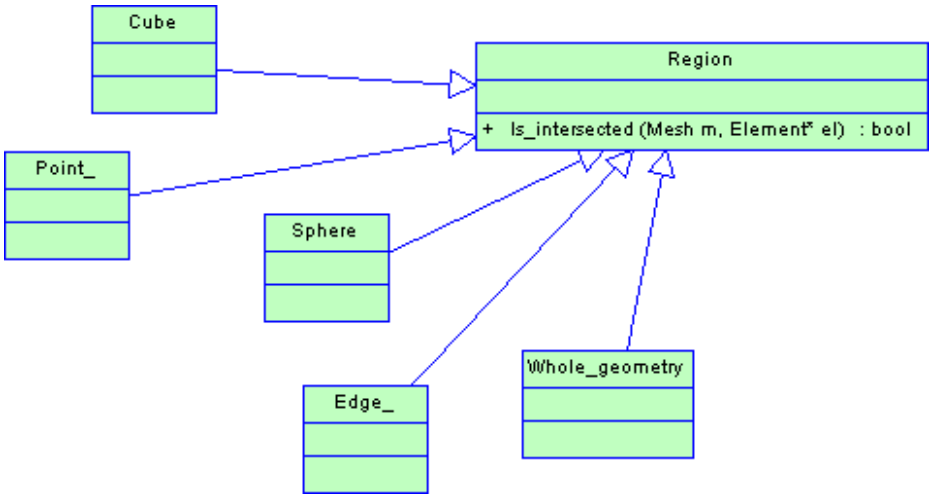


Figura 3.7: Diagrama de clases para las regiones

3.2.3.7 GenerateFinalMeshAlgorithm

La figura 3.8 muestra el diagrama de clases de los algoritmos de generación de la malla final. En la clase abstracta `GenerateFinalMeshAlgorithm` se define el método virtual `Apply`, el cual recibe como entrada la malla y la teselación final vacía. Luego el algoritmo elegido genera la malla final y devuelve la teselación. Se han definido dos subclases, `DelaunayFinalMeshAlgorithm` y `PatternsFinalMeshAlgorithm`. Nuevamente se ha agregado una clase cuando no se requiere generar la malla final, es decir `DummyFinalMeshAlgorithm`.

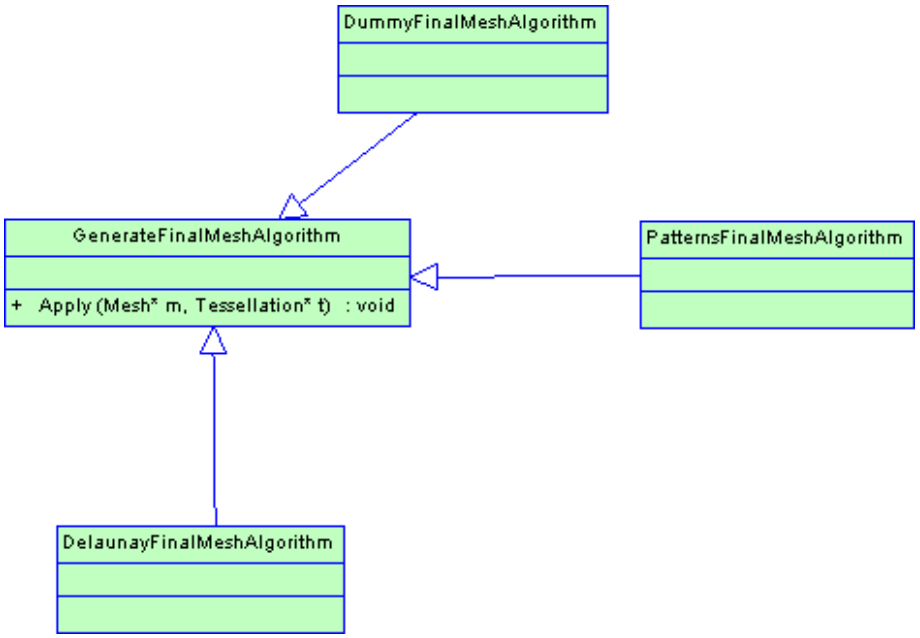


Figura 3.8: Diagrama de clases para los algoritmos de malla final

3.2.3.8 Input y Output

Estas clases representan los distintos formatos de entrada y de salida que permite el generador de mallas mixtas. Para cada formato se debe implementar su subclase específica que mantiene toda la información de dicho formato.

3.2.3.9 GenerateMesh

Representa al *Main* del generador, el cual usa los procesos para crear, refinar, mejorar y generar la malla final.

Capítulo 4

Diseño e Implementación de un Algoritmo de Malla Final Delaunay

4.1 Descripción del Algoritmo de particionamiento

4.1.1 Descripción General

Dada la explicación general del diseño según 3.1, nos centraremos en lo concerniente a la partición de un cubo dada una configuración de puntos de Steiner en él. Este cubo es resultado del proceso de refinamiento y mejoramiento de la malla en sus fases previas. Dados estos procesos, quedan elementos con configuraciones con puntos en sus aristas (puntos de Steiner), y en la fase de la generación de la malla final se aplica el algoritmo descrito a continuación.

El funcionamiento en esta parte del generador se basa en el uso de un cubo creado por medio de 8 puntos. Con este cubo más la asignación de uno o más puntos de Steiner en algunas aristas, se crea una configuración. Con esta configuración se revisa un conjunto de plantillas para entregar un resultado inmediato de una teselación, la que puede incluir tetraedros, pirámides o prismas.

A continuación se describe el proceso de obtención de una teselación para un cuboide 1-irregular:

- Se crea un Grid_info, el cual es un objeto que guarda la información de toda la malla (en este caso sólo de un cuboide 1-irregular).

- Se crean arreglos para los índices de los vértices del cuboide y para los puntos medios posibles en las aristas (Steiner points).

- Se asignan las dimensiones del cuboide, dados sus dos puntos extremos.

- Con estos datos se asignan los valores a todos los puntos y se asignan además los valores a todos los puntos medios posibles.

- Obtener el cuboide 1-irregular a procesar con uno o más puntos de Steiner.

Todo lo descrito anteriormente corresponde al almacenamiento preliminar del cuboide con el objetivo de poder realizar las operaciones siguientes, los cuales son:

- Verificar si es particionable el cuboide irregular por los métodos basados en templates (partición pre-almacenada en arreglo).

- Si es particionable.
    - Obtener la teselación predefinida capturando los elementos a través del objeto `patterns`.

- Si no es particionable.
    - Decir que no es particionable.

4.1.2 Métodos y Estructuras de datos

Las particiones de elementos 1-irregular se manejan a través de la clase `Patterns`. Esta clase es la más importante para el estudio del algoritmo.

La clase `Patterns` almacena la información de varias configuraciones de elementos 1-irregular que podrían aparecer. Los elementos 1-irregular pueden ser cuboides, tetraedros, prismas, pirámides o el complemento del cuboide, todos ellos con uno o más puntos medios en sus aristas. Lo interesante de este módulo es que al reconocer el patrón, entrega inmediatamente su partición en elementos. Por ejemplo, un cuboide 1-irregular con sólo una arista con punto de Steiner ya está en la lista de patrones, y al solicitar su teselación entrega una lista de cuatro pirámides.

A continuación se muestran algunos métodos importantes de esta clase:

```
INTEGER Is_splittable (Element_types elt_type, Bit_table pattern);
```

`Is_splittable` retorna si nosotros conocemos cómo dividir un elemento 1-irregular, con un patrón asociado, en tetraedros, prismas o pirámides. Retorna si se puede o no dividir este elemento. Los parámetros son:

- `Element_types elt_type`: El elemento con el cual estamos trabajando (cuboide, prisma, etc).
- `Bit_table pattern`: Es el patron asociado al elemento. Bit_table es un entero que según su número binario, indica cuál arista o cara está marcado como punto de Steiner.

```
INTEGER Pattern_type (Element_types elt_type, Bit_table pattern);
```

`Pattern_type` retorna el template correspondiente a un patrón dado. Los parámetros son los mismos que en la definición anterior.

4.2 Diseño e Implementación de un Nuevo Algoritmo de Malla Final

En la figura 4.1 se aprecia el algoritmo de generación de malla final en el cual se basa este capítulo, y en general la implementación realizada en este trabajo de titulación. La subclase es `DelaunayFinalMeshAlgorithm` y es la implementación de un algoritmo de generación de mallas que resuelve el problema de entradas de cuboides 1 irregular que son resultado de los procesos anteriores, donde tanto la refinación como el mejoramiento de la malla entregan algunos elementos que tienen a lo más un punto extra en sus vértices, estos puntos son los llamados puntos de Steiner y generan configuraciones que el algoritmo presentado es capaz de resolver, y que junto con el método de Patterns son capaces de entregar una malla final de elementos mixtos.

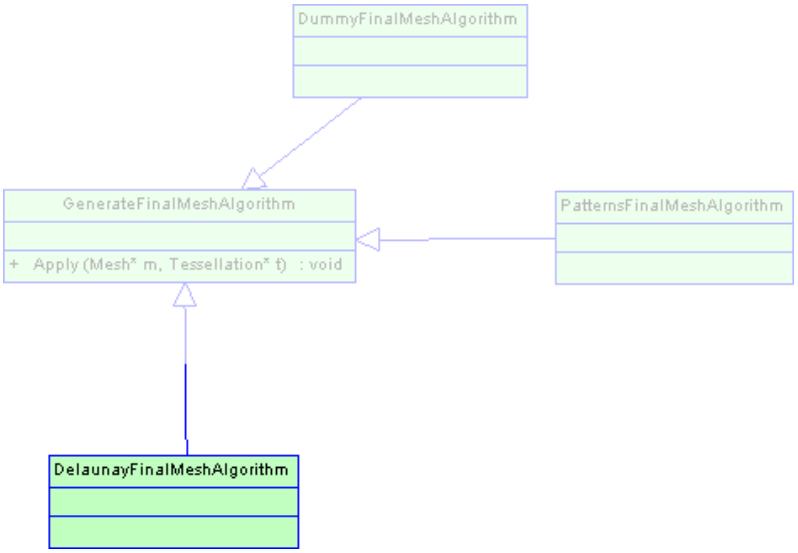


Figura 4.1: El detalle de la clase `DelaunayFinalMeshAlgorithm` será descrito en este capítulo.

4.2.1 Idea General y Convenciones Sobre Figuras a Usar

Una teselación en 3 dimensiones puede ser expresada como una 4-tupla formada por elementos, caras, aristas y puntos. La teselación que entregará como resultado esta memoria es una triangulación en 3D, por lo tanto está basada en tetraedros.

Dada esta teselación, necesitamos almacenar cada uno de los datos de ella. Es por esto que se debe construir una estructura de datos capaz de realizar las siguientes operaciones:

- Agregar y eliminar un elemento (tetraedros).
- Agregar y eliminar una cara.
- Agregar y eliminar una arista.
- Entregar los elementos que comparten una cara (para refinar la partición incluyendo el algoritmo del test de la esfera).
- Entregar los elementos que comparten una arista (para particionar los elementos recorriendo las aristas con puntos de Steiner asociados).

A continuación se describirán los elementos que formarán parte de una teselación y se detallarán las convenciones usadas con respecto a cada figura:

4.2.1.1 Cuboide

Es el elemento principal, ya que el generador de mallas que se está modelando está basado en octrees. El cuboide es el elemento mas pequeño de un octree y es el resultado de la división de cada cuboide más grande en 8 octantes. Si esta división recursiva es la última, entonces estamos en frente a un cuboide con la cual el modelamiento sobre la que trata esta memoria trabajará.

Un cuboide está formado por 6 caras, 12 aristas y 8 puntos, cada uno de ellos claramente identificados. Sus seis caras son cuadradas y congruentes, además todas sus caras son de cuatro lados y paralelas dos a dos.

En la figura 4.2 se muestra un cuboide y sus convenciones con respecto a los vértices, aristas y caras.

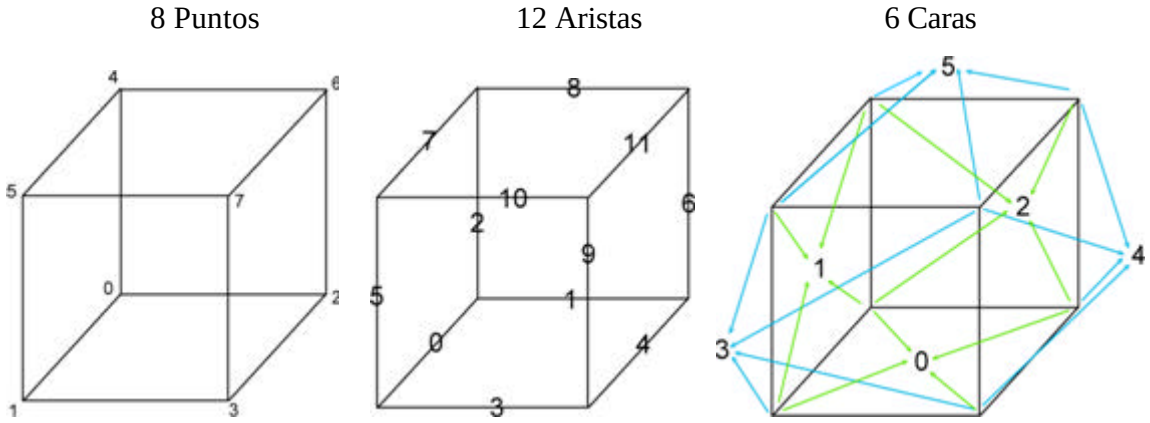


Figura 4.2: Un cuboide representado por sus índices para los puntos, aristas y caras.

4.2.1.2 Tetraedro

Un tetraedro es un poliedro formado por cuatro caras, cuatro vértices y seis aristas. En cada cara concurren 3 vértices. Cada uno de sus puntos, aristas y caras están debidamente identificados según la figura 4.3.

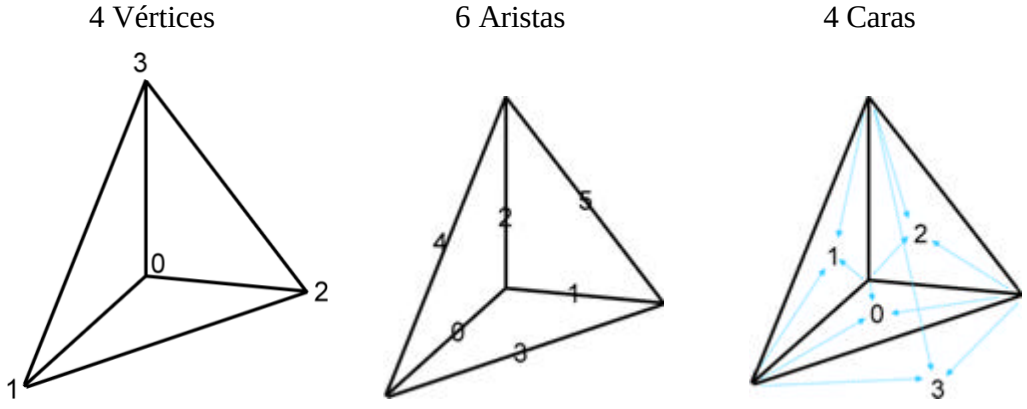


Figura 4.3: Representación interna de un tetraedro, según sus vértices, aristas y caras.

4.2.1.3 Otras figuras

Los elementos de la figura 4.4 se muestran con su estructura definida en cuanto a los identificadores de cada vértice, arista y cara. Estos elementos forman parte del generador de mallas *met*, pero no están incluidos en esta memoria. Se espera que en próximos trabajos de titulación se trabaje con ellos.

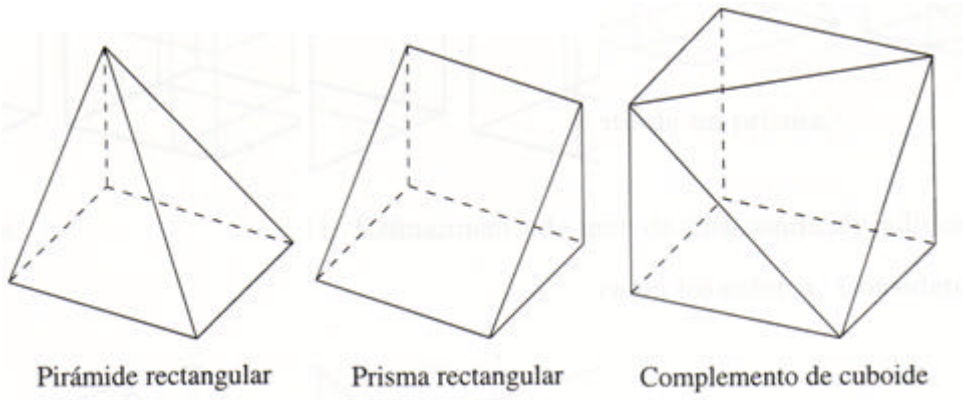


Figura 4.4: Elementos que forman parte de una malla mixta.

4.2.2 Desarrollo y Funcionamiento del Algoritmo

4.2.2.1 Los templates

El generador de mallas mixtas *met* contiene un archivo muy importante dentro de su implementación. Este archivo es llamado `tree_constants.C` y contiene la información de las tablas y constantes usadas en el generador.

Al extender la aplicación anterior, se han agregado a este archivo la información necesaria para el modelamiento de la malla que se presenta en esta memoria. Esta información se refiere a los índices usados en la malla inicial, y además se han registrado todos los casos de partición según los puntos de Steiner que contiene la información inicial y los casos para el mejoramiento tras la aplicación del Test de la Esfera. Con esta información registrada se busca disminuir el tiempo de procesamiento de la partición, pues por cada punto de Steiner, se tiene una

configuración almacenada que se utiliza inmediatamente luego de encontrarla. Y por cada caso que implique una modificación de la partición al refinar, se accesan las nuevas configuraciones en forma rápida.

Al definir estos templates, se puede dar paso a la descripción de los distintos algoritmos.

4.2.2.2 La malla inicial

La malla inicial está compuesta por un cuboide con diagonales en todas sus caras. Con esto formamos una malla con cinco tetraedros, tal como en la figura 4.5.

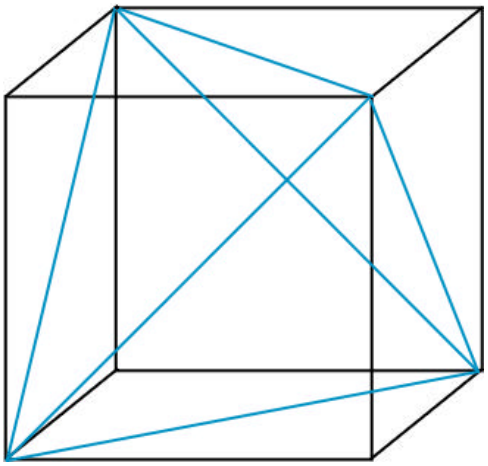


Figura 4.5: Teselación inicial de tetraedros.

Al ingresar seis diagonales, se crean nuevas aristas, nuevas caras y nuevos elementos. Las estructuras de datos implementadas soportan las operaciones necesarias y toda la nueva información se registra adecuadamente para entregar una teselación lista para su partición según los puntos de Steiner y posteriormente para su mejoramiento.

4.2.2.3 Algoritmo de Particionamiento

Dada una malla inicial con puntos de Steiner en sus aristas (pueden ser en las diagonales). Dado que la información de los puntos medios está registrada en una `bit_table`, el algoritmo de particionamiento se puede desglosar de la siguiente manera:

```
//cada bit en bit_table representa una arista
for cada bit en bit_table
        if bit está prendido (entonces hay punto de Steiner en arista)
                for cada elemento que comparte arista que tiene punto de Steiner
                        Particionar elementos que comparten arista según casos (son 6 casos)
                fin for
        fin if
fin for
```

Para mostrar de manera sencilla lo que hace el algoritmo supongamos un cuboide 1-irregular con 3 puntos de Steiner (3 aristas tienen puntos en su mitad) como el de la figura 4.6.

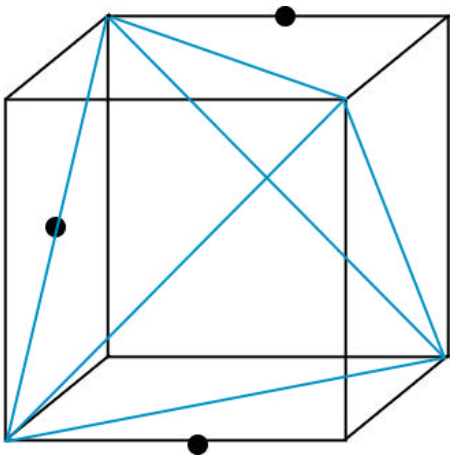


Figura 4.6: Un cuboide con 3 puntos de Steiner.

Según la figura 4.6, hay dos puntos en aristas normales y además hay un punto de Steiner en la mitad de una cara, que es lo mismo a que se encuentre en la mitad de una diagonal.

El algoritmo recorre las doce aristas del cuboide más sus seis diagonales y por cada vez que encuentre un punto medio marcado, particiona los elementos que comparten esa arista 1-irregular en dos o más elementos.

La partición para la configuración mostrada en la figura 4.6, quedaría de la forma que se muestra en la figura 4.7. Al recorrer las aristas buscando puntos de Steiner en orden (según su índice) nos encontramos con la arista 3, la cual se divide en 2 aristas y además se agregan otras 2 aristas (en verde) más 2 elementos. Se actualiza toda la información y se continúa buscando puntos de Steiner. Con el punto en la arista 8 pasa lo mismo: esta arista se divide en 2, se agregan 2 aristas nuevas y se elimina el tetraedro que contiene la arista 8 y se crean 2 nuevos tetraedros que ocupan su lugar. Al seguir recorriendo la `bit_table`, nos encontramos con la arista 12. Esta arista al ser diagonal es compartida por 3 elementos. El proceso es el mismo, se divide la arista en 2, y se dividen los 3 elementos en 2 cada uno. Como resultados aparecen 4 nuevas aristas (en rojo), y 6 nuevos elementos ocupando el lugar de los 3 que se dividieron. Se actualiza toda la información y si no hay más puntos de Steiner se puede continuar con los procesos siguientes.

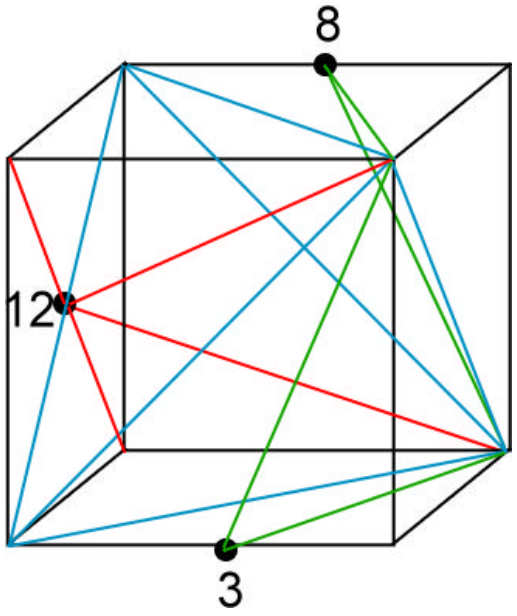


Figura 4.7: Particionamiento de un cuboide 1-irregular con 3 puntos de Steiner (uno de ellos sobre una diagonal).

Cuando se encuentra un punto de Steiner en una arista, el algoritmo debe analizar cada elemento que comparte esa arista. Aunque parece simple la división de un tetraedro en dos tetraedros, la información que maneja la teselación requiere un máximo cuidado, pues se debe

actualizar toda la información interna de los puntos, aristas y caras que participan del proceso. Es por ello que se han definido templates para cada caso de particionamiento que requiere un tetraedro, esto es con el fin de obtener la información más rápido y asegurarse de que sea consistente. Un tetraedro se compone de 6 aristas y se almacenan en un orden establecido (ver 4.2.1.2). Cada arista tiene la misma posibilidad de contener un punto de Steiner y en cada paso se procesa una sola arista, por lo tanto se definen 6 casos de particionamiento de tetraedros y el número que identifica el caso corresponde al número interno de la arista dentro del tetraedro (ver 4.2.1.2).

Los siguientes son los casos que dependiendo de en cuál arista de cada elemento (tetraedro):

Nota: En las figuras, los números celestes representan a las aristas, los verdes a las caras, los negros a los vértices y los azules son para bs elementos. Para todos los casos se mostrará su figura con punto de Steiner en caso base en la izquierda y a la derecha se mostrará el tetraedro después de haberse aplicado el algoritmo de particionamiento.

Caso Base: Los tetraedros mantienen su información según la figura 4.8. Notar que la cara 3 está formada por los puntos 1, 2 y 3.

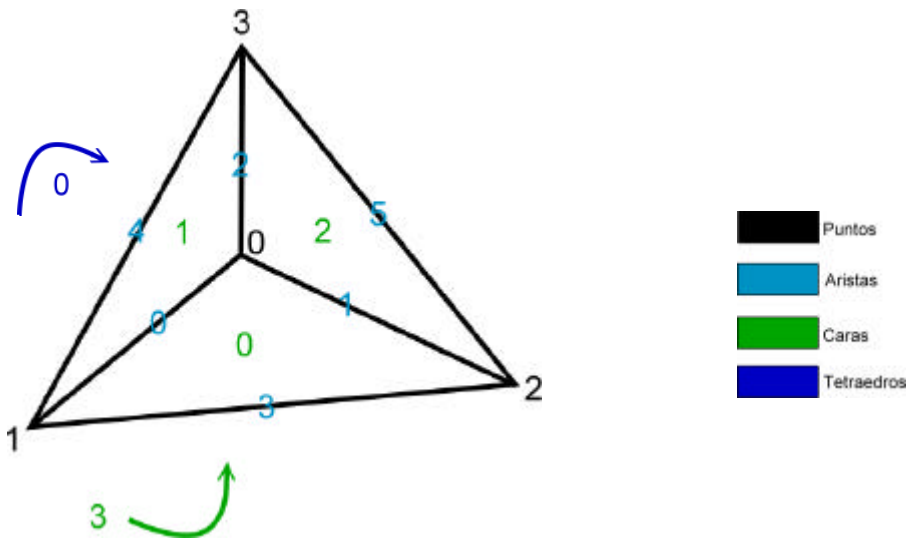


Figura 4.8: Caso base

Caso 0: La arista 0 tiene un punto de Steiner. Ver figura 4.9.

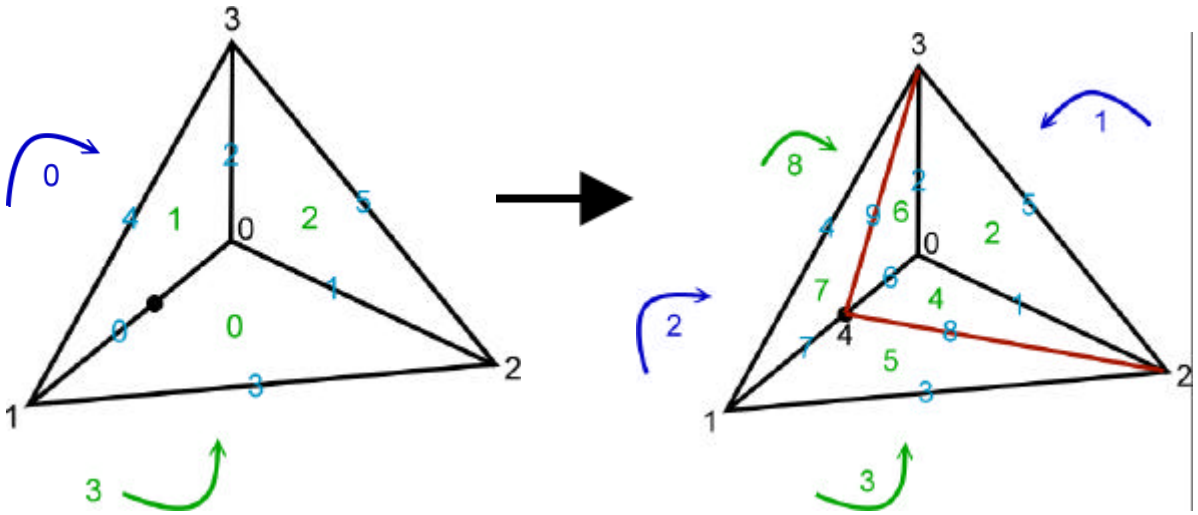


Figura 4.9: Caso 0. La cara 3 está formada por los puntos 1,2 y 3. La cara 8 está formada por los puntos 2,3 y 4.

Caso 1: La arista 1 tiene un punto de Steiner. Ver figura 4.10

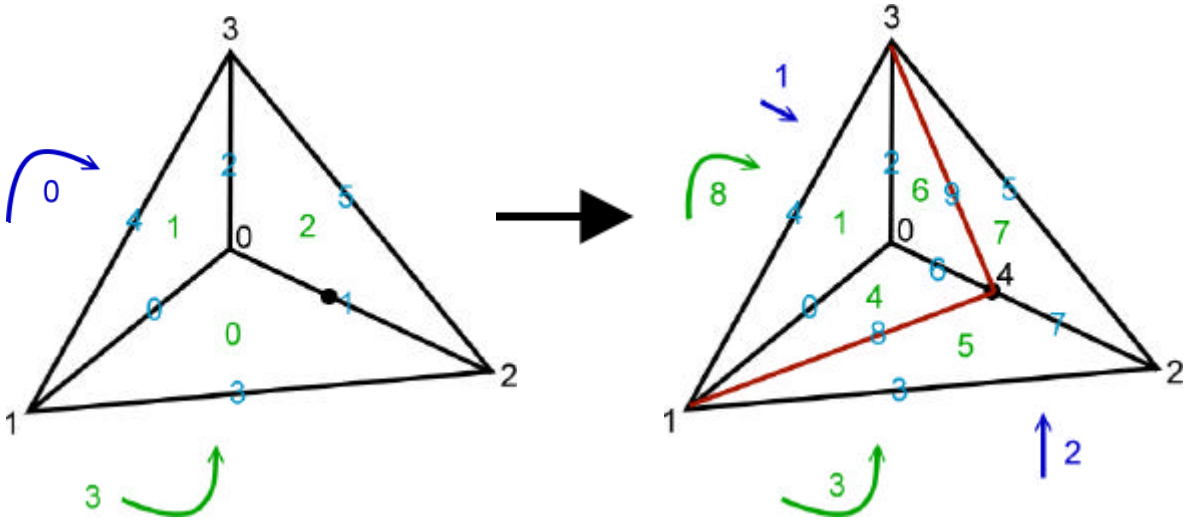


Figura 4.10: Caso 1. La cara 3 está formada por los puntos 1,2 y 3. La cara 8 está formada por los puntos 1,3 y 4.

Caso 2: La arista 2 tiene un punto de Steiner. Ver figura 4.11

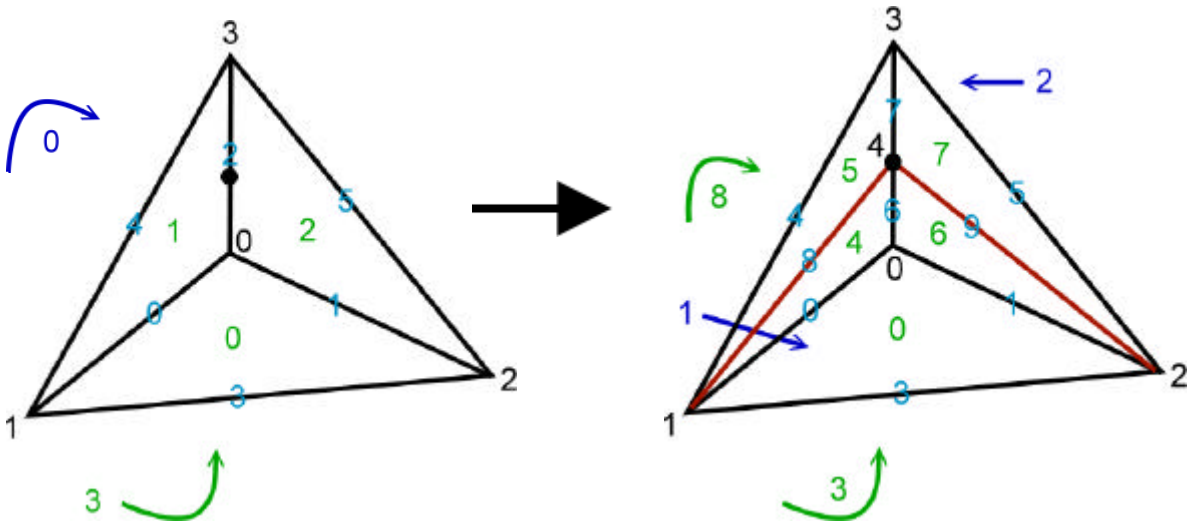


Figura 4.11: Caso 2. La cara 3 está formada por los puntos 1,2 y 3. La cara 8 está formada por los puntos 1,2 y 4.

Caso 3: La arista 3 tiene un punto de Steiner. Ver figura 4.12

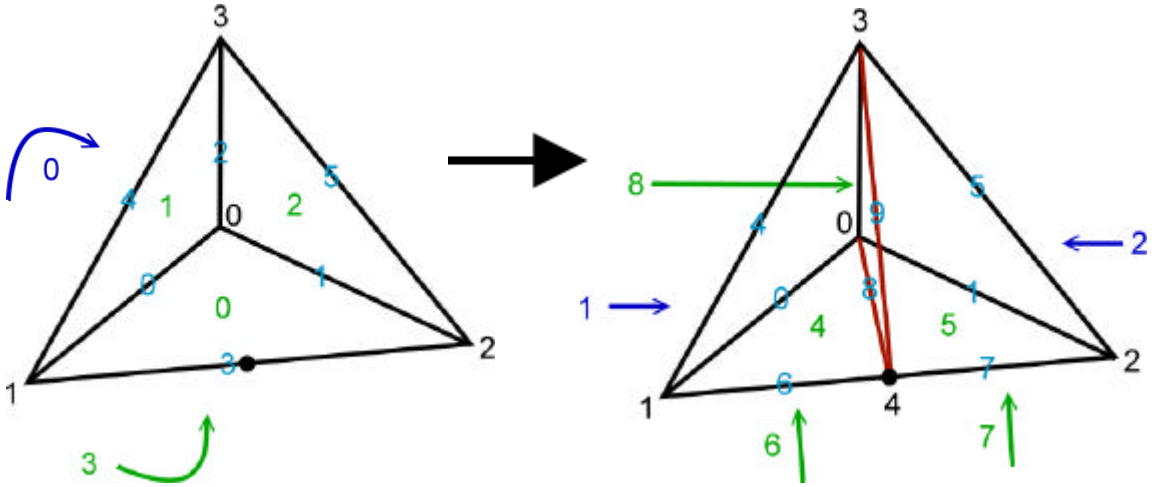


Figura 4.12: Caso 3. La cara 6 está formada por los puntos 1, 3 y 4. La cara 7 está formada por los puntos 2, 3 y 4. La cara 8 está formada por los puntos 0, 3 y 4.

Caso 4: La arista 4 tiene un punto de Steiner. Ver figura 4.13

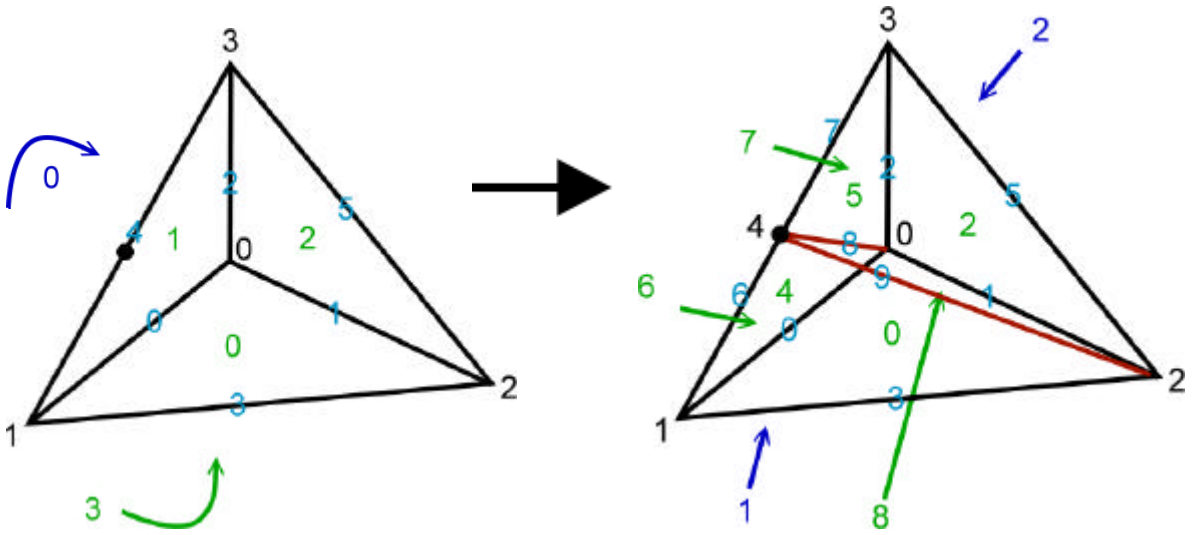


Figura 4.13: Caso 4. La cara 6 está formada por los puntos 0,1 y 4. La cara 7 está formada por los puntos 0, 3 y 4. La cara 8 está formada por los puntos 0, 2 y 4.

Caso 5: La arista 5 tiene un punto de Steiner. Ver figura 4.14

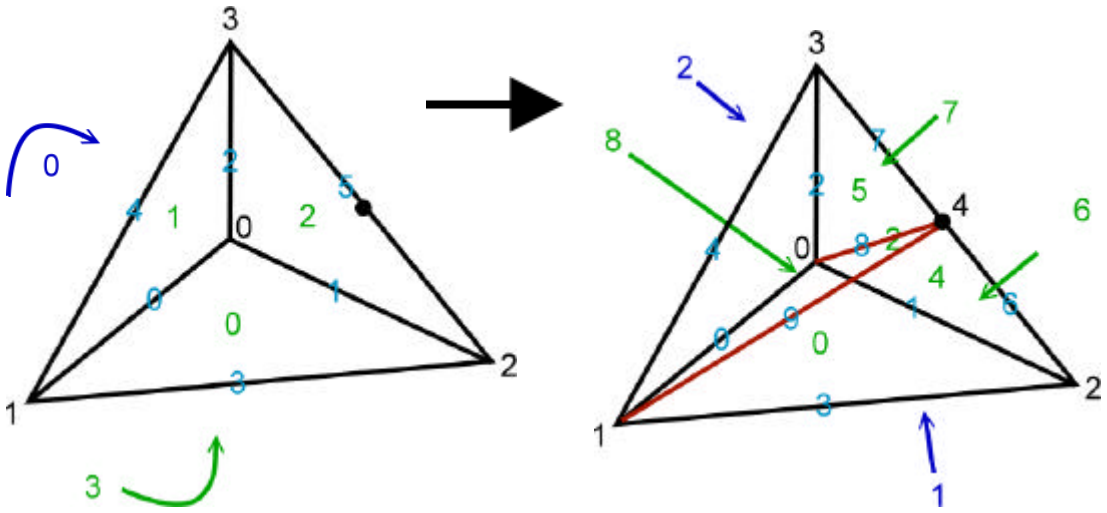


Figura 4.14: Caso 5. La cara 6 está formada por los puntos 1, 2 y 4. La cara 7 está formada por los puntos 1, 3 y 4. La cara 8 está formada por los puntos 0, 1 y 4.

Todo lo mostrado anteriormente con respecto al particionamiento de un tetraedro de acuerdo a la ubicación de un punto de Steiner se puede resumir en la tabla que se muestra en la figura 4.15, en la cual cada línea representa un caso y a la derecha se muestran los dos tetraedros que son creados con la información de sus índices reales guardados en su estructura interna en la posición indicada en número pequeño.

	Tetraedro 1														Tetraedro 2											
	Puntos				Aristas						Caras				Puntos				Aristas						Caras			
Caso	0	1	2	3	0	1	2	3	4	5	0	1	2	3	0	1	2	3	0	1	2	3	4	5	0	1	2	3
0	0	4	2	3	6	1	2	8	9	5	4	6	2	8	4	1	2	3	7	8	9	3	4	5	5	7	8	3
1	0	1	4	3	0	6	2	8	4	9	4	1	6	8	4	1	2	3	8	7	9	3	4	5	5	8	7	3
2	0	1	2	4	0	1	6	3	8	9	0	4	6	8	4	1	2	3	8	9	7	3	4	5	8	5	7	3
3	0	1	4	3	0	8	2	6	4	9	4	1	8	6	0	4	2	3	8	1	2	7	9	5	5	8	2	7
4	0	1	2	4	0	1	8	3	6	9	0	4	8	6	0	4	2	3	8	1	2	9	7	5	8	5	2	7
5	0	1	2	4	0	1	8	3	9	6	0	8	4	6	0	1	4	3	0	8	2	9	4	7	8	1	5	7

Figura 5.15: Resumen de casos de particionamiento de tetraedros.

4.2.2.4 Algoritmo de Mejoramiento

Dada una teselación de tetraedros, el algoritmo de mejoramiento busca optimizar esta teselación utilizando algún algoritmo para ello. En esta implementación se eligió el Test de la Esfera, el cual asegura que la teselación cumplirá con la condición de Delaunay. Dado que se sigue el patrón de diseño Strategy, el incluir otro algoritmo de mejoramiento de la teselación no traería mayores complicaciones.

Para realizar el cálculo del test de la esfera, se utilizan los métodos de cálculo de test de la esfera y el test de la orientación sugeridas por J. Shewchuk [7]. Para ello, se utilizan los determinantes que las definen. Ante problemas de robustez, debido al uso de aritmética de punto flotante, se usa una tolerancia *epsilon*.

Los determinantes usados son:

$$Orient3D(a,b,c,d)=\begin{vmatrix} a_x & a_y & a_z & 1 \\ b_x & b_y & b_z & 1 \\ c_x & c_y & c_z & 1 \\ d_x & d_y & d_z & 1 \end{vmatrix}$$

$$InSphere(a,b,c,d,e)=\begin{vmatrix} a_x & a_y & a_z & a_x^2+a_y^2+a_z^2 & 1 \\ b_x & b_y & b_z & b_x^2+b_y^2+b_z^2 & 1 \\ c_x & c_y & c_z & c_x^2+c_y^2+c_z^2 & 1 \\ d_x & d_y & d_z & d_x^2+d_y^2+d_z^2 & 1 \\ e_x & e_y & e_z & e_x^2+e_y^2+e_z^2 & 1 \end{vmatrix}$$

El determinante *Orient3D* verifica la orientación de los puntos *a, b, c* respecto del punto *d*. Si el resultado es positivo, los puntos están orientados en contra del sentido de los punteros del reloj mirando al punto *d* debajo de la cara formada por *a, b* y *c*. Si el valor es negativo, los puntos están orientados en el sentido de los punteros del reloj. Si es igual a cero los puntos son coplanares. La figura 4.16 muestra dos test de orientación. El punto *d* se encuentra debajo del plano formado por *a, b* y *c*, (imaginarse mirar a través del triángulo y ver el punto *d* más lejos). A la izquierda el test es positivo, pues el triángulo formado por *a, b* y *c* está orientado en contra

de los sentidos del reloj. En el caso de la derecha el test de orientación da negativo, pues el triangulo está orientado en el sentido de los punteros del reloj.

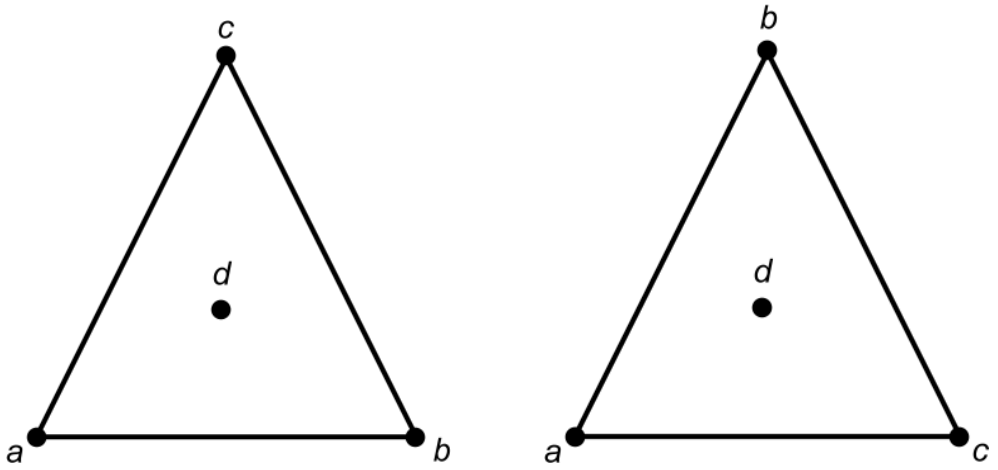


Figura 4.16: Los puntos de la izquierda dan un test de orientación positivo.
Los puntos de la derecha dan un test de orientación negativo.

El determinante *InSphere* verifica el test de la esfera de un tetraedro formado por los puntos *a, b, c, d* respecto del punto *e* (ver 2.1.8). Si consideramos que los puntos del tetraedro están orientados en contra del sentido de los punteros del relo j, el resultado es positivo si el punto *e* está dentro de la esfera, y negativo si *e* está fuera de la esfera. En caso que el resultado sea cero, el punto *e* está sobre la esfera que contiene al tetraedro. Si la orientación de *a,b,c,d* es negativa y el test de la esfera es negativo, entonces el punto *e* está dentro de la esfera, en caso contrario está fuera. En resumen:

- *orient3D(a,b,c,d) * inSphere(a,b,c,d,e) > 0* entonces el punto *e* está dentro de la esfera.
- *orient3D(a,b,c,d) * inSphere(a,b,c,d,e) < 0* entonces el punto *e* está fuera de la esfera.
- *orient3D(a,b,c,d) * inSphere(a,b,c,d,e) = 0* entonces el punto *e* está sobre la esfera.

Sea *abc* una cara interior en una triangulación T en 3D. Entonces *abc* incide con dos tetraedros *abcd* y *abce* [6]. Se asigna a la cara *abc* el tipo de la forma T*rs* o N*rs* donde T significa que puede ser transformable, N no transformable, y los posibles tipos (o configuraciones) son T23, T32, N32, T22, T44, N44, N30 y N20. Excepto para el caso $r = s = 4$, *r* es el número de tetraedros en la triangulación de *a, b, c, d, e* que contiene *abcd* y *abce,* y *s* es 0 si la configuración sólo tiene una posible triangulación, ó *s* es el número de tetraedros en otra

configuración de *a, b, c, d, e* que resulta de una transformación local. Los tipos T44 y N44 envuelven a seis vértices y un par de transformaciones.

En la figura 4.17 se muestran las distintas configuraciones descritas en el párrafo anterior. Es importante notar que el tipo T44 se forma según la configuración 3A o 3B siempre y cuando la cara que comparte puntos coplanares tenga a dos tetraedros para el otro lado compartiendo un vértice. Se aprecia que las configuraciones 1A, 1B, 3A y 3B son las que posiblemente podrían transformarse. 1A pasaría a la forma 1B, 1B a la forma 1A, 3A a 3B y 3B a 3A. Todo esto según el test de la esfera. Los tipos T44 son análogos a T22. En color rojo se muestra el tipo según la configuración.

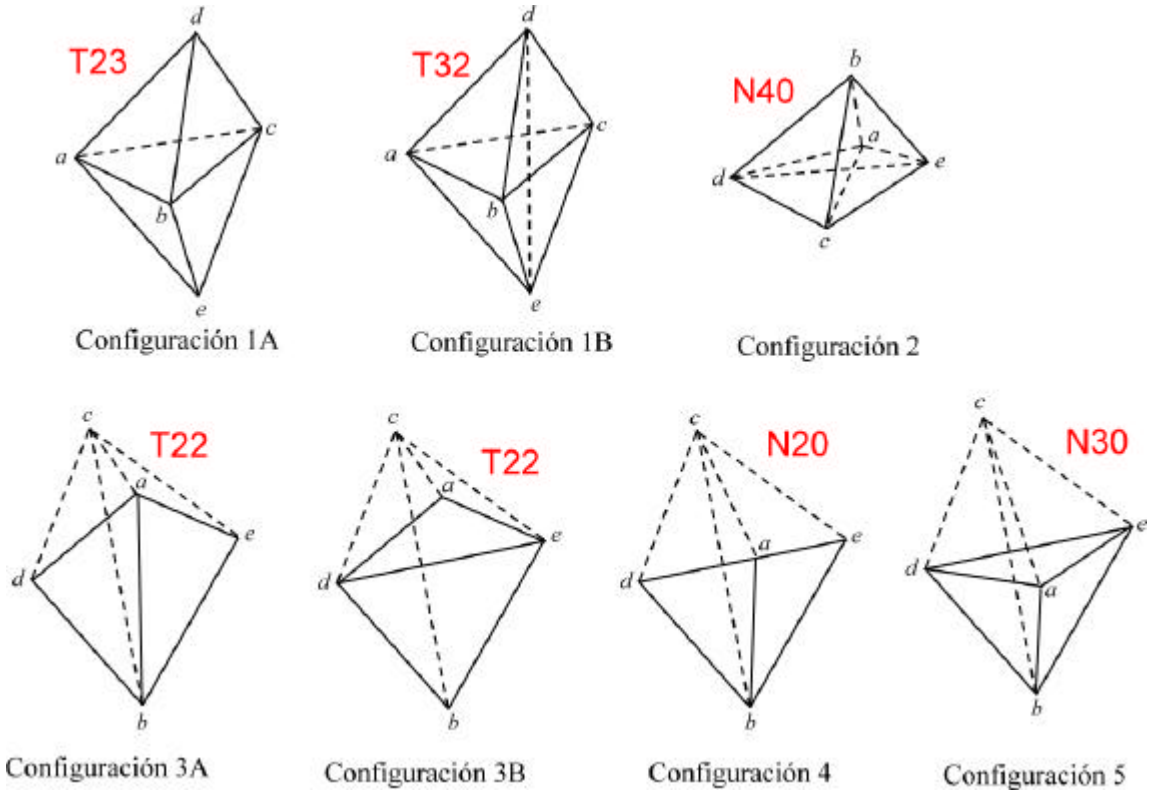


Figura 4.17: Los tetraedros son (1A) *abcd, abce*; (1B) *abde, acde, bcde*; (2) *abcd, abce, abde, acde*; (3A) *abcd, abce*; (3B) *acde, bcde*; (4) *abcd, abce*; (5) *abcd, abce, acde.* En los tetraedros de la segunda fila, los puntos *a, b, d, e* son coplanares (aristas que inciden con *c* están con líneas punteadas). En la configuración 4, *a, d, e* son colineales.

Una cara no transformable *abc* puede surgir debido a la presencia de caras extras que inciden en una arista. Para el caso N32, es cuando el tetraedro *bcde* no existe dentro de la malla y para el caso N44 cuando la caras que comparten puntos coplanares tienen más de un punto en común por el otro lado. Las configuraciones N40, N30 y N20 tienen sólo una triangulación posible para los puntos *a,b,c,d* y *e*, y son las que se muestran en la figura 4.17.

En [6] se presenta un algoritmo para mejorar una triangulación dada, usando otras medidas de calidad, pero adaptable perfectamente al test de la esfera. La base de este algoritmo es usado para crear el algoritmo de mejoramiento de la teselación y se presenta a continuación:

```
Algoritmo de mejoramiento de una teselación.
#Input: Teselacion T
#Output: Teselación mejorada
        Poner todas las caras interiors de T en una cola, fijar front y back
        como punteros al primer y al ultimo elemento de la cola
    flip(T,front,back)

flip(T,front,back)
#Input y output: Triangulation T, punteros de la cola front y back
{
        while front ? null
                abc := primer elemento de la cola de caras
                front := próxima cara en la cola
                if abc está en teselación
                        if abc tiene tipo T23, T32, T22, or T44 then
                                if InSphere > 0 con respecto a abc y tetraedro(s) vecinos
                                        Aplicar la transformación local apropiada al caso y
                                        actualizar la información de tetraedros, vértices
                                        caras y aristas. Agregar las caras internas nuevas en la cola.
                                endif
                        endif
                endif
        endwhile
}
```

Un importante paso en el algoritmo descrito anteriormente es analizar las caras internas y con ellas encontrar un tipo válido que pueda ser transformado (T23, T32, T22, T44). Esta operación se resolvió gracias al test de orientación de la siguiente forma. Dado que una cara interna siempre comparte sólo dos tetraedros, se elige un tetraedro como base y se analiza el punto opuesto a la cara del segundo tetraedro. Según la ubicación de este punto con respecto al

primer tetraedro podemos encontrar su configuración. La configuración puede ser cualquiera de las que se muestran en la figura 4.17 y si alguna coincide con las que se pueden transformar localmente, se aplica el test de la esfera.

Sea *abc* la cara interna que se está evaluando y sean los tetraedros *abcd* y *abce*. Sea *abcd* el tetraedro base y sea *e* el punto opuesto a la cara *abc*, punto que pertenece al tetraedro *abce*. Si nos situamos en el tetraedro base (*abcd*), encontramos que *a* es el punto opuesto a la cara *bcd*, *b* es el punto opuesto a la cara *acd*, *c* es el punto opuesto a la cara *abd*, y *d* es el punto opuesto a la cara *abc*. Con esta información calculamos la orientación de las caras *abd*, *acd* y *bcd* con respecto a sus puntos opuestos. Esta orientación podría ser negativa o positiva dependiendo del orden de los puntos. Posteriormente se calcula la orientación de las mismas caras (*abd*, *acd* y *bcd*) con respecto al punto *e*. Ahora se comparan las orientaciones de cada cara con su punto opuesto en el tetraedro base y con el punto *e*. Si las orientaciones coinciden en signo significa que ambos puntos (el opuesto a la cara) y el punto a analizar (*e*) están en el mismo lado del plano formado por la cara en cuestión. Si difieren en signo, un punto está a un lado de la cara y el otro punto está al otro lado. En la figura 4.18 se aprecian los casos al calcular las orientaciones con respecto a una cara. En la izquierda se ve que ambos puntos, *b* y *e* se encuentran al mismo lado del plano formado por los puntos *acd*, por lo tanto sus orientaciones tienen el mismo signo. En la derecha se aprecia que los puntos se encuentran uno a cada lado del plano, por lo tanto las orientaciones difieren en signo, pese a que ambas configuraciones son válidas y dos tetraedros comparten la cara *abc*.

Con las orientaciones explicadas claramente, nos podemos dedicar a encontrar los tipos T23, T32, T22 y T44 y analizar sus test de esfera.

- T23: Dada la cara *abc* y el tetraedro base *abcd*. Los tetraedros *abcd* con *abce* forman un tipo T23 cuando las otras caras del tetraedro base (esto es exceptuando *abc*) tienen la misma orientación 3D con su punto opuesto y con el punto *e*. Se realiza el test de la esfera para *abcde* con *e* el punto a evaluar. Si el test de la esfera retorna *falso*, se agrega la arista *de* a la teselación y se forman tres nuevos tetraedros en lugar de los dos anteriores. Gráficamente se pasaría de la configuración 1A a la configuración 1B (Ver figura 4.17).

- T32: Dada la cara *abc* y el tetraedro base *abcd.* Los tetraedros *abcd* y *abce* forman una configuración de tipo T32 cuando calculadas las orientaciones de las otras caras del tetraedro base (esto es exceptuando *abc*) con sus puntos opuestos y con el punto *e* resulta que hay sólo una cara que tiene orientaciones distintas con respecto a su punto opuesto y a *e* y además si la arista formada por *d* y *e* existe en la teselación. (Ver configuración 1B en figura 4.17, en ese caso la cara base es *bed* y el tetraedro base es *bdea*, el elemento opuesto es *bdec*, y si la arista *ac* existe, entonces tenemos el caso T32 . Si la arista *ac* no existe, significa que hay más de un tetraedro ocupando el espacio que falta). El test de la esfera, mirando la configuración 1B de la figura 4.17, se realiza llamando a *inSphere(a,b,c,d,e)* y si este test es *verdadero*, se reemplazan los tres tetraedros por dos, esto es eliminando la arista interior. Gráficamente es pasar de la configuración 1B a la configuración 1A.

- T22: Este caso ocurre sólo en los bordes y ocurre cuando un test de orientación da como resultado cero, esto es decir, hay 4 puntos coplanares. Esto ocurre en los bordes ya que si fuera en un sector interno de la teselación, habrían más de 2 tetraedros analizados. Aunque hay que aclarar que estamos analizando una cara interna. En la figura 4.17, configuración 3A, la cara base es claramente *abc*. En este caso el test de la esfera es *inSphere(a,b,c,d,e)* y si es *verdadero* se intercambia la diagonal *ab* por *de* formando dos nuevos tetraedros en lugar de los dos anteriores. Gráficamente es pasar de la configuración 3A a 3B.

- T44: Este caso ocurre cuando existe un test de orientación 3D que da cero en el sector interno de la teselación. Si para ambos lados de las caras que juntas son coplanares hay un punto que une a dos tetraedros por lado, estamos hablando del caso T44. En otro caso se continuaría buscando. El test de la esfera se realiza en forma independiente por cada lado. El proceso es igual que en T22, y si ambos test de la esfera son *falsos*, entonces se intercambia la diagonal que forman parte del sector coplanar.

Las otras configuraciones no son buscadas, ya que no se buscan resolver.

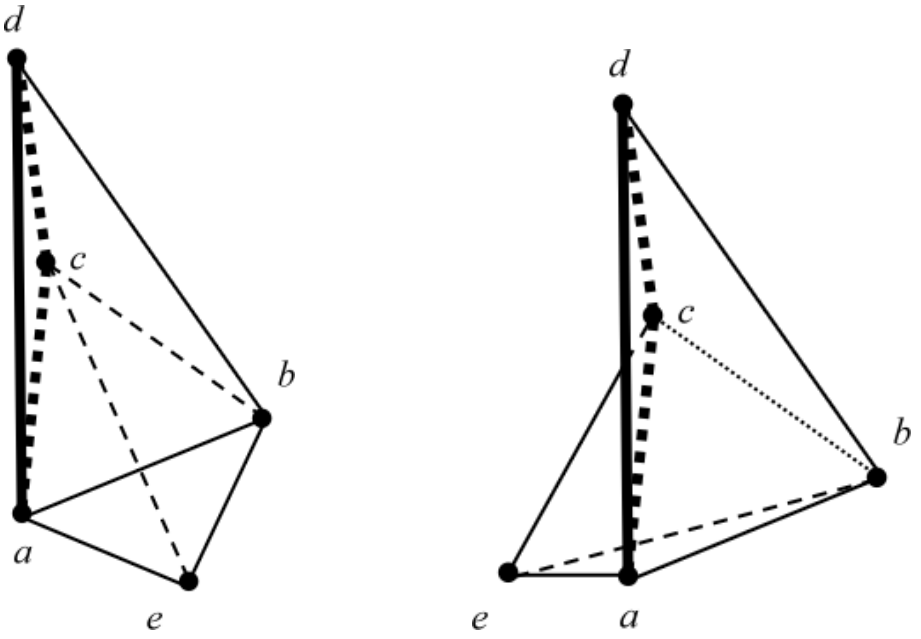


Figura 4.18: Distintos test de orientación para una cara compartida por 2 tetraedros. A la izquierda los puntos *b* y *e* son opuestos a la cara *acd* y están ubicados al mismo lado del plano formado por esa cara, por lo tanto sus orientaciones 3D tienen el mismo signo (en este caso es negativo). En cambio, a la derecha, aunque los puntos *b* y *e* son opuestos a la cara *acd*, ellos están uno a cada lado del plano formado por *acd* y tienen distinto signo en sus orientaciones 3D (*acdb* es negativa y *acde* positiva).

4.2.3 Diagrama de Clases

La estructura del diagrama de clases soporta la aplicación de los distintos algoritmos para generar la malla inicial, particionarla, y luego refinarla. La base está en la clase Tessellation, la cual contiene la información de toda la malla dada la configuración inicial. Todo el proceso parte cuando se aplica un patrón a un elemento (cuboide en este caso), y su clase (brick_1_irregular) se asocia a su teselación dado un patrón. Ver figura 4.19.

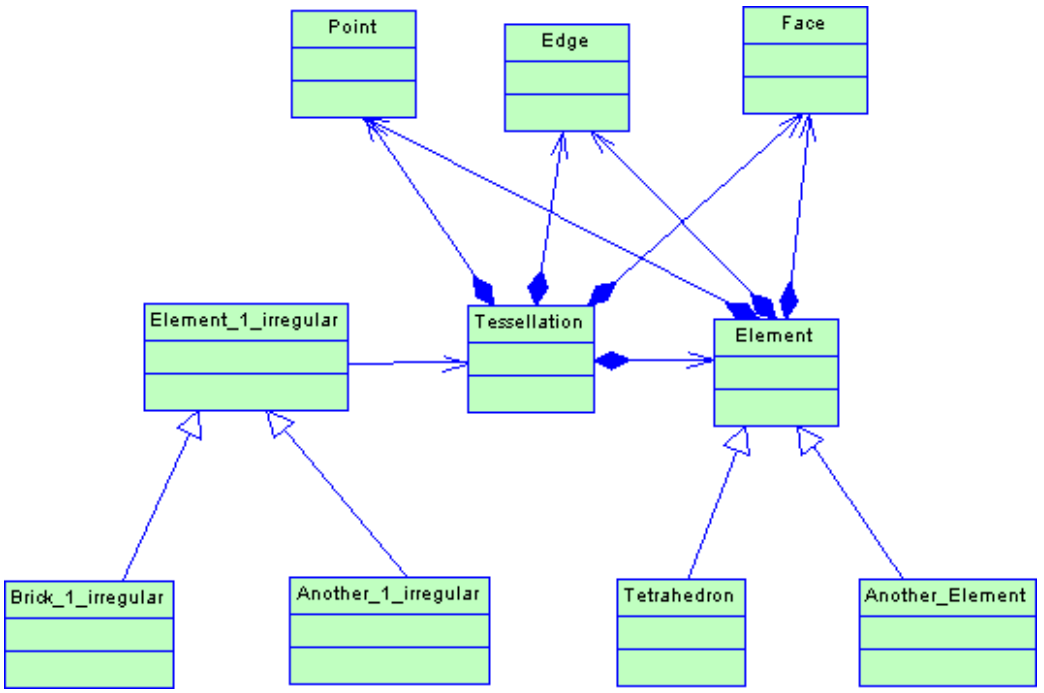


Figura 4.19: Diagrama de Clases Tessellation

El detalle de las clases de este diagrama se muestra en los Anexos en la sección 8.1.

A continuación se describirán las clases que pertenecen al modelo de la figura 4.19.

4.2.3.1 Clase Point

La clase Point es la base para construir los distintos objetos que forman parte de la malla. Con Point se crean las aristas y las caras, y con todo esto se generan los elementos, y sumado a los puntos de Steiner se puede particionar la malla en tetraedros.

4.2.3.2 Clase Edge

La clase Edge corresponde a una arista, la cual está formada por dos puntos (Point). Con esta clase es importante para aplicar el test de la esfera y así poder refinar la malla.

4.2.3.3 Clase Face

La clase Face se encarga de la descripción de una cara. En este trabajo, las caras son triángulos, pero la definición de esta clase permite incluir caras de más de tres puntos.

4.2.3.4 Clase Element

La clase Element corresponde a la representación de un elemento.

4.2.3.5 Clase Tetrahedron

La clase Tetrahedron, corresponde a la descripción de la figura tetraedro. Esta es una clase hija de la clase Element, por lo tanto hereda sus variables de instancia y sus métodos.

4.2.3.6 Clase Element_1_irregular

La clase Element_1_irregular corresponde a la representación de un elemento 1-irregular genérico. Esta clase es la clase padre de los elementos 1-irregular específicos, como por ejemplo de los cuboide 1-irregular, prismas 1-irregular, etc.

4.2.3.7 Clase Brick_1_irregular

La clase Brick_1_irregular es la encargada de representar un cuboide 1-irregular. A partir de esta clase se genera la teselación.

4.2.3.8 Clase Tessellation

La clase Tessellation es la más importante dentro del conjunto de clases. En ella se almacena toda la información de la malla de tetraedros asociada a un cuboide 1-irregular.

Capítulo 5

Resultados

Antes de entregar los resultados, se debe definir el concepto de patrón como la representación binaria de un número decimal que se registra en una `bit_table`, y de la cual se deducen las aristas con puntos 1-irregular. Esto quiere decir que por ejemplo el patrón 70688 cuyo número es igual a la suma de $2^5 + 2^{10} + 2^{12} + 2^{16}$ tiene puntos irregulares en las aristas 5, 10, 12 y 16 según la numeración entregada para un malla inicial de un cuboide con diagonales (ver figura 5.1). Con esto aclarado, los patrones se pueden identificar como un número entero, del cual se pueden deducir sus aristas con punto 1-irregular, y los cuales se usan para probar los algoritmos.

5.1 Etapa de Particionamiento Según Puntos de Steiner

El algoritmo de particionamiento se probó para todas las configuraciones ($2^{18} - 1 = 262143$ patrones), desde la más simple hasta la más compleja, todo esto en un cuboide de lado 1 y además para un paralelepípedo de ancho 1, largo 2 y alto 4. Se eligieron algunos casos de prueba para mostrar en este informe, incluida la generación de la malla inicial.

La generación de la malla inicial para un cuboide, usando la convención de numeración de aristas, queda según la figura 5.1. Esta configuración es la misma que se usa en un paralelepípedo regular, ya que en estas figuras sólo cambian las longitudes de los lados.

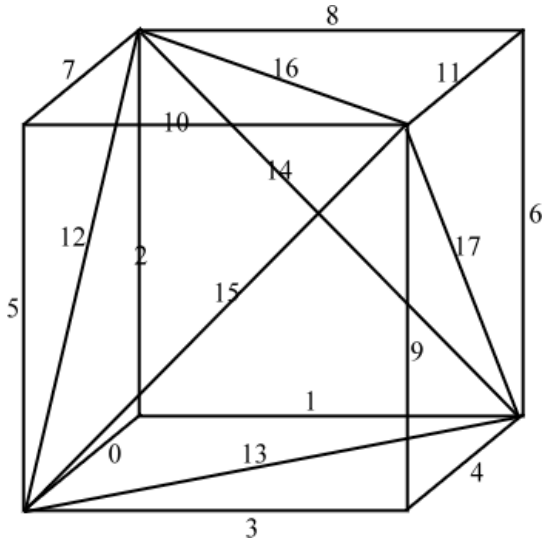


Figura 5.1: Malla inicial y su numeración en un cuboide de lado 1.

A continuación se mostrarán la configuración con puntos de Steiner a la izquierda y el particionamiento formado para un cuboide en el centro y para un paralelepípedo a la derecha:

- **Aristas 1-irregular no compartiendo elemento:** Se probó la configuración con aristas con puntos 1-irregular, las cuales no comparten ningún elemento de la malla base. El particionamiento se muestra en la figura 5.2. A la izquierda está la configuración, al centro el particionamiento para un cuboide y a la derecha para un paralelepípedo.

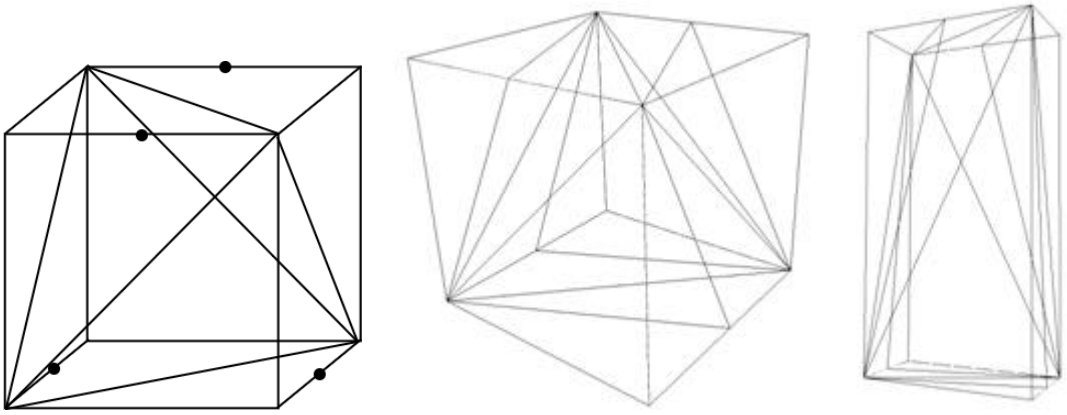


Figura 5.2: Particionamiento con puntos 1-irregular en las aristas 0, 4, 8 y 10.

- **Algunas aristas compartiendo elementos**: Se probó para la configuración mostrada en la figura 5.3 a la izquierda. Las aristas 1-irregular no son diagonales y algunas comparten un tetraedro. Los resultados del particionamiento se muestran al centro para el cuboide y a la derecha para el paralelepípedo con la misma configuración de puntos.

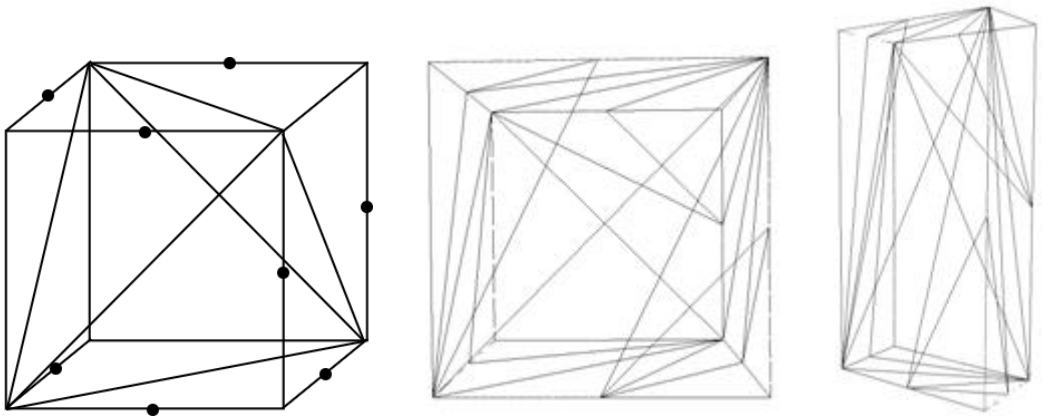


Figura 5.3: Particionamiento con puntos 1-irregular en las aristas 0, 3, 4, 6, 7, 8, 9 y 10.

- **Arista diagonal**: Se fijó la configuración con un punto 1-irregular justo en la primera diagonal según la numeración, y según como se muestra en la figura 5.4. El particionamiento se muestra al centro para el cuboide y a la derecha para el paralelepípedo. Este es el particionamiento más simple para una diagonal.

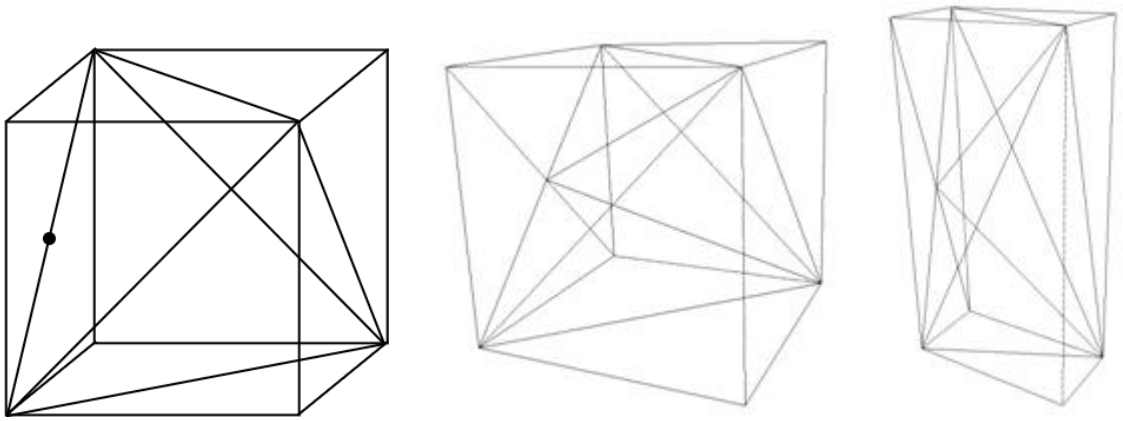


Figura 5.4: Particionamiento con puntos 1-irregular en la arista 12.

- **Arista diagonal compartiendo elementos**: Según se ve en la figura 5.5 a la izquierda, la configuración mostrada corresponde a varias aristas 1-irregular, que comparten elementos

y además los elementos, a la vez, comparten una diagonal. El resultado del particionamiento se muestra en el centro para un cuboide y a la derecha para un paralelepípedo.

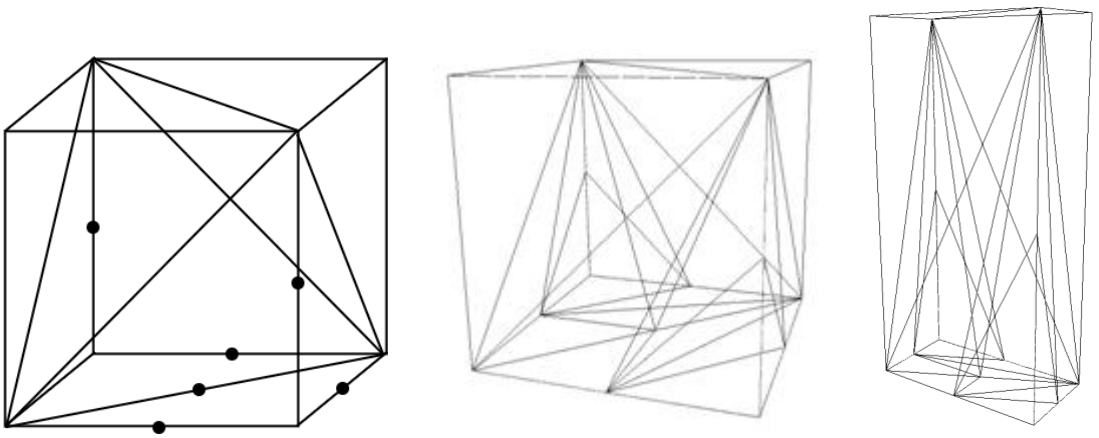


Figura 5.5: Particionamiento con puntos 1-irregular en las aristas 0, 1, 2, 3, 4, 9 y 13.

- **Sólo aristas diagonales**: La configuración que se muestra en la figura 5.6 a la izquierda, corresponde a una configuración en donde todas las aristas diagonales tienen un punto 1-irregular en su mitad. El resultado del particionamiento se muestra al centro para un cuboide y a la derecha para un paralelepípedo.

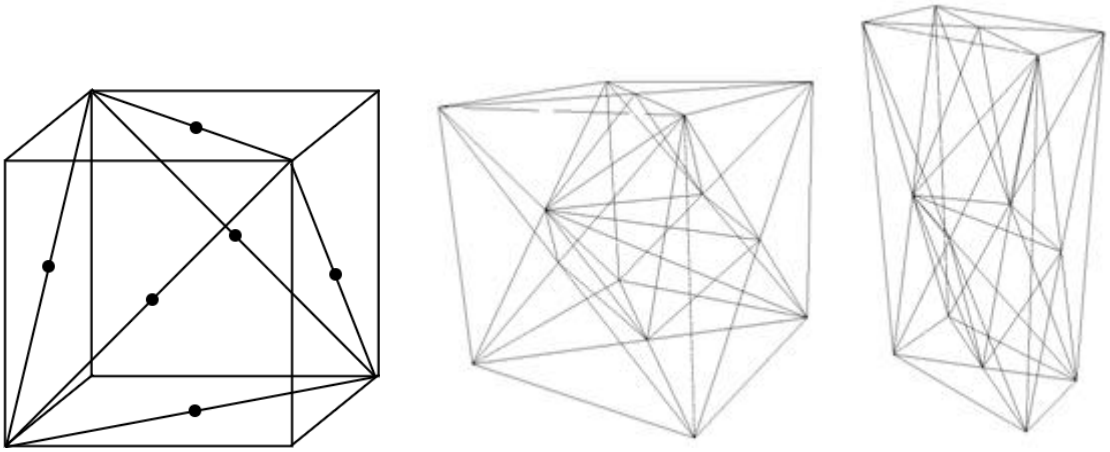


Figura 5.6: Particionamiento con puntos 1-irregular en las aristas diagonales 12, 13, 14, 15, 16 y 17.

- **Todas las aristas**: La figura 5.7 a la izquierda muestra la configuración para el último patrón (el 262143), el que corresponde a todas las aristas con un punto 1-irregular en su

mitad. Este es el patrón que realiza más cálculos y comparte más información para obtener su particionamiento. El resultado se muestra en la figura al centro para un cuboide y en la derecha para un paralelepípedo.

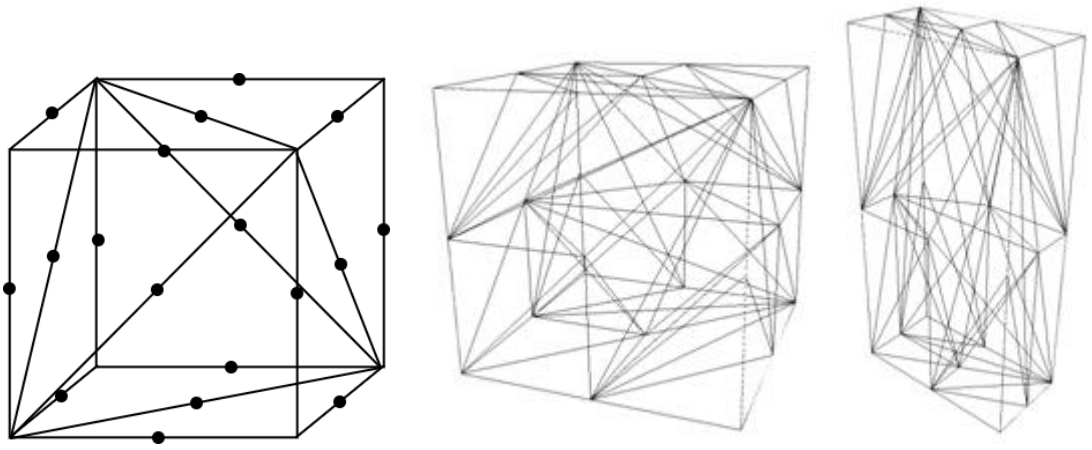


Figura 5.7: Particionamiento con puntos 1-irregular en todas las aristas.

5.2 Etapa de Mejoramiento de la Teselación

Posterior a la etapa de particionamiento del cuboide 1-irregular, se procedió a analizar cada resultado con respecto a sus posibles transformaciones locales que las convertirían en una telesación de Delaunay. Esto es, identificando sus configuraciones internas según el algoritmo mostrado en la descripción del algoritmo de mejoramiento.

Para dar un inicio al análisis se debe mencionar que dado que las configuraciones T22 siempre están en el borde del cuboide, y cada cara de un cuboide tiene un vecino que tiene su partición propia. Si los tetraedros de una configuración T22 retornan *falso* su test de la esfera, lamentablemente no se puede realizar una transformación local en este caso, ya que un cambio en su información afecta directamente al elemento vecino. Es así como se han logrado identificar otras configuraciones transformables, las cuales se pasan a describir a continuación.

Analizando todos los patrones, se puede obtener la siguiente información para las pruebas realizadas sobre un cuboide 1-irregular:

- El 50% tiene a lo menos una configuración T22, las cuales se distribuyen casi uniformemente entre los primeros patrones hasta los últimos.

- El 12,6% tiene a lo menos una configuración T23, la cual conduce a una transformación local. Estas configuraciones se concentran en el intervalo (patrón 60.000 hasta patrón 120.000), y el primer patrón que contiene esta configuración es el 12304 (aristas 4, 12 y 13 con punto1-irregular), con lo cual se aprecia claramente que sólo con diagonales 1-irregular podría encontrarse este tipo de transformación.

- El 41,4% tiene a lo menos una configuración T44. Esta configuración se concentra mayormente en los patrones de mayor número, es decir va en aumento según aumenta el número de aristas 1-irregular. Cabe destacar que el primer patrón encontrado con esta configuración es el 70688 (aristas 5, 10, 12, 16), y nuevamente se aprecia que sólo con puntos de Steiner en diagonales se podría encontrar este tipo de transformación.

- Dado lo anterior, se puede confirmar que más de la mitad de los patrones admiten algún tipo de transformación local que la llevaría a ser una malla tipo Delaunay.

- El 27,5% de todos los patrones entrega un particionamiento de Delaunay en forma primitiva, es decir, no se le detectan ningún tipo de transformación local que rechace el test de la esfera. Por lo tanto, el 72,5% encuentra algún tipo de configuración cuyo test de la esfera es falso (incluyendo los casos T22 que no se pueden transformar).

- Sobre los patrones que entregan una partición de Delaunay, el 75% de ellos se encuentra antes de la primera mitad del conjunto de patrones, es decir, antes del patrón 131070, y va disminuyendo su frecuencia a medida que aumenta el número de patrón. Esto quiere decir que mientras más aristas tienen puntos 1-irregular, menos probable es que el particionamiento entregado por el algoritmo sea una malla Delaunay.

Los resultados para un paralelepípedo 1-irregular son muy similares a los del cuboide. La analogía es la misma y sólo cambian un poco los porcentajes, esto es debido a la longitud de los lados, las cuales influyen en el momento de realizar los cálculos. En resumen:

- Al igual que para un cuboide, el 50% tiene al lo menos una configuración T22 aplicando el algoritmo en un paralelepípedo de lados 1,2 y 4.

- El 26,4% tiene a lo menos una configuración T23. Esto es un poco más del doble a la obtenida en un cuboide 1-irregular (12,6%).

- El 46,2% tiene a lo menos una configuración T44. Esto es sólo un 3% más que para el caso del cuboide.

- El 19,5% de todos los patrones en este paralelepípedo entrega un particionamiento de Delaunay en forma primitiva. Esto es un 8% menos que para el caso del cuboide 1-irregular (27,5%).

- Dado lo anterior, más de dos tercias partes de los patrones admiten algún tipo de transformación local para este paralelepípedo, lo cual lo llevaría a ser una malla tipo Delaunay.

- Se puede deducir que al aplicar el algoritmo de particionamiento sobre un paralelepípedo de lados de longitud distinta, el número de transformaciones aumentaría en relación a la diferencia entre sus lados y esto implicaría que las configuraciones con un particionamiento de Delaunay en forma primitiva (sin necesitar transformaciones locales) disminuirían según la misma condición.

Cabe mencionar que el proceso de pruebas se realizó en pequeños intervalos de datos a la vez, registrando los resultados para una vez concluida esta etapa, analizarlos. Esto debido a una combinación de escasez de hardware requerido, por ejemplo en memorial principal, velocidad de procesamiento y memoria secundaria. Aun así, esta limitante no fue obstáculo para que la ejecución del software programado fuera exitosa.

Capítulo 6

Conclusiones

En esta memoria se estudio y diseñó un modelo orientado a objetos de un generador de mallas mixtas. Además se implementó uno de los algoritmos de generación de malla final, el cual resuelve todas las configuraciones posibles de puntos irregulares sobre un cuboide.

El diagrama de clases sigue la filosofía del patrón de diseño Strategy en el mecanismo de selección de algoritmos. Los algoritmos que usa un generador de mallas son: generación de malla inicial, refinamiento, mejoramiento, y generación de malla final. Además se debe permitir el uso de distintos formatos tanto de entrada como de salida y se debe acceder a distintos criterios aplicados sobre una región para los procesos de toma de decisiones en el refinamiento o mejoramiento de la malla.

Una malla, luego del proceso de refinamiento y mejoramiento, queda con aristas que inciden en otras aristas de otros elementos al interior. Esto provoca que, si se considera cada elemento final como un objeto independiente, en una o más aristas se ven puntos aislados generalmente en su mitad. Esto es causa de una no perfecta intersección de un elemento con su vecino en alguna de sus aristas o caras. A esos elementos finales que tienen puntos en sus mitades (llamados puntos de Steiner) apunta el diseño e implementación del algoritmo presentado en esta memoria.

Esta memoria se enfocó sólo en los cuboides 1-irregulares (por extensión además se probó el algoritmo para un paralelepípedo), queda abierto para trabajos futuros la resolución de otros elementos irregulares, tales como prismas, pirámides o tetraedros, todos elementos básicos que pertenecen a la malla de elementos mixtos.

Dado que un cuboide 1-irregular se puede observar como una unidad independiente, es correcto trabajar con él sin mirar a sus vecinos. El algoritmo implementado en este trabajo de título recorre todas las aristas del cuboide (incluidas sus diagonales) en forma iterativa, y va

bisectando los tetraedros desde la malla inicial definida, hasta la última arista recorrida. El resultado es una malla de tetraedros. El proceso interno de este particionamiento se resuelve con templates para cada tetraedro definidos con antelación durante el estudio del problema usando sus seis vértices como base. Esto produce un aumento del rendimiento del algoritmo, ya que al detectar qué arista tiene el punto medio, inmediatamente se aplica una solución ya predefinida. Aun así, se puede aumentar significativamente el rendimiento de la implementación, eligiendo algunas estructuras de datos para ciertas decisiones o preguntas para encontrar por ejemplo una configuración. Es posible que utilizando tablas de hash, o más información redundante se podría aumentar la velocidad del procesamiento.

La etapa de mejoramiento de la partición utilizando la condición de Delaunay no resultó lo simple que se esperaba. Aunque la analogía entre Delaunay 2D y Delaunay 3D podría verse trivial, es mucho más complicado de lo que parece. En 2D es simple aplicar el test del círculo, no hay mucha información en un plano, y es por ello que el problema de la geometría computacional en 2D se considera resuelto. En 3D el número de configuraciones posibles para dos tetraedros vecinos es alto, hay que analizar sus inclinaciones, vecinos, ángulos y otras datos, hasta poder capturar una configuración que sea posible de resolver. Es por ello que la investigación apunta a 3D, y se revisó bastante bibliografía para encontrar una respuesta a los requerimientos de este trabajo de título. Es así como, basándose en un artículo, se decidió usar su algoritmo que recorre las caras internas y luego de detectar una configuración, si es posible transformarla, se transforma, actualizando todos los índices, para mantener correctamente la información y robustez de la teselación. Para encontrar las configuraciones transformables, al igual que en 2D, se usó el test de orientación, el cual resultó muy útil y simple usarlo, aunque no tan fácil de deducirlo.

Para la visualización de la teselación, se decidió el uso el programa opensource GeomView. Se implementó un método que genera el formato vectorial de una malla en 3D. Gracias a la visualización, se pudo comprobar que lo que uno se imaginaba en forma abstracta realmente funcionaba, incluso para patrones difíciles, y gracias a la visualización, se resolvieron muchos problemas encontrados durante el desarrollo. La visualización de una malla es primordial en este tipo de problemas, y tiene que tener prioridad alta desde el comienzo de un proyecto de generación de mallas.

Como se mencionó anteriormente, dado que está presentado el diseño orientado a objetos del generador, para el futuro se debe adaptar el generador anterior, y dada su extensibilidad y mantenibilidad, se pueden agregar algoritmos en forma simple cuando aparezcan nuevos requerimientos para las mallas en 3D. La extensión de otros elementos 1-irregular sería conveniente en el futuro, y también la resolución de configuraciones que tengan puntos irregulares en cualquier posición de una arista. Así el generador quedaría mucho más completo. Para extender las configuraciones 1-irregular para permitir puntos que no están en la mitad de una arista, bastaría una simple modificación, una de ellas (entre muchas que podrían existir), sería agregar un nuevo vector que almacene la posición del punto irregular y asociar este vector en relación uno a uno con el vector original de aristas. Otra opción sería incluir la posición del punto 1-irregular como una variable de instancia dentro de la clase `Edge`.

Para trabajar con otro tipo de elementos y permitir configuraciones 1-irregular en ellos para su posterior particionamiento, se debe trabajar con una generalización de la clase abstracta `Element_1_irregular` y definir sus templates, métodos, etc, similar a lo hecho con `Brick_1_irregular`.

Como conclusión final, se puede afirmar que se logró el objetivo de la memoria, se presentó un nuevo diseño para el generador de mallas mixtas cumpliendo los requerimientos, y se implementó un algoritmo que particiona cuboides 1-irregular con un diseño que permitirá resolver otros elementos 1-irregular.

Capítulo 7

Bibliografía

[1] N.Hitschfeld-Kahler, *Generation of mixed element meshes using a flexible refinement appro*ach. Engineering with Computers, 21(2):101-114, 2005.

[2] M.C. Bastarrica, N. Hitschfeld-Kahler, *Designing a product family of meshing tools. Advances in Engineering Software*, 17:1-10, 2006.

[3] Nancy Hitschfeld-Kahler, Carlos Lillo, Ana Cáceres, María Cecilia Bastarrica and María Cecilia Rivara, *Building a 3D Meshing Framework Using Good Software Engineering Practices*. Advanced Software Engineering: Expanding the Frontiers of Software Technology. IFIP 19th World Computer Congress, First International Workshop on Advanced Software Engineering, August 25, 2006, Santiago, Chile. Series: IFIP International Federation for Information Processing, Springer-verlag- 2006, pp:162-170.

[4] N. Hitschfeld and R. Farías: *1-irregular element tessellation in Mixed Element Meshes for the Control Volume Discretization Method.* Proceedings of the 5th Annual International Meshing Roundtable. October 1996, Pittsburgh, Pennsylvania, USA, pp: 195-204.

[5] N. Hitschfeld, G. Navarro, R. Farías. *1-irregular configurations in cuboids*, Actas del 9th Annual International Meshing Roundtable. 2-5 Octubre 2000, New Orleans, U.S.A, pp: 275-282.

[6] B. Joe, *Construction of three-dimensional improved-quality triangulations using local transformations*, SIAM J. Sci. Comput., 16, 1995, pp. 1292-1307.

[7] Ana Cáceres. *Algoritmo de Delaunay restringido basado en la cavidad por capas de Baker*. Memoria de Ingeniería Civil en Computación, Universidad de Chile, 2003.

[8] Francisco Moreno. *Generador de particiones de elementos 1-irregular*. Memoria de Ingeniería Civil en Computación, Universidad de Chile, 2000.

[9] Nancy Hitschfeld. *Grid Generación for Three-Dimensional Non-Rectangular Semiconductor Devices*. Series in microelectronics; Vol 21. Zürich, Eidgeöss. Hochsch, Diss., 1993.

[10] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. *Design Patterns: Element of Reusable Object Oriented Software*. Addison-Wesley, 1995.

Capítulo 8

Anexos

8.1 Detalle del Diagrama de Clases de la sección 4.2.3

El diagrama de clases que se presentó en la sección 4.2.3 se muestra en la figura 8.1

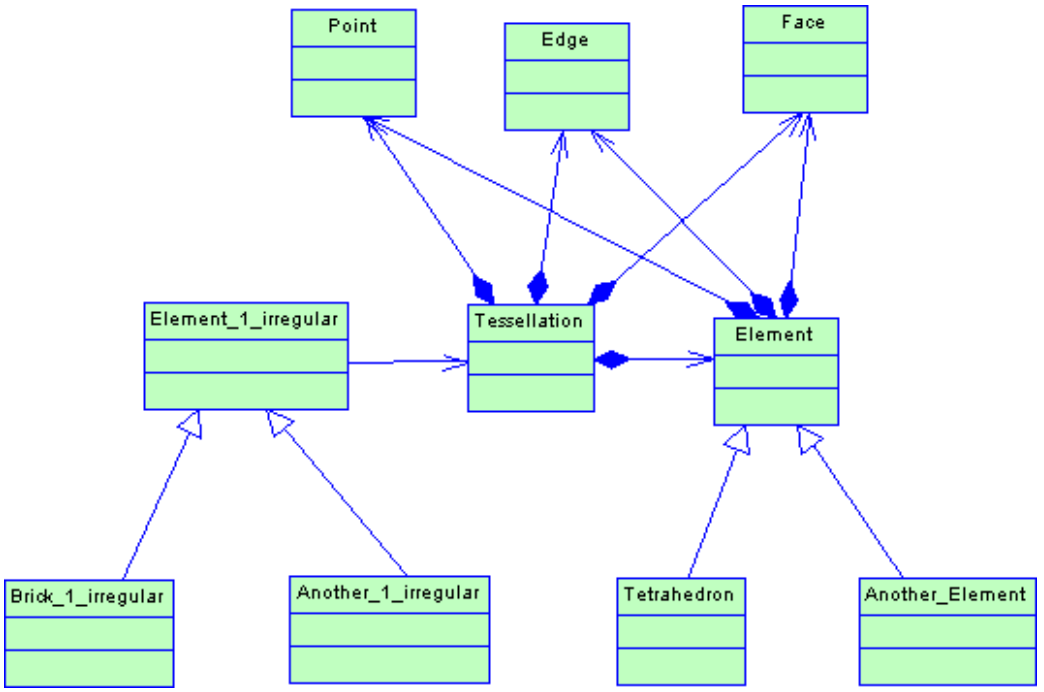
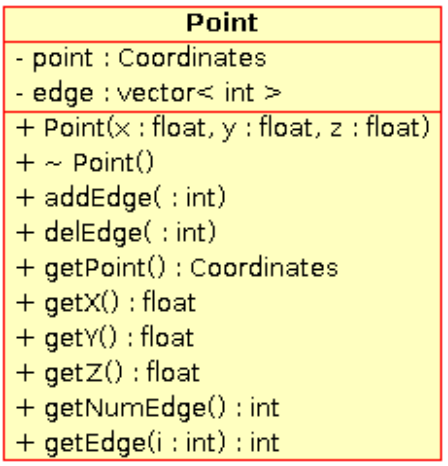


Figura 8.1: Diagrama de Clases Tessellation

A continuación se mostrará el detalle para cada clase de este diagrama:

8.1.1 Clase Point



Un Point está representado por:

point: Utiliza la clase Coordinates del generador de mallas mixtas *met*. Coordinates es una clase que almacena las coordenadas de un punto en 3D (x,y,z).

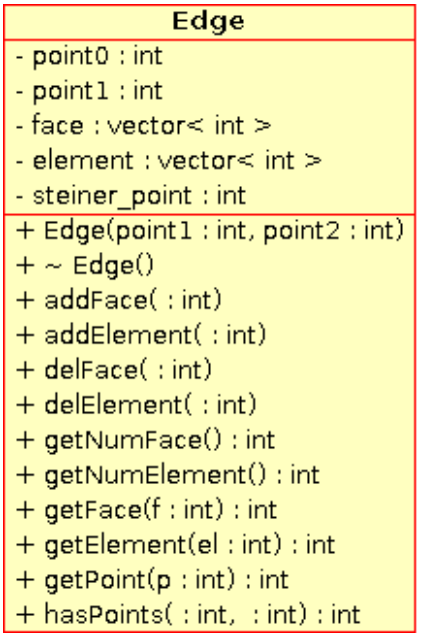
edge: Es un vector con las aristas que están compartiendo el punto. Guarda el identificador de la arista, el cual es un int y que está registrado en la clase Tessellation.

Los métodos importantes de Point son:

addEdge: Agrega una arista al vector de aristas. Esto sucede cuando hay cambios en la malla, ya sea en un proceso de particionamiento o cuando se refina.

delEdge: Borra una arista. Generalmente esto sucede al particionar o refinar la malla cuando desaparecen aristas.

8.1.2 Clase Edge



Una Edge está representada por:

edge: Es el constructor que recibe dos índices a puntos y con ellos crea la arista.

point0: Corresponde al primer punto con el cual se forma la arista. Almacena el índice del punto generado en la clase Tessellation.

point1: Corresponde al segundo punto con el que se puede formar una arista. Almacena el índice del punto generado en la clase Tessellation.

face: Es un vector que almacena la lista de caras que comparten la arista. Almacena los índices a las caras.

element: Es un vector que almacena todos los elementos que comparten la arista. Almacena los índices a los elementos.

Los métodos importantes de la clase Edge son:

addFace: Agrega una cara a la lista de caras.

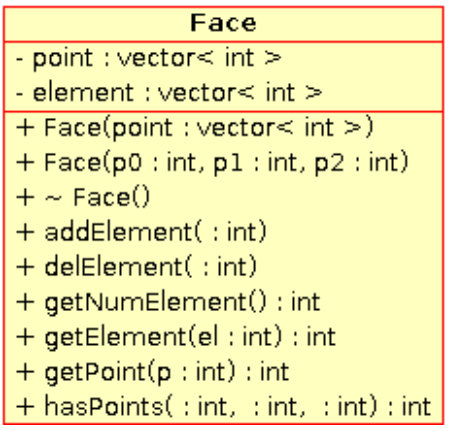
addElement: Agrega un elemento a la lista de elementos.

delFace: Borra una cara de la lista de caras.

delElement: Borra un elemento de la lista de elementos.

getNumFace:	Retorna el número de caras que comparten la arista.
getNumElement:	Retorna el número de elementos que compartes la arista.
getFace:	Retorna el índice de cierta cara.
getElement:	Retorna el índice a cierto elemento.
getPoint:	Devuelve el índice a un punto de la arista.
hasPoints:	Verifica si los dos puntos son iguales a los puntos de la arista.

8.1.3 Clase Face



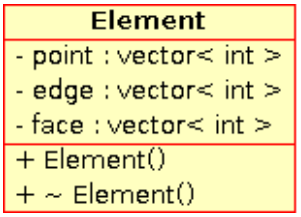
Una Face está representada por:

point: Vector de índices a puntos los cuales forman la cara.
element: Vector con la lista de índices a elementos que comparten esta cara.

Los métodos más importantes de la clase Face son:

face: Crea una cara a partir de un vector de índices a puntos.
addElement: Agrega un índice de elemento a la lista de elementos que comparten esta cara.
delElement: Borra un elemento según su índice.
getNumElement: Devuelve el número de elementos que comparten la cara.
getElement: Retorna el índice único y original generado por la teselación para cierto elemento guardado en la lista de elementos que comparten esta cara.
getPoint: Devuelve el índice a un punto de la cara.
hasPoints: Retorna si esta cara tiene los tres puntos indicados. Es para verificar en el proceso de particionamiento si es que esta cara ya existe o no.

8.1.4 Clase Element



Un Element está representado por:

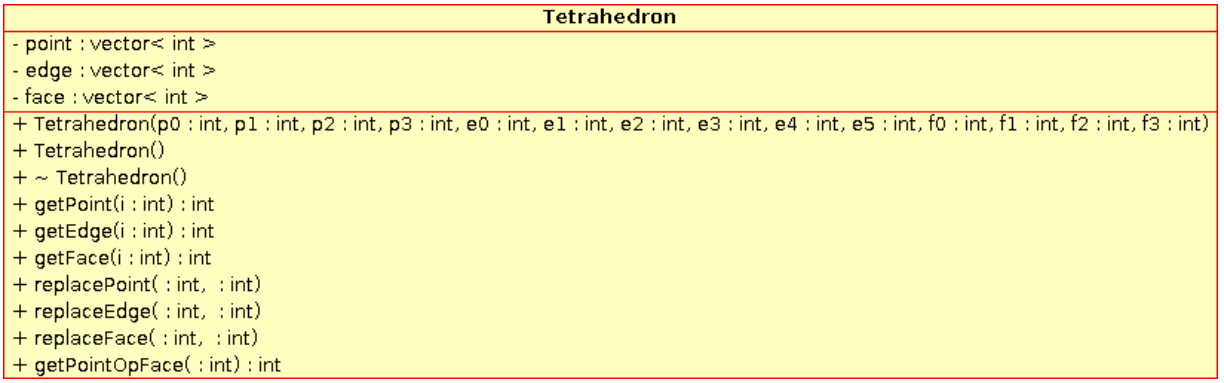
point: Es un vector con la lista de índices a los puntos que forman parte de este elemento.

edge: Es un vector con la lista de índices a las aristas que forman parte de este elemento.

face: Es un vector con la lista de índices a las caras que forman parte de este elemento.

Los métodos más importantes de la clase Element se heredan en Tetrahedron, y en la sección de esta clase serán explicados.

8.1.5 Clase Tetrahedron



Un Tetrahedron está representado por:

point: Es un vector con la lista de índices a los puntos que forman parte de este tetraedro.

edge: Es un vector con la lista de índices a las aristas que forman parte de este tetraedro.

face: Es un vector con la lista de índices a las caras que forman parte de este tetraedro.

Los métodos más importantes de la clase Tetrahedron son:

tetrahedron: Es el constructor, recibe como parámetros los índices a sus cuatro puntos, seis aristas y cuatro caras.

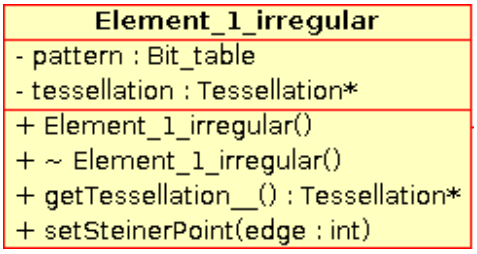
getPoint: Retorna el índice a un punto.

getEdge: Retorna el índice a una arista.

getElement: Retorna el índice a un elemento.

getPointOpFace: Devuelve el punto opuesto a una cara. Esto se usa en el test de la esfera.

8.1.6 Clase Element_1_irregular



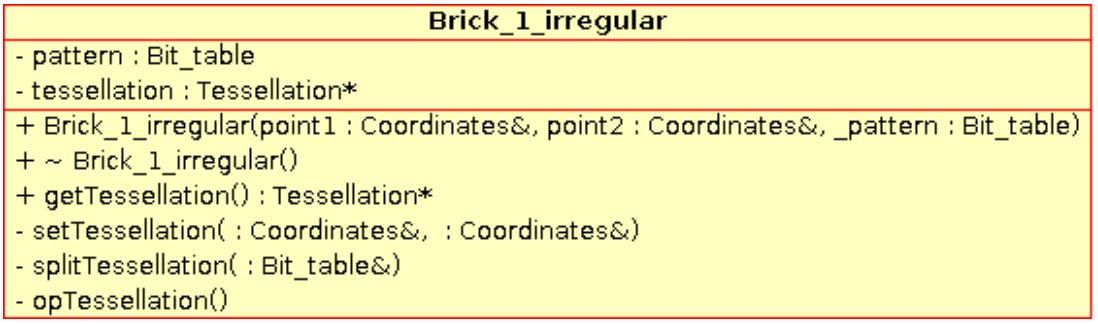
Un Element_1_irregular está representado por:

pattern: Corresponde al patrón de bit_table, el cual es la base para la generación de la teselación.

tessellation: Es la teselación generada con el pattern como dato de entrada.

En la descripción de la clase hija de Element_1_irregular, es decir, de Brick_1_irregular, se explicarán con detalle los métodos importantes.

8.1.7 Clase Brick_1_irregular



Un Brick_1_irregular está representado por:

pattern: Corresponde al patrón de bit_table, el cual es la base para la generación de la teselación.

tessellation: Es la teselación generada con el pattern como dato de entrada.

Los métodos más importantes de Brick_1_irregular son:

brick_1_irregular: Constructor del cuboide 1-irregular.

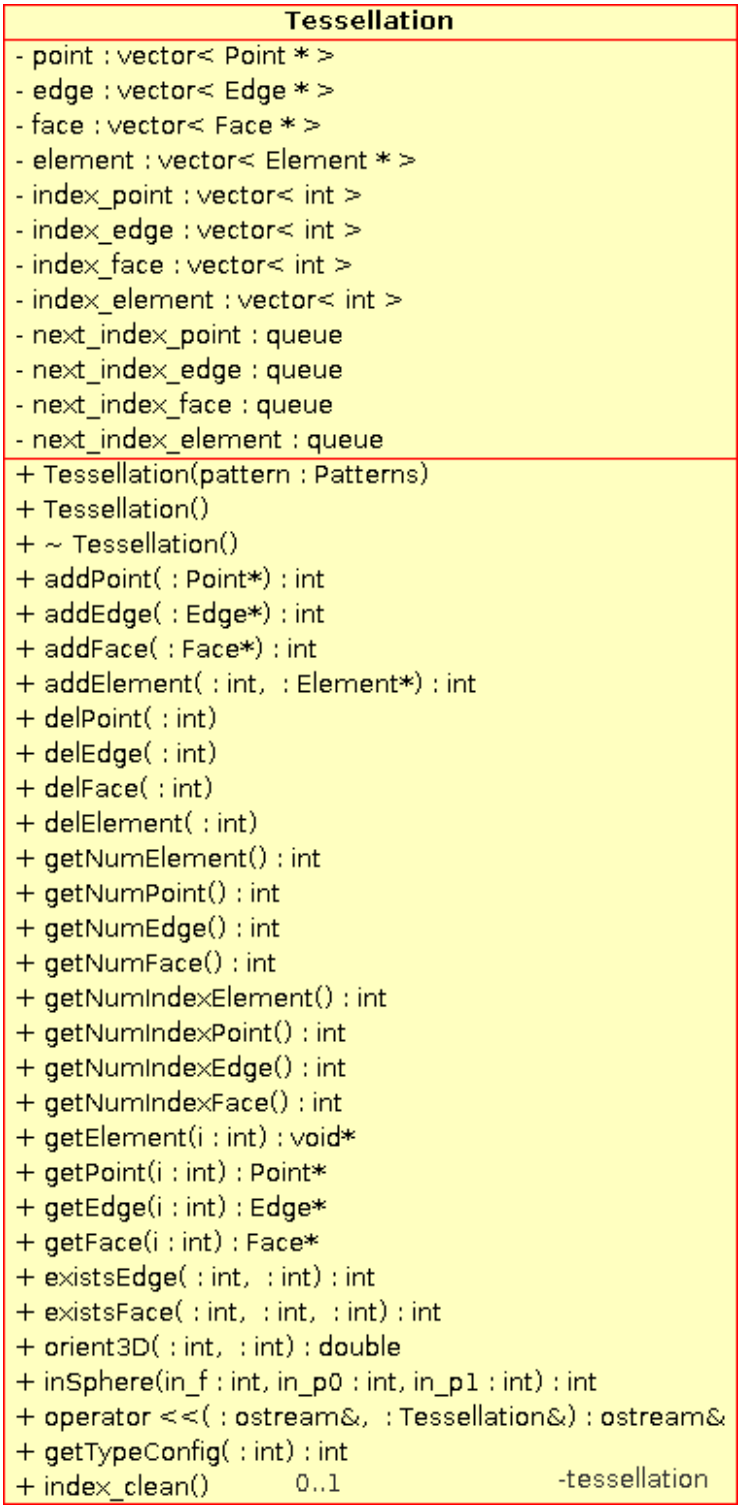
getTessellation: Devuelve la teselación asociada al cuboide 1-irregular.

setTessellation: Crea una teselación, dentro de este método se llama a splitTessellation y opTessellation.

spliTessellation: Particiona la teselación.

opTessellation: Refina la malla.

8.1.8 Clase Tessellation



La clase Tessellation está representada por:

point:	Vector de los puntos que forman parte de la teselación.
edge:	Vector con las aristas que forman parte de la teselación.
face:	Vector con las caras que forman parte de la teselación.
element:	Vector con los elementos que forman parte de la teselación.
index_point:	Vector con los índices de los puntos.
index_edge:	Vector con los índices de las aristas.
index_face:	Vector con los índices de las caras.
index_element:	Vector con los índices de los elementos.
next_index_point:	Cola para indicar el próximo índice a utilizar al guardar un punto.
next_index_edge:	Cola que indica el próximo índice a utilizar para guardar una arista.
next_index_face:	Cola que indica el próximo índice a utilizar para guardar una cara.
next_index_element:	Cola que indica el próximo índice a utilizar para guardar un elemento.

Los métodos más importantes de la clase Tessellation son:

addPoint:	Agrega un punto.
addEdge:	Agrega una arista, retorna la posición donde se guardó.
addFace:	Agrega una cara.
addElement:	Agrega un elemento.
delPoint:	Borra el punto en la posición indicada.
delEdge:	Borra la arista en la posición indicada.
delFace:	Borra la cara en la posición indicada.
delElement:	Borra el elemento en la posición indicada.
getNumElement:	Devuelve el número de elementos.
getNumPoint:	Devuelve el número de puntos.
getNumEdge:	Devuelve el número de aristas.
getNumFace:	Devuelve el número de caras.
getNumIndexElement:	Devuelve el número de índices de elementos.
getNumIndexPoint:	Devuelve el número de índices de puntos.
getNumIndexEdge:	Devuelve el número de índices de aristas.
getNumIndexFace:	Devuelve el número de índices de caras.

getElement:	Devuelve el elemento indicado.
getPoint:	Devuelve el punto indicado.
getEdge:	Devuelve la arista indicada.
getFace:	Devuelve la cara indicada.
existsEdge:	Devuelve el índice de una arista si es que existe.
existsFace:	Devuelve el índice de una cara si es que existe.
orient3D:	Indica la orientación de un tetraedro.
inSphere:	Indica si un punto está dentro de la esfera.
operator<<:	Sobrecarga <<. Esto se ocupa al imprimir una teselación
getTypeConfig:	Retorna el tipo de configuración según la cara.
index_clean:	Limpia los vectores index_.