UNIVERSIDAD TECNICA FEDERICO SANTA MARIA

Repositorio Digital USM **https://repositorio.usm.cl**

Tesis USM TESIS de Pregrado de acceso ABIERTO

2017

REPRESENTACION DE CARACTERÍSTICAS INTERNAS EN UNA MALLA DE VOLUMEN

FIGUEROA GARRIDO, JAVIER ALEJANDRO

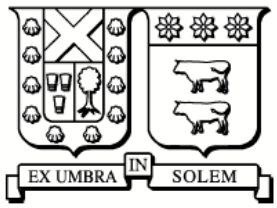
http://hdl.handle.net/11673/21411

Repositorio Digital USM, UNIVERSIDAD TECNICA FEDERICO SANTA MARIA

UNIVERSIDAD TÉCNICA FEDERICO SANTA MARÍA

Departamento de Informática

Santiago - Chile



Representación de características internas en una malla de volumen

Javier Figueroa

memoria de titulacion para optar al título de ingeniero civil informático

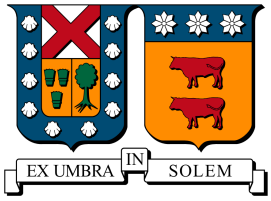
Profesor Guía: Claudio Lobos

AGOSTO 2017

UNIVERSIDAD TÉCNICA FEDERICO SANTA MARÍA

Departamento de Informática

Santiago - Chile



Representación de características internas en una malla de volumen

Javier Figueroa

memoria de titulacion para optar al título de ingeniero civil informático

Profesor Guía: Claudio Lobos

AGOSTO 2017

Agradecimientos

Quiero agradecer a mi familia por el apoyo incondicional que me ha dado en estos meses que he dedicado a este trabajo, y en general en las dificultades que he tenido que afrontar en mi vida. Muchas gracias mamá y papá, de todo corazón, ya que ustedes me enseñaron a ser responsable, honesto, y a no rendirme. También a mi hermana que se encuentra en una etapa similar a la mía, cercana a concluir sus estudios. Extiendo este saludo al resto de mi familia: mis abuelos, mis tíos y primos.

También agradecer a mis amigos, que me han dado momentos de distracción y relajo en algunos periodos de estrés. A todos quienes he conocido en la Universidad y me han acompañado y apoyado en este largo camino, no sólo compañeros, también incluyo a profesores y ayudantes que me han aportado no sólo conocimiento, sino también valores, a los guardias, las tías del aseo y otros funcionarios. Es imposible nombrarlos a todos, me siento realmente afortunado de la gente buena que he conocido en esta vida.

Finalmente, agradecer a mi profesor guía, Claudio Lobos, por darme este interesante tema y apoyarme siempre que lo necesité. Sus consejos y guía fueron claves para poder concluir este trabajo.

Resumen

Las mallas geométricas corresponden a representaciones de objetos compuestas de elementos que pueden ser tridimensionales o bidimensionales. Los métodos de simulación numérica permiten realizar simulaciones de sucesos reales y necesitan, en la mayoría de los casos, una malla geométrica. En la actualidad se cuenta con una técnica de mallado capaz de manejar simultáneamente varias superficies, sin embargo, no se han representado las características internas que se producen por la intersección entre ellas. En este trabajo se propone una solución para representar correctamente las características internas en un entorno tridimensional.

Abstract

Geometric meshes correspond to representations of objects composed of elements that can be three-dimensional or two-dimensional. Numerical simulation methods allow simulations of real events and, in most cases, require a geometric mesh. At present, there is a meshing technique capable of simultaneously handling several surfaces, however, the internal characteristics produced by the intersection between them have not been represented. In this paper we propose a solution to correctly represent the internal characteristics in a three-dimensional environment.

Glosario

Vértice: Lugar geométrico sobre el cual dos o más arcos coinciden.

Arco: Segmento que une dos vértices.

Cara: Superficie plana que encierra un cuerpo geométrico.

Cuerpo geométrico: Figura geométrica tridimensional que ocupa un lugar en el espacio.

Malla de superficie: Representación geométrica de la superficie de un objeto.

Malla de volumen: Representación geométrica de la superficie de un objeto y su interior.

Topología: Se refiere a propiedades y características que poseen los cuerpos geométricos que se mantienen sin alteraciones.

Índice general

Agradecimientos II

Resumen III

Abstract IV

Glosario V

Introducción VIII

1. Definición del Problema, objetivos y metodología 1

1.1. Objetivos 2

1.1.1. Objetivo Principal 2

1.1.2. Objetivos Específicos 2

1.2. Metodología 2

2. Estado del Arte 4

2.1. Patrones de Elementos Mixtos 4

2.2. Técnicas de generación de mallas de volumen 7

3. Propuesta de Solución 13

3.1. Descripción General de la Solución 13

3.2. Estudio de la herramienta *Mixed-elements Octree* 13

3.3. Patrones de Representación Interna 16

3.4. Problemas en la representación interna 18

3.4.1. Inconsistencia topológica entre patrones internos y de superficie. 19

3.4.2. Estado de borde de nodos y elementos desactualizado 31

3.4.3. Criterio incorrecto de selección de patrones internos 34

3.4.4. Errores de código 35

4. Resultados **37**

4.1. Estadísticas 41

4.1.1. Estadísticas de *Mixed-elements Octree* sin representación de características internas 41

4.1.2. Estadísticas de *Mixed-elements Octree* con representación de características internas 42

Conclusiones **45**

Trabajo Futuro 47

Bibliografía **51**

Introducción

Cuando se investiga una o varias hipótesis, se suelen realizar simulaciones que aproximen sucesos reales. Una forma de realizar esto es modelar elementos de la vida real en representaciones virtuales. Si se necesita modelar elementos complejos, como un esqueleto humano, generalmente es imposible describir dicho elemento con una función. En dicho caso, es necesario discretizar el espacio continuo. Dicha discretización se conoce como malla.

Se define una malla geométrica como una representación de objetos tridimensionales compuestas de elementos que pueden ser bidimensionales o tridimensionales. Cuando sólo se considera la representación externa del objeto, se conocen como mallas de superficie. Si además importa la representación interna, se conocen como mallas volumétricas.

Una de las aplicaciones más importante de una malla de volumen es la simulación de ecuaciones físicas para poder realizar distintos tipos de análisis (sobre estructura, deformaciones, fracturas, efecto del calor, entre otros). Estas simulaciones son descritas mediante ecuaciones diferenciales parciales y su solución es obtenida mediante un método numérico como Diferencias Finitas o Elementos Finitos.

Los algoritmos de generación de mallas son de alta complejidad y utilizan grandes cantidades de memoria, por lo que la implementación de los mismos se realiza de forma que ocupe la menor cantidad de memoria y tiempo de procesamiento posible. Una forma de enfrentar este problema es utilizar distintos niveles de refinamiento en zonas de la malla para una representación con mayor o menor detalle dependiendo de la importancia que se le quiera dar a cada zona.

Capítulo 1

Definición del Problema, objetivos y metodología

Las mallas hexaédricas son capaces de representar correctamente dominios geométricos sin esquinas, en los que no existen retricciones topológicas, y son comúnmente utilizadas en distintos problemas de simulaciones ya que la cantidad de elementos generados es menor comparado a las mallas tetraédricas. El problema es que estas mallas no realizan una correcta representación del dominio geométrico si este es muy complejo. Cuando se intersectan superficies, existen ciertas caracterísicas tales como bordes o esquinas que dificultan la representación de la malla resultante. Estas características se llaman *características finas*.

Una solución para mejorar dicha representación es no utilizar solamente hexaedros en la malla. En [1] se propone como alternativa utilizar adicionalmente elementos como prismas, pirámides y tetraedros. Esta técnica se encuentra implementada en el generador de mallas *Mixed-elements Octree* para la representación externa del dominio, es decir, para una malla de superficie. Sin embargo, si se necesita contar con una malla que represente un fémur y una prótesis incrustada en este, la malla geométrica debería tener internamente elementos que permitan diferenciar las zonas correspondientes al hueso y a la prótesis, es decir, su representación interna. Para estos casos, la implementación actual genera inconsistencias que producen caras inválidas en los bordes de las superficies.

Por lo tanto, el objetivo de esta memoria es desarrollar un conjunto de patrones que permita aproximar de forma correcta, el conjunto de características internas producidas por la intersec-

ción de una o más superficies sin generar caras inválidas. Estos patrones deben ser capaces de reconocer y representar automáticamente todos los casos en un entorno tridimensional.

1.1. Objetivos

1.1.1. Objetivo Principal

Representar las características internas de una malla volumétrica para la técnica de mallado *Mixed-elements Octree*.

1.1.2. Objetivos Específicos

- Identificar y solucionar los problemas asociados a las inconsistencias que producen caras inválidas en los bordes de las superficies.
- Comparar los resultados de la generación de mallas volumétricas sin la solución propuesta y con ella.

1.2. Metodología

Teniendo en cuenta los objetivos anteriormente listados, se realizó el siguiente plan de trabajo consistente en cinco partes:

1. **Estudio de la literatura sobre patrones de superficie y técnicas utilizadas en generación de mallas volumétricas (octree, elementos mixtos).**

   Se realizó un completo estudio, el cual es detallado en la siguiente sección. Se profundizó en la técnica Octree y la técnica complementaria usando elementos mixtos, pero además, se vieron otras técnicas de mallado utilizadas para representar elementos tridimensionales.

2. **Estudio de la herramienta de generación de mallas volumétricas del profesor Claudio Lobos.**

   Consistió en estudiar el flujo del código de la herramienta de generación de mallas volumétricas del profesor Claudio Lobos, en adelante denominada *Mixed-elements Octree*. El enfoque está en la implementación de los patrones de superficie externa, y la estructura de las clases, para comprender la arquitectura de la herramienta propuesta e implementada por [2].

3. **Identificación del problema**

   Se identificaron las causas que provocan las caras inválidas entre octantes vecinos. Esto se hizo por medio de *debugging* geométrico, separando los octantes que compartían la cara conflictiva, comprobando los patrones que fueron implementados para ambos y el estado de los nodos asociados. Si no se encontró ningún error en esta fase, se realizó *debugging* en otras zonas del código de *Mixed-elements Octree* hasta identificar la causa del problema.

4. **Implementación de la solución**

   Una vez identificadas las causas de cada problema, se implementó una solución para cada uno.

5. **Análisis de los resultados de la solución propuesta**

   Se realizaron comparaciones visuales y estadísticas entre la herramienta de generación de mallas volumétricas con y sin la solución implementada. Finalmente se concluye sobre estos resultados.

Capítulo 2

Estado del Arte

2.1. Patrones de Elementos Mixtos

Las mallas hexaédricas son comúnmente utilizadas por el método de Elementos Finitos (FEM) en distintos problemas de simulación. Sin embargo en [1] se presentan patrones que permiten reemplazar hexaedros por otros tipos de elementos, tales como prismas, pirámides y tetraedros, para así mejorar la representación del dominio mallado. Estos patrones son capaces de manejar todas las configuraciones posibles, además de que, excepto por un patrón, no incorporan nodos adicionales.

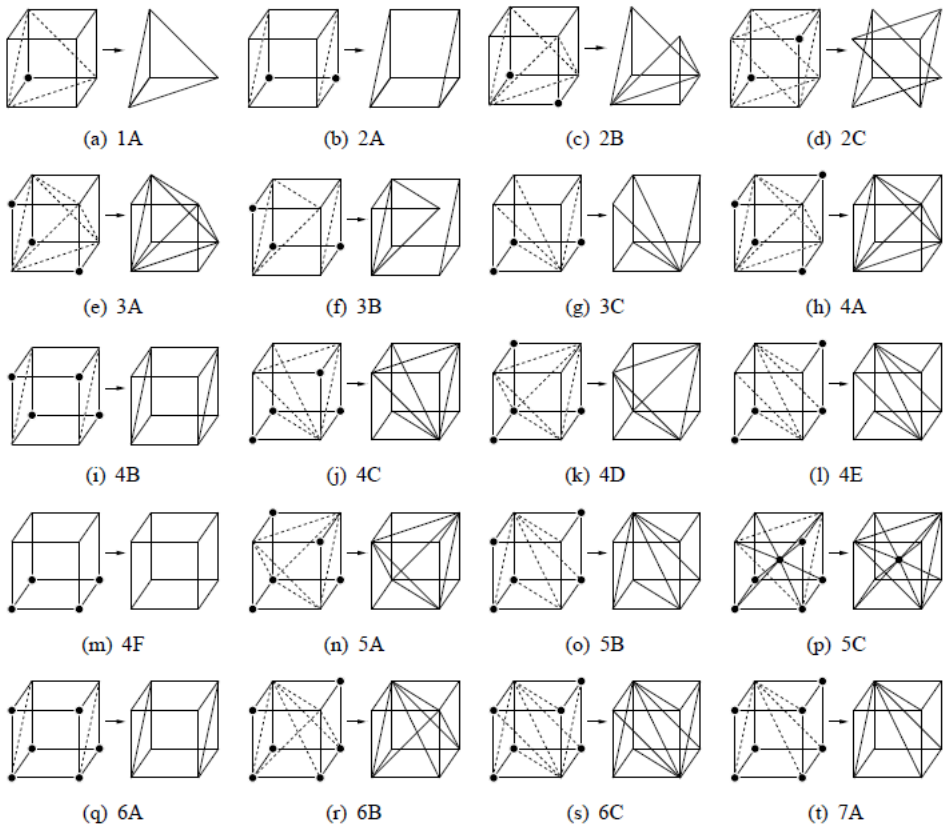


Figura 2.1: Patrones de elementos mixtos para aproximaciones de la frontera del dominio en mallas hexaédricas. Imagen extraída de [1]

Estos patrones por sí solo no pueden ser considerados una técnica de mallado, sino un complemento para técnicas de mallado hexaédricas. Este trabajo en particular se enfoca en los nodos externos, sin afectar los nodos internos de la malla.

Un año después, en [3], se presentan patrones de elementos mixtos (reemplazando hexaedros por los mismos elementos en [1]) para la técnica *octree*.

La técnica *octree* es una de las más utilizadas para concentrar nodos en una región del dominio de una malla. Consiste en dividir un octante en nuevos octantes de forma recursiva, de esta forma se genera una estructura de árbol en el cual cada nodo tiene exactamente ocho hijos. Un nodo del árbol es llamado octante. La idea es dividir cada octante que intersecta una porción de la entrada de los límites del dominio. Esta división se rige por algunas restricciones

geométricas, ya que una vez que el proceso de división o refinamiento es finalizado, quedarán algunas inconsistencias topológicas. Esto se debe a que el nivel de refinamiento, es decir, la cantidad de divisiones recursivas que se ha realizado a un octante, no es constante en la malla. Por lo tanto, es necesario realizar transiciones entre un nivel de refinamiento y otro dentro de una malla. Existen dos maneras de hacerlo, utilizando hexaedros o elementos mixtos. Cuando se usan hexaedros, un octante que es refinado se divide en 27 nuevos octantes, pero cuando se usan elementos mixtos, sólo 8 nuevos octantes son generados. Esto permite manejar de mejor manera la cantidad de nodos finales de la malla resultante.

En total, en esta investigación, se generan un total de 325 patrones, que reducen la cantidad final de nodos de la malla. Estos patrones en algunos casos no pueden discretizar el dominio con elementos básicos (Ver Figura 2.2) debido a las restricciones impuestas por los patrones sobre las caras. (Ver Figura 2.3)

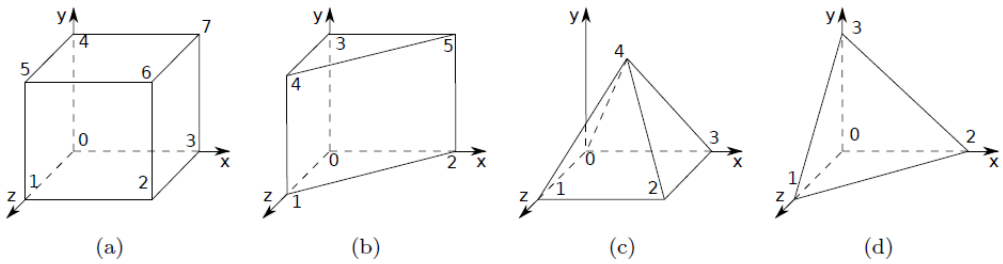


Figura 2.2: Los cuatro elementos básicos utilizados en la técnica de elementos mixtos. Imagen extraída de [3]

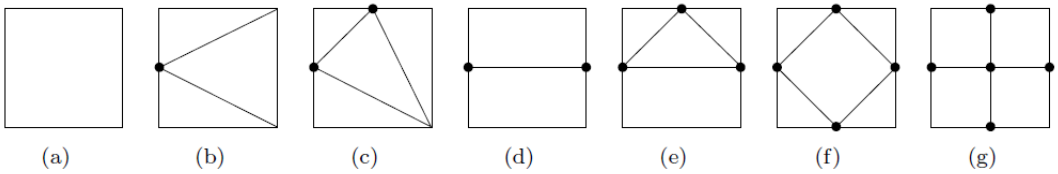


Figura 2.3: Se presentan los casos posibles para nodos en los bordes o en la cara central. Imagen extraída de [3]

Estos patrones presentados en [1] y [3] fueron implementados en la herramienta de genera-

ción de mallas geométricas usando elementos mixtos para representar las características externas en una malla de volumen y son implementados para representar las características internas del dominio en este trabajo.

2.2. Técnicas de generación de mallas de volumen

A continuación se presenta la investigación referente a otras técnicas de generación de mallas utilizando otras técnicas de mallado diferentes a octree, y otras restricciones orientadas a mejorar la calidad de la malla resultante o disminuir la cantidad de nodos manteniendo una buena representación del dominio.

En [4] se presenta un método para generar mallas geométricas utilizando tetraedros. En la herramienta TETGEN, se utilizan dos técnicas características:

- Triangulación de Delaunay: Una triangulación de Delaunay es una red de triángulos tal que la circunferencia circunscrita de cada triángulo de la red no debe contener ningún vértice de otro triángulo. Esta condición se conoce como condición de Delaunay.
- Diagrama de Voronoi: Es un método matemático fundamental dentro de la Geometría Computacional, almacena toda la información referente a la proximidad entre puntos.

Como el algoritmo esta basado en Delaunay, es capaz de preservar complejas topologías y geometrías. Los algoritmos para generar la red de triángulos que cumplan la condición de Delaunay fueron, el algoritmo de Bowyer-Watson [5] y el algoritmo $incrementalflip$ [4]. Ambos tienen un tiempo de complejidad de $O(n^2)$ en el peor de los casos. El tiempo esperado de complejidad es de $O(nlog(n))$ por lo que suelen ser bastante rápidos. Además se utilizaron técnicas avanzadas de geometría computacional para mejorar la robustez del método desarrollado.

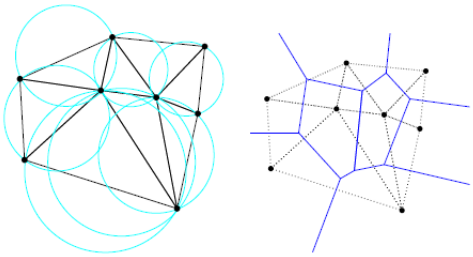


Figura 2.4: A la izquierda se muestra un ejemplo de triangulación de Delaunay y su respecto Diagrama de Voronoi se muestra a la derecha. Imagen extraída de [4]

En la herramienta NETGEN [6] se generan mallas tetraédricas de calidad utilizando una estrategia "de arriba a abajo" (top-down). Para generar mallas, primero se computan las esquinas, luego, los bordes son definidos y mallados en segmentos. A continuación, las caras son malladas y luego optimizadas. Los sub-dominios son llenados por tetraedros uno a uno, para esto se usa un algoritmo rápido de Delaunay, que falla en ciertos puntos del dominio, por lo que en esos casos se utiliza un algoritmo basado en reglas de backtracking. Finalmente el volumen es optimizado usando algoritmos de división, intercambio de elementos y movimiento de nodos.

En Gmsh [7] se enfocan en la velocidad y flexibilidad de la malla resultante. La herramienta presenta opciones para utilizar técnicas externas como Triangulación de Delaunay, algoritmo de Bowyer-Watson, así como usar un generador de mallas *third-party* como NETGEN y aprovechar las funciones de pre y post procesado. Lo que interesa estudiar son las rutinas propias de la herramienta para la generación de mallas. Ellos definen el tamaño de campo de la malla $\delta(x, y, z)$ como una función que define en cada punto del dominio, un tamaño objetivo para los elementos en ese punto. Las maneras de definir esta función son: tamaños de malla prescritos en los vértices del modelo e interpolados linealmente en los bordes del modelo, gradaciones de malla prescritas en los bordes del modelo, tamaños de malla definidos en otra malla del dominio y tamaños de malla que se adaptan a la curvatura principal de las entidades del modelo. Definiendo el largo del borde con respecto al tamaño del campo como:

$$l_e = \int_e \frac{dl}{\delta(x,y,z))}$$

El objetivo del proceso de generación de mallas consiste en dos pasos:

1. Generar una malla tal que cada borde de malla e sea de un tamaño cercano a $l_e = 1$
2. Generar una malla tal que cada elemento K esté $bien formado$

Esto se realiza para que la generación de la malla sea capaz de construir una malla de buena calidad cumpliendo con el tamaño de campo de la malla.

Además, Gmsh implementa su propio procedimiento de optimización de mallas que permite mejorar la calidad de las mallas tetraédricas por medio de intercambios de bordes y reubicaciones de vértices.

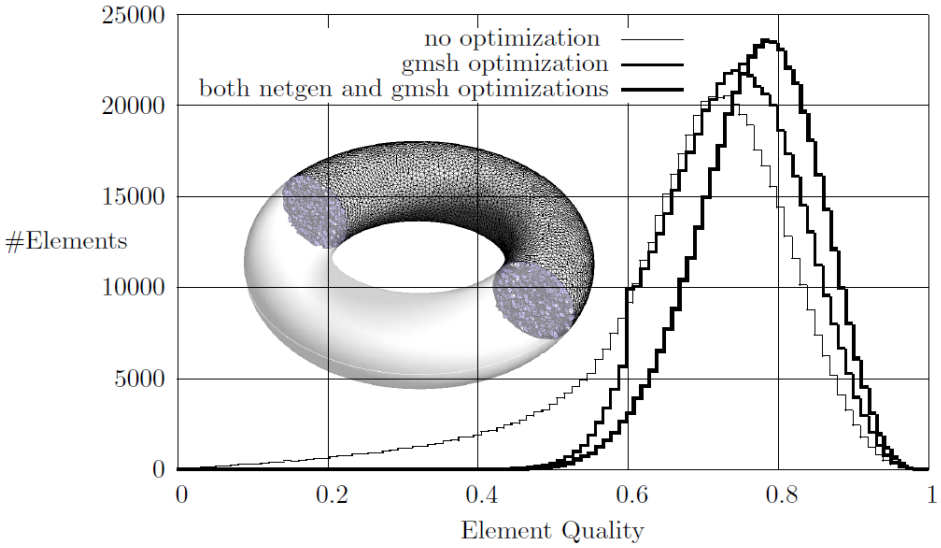


Figura 2.5: Distribución de la calidad de los elementos de una malla de un toroide que contiene aproximadamente 600.000 tetraedros. Se muestran los resultados de aplicar el algoritmo optimizador de Gmsh y el de NETGEN. Imagen extraída de [7]

Una herramienta de simulación con múltiples aplicaciones es ADINA [8]. Se mencionarán específicamente sus técnicas de mallado: Automatic Free-Form Meshing of All-Brick Elements y Automatic Hex Dominant Meshing.

- **Automatic Free-Form Meshing of All-Brick Elements:** Es posible crear una malla completamente hexaédrica cuando el modelo geométrico no tiene esquinas, esto puede suceder en muchas situaciones reales como Tomografías Computacionales (CT) o Imágenes de Resonancia Magnética (MRI). En estos casos, no hay restricciones topológicas para el mallado hexaédrico que impidan la generación de una malla que conste sólo de bloques. La Figura 2.6 presenta la generación de un torso humano utilizando hexaedros. Esta malla esta compuesta completamente de bloques lo que es de gran importancia ya que la cantidad

de elementos generados es menor al de mallas tetraédricas e incluso mallas compuestas de elementos mixtos.

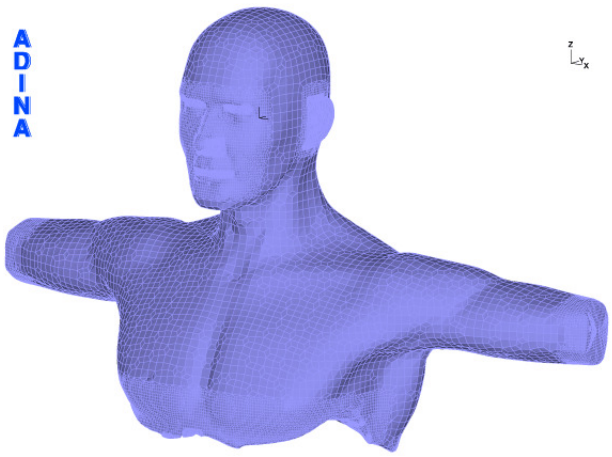


Figura 2.6: Malla de un torso humano generado por ADINA desde un archivo STL. Imagen extraída del sitio web del autor[2]

- **Automatic Hex Dominant Meshing:** Las situaciones en que el método anterior no puede ser utilizado, como subdivisiones no uniformes, pueden ser representadas por un mallado dominante hexaédrico. Idealmente se quiere generar un mallado compuesto sólo por elementos hexaédricos, sin embargo esto no es posible para cualquier geometría. Por esto, el mallado dominante hexaédrico se utiliza para generar una malla con una mezcla de hexaedros, pirámides y tetraedros con el énfasis de generar la mayor cantidad de hexaedros posible, lo que lo diferencia de la técnica de elementos mixtos. En la Figura 2.7 se presentan dos modelos complejos imposibles de representar utilizando sólo elementos hexaédricos.

[2] `http://www.adina.com/newsgH80.shtml`

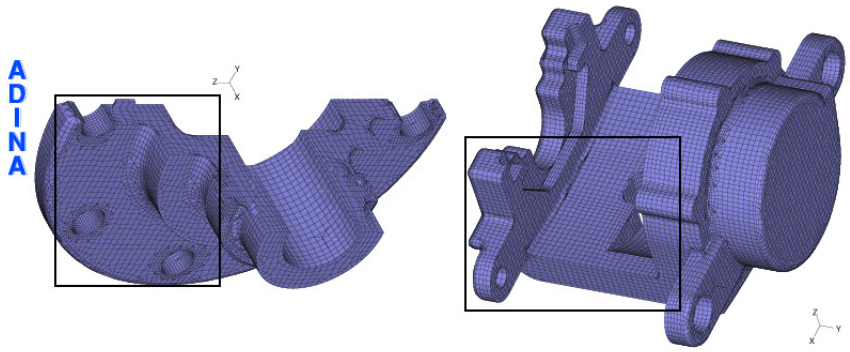


Figura 2.7: Dos mallas con partes complejas malladas correctamente utilizando la técnica del mallado dominante hexaédrico. Esto es posible a partir de la versión 8.7 de ADINA. Imagen extraída del sitio web del autor[4]

Finalmente se presenta a CUBIT[5] que es capaz de generar mallas 3D usando tetraedros y hexaedros utilizando el método de mallado llamado barrido (sweeping) realizando hex sweeping y multi-sweeping. Este método de mallado complementa el *free mesher*. Si la topología de un cuerpo es reconocida como barrible, el cuerpo puede ser mallado de manera muy eficiente con elementos hexaédricos y porciones (*wedge*) tetraédricas utilizando esta técnica. El número de nodos y elementos para cuerpos barridos es generalmente mucho más pequeño que los que fueron mallados con un *free mesher*. Además, el tiempo para crear estos elementos es mucho más pequeño[6]. En la Figura 3.1 se puede ver una malla generada utilizando barrido o sweeping.

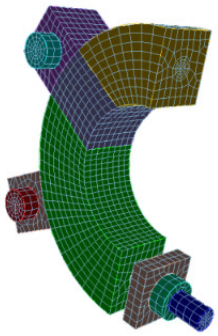


Figura 2.8: Ejemplo de sweeping. Imagen extraída del sitio web del autor[7]

[4] http://www.adina.com/newsgH80.shtml

[5] https://cubit.sandia.gov

[6] https://www.sharcnet.ca/Software/Ansys/16.2.3/en-us/help/wb_msh/ds_Mesh_Sweeping.html

CUBIT también contiene algoritmos para controlar y automatizar bastante el proceso de mallado, como selección de esquema de mallado automático (usando información topológica y geométrica para seleccionar la mejor herramienta), coincidencia de intervalo, agrupación de barridos y verificación de barridos.

[7] `https://cubit.sandia.gov`

Capítulo 3

Propuesta de Solución

3.1. Descripción General de la Solución

Utilizando la herramienta de generación de mallas volumétricas, se parte de una base de patrones de representación interna desarrollados previamente. La solución consiste en:

1. Detectar los problemas asociados a los patrones de representación interna que generan caras inválidas. Se implementa un algoritmo de debugging para identificar las causas.
2. Proponer e implementar una solución para cada problema identificado.

3.2. Estudio de la herramienta *Mixed-elements Octree*

El código fuente de *Mixed-elements Octree* fue realizado en C++. Su funcionamiento, a grandes rasgos, consiste en:

1. Recibir como entrada un conjunto de modelos de superficie y un nivel de refinamiento requerido para dicho conjunto. Es posible especificar un nivel de refinamiento mayor para una región en específico de la malla.[1]
2. Dividir los octantes según el refinamiento especificado y aplicar patrones de transición si existen distintos niveles de refinamiento según lo establecido en la técnica de elementos mixtos.

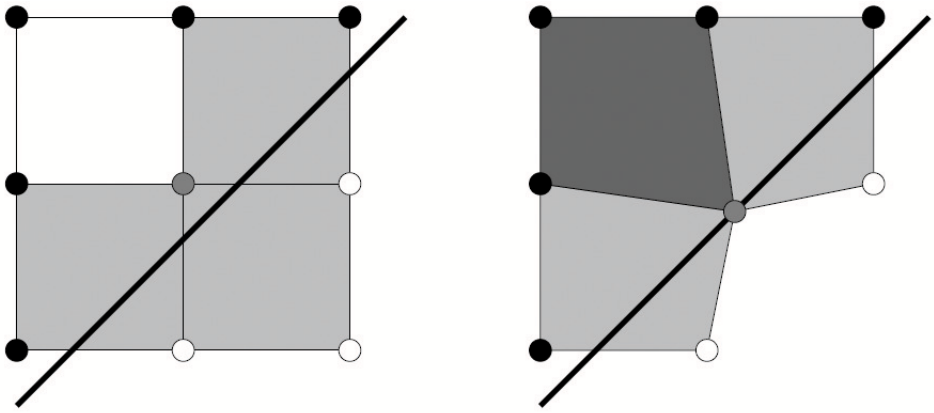
[1] Es posible definir el nivel de refinamiento de la malla resultante utilizando el comando -a si se pretende dividir los octantes de la malla de manera uniforme y -s si se quiere dividir exclusivamente los octantes cerca de la superficie de la malla, y aplicar patrones de transición para mantener la consistencia entre niveles.

3. Etiquetar nodos y elementos según su estado de borde. Eliminar elementos fuera de toda superficie.
4. Proyectar nodos cercanos a la superficie externa.
5. Proyectar nodos cercanos a la superficie interna.
6. Aplicar patrones de representación externa.
7. Aplicar patrones de representación interna.
8. Proyectar nodos de los elementos internos hacia la superficie
9. Retornar un archivo con la malla geométrica terminada.

El algoritmo genera una malla de rejilla y divide su arco más largo 2^{rl} veces, donde rl corresponde al nivel de refinamiento deseado. Tras la división, la malla tendrá elementos completamente fuera de la malla, completamente dentro y algunos con nodos dentro y fuera de la malla. Estos últimos se denominarán elementos de superficie. La *bounding box* se genera a partir de la malla triangular ingresada.

El proceso de etiquetado se realiza para cada nodo y elemento ingresado. Se etiqueta como externo o interno a cada uno sobre cada malla ingresada en función de si está dentro o fuera de ellas. Esto permite definir a cada elemento, en función de sus nodos, como externo, interno, o de borde. Los elementos externos son eliminados.

A continuación, se realiza un proceso de optimización de la malla, debido a que existirán elementos con baja intersección dentro de la malla, es decir, que la gran mayoría de su volumen se encontrará fuera de la superficie o dentro de ella. Esto perjudica a la representación, por lo que el algoritmo remueve completamente estos elementos, y en su lugar, proyecta nodos internos hacia la superficie. De esta forma se evita que al aplicar los patrones de representación interna se generen elementos demasiado pequeños lo que también ayuda a reducir la cantidad de elementos de la malla resultante.



(a) Situación con bajo nivel de intersección (b) Conservación de características

Figura 3.1: Ejemplo de proyección de nodos. El punto ha sido proyectado sobre la superficie, pasando a estar en el borde. Las consecuencias de esto forman parte de uno de los problemas a solucionar en este trabajo, que será mencionado más adelante. Imagen extraída de [9]

También sucede lo mismo con la representación interna, existiendo elementos que estén en su mayoría fuera de todas las superficies o en su mayoría intersectando cada una de las superficies ingresadas, por lo que se realiza el algoritmo de forma similar al caso externo.

Luego, se aplican los patrones de superficie que dividen los octantes en relación a los nodos internos y externos que posean. Se generan elementos mixtos. Finalmente hace una revisión del estado de borde de los nuevos nodos y elementos.

Se realiza el mismo procedimiento con los patrones internos, en este punto es donde se produce la inconsistencia que genera caras inválidas entre dos octantes.

A continuación, se vuelve a realizar un proceso de optimización, eliminando los elementos de la superficie que se encuentren muy fuera o muy dentro de ella, y proyectando los elementos internos hacia la superficie. Debido a esto, algunos elementos que eran considerados como internos pasan a ser externos en la malla resultante.

Finalmente, se guarda la información de la malla y se guardan los nodos externos en un archivo mdl.

3.3. Patrones de Representación Interna

Se define una malla compuesta por una superficie A y una superficie B. Para la representación interna de este ejemplo, existen 3 casos posibles para un hexaedro:

1. Un nodo pertenece a la superficie A.
2. Un nodo pertenece a la superficie B.
3. Un nodo pertenece a la superficie A y la superficie B.

El caso que define la representación interna de la malla es el caso 3. En este caso, del hexaedro se mantendrán los nodos pertenecientes a ambas superficies y eliminarán los nodos que pertenezcan sólo a una superficie. Sin embargo, usando sólo hexaedros esta representación presenta baja calidad. Se propone como solución dividir el hexaedro en elementos mixtos. Para dividir un hexaedro en elementos mixtos, sus caras pueden ser divididas en dos triángulos, dando lugar a un gran conjunto de configuraciones posibles para el hexaedro, dicho conjunto debe ser acotado para que exista consistencia topológica entre octantes vecinos.

En [1] se definen patrones de consistencia para las caras que se presentan en la Figura 3.8. Las configuraciones posibles debiesen ser iguales a las descritas en [1] para representar los elementos internos, con una diferencia conceptual: en lugar de ser considerados como nodos internos los nodos dentro de al menos una superficie y nodos externos los nodos fuera de todas las superficies, se consideran como nodos internos los que intersectan a todas las superficies y el resto de nodos se consideran como nodos externos. Esto resulta en que no existirán subelementos internos eliminados en la malla resultante, ya que no habrá elementos fuera del dominio, por lo tanto, la unión de todos los elementos mixtos colocados en un orden establecido debe conformar el hexaedro original.

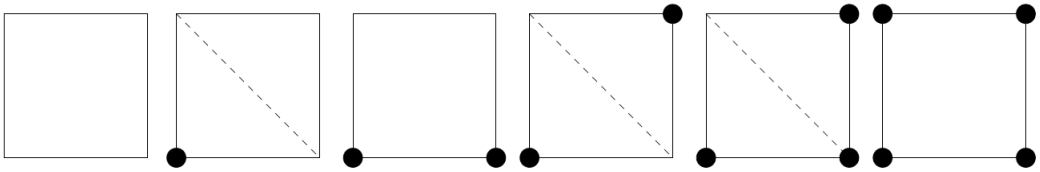


Figura 3.2: Patrones de consistencia de las caras del hexaedro. Las líneas punteadas corresponden a las diagonales que generan los dos triángulos y los círculos son los nodos correspondientes a una superficie. Imagen extraída de [1]

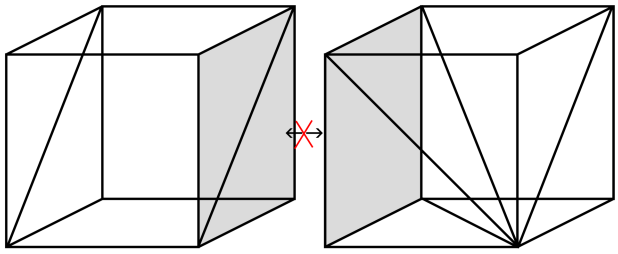


Figura 3.3: Ejemplo de inconsistencia topológica entre el patrón 7A y el patrón 3A. Si ambos octantes son vecinos tal como está presentado en la figura, la cara que comparten estará dividida en dos triángulos para el patrón 7A, mientras que para el patrón 3A será un cuadrado. Esta inconsistencia genera una cara flotante en la visualización de la malla.

Lo primero que se revisó fue la implementación de los patrones de representación interna actualmente implementados. Estos son iguales a los patrones de superficie descritos en [1], pero en lugar de elementos internos o externos, se definen elementos pertenecientes a una malla de superficie y otros pertenecientes a otra malla.

Se generó un modelo 3D de todos los patrones implementados, estos se muestran en la siguiente imagen:

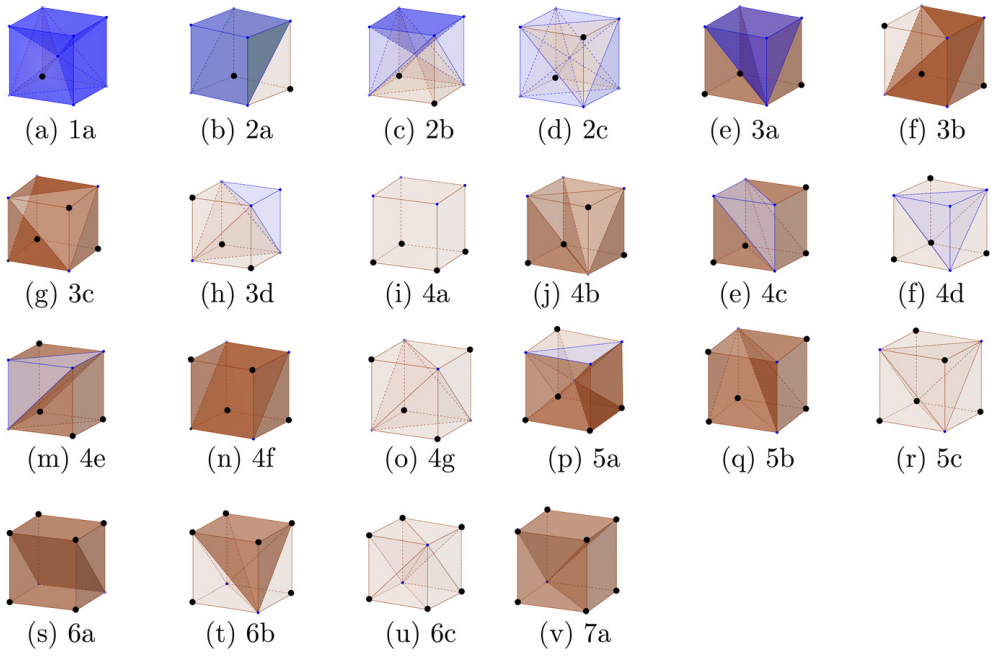


Figura 3.4: Patrones de representación interna. Los nodos destacados corresponden a nodos internos. Los subelementos azules corresponden a elementos mixtos que no poseen nodos internos. Los subelementos cafés poseen al menos un nodo interno.

Se pudo comprobar que los patrones están correctamente implementados y que no existen orificios al interior del cubo ni superposiciones de ningún elemento en ningún caso. También existe consistencia topológica entre ellos, es decir, estos patrones cumplen la regla sobre las caras establecida en [1].

Por lo tanto, el paso siguiente fue realizar un debugging de la herramienta *Mixed-elements Octree*. Utilizando la herramienta *Geomview* que permite identificar las caras inválidas como caras flotantes en la representación interna, se aislaron las caras inválidas encontradas tras aplicar la herramienta *Mixed-elements Octree* sobre un cilindro y una caja, y sobre un fémur y una prótesis.

3.4. Problemas en la representación interna

A continuación se detallarán los problemas identificados que generan las caras inválidas en la implementación de los patrones internos. Para cada problema se describe su solución implementada.

3.4.1. Inconsistencia topológica entre patrones internos y de superficie.

Se implementó una función de debugging en el código fuente que detecta e imprime los patrones de representación interna que fueron implementados en ambos octantes. Se aisló una de las caras inválidas identificadas y los octantes que la compartían, el resultado fue que sólo se imprimió un patrón para el Octante B (Patrón 2A). Esto significa que el Octante A no tiene un patrón de representación interna implementado.

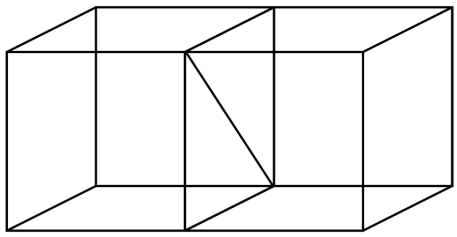


Figura 3.5: Los octantes involucrados en el error y la cara triangular que es visible en la malla. Se llamará Octante A al octante izquierdo y Octante B al octante derecho

Para determinar la razón de que no se aplicara un patrón interno en el Octante A, se creó un algoritmo de *debugging* que recorre todos los elementos justo antes de que se ejecuten los patrones de representación interna. El resultado mostró que el Octante A estaba compuesto de elementos mixtos, lo que significa que se aplicó un patrón de superficie a dicho octante, en lugar de un patrón interno. Para identificar que patrón se ejecutó, se aplicó un mismo algoritmo de detección de patrones de reprentación interna y de superficie. Se obtuvo que para el Octante A se ejecutó el patrón 7A de superficie y para el Octante B el patrón 2A interno.

El *debugging* permitió identificar la causa que genera las caras inválidas: no existe consistencia topológica entre patrones de representación interna y patrones de representación externa. Esto se debe a la anteriormente mencionada diferencia conceptual entre patrones externos e internos, ya que para los patrones de superficie los nodos que se eliminan de un octante son los que se encuentran fuera de toda superficie, mientras que para los patrones internos, los nodos que se eliminan de un octante son los que no intersectan a todas las mallas de superficie. En

el caso de la cara inválida que se está estudiando, para el Octante A los cuatro nodos pertenecientes a esa cara están dentro de la malla, por lo que deben permanecer. Mientras que para el Octante B, tres de estos cuatro nodos desaparecen, ya que sólo pertenecen a una de las mallas. Esto significa que el patrón 7A de superficie no agrega una diagonal a la cara que el Octante A comparte con el Octante B, mientras que el patrón 2A interno si la agrega. Esto produce una inconsistencia topológica entre octantes vecinos.

Este problema se produce cuando un patrón interno se aplica a un octante que tiene como vecino uno al que se le aplicó un patrón externo. El objetivo es entonces compatibilizar los patrones internos con los externos, en los casos en que esta posibilidad se produzca.

Solución

La solución pasa por, obviamente, identificar las caras inconsistentes y solventar dicha inconsistencia, osea, mantener la topología de la superficie cuando pasa al interior de la malla. Las caras inválidas se generan cuando un patrón de superficie posee una cara cuadrangular y un patrón interno posee una cara triangular. El caso inverso implicaría que para la cara del patrón de superficie exista al menos un nodo fuera toda superficies, lo que haría imposible que fuese vecino de un patrón interno en el que todos sus nodos están dentro de al menos una superficie, por lo que esta solución no debe abarcar dicho escenario, puesto que de ocurrir, sería producto de un error diferente al que se ha identificado en este caso.

La implementación de la solución se realiza en el método *applyBoundaryTemplates* de la clase *Mesher*, encargada de recorrer los elementos y definir si se les debe aplicar los patrones de representación interna, además, guarda los nuevos elementos generados por los patrones y los elementos a los que no se les aplicó un patrón en una nueva lista de elementos.

La clase *Mesher* llama a otro método *applyBoundaryTemplates* de la clase *EnhancedElement* que define qué clase de patrones corresponde aplicar al elemento dado, y llama a las clases que dividen los elementos. No se debe confundir dicho método con el método *applyBoundaryTemplates* de la clase *Mesher*.

Se presenta la estructura del método *applyBoundaryTemplates* de la clase *Mesher* en el Algoritmo 1:

```
para cada lista elements de elementos i hacer
    si EsElementoDeBorde() es Verdadero entonces
        // Si es de borde externo, no hay que aplicar patrones
        //se agrega el elemento i actual a la nueva lista de elementos
        elements[i] ← Empujar a Nueva Lista de Elementos;
        Continuar;
    fin
    // applyBoundaryTemplates define qué clase de patrones corresponde aplicar al
     //elemento i
    si elements[i].applyBoundaryTemplates() es Falso entonces
        // Si no corresponde aplicarlos, se agrega el elemento i actual a la nueva lista de
         //elementos
        elements[i] ← Empujar a Nueva Lista de Elementos;
        Continuar;
    fin
    Comprobar estado nodos;
    elements ← limpiar ;
    elements ← copiar Nueva Lista de Elementos;
    Chequear estado de intersección de nuevos elementos;
fin
```

Algoritmo 1: Algoritmo del método *applyBoundaryTemplates* en *Mixed-elements Octree* original.

La solución se aplica en el loop que recorre los elementos, antes de definir si se le aplica un patrón al elemento.

Primero, se identificaron los elementos mixtos generados por los patrones de superficie. Estos pueden ser hexaedros, pirámides, prismas y tetraedros. Como los tetraedros no generan caras cuadrangulares, no forman parte de la solución. Las pirámides y los prismas necesariamente son generadas por los patrones de superficie, puesto que en el instante en que se ejecuta la solución, los elementos no han sido divididos por los patrones internos. Se consideran sólo las pirámides y los prismas que poseen al menos un nodo dentro de toda superficie en su cara cuadrangular, ya que de no ser así, nunca serán vecinos de un patrón interno. Finalmente, sólo los hexaedros que poseen al menos un nodo fuera de toda superficie son considerados, puesto que sólo se pueden haber generado debido a un patrón de superficie, además se exige que posean al menos un nodo dentro de todas las superficies, ya que si no fuese así, nunca serán vecinos de un patrón

interno. Estas consideraciones filtran eficientemente sólo los subelementos externos que generan inconsistencia con los subelementos de los patrones internos.

Estos subelementos poseen al menos un nodo fuera de toda superficie y un nodo dentro de toda superficie, por lo que son subelementos *híbridos*, es decir, definen tanto la representación externa como interna. La consideración de que al menos un nodo pertenezca a toda superficie en la cara cuadrangular permite saber de antemano que dicha cara deberá ser dividida para mantener consistencia topológica con su elemento vecino. Esta consideración no se hace de forma explícita en el caso de los prismas y las pirámides, si no que se cubren inmediatamente los casos posibles de nodos en ambas superficies y nodos en una o ninguna superficie.

Como ha sido mencionado, la cara cuadrangular que genera conflicto es dividida en dos triángulos para ser consistente con la cara triangular del patrón interno, para ello se dividen los elementos mixtos de los patrones de superficie en conflicto, es decir, los elementos híbridos que presentan tanto características externas como internas, de manera que en la caras cuadrangulares que deban tener representación interna exista una división que cumpla con la definición conceptual de nodos internos y externos de los patrones internos. De esta forma, no es necesario saber qué patrón fue implementado en el octante vecino, puesto que sea cual sea será dividido de la misma forma en la cara cuadrangular conflictiva.

Conceptualmente, la solución consiste en representar las características internas de los elemento híbridos, los cuales hasta ahora habían sido tratados como elementos de superficie, lo que hará que sean consistentes con sus octantes vecinos.

Al igual que con los hexaedros, existen varias configuraciones posibles para la división de las pirámides y los prismas híbridos que cumplan con la regla en la cara cuadrangular y mantengan la consistencia topológica en las otras caras.
Para el caso de las pirámides híbridas, estas tienen sólo una cara cuadrangular, por lo que la diagonal sólo se incluye en dicha cara.

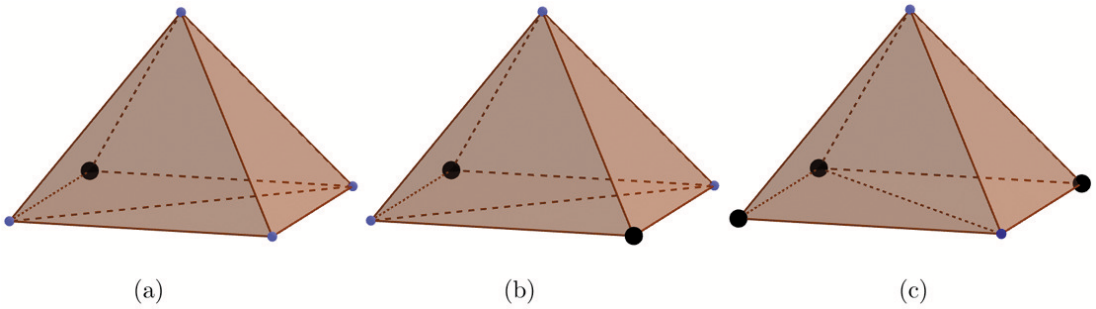


Figura 3.6: Las configuraciones posibles para la división de las pirámides. Los nodos destacados corresponden a nodos internos.

Para el caso de los prismas híbridos existen 3 caras cuadrangulares, 2 de ellas se comparten con octantes vecinos, y la cara cuadrangular diagonal puede ser compartida por hexaedros, pirámides o tetraedros. Se debió asegurar que las únicas caras que tuviesen cambios fuesen las caras cuadrangulares, por ello fue necesario agregar un nodo adicional para mantener la consistencia en la cara cuadrangular diagonal.

Como existen 2 caras cuadrangulares candidatas a ser vecinas de un octante interno, la cantidad de configuraciones posibles es mayor al de la división del prisma híbrido.

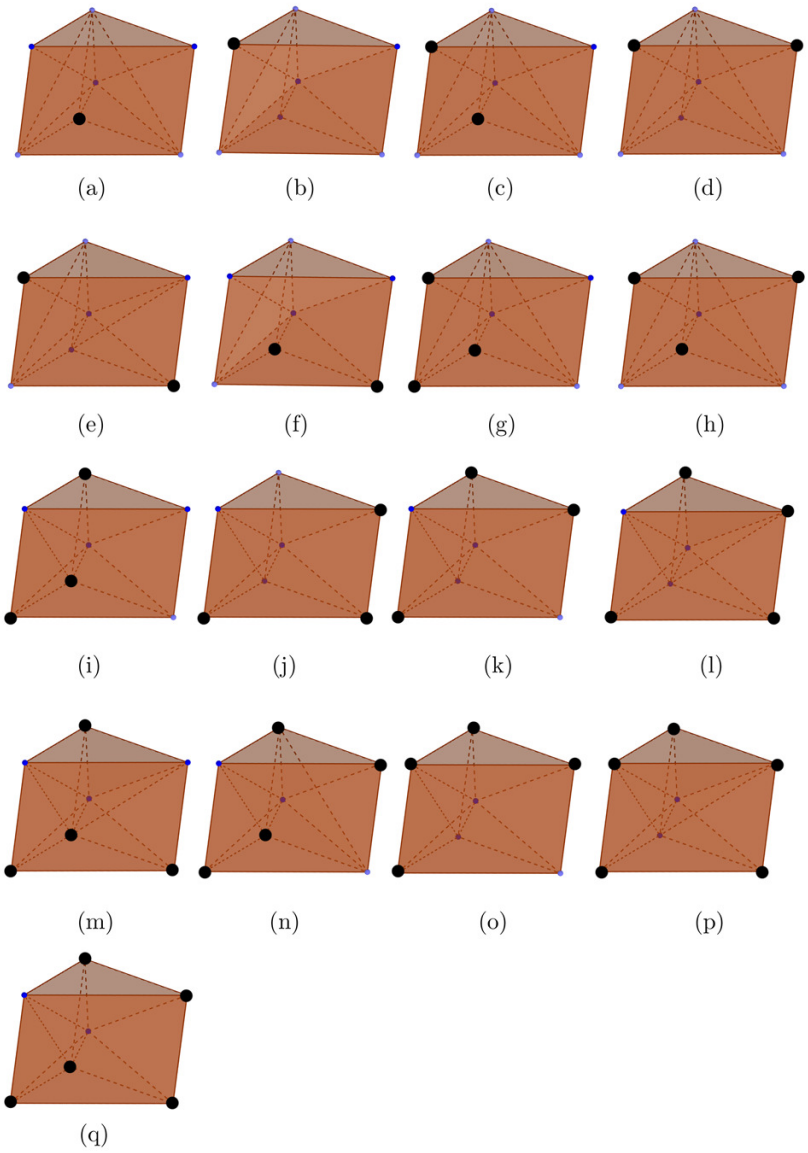


Figura 3.7: Las configuraciones posibles para la división de los prismas. Los nodos destacados corresponden a nodos internos.

Para la selección de los patrones a aplicar a las pirámides y los prismas, se determinó cuantos nodos internos según la representación interna, es decir, en todas las superficies, habían en la o las caras cuadrangulares. En función de cuantos nodos internos había, se consideraron los patrones de división respectivos que se debían aplicar. Se creó un método para cada configuración de división o patrón para las pirámides y los prismas, de forma de hacer el código más sencillo de entender. Los subelementos obligatorios en todas las divisiones se generaron en un método

adicional, para optimizar código.

A continuación se presenta el Algoritmo 2 que resume la selección de los patrones para las pirámides:

```
si PuntosElemento.Tamaño() es 5 entonces
    TotalNodosIn ← 0 ; Nodo[ ] ← 0;
    para cada nodo j de la lista PuntosElemento excepto el último hacer
        j ← Evaluar intersección mallas ;
        si j intersecta todas las mallas entonces
            Nodo[j] ← Verdadero ;
            TotalNodosIn ← TotalNodosIn + 1
        en otro caso
            Nodo[j]← Falso
        fin
    fin
    si TotalNodosIn es 1 entonces
        si Nodo[0] es Verdadero o Nodo[2] es Verdadero entonces
            PatronAPirámide()
        en otro caso
            PatronBPirámide()
        fin
    fin
    si TotalNodosIn es 2 entonces
        si Nodo[0] es Verdadero o Nodo[2] es Verdadero entonces
            PatronAPirámide()
        fin
        si Nodo[1] es Verdadero o Nodo[3] es Verdadero entonces
            PatronBPirámide()
        fin
    fin
    si TotalNodosIn es 3 entonces
        (...)
    fin
fin
```

Algoritmo 2: Algoritmo que selecciona el patrón a aplicar a una pirámide

La selección de patrones para los prismas sigue la misma lógica que la de las pirámides, pero

con 8 patrones posibles más el patrón adicional que contiene los subelementos obligados. Al ser mucho más complejo, el Algoritmo 3 lo resume mucho más, utilizando una función *SeleccionPatron* que realiza el mismo proceso de evaluación de nodos y casos posibles que el algoritmo de las pirámides, y retorna qué patrón debe ser aplicado.

si *PuntosElemento.Tamaño() es 6* **entonces**
  si *SeleccionPatron() es 'A'* **entonces**
    PatronAPrisma()
  fin
  si *SeleccionPatron() es 'B'* **entonces**
    PatronBPrisma()
  fin
  si *SeleccionPatron() es 'C'* **entonces**
    PatronCPrisma()
  fin
  si *SeleccionPatron() es 'D'* **entonces**
    PatronDPrisma()
  fin
  si *SeleccionPatron() es 'E'* **entonces**
    PatronEPrisma()
  fin
  si *SeleccionPatron() es 'F'* **entonces**
    PatronFPrisma()
  fin
  si *SeleccionPatron() es 'G'* **entonces**
    PatronGPrisma()
  fin
  si *SeleccionPatron() es 'H'* **entonces**
    PatronHPrisma()
  fin
  si *Un patrón fue seleccionado* **entonces**
    ElementosObligatorios()
  fin
fin

Algoritmo 3: Algoritmo que selecciona el patrón a aplicar a un prisma

Para crear los nuevos elementos en cada patrón, se crea un vector que posea la posición de cada nodo correspondiente al nuevo elemento. El orden de los nodos no es al azar, ya que debe ser consistente con el orden relativo o índice de los nodos para cada elemento, el cual se presenta

en la siguiente figura:

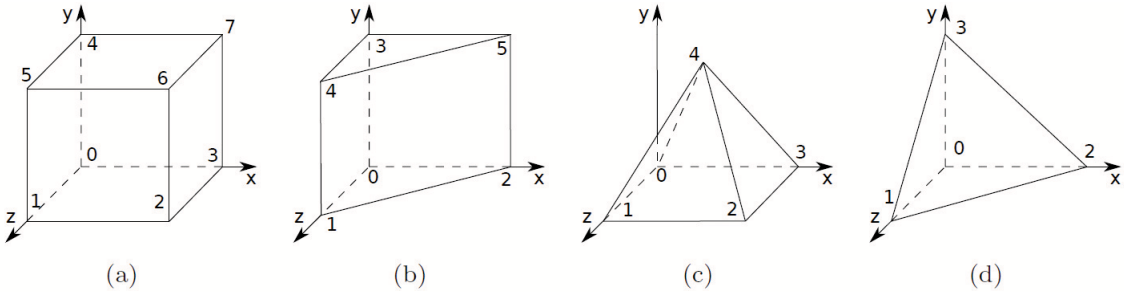


Figura 3.8: Índices de los cuatro elementos básicos considerados en este trabajo: (a) hexaedro, (b) prisma, (c) pirámide y (d) tetraedro. Imagen extraída de [3]

Se conoce la posición relativa de él o los nodos internos de la cara cuadrangular, y en base a esa posición se realizan las divisiones, ya que determinan qué nodos unirá la nueva diagonal que se incorpora a la cara cuadrangular. Se tiene una función llamada *getPoints()* que entrega un vector de posiciones absolutas de los nodos de un elemento ordenados en su orden relativo, lo que permite saber exactamente donde posicionar los nodos de los nuevos elementos generados, en base a la posición relativa de cada nodo de la cara cuadrangular y la posición absoluta de los mismos. Finalmente, se crea un objeto *EnhancedElement* con el vector de nodos del nuevo elemento y se agrega al vector de nuevos elementos. En el caso de los prismas, es necesario agregar un nodo adicional para mantener la consistencia topológica con la cara cuadrangular diagonal, que no es vecina de ningún hexaedro.

A continuación se presenta el Algoritmo 4 que resume la estructura de un patrón de división de los prismas:

```
vector tetraedro1(4),tetraedro2(4),tetraedro(3),tetraedro(4);
vector PuntosElemento[] ;
mid ← Agregar nodo intermedio ;
tetraedro1[0] ← PuntosElemento[0];
tetraedro1[1] ← PuntosElemento[3];
tetraedro1[2] ← PuntosElemento[1];
tetraedro1[3] ← mid;
tetraedro2[0] ← PuntosElemento[3];
tetraedro2[1] ← PuntosElemento[4];
tetraedro2[2] ← PuntosElemento[1];
tetraedro2[3] ← mid;
tetraedro3[0] ← PuntosElemento[0];
tetraedro3[1] ← PuntosElemento[2];
tetraedro3[2] ← PuntosElemento[3];
tetraedro3[3] ← mid;
tetraedro4[0] ← PuntosElemento[2];
tetraedro4[1] ← PuntosElemento[3];
tetraedro4[2] ← PuntosElemento[5];
tetraedro4[3] ← mid;
// Se crea un objeto EnhancedElement para cada subelemento
AgregarElemento(tetra1);
AgregarElemento(tetra2);
AgregarElemento(tetra3);
AgregarElemento(tetra4);
```

Algoritmo 4: Algoritmo que aplica un patrón de división del prisma. El vector *PuntosElemento* corresponde al vector de posiciones absolutas del prisma en su orden relativo. Para el primer tetraedro, su primer nodo relativo se posiciona en la posición absoluta del primer nodo relativo del prisma, su segundo nodo relativo en la posición absoluta del cuarto nodo relativo del prisma, y así sucesivamente. Este patrón corresponde al caso (c) de la Figura 3.7

Se creó un método llamado *fixConflictingElements* en la clase *EnhancedElement*. Este método es llamado con el objeto *elements* en la clase *Mesher*.

En *fixConflictingElements* se obtiene el tamaño del elemento en cuestión, la cantidad de nodos internos y el estado específico de cada nodo. En base a estas variables se selecciona la configuración de división del subelemento, si es que corresponde que sea dividido.

La función *fixConflictingElements* retorna *verdadero* si realizó la división del elemento y *falso* de no haberla hecho. Esta corresponde al algoritmo de selección de patrones de las pirámides y el algoritmo selección de patrones de los prismas.

Para el caso de los hexaedros híbridos, se optó por aplicar los patrones de representación interna en ellos. Esto se debe a que en el estado actual de la herramienta *Mixed-elements Octree*, para aplicar los patrones de representación interna la herramienta omite inmediatamente los hexaedros que tengan un nodo fuera de toda superficie, pero no considera que puedan tener algún nodo dentro de toda superficie y por lo tanto, existan características internas que deban ser representadas. Al ser un elemento de borde, significa que se le aplicó el patrón de superficie 4F, lo que significa que cuatro nodos pertenecientes a una de sus caras no pertenecen a ninguna superficie, esto permite que se les pueda aplicar un patrón interno sin temor a que se genere una inconsistencia inesperada, dado que sólo los otros cuatro nodos que son considerados internos por el patrón de superficie son candidatos a ser considerados internos por el patrón interno. Esto hace que, si o si, todos los nodos que están en toda superficie y por tanto son considerados internos por el patrón interno, tengan como vecinos elementos internos o elementos híbridos, y al ser aplicado el patrón interno al hexaedro híbrido, este siempre mantendrá la consistencia topológica con ellos.

Debido a esto, la solución eficiente fue modificar la función *applyBoundaryTemplates* para que cuando se diera dicha situación, permitiese que el hexaedro fuese dividido por un patrón interno. Esto obligó a realizar una modificación a la clase *EnhancedElement*, que también omitía inmediatamente los hexaedros que tuvieran un nodo fuera de toda superficie.

Tras esto, se detiene el *loop* actual del recorrido de elementos y se salta al siguiente elemento, para evitar que el elemento actual se agregue al vector de nuevos elementos, lo que sucedería si se continuase el *loop* actual. Es una manera eficiente de borrar el elemento que es reemplazado, sin tener que eliminarlo de la lista utilizando iteradores.

Finalmente, se presenta el Algoritmo 5 correspondiente al método *applyBoundaryTemplates* de la clase Mesher con todos los cambios introducidos para corregir la inconsistencia topológica entre patrones internos y de superficie. Notar las diferencias entre este Algoritmo 5 y el Algoritmo 1.

```
para cada lista elements de elementos i hacer
    si elements[i].fixConflictingElements() es Verdadero entonces
        Continuar;
    fin
    bool PermitirDivisión ← Falso ;
    si elements[i] es Hexaedro entonces
        si EsHibrido(elements[i]) es Verdadero entonces
            PermitirDivisión ← Verdadero ;
        fin
    fin
    si PermitirDivisión es Falso y EsElementoDeBorde() es Falso entonces
        // Si no es de borde (interno), no hay que aplicar patrones
        //se agrega el elemento i actual a la nueva lista de elementos
        elements[i] ← Empujar a Nueva Lista de Elementos;
        Continuar;
    fin
    // applyBoundaryTemplates define qué clase de patrones corresponde aplicar al
     //elemento i
    si elements[i].applyBoundaryTemplates() es Falso entonces
        // Si no corresponde aplicarlos, se agrega el elemento i actual a la nueva lista de
         //elementos
        elements[i] ← Empujar a Nueva Lista de Elementos;
        Continuar;
    fin
    Comprobar estado nodos;
    elements ← limpiar ;
    elements ← copiar Nueva Lista de Elementos;
    Chequear estado de intersección de nuevos elementos;
fin
```

Algoritmo 5: Nuevo Algoritmo del método *applyBoundaryTemplates* que recorre los elementos y define si se les debe aplicar un patrón de representación interna, que corrige la inconsistencia topológica entre patrones de superficie y patrones internos. También guarda los nuevos elementos generados por los patrones y los elementos a los que no se les aplicó un patrón en una nueva lista de elementos.

3.4.2. Estado de borde de nodos y elementos desactualizado

Dentro de la clase *EnhancedElement* se encuentra definido un vector llamado *border*. Este vector de tamaño *n_meshes*, siendo *n_meshes* el número de mallas ingresadas, representa si el elemento en cuestión se encuentra dentro o fuera de cada malla. En el método *labelNodesAndElements* de la clase *Mesher*, se definen los valores de dicho vector para cada elemento, en función de sus nodos. Dicha asignación se hace por medio del método *setBorderState* de la clase *EnhancedElement*.

Como se puede ver en el flujo del funcionamiento de *Mixed-elements Octree* anteriormente definido, el método *labelNodesAndElements*, que corresponde al Paso 3, es llamado antes de la proyección de nodos mayormente alejados del borde, esto hace que el estado de algunos nodos cambie, ya sea saliendo de algunas mallas, o entrando en otras. Sin embargo en la versión actual de la herramienta, la actualización de dicho cambio de estado no se realiza. Esto genera que se apliquen patrones de superficie incorrectos a su estado actual. Sólo los nuevos nodos y elementos mixtos agregados por los patrones son etiquetados tras aplicar los patrones de superficie, resultando que algunos elementos que no fueron divididos, los hexaedros, se encuentren con un vector *border* desactualizado, lo que hace que se le aplique un patrón interno incorrecto a su estado actual, generando caras inválidas.

En resumen, esta desactualización genera tanto patrones internos como patrones externos erróneos. Esto sólo puede suceder en las zonas de la malla donde se realizan proyecciones.

Este problema se pudo comprobar realizando debugging a la herramienta, y comprobando que cierto elemento no contenía la cantidad de nodos internos correspondiente al patrón aplicado. Se compararon los valores del vector *border* de dicho elemento con la función *getIOState()* que determina si un punto se encuentra o no dentro de determinada superficie, y se encontró una inconsistencia entre el estado del vector para una de las mallas y el valor que *getIOState()* retornaba sobre la misma malla en un punto.

Solución

La solución consistió en actualizar el vector *border* de los nodos y elementos, para ello se creó un método en la clase *Mesher* que realiza dicha operación, siendo este una variante del método *labelNodesAndElements* que corresponde al Paso 3 del funcionamiento de *Mixed-elements Octree* detallado anteriormente. Esta variante no elimina elementos fuera de toda superficie ya que esta

tarea fue realizada previamente, y se llama antes de aplicar los patrones, de forma que estos se apliquen correctamente a los elementos. El método se llama tras la proyección de nodos.

A continuación se presenta el Algoritmo 6 correspondiente al método en cuestión:

```
para cada lista points de puntos i hacer
    para cada lista n_meshes de mallas j hacer
        si IntersectaMalla(i,j) es Verdadero entonces
            points[i] ← Asignar como interno de malla j;
        fin
    fin
    points[i] ← Marcar como chequeado;
fin
para cada lista elements de puntos i hacer
    para cada lista n_meshes de mallas j hacer
        TodosAdentro ← Verdadero ;
        TodosAfuera ← Verdadero ;
        epts ← Puntos del elemento i ;
        para cada lista epts de puntos k hacer
            si points[k].DentroDeMalla(j) es Verdadero entonces
                TodosAfuera ← Falso ;
            en otro caso
                TodosAdentro ← Falso ;
            fin
        fin
        si TodosAfuera es Falso entonces
            si TodosAdentro es Falso entonces
                elements[i] ← Asignar estado de borde Verdadero para malla j
            en otro caso
                elements[i] ← Asignar estado de borde Falso para malla j
            fin
        fin
    fin
fin
```

Algoritmo 6: Algoritmo que actualiza el estado de borde de los elementos, almacenado en el vector *border*. *DentroDeMalla* corresponde al método *getIOState* y Asignar estado de borde corresponde al método *setBorderState*.

3.4.3. Criterio incorrecto de selección de patrones internos

Este problema se debe a un error en una variable al momento de definir qué patrón se debe aplicar a un elemento interno. Para poder explicar este error, primero debe definirse la variable *intersected_surf* que se declara en el método *applyBoundaryTemplate*s de la clase *Mesher*. Esta variable se usa en el loop que itera sobre los elementos dentro del cual se aplican los patrones internos. El Algoritmo 7 es el utilizado, el cual está un poco antes de definir si se le aplica o no un patrón al elemento actual:

```
bool found = false;
unsigned int intersected_surf;
para cada unsigned int j=0; j¡n_meshes; j++ hacer
    si elements[i].getBorderState(j) entonces
        intersected_surf = j;
        found = true;
        break;
    fin
fin
```

Algoritmo 7: Algoritmo que determina si un elemento intersecta al menos una superficie

Este algoritmo recorre todas las mallas y por medio de *getBorderState* evalúa si el elemento intersecta dicha malla, si es así, se asigna *true* a la variable *found* y termina el loop. Se guarda en *intersected_surf* el entero correspondiente a la malla que intersecta, la cual no necesariamente es la única malla que intersecta, sino la primera que encuentra. La variable *found* es importante, ya que si la malla no intersecta ninguna superficie, no corresponde que al elemento se le aplique un patrón.

La variable *intersected_surf* es parámetro del método *applyBoundaryTemplates* de la clase *EnhancedElement*. Es utilizada para definir cuales nodos del elemento ingresado son etiquetados como internos y cuales como externos: en base al valor de *intersected_surf*, que es el valor de una de las mallas, se evalúa si los nodos están dentro de esa malla o no, en caso de estarlo se clasifican como internos y de no estarlos se clasifican como externos. Esto es un error, para que un nodo sea clasificado como interno debe estar dentro de todas las mallas ingresadas en *Mixed-elements Octree*. Como el algoritmo considera sólo una malla, se aplican patrones incorrectos en los elementos, no pudiéndose garantizar consistencia topológica entre octantes vecinos. Para el caso encontrado en el debugging, se aplicó a un hexaedro el patrón 6a, siendo 3a el patrón que le correspondía.

Solución

Evidentemente la solución consistió en reemplazar el algoritmo por otro que considera todas las mallas ingresadas en *Mixed-elements Octree*. Este es presentado en el Algoritmo 8.

```
para cada i=0 hasta elemento.tamaño hacer
    bool dentro_todos = true;
    para cada j=0 hasta border.tamaño hacer
        si !border[j] entonces
            continue;
        fin
        si i no está dentro de la malla j entonces
            dentro_todos = false;
        fin
    fin
    si dentro_todos entonces
        NodoInterno.empujar(i);
    en otro caso
        NodoExterno.empujar(i);
    fin
fin
```

Algoritmo 8: Algoritmo que determina si un nodo es etiquetado como interno o externo

Al aplicar el algoritmo se solucionaron los casos de caras inválidas derivados de este problema.

3.4.4. Errores de código

En la implementación actual existen algunos errores de código que generan caras inválidas. Se encontró uno en la clase *BoundaryTemplate3*, encargada de implementar los patrones para 3 nodos internos. El patrón 3a no insertaba correctamente una de las pirámides, debido a un error de tipeo en la asignación de nodos en la misma.

Solución

Corregir el error de tipeo.

Otro problema se debe a una implementación sin concluir de patrones de división de pirámides y prismas que se encontraba en el código actual. Esta implementación genera caras inválidas. La división de pirámides y prismas completa ya se encuentra implementada en este trabajo.

Solución

La implementación sin concluir fue removida de *Mixed-elements Octree*. Esta se encontraba en el método *getSubElements* de la clase *EnhancedElement*.

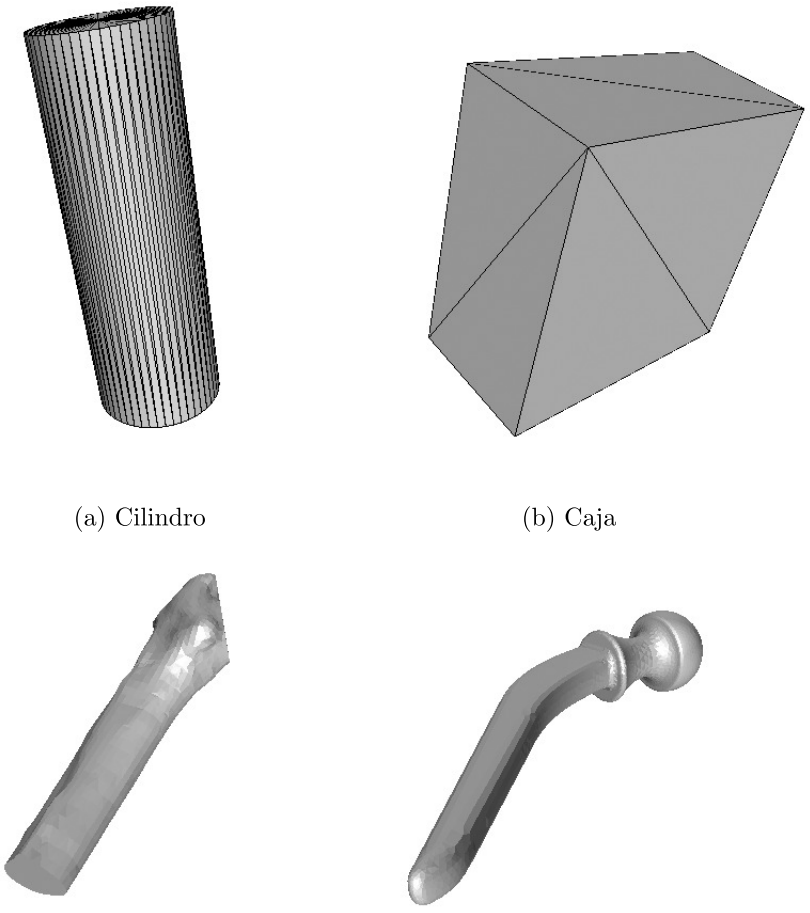
Capítulo 4

Resultados

A partir de la propuesta de solución de la sección anterior, se realizarán pruebas en algunas mallas con distintos niveles de refinamiento, para comparar los resultados de la generación de mallas volumétricas intersectadas, con la solución presentada en este trabajo y sin ella.

Se presentarán imágenes de las mallas tanto externamente como de sus características internas y se extraerá información de ellas, como el tamaño total del archivo, cantidad de nodos finales y cantidad de elementos finales.

La Figura 4.1 presenta los modelos en tres dimensiones de las mallas utilizadas en las pruebas. La malla correspondiente a la caja representa un dominio rectangular y el cilindro un dominio parcialmente curvo. En el caso del fémur y la prótesis, ambos corresponden a dominios altamente complejos y que conllevan mayor dificultad para representar correctamente sus características internas.



(a) Cilindro (b) Caja

(c) Fémur (d) Prótesis

Figura 4.1: Las cuatro mallas de superficie utilizadas en las pruebas.

A continuación, se realizan dos intersecciones: cilindro-caja y fémur-prótesis. La versión actual de *Mixed-elements Octree* no tiene implementada esta característica, que deberá ser incorporada para trasladar este trabajo. A continuación se presentan imágenes de las mallas generadas producto de su intersección con distintos niveles de refinamiento. Se muestra tanto la vista externa de la malla como la vista interna.

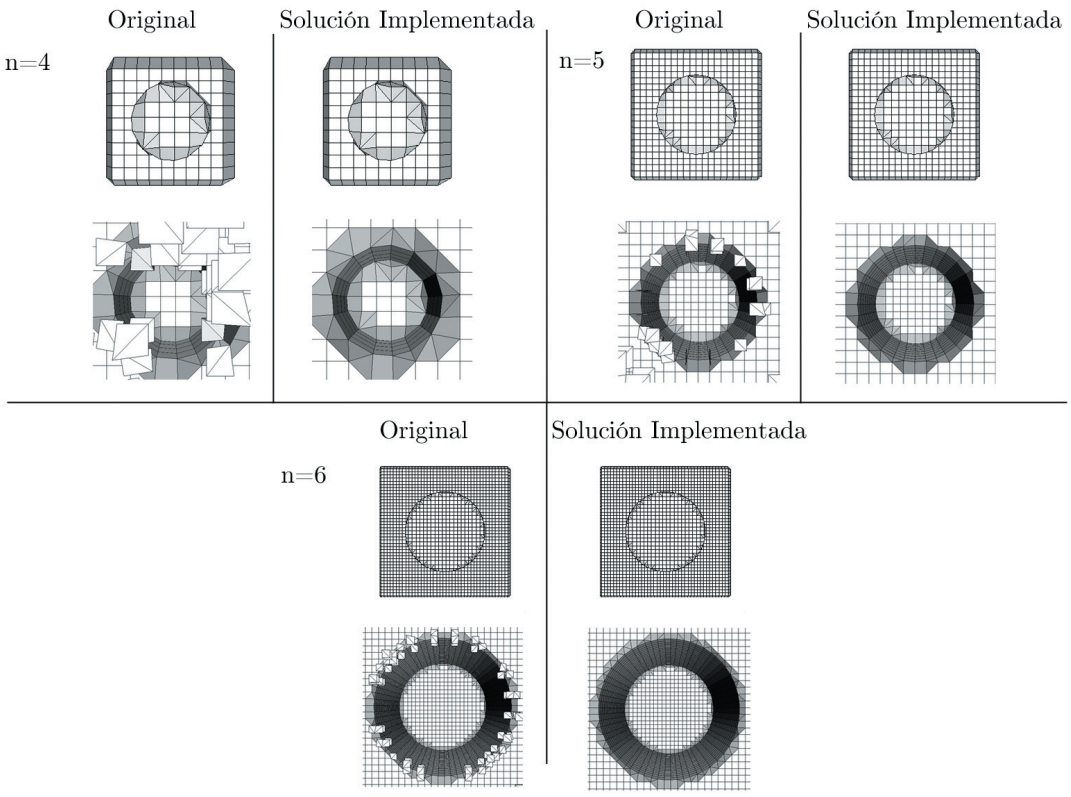


Figura 4.2: Comparación donde n corresponde al nivel de refinamiento de la malla.

Las caras inválidas generadas en la versión original de *Mixed-elements Octree* fueron eliminadas con la solución de este trabajo implementada. La representación externa actual es la correspondiente al estado de los nodos debidamente actualizados, por lo que fue muy ligeramente afectada en algunos casos, ya que cambiaron algunos patrones de superficie que se aplicaban erróneamente en *Mixed-elements Octree* original. Finalmente, no fueron encontrados más casos de caras inválidas con niveles de refinamiento superiores.

La Figura 4.3 presenta la intersección fémur-prótesis.

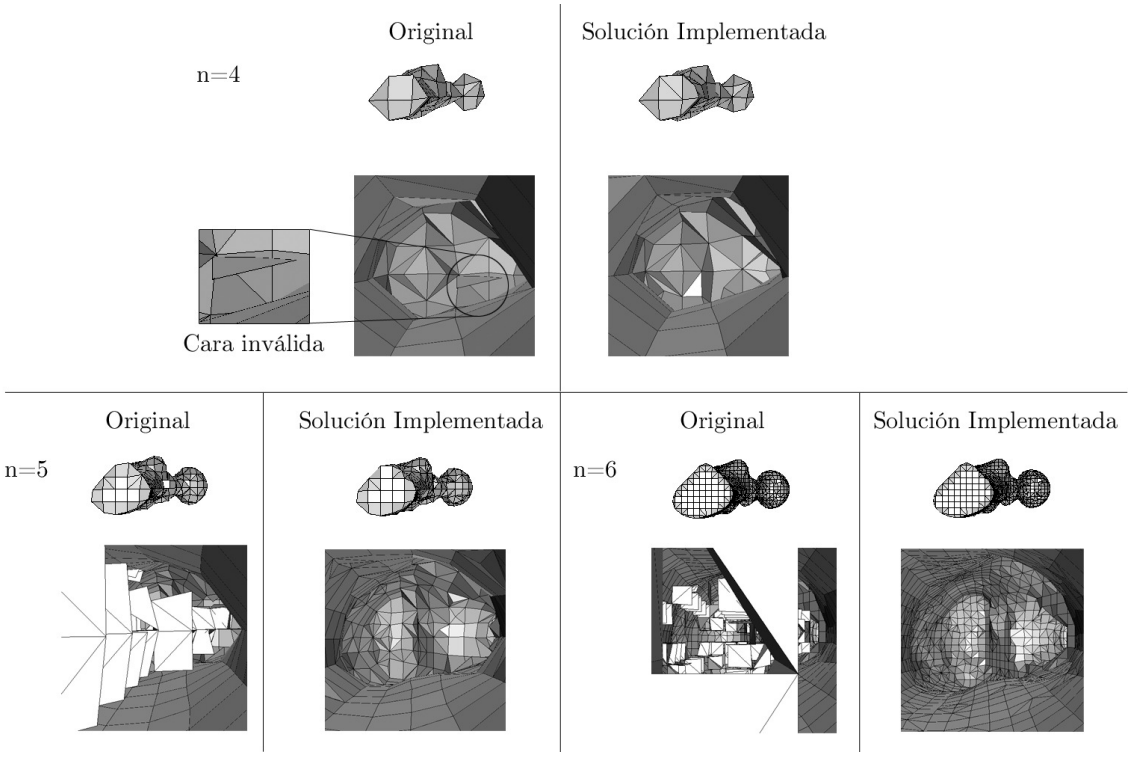


Figura 4.3: Comparación donde n corresponde al nivel de refinamiento de la malla.

En la comparación es posible apreciar que las caras inválidas desaparecen con la solución implementada. La representación externa tuvo más cambios que en el caso cilindro-caja, esto se debe a que se proyectaron más nodos hacia el borde de la superficie, lo que cambió el patrón de superficie aplicado para más hexaedros.

Puede parecer que en el nivel de refinamiento 4 sólo existe una cara inválida solucionada, pero en realidad esto se debe a que en *Mixed-elements Octree* original los patrones aplicados a los elementos internos son incorrectos (revisar sección anterior) y se dio que por casualidad los elementos internos coincidieran topológicamente con los externos en algunos casos. Una vez actualizados los valores del vector border para los elementos, es posible observar que en realidad existen muchos más casos de caras inválidas. La Figura 4.4 compila los tres escenarios, para apreciar la mejora real en la representación interna en este caso.

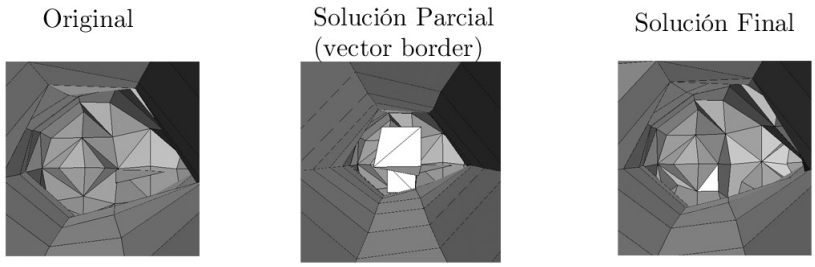


Figura 4.4: Comparación entre la malla generada con la implementación actual de la herramienta, la malla con la corrección del vector border y la malla con la solución propuesta implementada.

4.1. Estadísticas

En esta subsección se realizan comparaciones estadísticas en las mallas de prueba con y sin representación de características internas. Se compara el tamaño final del archivo m3d generado, el tiempo de ejecución, la cantidad de nodos finales y la cantidad de elementos finales.

Las pruebas se realizaron en un sistema operativo Ubuntu de 64 bit con 2048 MB de RAM y un procesador Intel I5-2400 de cuatro núcleos.

4.1.1. Estadísticas de *Mixed-elements Octree* sin representación de características internas

Cuadro 4.1: Estadísticas de la malla cilindro-caja sin la representación de características internas implementada.

n	# nodos	# elementos	Tiempo (s)	Peso (Mb)
4	3064	1309	1	0.2
5	18023	8318	7	1.4
6	133734	58298	46	10.5

Cuadro 4.2: Estadísticas de la malla fémur-prótesis sin la representación de características internas implementada.

n	# nodos	# elementos	Tiempo (s)	Peso (Mb)
4	977	233	17	0.6
5	5493	1811	98	0.3
6	33954	11303	617	2.4

4.1.2. Estadísticas de *Mixed-elements Octree* con representación de características internas

Cuadro 4.3: Estadísticas de la malla cilindro-caja con la representación de características internas implementada.

n	# nodos	# elementos	Tiempo (s)	Peso (Mb)
4	3084	1445	1	0.2
5	18055	8692	10	1.4
6	133812	59285	74	10.5

Cuadro 4.4: Estadísticas de la malla fémur-prótesis con la representación de características internas implementada.

n	# nodos	# elementos	Tiempo (s)	Peso (Mb)
4	1001	465	31	0.06
5	5563	2495	176	0.3
6	34294	14734	1121	2.4

Si bien, el peso del archivo resultante es prácticamente el mismo en todos los casos, se puede apreciar un incremento en la cantidad de nodos, cantidad de elementos y el tiempo de ejecución. Se realiza una comparación gráfica en la Figura 4.5 entre estas tres variables con y sin la solución implementada.

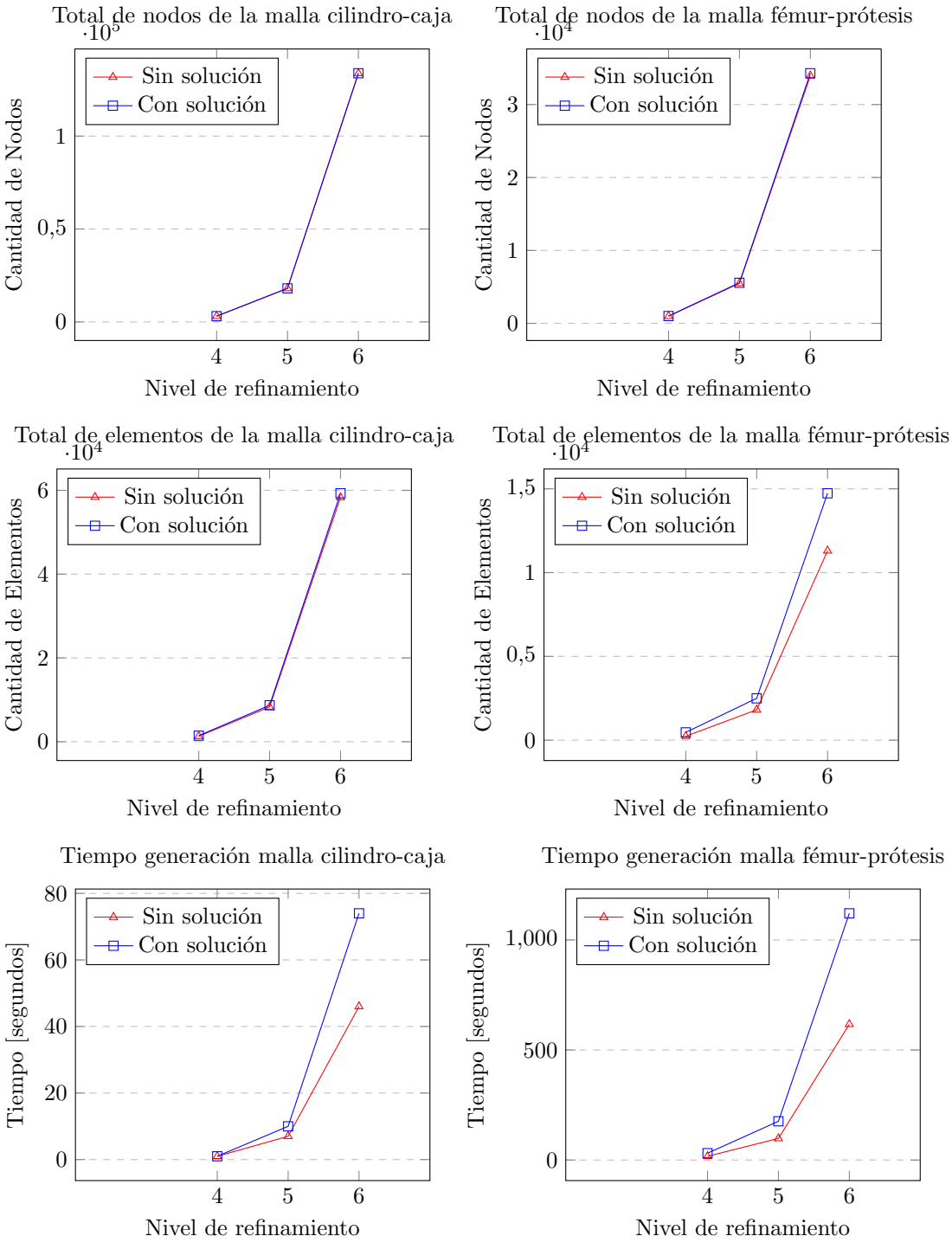


Figura 4.5: Total de nodos, total de elementos y tiempo de generación de las mallas cilindro-caja y fémur-prótesis sin y con la representación de características internas implementada.

Se puede apreciar que el incremento de nodos no es significativo, siendo prácticamente des-

preciable. Esto se debe a que la división del prisma es la única que incrementa el número total de nodos en la solución a la inconsistencia topológica, al no ser un caso muy frecuente no tiene un real impacto en la cantidad de nodos de la malla.

El incremento en la cantidad de elementos en el caso simple de cilindro-caja es despreciable, sin embargo, en el caso complejo fémur-prótesis si se ve un impacto. Esto se debe a que para solucionar la inconsistencia topológica se tuvo que dividir los prismas, pirámides y hexaedros híbridos. Estos casos de inconsistencia se dieron con mucha frecuencia en el caso complejo, por lo que es esperable el impacto que tuvo en la cantidad de elementos de la malla.

Finalmente, el mayor incremento fue el del tiempo de ejecución, el cual es esperable debido al impacto de la solución implementada en *Mixed-elements Octree* y su comportamiento general a mayores niveles de refinamiento.

Conclusiones

En el trabajo presentado se propone un conjunto de soluciones para representar correctamente las características internas de la intersección de dos o más mallas de superficie. Estas fueron capaces de resolver los problemas en la representación interna de la malla resultante de la intersección.

Se realizó un estudio de la herramienta de generación de mallas geométricas *Mixed-elements Octree*, para entender su funcionamiento, el cual consiste en recibir como entrada una o varias mallas de superficie y un nivel de refinamiento a utilizar en la técnica de división, se dividen los octantes y etiquetan los nodos, eliminando los elementos fuera de toda superficie. Se proyectan los nodos cercanos tanto a la superficie externa como interna, se aplican los patrones de representación externa e interna, se vuelven a proyectar los nodos internos y se retorna un archivo con la malla geométrica terminada.

Para la representación interna, el concepto de nodo interno corresponde al nodo que intersecta todas las superficies de la malla, por lo tanto, en estos casos se eliminan todos los nodos que no cumplan dicha condición. Para mantener la consistencia topológica entre octantes vecinos, se definen todas las configuraciones posibles para los nodos internos. Se comprobó que dichas configuraciones estaban bien implementadas en *Mixed-elements Octree*. Osea, no existían orificios al interior de un hexaedro ni superposiciones de los subelementos.

Para identificar los problemas, se visualizó una malla de prueba utilizando la herramienta *GeomView*. Se vio que internamente existían caras flotantes, lo cual es indicador de inconsistencia topológica entre octantes vecinos. Por medio de *debugging*, se pudo aislar los octantes involucrados en la falla. Por medio del estudio de los elementos involucrados, el estado de los nodos y el flujo de código se pudieron identificar las causas de los problemas encontrados. Estas fueron:

- Inconsistencia topológica entre patrones internos y de superficie. Esta se debe a la diferencia conceptual entre patrones externos e internos: para los patrones de superficie los nodos que se encuentran fuera de la malla son los que deben ser eliminados, mientras que para los patrones de representación interna los nodos que no intersectan todas las superficies son los que deben ser eliminados. Esto se produce cuando un patrón interno se aplica a un octante que tiene como vecino un elemento híbrido, es decir, un elemento correspondiente a la representación externa que además tiene características internas.

- Estado de borde desactualizado de nodos y elementos. El vector *border* de la clase *EnhancedElement* se define para cada elemento y representa si este se encuentra dentro o fuera de cada una de las mallas ingresadas. Los valores de dicho vector para cada elemento son definidos antes del procedimiento de proyección de nodos. Esto no está mal ya que estos valores son requeridos antes, pero tras la proyección de nodos dichos valores quedan desactualizados. Esto produce que se aplique un patrón incorrecto a algunos elementos, generando caras inválidas.

- Criterio incorrecto de selección de patrones internos. El algoritmo que determina si un elemento intersecta al menos una superficie está mal implementado, debido a que sólo hace dicha comprobación en una superficie. Debido a esto, se aplican patrones incorrectos en los elementos, lo que genera inconsistencia topológica en algunos casos.

- Errores de código. Errores menores encontrados en la implementación, un error de tipeo y una implementación incompleta de patrones de división de pirámides y prismas que no había sido eliminada.

Las soluciones a dichos problemas corresponden a:

- Dividir los elementos mixtos generados por los patrones de superficie que son vecinos de un elemento interno. Se generó un algoritmo que detectase dichos elementos en función de la cantidad de mallas que intersecta cada nodo, y se reemplazaron por elementos mixtos de manera que en la cara cuadrangular que se identificó como conflictiva se agregara una diagonal en función de sus nodos internos, de forma de mantener la consistencia con el octante vecino interno sin necesidad de conocer el patrón interno que le será implementado.

- Actualizar el estado de la intersección de los elementos a cada malla ingresada, representado por el vector *border*, para evitar la selección errónea de patrones internos que genera inconsistencia topológica.

- Se corrigió el error conceptual de selección de patrones internos existente en la versión actual de *Mixed-elements Octree*. El algoritmo, que sólo tenía en cuenta una malla, se generalizó para n mallas, de forma que la cantidad de nodos internos calculada para la selección de los patrones internos, se haga con el criterio de nodo interno establecido conceptualmente para los mismos, esto es, un nodo interno debe intersectar todas las superficies de la malla.
- Finalmente, se corrigieron algunos errores menores de código que generaban caras inválidas en la representación interna.

A continuación, se realizó una comparación entre las mallas sin representación interna y con ella. Se utilizaron cuatro mallas de prueba: cilindro, caja, fémur y prótesis. Estas mallas se intersectaron de forma de probar dos casos: uno no complejo correspondiente a la intersección cilindro-caja, y uno complejo correspondiente a la intersección fémur prótesis. Se realizaron comparaciones sobre las vistas internas de las mallas, y sobre las estadísticas de los datos más relevantes en los cuales la solución implementada generase cambios: cantidad de nodos, cantidad de elementos, tiempo de ejecución y peso del archivo de malla generado.

Se pudo comprobar que todos los casos de caras inválidas que se habían identificado fueron resueltos. En general, la cantidad de caras inválidas encontradas y solucionadas fue mucho mayor en el caso complejo que en caso no complejo. La representación externa también cambió ligeramente, debido a que se aplicaron los patrones de superficie correspondientes al estado de los nodos actualizado por el vector *border*. No se identificó algún otro problema a nivel visual en las mallas de prueba.

Los resultados de esta comparación también comprobaron que tanto el incremento del peso del archivo generado por *Mixed-elements Octree* como el incremento en la cantidad de nodos es despreciable. Sin embargo, si se debe considerar el incremento en la cantidad de elementos y el tiempo de ejecución, los cuales son significativos.

Trabajo Futuro

Traslado a versión oficial de *Mixed-elements Octree*

La versión de *Mixed-elements Octree* utilizada cuenta con algunas funcionalidades adicionales a última versión oficial. Una de ellas es la capacidad de aceptar n mallas como entrada, a diferencia de la versión oficial que sólo acepta una. Para poder trasladar este trabajo a la

versión oficial que cuenta con la estructura establecida en [2] que mejora la mantenibilidad y escalabilidad de *Mixed-elements Octree*, es necesario trasladar también dicha funcionalidad.

Además, la estructura de *Mixed-elements Octree* establecida en [2] se realiza de forma que las nuevas funcionalidades se incorporen sin modificar el código actual, en una clase adicional llamada patrón visitante. Se recomienda tratar cada solución implementada para un problema en específico como un patrón visitante independiente, ya que no están relacionadas entre ellas y se encuentran implementadas en distintas partes de *Mixed-elements Octree*. Es decir, la solución a la inconsistencia topológica entre patrones internos y externos debería estar en un patrón visitante independiente a la solución al criterio incorrecto de selección de patrones internos. Las otras dos soluciones, el estado de borde desactualizado de nodos y elementos y los errores de código, debieran editarse directamente en el código original, al ser correcciones de partes erróneas.

De forma de hacer más sencillo el traslado, se describen las modificaciones realizadas en cada archivo:

En Mesher.cpp:

- Agregado método *updateLabels*, este se llama en el método *generateMesh* antes de aplicar los patrones de superficie, o sea, anterior a *applySurfaceTemplates(pClientData,pClientPointTestFunc)*.
- En el método *applyBoundaryTemplates*, línea

  ```
  double old_md = elements[i].getMaxDistance();
  ```

  trasladada al inicio del loop *for*
- Agregado script en método *applyBoundaryTemplates*, después de

  ```
  double old_md = elements[i].getMaxDistance
  ```
- Agregada nueva condición en método *applyBoundaryTemplates*

  ```
  surf_conf ==  false
  ```

  antes de

  ```
  for (unsigned int j=0; j<n_meshes; j++)
  ```

y antes de

```
if (!found)
```

En Mesher.h

- Agregado método *updateLabels*

En EnhancedElement.cpp

- En método applyBoundaryTemplates, se eliminaron:

  Código 1:

```
if(!insideBorder(meshpoints)){
                return false;
```

  Código 2:

```
 for (unsigned int i=0; i<pointindex.size(); i++){
 if(meshpoints.at(pointindex[i]).getIOState(0) and
 meshpoints.at(pointindex[i]).getIOState(1))
                        tmpin.push_back(i);
                else
                        tmpout.push_back(i);
                }
```

  Código 3:

```
if (pointindex.size()==5) {
        BoundaryTemplatePyramid btpy;
        return btpy.getSubelements(pointindex,inpts,newsub,newsub_out);
        }


if (pointindex.size()==6) {
        BoundaryTemplatePrism btpr;
        return btpr.getSubelements(pointindex,inpts,outpts,newsub,newsub_out);
}
```

  Agregados *scripts* en reemplazo de Código 2 y 3.

- Agregados métodos *fixconflictingelements, fixpyrPatternA, fixpyrPatternB, fixprismPatternA, fixprismPatternB, fixprismPatternC, fixprismPatternD, fixprismPatternE, fixprismPatternF, fixprismPatternG, fixprismPatternH, fixprismMandatoryElements.*

En EnhancedElement.h:

- Agregados métodos *fixconflictingelements, fixpyrPatternA, fixpyrPatternB, fixprismPatternA, fixprismPatternB, fixprismPatternC, fixprismPatternD, fixprismPatternE, fixprismPatternF, fixprismPatternG, fixprismPatternH, fixprismMandatoryElements.*

En BoundaryTemplate3.cpp:

- En método PatternA, se reemplaza:

```
pyramid2[0] = all[4];
pyramid2[1] = all[7];
pyramid2[2] = all[6];
pyramid2[3] = all[5];
pyramid2[4] = all[2];
```

por,

```
pyramid3[0] = all[4];
pyramid3[1] = all[7];
pyramid3[2] = all[6];
pyramid3[3] = all[5];
pyramid3[4] = all[2];
```

Bibliografía

[1] C. Lobos. A set of mixed-elements patterns for domain boundary approximation in hexahedral meshes. *Studies in health technology and informatics*, 184:268–272, 2013. Proceedings of MMVR20.

[2] Sebastían Tobar. Mejoramiento de la mantenibilidad y extensibilidad de una herramienta de generación de mallas volumétricas. *Memoria para optar al título de Ingeniero Civil en Informática,*. Universidad Técnica Federico Santa María, Abril 2016.

[3] Eugenio González and Claudio Lobos. A set of mixed-element transition patterns for adaptive 3d meshing. Technical Report 2014/01, Departamento de Informática, UTFSM, October 2014.

[4] Hang Si. Tetgen, a delaunay-based quality tetrahedral mesh generator. *ACM Trans. Math. Softw.*, 41(2):11:1–11:36, February 2015.

[5] A. Bowyer. Computing dirichlet tessellations*. *The Computer Journal*, 24(2):162–166, 1981.

[6] Joachim Schöberl. Netgen an advancing front 2d/3d-mesh generator based on abstract rules. *Computing and Visualization in Science*, 1(1):41–52, 1997.

[7] Christophe Geuzaine and Jean-François Remacle. Gmsh: A 3-D finite element mesh generator with built-in pre- and post-processing facilities. *International Journal for Numerical Methods in Engineering*, 79:1309 – 1331, 2009.

[8] K. J. Bathe. Front matter. In K.J. BATHE, editor, *Nonlinear Finite Element Analysis and Adina*. Pergamon, 1983.

[9] Christopher Arenas. Detección y representación de características finas en mallas de volumen tipo octree. *Memoria para optar al título de Ingeniero Civil en Informática,*. Universidad Técnica Federico Santa María, Noviembre 2015.