

UNIVERSIDAD TÉCNICA FEDERICO SANTA MARÍA
DEPARTAMENTO DE INFORMÁTICA
SANTIAGO - CHILE



“OPTIMIZACIÓN DE MÉTODO RÁPIDO DE REPRESENTACIÓN DE CARACTERÍSTICAS FINAS EN UNA MALLA OCTREE”

SEBASTIÁN CASTRO DÍAZ

MEMORIA PARA OPTAR AL TÍTULO DE
INGENIERO CIVIL EN INFORMÁTICA

Profesor Guía: Claudio Lobos
Profesor Correferente: Jorge Díaz Matte

Julio - 2025



CONSTANCIA DE VALIDACIÓN Y CONFIDENCIALIDAD DE MONOGRAFÍA A REPOSITORIO ACADÉMICO

1.- IDENTIFICACIÓN DEL TRABAJO ACADÉMICO

Tipo de monografía (marcar una opción): ☒ Memoria o trabajo de título; ☐ Tesis de Postgrado;

Título del trabajo: OPTIMIZACIÓN DE MÉTODO RÁPIDO DE REPRESENTACIÓN DE CARACTERÍSTICAS FINAS EN UNA MALLA OCTREE

Nombre del candidato(a): Sebastián Castro Díaz

Carrera / Grado: INGENIERO CIVIL EN INFORMÁTICA

Campus: Santiago San Joaquín; **Departamento:** INFORMÁTICA

2.- VALIDACIÓN DEL PROFESOR GUÍA/DIRECTOR DE TESIS

Yo, Claudio Lobos, en mi calidad de profesor(a) guía/director(a) del trabajo académico mencionado anteriormente **DEJO CONSTANCIA** que:

- He revisado esta versión del documento y corresponde a la versión final aprobada del trabajo.
- El trabajo cumple con los requisitos académicos y de formato establecidos por la institución

3.- EVALUACIÓN DE CONFIDENCIALIDAD POR PROPIEDAD INDUSTRIAL

☒ El trabajo **NO contiene información que amerite confidencialidad** y puede ser publicado de inmediato en repositorio con acceso abierto.

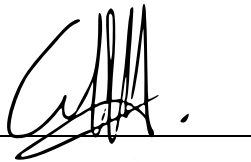
☐ El trabajo **CONTIENE** información con potenciales implicancias de propiedad industrial o intelectual y requiere un periodo de confidencialidad (embargo) por:

☐ 6 meses; ☐ 12 meses; ☐ 2 años; ☐ 3 años; ☐ 5 años; ☐ 10 años

Fundamentación de la necesidad de confidencialidad (obligatorio si se solicita embargo):

4.- FIRMAS

Profesor(a) guía o director(a) de memoria o tesis:

Fecha: 30/07/2025; **Firma:** 

Estudiante o Candidato(a):

Fecha: 30/07/2025; **Firma:** 

Este formulario debe ser insertado como página 2 de la memoria o tesis, completado y firmado por estudiante y profesor(a) antes de la entrega en portal PRISMA de Biblioteca USM.

DEDICATORIA

A todas las personas que estuvieron conmigo en cada paso del camino. Gracias por su apoyo, su paciencia y su cariño incondicional.

RESUMEN

Resumen—

Este trabajo propone una modificación a un algoritmo de generación de mallas volumétricas para mejorar la representación de características finas en una figura. La principal contribución consiste en modificar el proceso de proyección de nodos: al proyectar un nodo sobre la superficie, se ajustan también los nodos vecinos para evitar la deformación del octante. Además, se introdujeron dos nuevos patrones de superficie que permiten transformar ciertas regiones en prismas triangulares cuando es necesario. Los resultados muestran una mejora en la representación visual de características finas, con un aumento leve en el tiempo de generación de malla y una ligera disminución en la calidad promedio resultante, ambos considerados insignificantes. No obstante, el método presenta limitaciones al aplicarse a geometrías más complejas, lo que indica la necesidad de continuar desarrollando el algoritmo mediante nuevos patrones, mejoras en proyección y otras estrategias de refinamiento.

Palabras Clave— Malla octree; Patrones de superficie; Características Finas

ABSTRACT

Abstract— This work proposes a modification to a volumetric mesh generation algorithm to improve the representation of fine features in a figure. The main contribution consists of modifying the node projection process: when projecting a node onto the surface, neighboring nodes are also adjusted to prevent octant deformation. Additionally, two new surface patterns were introduced, allowing certain regions to be transformed into triangular prisms when necessary. The results show an improvement in the visual representation of fine features, with a slight increase in mesh generation time and a slight decrease in the resulting average quality—both considered insignificant. However, the method presents limitations when applied to more complex geometries, indicating the need to continue developing the algorithm through new patterns, improvements in projection, and other refinement strategies.

Keywords— Octree mesh; Surface patterns; Fine features

GLOSARIO

J_{ENS} = element normalized scaled Jacobian

J_S = scaled Jacobian

ÍNDICE DE CONTENIDOS

RESUMEN	III
ABSTRACT	III
GLOSARIO	IV
ÍNDICE DE FIGURAS	VII
ÍNDICE DE TABLAS	IX
INTRODUCCIÓN	1
CAPÍTULO 1: DEFINICIÓN DEL PROBLEMA	2
1.1 DEFINICIÓN	2
1.2 OBJETIVOS	3
1.2.1 Objetivo general	3
1.2.2 Objetivos específicos	3
CAPÍTULO 2: MARCO CONCEPTUAL	4
2.1 Malla volumétrica	4
2.2 Método Octree	5
2.3 Mallas mixtas	6
2.4 Características finas	6
2.5 Métrica para la calidad de los elementos	7
2.6 Patrón de transición	8
2.7 Patrón de superficie	9
2.8 Heurística	10
2.9 Algoritmo actual	10
CAPÍTULO 3: PROPUESTA DE SOLUCIÓN	12
3.1 Método isItIn	12
3.2 Método projectCloseToBoundaryNodes	13
3.3 Método getHexInfo	17
3.4 Método deleteVolume	19
3.5 Método allJENS_sharp_points	21
3.6 Método orderedOctantGenerator	22
3.7 Método applySurfacePatterns	23
CAPÍTULO 4: VALIDACIÓN DE LA SOLUCIÓN	41
4.1 Hex Wedge	42
4.2 Fiducials	44
4.3 Hex	49
4.4 Socket	51

4.5 Cortex	58
CAPÍTULO 5: CONCLUSIONES	60
REFERENCIAS BIBLIOGRÁFICAS	65
ANEXOS	66

ÍNDICE DE FIGURAS

1	Árbol del problema	4
2	Malla volumétrica [2]	5
3	Poliedros (a) hexaedro, (b) cuña, (c) pirámide, (d) tetraedro [6]	5
4	Método octree aplicados sobre un cubo[6]	6
5	Ejemplo de cerebro modelado como malla mixta [4]	6
6	Ejemplo de un cubo donde se suavizan características finas [6]	7
7	Hexaedro con nodo a , b , c y p [6]	8
8	Uso de patrón de transición sobre poliedro B, para conexión entre poliedro A y C [4]	9
9	Uso de patron de superficie sobre un hexaedro, en una superficie definida [4]	9
10	Octantes con nodos 0 y 6 marcados (elaboración propia)	12
11	Ejemplo de hex wedge sin aplicar projectCloseToBoundaryNodes (elaboración propia)	13
12	Hexaedro ilustrando la proyección de un nodo (elaboración propia)	14
13	Hexaedro ilustrando la máxima distancia permitida (elaboración propia)	14
14	Ejemplo de hex wedge al aplicarle projectCloseToBoundaryNodes sin modificar (elaboración propia)	15
15	Ejemplo de un cuña hexagonal al aplicarle projectCloseToBoundaryNodes modificado (elaboración propia)	16
16	Tetraedros para el calculo del volumen (elaboración propia)	18
17	Tetraedro con nodos 1, 4, 5 y 6 (elaboración propia)	18
18	Ejemplo octantes que se eliminaran en figura socket (elaboración propia)	20
19	Ejemplo octantes eliminados en figura socket (elaboración propia)	21
20	Resultado de aplicar patrón con un solo nodo interno (elaboración propia)	24
21	Aplicación del patrón para un solo nodo interno con nodos 4 y 7 de mala calidad (elaboración propia)	24
22	Aplicación del patrón para un solo nodo interno con nodos 0, 3, 4 y 7 de mala calidad (elaboración propia)	25
23	Aplicacion de PatternOct1 (elaboración propia)	25
24	Casos donde se debe aplicar el patternOct1 (elaboración propia)	26
25	Aplicación del patternOct1 (elaboración propia)	27
26	Aplicacion de PatternOct2 (elaboración propia)	29
27	Casos donde se debe aplicar el patternOct2 (elaboración propia)	30
28	Casos donde se debe aplicar el patternOct2 (elaboración propia)	30
29	Aplicación del patternOct2 (elaboración propia)	31
30	Figura con octantes A y B afectados	33
31	Patrón aplicado sobre A y B (elaboración propia)	34
32	Patrón aplicado sobre A, pero no sobre B (elaboración propia)	34
33	Tetraedro con los nodos marcados (elaboración propia)	36
34	Octante con nodos internos 0 y 1 (elaboración propia)	36
35	Resultado de aplicación de patrón A cuando hay dos nodos internos (elaboración propia)	37

36	Octantes descartados (elaboración propia).	39
37	Figura con octantes descartados (elaboración propia).	40
38	Figuras utilizadas para probar la solución.	41
39	Resultado Hex Wedge con algoritmo modificado (elaboración propia).	43
40	Resultado Hex Wedge con algoritmo modificado. Refinamiento 3, tolerancia 0.8 (elaboración propia).	43
41	Resultado Hex Wedge con algoritmo original. Refinamiento 3, tolerancia 0.8 (elaboración propia).	44
42	Resultado Fiducials con algoritmo modificado. Refinamiento 3 (elaboración propia).	45
43	Resultado Fiducials con algoritmo modificado. Refinamiento 4 (elaboración propia).	46
44	Resultado Fiducials con algoritmo modificado. Refinamiento 5 (elaboración propia).	46
45	Resultado Fiducials con algoritmo modificado. Refinamiento 3, tolerancia 0.8 (elaboración propia).	47
46	Resultado Fiducials con algoritmo original. Refinamiento 3, tolerancia 0.5 (elaboración propia).	48
47	Interior Fiducials. Refinamiento 3, tolerancia 0.5 (elaboración propia).	49
48	Resultado Hex con algoritmo modificado. (elaboración propia).	50
49	Resultado Hex con algoritmo original. (elaboración propia).	50
50	Resultado Socket con algoritmo modificado. Refinamiento 3 y tolerancia 0.2 (elaboración propia).	52
51	Resultado Socket con algoritmo modificado. Refinamiento 3 y tolerancia 0.5 (elaboración propia).	52
52	Resultado Socket con algoritmo modificado. Refinamiento 4 y tolerancia 0.5 (elaboración propia).	53
53	Resultado Socket con algoritmo modificado. Refinamiento 4 y tolerancia 0.8 (elaboración propia).	54
54	Resultado Socket con algoritmo modificado. Refinamiento 5 y tolerancia 0.5 (elaboración propia).	54
55	Resultado Socket con algoritmo original. Refinamiento 3 y tolerancia 0.2 (elaboración propia).	55
56	Resultado Socket con algoritmo original. Refinamiento 3 y tolerancia 0.5 (elaboración propia).	56
57	Resultado Socket con algoritmo original. Refinamiento 4 y tolerancia 0.5 (elaboración propia).	56
58	Resultado Socket con algoritmo original. Refinamiento 4 y tolerancia 0.8 (elaboración propia).	57
59	Resultado Socket con algoritmo original. Refinamiento 5 y tolerancia 0.5 (elaboración propia).	57
60	Resultado Cortex con algoritmo modificado. (elaboración propia).	58
61	Resultado Cortex con algoritmo original (elaboración propia).	59

62	Interior figura cortex con patrón de transición mal aplicado (elaboración propia).	62
63	Figura socket con patrones de superficie aplicados (elaboración propia).	63
64	Patrones de superficie aplicados a octantes según la cantidad de nodos internos.	66

ÍNDICE DE TABLAS

1	Valores para el conjunto de pruebas.	42
2	Comparación de calidad y tiempo con algoritmo modificado (Hex Wedge).	42
3	Comparación de calidad y tiempo con algoritmo original (Hex Wedge).	44
4	Comparación de calidad y tiempo con algoritmo modificado (Fiducials).	45
5	Comparación de calidad y tiempo con algoritmo original (Fiducials).	47
6	Comparación de calidad y tiempo con algoritmo modificado (Hex).	49
7	Comparación de calidad y tiempo con algoritmo original (Hex).	50
8	Comparación de calidad y tiempo con algoritmo modificado (Socket).	51
9	Comparación de calidad y tiempo con algoritmo original (Socket).	55
10	Comparación de calidad y tiempo con algoritmo modificado (Cortex).	58
11	Comparación de calidad y tiempo con algoritmo original (Cortex).	58

INTRODUCCIÓN

Las mallas volumétricas son un conjunto de poliedros, que forman una figura. Estas figuras pueden ser de distintas índoles, poseyendo geometrías variadas, y tamaños variados. Éstas, posteriormente se usan en simulaciones de una situación real, con interacciones físicas. Dado esas interacciones se requieren que posean volumen para un comportamiento más real.

Los algoritmos que generan una malla volumétrica suelen requerir indicar el grado de representación de ésta. Esto se refiere a cuanta precisión tendrá la malla resultante. Mientras más grande, es más fiel a la figura inicial y conformada por más poliedros. Cuando este valor es grande, el tiempo de generación también es elevado, por lo que se tiene que decidir el grado más adecuado, para tener un tiempo aceptable y que se obtenga una figura suficientemente acorde a la original.

El presente trabajo utiliza un el generador de elementos mixtos de Lobos y González. Éste utiliza el método octree, que va dividiendo la figura en octantes, y descartando los que no necesita, y luego utiliza patrones de superficie, para adecuarla a la forma del objeto, pero enfocado en geometrías curvas, junto con la heurística de Figueroa que filtra la aplicación de patrones de superficie, para mejorar la representación de geometrías rectas, que se conocen como características finas.

El objetivo del trabajo es mejorar la representación de las características finas, al adecuar el método que se encarga de sacar octantes que poseerán poco volumen, y analizar el algoritmo actual para buscar los problemas que impiden que se obtengan las características finas y corregirlos. Con el principal fin que mejore visualmente, sin afectar en tiempo y calidad.

La estructura de esta memoria se divide en 5 capítulos. En el primero **definición del problema** se aborda el problema y se establece el objetivo principal y los objetivos secundarios para llevarlo a cabo. En el segundo **marco conceptual** se explican los más relevantes, para contextualizar y tener claridad sobre lo desarrollado, como las mallas volumétricas, método octree, métricas de calidad, entre otros. En el tercero **propuesta de solución** se presenta la solución desarrollada, acompañado de imágenes de donde se aplicarían y sus resultados esperados, en conjunto de un pseudocódigo para mostrar su implementación. En el cuarto **validación de la solución** se muestran cuáles pruebas se realizaron en conjunto con sus resultados, de forma visual y en conjunto con los datos más relevantes de éstas y por último en el quinto **conclusiones** se muestra un análisis sobre lo realizado y se habla sobre el trabajo futuro.

CAPÍTULO 1

DEFINICIÓN DEL PROBLEMA

1.1. DEFINICIÓN

El área de la computación gráfica, corresponde al campo de la informática visual [3], donde se usan las computadoras para crear y manipular imágenes [5]. La principal categoría donde se ubica el problema que se analizará es en el modelamiento de las propiedades y formas de un objeto para que se pueda almacenar y visualizar [5]. Para modelar los objetos se usan mallas geométricas que corresponden a un conjunto de distintos polígonos. Más exactamente, se usará una malla volumétrica que es un conjunto de distintos poliedros (cuerpos geométricos).

Se busca emplear el modelamiento en el tema de la simulación, donde las principales dificultades son modelar objetos de forma precisa, es decir con mayor detalle, y que se haga de forma rápida, en un corto periodo de tiempo. Estos son incompatibles, debido a que cuando se enfoca en la precisión, no se considera el tiempo que se demora en generarse, dando tiempos elevados, mientras que cuando se enfoca en el tiempo, se acota al solo considerar la precisión donde se centra la simulación, por lo que se tiene que hacer un trade-off de estos [4].

La solución creada por Lobos y González está enfocada en la biomedicina. Logró una alta precisión en tiempo real, siendo útil para generar partes del cuerpo humano que permitan recrear su comportamiento y realizar simulaciones, como cirugías [4]. Esta solución utiliza el método Octree, que consiste en representar un objeto a través de octantes de distintos tamaños; aquí un octante corresponde a un hexaedro. Se comienza definiendo un conjunto de octantes sobre la forma del objeto a modelar; cada octante se subdivide en ocho nuevos octantes, y luego se compara cada parte para determinar si está dentro del objeto, fuera del objeto o parcialmente dentro de él. Las partes vacías y llenas se marcan, y en aquellas que contienen el objeto parcialmente, se repite el proceso de subdivisión en ocho hasta alcanzar la resolución deseada [4][8]. Además, se considera generar una malla conforme, es decir, que cuando dos caras de los octantes coinciden, deben alinearse perfectamente [1]. Para esto se hace que la malla esté balanceada, es decir, que en el nivel de refinamiento entre octantes no puede diferir en más de uno y, en caso de que pase, los octantes que difieran se deben dividir para que se cumpla esto y la división de un octante en un conjunto de poliedros para que, en caso de que haya una cara que se dividió, la cara vecina también se pueda dividir y hacer que coincidan. También se aplica la transición a elementos mixtos, es decir, transformar los octantes a otros poliedros. Esto se hace en dos casos:

- La ya mencionada, que se utiliza para lograr la correcta conexión entre una parte que se dividió y otras que no, por ejemplo, se tiene una parte que se dividió en 8 hexaedros, mientras que la parte vecina no lo hizo, pero comparte nodos definidos por el vecino. A su vez, esta parte tiene otro vecino que tampoco se dividió, con el cual también

comparte nodos. Este elemento central, para hacer calzar los nodos que poseen de sus dos vecinos, realizará una transición a una división de tetraedros. Siendo este el patrón de transición de los elementos [4].

- Para una mejor representación sobre la superficie, dado que existen casos donde el hexaedro puede producir una mala calidad, siendo necesario utilizar otro cuerpo geométrico para tener una malla válida. Siendo este el patrón de superficie [4].

Los poliedros que se usaron, aparte del hexaedro, son el prisma (cuña), pirámide y tetraedro. Estos se aplican una vez que el método Octree ha finalizado. Primero se aplica el patrón de transición y, a continuación, el patrón de superficie [4].

Se utilizará esta solución de base para poder resolver el problema de simular todo tipo de objetos, y así poder hacer diversos tipos de simulaciones en tiempo real, más allá del área de la biomedicina. La dificultad que se encuentra es que esta solución, al estar desarrollada para modelar órganos, está orientada a generar mallas curvas, por lo que no es capaz de generar correctamente objetos que presentan partes rectas, ya que dichas partes son suavizadas. Cabe destacar que este problema ya ha sido abordado por Figueroa a través de una heurística que no afecta en gran medida el tiempo ni la calidad de la solución de Lobos y González. La heurística consiste en crear un umbral de calidad para que se apliquen los patrones de superficie. Es decir, si cierto elemento está sobre el umbral de calidad, no se aplicarán los patrones y se mantendrá. Este cambio aún presenta fallas, debido a que no representa correctamente los ángulos que poseen cierto grado de inclinación, ni tampoco los ángulos rectos que presentan una curvatura [6].

Por lo que para poder resolverlo en su totalidad se tienen los siguientes objetivos:

1.2. OBJETIVOS

1.2.1. Objetivo general

- Implementar un método que represente correctamente las características finas de una malla tipo octree, considerando la calidad de todos los poliedros que intervienen en el proceso.

1.2.2. Objetivos específicos

- Optimizar el posicionamiento de los nodos internos de acuerdo a los nodos externos que fueron proyectados.
- Análisis y corrección de problemas detectados en el algoritmo original.

A continuación se presenta el árbol del problema:

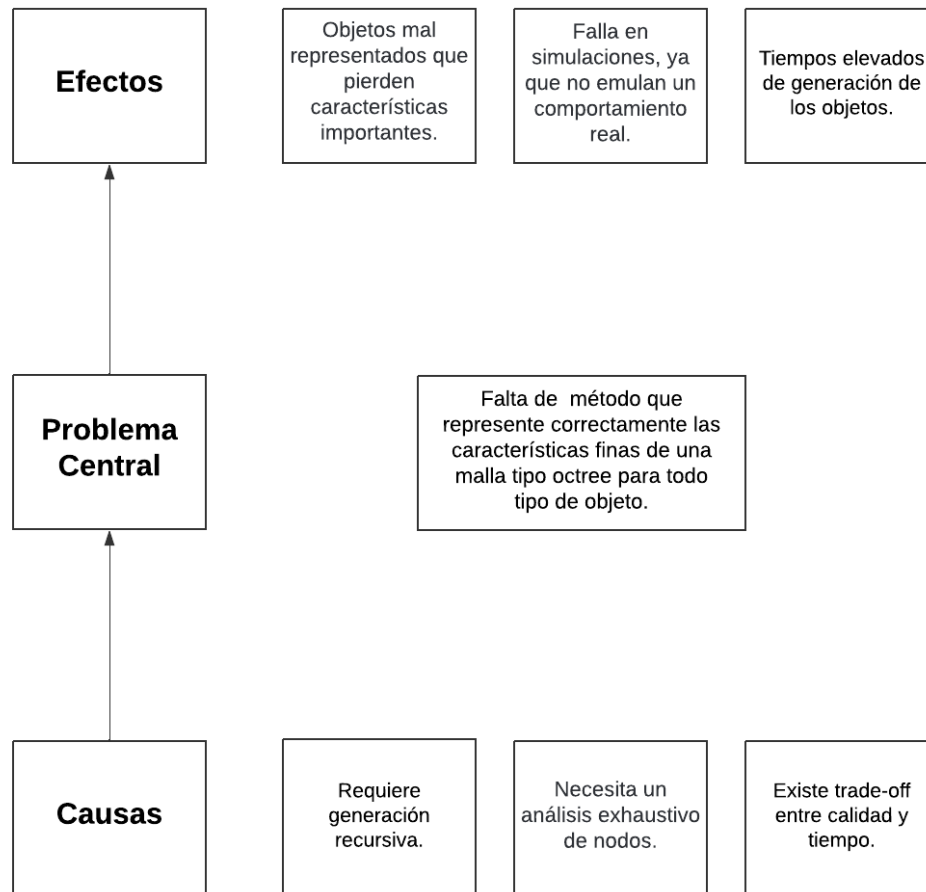


Figura 1: Árbol del problema

CAPÍTULO 2

MARCO CONCEPTUAL

2.1. Malla volumétrica

Se define una malla volumétrica como un conjunto de poliedros del mismo o de distinto tipo (en el caso de ser de distinto tipo también se considera una malla mixta). Estos poliedros pueden ser hexaedros, prismas (cuñas), pirámides, tetraedros, entre otros. Cada poliedro está definido por un conjunto de nodos en un espacio determinado, y estos nodos los pueden compartir con otros poliedros. Cabe destacar que la malla volumétrica no solo corresponde al cascarón de una figura, sino que incluye todo su interior. En la Figura 2 se observa una malla

volumétrica compuesta por tetraedros, y en la Figura 3 se muestran los distintos poliedros que pueden componer una malla volumétrica.

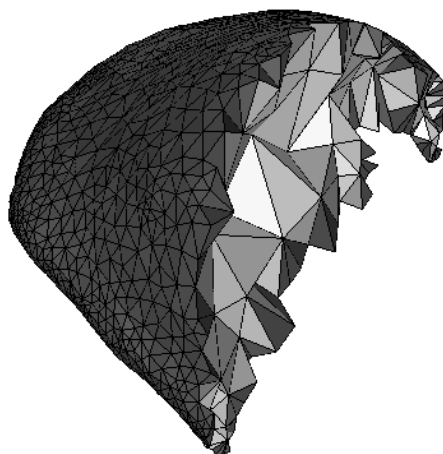


Figura 2: Malla volumétrica [2]

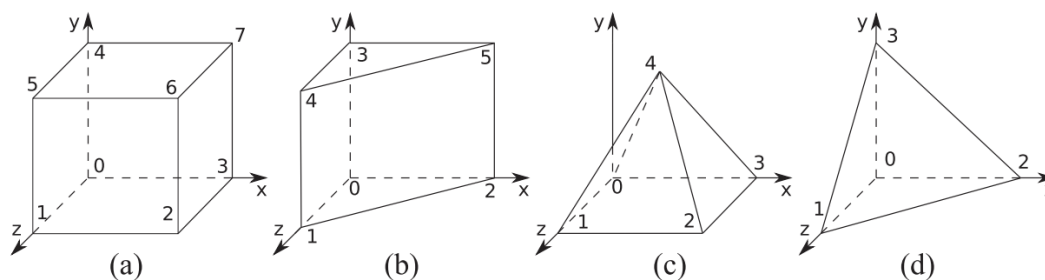


Figura 3: Poliedros (a) hexaedro, (b) cuña, (c) pirámide, (d) tetraedro [6]

2.2. Método Octree

El método Octree recibe como entrada un cascarón del objeto del que se desea obtener la malla volumétrica. Se define un conjunto de octantes de partida, los que contienen al cascarón que define el dominio, y estos se van dividiendo recursivamente hasta alcanzar la resolución deseada. Entre mayor sea la resolución deseada, mayor será la recursividad, generando octantes muy pequeños. Además, entre mayor sea la recursividad, mayor será el tiempo que se tome, siendo para algunas resoluciones inviable generarlas sin los recursos computacionales adecuados. En la Figura 4 se muestra un ejemplo del método octree aplicado con poliedros tipo hexaedro, junto con un árbol que ejemplifica las subdivisiones.

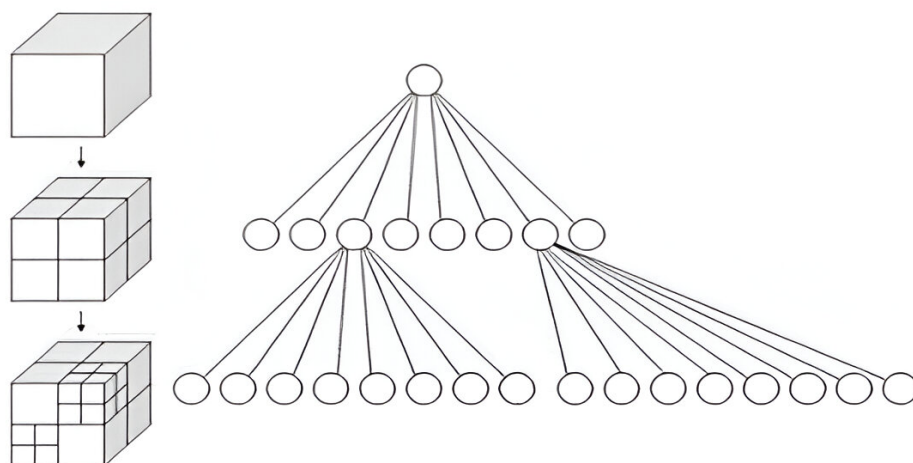


Figura 4: Método octree aplicados sobre un cubo[6]

2.3. Mallas mixtas

Las mallas mixtas consisten en mallas compuestas por diferentes tipos de poliedros, como ya se mencionó, estos son hexaedros, prismas (cuñas), pirámides y tetraedros, entre otros. En la Figura 5 se muestra un ejemplo de una malla mixta.

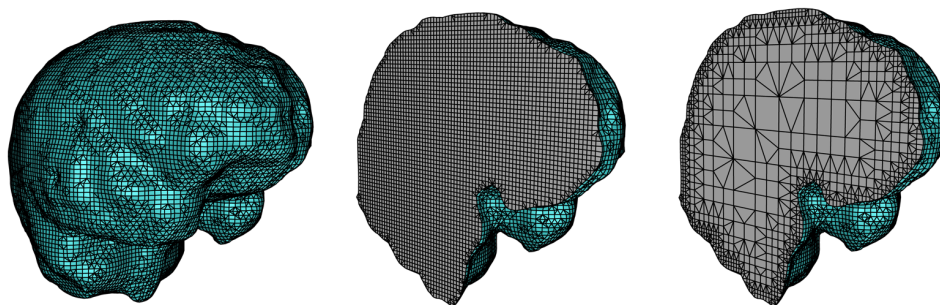
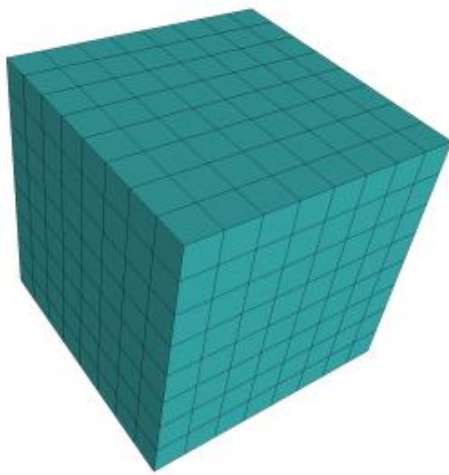


Figura 5: Ejemplo de cerebro modelado como malla mixta [4]

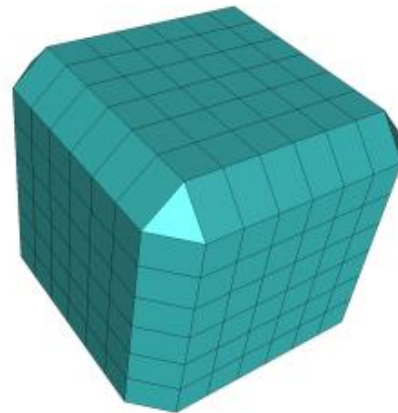
2.4. Características finas

Las características finas corresponden a regiones que tienen un ángulo definido en una malla volumétrica; estos son parte del objeto original, al que se le genera la malla y, por tanto, no deberían suavizarse.

A continuación se muestra un ejemplo de una malla volumétrica en forma de cubo con bordes y esquinas definidos, las cuales corresponden a las características finas. Junto a ella, se presenta otra malla volumétrica en forma de cubo con bordes y esquinas, pero suavizadas [6].



(a) Malla con características finas



(b) Malla con características finas suavizadas

Figura 6: Ejemplo de un cubo donde se suavizan características finas [6]

2.5. Métrica para la calidad de los elementos

La métrica que se utiliza en la solución para medir la calidad de los poliedros generados es el jacobiano escalado normalizado (J_{ENS}), basado en el jacobiano escalado (J_S), que corresponde a una métrica para medir la calidad de un hexaedro.

El jacobiano escalado es el mínimo valor del cálculo de los J_S^x , siendo este último calculado para cada uno de los nodos del hexaedro. J_S^x se calcula a través del producto punto y producto cruz de los vectores normalizados adyacentes al nodo x . A continuación, se realiza un ejemplo de dicho cálculo, respecto a la figura 7, que corresponde a un hexaedro con los nodos, a , b , c y p :

$$J_S^p : \text{Jacobiano escalado para el nodo } p \quad (1)$$

$$J_S^p = \hat{v}_{pa} \cdot (\hat{v}_{pc} \times \hat{v}_{pb}) = \hat{v}_{pb} \cdot (\hat{v}_{pa} \times \hat{v}_{pc}) = \hat{v}_{pc} \cdot (\hat{v}_{pb} \times \hat{v}_{pa}) \quad (2)$$

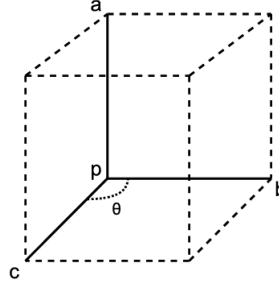


Figura 7: Hexaedro con nodo a, b, c y p [6].

El valor resultante en J_S es menor o igual a 1. Cuando es <0 significa que el poliedro es inválido, cuando es $0.2 < J_S \leq 1$ es razonablemente bueno, y cuando es igual a 1 es perfecto.

Esta métrica es válida para medir la calidad de los hexaedros, pero no para el resto de poliedros. Entonces lo que se hace es normalizar J_S para todos los poliedros, dando J_{ENS}

Para calcular el J_{ENS} , primero se calcula el J_{ENS}^n para cada nodo:

$$J_{ENS}^n = \begin{cases} (1 + k^e) - J_S & \text{si } J_S > k^e \\ J_S / k^e & \text{si } -k^e \leq J_S \leq k^e \\ -(1 + k^e) - J_S & \text{si } J_S < -k^e \end{cases}$$

Siendo el valor k^e una constante determinada para el tipo de poliedro, para tetraedros y pirámides, este toma el valor de $\sqrt{2}/2$, para los prismas toma el valor de $\sqrt{3}/2$ y para hexaedros toma el valor de 1.

Con esto, J_{ENS} viene definida por:

$$J_{ENS} = \begin{cases} \min\{J_{ENS}^i\} & \text{si } \forall J_{ENS}^i > 0 \\ \max\{J_{ENS}^a\} : J_{ENS}^a < 0 & \text{si } \exists J_{ENS}^i > 0 \end{cases}$$

El cual mantiene que si el resultado es <0 significa que el poliedro es inválido, cuando es $0.2 < J_{ENS} \leq 1$ es razonablemente bueno, y cuando es igual a 1 es perfecto. Por lo tanto, esta expresión será la encargada de definir la calidad de los poliedros [4][6].

2.6. Patrón de transición

Los patrones de transición son algoritmos diseñados para reemplazar poliedros, con el objetivo de asegurar una conexión adecuada con los poliedros vecinos. Estos se aplican

cuando un poliedro presenta una diferencia en el número de nodos respecto a sus vecinos, lo que requiere cambiarlo por otro para garantizar una conexión correcta con los poliedros adyacentes.

Por ejemplo, en la figura 8, los nodos proyectados del conjunto de hexaedros A sobre el hexaedro B son más numerosos que los que posee B. Por lo tanto, se aplica un patrón de transición para modificarlo, de modo que pueda conectarse correctamente con el hexaedro C. Como resultado, el hexaedro B se transforma en un conjunto de pirámides y tetraedros.

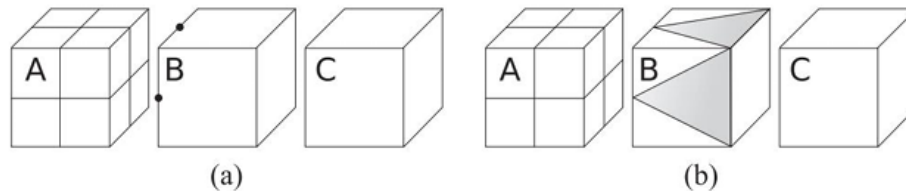


Figura 8: Uso de patrón de transición sobre poliedro B, para conexión entre poliedro A y C [4]

2.7. Patrón de superficie

Los patrones de superficie son algoritmos para reemplazar los poliedros que se encuentran en la superficie. Lo que se busca con esto es que los poliedros se adapten con una mejor calidad a la superficie, al proyectar los nodos externos sobre ésta. Según la calidad resultante tras la proyección, se determina si debe ser reemplazado o no, por un poliedro más adecuado. Se reemplaza si posee una mala calidad.

Por ejemplo, en la figura 9, los nodos se proyectan sobre la superficie, pero al evaluar la calidad, se determina que esta es mala. Por lo tanto, se reemplaza el hexaedro por un poliedro más adecuado, que en este caso corresponde a la cuña.

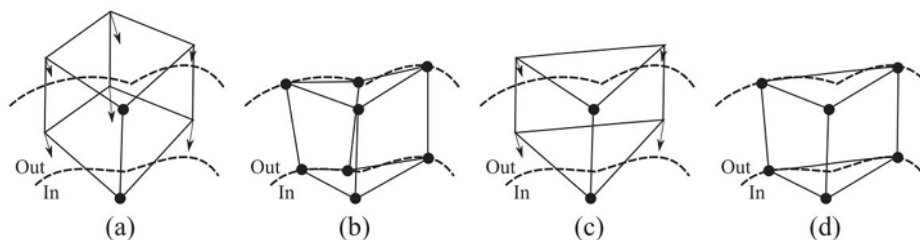


Figura 9: Uso de patron de superficie sobre un hexaedro, en una superficie definida [4]

2.8. Heurística

La heurística utilizada calcula el valor de J_{ENS}^n para todos los nodos de un poliedro exterior y el resultado de cada uno se compara con el valor de una tolerancia entregada por el usuario. Si es inferior, un atributo booleano del nodo cambia a falso. El atributo se llama qualityTol.

Este valor almacenado se utiliza para determinar si se deben realizar los patrones de superficie. Cuando se tiene que uno de los nodos posea el atributo con valor falso, se utiliza el patrón; de otro modo, significa que cumple con la calidad.

El algoritmo que calcula el valor de qualityTol por nodo, es el siguiente [6]:

Algorithm 1 Cómputo de J_{ENS} para nodos

```
1: Input:  $V_E$  vector de elementos
2: for each  $E$  in  $V_E$ 
3:   resultadosJENS  $\leftarrow$  calcularJENS( $E$ )
4:   for each  $p$  in resultadosJENS
5:     if resultadosJENS( $p$ ) < tol then
6:        $p$ .qualityTol  $\leftarrow$  false
7:     end if
8:   end for
9: end for
```

2.9. Algoritmo actual

El algoritmo actual consiste en [6]:

Algorithm 2 Método rápido de representación de características finas

```

1: Input:  $SM$  malla de superficie,  $q$  parámetro de calidad
2: Output:  $VM$  malla de volumen
3: Octantes  $\leftarrow$  generarOctantes( $SM$ )
4: Octantes.removeOctantesExternos
5: clasificarNodos( $SM$ )
6: proyectarNodosInternos( $SM$ )
7: eliminarElementosExternos( $SM$ )
8: proyectarNodosExternos( $SM$ )
9: calcularCalidadHexaedrosSuperficiales()
10: for Elemento in Octantes
11:   if Elemento.calidad  $\geq q$  then
12:     continue
13:   else
14:     aplicarPatronesSuperficie()
15:     break
16:   end if
17: end for
18: eliminarElementosExternos( $SM$ )
19:  $VM \leftarrow$  guardarMallaResultante( $SM$ )

```

Los métodos más importantes que cumplen las funciones que se muestran en el pseudo-código se llaman generateOctreeMesh, projectCloseToBoundaryNodes, removeOnSurface, shrinkToBoundary, elementsJENS y applySurfacePatterns, donde en el método applySurfacePatterns, es en el cual se aplicó la heurística. Este va revisando octante por octante y verifica si este presenta un nodo con una calidad inferior a la tolerancia, y si es así procede a aplicarle el patrones de superficie. A todos estos se les tiene que llevar a cabo un analisis para lograr cumplir los objetivos planteados.

El algoritmo, en el peor de los casos, es tan solo un 1 % más lento que el algoritmo original. Esto se logra, ya que a pesar de que se introdujeron cálculos adicionales en la heurística, se compensa porque no se aplican los patrones de superficie a todos los elementos superficiales, ahorrando así tiempo y manteniendo su objetivo inicial [6].

CAPÍTULO 3

PROPUESTA DE SOLUCIÓN

Para desarrollar la solución, se llevó a cabo un análisis exhaustivo del funcionamiento de cada método que se utiliza. Los principales métodos son `generateOctreeMesh`, `ProjectCloseToBoundaryNodes`, `removeOnSurface`, `ShrinkToBoundary`, `elementsJENS`, y `applySurfacePatterns`. De estos, los que requieren ser modificados son `ProjectCloseToBoundaryNodes`, `applySurfacePatterns`, el método `isItIn` que se utiliza en `generateOctreeMesh` y el método `allJENS_sharp_points` que se utiliza en `elementsJENS`. Asimismo, se implementaron los métodos `getHexInfo`, `deleteVolume` y `orderedOctantGenerator`.

3.1. Método `isItIn`

El método `isItIn` se utiliza en `generateOctreeMesh` para indicar si un octante en la malla está dentro o fuera de la figura. Para realizar esto, primero toma como referencia los nodos 0 y 6, los que tienen las coordenadas mínimas y máximas en los tres ejes respectivamente. Para cada uno, determina la distancia que tienen a la malla input. Dado que, por convención, todas las normales de las caras del input apuntan hacia afuera, si la distancia es positiva, se dirá que el punto está fuera, de lo contrario, estará dentro. Cuando ambos puntos estén dentro o ambos fuera, se retorna ese estado para el octante. De lo contrario, se repite el proceso, pero ahora considerando todas las caras y solo para el nodo 0. El problema de esto es que el cálculo puede fallar para casos bordes, por lo que basarse en un solo nodo cuando los resultados no coinciden no es adecuado.[\[4\]](#).

A continuación, se muestra un diagrama de los nodos que se utilizan para determinar si el octante está dentro o fuera:

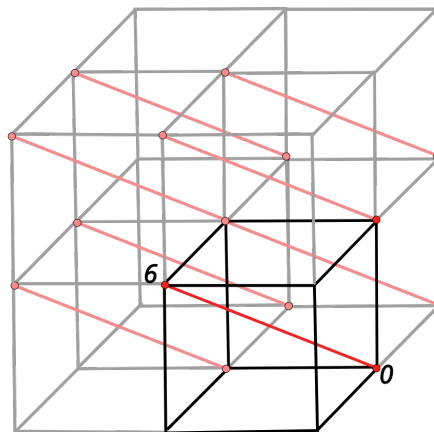


Figura 10: Octantes con nodos 0 y 6 marcados (elaboración propia).

Para solucionar esto, y poder clasificar correctamente si el octante está dentro o fuera, se adoptó un enfoque que prioriza la conservación de los octantes. Mientras uno de los nodos

indique que este está dentro, se considera que el octante lo está. Esto dado que significa que todo el octante o una parte de este está dentro de la figura.

El pseudocódigo resultante es:

Algorithm 3 IsItIn Modificado

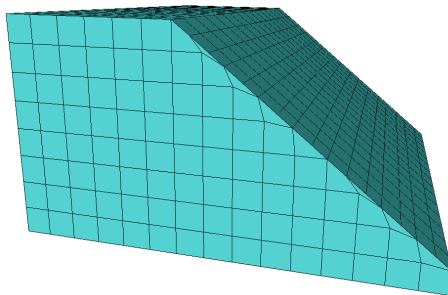
```

1: Input: mesh malla figura, faces, coords coordenadas nodo
2: Output: bool indica si está dentro o fuera.
3: point0  $\leftarrow$  coords[0]
4: point1  $\leftarrow$  coords[1]
5: first  $\leftarrow$  pointIsInMesh(point0, faces)
6: second  $\leftarrow$  pointIsInMesh(point6, faces)
7: return first || second

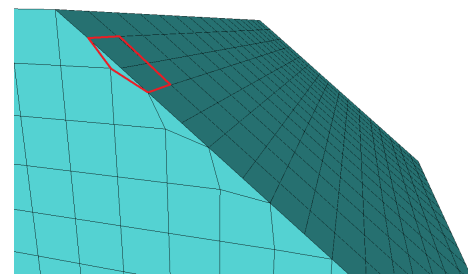
```

3.2. Método projectCloseToBoundaryNodes

El método projectCloseToBoundaryNodes se encarga de proyectar los nodos internos de un octante hacia la superficie de la figura. Esto se hace, ya que posteriormente se aplica el método shrinkToBoundary, que también proyecta los nodos, pero en este caso los externos, resultando en octantes con mucho menor volumen en comparación con el resto. El objetivo de projectCloseToBoundaryNodes es mover el octante que en un futuro quedará con poco volumen fuera de la malla y que este volumen se transfiera a un octante vecino a él. Los octantes que quedan fuera de la malla son removidos por el método removeOnSurface, que se aplica justo después del projectCloseToBoundaryNodes; éste elimina octantes que tengan todos sus nodos fuera.



(a) Figura hex wedge



(b) Figura hex wedge zoom

Figura 11: Ejemplo de hex wedge sin aplicar projectCloseToBoundaryNodes (elaboración propia).

Como se ve en la figura 11 (b), el caso que se quiere evitar es el marcado con rojo. El poliedro se presenta muy aplastado y de menor volumen en comparación al resto.

El método funciona de la siguiente manera: para cada nodo, se le calcula una proyección a través de su posición actual y las caras que intersectan. Si la distancia entre la posición actual y la proyección es menor a la máxima distancia permitida, se acepta la proyección

y se actualiza el nodo. La máxima distancia permitida se calcula según el poliedro en que se encuentre; éste toma los dos nodos más lejanos, es decir, los nodos 0 y 6, y calcula la distancia entre ellos. Dicha distancia se multiplica por un factor de 0.3 y esa magnitud será la distancia máxima permitida de desplazamiento.

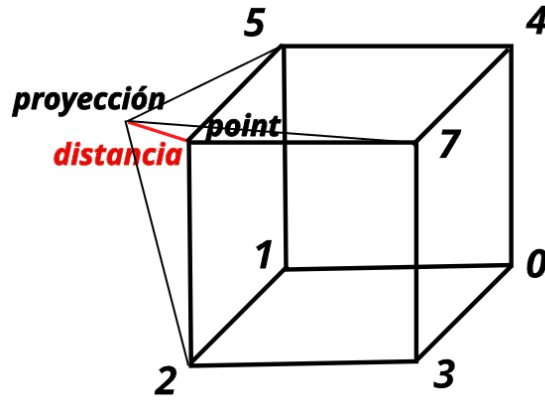


Figura 12: Hexaedro ilustrando la proyección de un nodo (elaboración propia).

$$Distancia = (point - proyeccion).Norm()$$

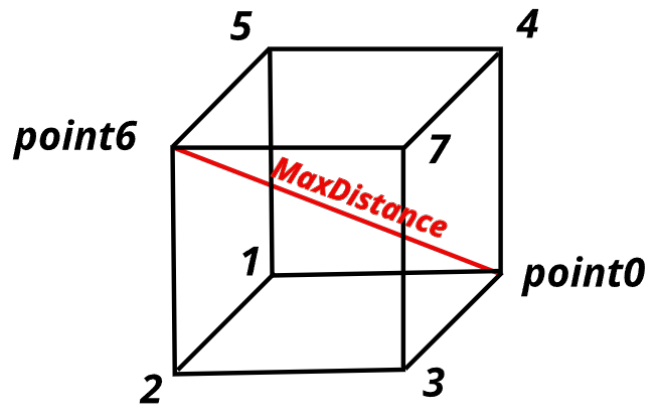
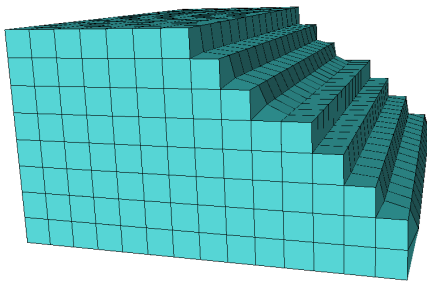


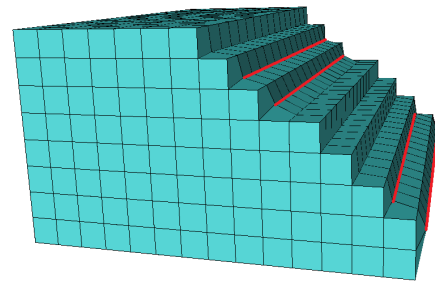
Figura 13: Hexaedro ilustrando la máxima distancia permitida (elaboración propia).

$$MaxDistance = 0,3 * (point0 - point6).Norm()$$

El problema que presenta este método es que la proyección solo se aceptaba para algunos de los nodos en el poliedro, causando que se deforme. Esto afecta en la aplicación de los patrones de superficie, dado que el cálculo del JENS para un nodo se hace a través de sus tres vecinos.



(a) Figura hex wedge



(b) Destacando proyección

Figura 14: Ejemplo de hex wedge al aplicarle `projectCloseToBoundaryNodes` sin modificar (elaboración propia).

En la figura 14 se muestra el resultado de aplicar `projectCloseToBoundaryNodes` (sin la aplicación del `shrinkToBoundary`) sobre una cuña hexagonal, donde se ve que las proyecciones solo ocurrieron en algunos nodos de los octantes, causando que se deformen.

La solución implementada consiste en que, cuando se proyecta un nodo, cada uno de los ejes de los nodos vecinos se traslada la misma distancia que se desplazó el respectivo eje del nodo proyectado, logrando así mantener la forma. Para ello, se recorren los nodos vecinos de los nodos proyectados y se les suma, por eje, la traslación aplicada al nodo proyectado, pero únicamente si se cumplen ciertas condiciones:

1. La proyección realizada al nodo interno modifica el valor de ese eje (es decir, la traslación en dicho eje es distinta de cero).
2. El nodo interno antes de proyectarse y el nodo vecino se encuentran alineados en ese eje.
3. La traslación en el eje es mayor que cualquier otra traslación calculada a partir de otro nodo interno proyectado. Esto es en caso que un nodo sea vecino de mas de un nodo proyectado.

Con esto los octantes que tienen un nodo proyectado, no se deforman.

El resultado de esto se muestra en la figura 15, donde se observa la aplicación `projectCloseToBoundaryNodes` modificado (sin la aplicación del `shrinkToBoundary`) sobre una cuña hexagonal.

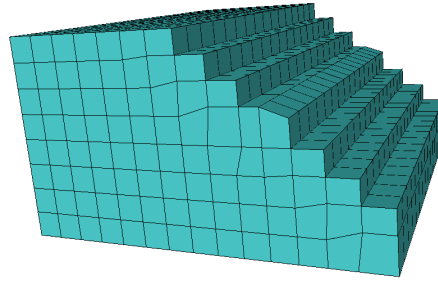


Figura 15: Ejemplo de un cuña hexagonal al aplicarle projectCloseToBoundaryNodes modificado (elaboración propia).

A continuación se muestra el pseudocódigo del método, junto con la modificación agregada:

Algorithm 4 projectCloseToBoundaryNodes Modificado

```
1: Input: input malla figura
2: Output: Sin retorno
3: for octant in octants
4:   if octant.isSurface() then
5:     continue
6:   else
7:     octant.computeMaxDistance(points)
8:   end if
9: end for
10: for octant in octants
11:   points = octant.getPoints();
12:   for point in points
13:     if point.isInside then
14:       in_nodes.push_back(point);
15:       for pointNeighbor in points
16:         if point != pointNeighbor then
17:           neighbors_nodes[point].push_back(pointNeighbor);
18:         end if
19:       end for
20:     end if
21:   end for
22: end for
```

Las modificaciones realizadas corresponden a las líneas 15-18, y se agregó toda la sección siguiente, después de la línea 22.

```

1: for point in points_in
2:   if !point.wasProjected then
3:     continue
4:   end if
5:   changePoint = point.getPoint()
6:   oldPoint = point.getPointOld()
7:   it = neighbors_nodes.find(point)
8:   if it == neighbors_nodes.end() then continue
9:   end if
10:  for neighbor : it->second)
11:    if neighbor == point then continue
12:    end if
13:    currentPoint = neighbor.getPoint()
14:    currentPointOld = neighbor.getPointOld()
15:    if oldPoint.X() != changePoint.X() && oldPoint.X() == currentPointOld.X() then
16:      if abs((point - pointOld).X) > abs((neighbor_point - neighbor_pointOld).X) then
17:        neighbor_pointOld.X ← neighbor_point.X
18:        neighbor_point.X ← neighbor_point.X + (point - pointOld).X
19:      end if
20:    end if
21:    if oldPoint.Y() != changePoint.Y() && oldPoint.Y() == currentPointOld.Y() then
22:      if abs((point - pointOld).Y) > abs((neighbor_point - neighbor_pointOld).Y) then
23:        neighbor_pointOld.Y ← neighbor_point.Y
24:        neighbor_point.Y ← neighbor_point.Y + (point - pointOld).Y
25:      end if
26:    end if
27:    if oldPoint.Z() != changePoint.Z() && oldPoint.Z() == currentPointOld.Z() then
28:      if abs((point - pointOld).Z) > abs((neighbor_point - neighbor_pointOld).Z) then
29:        neighbor_pointOld.Z ← neighbor_point.Z
30:        neighbor_point.Z ← neighbor_point.Z + (point - pointOld).Z
31:      end if
32:    end if
33:  end for
34: end for

```

Es importante señalar que los nodos internos proyectados se convierten en nodos externos. Estos están en la superficie de la malla.

3.3. Método getHexInfo

El método getHexInfo se agregó luego de aplicar el método shrinkToBoundary. Éste se encarga de obtener el volumen de cada octante y de calcular el promedio de los volúmenes de los octantes en la malla. Para determinar el volumen de un octante, éste se divide en cinco tetraedros, de la siguiente forma:

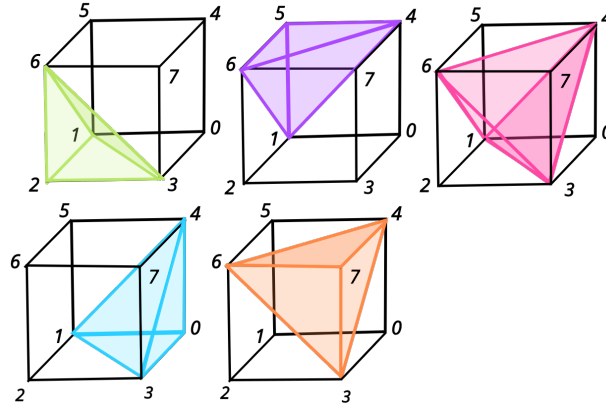


Figura 16: Tetraedros para el calculo del volumen (elaboración propia).

Donde a cada uno de ellos se le calcula el volumen mediante la siguiente fórmula:

$$Volumen\ Tetraedro = \frac{1}{6} \left| \vec{AD} \cdot (\vec{BD} \times \vec{CD}) \right|$$

Por ejemplo, para el siguiente tetraedro:

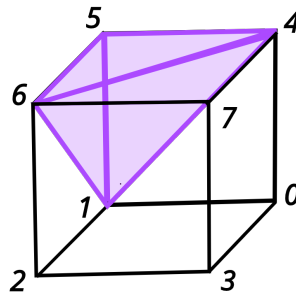


Figura 17: Tetraedro con nodos 1, 4, 5 y 6 (elaboración propia).

Donde: $\vec{AD} = point1 - point6$

$\vec{BD} = point4 - point6$

$\vec{CD} = point5 - point6$

Tras obtener el volumen de cada uno de los tetraedros, estos se suman, obteniéndose así el volumen del octante [7]. Este volumen total se agrega a la variable mediaVolumen, la cual, una vez finalizado el cálculo para todos los octantes, se divide por la cantidad de estos, obteniéndose la media de los volúmenes de la figura. Tanto el volumen como la media y el identificador del octante se almacenan para ser utilizados posteriormente en el método deleteVolume, con el fin de eliminar los octantes según corresponda.

Aquí está el pseudocódigo de getHexInfo:

Algorithm 5 getHexInfo

```

1: Input: Sin parámetros
2: Output: Sin retorno
3: for octant in octants
4:   if octant.isInside() then
5:     continue
6:   end if
7:   epts  $\leftarrow$  octant.getPoints()
8:   hex  $\leftarrow$  []
9:   for nodo in epts
10:    hex.push_back(points[nodo])
11:   end for
12:   volumen  $\leftarrow$  hexahedronVolume(hex)
13:   mediaVolumen  $\leftarrow$  volumen + mediaVolumen
14:   octelementsWithVolumen.push_back(make_tuple(octant, volumen))
15: end for
16: mediaVolumen  $\leftarrow$  mediaVolumen/octelementsWithVolumen

```

Y el pseudocódigo de hexahedronVolumen, que se utiliza en getHexInfo:

Algorithm 6 hexahedronVolumen

```

1: Input: hex vector con los nodos del hexaedro
2: Output:
3: function tetrahedronVolume(a, b, c, d)
4:   ad  $\leftarrow$  a - d
5:   bd  $\leftarrow$  b - d
6:   cd  $\leftarrow$  c - d
7:   return |dot(ad, cross(bd, cd))|  $\cdot \frac{1}{6}$ 
8: end function
9: vol  $\leftarrow$  0
10: vol  $\leftarrow$  tetrahedronVolume(hex[0], hex[1], hex[3], hex[4])
11: vol  $\leftarrow$  tetrahedronVolume(hex[1], hex[2], hex[3], hex[6])
12: vol  $\leftarrow$  tetrahedronVolume(hex[1], hex[4], hex[5], hex[6])
13: vol  $\leftarrow$  tetrahedronVolume(hex[3], hex[4], hex[6], hex[7])
14: vol  $\leftarrow$  tetrahedronVolume(hex[1], hex[3], hex[4], hex[6])
15: return vol

```

3.4. Método deleteVolume

El método deleteVolume se agregó después de getHexInfo. Su propósito es eliminar los octantes que presentan un volumen muy pequeño en comparación con la media de volúmenes de la figura. Fue necesario incorporarlo, ya que, existen casos donde el project-CloseToBoundaryNodes no logra trasladar los octantes fuera de la malla y, tras aplicarle el

shrinkToBoundary, estos quedan con un volumen mínimo respecto de los demás, llegando a verse casi completamente aplastados. Al estar en estas condiciones, no aportan valor a la figura, y su eliminación no modifica la representación de ésta. Cabe mencionar que estos octantes aparecen con la modificación del projectCloseToBoundaryNodes.

Este método recorre cada octante y elimina aquellos que tengan volumen menor a un cierto porcentaje de la media de los volúmenes. Este porcentaje se define mediante un factor que puede variar entre 0 y 1; por ejemplo, un factor de 0.1 indica que se eliminarán los octantes cuyo volumen sea menor al 10 % de la media. A través de las pruebas realizadas, se determinó que el factor que elimina los octantes mencionados es 0.05.

A modo de ejemplo, aquellos octantes marcados en rojo en la figura 18 son aquellos que tienen un volumen muy pequeño y, en consecuencia, serán eliminados.

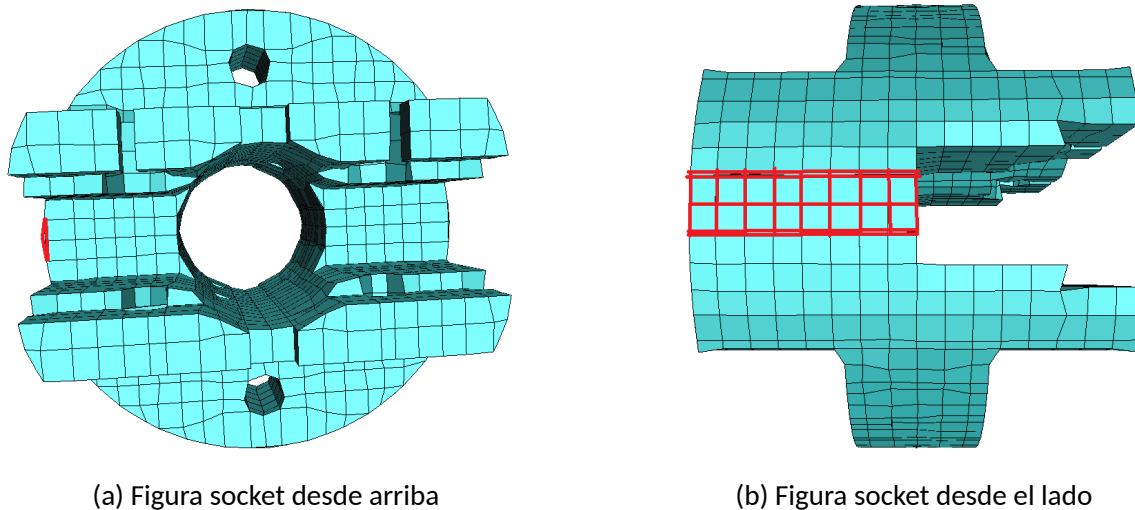
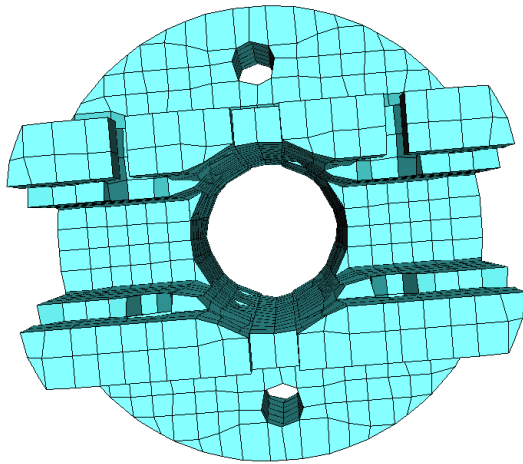
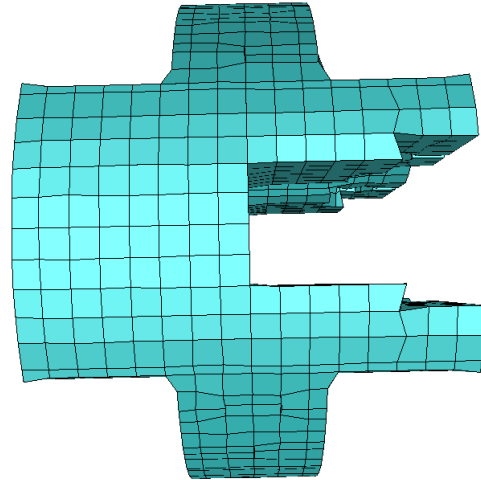


Figura 18: Ejemplo octantes que se eliminaran en figura socket (elaboración propia).

El resultado de eliminarlos se ve en la figura 19, donde los octantes que antes estaban marcados en rojo ya no se encuentran presentes:



(a) Figura socket desde arriba



(b) Figura socket desde el lado

Figura 19: Ejemplo octantes eliminados en figura socket (elaboración propia).

El pseudocódigo es:

Algorithm 7 deleteVolume

```
1: Input: Sin parámetros
2: Output: Sin retorno
3: for octant in octelementsWithVolumen
4:   volumen  $\leftarrow$  octant
5:   if volumen < mediaVolumen*factor then
6:     Delete octant
7:   end if
8: end for
```

3.5. Método allJENS_sharp_points

El método allJENS_sharp_points recorre cada poliedro dentro de cada octante y se encarga de evaluar la calidad de cada uno de sus nodos, marcando aquellos que presentan mala calidad. Estos son aquellos que no superan la tolerancia, definida por la heurística.

Este método se modificó para contar la cantidad de veces que aparece cada nodo en los distintos poliedros. Para ello, se agregó a cada nodo un contador, el que incrementa en uno cada vez que se encuentra. Este contador se utilizará para poder aplicar correctamente los nuevos patrones agregados.

También se agregó un booleano llamado priority, asociado al octante que se está revisando. Este sirve para marcar si en ese octante el cálculo de la calidad para uno o más nodos no supera la tolerancia. Su valor por defecto es de **false**, y cuando un nodo no supera la

tolerancia, cambia su valor a **true**. Esto se usará para generar un orden para la revisión de los octantes en `orderedOctantGenerator`.

El pseudocódigo es:

Algorithm 8 `allJENS_sharp_points`

```

1: Input: points vector de coordenadas, octants vector de octantes,
2:      elements vector de elementos (poliedros)
3: Output: Sin retorno
4: for i = 0; i < elements.size(); i++
5:     jens = elements[i]->getJENS()
6:     node = elements[i]->getPoints()
7:     for j = 0; j < jens.size(); j++
8:         points[node[j]].countConexion++
9:         if jens[j] < tol then
10:            points[node[j]].qualityTol ← false
11:            octants[i]->priority ← true
12:         end if
13:
14:     end for
15: end for

```

Aquí las líneas 8 y 11 fueron agregadas.

3.6. Método `orderedOctantGenerator`

El método `orderedOctantGenerator` se utiliza para generar un orden en los octantes que poseen nodos con mala calidad, para poder aplicar correctamente los patrones en `applySurfacePatterns`. El orden generado se basa en dos criterios: el atributo *priority* del octante (definido previamente en `allJENS_sharp_points`) y la cantidad de nodos internos. Los primeros en el orden son todos los octantes que tienen *priority* **true**, a la vez que están ordenados de mayor a menor por la cantidad de nodos internos, y luego están los de *priority* **false**, también ordenados de mayor a menor por la cantidad de nodos internos.

Para lograr esto, se revisa cada octante obteniendo la cantidad de nodos internos que posee y si posee algún nodo de mala calidad. Solo si este último criterio se cumple, se agrega a una lista en una tupla, la cual contiene el atributo *priority* del octante, la cantidad de nodos internos y el índice del octante. Por último, tras terminar de revisar todos los octantes, se ordena como se describió anteriormente.

A continuación se muestra el pseudocódigo:

Algorithm 9 orderedOctantGenerator

```

1: Input: mesh malla figura
2: Output: octantOrder lista de tuplas con orden de octantes
3: for octant in octants
4:   if octant.isSurface() then
5:     countQualityFalse  $\leftarrow$  0
6:     countinpts  $\leftarrow$  0
7:     for point in octant
8:       if mesh.at(point).isInside() then
9:         countinpts++
10:      end if
11:      if point.qualityTol then
12:        countQualityFalse++
13:      end if
14:    end for
15:    if countQualityFalse > 0 then
16:      octantOrder.push_back(octant.priority, countinpts, octant.index);
17:    end if
18:  end if
19: end for
20: octantOrder.sort(greater<tuple<bool, int, int> >())

```

3.7. Método applySurfacePatterns

El método applySurfacePatterns se encarga de aplicar los patrones de superficie a los octantes de la malla que están en contacto con la superficie, con el objetivo de mejorar su calidad y su representación. Esto es necesario porque, al aplicar el método ShrinkToBoundary, estos octantes pueden deformarse, quedando con nodos de mala calidad. Como se explicó anteriormente, ShrinkToBoundary proyecta los nodos externos sobre la superficie.

Los patrones de superficie evalúan los nodos internos y externos del octante y lo transforman a uno o un conjunto de poliedros que se adecúen mejor a la forma de la figura, mejorando la calidad de los nodos.

Sin embargo, el problema que se presenta aquí es que, cuando se aplican los patrones a un octante, no se considera cuáles son los nodos que presentan mala calidad en el momento de la transformación. Como resultado, se obtienen poliedros que no representan adecuadamente el contorno de la figura en la que se está trabajando. Para solucionar esto, se agregaron nuevos patrones que sí consideran los nodos de mala calidad cuando se transforma el octante. Más específicamente, se agregaron patrones para cuando se tiene solo un nodo interno.

Se agregaron para estos casos, porque independientemente de cuál sea el nodo interno o cuáles nodos presenten mala calidad, siempre se aplica un único patrón que transforma el

octante en un tetraedro.

Por ejemplo, a continuación se tiene la aplicación del patrón cuando se tiene el nodo interno 1:

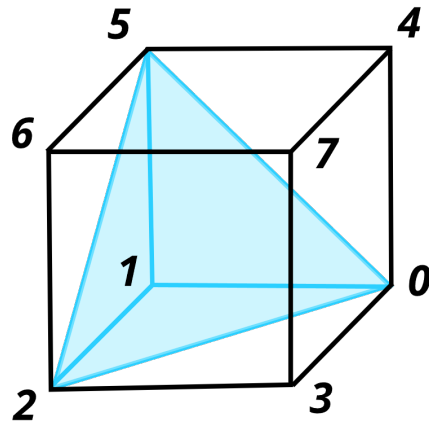


Figura 20: Resultado de aplicar patrón con un solo nodo interno (elaboración propia).

Esta figura no se adecuará para todos los casos. Por ejemplo, cuando se presenten 2 o 4 nodos con mala calidad, lo más apropiado para ambos casos sería convertir el octante a un prisma triangular. A continuación se detalla la situación:

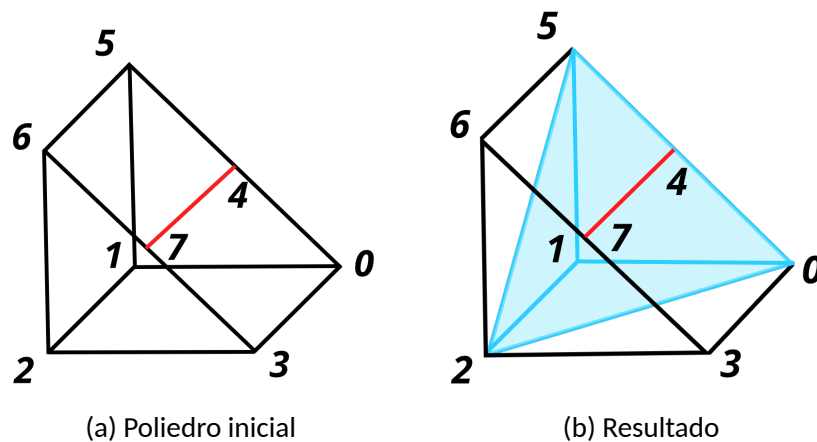


Figura 21: Aplicación del patrón para un solo nodo interno con nodos 4 y 7 de mala calidad (elaboración propia).

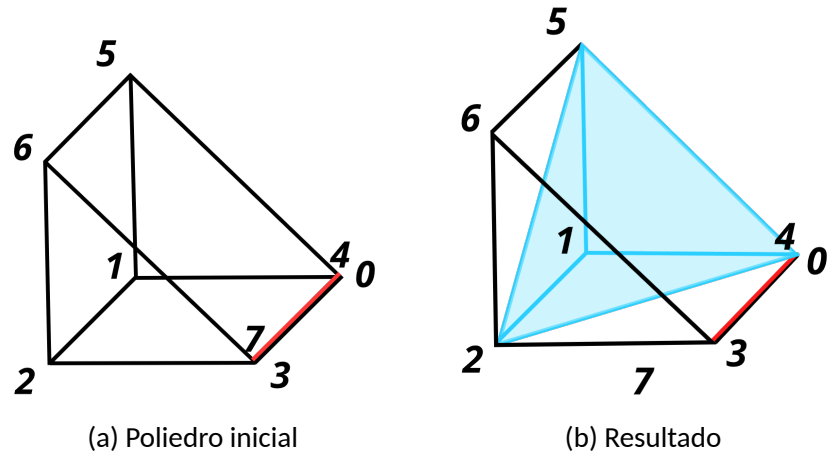


Figura 22: Aplicación del patrón para un solo nodo interno con nodos 0, 3, 4 y 7 de mala calidad (elaboración propia).

Aquí, en ambos casos, el resultado solo considera los nodos 0, 1, 2 y 5, eliminando aquellos que generan la mala calidad, pero sin obtener el mejor poliedro que mantenga la forma original, que, como se mencionó previamente, sería un prisma triangular.

Para solucionar esto, se crearon dos patrones, patternOct1 y patternOct2, ambos con el fin de transformar el octante en un prisma triangular. El patternOct1 se aplica cuando hay 2 o 3 nodos de mala calidad, mientras que el patternOct2 se aplica cuando hay entre 4 y 6 nodos de mala calidad.

El patternOct1 se utiliza en la situación de la figura 21, donde los dos nodos centrales son los que están mal posicionados. Usualmente, estos son los nodos con mala calidad que se usarán para aplicar correctamente el patrón; pueden suceder otros casos que se explicarán más adelante. Para este caso, se obtendría como resultado:

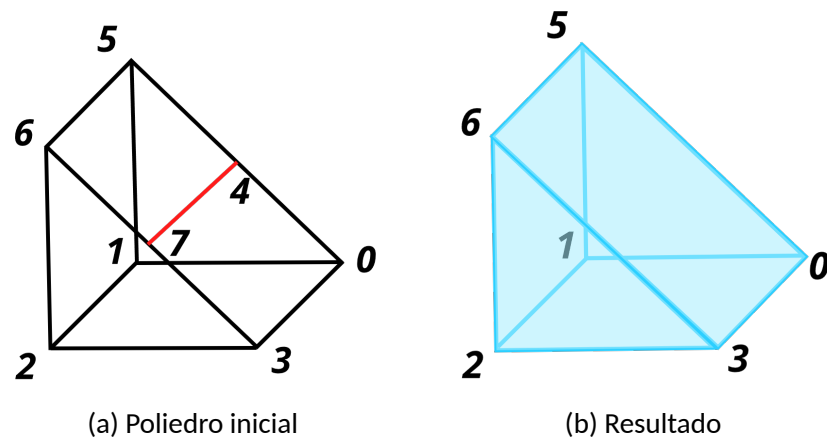


Figura 23: Aplicacion de PatternOct1 (elaboración propia).

Manteniéndose los nodos 0, 1, 2, 3, 5 y 6, logrando así conservar la mejor forma para este poliedro.

El patternOct1 utiliza el vector position; este contiene los nodos de mala calidad. Estos nodos son los que se utilizan para generar el correcto prisma triangular. Es importante señalar que, cuando el tamaño del vector es igual a 3, es necesario reordenarlo para colocar los nodos en los que se debe enfocar en las dos primeras posiciones. La lógica para reordenar se divide en tres casos:

- Si un nodo interno está en el vector, éste se mueve para el final y los otros dos al inicio. Estos dos corresponden a los que se debe enfocar para eliminarlos en la posición central.
- Si se tiene un nodo con countConexion igual a 1, significa que es exclusivo del octante y es el que está en la posición central. Con éste se evalúa la distancia a los otros dos y, el que esté más lejano, es el que también se encuentra en la posición central. Estos dos se mueven a las dos primeras posiciones.
- Si los nodos centrales no presentan mala calidad, en tal caso se verifica si alguno de los pares de nodos posibles corresponde al par formado por el primero con el tercero o el segundo con el tercero. Estos casos requieren que el vector se reordene, moviendo la coincidencia a las dos primeras posiciones. Algunos ejemplos de pares posibles son (1, 2), (0, 3), entre otros.

Ahora, utilizando el vector ordenado y la posición del nodo interno, se logra tomar los nodos adecuados para formar el prisma triangular. Aquí también se tendrán tres casos: el que ya vimos, en la figura 23, donde se toman los nodos 0, 1, 2, 3, 5 y 6. Éste se consigue cuando se tienen los nodos con mala calidad 4 y 7 y el nodo interno 1 o el 2.

Los otros dos casos corresponden a cuando los nodos centrales no son los que indican mala calidad, por ejemplo, que los nodos 4 y 7 presenten mala calidad, pero los nodos centrales sean el 5 y 6 o el 0 y 3, como se ve a continuación:

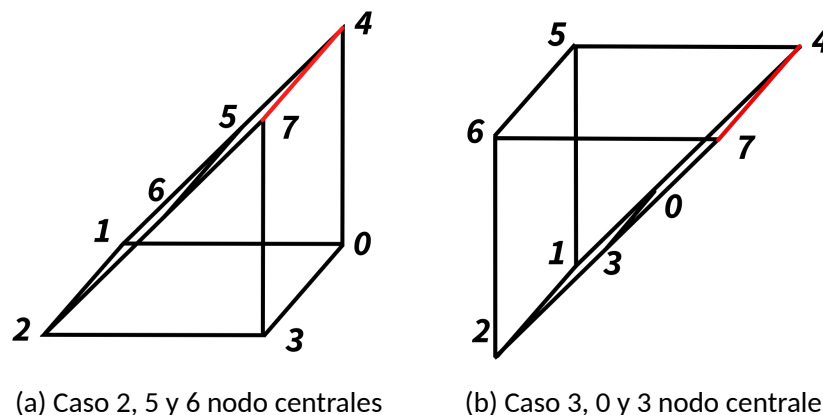


Figura 24: Casos donde se debe aplicar el patternOct1 (elaboración propia)

En ambos casos varían los nodos internos. Para el caso de la figura 24 (a), corresponden al nodo 0 o al 3, mientras que para el caso de la figura 24 (b) son el 5 o el 6. Por tanto, cuando el nodo interno es el 0 o el 3, los nodos que se eliminan son el 5 y el 6, quedando el 0, 1, 2, 3, 4 y el 7, y en caso de que el nodo interno fuera el 5 o el 6, se eliminan el 0 y el 3, quedando el 1, 2, 4, 5, 6 y el 7. De esta manera se mantiene la forma y las caras que sí corresponden. El resultado de aplicar patternOct1 a estos dos casos es el siguiente:

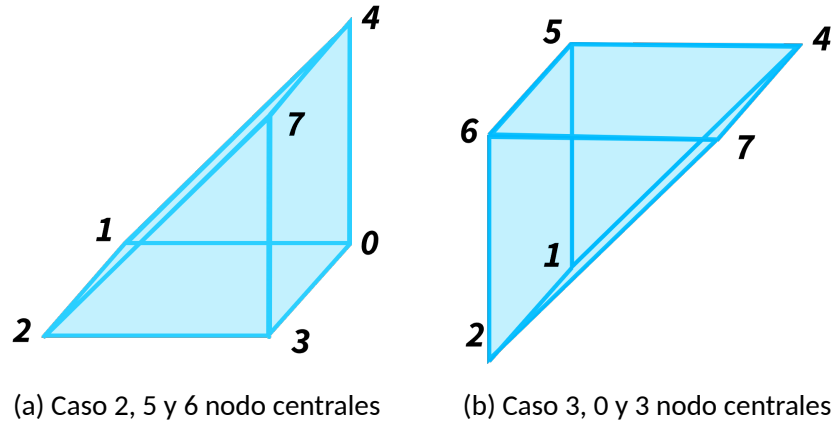


Figura 25: Aplicación del patternOct1 (elaboración propia)

El pseudocódigo es:

Algorithm 10 patternOct1

```

1: Input: points vector de coordenadas, original vector de nodos del octante, counter cantidad de
   nodos con mala calidad, position vector con los índices de los nodos con mala calidad, in vector
   con nodo interno
2: Output: ele1 vector de nodos de la figura resultante
3:
4: if counter == 3 then
5:     reorder(position)
6: end if
7:
8: if position[0] == 4 && position[1] == 7 then
9:     if in[0] == 1 || in[0] == 2 then
10:         ele1.push_back(original[1]);
11:         ele1.push_back(original[0]);
12:         ele1.push_back(original[5]);
13:         ele1.push_back(original[2]);
14:         ele1.push_back(original[3]);
15:         ele1.push_back(original[6]);
16:     else
17:         if in[0] == 0 || in[0] == 3 then
18:             ele1.push_back(original[0]);
19:             ele1.push_back(original[4]);
20:             ele1.push_back(original[1]);
21:             ele1.push_back(original[3]);
22:             ele1.push_back(original[7]);
23:             ele1.push_back(original[2]);
24:         else
25:             ele1.push_back(original[5]);
26:             ele1.push_back(original[1]);
27:             ele1.push_back(original[4]);
28:             ele1.push_back(original[6]);
29:             ele1.push_back(original[2]);
30:             ele1.push_back(original[7]);
31:         end if
32:     end if
33:     return ele1;
34: end if
35: ...

```

Por otro lado, el patternOct2 es para cuando se tenga la situación de la figura 22, donde existen dos pares de nodos muy próximos entre sí o incluso apilados. Aquí también puede suceder que uno o dos nodos adicionales indiquen mala calidad por su cercanía.

Con este patrón para el caso de la figura 22 con los nodos 0, 3, 4 y 7 de mala calidad, se tendría lo siguiente:

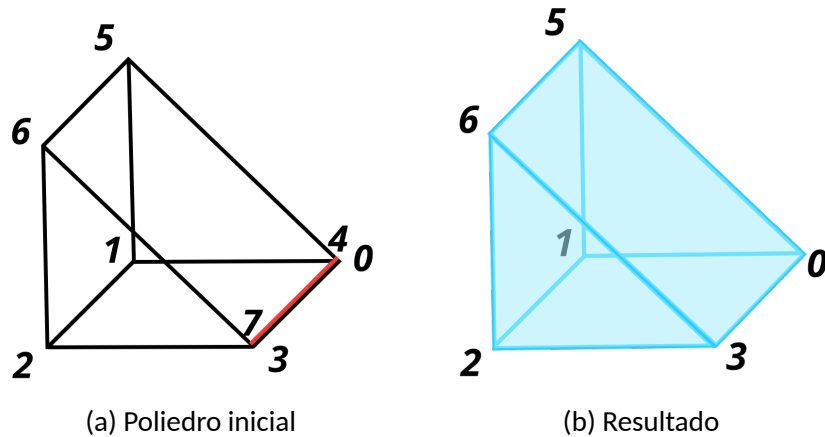


Figura 26: Aplicacion de PatternOct2 (elaboración propia)

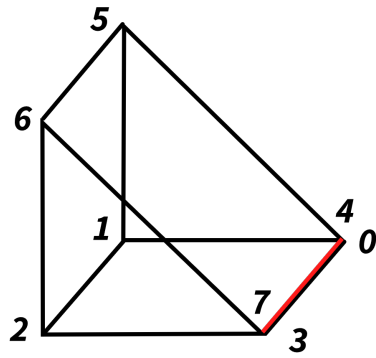
Resultando en un prisma triangular formado por los nodos 0, 1, 2, 3, 5 y 6, conservando la forma según la distribución de los nodos.

El patternOct2 primero se encarga de reordenar en caso de tener 5 o 6 nodos con mala calidad. Estos reordenamientos se explican a continuación:

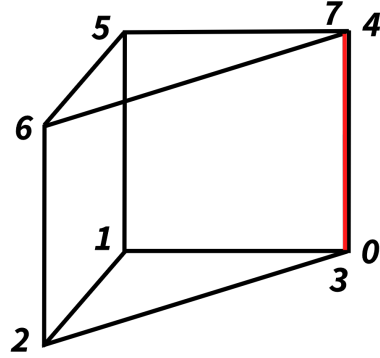
- En el caso de que sean 5 nodos, se compara una lista de los casos posibles y se busca la que coincide con los nodos presentes en el vector. Estos se mueven a las primeras posiciones de éste. Por ejemplo, un caso posible es 1, 2, 5 y 6.
- En el caso de que sean 6 nodos, se busca cuáles son los nodos que no se presentan; con esto se sabe con cuáles nodos se comparará y se guardan los que están más cercanos entre sí. Por ejemplo, si el 2 y el 6 no se encuentran en el vector, significa que presentan mala calidad el 0, 1, 3, 4, 5 y 7. Luego se evalúa cuál es más cercano, el 0 con el 5 o el 0 con el 7. Si es el primer caso, se mueven a las primeras posiciones el 0, 1, 4 y 5, mientras que si es el segundo caso, se mueve a las primeras posiciones el 0, 3, 4 y 7. Obteniéndose el conjunto de nodos más cercanos entre sí.

A partir de esto, se utiliza el vector ordenado junto con el nodo con countConexion igual a 1, para considerar los nodos adecuados para formar el prisma triangular.

Aquí se tienen cuatro casos al momento de generar el prisma triangular. Dos de estos se presentan porque se tienen dos orientaciones para los nodos con mala calidad. Por ejemplo, si es que los nodos 0, 3, 4 y 7 presentan mala calidad, puede ser porque 0 está muy cerca de 4 y 3 muy cerca de 7 o que 0 está muy cerca de 3 y 4 muy cerca de 7, como se ve a continuación:



(a) Caso 1, 0 cerca de 4 y 3 cerca de 7

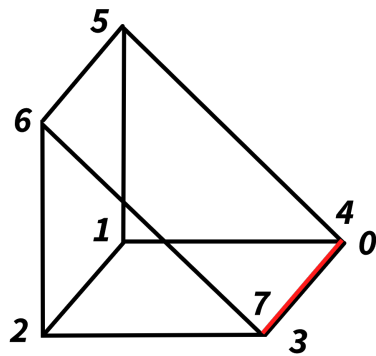


(b) Caso 2, 0 cerca de 3 y 4 cerca de 7

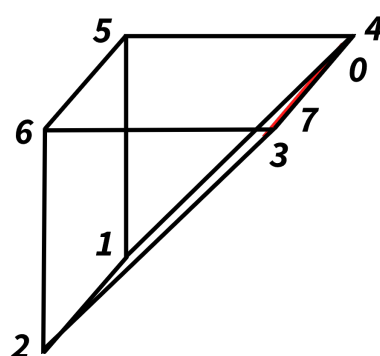
Figura 27: Casos donde se debe aplicar el patternOct2 (elaboración propia)

Para saber a cuál corresponde, se calcula la distancia desde uno de los nodos a otros dos. Con esto se sabrá cuáles son los que están más cercanos y así determinar la orientación. En el caso ya mencionado, se ve si es que la distancia del nodo 3 al 7 es menor que la distancia del nodo 0 al 3.

Los otros dos casos surgen al tener que distinguir, por ejemplo en el caso 1, si fueron trasladados hacia arriba los nodos 0 y 3, o hacia abajo los nodos 4 y 7. A continuación se muestran estas dos situaciones:



(a) Caso 3, nodo 0 y 3 compartido

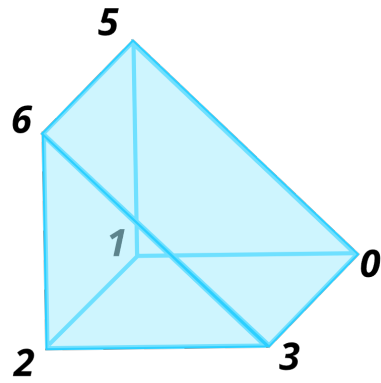


(b) Caso 4, nodo 4 y 7 compartido

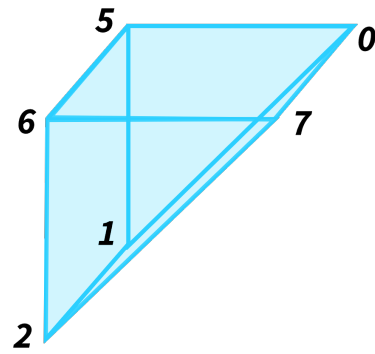
Figura 28: Casos donde se debe aplicar el patternOct2 (elaboración propia)

Para poder distinguirlo, hay que considerar que el par que se trasladó posee un nodo que no es compartido por otro poliedro, es decir, su countConexion es igual a 1.

En el caso 3, el par de nodos 4 y 7 contiene el nodo que no es compartido, por lo que deben eliminarse, manteniendo los nodos 0, 1, 2, 3, 5 y 6. En cambio, para el caso 4, el par de nodos 0 y 3 presenta el nodo que no es compartido, por lo que ahora son los que deben eliminarse, manteniendo los nodos 1, 2, 4, 5, 6 y 7. Las figuras resultantes son:



(a) Caso 3, nodo 0 y 3 compartido



(b) Caso 4, nodo 4 y 7 compartido

Figura 29: Aplicación del patternOct2 (elaboración propia)

El pseudocódigo es:

Algorithm 11 patternOct2

```

1: Input: points vector de coordenadas, original vector de nodos del octante, counter cantidad de
   nodos con mala calidad, position vector con los indices de los nodos con mala calidad, in vector
   con nodo interno
2: Output: ele1 vector de nodos de la figura resultante
3: if position[0] == 0 && position[1] == 3 && position[2] == 4 && position[3] == 7 then
4:   if (points[original[3]] - points[original[7]]).Norm() <
5:     (points[original[0]] - points[original[3]]).Norm() then
6:       if (points[original[4]].countConexion == 1 || points[original[7]].countConexion == 1 then
7:         ele1.push_back(original[1]);
8:         ele1.push_back(original[0]);
9:         ele1.push_back(original[5]);
10:        ele1.push_back(original[2]);
11:        ele1.push_back(original[3]);
12:        ele1.push_back(original[6]);
13:      else
14:        ele1.push_back(original[5]);
15:        ele1.push_back(original[1]);
16:        ele1.push_back(original[4]);
17:        ele1.push_back(original[6]);
18:        ele1.push_back(original[2]);
19:        ele1.push_back(original[7]);
20:      end if
21:    else
22:      if (points[original[3]].countConexion == 1 || points[original[7]].countConexion == 1 then
23:        ele1.push_back(original[1]);
24:        ele1.push_back(original[2]);
25:        ele1.push_back(original[0]);
26:        ele1.push_back(original[5]);
27:        ele1.push_back(original[6]);
28:        ele1.push_back(original[4]);
29:      else
30:        ele1.push_back(original[6]);
31:        ele1.push_back(original[7]);
32:        ele1.push_back(original[5]);
33:        ele1.push_back(original[2]);
34:        ele1.push_back(original[3]);
35:        ele1.push_back(original[1]);
36:      end if
37:    end if
38:  end if
39:  ...

```

Cabe destacar que estos patrones están pensados para que se apliquen en situaciones cuando exista un countConexion igual a 1.

Otro problema que presenta applySurfacePatterns es que, al agregar los patrones explicados, existen casos donde ciertos patrones no deberían aplicarse. Esto se debe a que existen dos situaciones para que se apliquen los patrones a los octantes:

- Al calcular la calidad del octante al menos uno de los nodos presenta mala calidad.
- El octante comparte nodos con otro octante, en el cual los nodos presentan mala calidad.

Entonces, pueden ocurrir casos donde, al cambiar un octante en la primera situación, no necesariamente se deban cambiar los octantes vecinos que se encuentren en la segunda situación. Esto se puede ver en la figura 30:

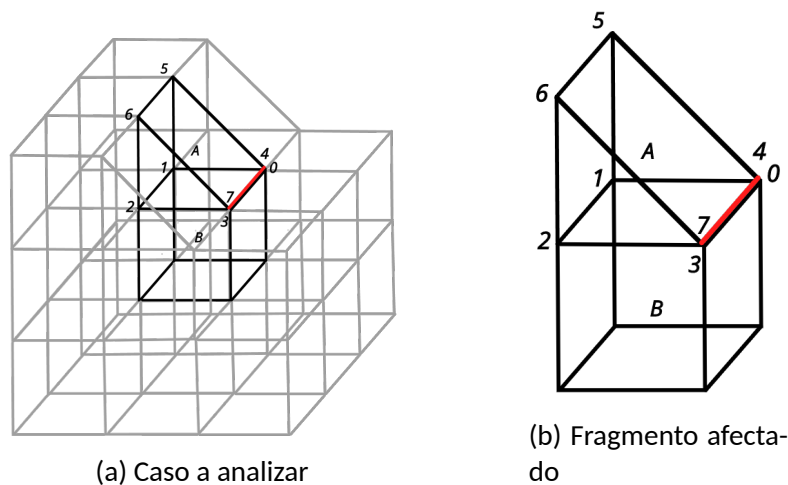


Figura 30: Figura con octantes A y B afectados

En este caso, el octante A pertenece a la primera situación donde los nodos 0, 3, 4 y 7 presentan mala calidad. Por su parte, el octante B pertenece a la segunda situación, ya que comparte nodos de mala calidad con el octante A; estos corresponden a los nodos 0 y 3. Lo más adecuado en este caso es aplicar un patrón al octante A para que mantenga los nodos 0, 1, 2, 3, 5 y 6, mientras que al octante B no se le aplique ningún patrón. No obstante, el octante B al tener los nodos 0 y 3, el patrón igual se aplica.

El resultado de aplicar los patrones es:

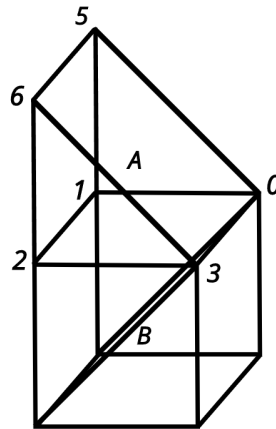


Figura 31: Patrón aplicado sobre A y B (elaboración propia).

Para esto, se cambió el orden en que se revisan los octantes mediante el método `ordereOctantGenerator`, y se agregaron marcas en los nodos para indicar si es que debe aplicarse el patrón. Primero se aplican los patrones a los octantes que pertenezcan a la primera situación y después los de la segunda situación. Cada vez que se aplica un patrón, se marcan los nodos afectados. Los patrones correspondientes a la primera situación siempre se aplican; en cambio, los de la segunda situación solo se aplican si es que contienen los nodos marcados que verifican que se requiere aplicar patrón.

En el caso del ejemplo, los nodos marcados del octante A son el 4 y el 7; por tanto, no se cumplen las condiciones para aplicar el patrón al octante B, conservando su forma original. El resultado será el siguiente:

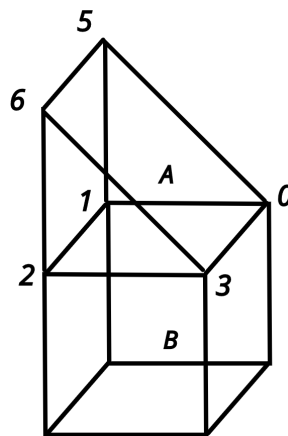


Figura 32: Patrón aplicado sobre A, pero no sobre B (elaboración propia).

Los marcadores se agregaron a todos los patrones. En el anexo, se puede ver en las Figuras 64a a 64g con rojo los nodos que se marcan y en color las figuras resultantes al aplicar el patrón.

De forma general, funciona de la siguiente manera:

Algorithm 12 SurfaceTemplate

```

1: Input: hex (revisar qué es exactamente)
2: Output: Sin retorno
3: if !priority then
4:   for (i = 0; i < octantNodes.size(); ++i)
5:     if octantNodes[i].requieres == true then
6:       requiresPattern ← true
7:       break
8:     end if
9:   end for
10: end if
11: if priority || requiresPattern then
12:   if requiresPattern then
13:     if !(octantNodes[1].requires && octantNodes[2].requires) then
14:       octants.push_back(octantNodes)
15:       return true
16:     end if
17:   end if
18:   ApplyPattern ▷ //Requiere nodos 1 y 2 marcados si es que no presenta el la mala calidad
19: end if

```

Cabe mencionar que, para los patrones con un solo nodo interno, también se estableció un orden específico para determinar cuál patrón se debe aplicar. Independientemente de si corresponde a la primera o la segunda situación, se revisan los nodos marcados y, si corresponde, se aplica el patrón que transforma el octante en un tetraedro. Esto con el fin de mantener la forma cuando se presenten dominios curvos. Si éste no corresponde, se aplican los nuevos patrones; y si es que ninguno encaja, se vuelve a revisar el patrón del tetraedro. Esto requiere que primero se apliquen los patrones a los demás octantes que posean más nodos internos, para que marquen los nodos correspondientes y se pueda verificar correctamente si es necesario aplicar el patrón. Este orden se estableció también con el método `orderedOctantGenerator`.

A continuación se muestra los nodos que se deben marcar para que sea correcto aplicar el patrón que transforma el octante a tetraedro:

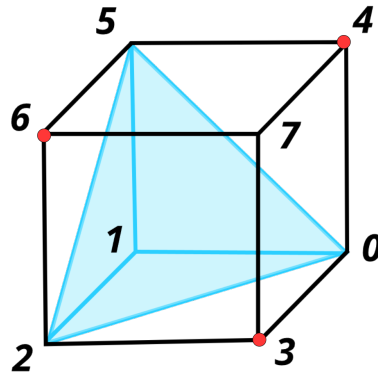


Figura 33: Tetraedro con los nodos marcados (elaboración propia).

Para que se aplique el patrón, deben estar marcados uno de los siguientes pares de nodos: el 3 y el 6, el 3 y el 4, o el 4 y el 6.

También en los patrones que se aplican cuando hay dos nodos internos, se restringió el patrón que transforma el octante en un prisma triangular. Éste solo consideraba cuáles eran los nodos internos para aplicarse, con lo que no se mantenía la forma que correspondía en ciertas situaciones. Por esto se limitó su aplicación: cuando se tienen dos nodos con mala calidad, el octante pertenece a la primera situación y no posee ningún nodo marcado. Con estas condiciones solo se aplicará si es que los nodos con mala calidad corresponden a los que están en paralelo al nodo interno o cuando los nodos de mala calidad que se presenten estén muy cercanos.

Estas condiciones son para poder mantener una correcta transición con los octantes vecinos.

A continuación se ejemplifica cuándo se aplicaría:

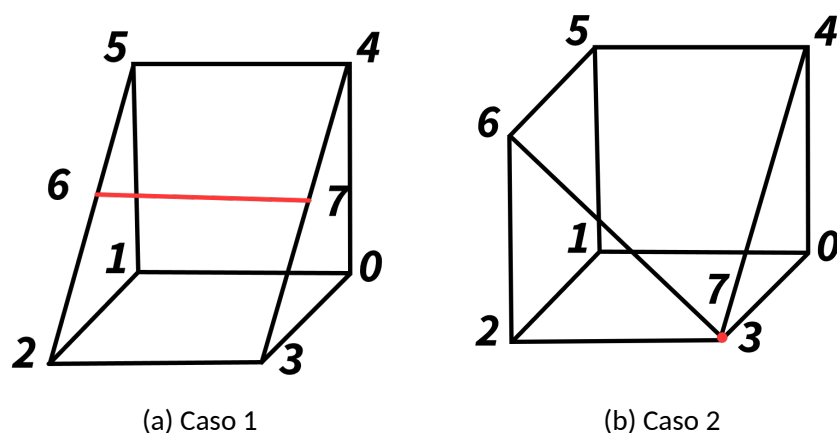


Figura 34: Octante con nodos internos 0 y 1 (elaboración propia).

En el caso 1, los nodos con mala calidad son el 6 y el 7, los cuales están alineados en paralelo respecto a los nodos internos 0 y 1. En el caso 2, los nodos con mala calidad son el 3 y el 7,

que están en la misma posición o muy cercanos entre sí. En ambos casos el resultado es un prisma triangular, como se ve a continuación:

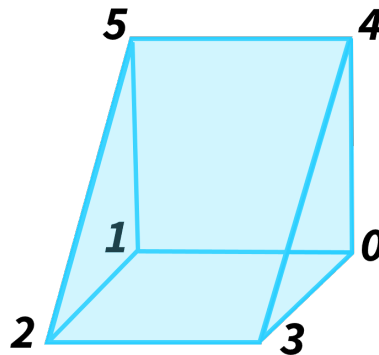


Figura 35: Resultado de aplicación de patrón A cuando hay dos nodos internos (elaboración propia).

Para determinar si corresponde al primer caso, se evaluó cuando las coordenadas son diferentes entre sí. En este caso debe haber una coordenada de diferencia entre los nodos que están mal posicionados y los nodos internos. Por ejemplo, en la figura 34 (a), los nodos 1 y 6, así como los nodos 0 y 7 deben diferir en una única coordenada (x , y o z); si ocurre esta situación corresponde al caso 1 y se aplica el patrón.

Para el caso 2, se evaluaron las coordenadas de los nodos con mala calidad. Se verificó si la diferencia de sus coordenadas (x , y , z), era menor o igual a una tolerancia de 1. Si en las tres coordenadas se cumplía esto correspondía al caso 2, y se aplicaba el patrón.

A continuación se tiene el pseudocódigo:

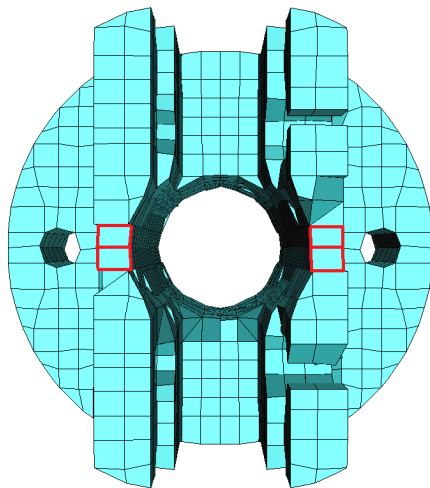
Algorithm 13 SurfTemplate2 modificado

```
1: Input: points vector de coordenadas, original vector de nodos del octante, priority valor que
   indica si es prioritario in vector con nodos internos
2: Output: ele1 vector de nodos de la figura resultante
3: counter  $\leftarrow$  0
4: for i = 0; i < original.size(); ++i
5:   if points[original[i]].qualityTol == false then counter  $\leftarrow$  counter + 1; position.push_back(i);
6:   end if
7: end for
8:
9: for i = 0; i < original.size(); ++i
10:   if points[original[i]].requires == true then requiresPattern  $\leftarrow$  true; break;
11:   end if
12: end for
13:
14: applyAcoordingPosition = false;
15: applyAcoordingCloseness = false;
16:
17: if priority && !requiresPattern && counter == 2 then
18:   for i = 0; i < original.size(); ++i
19:     if points[all[i]].requires == true then
20:       requiresPattern  $\leftarrow$  true;
21:       break;
22:     end if
23:   end for
24:   differences1  $\leftarrow$  differences(points[original[in[0]]], points[original[position[0]]]);
25:   differences2  $\leftarrow$  differences(points[original[in[1]]], points[original[position[1]]]);
26:   differences3  $\leftarrow$  differences(points[original[in[0]]], points[original[position[1]]]);
27:   differences4  $\leftarrow$  differences(points[original[in[1]]], points[original[position[0]]]);
28:
29:   applyAcoordingPosition  $\leftarrow$  ((differences1 == differences2 && quantity_one(differences1) == 1)
30:     || (differences3 == differences4 && quantity_one(differences3) == 1));
31:
32:   if points[position[1]].X() - 1 <= points[all[position[0]].X() <= points[position[1]].X() + 1 &&
33:     points[position[1]].Y() - 1 <= points[all[position[0]].Y() <= points[position[1]].Y() + 1 &&
34:     points[position[1]].Z() - 1 <= points[all[position[0]].Z() <= points[position[1]].Z() + 1 then
35:     applyAcoordingCloseness  $\leftarrow$  true;
36:   end if
37: end if
38:
39: if (priority && counter != 2) || requiresPattern || (applyAcoordingCloseness && counter == 2) ||
   (applyAcoordingPosition && counter == 2) then
40:   Aplicacion de patrones
41:   ...
42: end if
```

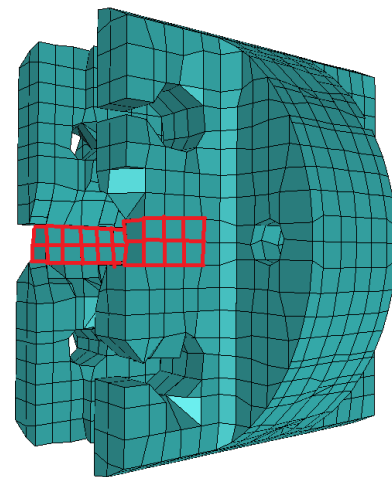
El fragmento completo corresponde a código nuevo. El código original debe ubicarse dentro del bloque if de la línea 39.

Por último, se aplicó una modificación respecto a cuando todos los nodos del octante se consideraban externos. Los patrones están diseñados para que al menos tenga un nodo interno, si no lo posee, se asume que está fuera de la figura y lo descarta. Esto no es lo correcto, ya que a pesar de que sus nodos son considerados externos, están sobre la malla, y el volumen que poseen, sigue estando dentro de la figura. Por tanto, no corresponde eliminarlos.

A continuación se muestra un ejemplo de los octantes que tienen todos sus nodos considerados externos, en la figura Socket:



(a) Caso a analizar mirado desde arriba



(b) Caso a analizar mirado desde el lado

Figura 36: Octantes descartados (elaboración propia).

Estos octantes marcados se eliminan al considerarse que todos sus nodos están fuera, quedando:

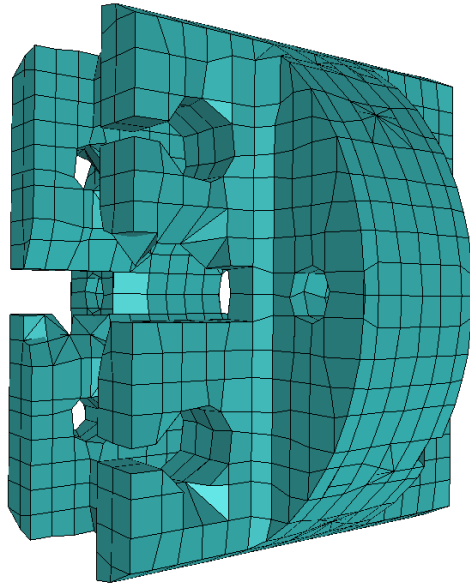


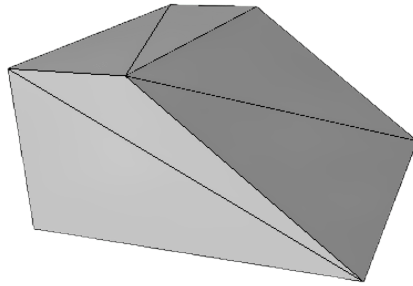
Figura 37: Figura con octantes descartados (elaboración propia).

Para solucionar esto, en vez de descartar estos octantes, se mantienen, dado que los octantes que no pertenecen a la figura ya han sido removidos en uno de los métodos anteriores; este corresponde al método `removeOnSurface`.

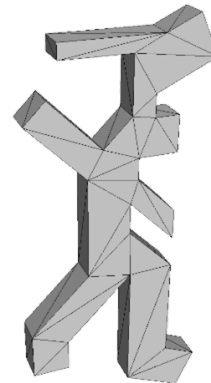
CAPÍTULO 4

VALIDACIÓN DE LA SOLUCIÓN

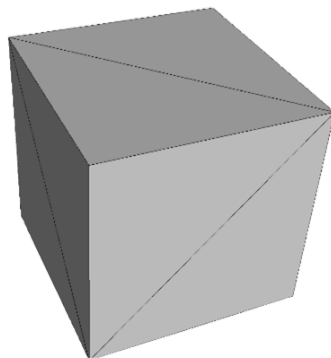
Para validar la solución planteada se utilizaron distintas figuras que presentan tanto características finas como dominios curvos. A continuación, se muestran estas figuras:



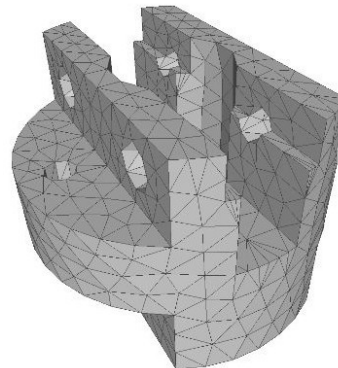
(a) Figura Hex Wedge (proporcionada por el profesor guía).



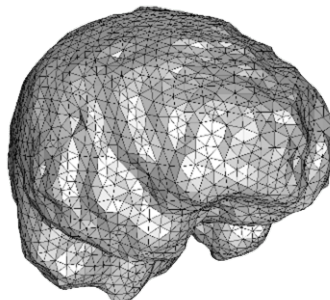
(b) Figura Fiducials [6].



(c) Figura Hex [6].



(d) Figura Socket [6].



(e) Figura Cortex [6].

Figura 38: Figuras utilizadas para probar la solución.

Cabe mencionar que el desarrollo se basó principalmente en dos de éstas, las figuras Hex Wedge y Fiducials. A continuación, se detallan los distintos valores para los conjuntos de prueba que se realizaron:

Tabla 1: Valores para el conjunto de pruebas.

Fuente: Elaboración Propia.

Refinamiento	Tolerancia
3	0.1
4	0.2
5	0.5
	0.8

Se evaluarán todas las combinaciones de estos valores y se analizarán los siguientes resultados: el promedio de la calidad, el tiempo demorado y una imagen del resultado. Las pruebas se harán con el algoritmo original y el modificado para comparar mejoras y deficiencias. Los resultados se presentarán en tablas donde el primer valor indicará la calidad promedio obtenida y el segundo valor indicará el tiempo que se requirió en generar la figura.

Las pruebas se realizaron en un computador con las siguientes características:

- Sistema operativo: Ubuntu 24.04.2 LTS
- Procesador: 12th Gen Intel(R) Core(TM) i5-12400F
- Memoria RAM: 8 GiB de RAM

Cabe mencionar que, en los casos donde la tolerancia no afecta significativamente los valores, el refinamiento con tolerancia 0.5 mostró resultados representativos, por lo que se usará esta para mostrar los resultados.

4.1. Hex Wedge

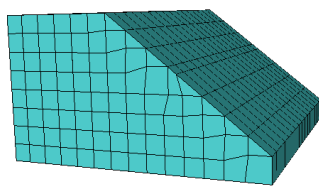
Para esta figura, los valores de tiempo y calidad promedio de los poliedros resultantes no se ven afectados significativamente por las distintas tolerancias, por lo que solo se analizará el refinamiento. A continuación, se presentan los valores obtenidos:

Tabla 2: Comparación de calidad y tiempo con algoritmo modificado (Hex Wedge).

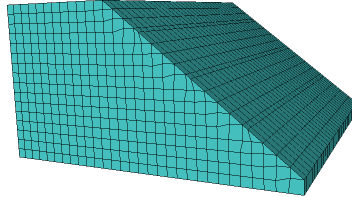
Fuente: Elaboración propia.

Ref.	Tol. 0.5	
	Cal.	Tpo [s]
3	0.938	0.079
4	0.938	0.397
5	0.938	1.775

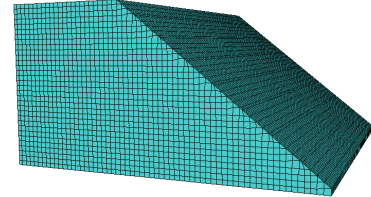
Visualmente, los resultados son los siguientes:



(a) Refinamiento 3



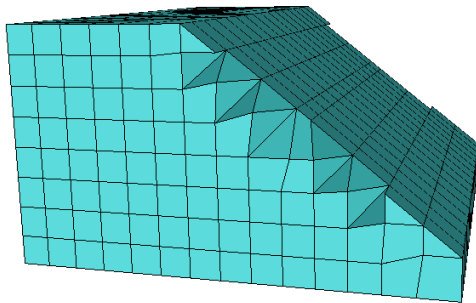
(b) Refinamiento 4



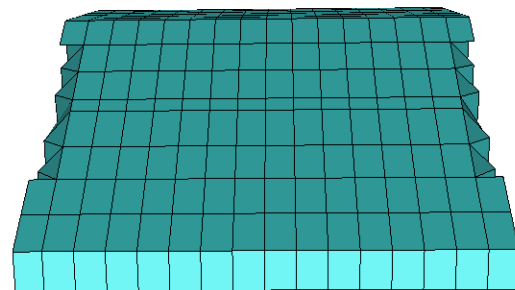
(c) Refinamiento 5

Figura 39: Resultado Hex Wedge con algoritmo modificado (elaboración propia).

El único caso a destacar es cuando la tolerancia es 0.8, ya que no representa correctamente el poliedro. Esto se debe a que los nuevos patrones requieren los nodos que presentan mala calidad para funcionar correctamente, y al establecer una tolerancia muy elevada, se consideran nodos como malos que sí están bien posicionados, provocando que el patrón no se aplique correctamente.



(a) Resultado Hex Wedge costado



(b) Resultado Hex Wedge frente

Figura 40: Resultado Hex Wedge con algoritmo modificado. Refinamiento 3, tolerancia 0.8 (elaboración propia).

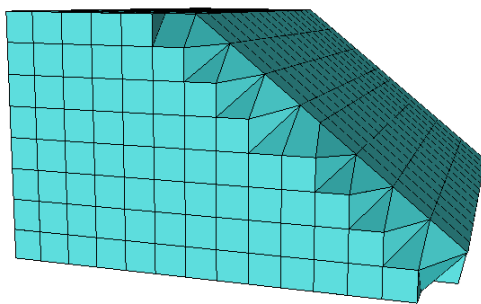
Sucede el mismo comportamiento para el algoritmo original, ya que la tolerancia no afecta en mayor grado su tiempo y calidad promedio. Sin embargo, en este caso, todos los resultados no reflejan la geometría de la figura. Los resultados según el nivel de refinamiento son:

Tabla 3: Comparación de calidad y tiempo con algoritmo original (Hex Wedge).

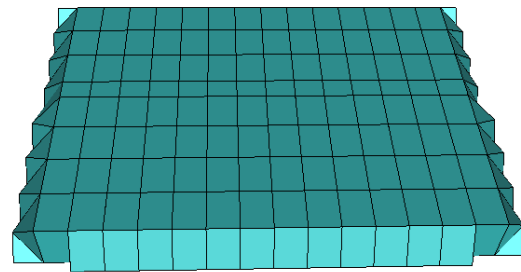
Fuente: Elaboración propia.

Ref.	Tol. 0.5	
	Cal.	Tpo. [s]
3	0.959	0.072
4	0.959	0.379
5	0.957	1.660

Mientras que los resultados visuales:



(a) Resultado Hex Wedge costado



(b) Resultado Hex Wedge frente

Figura 41: Resultado Hex Wedge con algoritmo original. Refinamiento 3, tolerancia 0.8 (elaboración propia).

Para las demás tolerancias y refinamientos, el resultado es similar a este, donde los poliedros mal representados son los de los costados, y en ningún caso muestra correctamente ese sector.

Como se observa, la representación de características finas mejora, logrando una correcta representación, con una pequeña diferencia de tiempo y de calidad. Estos valores son peores en ambos casos para el algoritmo modificado. Cabe mencionar que la única limitación encontrada es una tolerancia elevada.

4.2. Fiducials

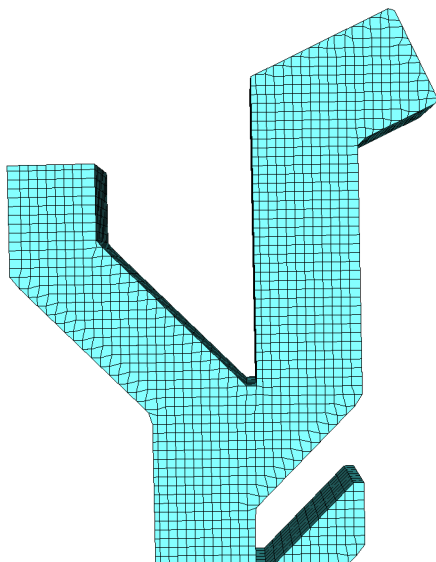
Para esta figura, los valores de tiempo y calidad promedio de los poliedros resultantes tampoco se ven afectados significativamente por las distintas tolerancias, por lo que solo se analizará el refinamiento. A continuación, se presentan los valores obtenidos:

Tabla 4: Comparación de calidad y tiempo con algoritmo modificado (Fiducials).

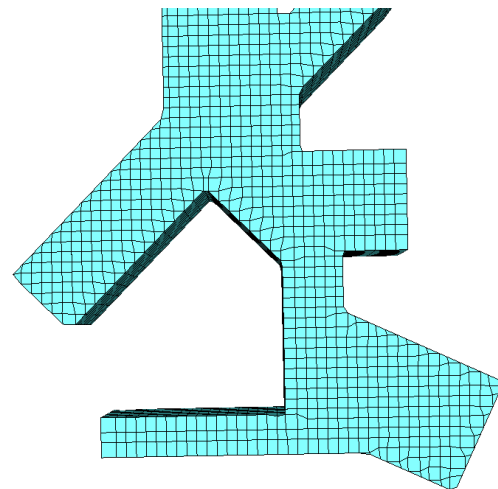
Fuente: Elaboración propia.

Ref.	Tol. 0.5	
	Cal.	Tpo. [s]
3	0.931	0.951
4	0.942	4.406
5	0.945	19.822

Visualmente, los resultados son los siguientes:



(a) Parte superior



(b) Parte inferior

Figura 42: Resultado Fiducials con algoritmo modificado. Refinamiento 3 (elaboración propia).

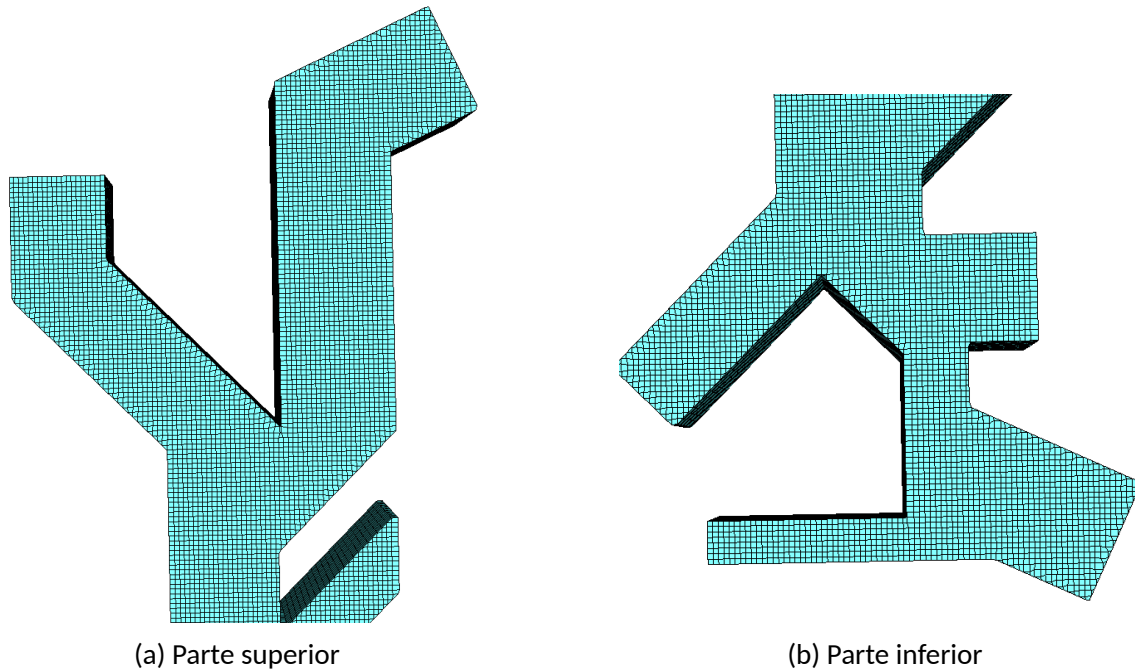


Figura 43: Resultado Fiducials con algoritmo modificado. Refinamiento 4 (elaboración propia).

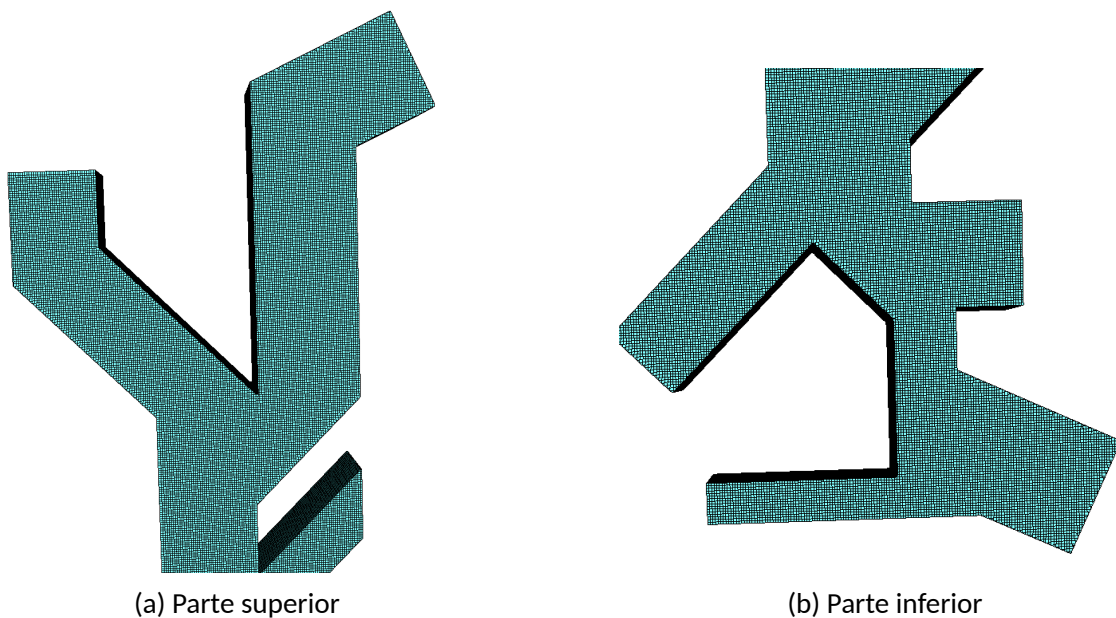


Figura 44: Resultado Fiducials con algoritmo modificado. Refinamiento 5 (elaboración propia).

También ocurre el mismo problema con la tolerancia 0.8. Con esta, no se logra una correcta representación, y se debe a los mismos motivos. En este caso, la figura resultante sería:

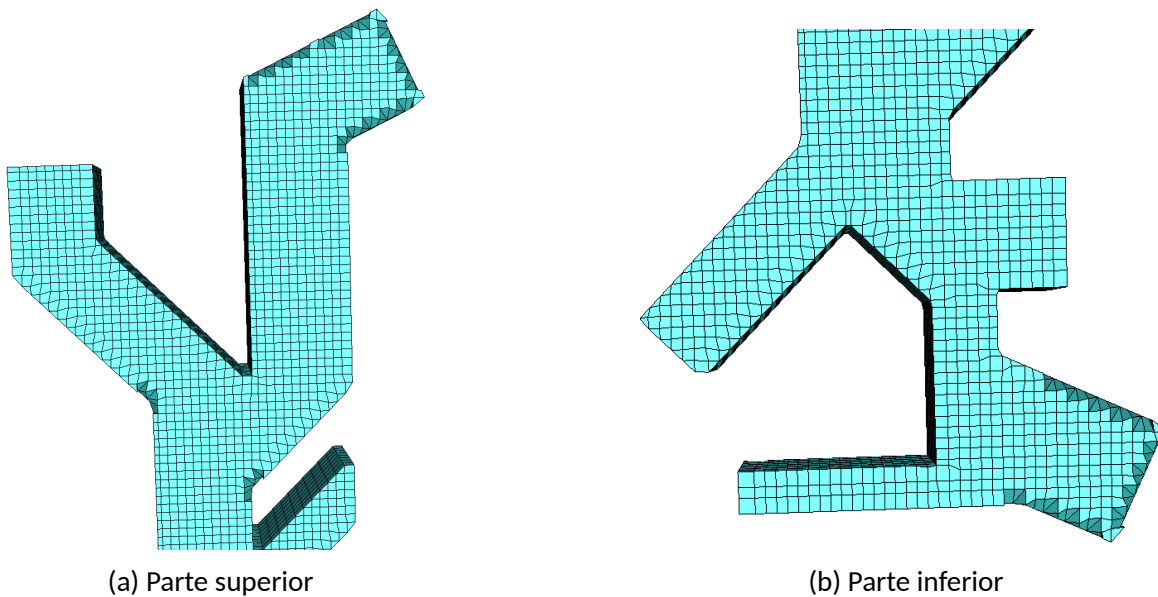


Figura 45: Resultado Fiducials con algoritmo modificado. Refinamiento 3, tolerancia 0.8 (elaboración propia).

Para el algoritmo original también sucede el mismo comportamiento, donde la tolerancia no afecta significativamente su tiempo y calidad, y todos los resultados presentan una mala representación visual. Los resultados según refinamiento son:

Tabla 5: Comparación de calidad y tiempo con algoritmo original (Fiducials).

Fuente: Elaboración propia.

Ref.	Tol. 0.5	
	Cal.	Tpo. [s]
3	0.946	0.892
4	0.958	4.164
5	0.963	17.804

Mientras que los resultados visuales, para todas las combinaciones, presentan siempre el mismo comportamiento en los bordes:

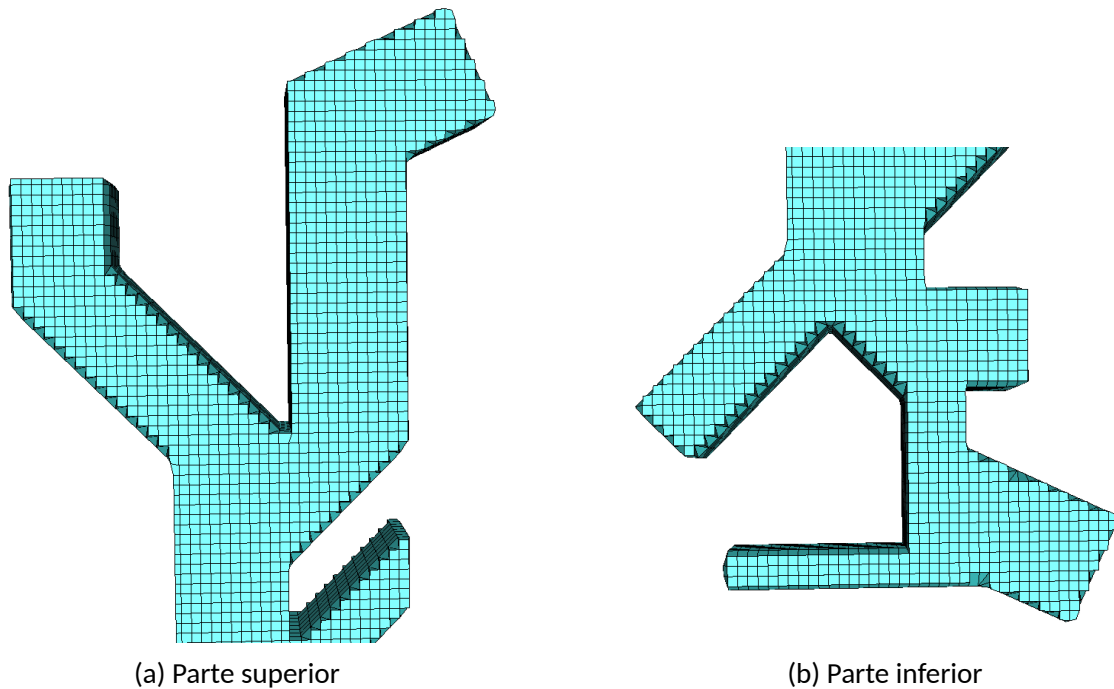


Figura 46: Resultado Fiducials con algoritmo original. Refinamiento 3, tolerancia 0.5 (elaboración propia).

Comparando, se determinó que sucede lo mismo que con la figura Hex Wedge: un pequeño aumento de tiempo y una pequeña disminución en la calidad, pero representando correctamente al objeto.

En esta figura se puede visualizar otro resultado. Este corresponde a la correcta consideración de los octantes que conforman la figura. Se tiene que, para cualquier combinación de atributos, el algoritmo modificado considera correctamente qué octantes pertenecen a la figura, en comparación con el algoritmo original.

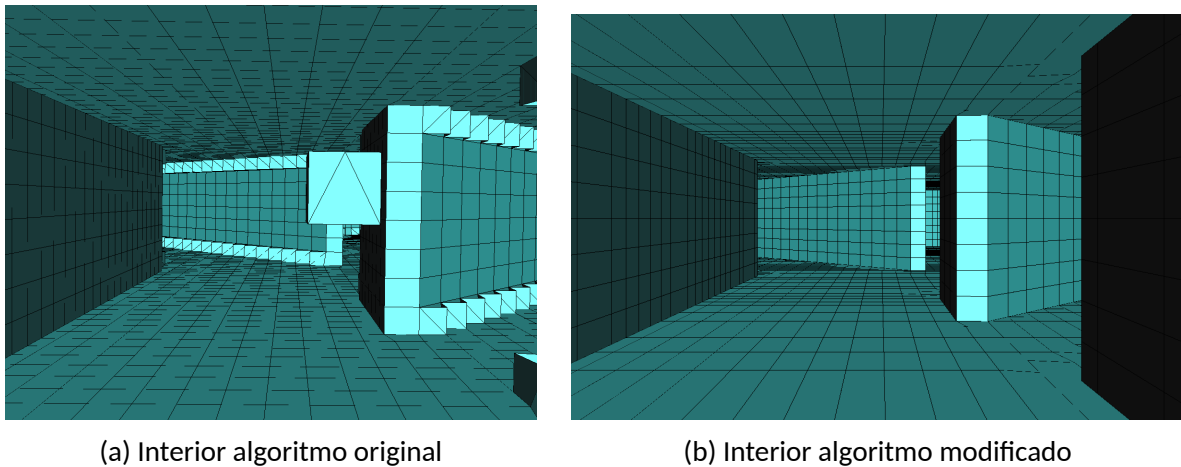


Figura 47: Interior Fiducials. Refinamiento 3, tolerancia 0.5 (elaboración propia).

En la figura 47a, se aprecia que el algoritmo original generaba un error, en donde el hecho de que existan caras “flotando” en el interior del dominio, demuestra que se generaba un problema de inconsistencia entre octantes vecinos. En la figura 47b se aprecia que la congruencia interna se ha restablecido, por lo que ya no hay caras internas “flotando”.

4.3. Hex

Para esta figura, con todas las tolerancias y refinamientos, se tiene una calidad resultante de 1. La principal variación se encuentra en el tiempo: cuanto mayor sea el refinamiento, mayor será este. Los resultados se presentan a continuación:

Tabla 6: Comparación de calidad y tiempo con algoritmo modificado (Hex).

Fuente: Elaboración propia.

Tol. 0.5		
Ref.	Cal.	Tpo. [s]
3	1	0.033
4	1	0.186
5	1	0.673

La figura resultante es siempre la misma, un hexaedro conformado por hexaedros cuya cantidad va a depender del nivel de refinamiento aplicado; esto se ve a continuación:

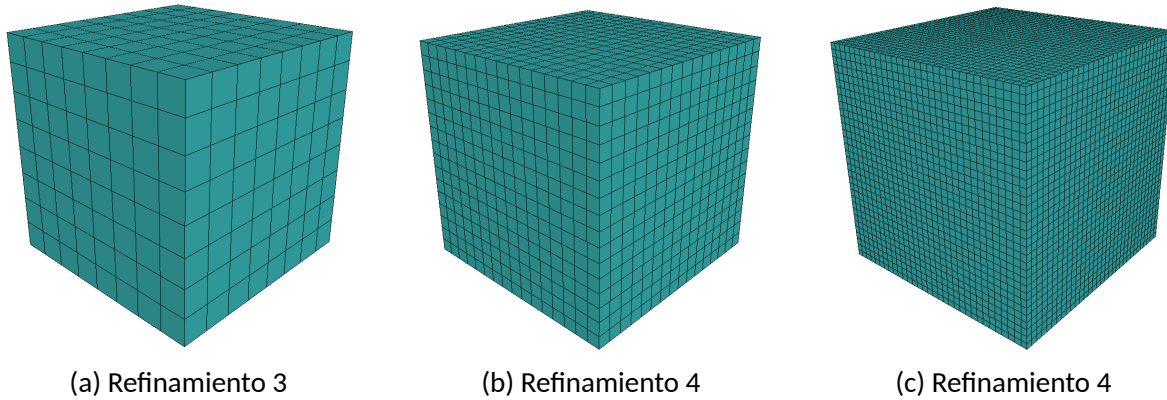


Figura 48: Resultado Hex con algoritmo modificado. (elaboración propia).

Mientras que en el algoritmo original se tiene la misma situación descrita:

Tabla 7: Comparación de calidad y tiempo con algoritmo original (Hex).

Fuente: Elaboración propia.

Tol. 0.5		
Ref.	Cal.	Tpo. [s]
3	1	0.028
4	1	0.147
5	1	0.573

Visualmente, los resultados son:

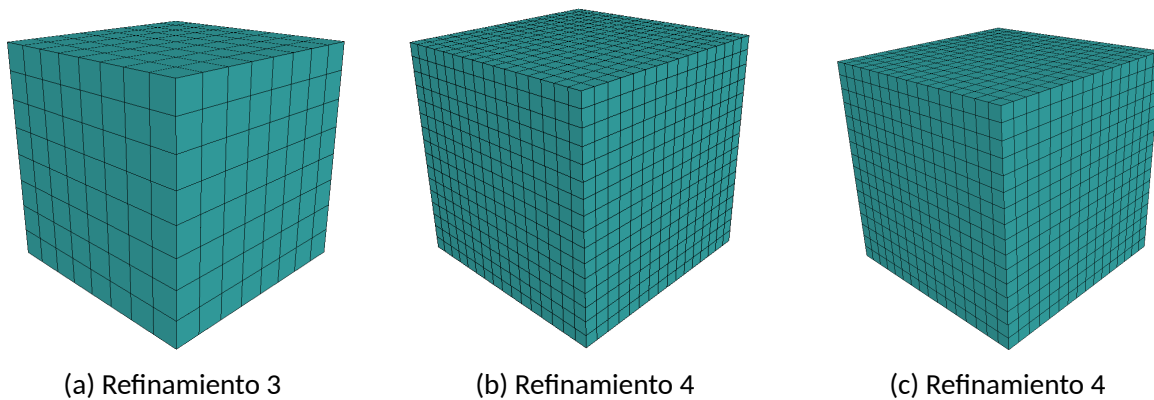


Figura 49: Resultado Hex con algoritmo original. (elaboración propia).

Como se aprecia, los resultados visuales y de calidad son los mismos, con un tiempo similar, destacando que en el algoritmo modificado el tiempo es mayor por una pequeña cantidad.

Para el refinamiento 3, es aproximadamente 0.007 [s]; para el refinamiento 4, es aproximadamente 0.04[s]; y para el refinamiento 5, es aproximadamente 0.1[s]. Siendo una diferencia mínima para cada caso.

4.4. Socket

Esta figura requiere un mayor análisis y trabajo futuro, dado que con las mejoras propuestas aún no se logra representar correctamente. Ésta presenta varias áreas de mejoras con las que se pueden llegar a obtener una solución que funcione para cada objeto.

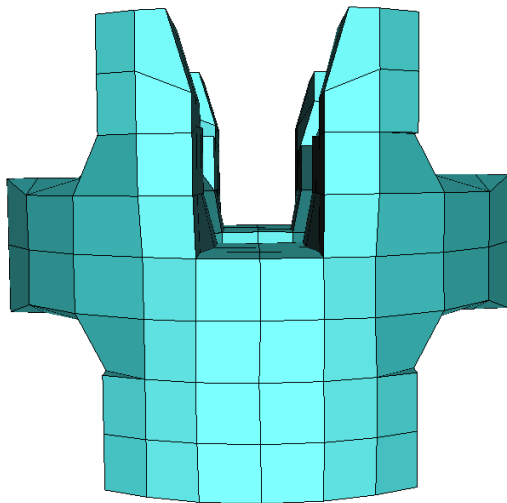
A continuación se presenta la tabla con los resultados, donde se ve que unas variaciones destacables son con el refinamiento 3 y la tolerancia 0.2; es mejor que el refinamiento de 3 con la tolerancia 0.5. Mientras que con el refinamiento 4, con tolerancia 0.5, es mayor la calidad que con tolerancia 0.8. También cabe mencionar que el promedio de la calidad es más bajo en comparación con las otras figuras, considerando que las otras rondan alrededor de 0.9 y esta ronda alrededor de 0.6 a 0.7.

Tabla 8: Comparación de calidad y tiempo con algoritmo modificado (Socket).

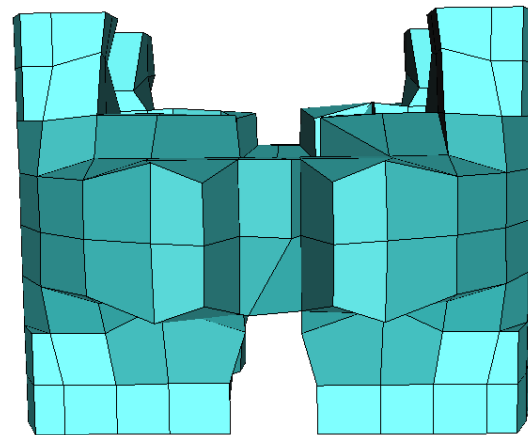
Fuente: Elaboración propia.

Ref.	Tol. 0.1		Tol. 0.2		Tol. 0.5		Tol. 0.8	
	Cal.	Tpo. [s]	Cal.	Tpo. [s]	Cal.	Tpo. [s]	Cal.	Tpo. [s]
3	0.637	0.126	0.696	0.156	0.652	0.154	0.662	0.1396
4	0.637	0.431	0.700	0.472	0.727	0.431	0.672	0.480
5	0.731	1.548	0.720	1.818	0.736	1.907	0.741	1.506

Los resultados visuales más destacables son con refinamiento 3 y tolerancia 0.2:



(a) Figura Socket frente

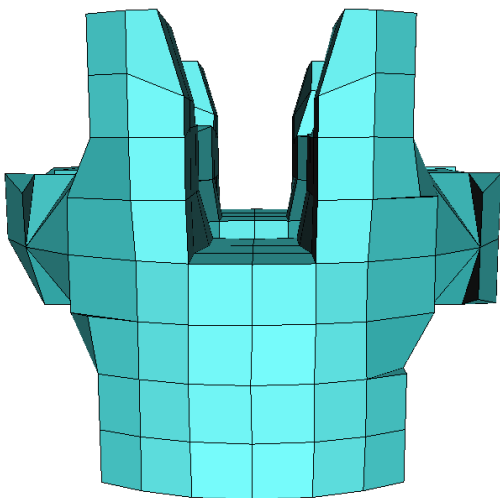


(b) Figura Socket lado

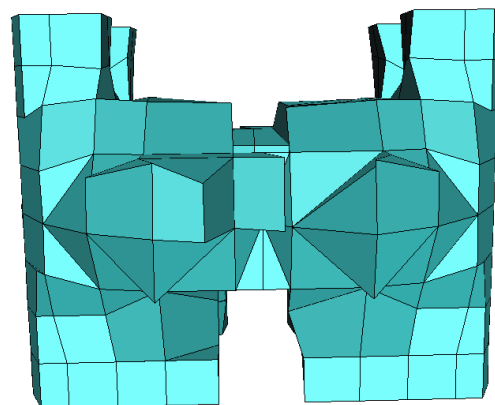
Figura 50: Resultado Socket con algoritmo modificado. Refinamiento 3 y tolerancia 0.2 (elaboración propia).

Aquí se mantiene la forma respecto a los patrones de superficie, pero la proyección de los nodos no es la más adecuada, provocando que se deforme. Esto se observa con mayor claridad en los poliedros del centro de la segunda figura.

La figura con refinamiento 3 y tolerancia 0.5:



(a) Figura Socket frente



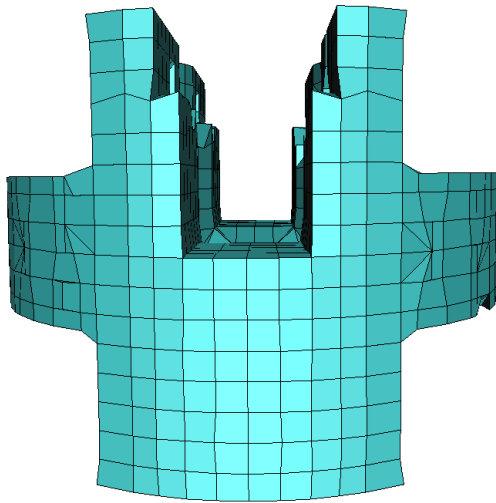
(b) Figura Socket lado

Figura 51: Resultado Socket con algoritmo modificado. Refinamiento 3 y tolerancia 0.5 (elaboración propia).

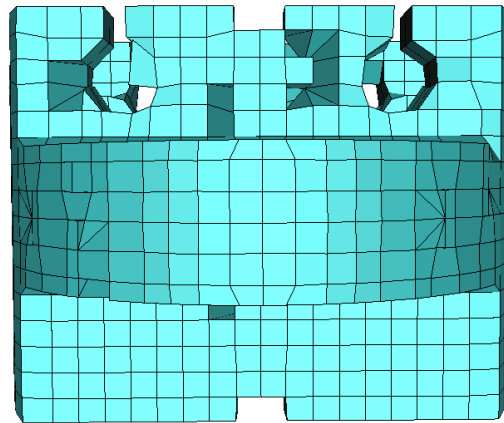
Aquí la forma no se mantiene por los patrones de superficie. Esto también se ve más claro

en los poliedros del centro de la segunda figura.

Para el refinamiento 4 y tolerancia 0.5 se tiene:



(a) Figura Socket frente

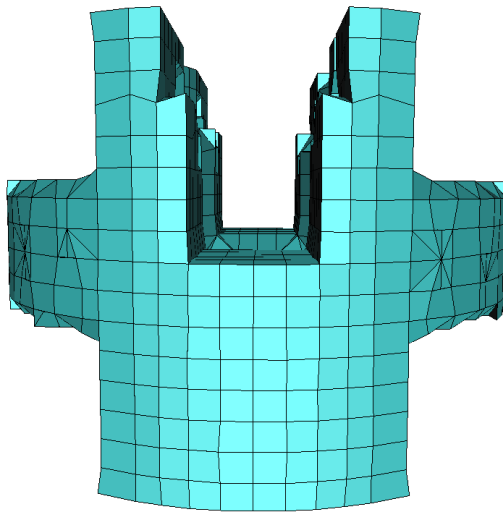


(b) Figura Socket lado

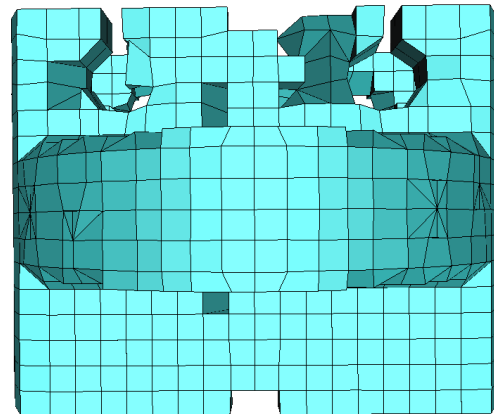
Figura 52: Resultado Socket con algoritmo modificado. Refinamiento 4 y tolerancia 0.5 (elaboración propia).

Se mantiene la forma en gran parte de la figura, pero hay sectores, como cerca de los círculos, que no mantienen las características finas por los patrones de superficie.

Para el refinamiento 4 y tolerancia 0.8 se tiene:



(a) Figura Socket frente

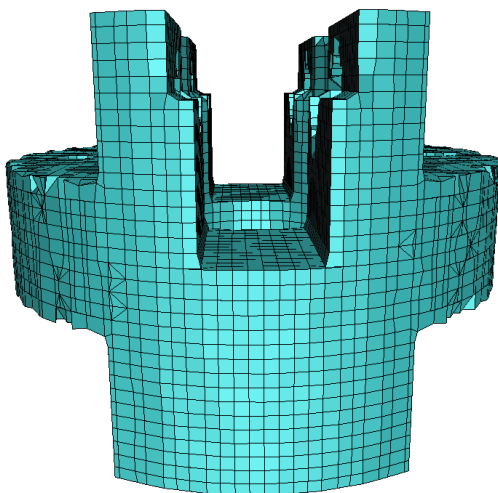


(b) Figura Socket lado

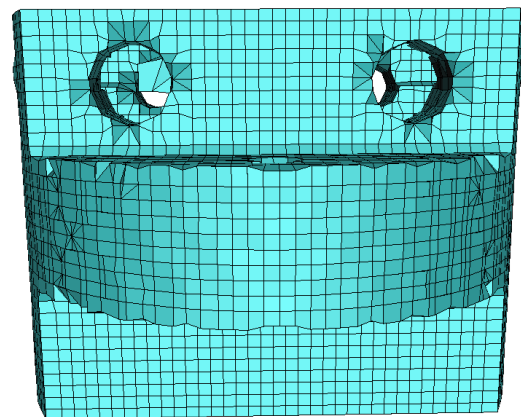
Figura 53: Resultado Socket con algoritmo modificado. Refinamiento 4 y tolerancia 0.8 (elaboración propia).

Aquí se observa que se pierden aún más las características finas por los patrones de superficie. Aquí sucede lo mismo que en los casos ya antes mencionados con tolerancias muy altas; no se aplican correctamente los nuevos patrones porque se consideran más nodos de mala calidad de los que corresponden.

Para el refinamiento 5 y tolerancia 0.5 se tiene:



(a) Figura Socket frente



(b) Figura Socket lado

Figura 54: Resultado Socket con algoritmo modificado. Refinamiento 5 y tolerancia 0.5 (elaboración propia).

Donde se junta que la forma no se mantiene por los patrones de superficie y la proyección hecha. Esto se ve en la zona central y en los círculos de la segunda imagen.

Con todo esto, es correcto afirmar que esta figura requiere un mayor análisis, el cual permitirá mejorar el algoritmo.

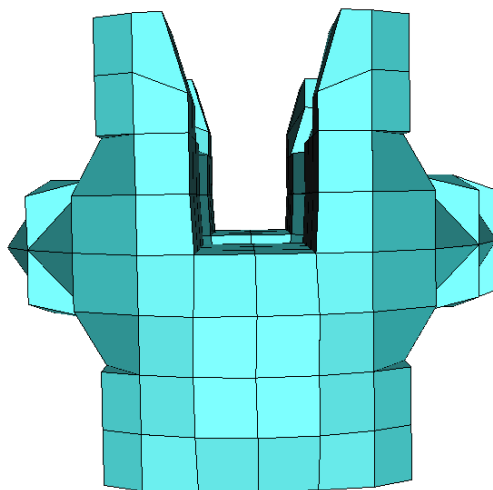
Respecto al algoritmo original, se tienen los siguientes valores:

Tabla 9: Comparación de calidad y tiempo con algoritmo original (Socket).

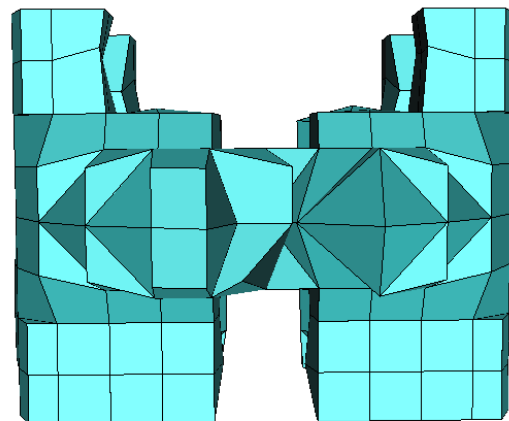
Fuente: Elaboración propia.

Ref.	Tol. 0.1		Tol. 0.2		Tol. 0.5		Tol. 0.8	
	Cal.	Tpo. [s]	Cal.	Tpo. [s]	Cal.	Tpo. [s]	Cal.	Tpo. [s]
3	0.766	0.136	0.832	0.128	0.841	0.133	0.734	0.141
4	0.782	0.365	0.785	0.435	0.769	0.381	0.733	0.459
5	0.849	1.592	0.850	1.657	0.847	1.670	0.822	1.634

Estas calidades son mejores que las presentadas por el algoritmo modificado, pero visualmente en ningún caso mantienen las características finas, debido a los patrones de superficie. A continuación se muestran las mismas figuras con el refinamiento y tolerancia que se mostró para el algoritmo modificado:

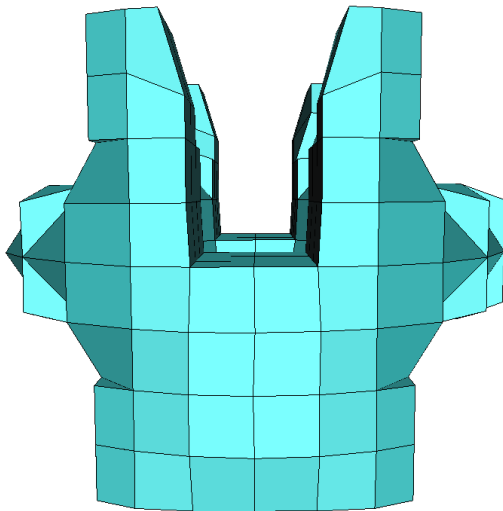


(a) Figura Socket frente

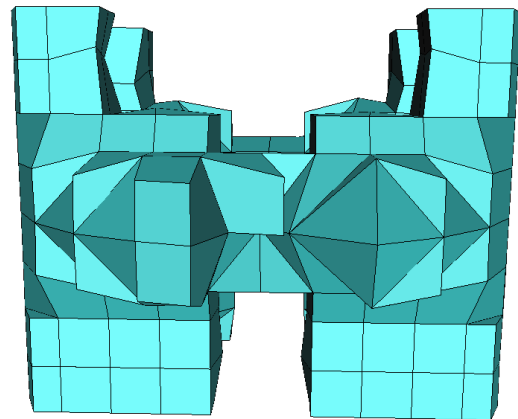


(b) Figura Socket lado

Figura 55: Resultado Socket con algoritmo original. Refinamiento 3 y tolerancia 0.2 (elaboración propia).

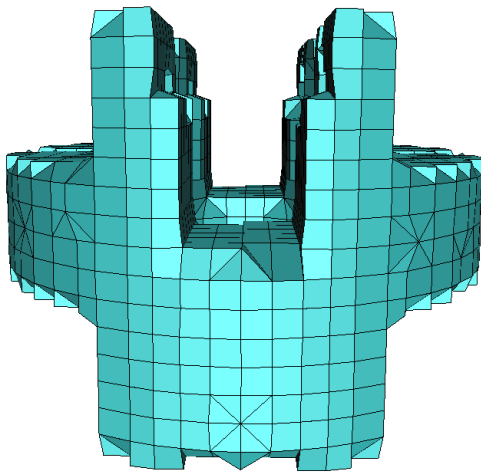


(a) Figura Socket frente

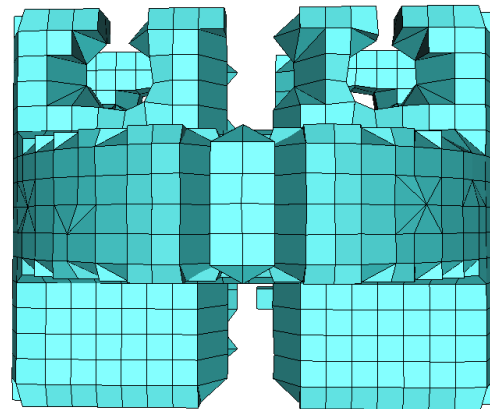


(b) Figura Socket lado

Figura 56: Resultado Socket con algoritmo original. Refinamiento 3 y tolerancia 0.5 (elaboración propia).

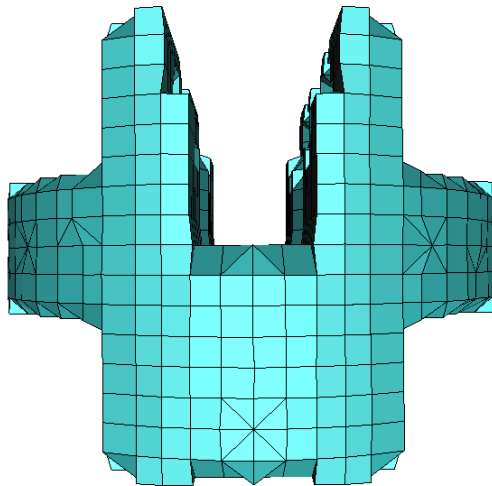


(a) Figura Socket frente

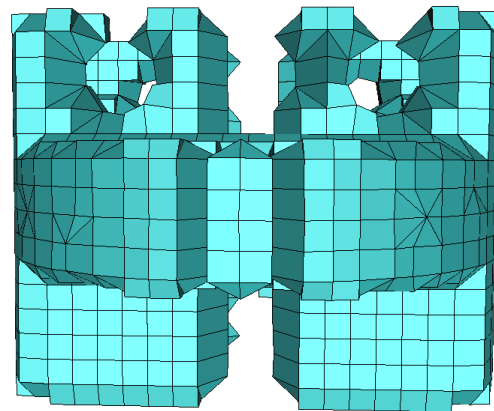


(b) Figura Socket lado

Figura 57: Resultado Socket con algoritmo original. Refinamiento 4 y tolerancia 0.5 (elaboración propia).

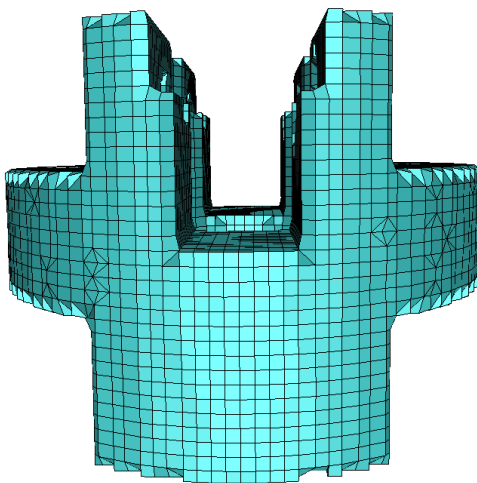


(a) Figura Socket frente

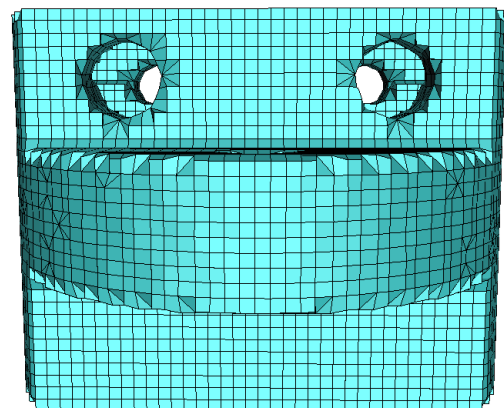


(b) Figura Socket lado

Figura 58: Resultado Socket con algoritmo original. Refinamiento 4 y tolerancia 0.8 (elaboración propia).



(a) Figura Socket frente



(b) Figura Socket lado

Figura 59: Resultado Socket con algoritmo original. Refinamiento 5 y tolerancia 0.5 (elaboración propia).

Cabe comparar el caso del refinamiento 4 y tolerancia 0.5. En el algoritmo original, los octantes que están en el centro de forma vertical desaparecen a pesar de ser parte de la figura, ya que se considera que todos sus nodos están fuera. En el algoritmo modificado, estos se mantienen, dado que a pesar de que sus nodos estén fuera, su volumen está dentro de la figura. Esto requiere trabajo futuro debido a que estos octantes pueden necesitar que se les apliquen patrones de superficie, pero al no tener nodos internos, no hay forma de aplicar los patrones actuales.

4.5. Cortex

Para esta figura, al igual que las anteriores, los valores de tiempo y calidad promedio de los poliedros resultantes tampoco se ven afectados significativamente por las distintas tolerancias. Por tanto, solo se analizará el refinamiento, en el que se encuentran diferencias más destacables que en las otras figuras. A continuación, se presentan los valores obtenidos:

Tabla 10: Comparación de calidad y tiempo con algoritmo modificado (Cortex).

Fuente: Elaboración propia.

Tol. 0.5		
Ref.	Cal.	Tpo. [s]
3	0.454	0.286
4	0.457	0.745
5	0.464	1.892

Visualmente se tiene:

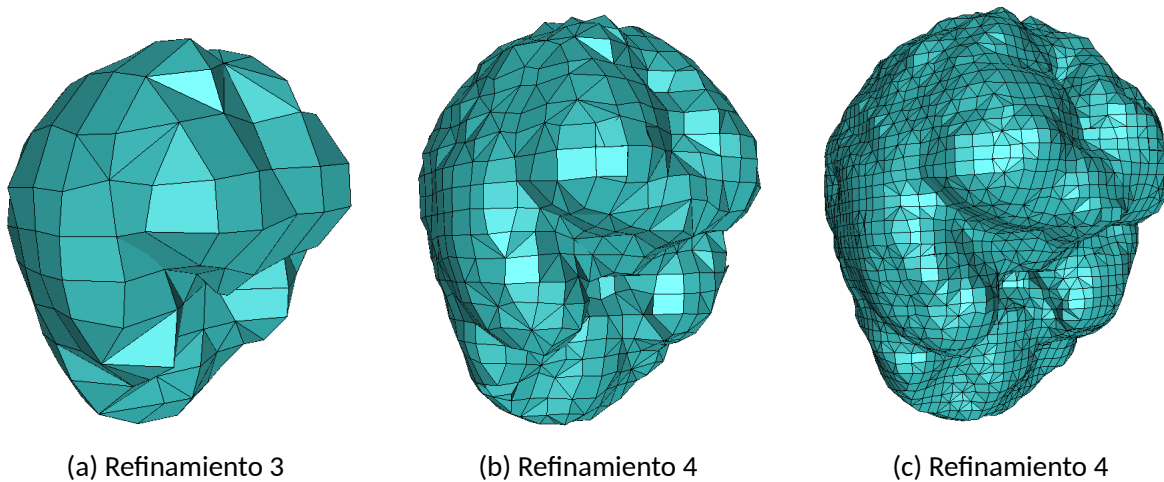


Figura 60: Resultado Cortex con algoritmo modificado. (elaboración propia).

Mientras que el algoritmo original tiene los siguientes valores:

Tabla 11: Comparación de calidad y tiempo con algoritmo original (Cortex).

Fuente: Elaboración propia.

Tol. 0.5		
Ref.	Cal.	Tpo. [s]
3	0.601	0.328
4	0.626	0.726
5	0.624	1.786

Los resultados visuales son:

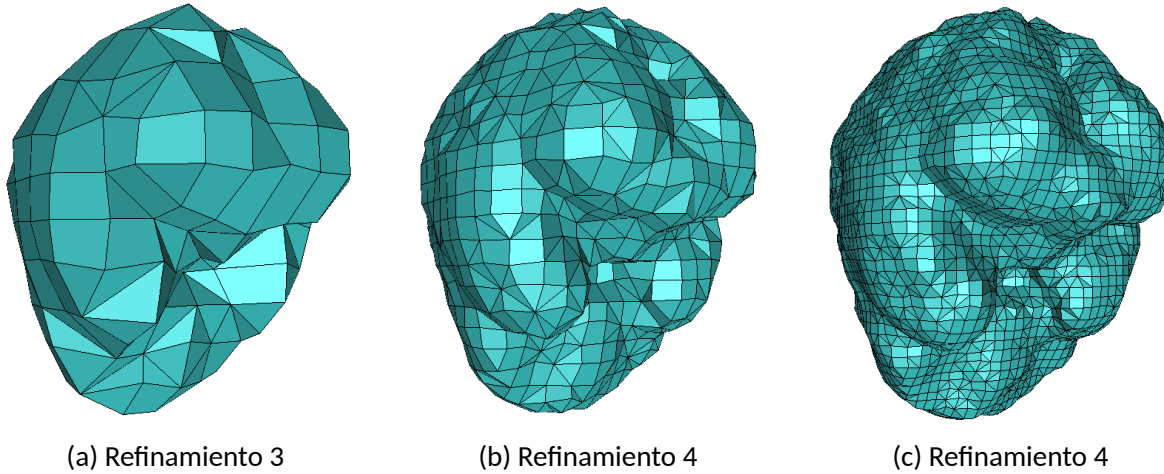


Figura 61: Resultado Cortex con algoritmo original (elaboración propia).

Comparándolos visualmente no se ven diferencias distinguibles, pero la calidad bajó considerablemente, en un 0.2. El tiempo aumentó un poco, de manera similar que el resto de figuras. Cabe destacar que con el algoritmo original la figura resultante sigue presentando una mala calidad.

Esta figura, al igual que la figura Socket, requiere un análisis más profundo, el cual también permitirá mejorar el algoritmo.

CAPÍTULO 5

CONCLUSIONES

El problema principal abordado es la corrección de la representación de características finas en un algoritmo creado para curvar las figuras. Este problema ya se había intentado solucionar a través de una heurística que filtraba cuándo se debían aplicar los patrones, dada la calidad de los nodos y una tolerancia, pero que aun así no los representa correctamente para todos los casos.

Las figuras que se utilizaron para modificar y probar el algoritmo fueron principalmente la Hex Wedge, Fiducials y, en menor medida, las Hex, Socket y Cortex.

El algoritmo, una vez modificado, fue evaluado en comparación con la versión original. En las dos primeras figuras se presentó una buena calidad promedio, con una representación correcta de las características finas, aunque con una pequeña disminución en la calidad y un leve aumento en el tiempo. En la figura Hex, al ya estar correctamente representado por el algoritmo original, solo tuvo un pequeño aumento en el tiempo. Para el caso del Socket, el algoritmo modificado sigue fallando en representar las características finas; presenta mala calidad, con una disminución y un pequeño aumento del tiempo. Por último, la figura Cortex presenta la calidad más baja, en comparación con las otras figuras. Esta disminuyó considerablemente y tiene tiempos variados entre los dos algoritmos.

Considerado lo realizado y lo observado, se concluye que la solución propuesta no corrige todos los aspectos necesarios del algoritmo, pero sí cumple parcialmente el objetivo principal de representar correctamente las características finas de una malla tipo octree. Los casos donde se cumple de forma completa se ven en las figuras Hex Wedge y Fiducials, pero no lo logra en la figura del Socket.

El objetivo secundario del posicionamiento de los nodos internos de acuerdo a los nodos externos se cumple a través de la modificación del `projectCloseToBoundaryNodes`. Ahora, cuando se proyecta un nodo, se revisan los nodos vecinos, y a cada uno se le aplica la misma traslación que al nodo proyectado, pero únicamente en los ejes que coinciden con los del nodo antes de proyectarse. Si un nodo es vecino de más de un nodo proyectado, se hace la traslación de mayor magnitud entre las posibles. De esta forma, se trasladan los nodos internos del octante, según los nodos externos. Este cambio permite identificar de mejor manera los nodos que presentan mala calidad cuando se aplica `shrinkToBoundary` y se proyecten los nodos externos sobre la malla. El `projectCloseToBoundaryNodes` aún presenta área de mejora por dos motivos: existen casos donde los nodos proyectados no terminan de mover el octante fuera de la malla, y existen casos donde no ocurre correctamente la traslación de los nodos vecinos, haciendo que se deforme el octante. Para el primer caso, se agregó un eliminador de octantes que compara la media de volúmenes de los octantes y elimina aquellos que presentan muy poco volumen en comparación. Cabe mencionar que lo más adecuado es cambiar la condición de proyección o modificar hacia dónde se está

proyectando. Para lo segundo, se requiere un mayor análisis del método y las situaciones en que sucede para determinar su causante.

El segundo objetivo secundario de analizar y corregir problemas detectados en el algoritmo original también se cumplió. Con este fin, se analizó el algoritmo y se hallaron los siguientes problemas en distintas secciones del código:

- Problemas en la generación de la malla octree, lo que puede provocar que ciertos octantes no sean considerados en esta.
- Falta de patrones de superficie para mantener la forma de la figura en ciertos casos.
- Descarte de octantes necesarios, cuando se aplican los patrones de superficie.

El primero se hallaba en el método `isItIn`, el cual analizaba dos nodos en los extremos del octante y según eso determinaba si es que se incluía o no, pero en caso de que ambos difieran, solo se consideraba el resultado de uno de los nodos. Para esto se modificó de forma que, si se considera que uno de los nodos está dentro, también incluya el octante, y de esta forma no se pierde parte de la figura, ya que dicho octante tendría parte o estaría completamente en la figura.

El segundo se identificó en los patrones que manejan los octantes con un nodo interno. Estos no lograban representar correctamente cuando se requería transformar el octante en un prisma triangular. En el algoritmo original, solo existía un patrón para este caso, el cual siempre daba como resultado un tetraedro. Por esto, se agregaron dos patrones, ambos generan prismas triangulares. Según la cantidad de nodos que presentan mala calidad y cuáles son, se transforma en un prisma triangular si corresponde; en otro caso se transforma en un tetraedro. Las limitantes que presentan estos patrones son las siguientes:

- Combinaciones no vistas para los nodos que presentan mala calidad, dado que existieron casos que se analizaron y resolvieron para cuando los nodos que presentan mala calidad están ligados a un nodo interno. Pueden existir otras combinaciones inesperadas como ésta, que no se han presentado.
- Cuando todos los nodos de un octante aparecen en más de un poliedro, ya que los patrones fueron creados, para la situación en que exista un nodo que solo esté presente una única vez en el octante al que se le aplicará el patrón.

Estas dos limitaciones requieren un mayor análisis para determinar en qué situaciones podría presentarse y cuáles son los poliedros que más se adecuarían. La solución a éstas puede requerir la creación de nuevos patrones o la modificación de los actuales.

Cabe mencionar que, al agregar estos nuevos patrones, existen casos donde no se deben aplicar ciertos patrones a los octantes vecinos. Por ello, primero se aplicaron los patrones a los octantes en los que el cálculo dio mala calidad, y después se aplicaron los patrones a los octantes que comparten nodos con estos, pero solo si lo requieren los resultados de la

aplicación de los primeros patrones.

El último caso ocurre cuando todos los nodos del octante se consideran externos, ya que se encuentran posicionados sobre la superficie de la malla, pero puede ocurrir que el volumen de dicho octante esté dentro de ésta. Estos octantes se descartan cuando se aplican los patrones de superficie, dado que funcionan con al menos un nodo interno. Cabe mencionar que para este punto, solo deben existir octantes que realmente pertenezcan a la figura. Por ello, en vez de eliminarlos, estos se mantuvieron para que la figura fuera más representativa. No obstante, este cambio no contempla si es que el octante requiere alguna transformación para que sea el más adecuado a la figura, por lo que hay que crear patrones específicos para estas situaciones.

Todavía hay casos pendientes. Existen problemas previos a las modificaciones que fueron realizadas. En particular, son casos pendientes con los patrones de transición y los patrones de superficie. Estos se detallan a continuación:

Los patrones de transición se aplican primero que los de superficie. En la solución original siempre se aplicaban los patrones de superficie, conectando correctamente con los de transición, pero actualmente, con la heurística y los cambios realizados, los patrones de superficie no siempre se aplican, o no existe un patrón de transición adecuado para complementar al patrón de superficie. Por ello, al aplicarse primero los de transición, no toman en cuenta los resultados posteriores, generando una mala conexión entre los octantes, como se muestra a continuación:

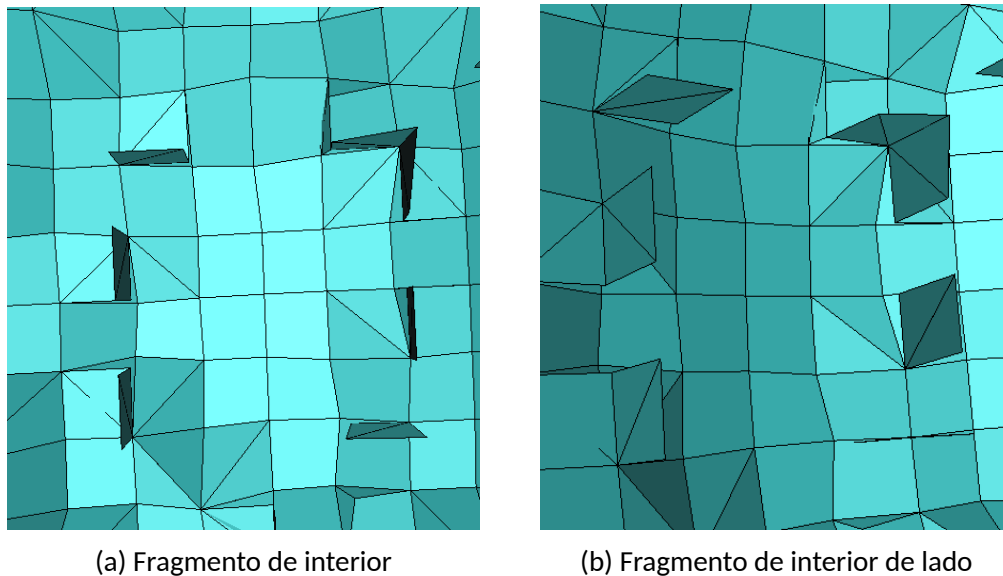


Figura 62: Interior figura cortex con patrón de transición mal aplicado (elaboración propia).

Como se ve en la figura 62, se presentan caras flotando; esto ocurre porque las caras que coinciden entre dos octantes entre sí no son iguales; por ejemplo, una que es solo un cuadrado y otra que son dos triángulos juntos. Esto se debe a que no se están aplicando

correctamente los patrones de transición. Lo ideal es que se mueva la lógica tras la aplicación de los patrones de superficie, permitiendo saber cómo debe modificar los octantes vecinos para lograr correctamente las transiciones.

Los patrones de superficie tienen problemas respecto a la calidad y a la forma resultante. Lo primero ocurre porque, a pesar de la aplicación de los patrones, se sigue obteniendo poliedros con calidad cercana a 0 o negativa. Lo segundo ocurre porque hay casos en que los patrones se aplican si es que el octante comparte nodos de mala calidad, y no siempre el patrón aplicado es el más adecuado para mantener la forma de la figura; esto se ve a continuación:

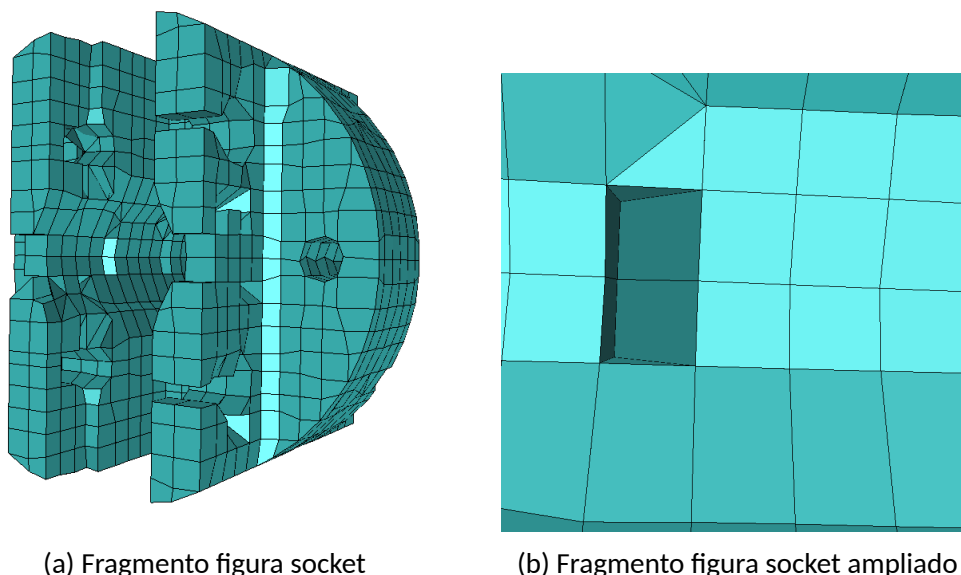


Figura 63: Figura socket con patrones de superficie aplicados (elaboración propia).

Aquí se muestra la figura del Socket con los patrones de superficie aplicados. En particular, se destaca la figura 63(b), donde se observa que dos octantes se transforman en prismas triangulares. Esto ocurre porque uno de los octantes comparte una cara con otro, y dicha cara está formada por dos triángulos. Aunque el prisma triangular se adapta a esta situación, no es la solución más adecuada para mantener la forma. En este caso, lo más conveniente sería transformar el octante en un conjunto de pirámides. Este conjunto genera una cara que coincide con la formada por triángulos y, además, conserva una cara cuadrilátera que se comparte con el otro octante convertido en prisma triangular. Por ello, es necesario que este último mantenga su forma original.

Para resolver estos casos, es necesario definir nuevos patrones. Esto requiere un análisis detallado de los patrones existentes y de las condiciones en que se utilizan, con el fin de asegurar una calidad adecuada y conservar la forma original.

Por último, es importante destacar los aportes más significativos de este trabajo. Estos

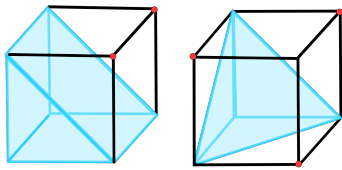
corresponden a los patrones agregados para cuando se presente un nodo interno, y el mecanismo propuesto para diferenciar cuándo se deben aplicar o no los patrones. Ambos, para futuras modificaciones, serán muy necesarios.

Con todo esto, se aprecia que el algoritmo aún presenta limitaciones para ser aplicado en todo tipo de geometrías. Sin embargo, los tiempos de ejecución con las modificaciones son pequeños, y la calidad para la mayoría de las figuras también es buena. Ambas métricas no variaron significativamente en la mayoría de los casos. Por ello, es recomendable continuar su análisis y mejorar este algoritmo, para que en un futuro sí se pueda usar para todo tipo de objetos. Para ello, es necesario centrarse en los aspectos previamente abordados y hacer pruebas tanto en figuras con partes curvas como con características finas. El análisis de estas figuras puede ser de gran ayuda para modificar el algoritmo.

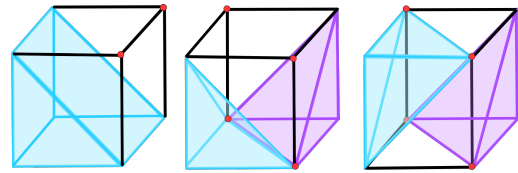
REFERENCIAS BIBLIOGRÁFICAS

- [1] Christoffer (2024). Conformal meshing in simcenter star?ccm+. <https://volume.com/simcenter-star-ccm/conformal-meshing-in-simcenter-star-ccm/>. Acces-
sed: 2025-07-02.
- [2] de Almeida, J. P. Moitinho and Díez, P. and Tiago, C. and others (2013). De imágenes mé-
dicas segmentadas a mallas de superficie y volumen, utilizando herramientas y algoritmos
existentes. *Modelado y Simulación Adaptativa*, p. 115.
- [3] Ecured (s/f). Computación gráfica. https://www.ecured.cu/Computaci%C3%B3n_Gr%C3%A1fica. Recuperado el 10 de septiembre de 2024.
- [4] Lobos, C. y González, E. (2015). Mixed-element octree: a meshing technique toward fast
and real-time simulations in biomedical applications. *International journal for numerical
methods in biomedical engineering*, 31(12).
- [5] Marschner, Steve y Shirley, P. (2009). *Fundamentos de gráficos por computadora*. CRC
press.
- [6] Ramírez, I. A. F. (2023). Método rápido de representación de características finas en una
malla octree. Memoria para optar al título de ingeniero civil en informática, Universidad
Técnica Federico Santa María.
- [7] S/f (2025). Nq5 (pdf document). [https://blogs.ed.ac.uk/christopherhall/
wp-content/uploads/sites/6693/2024/04/NQ5.pdf](https://blogs.ed.ac.uk/christopherhall/wp-content/uploads/sites/6693/2024/04/NQ5.pdf). Recuperado el 23 de junio de
2025.
- [8] Yerry, M. A. y Shephard, M. S. (1984). Automatic three-dimensional mesh generation by
the modified-octree technique. *International journal for numerical methods in enginee-
ring*, 20(11):1965–1990.

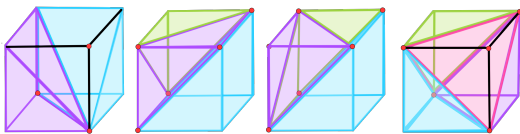
ANEXOS



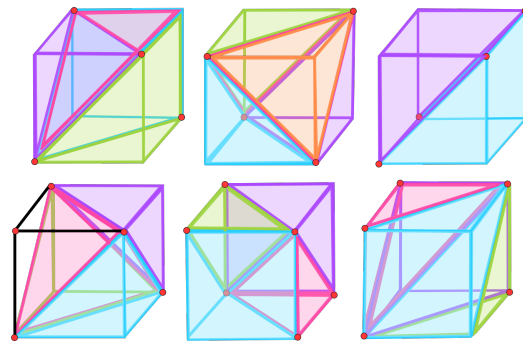
(a) SurfeTemplate 1: Octantes con un nodo interno.



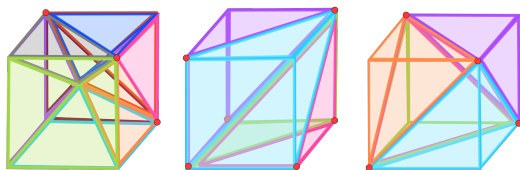
(b) SurfeTemplate 2: Octantes con dos nodos internos.



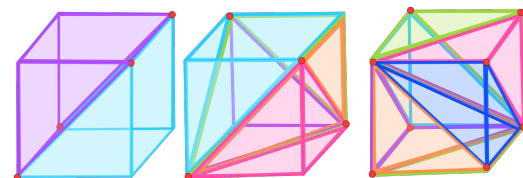
(c) SurfeTemplate 3: Octantes con tres nodos internos.



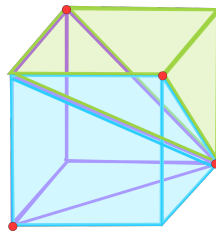
(d) SurfeTemplate 4: Octantes con cuatro nodos internos.



(e) SurfeTemplate 5: Octantes con cinco nodos internos.



(f) SurfeTemplate 6: Octantes con seis nodos internos.



(g) SurfeTemplate 7: Octantes con siete nodos internos.

Figura 64: Patrones de superficie aplicados a octantes según la cantidad de nodos internos.