

UNIVERSIDAD TÉCNICA FEDERICO SANTA MARÍA
DEPARTAMENTO DE INFORMÁTICA
SANTIAGO - CHILE



“CÁLCULO Y GUARDADO DE INFORMACIÓN RELEVANTE DE UNA MALLA OCTREE.”

JOAQUÍN GONZALO CONTRERAS MAULEN

MEMORIA PARA OPTAR AL TÍTULO DE
INGENIERO CIVIL EN INFORMÁTICA

Profesor Guía: Jorge Díaz Matte
Profesor Correferente: Claudio Lobos Yañez

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: CÁLCULO Y GUARDADO DE INFORMACIÓN RELEVANTE DE UNA MALLA OCTREE.

Nombre del candidato(a): Joaquin Gonzalo Contreras Maulen

Carrera / Grado: Ingeniería Civil Informática

Campus: Santiago San Joaquín ; Departamento: Informática

2.- VALIDACIÓN DEL PROFESOR GUÍA/DIRECTOR DE TESIS

Yo, Jorge Díaz Matte, 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: 20/8/25 ; Firma:

Estudiante o Candidato(a):

Fecha: 12/08/2025

; Firma: Joaquin Contreras

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

Dedico este trabajo a mi familia y amistades junto a todos aquellos que formaron parte de alguna forma u otra en esta etapa de mi vida, les estoy profundamente agradecido.

AGRADECIMIENTOS

Me considero una persona de pocas palabras y no creo poder hacer justicia a la gratitud que le debo a múltiples personas que me han ayudado y apoyado en mi proceso de titulación. En primer lugar creo que debo reconocer todo lo que me ha brindado mi familia, mi madre María Eugenia y mi Padre Ramón, quienes no han escatimado esfuerzos en permitirme desarrollarme como profesional y persona. Adicionalmente otro gran aporte que pude recibir en esta tarea fue el poder compartir mis preocupaciones y problemas con mi hermano Javier, mis amistades y aquellos que viven el mismo proceso que yo, entre ellos podemos nombrar a Martín Rodríguez, Alonso Mancini, Nicolás Moraga, Fabian Muñoz, Pablo Contreras, Tomas Rebolledo, Antonio Briganti.

Adicionalmente debo destacar la dedicación y apoyo que he recibido de mi profesor guía Jorge Diaz quien no solo me acepto como su memorista sino que además siempre veló por mi éxito y atendió a todos los aspectos que requerí durante la elaboración de mi memoria, forma oportuna y con la mayor de las consideraciones a pesar de las dificultades que me surgieron y que pude haberle ocasionado. De la misma forma me gustaría agradecerle al profesor Claudio Lobos por encomendarme al profesor Jorge Diaz. Cuando le fui a solicitar algún tema, el fue capaz de asumir no poder darme el tiempo que un alumno requeriría, pero no me rechazó, sino que me presento mi actual profesor guía, lo cual interpreto como una postura que pone en primer lugar brindar las mejores condiciones al alumno, la cual aprecio profundamente.

A pesar de lo mencionado al inicio, espero poder continuar agradeciendo a las personas mencionadas y muchas mas en el futuro, con mis acciones y con la forma en la que decidiré vivir mi vida de ahora en adelante.

RESUMEN

Resumen— Las mallas geométricas son utilizadas para discretizar un dominio continuo del espacio, empleadas en simulaciones numéricas, renderizado, moldeado y diseño. Un tipo de malla son las basadas en el octree, que son aquellas que se generan a partir de una estructura de árbol octal. En esta memoria se presenta una mejora en el generador de mallas tipo octree, la cual incorpora información adicional al archivo de salida tales como: las caras de la malla, caras pertenecientes a la superficie, vectores normales de dichas caras y vectores normales de los vértices superficiales.

Palabras Clave— Generación de Mallas, Malla Superficial, Vector normal de cara, Vector normal de vértice.

ABSTRACT

Abstract— Geometric meshes are used to discretize a continuous domain of space, and are employed in numerical simulations, rendering, modeling, and design. One type of mesh is the octree-based mesh, which is generated from an octal tree structure. This thesis presents an enhancement to the octree mesh generator, which incorporates additional information into the output file, such as: the mesh faces, the surface faces, the normal vectors of those faces, and the normal vectors of the surface vertices.

Keywords— Mesh Generation, Surface Mesh, Face normal vector, Vertex normal vector.

GLOSARIO

CAD: Computer-Aided Design (Diseño asistido por computador).

CSV: Comma-Separated Values, formato de archivo de texto plano utilizado para representar datos tabulares.

Edge Based Data Structure: Estructura que utiliza información de aristas para representar relaciones entre elementos.

Feature: Punto en la superficie con un ángulo fuera de un umbral de tolerancia, considerado significativo geométricamente.

FEM: Finite Element Method (Método de Elementos Finitos), técnica numérica para simulaciones físicas.

Glyph: Técnica de visualización que representa datos vectoriales como flechas o símbolos.

Mesh: Algoritmo generador de mallas basado en Quadtree/Octree descrito por [Díaz et al., 2023](#) y principal foco de esta memoria.

Mixed elements: Elementos de diferentes tipos (hexaedros, tetraedros, prismas, etc.) en una misma malla.

Newell method: Algoritmo para calcular la normal de un polígono no necesariamente plano.

Node Based Data Structure: Estructura de datos que representa regiones y vecindades a partir de nodos.

Normal de cara: Vector perpendicular a una cara poligonal, usado para sombreado o simulaciones físicas.

Normal de vértice: Vector promedio de las normales de las caras que comparten un vértice.

Producto cruz: Operación vectorial que entrega un vector perpendicular a dos vectores dados.

QuadTree / Octree: Estructura jerárquica de subdivisión espacial en 2D (QuadTree) o 3D (Octree).

Refinement Level (RL): Nivel de refinamiento que indica la cantidad de subdivisiones sucesivas de una celda.

Shrink to boundary: Proceso que proyecta nodos fuera de la superficie hacia el interior del dominio objetivo.

Splitting: Proceso automático del software Paraview que puede dividir vértices en caras adyacentes.

VTK: Visualization ToolKit, librería para visualización científica y su formato de archivo de salida.

ÍNDICE DE CONTENIDOS

RESUMEN	IV
ABSTRACT	IV
GLOSARIO	V
ÍNDICE DE FIGURAS	VIII
ÍNDICE DE TABLAS	VIII
INTRODUCCIÓN	1
CAPÍTULO 1: DEFINICIÓN DEL PROBLEMA	2
1.1 Contexto del problema	2
1.2 Objetivos	3
CAPÍTULO 2: MARCO CONCEPTUAL	5
2.1 Refinamiento	5
2.2 Balanced	6
2.3 Estado del arte (Estructuras de datos)	8
2.3.1 Estructura basada en árbol (<i>QuadTree/Octree</i>)	8
2.3.2 Estructura basada en arcos (<i>Edge Based Data Structure</i>)	9
2.3.3 Estructura basada en nodos (<i>Node Based Data Structure</i>)	10
2.4 Patrones de transición	10
2.5 <i>Adjust to boundary</i> y patrones de superficie	11
2.6 VTK (<i>The Visualization ToolKit</i>)	12
2.7 Normal de una cara	13
2.7.1 Normal de polígono triangular	14
2.7.2 Normal de un cuadrilátero	14
2.8 Normal de un vértice	15
CAPÍTULO 3: PROPUESTA DE SOLUCIÓN	16
3.1 Extracción de caras	16
3.1.1 Hexaedro	17
3.1.2 Tetraedro	18
3.1.3 Pirámide Cuadrada	19
3.1.4 Prisma triangular	20
3.2 Caras superficiales y sus normales	21
3.3 Normal del vértice	22

CAPÍTULO 4: VALIDACIÓN DE LA SOLUCIÓN	25
4.1 Extracción de caras	25
4.2 Normales de caras y vértices superficiales	27
CAPÍTULO 5: CONCLUSIONES	31
5.1 Conclusiones Generales	31
5.2 Cumplimiento de objetivos y validación	31
5.3 Trabajo Futuro	32
ANEXOS	33
REFERENCIAS BIBLIOGRÁFICAS	35

ÍNDICE DE FIGURAS

1	Ejemplo de refinamiento de un pentágono regular.	7
2	Ejemplo de QuadTree.	8
3	Ejemplos de estructura de datos basada en arcos	9
4	Patrones de transición 2D	11
5	Elementos básicos: hexaedro (a), prisma triangular (b), pirámide cuadrada (c) y tetraedro (d)	11
6	Ejemplo de elemento tipo hexaedro	17
7	Ejemplo de elemento tipo tetraedro	18
8	Ejemplo de elemento tipo pirámide cuadrada	19
9	Ejemplo de elemento tipo prisma triangular	20
10	Comparación de caras superficiales.	26
11	Comparación de normales de caras.	29
12	Comparación de normales de vértices.	30

ÍNDICE DE TABLAS

1	Estructura de arcos.	9
2	Estructura de nodos.	10
3	Índices de vértices hexaedro	17
4	Índices de vértices tetraedro	18
5	Índices de vértices pirámide cuadrada	19
6	Índices de vértices prisma triangular	20
7	Aceptación de normales para error máximo 10^{-6}	28

INTRODUCCIÓN

La generación de mallas es una etapa fundamental en el modelado geométrico y la simulación computacional. Una malla corresponde a la discretización de un dominio espacial mediante la subdivisión de dicho espacio en elementos geométricos simples, que permiten aproximar numéricamente fenómenos físicos, estructurales o visuales. Esta técnica es ampliamente utilizada en áreas como la ingeniería, biomecánica, visualización científica, gráficos por computador y análisis numérico, entre otros.

Particularmente, las mallas superficiales tienen como propósito representar de manera precisa únicamente la geometría externa de un objeto tridimensional, sin necesidad de discretizar su volumen. Esto resulta especialmente útil en contextos en los cuales el propósito de la malla es conocer información y simular interacciones con la superficie, como renderizado, animaciones o simulaciones físicas, como podría ser el flujo de fluidos, entre otras. Para garantizar un modelado adecuado, estas mallas deben no solo aproximar la forma del objeto, sino también capturar la orientación geométrica de sus caras y vértices. Aquí es donde cobra relevancia el cálculo de vectores normales, los cuales definen la dirección perpendicular de la superficie y permiten calcular correctamente interacciones con luz, fuerzas o condiciones del entorno.

En este contexto, la presente memoria se centra en una extensión de las capacidades del generador de mallas tipo Octree [Díaz et al., 2023]. Se realizó la investigación del funcionamiento del generador de malla, el estudio de su formato de salida de datos actual y las implicancias que este tiene sobre la elección de metodologías y algoritmos empleados en la solución. Permitiendo identificar automáticamente las caras de la superficie de la malla generada, calcular sus vectores normales y, a partir de ellos, determinar las normales de cada vértice superficial.

La solución implementada se integra directamente sobre el generador, permitiendo una ejecución autónoma sin depender de software externo. Además, se valida cuantitativamente mediante comparaciones con resultados obtenidos mediante el software Paraview, garantizando así la validez de los datos producidos. Este trabajo habilita el uso del generador en nuevas aplicaciones que requieren una información detallada de la superficie mallada.

CAPÍTULO 1

DEFINICIÓN DEL PROBLEMA

1.1. Contexto del problema

La generación de mallas es una etapa fundamental dentro del campo del modelado geométrico y el análisis numérico, especialmente en disciplinas como la ingeniería, la física computacional, la animación 3D y la medicina [George *et al.*, 2017]. Su origen se encuentra en el análisis por elementos finitos (FEM), un método numérico ampliamente utilizado desde mediados del siglo XX para resolver problemas complejos en mecánica estructural, dinámica de fluidos, transferencia de calor, entre otros.

Una malla consiste en una subdivisión del dominio espacial de interés en elementos geométricos simples, como triángulos y cuadriláteros para el caso 2D, o tetraedros, hexaedros, entre otros, para el caso 3D.

El uso de mallas es fundamental para modelar estructuras físicas en entornos digitales, lo que habilita desde simulaciones estructurales en puentes y edificios hasta modelado de órganos humanos para planificación quirúrgica. En computación gráfica, las mallas permiten representar superficies complejas con alta fidelidad visual.

Entre sus muchas formas, las mallas superficiales ocupan un lugar central en aplicaciones que involucran la representación y análisis de la geometría externa de un objeto tridimensional. Estas mallas están compuestas típicamente por elementos 2D, como triángulos o cuadriláteros, que aproximan la superficie de un objeto tridimensional sin discretizar su volumen interno.

Las mallas superficiales surgen en una amplia variedad de contextos, desde el diseño asistido por computadora (CAD) y la manufactura digital, hasta la animación y los videojuegos. En biomecánica, por ejemplo, permiten representar tejidos, órganos o estructuras óseas a partir de escaneos médicos. En ingeniería, sirven como entrada para simulaciones de contacto, análisis aerodinámico o propagación de ondas en medios con contornos complejos. En el ámbito académico y científico, permiten el estudio de geometrías complejas que no pueden representarse de manera analítica.

Una propiedad deseable de las mallas superficiales es la capacidad de representar no solo la forma, sino también la orientación geométrica de cada una de sus partes. Aquí es donde entra en juego el cálculo de normales, tanto a nivel de caras como de vértices:

- Las normales de caras son esenciales para definir la dirección perpendicular a cada cara, lo que influye directamente en cómo interactúa la superficie con fuentes de luz, fuerzas físicas u otras geometrías.
- Las normales de vértices, en cambio, no se definen directamente a partir de una sola entidad geométrica, sino como una media ponderada de las normales de las caras adyacentes. La normal de vértices suele utilizarse en métodos de suavizado al renderizar iluminación en representaciones gráficas (sombreado Phong o Gouraud), pero también en simulaciones físicas donde una transición suave entre regiones es deseable.

En visualización 3D, las normales determinan cómo se proyecta la luz sobre una superficie, afectando sombras, reflejos y textura visual. En simulaciones físicas, las normales orientan fuerzas como presión, viento o contacto mecánico.

En particular, esta memoria se centrará en el generador de mallas propuesto en [Díaz et al., 2023], al cual nos referiremos como "mesher". Entre sus funciones relevantes, se encuentra que actualmente permite generar mallas volumétricas de elementos mixtos en distintos formatos de archivo de salida. Sin embargo, si uno deseara contar con datos de la superficie, como puede ser la identificación de las caras superficiales o el cómputo de normales asociadas, se debe depender de software externo a la implementación. En vista de esto, esta memoria propone la implementación de funcionalidades que permitan contar con esta información.

1.2. Objetivos

El objetivo principal de esta memoria es incorporar una implementación al generador de mallas geométricas tipo Octree que le permita procesar y añadir la información deseada al archivo de salida. Dentro de este objetivo se pueden identificar los siguientes objetivos específicos o hitos:

- Diseñar e implementar un algoritmo que permita identificar las caras pertenecientes a la superficie de las mallas generadas.
- Implementar un algoritmo que permita computar las normales de las caras pertenecientes a la superficie de las mallas generadas.

- Implementar un algoritmo que permita computar las normales de los vértices pertenecientes a la superficie de las mallas generadas.

CAPÍTULO 2

MARCO CONCEPTUAL

Como se ha mencionado ya, para el alcance de esta memoria, resulta importante conocer y entender la historia y los procesos que comprenden al generador de mallas sobre el cual se trabajará, además de entender el formato de salida del mismo y las definiciones de los procesos a realizar sobre los datos. A continuación, se comenzará introduciendo dos conceptos clave dentro de los algoritmos generadores de mallas basados en *Quadtree/Octree* el refinamiento y balanceo.

2.1. Refinamiento

Dentro de una malla, el concepto de refinar hace referencia a la subdivisión de una región en particular de dicha malla con el fin de permitir que esta se ajuste de forma más precisa a la forma a representar. Por lo tanto, en la mayoría de los casos, al generar una malla se comienza a partir de un cuadrado en el caso 2D o un cubo en el caso 3D, el cual se encuentra estructurado en un sistema de coordenadas y contiene el *input* del objeto dentro de sí mismo. El cual es descrito por una secuencia de puntos conectados por segmentos rectos llamado *Polyline*.

Dicho cuadrante/octante será dividido en cuatro/ocho regiones iguales, las cuales posteriormente se dividirán también de forma sucesiva aquellas que intercepten a la superficie, hasta alcanzar el criterio de precisión deseado, descartando las regiones que no contengan parte del objeto. A partir de ahí, es posible definir el concepto de RL (*Refinement Level*) como la cantidad de divisiones que ha dado origen a una región en particular, siendo las regiones iniciales todas RL igual a cero.

El resultado del proceso de refinamiento será una *Grid Mesh* tal como se muestra en la figura [1](#)

2.2. Balanceo

Considerando el resultado del refinamiento de una malla y bajo la necesidad de obtener una malla congruente y con una transición suave entre Cuadrantes o Octantes de distintos niveles de refinamiento, se establecen una serie de patrones que permitirán modificar la conectividad de la malla [Jaillet y Lobos, 2022]. Sin embargo con la finalidad de limitar la gran cantidad de patrones requeridos para las distintas combinaciones de diferentes RL, se establece como una restricción interna el limitar las diferencias de RL en la malla a no mas de 1 nivel de diferencia. Para lograr este estado, llamado balanceado, se lleva a cabo el proceso de balancear la malla, el cual consiste en simplemente refinar aquellos cuadrantes/octantes que producto de un refinamiento ahora se encuentren en incumplimiento la regla. [Yerry y Shephard, 1983].

Dentro de la figura 1.f, es posible observar un estado no balanceado, en concreto, las regiones rojas no cumplen balance con las azules ya que existe una diferencia de RL igual a dos, por ende, para asegurar el balance se debe refinar las regiones mayores, logrando el estado de la figura 1.g. Además del proceso de balanceo se desprende que para poder realizar el balanceo de una región, es necesario poder obtener información del RL de las regiones vecinas, este dato es importante, ya que es uno de los criterios que más adelante las estructuras de datos deberán poder cumplir y es crucial en determinar la eficiencia de las mismas.

Puede ser el caso que al momento de balancear se produzcan nuevos conflictos o focos de desbalanceo, esa posibilidad debe ser abarcada por el algoritmo de balanceo, afortunadamente, este ejemplo no es el caso.

Adicionalmente un detalle a considerar son los resultados de [Jaillet y Lobos, 2022], los cuales indican que atender desbalanceos apenas se producen durante el refinamiento es más eficiente que resolverlos una vez refinada la malla por completo.

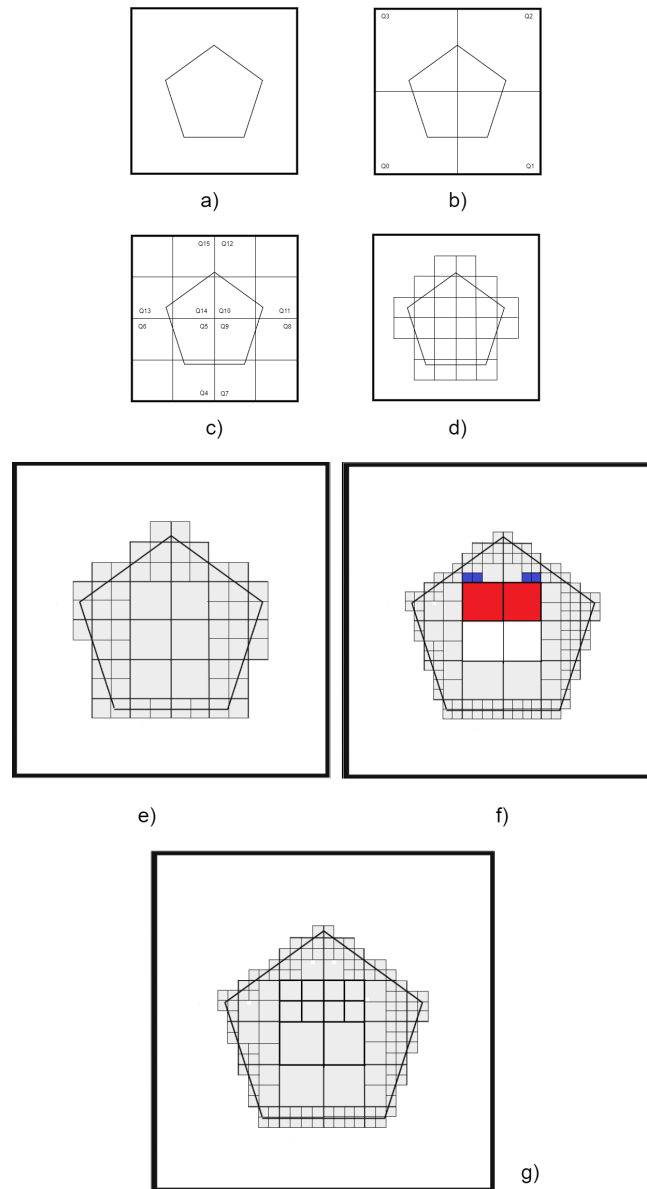


Figura 1: Ejemplo de refinamiento de un pentágono regular.

Fuente: Autoría propia.

2.3. Estado del arte (Estructuras de datos)

Hasta ahora se ha hablado de cómo generar una malla balanceada, pues a continuación se presentarán algunas estructuras de datos propuestas a la fecha que soportan la representación de estas mallas y los procesos de refinamiento y balanceo.

2.3.1. Estructura basada en árbol (*QuadTree/Octree*)

Como su nombre indica, la estructura de datos *QuadTree* consiste en representar una malla mediante un árbol cuaternario, es decir, que cada nodo que no sea una hoja deberá tener exactamente cuatro hijos, esto representa la relación que tiene un cuadrante cuando se divide en cuartos. En la figura 2 es posible observar un ejemplo de una *Grid Mesh* de un círculo, además de su respectiva representación como árbol (parcial) y el arreglo de almacenamiento, donde cero representa región vacía, uno representa que la región parcialmente contiene la figura y dos representa que la región se encuentra llena de la figura.

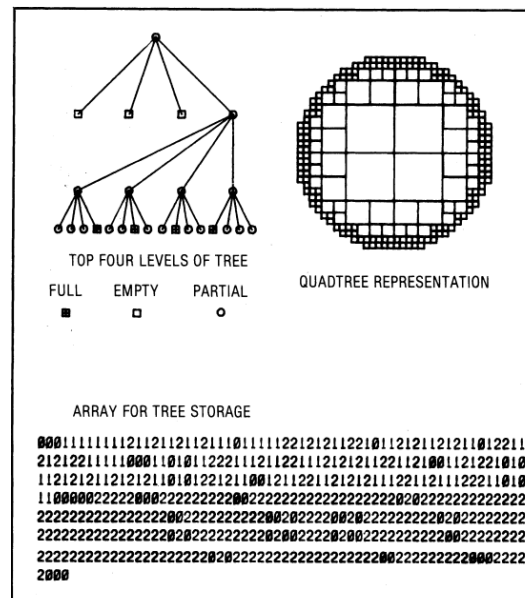


Figura 2: Ejemplo de QuadTree.

Fuente: [Yerry y Shephard, 1983].

Esta representación soporta el proceso de refinamiento y permite también verificar y solucionar problemas de balanceo mediante acceso a vecinos. Sin embargo cabe destacar que acceder a estos aumenta en complejidad a medida que el árbol se vuelve más profundo, es decir, a mayores niveles de refinamiento, su expansión al caso 3D es un árbol octal sin mayores diferencias.

2.3.2. Estructura basada en arcos (*Edge Based Data Structure*)

La estructura propuesta por [Jaillet y Lobos, 2022], plantea una alternativa que presenta beneficios especialmente a la hora de facilitar el acceso a información de regiones vecinas. La forma en que se logra esto es almacenando información de los arcos dentro de un *Map*, en donde la *Key* de cada arco es una tupla que contiene los nodos que definen el arco, y cada *Value* consistirá en un *array* que contendrá tres datos relevantes. El primero será el identificador de un nodo central (en el caso de no existir será cero); en segundo lugar, contiene un identificador del cuadrante izquierdo o superior adyacente (ya sea un arco horizontal o vertical); y el tercer dato será el identificador del cuadrante derecho o inferior adyacente. En el caso de que un arco forme parte de un borde, dependiendo de qué tipo de borde, una de sus regiones será identificada como el máximo valor de identificación posible.

Arreglo de enteros sin signo		
Nodo Central	Cuadrante Vecino ($\downarrow$ / $\rightarrow$)	Cuadrante Vecino ($\uparrow$ / $\leftarrow$)

Tabla 1: Estructura de arcos.

Fuente: Adaptada de [Díaz et al., 2023].

Para ejemplificar la forma en que esta estructura guarda la información relevante, se presenta dentro de la figura 3a algunos ejemplos de arreglos para los arcos de una región sin dividir, por ende sus arcos no presentan nodo medio. Por otro lado, la figura 3b presenta el mismo cuadrante una vez refinado, y nos muestra cómo se definen los sub-arcos además de las modificaciones que sufre el arco completo al crearse un nuevo nodo intermedio.

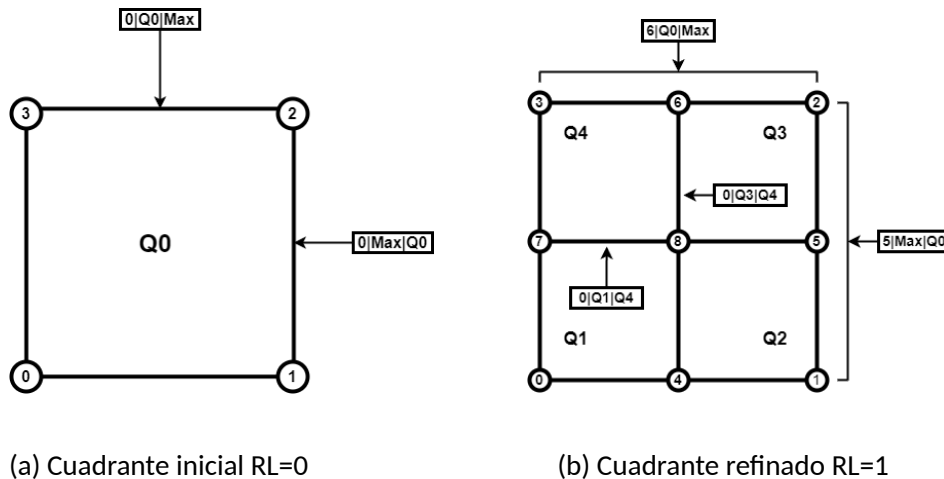


Figura 3: Ejemplos de estructura de datos basada en arcos

Fuente: Autoría Propia

2.3.3. Estructura basada en nodos (*Node Based Data Structure*)

Como se ha mencionado, el acceso rápido y eficiente a regiones vecinas es prioritario ya que es una operación frecuentemente empleada en los procesos de refinamiento y balanceo. Con esto en mente [Díaz et al., 2023] propone una estructura que asegura acceso constante a regiones vecinas mediante vectores que guardan información sobre los distintos nodos que corresponden a los vértices de los cuadrantes de la malla. Por ende, cada nodo considerará en su estructura un identificador de sus nodos vecinos, las regiones vecinas que compartan el nodo y finalmente sus respectivos RL.

Arreglo de enteros sin signo		
Nodos Vecinos	Cuadrantes Vecinos	RL de Cuadrantes

Tabla 2: Estructura de nodos.

Fuente: Adaptada de [Díaz et al., 2023].

Respecto al proceso de refinamiento aplicado a esta estructura, consiste en la creación de cinco nuevos nodos: el nodo central y los cuatro nodos intermedios. Estos deben asignarse entre sí, además de considerar la relación con los nodos vértice del cuadrante. Adicionalmente, se debe tomar en consideración que, una vez que se requiere crear un nuevo nodo entre un arco, existe la posibilidad de que dicho nodo ya exista debido al proceso de refinamiento del cuadrante adyacente. Si esto no se considerara, se generaría un conflicto en el cual haya un duplicado de un mismo nodo.

2.4. Patrones de transición

La aplicación de patrones de transición consiste en la modificación de la estructura de un cuadrante/octante con la finalidad de obtener una malla más congruente. Estos cambios en estructura no son aleatorios, se encuentran definidos y se aplican en función de cuál sea más adecuado. Para determinar cuál debe ser utilizado, es importante la configuración de nodos medios en los arcos de la región en específico, es decir, que las celdas que posean al menos un nodo medio en sus arcos se someterán a un patrón de transición.

Los patrones de transición para los casos 2D y 3D son descritos en [Lobos, 2015]. En la figura 4 se presentan los patrones 2D según las distintas configuraciones de nodos medios en sus arcos, por extensión de esta definición al caso 3D, la cantidad de combinaciones de nodos medios en las aristas del octante aumenta la cantidad de patrones de transición. De ello, en [Lobos, 2015] se identifican y describen 325 patrones; de ahí mismo se define que

estos patrones dan como resultado una combinación de elementos básicos, los cuales son presentados en la figura 5. Esta información resulta importante, ya que define el tipo de celdas presentes en la salida del generador de mallas y, por ende, la forma de extracción de caras. Además, define el orden de los vértices que constituyen cada cara.

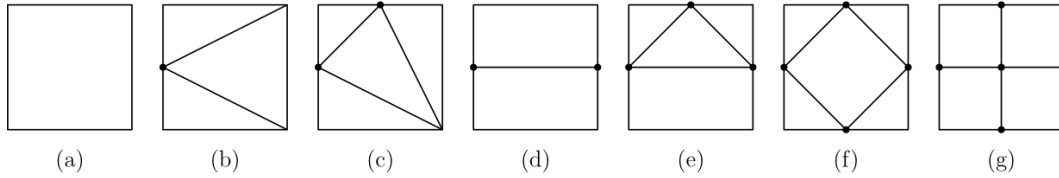


Figura 4: Patrones de transición 2D

Fuente: [Lobos, 2015]

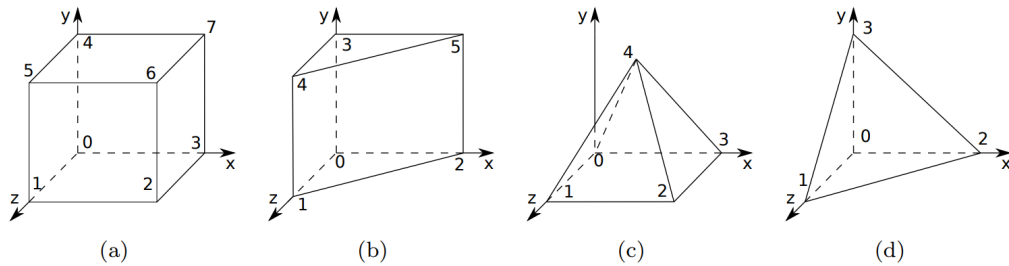


Figura 5: Elementos básicos: hexaedro (a), prisma triangular (b), pirámide cuadrada (c) y tetraedro (d)

Fuente: [Lobos, 2015]

2.5. *Adjust to boundary* y patrones de superficie

Otro proceso aplicado a la malla descrito en [Jaillet y Lobos, 2022] y a ser considerado implica el desplazamiento de nodos y la modificación de celdas con la finalidad de lograr un mejor ajuste de la malla a los límites de la superficie a representar. Para esto en primer lugar se detectan las celdas que contengan una "feature". Estando esta definida como un punto de la superficie cuyos segmentos compuestos por dicho punto crean un ángulo fuera de cierto intervalo de tolerancia. En [Jaillet y Lobos, 2022] dicho intervalo es definido en $]175^\circ, 185^\circ[$. En esencia estos puntos son considerados como relevantes al momento de definir la superficie. Una vez detectada una *feature*, la celda que la contenga verá desplazado su nodo mas cercano a la *feature* para ajustarse a la misma. Posteriormente se realiza un proceso denominado *shrink to boundary*, el cual consiste en detectar cualquier nodo que se encuentre fuera de la superficie y proyectarlo a la misma para ajustarlo. Finalmente se obtiene una malla que representa el dominio de entrada. Sin embargo la calidad de las celdas

de la superficie es baja, por ello se realiza un ultimo proceso en el cual se determinan dichas celdas y nodos son eliminados si estos nodos aportan poco valor a la malla.

Esta serie de procesos resultan relevantes para el alcance de esta memoria, ya que propician la posibilidad de que al ser modificados los elementos, surjan caras cuadriláteras no planas. Esto determina una problemática a considerar al momento de seleccionar un método para calcular sus normales. El método a elegir debe ser robusto y capaz de brindar un vector normal que caracterice lo mejor posible la orientación media de la cara, ya que métodos como el producto cruz de dos vectores de vértices no esta condicionado para caras no planas.

2.6. VTK (*The Visualization ToolKit*)

VTK es un *software* para manipular y visualizar datos científicos, ofrece herramientas de renderizado 3D y interacción[Kitware, a]. En particular nos resulta de valor profundizar en el formato de archivos .vtk [Kitware, b] y en como es utilizado para almacenar la información de salida del *mesher*.

En el listado 1 se presenta un ejemplo de la estructura empleada por un archivo de salida .vtk del *mesher*. El cual inicialmente incluye información relevante como: el tipo de archivo, la versión del mismo, el tipo de malla y la codificación utilizada. Para luego proceder con la escritura de los datos, se declara la escritura de los puntos, la cantidad N de estos y el tipo de dato utilizado para las 3 coordenadas espaciales del vértice. Luego se incluye la escritura de las M celdas las cuales suman un total de T vértices, cada celda comienza con el tamaño de la misma descrito por los k_i valores. Finalmente se escriben los M identificadores de tipo de celdas que se utilizan. Los códigos de tipos de celdas que pueden resultar de este *mesher* son: tetraedros(10), hexaedro(12), prisma triangular(13) y pirámide(14).

```
# vtk DataFile Version X.X
Unstructured Grid FileName
ASCII
```

```
DATASET UNSTRUCTURED_GRID
```

```
Points  $N$  float
```

```
 $x_0$   $y_0$   $z_0$ 
```

```
 $x_1$   $y_1$   $z_1$ 
```

```
 $\vdots$ 
```

```
 $x_{N-1}$   $y_{N-1}$   $z_{N-1}$ 
```

```
CELLS  $M$   $T$ 
```

```
 $k_0$   $i_{0,0}$   $i_{0,1}$   $\dots$   $i_{0,k_0-1}$ 
```

```
 $\vdots$ 
```

```
 $k_{M-1}$   $i_{M-1,0}$   $\dots$   $i_{M-1,k_{M-1}-1}$ 
```

```
CELL_TYPES  $M$ 
```

```
 $t_0$ 
```

```
 $\vdots$ 
```

```
 $t_{M-1}$ 
```

Listing 1: Estructura de salida en formato .vtk.

Fuente: Autoría propia

2.7. Normal de una cara

La normal de una cara (o normal de superficie) es un vector tridimensional que es perpendicular (ortogonal) a la superficie de esa cara en un punto dado. Para caras planas como triángulos o polígonos planos, la normal es la misma en toda la cara. La normal brinda información respecto a la orientación de la cara y posteriormente nos permitirá calcular la normal de los vértices de la superficie. Dentro de las caras superficiales que se encuentran en los elementos resultantes del *mesher* se tienen triángulos (siempre planos), y cuadriláteros. Estos últimos pueden no ser planos producto de desplazamientos en procesos tales como los descritos en la [Subsección 2.5](#). Conociendo esto es posible determinar los algoritmos mas adecuados para el calculo de normales de ambos tipos de polígonos presentes en nuestras caras.

2.7.1. Normal de polígono triangular

La normal $\vec{N}$ de un triángulo definido por los puntos P_0, P_1, P_2 está definida como el producto cruz de dos vectores $\vec{U}$ y $\vec{V}$ aristas de dicho triángulo definidos como:

$$P_0 = (x_0, y_0, z_0)$$

$$P_1 = (x_1, y_1, z_1)$$

$$P_2 = (x_2, y_2, z_2)$$

$$\vec{U} = P_1 - P_0 = (x_1 - x_0, y_1 - y_0, z_1 - z_0) = (u_x, u_y, u_z)$$

$$\vec{V} = P_2 - P_0 = (x_2 - x_0, y_2 - y_0, z_2 - z_0) = (v_x, v_y, v_z)$$

$$\vec{N} = \vec{U} \times \vec{V}$$

por lo tanto las componentes (N_x, N_y, N_z) del vector $\vec{N}$ están definidas de la siguiente manera:

$$N_x = u_y v_z - u_z v_y$$

$$N_y = u_z v_x - u_x v_z$$

$$N_z = u_x v_y - u_y v_x$$

Adicionalmente se desea obtener la normal como vector unitario $\hat{n}$. Para esto es necesario normalizar el vector $\vec{N}$ tal que $\hat{n} = \frac{\vec{N}}{\|\vec{N}\|}$, por lo tanto $\hat{n}$ se obtiene al dividir cada componente de $\vec{N}$ por la magnitud del mismo:

$$\|\vec{N}\| = \sqrt{N_x^2 + N_y^2 + N_z^2}$$

$$\hat{n} = \left(\frac{N_x}{\|\vec{N}\|}, \frac{N_y}{\|\vec{N}\|}, \frac{N_z}{\|\vec{N}\|} \right)$$

2.7.2. Normal de un cuadrilátero

Para calcular la normal de cuadriláteros, los cuales como se ha mencionado anteriormente, no son con seguridad planos, es necesario emplear un método robusto y que entregue una “normal media” para este tipo de polígono. El método a emplear es el método de *Newell* descrito en [Tampieri, 1992].

Para un polígono definido por sus n vértices $V_i = (x_i, y_i, z_i)$. El vector normal $\vec{N}$ se encuentra definido por la sumatoria del producto cruz de sus vértices $\vec{N} = \sum_{i=1}^n (V_i \times V_{i+1})$. Considerando a los vértices de forma cíclica el vértice $V_{n+1} = V_1$, al expandir el producto cruz obtenemos la definición de cada componente de $\vec{N}$ como:

$$\begin{aligned} N_x &= \sum_{i=1}^n (y_i - y_{i+1})(z_i + z_{i+1}) \\ N_y &= \sum_{i=1}^n (z_i - z_{i+1})(x_i + x_{i+1}) \\ N_z &= \sum_{i=1}^n (x_i - x_{i+1})(y_i + y_{i+1}) \end{aligned}$$

Finalmente se normaliza el vector $\vec{N}$ resultante de forma homologa a la utilizada para la normal de un triángulo.

2.8. Normal de un vértice

El concepto de normal de un vértice fue introducido en [Gouraud, 1971]. En el cual se plantea asignar a cada vértice de una superficie un vector que permita calcular transiciones continuas para la simulación de iluminación. Este vector inicialmente fue descrito como el promedio de los vectores normales de cada polígono que contenga dicho vértice. En [Jin et al., 2005] se presenta una recopilación y comparación entre múltiples propuestas de normales de vértices, las cuales implementan distintas formas de ponderar el peso de cada normal de un polígono. Sin embargo para el alcance de esta memoria nuestro foco se centra en el calculo mediante un promedio simple.

Para un vértice V el cual se encuentra presente en los m polígonos $K_1, K_2, \dots, K_m$, cuyas normales unitarias se encuentran definidas como $\hat{n}_{K_i}$, se define la normal $\vec{N}_V$ como:

$$\vec{N}_V = \sum_{i=1}^m \hat{n}_{K_i}, \quad \hat{n}_V = \frac{\vec{N}_V}{|\vec{N}_V|}$$

CAPÍTULO 3

PROPUESTA DE SOLUCIÓN

Con la finalidad de alcanzar los objetivos propuestos en esta memoria, se propuso incorporar dentro del mesher la capacidad de procesar los datos de la malla para identificar la superficie de la misma y computar valores que representen características de ella como pueden ser las normales de la superficie y sus vértices. Para lograr esto, se añadió una serie de funciones, en la siguiente sección se describirá a detalle la forma en que esta implementación aborda la problemática.

3.1. Extracción de caras

Como ya se ha mencionado en la sección [Subsección 2.4](#), la salida del mesher guarda en su estructura información de cada elemento 3D que conforma la malla, en forma de un arreglo con la id de los vértices. Sin embargo para la computación futura de datos requeridos, es necesario realizar un re formato de esta información. Se requiere contar con la información de todas las caras que componen la malla, esta información se encuentra implícita en el orden de vértices de cada elemento, y sigue una estructura regular. Afortunadamente se puede determinar el tipo de elemento mediante la cantidad de vértices, ya que todos los tipos cuentan con cantidades distintas. A continuación se explicará a detalle como se determina que vértices constituyen cada cara para cada tipo de elemento.

3.1.1. Hexaedro

Una vez se determina que un elemento es un hexaedro es necesario identificar los vértices que constituyen sus caras, para esto hay que tener en consideración la relación entre los vértices del hexaedro y el índice en el conjunto de vértices que los almacena. En la figura 6 se presenta la denominación de los vértices, junto a la planilla de construcción del hexaedro. Ahora que podemos visualizar cada cara desde el exterior nos es posible identificar las secuencias de vértices que construyen a las mismas. Al asegurarnos que el sentido de recorrido de los vértices sea anti-horario, podemos establecer la orientación de la cara que estamos extrayendo, las caras identificadas en este proceso se encuentran en la tabla 3

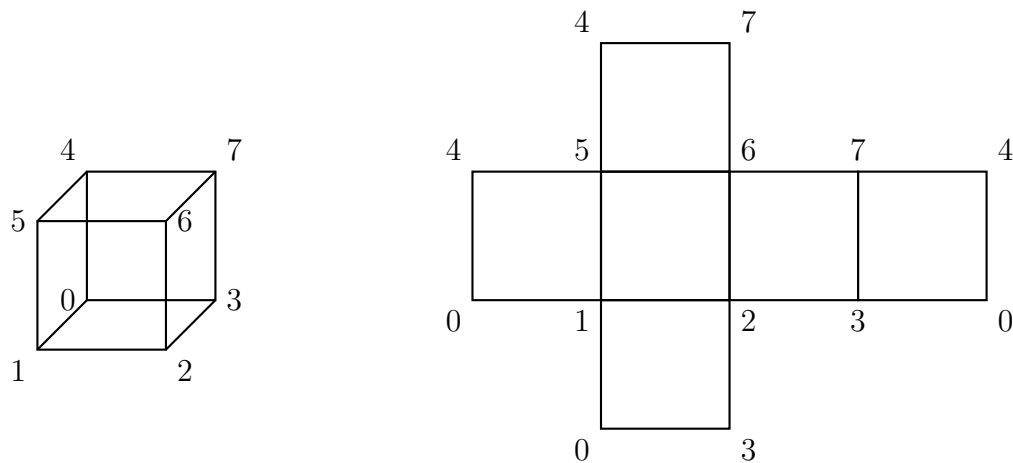


Figura 6: Ejemplo de elemento tipo hexaedro

Fuente: Autoría propia

Tabla 3: Índices de vértices hexaedro

Cara	Índices de Vértices
Cara Inferior	0, 3, 2, 1
Cara Frontal	1, 2, 6, 5
Cara Superior	5, 6, 7, 4
Cara Trasera	3, 0, 4, 7
Cara Izquierda	0, 1, 5, 4
Cara Derecha	2, 3, 7, 6

3.1.2. Tetraedro

De forma homologa al proceso empleado para el tetraedro, a partir de la figura 7 podemos interpretar las 4 caras triangulares que forman al elemento, resultando en el conjunto de caras descrito en la tabla 4.

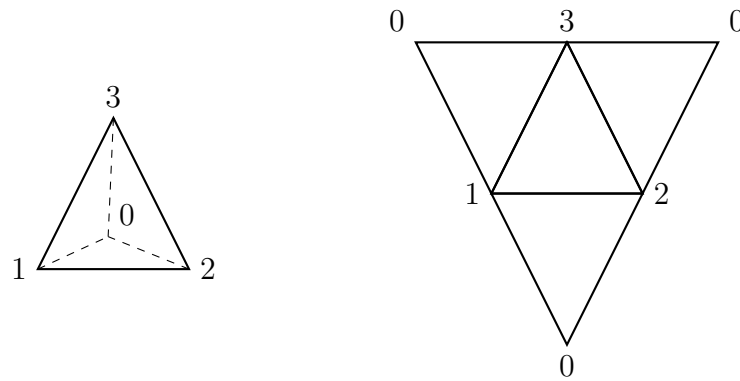


Figura 7: Ejemplo de elemento tipo tetraedro

Fuente: Autoría propia

Tabla 4: Índices de vértices tetraedro

Cara	Índices de Vértices
Cara Central	1, 2, 3
Cara Izquierda	1, 3, 0
Cara Derecha	2, 0, 3
Cara Inferior	0, 2, 1

3.1.3. Pirámide Cuadrada

Para la pirámide de base cuadrada y a partir de la figura 8 podemos interpretar las 4 caras triangulares y 1 cara cuadrilátera que forman al elemento, resultando en el conjunto de caras descrito en la tabla 5.

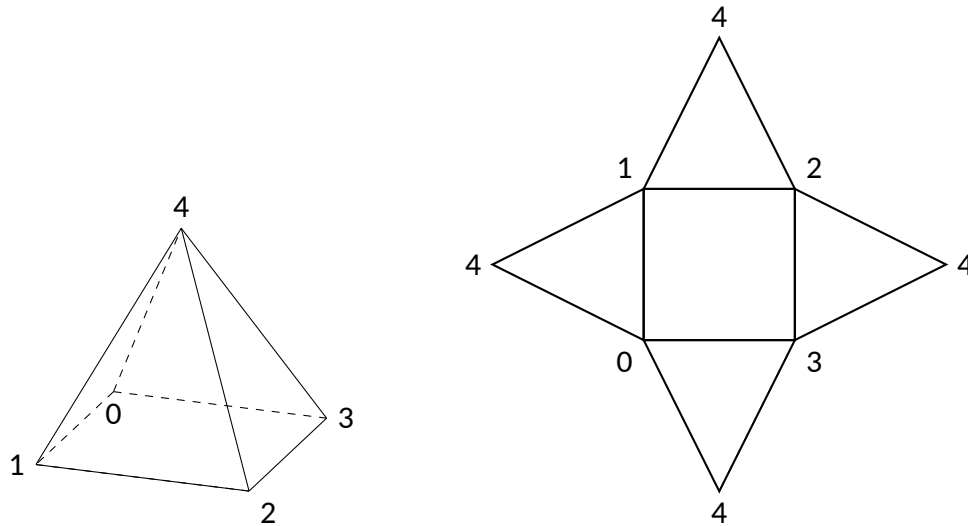


Figura 8: Ejemplo de elemento tipo pirámide cuadrada

Fuente: Autoría propia

Tabla 5: Índices de vértices pirámide cuadrada

Cara	Índices de Vértices
Cara Central	0, 3, 2, 1
Cara Izquierda	0, 1, 4
Cara Derecha	3, 4, 2
Cara Superior	1, 2, 4
Cara Inferior	0, 4, 3

3.1.4. Prisma triangular

Para prisma de base triangular y a partir de la figura 9 podemos interpretar las 2 caras triangulares y 3 caras cuadriláteras que forman al elemento, resultando en el conjunto de caras descrito en la tabla 6.

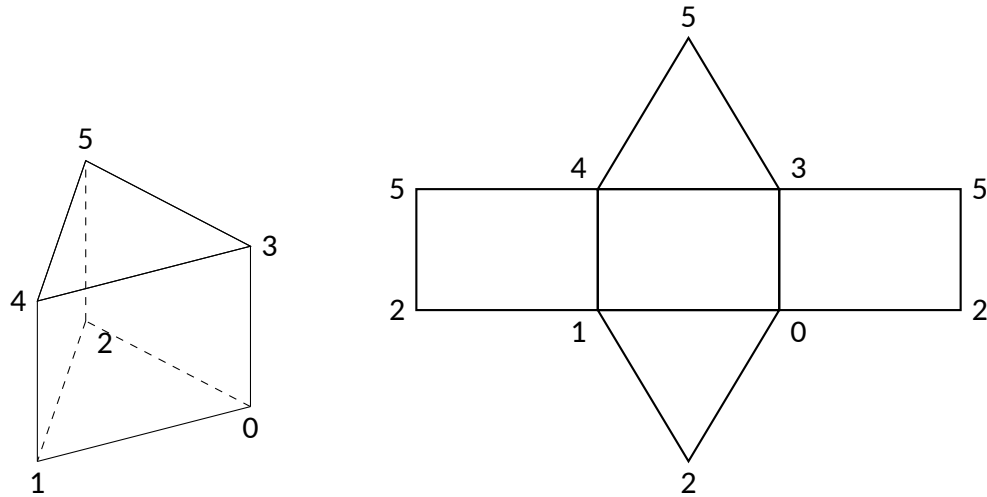


Figura 9: Ejemplo de elemento tipo prisma triangular

Fuente: Autoría propia

Tabla 6: Índices de vértices prisma triangular

Cara	Índices de Vértices
Cara Central	1, 0, 3, 4
Cara Izquierda	2, 1, 4, 5
Cara Derecha	0, 2, 5, 3
Cara Superior	4, 3, 5
Cara Inferior	2, 0, 1

Ahora que se ha propuesto una forma de extraer las caras de cada elemento, es momento de abordar una problemática. Debido a la adyacencia de los elementos en la malla, al extraer elemento por elemento las caras, es seguro que nos encontraremos con caras duplicadas, para ser mas precisos, cada cara interna de la malla, es compartida por exactamente 2 elementos. Dado a que no queremos duplicados en nuestro listado, cada vez que una cara sea extraída, guardaremos información de su frecuencia de aparición en una estructura tipo *map*. Cabe considerar que como las caras duplicadas no necesariamente se encuentran en el mismo orden es necesario ordenar sus vértices de mayor a menor para permitir su comparación, de esta manera podemos contabilizar las apariciones de cada cara independiente de el orden en que aparezcan y así unicamente añadir la primera aparición de cada cara a nuestro listado de caras. Convenientemente esta estructura nos permitirá también identificar caras superficiales mas adelante ya que estas tienen la característica de solo pertenecer a un elemento

3.2. Caras superficiales y sus normales

Con el fin de identificar las caras superficiales de la malla y computar las normales de cada una de dichas caras, se propone el uso de una estructura tipo *map* para contabilizar la frecuencia de caras, es decir la cantidad de veces que en el proceso de extracción dicha cara fue extraída independiente de su orden, esto nos permitirá una vez extraídas todas las caras identificar aquellas que pertenecen a la superficie siendo aquellas que tengan unicamente una aparición. Sin embargo como el *map* almacena como llave un *string* ordenado de los vértices de la cara, y deseamos evitar volver a recorrer cada cara, se construye un *map* adicional que permite relacionar el *string* ordenado de vértices de una cara con el orden original de la primera vez que dicha cara fue encontrada. De esta manera podemos iterar sobre el primer *map* identificando aquellas con valor de apariciones igual a 1 y con la llave recuperar en el segundo *map* la cara y su orden original para luego almacenarlo en una lista de caras superficiales.

Adicionalmente, una vez que una cara es identificada como superficial, podemos inmediatamente calcular su normal, la cual dependiendo de si se trata de un triangulo o cuadrilátero se empleara producto cruz o el método de Newell, como puede verse en las líneas [43-46] del algoritmo 1, cuyas implementaciones a detalle se encuentran descritas en los algoritmos 2 y 3. En el caso del producto cruz, la implementación es simple y altamente conocida, consiste en computar 2 vectores a partir de 3 puntos pertenecientes a la cara y mediante ellos operar el producto cruz, lo que producirá el vector normal el cual posteriormente es normalizado al dividir cada componente por la magnitud del vector resultante. En el caso del método de Newell, se itera cada punto y se determina las contribuciones al vector normal realizadas por el lado del cuadrilátero formado por el punto

actual y el siguiente.

3.3. Normal del vértice

Ahora que ya poseemos identificada tanto las caras de la superficie como computada sus normales, podemos continuar con el calculo de las normales de cada vértice perteneciente a la superficie, como fue descrito en la [Subsección 2.8](#), cada normal de un vértice esta compuesta por la sumatoria de las normales de las caras que contienen a dicho vértice. Sin embargo, en nuestro caso unicamente poseemos esa relación en el sentido opuesto, conocemos los vértices que pertenecen a cada cara. Así que gracias a la manera acumulativa que se obtiene la normal del vértice, en el algoritmo [4](#), se recorre cada cara y su normal se suma a la acumulación de normales de cada vértice que pertenezca a dicha cara. De esa forma, una vez recorrida cada cara, todos los vértices contarán con la sumatoria completa de las caras que requieren para calcular su propia normal, posteriormente se normaliza el resultado para obtener un vector unitario.

Algorithm 1: Extracción de caras superficiales y calculo de normales.

```

1 Face_Counter = MAP // Key{Cara ordenada como string}, Value{Contador de frecuencia}
2 Original_Faces = MAP // Key{Cara ordenada como string}, Value{Cara en el orden de su primera
   aparición} (los valores equivalen a la lista de caras sin repetición)
3 Surface_Faces = Empty List
4 Surface_Normals = Empty List
5 for each element in elements do
6     np = element.size
7     Element_Faces = Empty List
8     if np = 4 then
9         // Tetrahedro
10        Element_Faces.ADD (element[0], element[1], element[3])
11        Element_Faces.ADD (element[2], element[0], element[3])
12        Element_Faces.ADD (element[2], element[3], element[1])
13        Element_Faces.ADD (element[0], element[2], element[1])
14    else if np = 5 then
15        // Pirámide Cuadrada
16        Element_Faces.ADD (element[0], element[1], element[4])
17        Element_Faces.ADD (element[3], element[0], element[4])
18        Element_Faces.ADD (element[2], element[3], element[4])
19        Element_Faces.ADD (element[1], element[2], element[4])
20        Element_Faces.ADD (element[0], element[3], element[2], element[1])
21    else if np = 6 then
22        // Prisma triangular
23        Element_Faces.ADD (element[0], element[1], element[2])
24        Element_Faces.ADD (element[3], element[5], element[4])
25        Element_Faces.ADD (element[3], element[0], element[2], element[5])
26        Element_Faces.ADD (element[1], element[4], element[5], element[2])
27        Element_Faces.ADD (element[0], element[3], element[4], element[1])
28    else if np = 8 then
29        // Hexaedro
30        Element_Faces.ADD (element[0], element[1], element[5], element[4])
31        Element_Faces.ADD (element[1], element[2], element[6], element[5])
32        Element_Faces.ADD (element[2], element[3], element[7], element[6])
33        Element_Faces.ADD (element[3], element[0], element[4], element[7])
34        Element_Faces.ADD (element[2], element[1], element[0], element[3])
35        Element_Faces.ADD (element[4], element[5], element[6], element[7])
36    for each face in Element_Faces do
37        sorted_face = face.Ordenar()
38        sorted_face.String()
39        Face_Counter [key] = Face_Counter [key] + 1
40        if Face_Counter [key] = 1 then
41            Original_Faces [key] = face
42    for each (key, count) in Face_Counter do
43        if count = 1 then
44            face = Original_Faces [key]
45            Surface_Faces.ADD(face)
46            normal = Empty Vector
47            if face.size = 3 then
48                normal =ComputeTriangleNormal(face.vertices)
49            else
50                normal =ComputeQuadNormalNewell(face.vertices)
51            Surface_Normals.ADD(normal)

```

Algorithm 2: Cálculo de la normal de un triángulo**Entrada:** A,B,C: Vectores de vértices del triángulo**Salida :** normal: Vector normal unitario

```

1 computeTriangleNormal (A, B, C) begin
2   u = B[0]-A[0], B[1]-A[1], B[2]-A[2] // Vector de la arista A - B
3   v = C[0]-A[0], C[1]-A[1], C[2]-A[2] // Vector de la arista A - C
4   crossprod = { // Calcula el producto cruz
5       u[1]*v[2] - u[2]*v[1],
6       u[2]*v[0] - u[0]*v[2],
7       u[0]*v[1] - u[1]*v[0]
8   }
9   mag_sq = crossprod[0]*crossprod[0] + crossprod[1]*crossprod[1] + crossprod[2]*crossprod[2]
10  mag = sqrt(mag_sq)
11  return (crossprod[0]/mag, crossprod[1]/mag, crossprod[2]/mag)

```

Algorithm 3: Cálculo de la normal de un cuadrilátero usando el método de Newell**Entrada:** A, B, C, D: Vectores de vértices del cuadrilátero**Salida :** normal: Vector normal unitario

```

1 computeQuadNormalNewell (A, B, C, D) begin
2   normal = Vector(0.0,0.0,0.0)
3   points = A, B, C, D
4   n = points.size
5   for i= 0; i < n; ++i do
6       current = points[i]
7       next = points[(i + 1) % n]
8       normal [0] = normal [0] + (current[1]-next[1]) * (current[2]+next[2]);
9       normal [1] = normal [1] + (current[2]-next[2]) * (current[0]+next[0]);
10      normal [2] = normal [2] + (current[0]-next[0]) * (current[1]+next[1]);
11
12  mag_sq = normal[0]*normal[0] + normal[1]*normal[1] + normal[2]*normal[2]
13  mag = sqrt(mag_sq)
14  return (normal[0]/mag, normal[1]/mag, normal[2]/mag)

```

Algorithm 4: Calculo de normales de vértices superficiales

```

1 computeSurfacePointNormals (points, Surface_Faces, Surface_Normal) begin
2   num_points = points.size
3   point_normals = (num_points, Vector(0.0,0.0,0.0))
4   for each Face in Surface_Faces do
5       face_normal_to_add = Surface_Faces[Face.id]
6       for each Vertex in Face do
7           point_normals [Vertex][0] = point_normals [Vertex][0] + face_normal_to_add [0]
8           point_normals [Vertex][1] = point_normals [Vertex][1] + face_normal_to_add [1]
9           point_normals [Vertex][2] = point_normals [Vertex][2] + face_normal_to_add [2];
10
11  for each Normal in point_normals do
12      mag_sq = Normal[0]*Normal[0] + Normal[1]*Normal[1] + Normal[2]*Normal[2]
13      mag = sqrt(mag_sq)
14      Normal[0] = Normal[0] / mag
15      Normal[1] = Normal[1] / mag
16      Normal[2] = Normal[2] / mag
17
18  return point_normals

```

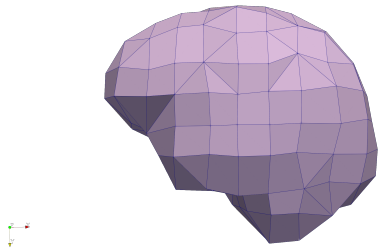
CAPÍTULO 4

VALIDACIÓN DE LA SOLUCIÓN

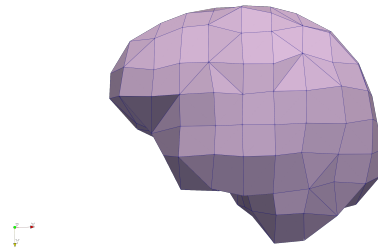
Con la finalidad de validar la implementación propuesta en esta memoria, es importante entender que tipo de resultados han sido obtenidos. Esta implementación principalmente se ha basado en procesar datos y computar información adicional a partir de ellos tales como el computo de vectores normales, en esta memoria es el principal interés comprobar que dichos datos sean correctos, y la forma propuesta de hacerlo es emplear un método comparativo con resultados respaldados y ya conocidos. En este caso se compararán los resultados con los obtenidos mediante el software de visualización Paraview, el cual presenta herramientas para visualizar los datos obtenidos por la implementación como así obtenerlos mediante el mismo software para ofrecer un punto de comparación para los resultados de esta propuesta.

4.1. Extracción de caras

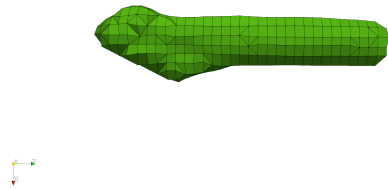
Para validar el proceso de extracción de caras, el cual consiste en identificar entre los elementos del output del mesher no solo la secuencia de vértices que constituyen cada cara, sino también cuáles de dichas caras pertenecen a la superficie de la malla, se propone como prueba escribir un nuevo archivo de salida en formato VTK para distintas figuras. Este archivo contiene como polígonos únicamente las caras superficiales identificadas por la implementación. Posteriormente, este archivo fue cargado en Paraview, y como punto de comparación se empleó el archivo de salida VTK original, al cual se le aplicó un filtro para extraer la superficie que proporciona Paraview. De esta manera, podemos comparar visualmente ambos resultados y obtener información relevante acerca de las superficies obtenidas mediante ambos métodos. En la figura 10 se observa una comparación entre tres figuras distintas cuyas mallas fueron generadas optando por un bajo nivel de refinamiento (RL 3), con tal de beneficiar la visibilidad de las caras. El número de caras para todas las figuras resultó coincidente y, como se puede apreciar, el resultado de la implementación no presenta errores visibles como agujeros, caras mal orientadas o mal definidas.



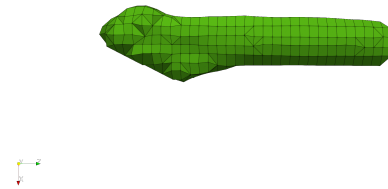
(a) Córtex con Paraview (267 caras)



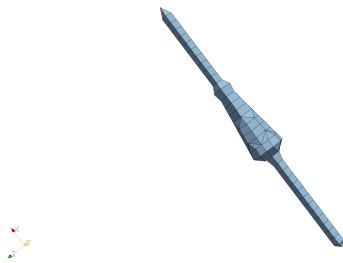
(b) Córtex de la implementación (267 caras)



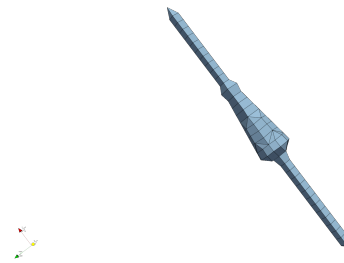
(c) Fémur con Paraview (533 caras)



(d) Fémur de la implementación (533 caras)



(e) Sutura con Paraview (196 caras)



(f) Sutura de la implementación (196 caras)

Figura 10: Comparación de caras superficiales.

4.2. Normales de caras y vértices superficiales

En el caso de la validación de los valores de normales obtenidos mediante la implementación, se propone nuevamente una comparativa con los resultados que puede ofrecer el software Paraview. se escribió un archivo VTK que contiene la información de las caras superficiales de la malla junto a los valores de normales de dichas caras y vértices superficiales, de manera similar se aplicaron filtros que permiten computar dichas normales sobre la malla superficial extraída por Paraview. El filtro fue aplicado con configuraciones por defecto a excepción de la desactivación de la ejecución de un proceso llamado *Splitting* el cual se encuentra fuera de los alcances de esta memoria. Posteriormente se aplicaron filtros de visualización tipo *Glyph*, los cuales permiten visualizar datos asociados a la malla, de esa manera es posible ver representado los vectores normales de ambos grupos de datos en forma de flechas.

En la figura 11 se presentan las comparaciones entre las normales de las caras para nuevamente tres figuras. Se puede apreciar que todas las caras visibles tienen un vector normal asociado correctamente, y es posible comparar a simple vista la dirección de estos vectores en contraste con los obtenidos por Paraview. De forma similar, en la figura 12 se presenta la comparación de las figuras, pero esta vez considerando las normales asociadas a cada vértice perteneciente a la superficie. Se pueden realizar las mismas observaciones que para las normales de caras: todas las normales de vértices visiblemente presentan un vector normal asociado, y las orientaciones de estas pueden ser contrastadas con las obtenidas por Paraview.

Si bien las pruebas ya presentadas pueden darnos una idea de la exactitud de los datos computados, se planteó la realización de una nueva prueba para obtener mayor seguridad sobre su validez. Esta prueba, mediante un *script*, permite comparar directamente los valores de normales obtenidos por la implementación y por Paraview. Para esto, se exportaron los datos de normales obtenidos por Paraview a un archivo en formato CSV. En el caso de los obtenidos mediante la implementación, estos fueron manejados mediante un archivo VTK, para luego leer y cargar los datos de ambos archivos mediante el mencionado *script* desarrollado en Python. El script itera sobre los puntos superficiales obtenidos por Paraview y busca el punto correspondiente en el archivo VTK. Luego compara cada una de las componentes del vector normal asociado a ambos grupos de datos.

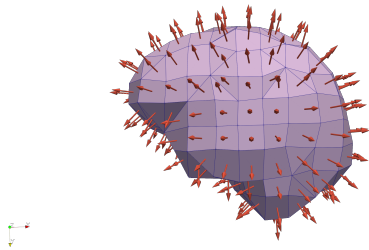
Al momento de comparar, se estableció una tolerancia de 10^{-6} para la diferencia absoluta de cada coordenada como margen de aceptación. Los datos sobre los cuales se realizó esta prueba corresponden a las tres figuras ya presentadas, para niveles de refinamiento RL3, RL5 y RL7, con el fin de representar el efecto de aumentar la precisión de la malla en el cómputo de normales. Los resultados se presentan en la tabla 7.

A partir de esta prueba, se determinó que los vectores normales de los vértices no presentan desviaciones en sus componentes (x, y, z) que superen la magnitud establecida de 10^{-6} . Una observación al respecto es que, al probar con una tolerancia más estricta, se pudo determinar que en ciertos casos existen diferencias en las componentes que varían en su magnitud. Estas pueden producirse por múltiples motivos, tales como pérdidas de precisión en el cálculo de normales y/o aproximaciones que ocurren en algún punto del proceso. Sin embargo, considerando que el vector normal es un vector unitario, es decir, de magnitud igual a uno, las variaciones observadas no representan un problema significativo y se consideran aceptables.

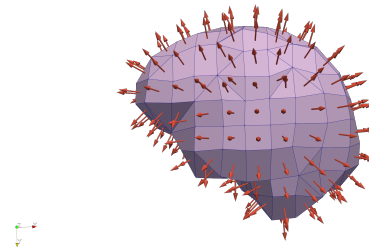
Adicionalmente, en la misma tabla es posible conocer el valor máximo de desviación angular presente en cada caso de prueba. Esto permite dimensionar el efecto del error en la dirección del vector normal. De los resultados de desviación angular es posible observar que, para cada caso de prueba, a medida que aumenta el nivel de refinamiento, el error angular máximo también aumenta. Sin embargo, este crecimiento es menor que la proporción de crecimiento en la cantidad de puntos, y en ninguno de los casos de prueba se superó una magnitud de $(10^{-2})^\circ$.

Tabla 7: Aceptación de normales para error máximo 10^{-6}

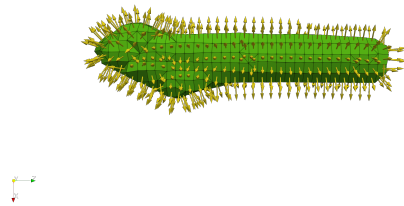
Prueba	Numero de normales	Normales aceptadas	Error angular máximo
Córtex RL3	188	188	0.023°
Córtex RL5	3372	3372	0.026°
Córtex RL7	55893	55893	0.028°
Fémur RL3	440	440	0.026°
Fémur RL5	8040	8040	0.027°
Fémur RL7	131476	131476	0.029°
Sutura RL3	174	174	0.019°
Sutura RL5	2794	2794	0.023°
Sutura RL7	51578	51578	0.027°



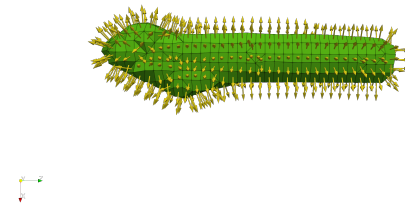
(a) Córtex Paraview



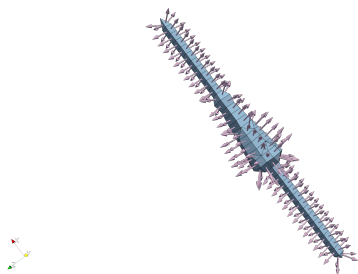
(b) Córtex de la implementación



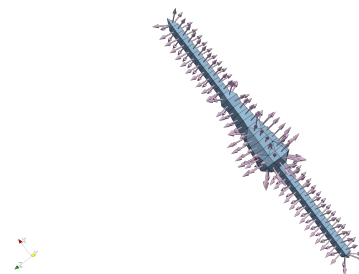
(c) Fémur Paraview



(d) Fémur de la implementación

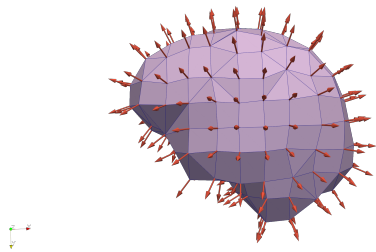


(e) Sutura con Paraview

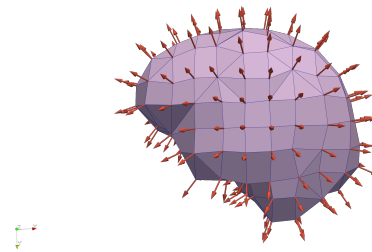


(f) Sutura de la implementación

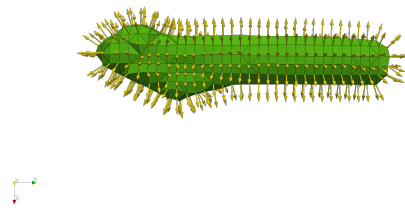
Figura 11: Comparación de normales de caras.



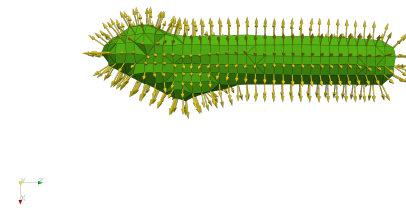
(a) Córtez con Paraview



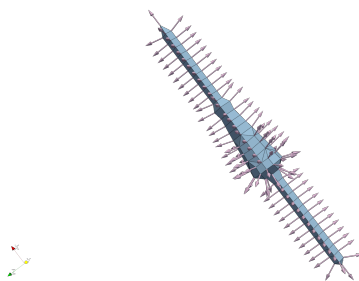
(b) Córtez de la implementación



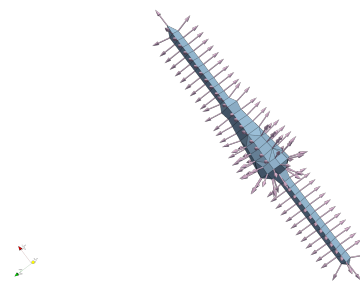
(c) Fémur con Paraview



(d) Fémur de la implementación



(e) Sutura con Paraview



(f) Sutura de la implementación

Figura 12: Comparación de normales de vértices.

CAPÍTULO 5

CONCLUSIONES

5.1. Conclusiones Generales

El desarrollo de esta memoria ha permitido integrar nuevas funcionalidades en el *mesher*, con el fin de extender sus capacidades hacia el análisis y visualización de la superficie de la malla generada. Esta extensión ha contemplado la identificación de caras superficiales, el cálculo de normales asociadas a dichas caras y la posterior obtención de normales en los vértices de la superficie. El trabajo ha logrado automatizar este proceso dentro del mismo entorno de generación de mallas, sin requerir herramientas externas, lo que representa un avance significativo en términos de autonomía, y alcance de los usos que puede proveer el *mesher*.

5.2. Cumplimiento de objetivos y validación

Entre los principales logros del trabajo se encuentra el diseño y la implementación de un algoritmo de extracción de caras que permite identificar correctamente la superficie de una malla conformada por elementos mixtos (hexaedros, tetraedros, prismas y pirámides), lo cual corresponde al primer objetivo específico propuesto. Una vez extraídas las caras fue posible identificar las pertenecientes a la superficie asegurándose de que dicha cara no estuviera presente en ningún otro elemento de la malla.

Además, se implementó el cálculo de normales de superficie a nivel de cara y vértice. Para las caras triangulares, se utilizó el producto cruz de sus aristas, mientras que para las caras cuadriláteras, se implementó el método de Newell, más robusto ante geometrías no planas. Para la normal de vértice, esta fue obtenida mediante la suma acumulada de las normales de las caras adyacentes, seguida de una normalización para obtener un vector unitario. Este enfoque cumple con el segundo y tercer objetivo específico propuesto por esta memoria y fue suficiente para entregar resultados adecuados a lo deseado como fue comprobado posteriormente mediante un proceso comparativo con el fin de validar los resultados.

La validación de estas implementaciones fue realizada mediante una doble comparación, gráfica y numérica. Primero, se comparó la salida de la implementación con los resultados obtenidos mediante el software Paraview, con tal de renderizar la superficie obtenida mediante ambos métodos y además visualizar la orientación de cada vector normal. A simple vista fue posible observar que coincidían sin evidencias de errores de orientación o agujeros.

Luego, se contrastaron las normales de vértice obtenidas con las de Paraview, empleando un *script* que determina que la diferencia de cada componente del vector normal no supere un límite establecido en 10^{-6} por componente, además, se determinaron los errores angulares máximos para cada caso de prueba y se concluyó que para estos, los errores son todos menores a $0,03^\circ$, lo cual es un error considerado aceptable, entre los posibles motivos que pueden explicar la presencia de variaciones en los valores obtenidos se encuentran:

- Paraview podría no estar trabajando con los mismos datos de entrada, si bien al momento de cargar las coordenadas de los puntos de la malla a Paraview nos estamos asegurando de usar la máxima precisión que la estructura de datos de tipo *Double* es capaz de proveer, no fue posible asegurar que Paraview esté computando con los valores empleando el mismo nivel de precisión numérica, esto podría explicar que los resultados sean similares y comiencen a diferir consistentemente a partir de cierta cifra significativa.
- Pueden existir diferencias en la implementación, Paraview emplea una forma generalizada de calcular normales de superficie, la cual es válida para cualquier tipo de polígono.
- Error de la implementación, siempre existe la posibilidad de que la implementación propuesta presente algún error que introduzca la leve variación observada en los resultados.

5.3. Trabajo Futuro

Con el fin de complementar este trabajo se identifican algunas áreas de mejora a ser consideradas en trabajos futuros. En primer lugar sería valioso poder determinar el origen de las variaciones obtenidas en los resultados, utilizando otra fuente de comparación, para así determinar si son producto de cálculos realizados con un mayor nivel de precisión en la entrada o algún error o diferencia en la implementación. Adicionalmente, la detección de bordes afilados o características geométricas singulares (*features*) no fue considerada, lo cual sería un paso lógico en el desarrollo de las herramientas del *mesher* en esta línea, técnicas como *splitting* pueden ser implementadas.

La eficiencia es otro punto con potencial de mejora, la implementación realizada en esta memoria presenta la posibilidad de paralelización de sus procesos tales como la identificación de caras y el cómputo de normales lo cual podría reducir su tiempo de ejecución significativamente.

ANEXOS

```

1 import pandas as pd
2 import numpy as np
3
4 def read_surface_VTK(filename):
5     points= []
6     point_normals= []
7
8     with open(filename, 'r') as f:
9         lines= f.readlines()
10
11     i= 0
12     while i < len(lines):
13         line = lines[i].strip()
14
15         if line.upper().startswith("POINTS"):
16             _, n_points, _ = line.split()
17             n_points= int(n_points)
18             i+= 1
19             while len(points)< n_points:
20                 parts= lines[i].strip().split()
21                 floats= list(map(float, parts))
22                 for j in range(0, len(floats), 3):
23                     points.append(floats[j:j+3])
24                 i+= 1
25             continue
26
27         if line.upper().startswith("POINT_DATA"):
28             _, n_point_data= line.split()
29             n_point_data= int(n_point_data)
30             i+= 1
31             if "NORMALS" in lines[i].upper():
32                 i += 1
33                 while len(point_normals) < n_point_data:
34                     parts= lines[i].strip().split()
35                     floats= list(map(float, parts))
36                     for j in range(0, len(floats), 3):
37                         point_normals.append(floats[j:j+3])
38                     i+= 1
39                 continue
40
41             i+= 1
42
43     return points, point_normals
44
45
46
47 def error_angular(v1, v2):
48     v1= np.array(v1)
49     v2= np.array(v2)
50     cos_theta= np.dot(v1, v2)
51     cos_theta= np.clip(cos_theta, -1.0, 1.0)
52     angulo= np.arccos(cos_theta)
53     angulo= np.degrees(angulo)
54     return angulo
55
56 def encontrar_punto_con_tolerancia(p, puntos2, tol=1e-12):
57     for j, q in enumerate(puntos2):

```

```

58     if all(abs(pi - qi) <= tol for pi, qi in zip(p, q)):
59         return j
60     return None
61
62
63 def comparar_normales(puntos, normales, puntos2, normales2, tol_n=1e-6):
64     mismatch_counter=0
65     match_counter=0
66     max_angular_error=0
67     for i, p in enumerate(puntos):
68         j = encontrar_punto_con_tolerancia(p, puntos2)
69         if j is None:
70             continue
71
72         n1= normales[i]
73         n2= normales2[j]
74
75         if any(abs(a - b) > tol_n for a, b in zip(n1, n2)):
76             mismatch_counter+=1
77             print(normales[i], normales2[j])
78
79         else:
80             angular_error=error_angular(n1,n2)
81             if angular_error>max_angular_error:
82                 max_angular_error=angular_error
83             match_counter+=1
84
85     print(f"Total de puntos: {len(puntos)}")
86     print(f"Total de fallos: {mismatch_counter}")
87     print(f"Total de matchs: {match_counter}")
88     print(f"Tolerancia de aceptacion de normal: {tol_n}")
89     print(f"Error angular maximo: {max_angular_error}°")
90
91
92
93 df = pd.read_csv("A2.csv")
94 puntos= df[["Points:0", "Points:1", "Points:2"]].values.tolist()
95 normales= df[["Normals:0", "Normals:1", "Normals:2"]].values.tolist()
96 puntos_implementacion, normales_implementacion= read_surface_VTK("surface.vtk")
97 comparar_normales(puntos, normales, puntos_implementacion, normales_implementacion)
98
99 #este codigo esta lo mas lejos posible de ser eficiente, pero fue funcional.

```

REFERENCIAS

BIBLIOGRÁFICAS

- [Díaz et al., 2023] Díaz, J., Lobos, C., y Hitschfeld-Kahler, N. (2023). Node-based data structure for balancing process optimization of quadtree meshes. En *2023 42nd IEEE International Conference of the Chilean Computer Science Society (SCCC)*, número 2, pp. 1–7.
- [George et al., 2017] George, P. L., Borouchaki, H., Alauzet, F., Laug, P., Loseille, A., Marcum, D., y Maréchal, L. (2017). *Mesh Generation and Mesh Adaptivity: Theory and Techniques*, pp. 1–51. John Wiley Sons, Ltd.
- [Gouraud, 1971] Gouraud, H. (1971). *Continuous shading of curved surfaces*. Ph.d. thesis, University of Utah.
- [Jaillet y Lobos, 2022] Jaillet, F. y Lobos, C. (2022). Fast quadtree/octree adaptive meshing and re-meshing with linear mixed elements. *Engineering with Computers*, 38(3):1–18.
- [Jin et al., 2005] Jin, S., Lewis, R., y West, D. (2005). A comparison of algorithms for vertex normal computation. *The Visual Computer*, 21:71–82.
- [Kitware, a] Kitware. Vtk - the visualization toolkit. Recuperado el 5 de mayo de 2025.
- [Kitware, b] Kitware. Vtk file formats. Recuperado el 5 de mayo de 2025.
- [Lobos, 2015] Lobos, C. (2015). A set of mixed-element transition patterns for adaptive 3d meshing.
- [Tampieri, 1992] Tampieri, F. (1992). V.5 - newell's method for computing the plane equation of a polygon. En KIRK, D., editor, *Graphics Gems III (IBM Version)*, pp. 231–232. Morgan Kaufmann, San Francisco.
- [Yerry y Shephard, 1983] Yerry, M. A. y Shephard, M. S. (1983). A modified quadtree approach to finite element mesh generation. *IEEE Computer Graphics and Applications*, 3(4):39–46.