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

ALGORITMO PARA LA PARALELIZACIÓN EN GPU DE DIAGRAMAS DE VORONOI

MEMORIA PARA OPTAR AL TÍTULO DE
INGENIERO CIVIL EN COMPUTACIÓN

PABLO RODRIGO POLANCO GALLEGUILLOS

PROFESORA GUÍA:
NANCY HITSCHFELD KAHLER

MIEMBROS DE LA COMISIÓN:
LUIS MATEU BRULE
SERGIO OCHOA DELORENZI
ÉRIC TANTER

SANTIAGO DE CHILE
2020

Resumen

En la computación gráfica la cantidad de simulaciones y sus dimensiones ha crecido a lo largo del tiempo. Sus avances en diferentes áreas expresa la necesidad de que los algoritmos que procesan y visualizan la información sean más rápidos y robustos. Una de estas visualizaciones son los diagramas de Voronoi, cuyo objetivo es teselar el espacio a través de polígonos cuyos elementos internos tienen en común un elemento: son más cercanos a ciertos puntos especificados, resolviendo problemas como por ejemplo la ubicación de los puestos de vigilancia en una reserva forestal.

Estos diagramas están asociados a otro tipo de grafos llamados Triangulaciones de Delaunay, siendo entre sí grafos duales. Existen varios algoritmos que calculan este último de manera paralela en GPU pero no así de los diagramas de Voronoi, dando pie al objetivo de este trabajo: Diseñar e implementar un nuevo algoritmo paralelo en GPU que permita el cálculo y visualización del diagrama de Voronoi utilizando como base una triangulación de Delaunay.

Tras la revisión de los algoritmos dedicados a estos diagramas, se decidió realizar una implementación compatible con la librería CLEAP, que calcula de manera paralela y en GPU una triangulación de Delaunay dada una malla de triángulos cualquiera. Esto hace que sea imprescindible el entendimiento de las estructuras de datos y procesos que la librería lleva a cabo, con el fin de optimizar el tiempo de cálculo del diagrama en cuestión.

Dadas las estructuras de datos existentes, se crearon nuevos elementos para almacenar la información relevante a los diagramas de Voronoi. Estas estructuras deben ser capaces de interoperar con los buffers que OpenGL utiliza para la visualización y con los arreglos de datos que CUDA maneja. Junto a esto, los algoritmos implementados deben calcular los vértices y aristas del nuevo diagrama de manera paralela sin que se produzcan pérdidas de información.

Los resultados obtenidos fueron positivos, permitiendo a la librería calcular un diagrama de Voronoi agregando a lo más 0.09 segundos de procesamiento adicional (equivalente a un 2 % del tiempo total) para mallas de 4.000.000 de puntos, sin alterar el comportamiento previo a la implementación. En conjunto a esto, el tiempo de procesamiento llega a ser 5 veces más rápido que otros algoritmos de similar método en CPU (Qhull).

Por otro lado, la nueva implementación aumenta el uso de memoria en un 250 %. Comparado con Qhull, el uso de memoria es bastante similar (si solo se considera lo almacenado a nivel de GPU). Se plantea una mejora como trabajo futuro para disminuir esta medición.

Agradecimientos

Quiero partir agradeciendo a mi familia: a mi abuela Práxedes que siempre veló por mi, me incentivó a hacer las cosas que amo y disfruto y me dió las fuerzas para llegar a esta instancia. A mi madre Violeta que ha sido uno de mis grandes apoyos en todos estos años siendo más que una madre, una confidente, una amiga del alma. A mi padre Dagoberto que por menos cosas que me diga tengo claro que siempre está pendiente de lo que sucede. A mis hermanos y hermanas, los amo con todo mi corazón y ya podremos celebrar en grande todo esto. A mis primas amadas, que han estado en las buenas y en las malas.

A mis amigos: las Oncecitas que han sido lo mejor de todo, sus risas, llantos, alegrías, apoyo y todo, los amo muchísimo. A los Malosql cada vez vamos quedando menos por salir del grupo, les agradezco el bullying con amor hecho sobre mi jajaja. A las Vacas, un grupo de gente de mucho amor y cariño que siempre están ahí. A las mesas de rol, gracias por tenerme tanta paciencia; volveré pronto! A Most Wanted - Team Rocket que por más tiempo que pase siempre están ahí para lo que necesite.

En particular, quiero agradecer a algunas personas: A Ian y Diego, gracias chiquillos por alojarme en su departamento durante todo este hermoso (not) año de pandemia, con Isis y mis crisis ansiosas. Ya están dentro de los saludos arriba, pero mi memoria no hubiese sido lo mismo sin su apoyo directo. A Mariana y Fran, por estos 10 años de amistad sin dudas (ojo que es el primer o segundo aviso de los pasajes a Disney jajaj). A mi amigui del alma Cami, que con 17 años encima esta cosa no muere, ya no te alejaste a tiempo de mi jajajaj. A Sergio M, estoy seguro que tus ojos azules me miran desde el cielo diciendo: "Se logró!". A Fernando que en el poco tiempo ha sido un apoyo enorme dentro de todo este proceso, sobretodo para mi paz mental. Gracias lindo.

A los diferentes coros y grupos musicales que siempre me han recibido con mucho afecto y se han preocupado por mi, al ADI que me ha dado un lugar de trabajo donde también he podido avanzar en mi memoria. A mi profesora guía Nancy, que ha sufrido tanto como yo este trabajo de título. A mis profesores del DCC que me han enseñado a ser una persona crítica pero sin dejar de lado el sentido humano. Y al Fondecyt 1181506 que apoyó este trabajo de título.

Finalmente pero no menor, quiero agradecerme por toda la entereza a lo largo de este trayecto. Nos lo merecemos de corazón. Al fin.

Tabla de Contenido

1. Introducción **1**
1.1. Justificación . . . 3
1.2. Objetivos de la Memoria . . . 4
1.3. Metodología . . . 5
1.4. Contenido de la Memoria . . . 5

2. Marco teórico **7**
2.1. Conceptos geométricos y mallas . . . 7
2.1.1. Mallas de polígonos . . . 7
2.1.2. Triangulación de Delaunay . . . 7
2.1.3. Teselación de Voronoi . . . 9
2.1.4. Dualidad Delaunay-Voronoi . . . 9
2.2. Programación en GPU . . . 10
2.2.1. CUDA . . . 10
2.2.2. OpenGL y lenguaje de Shaders . . . 13
2.2.3. Interoperabilidad entre CUDA y OpenGL . . . 16

3. Análisis y diseño de las librerías CLEAP y TIUQUE **18**
3.1. Introducción . . . 18
3.2. Estructuras de datos . . . 19
3.2.1. Vértices . . . 19
3.2.2. Aristas . . . 19
3.2.3. Triángulos . . . 20
3.2.4. Mallas . . . 20
3.3. Algoritmos en CLEAP . . . 20
3.3.1. Creación de estructuras y fill-in de datos . . . 20
3.3.2. Transformación de Delaunay . . . 25
3.4. Funcionalidades de TIUQUE . . . 28

4. Diseño e implementación del cálculo del Diagrama de Voronoi **31**
4.1. Integración . . . 31
4.2. Estructuras . . . 31
4.2.1. Circuncentros . . . 31
4.2.2. Aristas de Voronoi . . . 32
4.2.3. Índices-Aristas . . . 33
4.2.4. Relación aristas de Voronoi . . . 33

4.2.5. Polígonos 33
4.3. Algoritmo 34
4.3.1. Identificación de Cirucuncentros 34
4.3.2. Indexación de aristas 34
4.3.3. Proyección de puntos medios 35
4.3.4. Relación cirucuncentro-arista 35
4.3.5. Half-edges y Reconocimiento de polígonos 36
4.4. Visualización de Voronoi 40

5. Pruebas y Resultados **42**
5.1. Cambios de interfaz 42
5.2. Definición de pruebas 42
5.3. Uso de memoria 43
5.3.1. CLEAP: Delaunay vs Delaunay-Voronoi 44
5.3.2. Delaunay-Voronoi: CLEAP vs Qhull 44
5.4. Tiempos de ejecución 46
5.4.1. CLEAP: Delaunay vs Delaunay-Voronoi 46
5.4.2. Delaunay-Voronoi: CLEAP vs Qhull 46
5.5. Representaciones y correctitud 48

6. Conclusiones **56**

Bibliografía **57**

Anexos **58**
Anexo A Capítulo 2 58
Anexo B Capítulo 4 60
Anexo C Capítulo 5 63

Índice de Tablas

5.1. Comparación de uso de memoria entre el cálculo original de la librería y el de la nueva implementación . . . 44
5.2. Comparación de uso de memoria de CLEAP y Qhull para mallas con potencias de 10 puntos . . . 45
5.3. Comparación de uso de memoria de CLEAP y Qhull para mallas desde 1.000.000 hasta 4.000.000 de puntos . . . 45
5.4. Comparación de tiempos de ejecución entre el cálculo original de la librería y el de la nueva implementación . . . 47
5.5. Tiempos de ejecución para mallas de potencias de 10 puntos en CLEAP y Qhull 47
5.6. Comparación de tiempos de ejecución para mallas desde 1.000.000 hasta 4.000.000 de puntos entre CLEAP y Qhull . . . 47

Índice de Ilustraciones

1.1. Triangulación que cumple la condición de Delaunay 1
1.2. Triangulación de Delaunay y Voronoi para la misma nube de puntos 2
1.3. Transformaciones entre representaciones 2D, Delaunay y Voronoi. Se señalan algunos algoritmos posibles para las transformaciones. 3

2.1. Representación de una superficie 3D con triangulación de Delaunay, con una nube de puntos obtenida vía LiDAR 8
2.2. *Minimum spanning tree* en una malla de triángulos 8
2.3. Estudio sobre la fuente de infección de cólera en Londres, siglo XIX. Los puntos azules corresponden a fuentes de agua potable, mientras los rojos son los casos de cólera que se presentaron 10
2.4. Diagrama de Voronoi de los aeropuertos en Estados Unidos. Cada punto representa un aeropuerto 11
2.5. Estudio de cadenas de comida rápida reconocidas y sus áreas de alcance. Los puntos rojos corresponden a los locales 11
2.6. Diagrama de Voronoi de densidad poblacional 12
2.7. Teselación de Voronoi asociada a las áreas que cubre cada jugador en un partido de fútbol 12
2.8. Triangulación de Delaunay de una nube de puntos 13
2.9. Circumcentros asociados a la triangulación de Delaunay 13
2.10. Unión de los circumcentros y visualización de Diagrama de Voronoi 14
2.11. Estructuración interna del modelo de programación CUDA 14
2.12. Flujo de procesos y datos en CUDA 15
2.13. Máquina de estados de OpenGL 16
2.14. Asociación de data entre CUDA y OpenGL 17
2.15. Interoperabilidad entre CUDA y OpenGL serial 17

3.1. Comparación de estructuras asociadas a la malla. En el recuadro superior se encuentra la estructura a nivel de CPU, mientras que en el inferior es a nivel GPU 21
3.2. Ejemplo de estructuras, arreglos y punteros de una malla de triángulos. . . . 22
3.3. Hashes utilizados para la indexación de aristas. **root_hash** almacena un hash **hit** con una llave equivalente al mayor índice de los vértices que conforman la arista. Por otro lado, **hit** es un hash que contiene aristas temporales indexadas por el menor índice de los vértices de las mismas. 24

3.4. Generación de buffer de los vértices de la malla. En verde se destacan los arreglos asociados a *Device* y en rojo a *Host* 24
3.5. Cantidad de bloques, grillas e hilos que se utilizarán para la transformación de Delaunay 25
3.6. Hilos utilizados en la transformación de Delaunay. Figura tomada del artículo "A Parallel GPU-based Algorithm for Delaunay Edge-flips" [1] 25
3.7. Malla de triángulos previa y su estructura previa al proceso de transformación de Delaunay. Se puede apreciar que la suma de $\measuredangle ab$ y $\measuredangle c$d es mayor que π, por lo que no cumple con la condición de Delaunay. 26
3.8. Ejemplificación de exclusión de triángulos en GPU 27
3.9. Malla de triángulos luego de ejecutar el paso de Detección, Exclusión y Procesamiento. Es posible observar que la arista e ha sido girada, modificando su información. A su vez, las aristas b y d han quedado inconsistentes. 27
3.10. Malla de triángulos tras realizar la reparación de las aristas b y d 28
3.11. TIUQUE: interfaz gráfica que permite utilizar los métodos de CLEAP 29
3.12. Malla de triángulos cargada en TIUQUE 29
3.13. Tipos de visualización en TIUQUE. A la izquierda se observa solo la visualización de tipo sólida, mientras que en la derecha se exponen los $wireframes$ de la malla. 30
3.14. Malla de triángulos tras realizar la reparación de las aristas b y d 30

4.1. Simplificación de cambios en implementación a nivel de estructuras. Los elementos destacados son las estructuras y datos que se agregarán en la implementación 32
4.2. Triangulación de Delaunay y Diagrama de Voronoi para ejemplificación de proceso. 36
4.3. Proceso de swap en relación cirucuncentro-arista 37
4.4. Caso inicial de ejemplificación de reconocimiento de polígonos 38
4.5. Caso inicial de ejemplificación de reconocimiento de polígonos 39
4.6. Producto cruz entre vector formado por circuncentros y cirucuncentro-vértice de Delaunay. 40
4.7. Half edges y polígonos formados 40
4.8. Fragmento de código dedicado a la visualización del diagrama de Voronoi usando OpenGL y Shaders 41

5.1. Nueva interfaz gráfica de TIUQUE 43
5.2. Uso de memoria de CLEAP D-V y Qhull para mallas con potencias de 10 puntos 45
5.3. Uso de memoria de CLEAP y Qhull para mallas desde 1.000.000 hasta 4.000.000 de puntos 46
5.4. Tiempos de ejecución para mallas de potencias de 10 puntos 48
5.5. Tiempos de ejecución para mallas desde 1.000.000 hasta 4.000.000 48
5.6. Malla de triángulos del orden de 10 vértices previa a ser procesada 49
5.7. Triangulación de Delaunay obtenida por la librería 50
5.8. Visualización de la Triangulación de Delaunay y Diagrama de Voronoi asociados a la malla 50
5.9. Diagrama de Voronoi visualizado de manera independiente 51
5.10. Archivo .off resultante del proceso 51

5.11. Visualización del archivo de exportación del diagrama de Voronoi para malla de orden de 10 puntos 52
5.12. Malla de triángulos del orden de 10^4 previa a procesamiento 52
5.13. Visualización de diagramas de Delaunay y Voronoi para malla del orden de 10^4 puntos 53
5.14. Fragmento de malla de triángulos del orden de 10^6 vértices 53
5.15. Fragmento de visualización de Delaunay-Voronoi para 10^6 puntos 54
5.16. Malla de triángulos que representa la superficie de una manzana de 891 puntos 54
5.17. Visualización de Voronoi y Delaunay en la superficie de una manzana 55

6.1. Exposición de mallas con distintos polígonos 58
6.2. Diferentes tipos de malla sobre la superficie de un poliedro 59
6.3. Teselaciones con polígonos variados 59
6.4. Visualización de circumentros de una triangulación 60
6.5. Diagrama de Voronoi y Triangulación de Delaunay de una malla de triángulos 61
6.6. Variación de visualización 6.5 ocultando las aristas de la triangulación 61
6.7. Variación de visualización 6.5, exponiendo aristas de ambos diagramas simultaneamente 62
6.8. Archivo .off resultante del proceso. Se puede ver el número de vértices y aristas que componen el diagrama, así como algunas definiciones 63
6.9. Visualización del diagrama de Voronoi expuesto en 6.8 a través de RoffView 63
6.10. Visualización del diagrama de Voronoi expuesto en 6.8 a través de TIUQUE 64
6.11. Malla de triángulo de orden de 10^2 puntos 64
6.12. Delaunay y Voronoi de malla 6.11 65
6.13. Malla de triángulo de orden de 10^3 puntos 65
6.14. Triangulación de Delaunay de malla 6.13 66
6.15. Teselación de Voronoi de malla 6.13 66
6.16. Malla de triángulo de orden de 10^5 puntos 67
6.17. Zoom a Triangulación de Delaunay de malla 6.16 67
6.18. Zoom a Teselación de Voronoi de malla 6.16 68
6.19. Malla de triángulo con 38.000 puntos, representando la superficie de un conejo 68
6.20. Teselación de Voronoi de malla 6.19 69
6.21. Delaunay y Voronoi de malla 6.19 69
6.22. Malla de triángulo con 526 puntos, representando la superficie de una espada 70
6.23. Teselación de Voronoi de malla 6.22 70
6.24. Delaunay y Voronoi de malla 6.22 71
6.25. Teselación de Voronoi de malla 5.16 71

Capítulo 1

Introducción

La computación gráfica es un área que ha tenido un desarrollo muy grande en los últimos años, debido a la gran cantidad de simulaciones, visualizaciones y representaciones 2D y 3D que se requieren realizar con fines tanto científicos como otros más cotidianos. Dentro de esta área, se encuentra la necesidad de trabajar con mallas geométricas que representan la estructura de los cuerpos modelados, y sus interacciones físicas con otros elementos. Las mallas geométricas más usadas están estructuradas con triángulos o cuadriláteros en dos dimensiones, y tetraedros o hexaedros en tres dimensiones.

Una de las triangulaciones más usadas para la generación de mallas 2D es la de Delaunay [2], la cual a partir de una nube de puntos genera una malla donde todos los triángulos cumplen con el hecho de que su circunferencia circunscrita no contiene ningún otro vértice de la triangulación en su interior, aunque pueden haber vértices sobre la circunferencia. La Figura 1.1 muestra gráficamente la regla a cumplir.

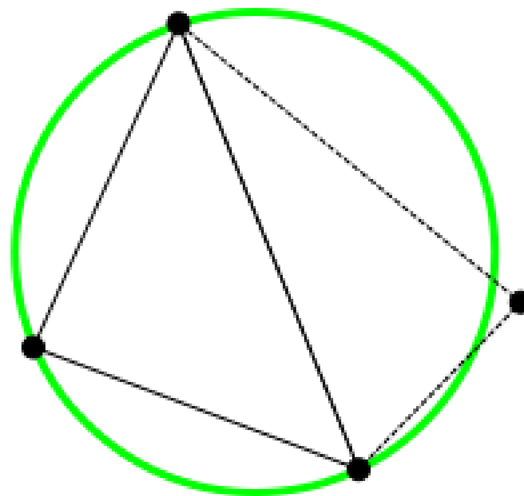

Figura 1.1: Triangulación que cumple la condición de Delaunay

Este método de triangulación permite realizar cálculos muy útiles para simulaciones o

visualizaciones, pero no es muy eficiente para dar respuesta a algunos requerimientos comunes, como por ejemplo, conocer el punto más cercano a una coordenada. Para abordar este segundo tipo de requerimiento, existe otro tipo de teselación (o generación de malla) llamada diagrama de Voronoi [2], donde la malla se genera a partir de regiones. Los circuncentros de los triángulos de Delaunay coinciden con los vértices de las regiones del diagrama de Voronoi, por lo que la construcción de uno de estos diagramas es bastante simple a partir de los datos del otro. Es decir, la triangulación de Delaunay y el diagrama de Voronoi de una serie de puntos son grafos duales [1]. La Figura 1.2 muestra una malla con las representaciones de Delaunay (en negro) y de Voronoi (en rojo) para el mismo conjunto de puntos.

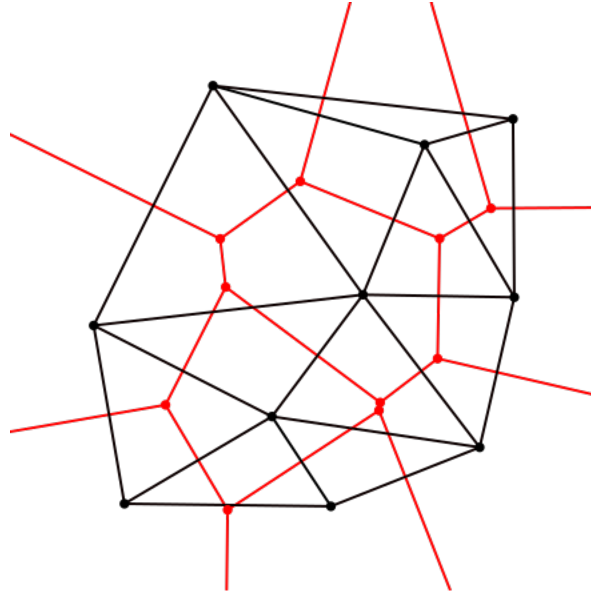

Figura 1.2: Triangulación de Delaunay y Voronoi para la misma nube de puntos

Dada la capacidad de transformar triangulaciones Delaunay a diagramas de Voronoi y vice versa, ante la necesidad de generar una malla geométrica es posible escoger el tipo de representación a utilizar en función del propósito de la teselación a realizar. Esto no reviste mayor desafío cuando la nube de puntos es pequeña; sin embargo, la creciente necesidad de modelar geometrías más complejas y de mayor dimensión (del orden de millones de puntos), ha llevado a la necesidad de mejorar los algoritmos, tanto de generación de mallas como de transformación entre representaciones de estas para que sean más eficientes, pues muchos servicios que utilizan a estas representaciones como base, requieren dar una respuesta en tiempo-real. Es por ello que los algoritmos de generación y transformación de mallas han ido pasando de ser secuenciales, a sus versiones equivalentes en algoritmos paralelos.

Las versiones paralelas de muchos de estos algoritmos secuenciales han sido implementadas para ser procesadas por Unidades de Procesamiento Gráfico (Graphics Processing Unit - GPU), las cuales fueron incluidas en los computadores con el fin de apoyar los cálculos de punto flotante y procesamiento gráfico. A lo largo del tiempo, las GPU han dejado de ser exclusivas para los usos anteriormente nombrados, pasando a ser un procesador de propósitos más genérico, facilitando la paralelización de varios algoritmos no necesariamente del área

[1] G_1 y G_2 son grafos duales si por cada región del G_1 existe un vértice de G_2 y viceversa; y que por cada arista de uno existe una arista en el otro que une las regiones vecinas.

de la computación gráfica[3][4]. Hoy en día se cuenta con algunos algoritmos paralelos que ejecutan en GPU para generar las mallas y realizar las transformaciones antes mencionadas; sin embargo, aún faltan implementaciones de varios algoritmos para que operen sobre GPU y realicen los cálculos en tiempos más reducidos.

Particularmente, las capacidades que proveen los diagramas de Voronoi para generar información en varios dominios de aplicación (por ejemplo, transporte, medicina, y medioambiente, entre otros) y la vasta cantidad de data (puntos de referencia a considerar) que se usan como base, generan una necesidad latente de procesar, en tiempo real y de manera eficiente, los cálculos asociados a este tipo de diagramas. La mayoría de los algoritmos que generan representaciones Voronoi no son lo suficientemente eficientes, no aprovechan la dualidad Delaunay-Voronoi, o simplemente no utilizan todas las herramientas de cálculo de punto flotante actuales. La Figura 1.3 muestra los distintos caminos para llegar a obtener una malla geométrica representada a través de un diagrama de Voronoi.

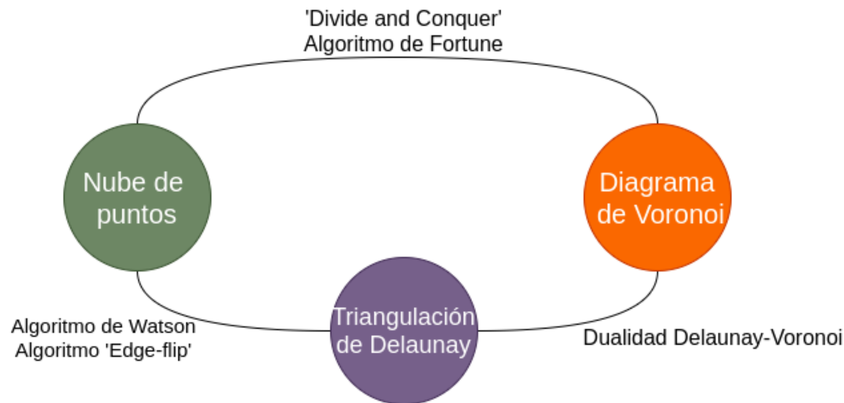

Figura 1.3: Transformaciones entre representaciones 2D, Delaunay y Voronoi. Se señalan algunos algoritmos posibles para las transformaciones.

Para contribuir a abordar el problema planteado, este trabajo de título propone el diseño e implementación de un nuevo algoritmo que, a través de la paralelización, el aprovechamiento de la dualidad Delaunay-Voronoi, y el uso de GPU, mejore los tiempos de ejecución y procesamiento de las teselaciones de Voronoi al momento de trabajar nubes de puntos bidimensionales. Para que dicho algoritmo quede disponible al público, se extenderá la librería open source llamada CLEAP [2] embebiendo el nuevo algoritmo, y garantizando que el resto de los servicios de la librería funcionen correctamente.

1.1. Justificación

Tal como se explicó anteriormente, el objetivo es diseñar e implementar un nuevo algoritmo para procesar teselaciones de Voronoi utilizando la dualidad Delaunay-Voronoi. Existe mucha información común entre ambos tipos de malla, pero a su vez el cálculo de las aristas y áreas

[2]CLEAP es una librería opensource en C++ que transforma mallas de triángulos 2D y 3D en teselaciones de Delaunay a través de GPU

no es trivial. Con el paso del tiempo, las nubes de puntos han ido creciendo de gran manera, haciendo que el cálculo de estas teselaciones sea cada vez más costoso. Por lo tanto, es necesario cambiar la forma de operar sobre los datos, pasando desde una manera secuencial a otra que sea en paralelo. Sin embargo, los algoritmos actuales que utilizan la dualidad Delaunay-Voronoi mantienen el cálculo de manera secuencial, haciendo que sean ineficientes al momento de trabajar con millones de puntos.

El uso de GPU para realizar los cálculos de punto flotante ayudaría en este proceso, ya que estos procesadores están diseñados para realizar operaciones en paralelo. Sin embargo, cada vez que se quiera realizar una operación, es necesario atomizar lo más posible los cálculos, de manera que cada uno de los hilos se encargue de retornar un resultado. Junto a esto, es necesario también entregar la data existente desde la CPU hacia la GPU, estableciendo los tipos de datos adecuados para la operación de punto flotante y realizar el manejo de los resultados de vuelta, haciendo engorroso el proceso de reconstrucción de la data en CPU. A su vez, las operaciones en GPU trabajan prácticamente como una caja negra, vale decir, es posible tener una idea de los procesos que se están llevando a cabo dentro de esta, pero no del detalle de cada uno, haciendo poco transparente el proceso de implementación y depuración del algoritmo.

Este algoritmo debe ser visible al mundo. Para ello, hemos de extender la librería CLEAP, permitiendo que las funcionalidades ya existentes no se vean alteradas, siendo necesario realizar un análisis de cómo funciona esta librería, aprovechar los datos que ya calcula la herramienta, y manejarlos de manera diferente sin interferir en el proceso original. Es por todo esto que se decidió realizar el diseño e implementación del algoritmo mencionado anteriormente.

1.2. Objetivos de la Memoria

El objetivo general de este trabajo es diseñar e implementar un nuevo algoritmo paralelo en GPU, que permita realizar transformaciones Delaunay - Voronoi en 2D, de manera más eficiente, comparado con la librería Triangle. Éste fue escogido ya que en la actualidad es una de la librerías de mayor impacto en el área de la geometría computacional, en particular, para las mallas de triángulos y sus optimizaciones.

Los objetivos específicos que se desprenden el objetivo general son los siguientes:

- Diseñar el algoritmo paralelo para generar el Diagrama de Voronoi a partir de una triangulación Delaunay.
- Implementar el nuevo algoritmo para que funcione en GPU
- Comparar el desempeño del nuevo algoritmo con Qhull.
- Extender la librería CLEAP para que pueda utilizar el nuevo algoritmo implementado.
- Extender el visualizador TIUQUE para visualizar el diagrama de Voronoi.

1.3. Metodología

La metodología a seguir incluye las siguientes macro-actividades:

1. Aprendizaje del lenguaje
    - Aprender nociones básicas de C++
    - Aprender nociones básicas de CUDA
    - Comprender interoperabilidad de OpenGL y CUDA en el manejo de buffers y data
2. Revisión de los algoritmos actuales
    - Revisión de los algoritmos de triangulación de Delaunay en paralelo
    - Revisión de los algoritmos de generación de diagramas de Voronoi en paralelo
    - Revisión de las implementaciones existentes en GPU para el proceso completo
3. Diseño e implementación del algoritmo en 2D para generar la transformación Delaunay-Voronoi
    - Identificación de las estructuras de datos a utilizar.
    - Diseño del algoritmo de transformación.
    - Implementación las funciones de actualización del diagrama (agregar, remover, reestructuración de las zonas).
    - Integración de funciones para obtener el algoritmo completo.
4. Extensión de librería CLEAP
    - Revisión y comprensión de la estructura de la librería CLEAP.
    - Extensión de la librería para incluir el nuevo algoritmo.
    - Revisión del correcto funcionamiento de la nueva versión de la librería.
5. Evaluación del algoritmo 2D
    - Preparación del set de pruebas a realizar al algoritmo.
    - Verificación y análisis del rendimiento del algoritmo nuevo en comparación a lo existente.
    - Comparación de los resultados de rendimiento con Qhull e identificación de aspectos de mejora.

1.4. Contenido de la Memoria

En el capítulo 2 se mostrará el marco teórico de este Trabajo de Título, explicando conceptos base como malla de polígonos, Delaunay, Voronoi, CUDA, OpenGL entre otros. Esto entregará el conocimiento base para los siguientes puntos.

Durante el capítulo 3 se explicará el funcionamiento de las librerías CLEAP y TIUQUE, las estructuras que almacenan la triangulación de Delaunay y el procesamiento de los datos para llegar desde una malla de triángulos cualquiera.

En el capítulo 4 se entrará en detalles con respecto al diseño e implementación del algoritmo objetivo de este trabajo. Se expondrán las distintas estructuras de datos, sus conformaciones y los métodos que poblarán, visualizarán y exportarán esta información.

El capítulo 5 presentará los resultados obtenidos tras la implementación del algoritmo. En él, se expondrán los tiempos de ejecución del nuevo proceso, las comparaciones con Qhull y las visualizaciones e interfaz gráfica modificadas.

Finalmente, en el capítulo 6 se hace un análisis con respecto a los resultados esperados y los obtenidos, así como se plantean posibles mejoras para una proxima iteración del trabajo.

Capítulo 2

Marco teórico

2.1. Conceptos geométricos y mallas

2.1.1. Mallas de polígonos

Una malla de polígonos es un conjunto de vértices y aristas que construyen la superficie y forma de un poliedro a través de los polígonos formados por estos elementos, compartiendo entre ellos una o más aristas. Los polígonos que la constituyen en general son triángulos, cuadriláteros (quads) u otros tipos simples, cóncavos o convexos.

Dentro de los tipos de polígonos que pueden conformar una malla, uno de los más utilizados son los triángulos, ya que presentan la propiedad de que siempre son planares, vale decir, sus aristas siempre compartirán un mismo plano sin que existan cruces entre ellas. Esto facilita el proceso de renderización de las superficies de polígonos.

2.1.2. Triangulación de Delaunay

Una triangulación de Delaunay es una malla de triángulos que cumplen con la condición de Delaunay. Esta condición consiste un triángulo debe cumplir que el interior de la circunferencia circunscrita del mismo no contiene ningún otro punto o vértice que los del triángulo en sí. Importante destacar que un triángulo cumple la condición incluso si puntos que no corresponden están sobre la circunferencia circunscrita.

Existen varios tipos de algoritmos para generar este tipo de triangulaciones, entre los cuales están:

- *Divide and Conquer*[5]: Dado un conjunto de puntos como entrada, se ordena según la coordenanda X de cada uno y se divide en mitades. Luego, estos subconjuntos se siguen subdividiendo hasta que el tamaño de éstos sea menor o igual a 3. Estos subconjuntos de puntos se unen entre si, formando triángulos y líneas. Una vez realizado esto, por

cada par de subconjuntos vecinos, se agregan aristas suficientes para unir formando triángulos (aquí se verifica si cumplen con la condición de Delaunay) hasta formar la malla de triángulos completa. Su orden es $O(n\log n)$ para el peor caso.

- Watson[6]: Dado un conjunto de puntos como entrada, se genera un triángulo que contiene todo el conjunto en su interior y luego, cada punto de la nube de datos se agrega a la triangulación, agregando las aristas entre el punto y los vértices del triángulo que lo contienen, realizando los cambios para que se cumpla a condición de Delaunay. Una vez se agregan todos los puntos, se elimina el triángulo ficticio originado al comienzo y todas la aristas que lleguen a él. Su orden es $O(n^2)$ para el peor caso.
- Edge-Flip[7]: En este algoritmo, se recorren todas las aristas internas revisando si el par de triángulos adyacentes a la arista cumple la condición de Delaunay de forma aislada. Si no la cumplen, se aplica una operación de giro de arista, de forma que va introduciendo la condición de Delaunay poco a poco en la triangulación. Si bien esta operación tiene asegurada la convergencia en $\mathbb{R}^2$), su orden es de $O(n^2)$ para el peor caso.

Actualmente hay muchos usos para este tipo de triangulaciones, entre ellos el modelamiento de superficies 3D (Figura 2.1) o encontrar el *minimum spanning tree* en una nube de puntos (Figura 2.2)

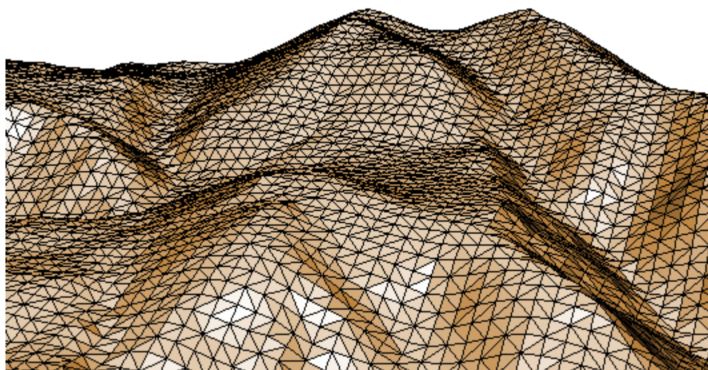

Figura 2.1: Representación de una superficie 3D con triangulación de Delaunay, con una nube de puntos obtenida vía LiDAR

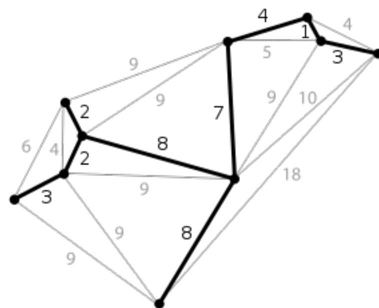

Figura 2.2: *Minimum spanning tree* en una malla de triángulos

2.1.3. Teselación de Voronoi

Una teselación de Voronoi [1] es un método de interpolación basado en distancia euclidiana, vale decir, es la división de un espacio (hablaremos de planos, ya que estamos trabajando en $\mathbb{R}^2$) en regiones, de forma que a cada punto de una nube de puntos se le asigna la región del espacio cuya distancia es la menor a cualquier otro punto. De esta forma, las zonas cuya distancia es equidistante entre dos o más puntos son los límites de los polígonos que se forman.

Existen varios algoritmos que son capaces de generar un diagrama de Voronoi, algunos de ellos son:

- Fuerza bruta: esta es la primera aproximación, pero su orden es $O(n^2 \log n)$, ya que por cada punto realiza $n-1$ semiplanos que se intersectarán para dar origen a la región del punto inicial.
- *Divide and Conquer*: Tal como dice su nombre, este algoritmo se basa en el paradigma de dividir y conquistar, de manera que se divide el conjunto de puntos de manera recursiva hasta llegar a un par sobre el cual se puede calcular el arista que divide sus regiones. Finalmente, se unen las aristas que se van formando hasta tener el plano completo. Su orden es $O(n \log n)$ para el peor caso.
- *Fortune*: este algoritmo se basa en una técnica de la geometría computacional llamada Barrido de Recta. Lo que hace es superponer una recta en el plano e ir avanzando desde un lado a otro (izquierda-derecha, arriba-abajo). Durante este barrido, se van formando las zonas en forma de parábolas que van intersectando con otras parábolas o con nuevos puntos de interés, formando la teselación. Su orden es $O(n \log n)$ para el peor caso.

De estos algoritmos existen varias implementaciones en paralelo e incluso en GPU, entregando buenos resultados pero todos en dominios de resolución bastante acotados.

Estas teselaciones han tenido un aumento en su uso en áreas fuera de la computación gráfica. Uno de sus usos más reconocidos fue en epidemiología, donde a través de estas es posible determinar la fuente de infección de Cólera en Londres durante el siglo XIX tal como se observa en la Figura 2.3. Otras áreas donde se ha masificado son transporte (Figura 2.4), marketing (Figura 2.5) u otros ejemplos.

2.1.4. Dualidad Delaunay-Voronoi

Es posible calcular Voronoi dada la dualidad con su triangulación de Delaunay correspondiente. Tal como se nombró anteriormente, existe una relación de grafos duales entre Voronoi y Delaunay, siendo posible construir uno con los datos del otro. El proceso es el siguiente:

- Por cada arista de la triangulación de Delaunay se calcula la bisectriz del ángulo opuesto a la arista.
- Por cada triángulo, se calcula el incentro del triángulo, que equivale al lugar donde las 3 bisectrices se intersectan. Estos incentros serán los vértices del diagrama de Voronoi

[1] También conocidos como teselación de Dirichlet, polígonos de Thiessen o diagramas de Voronoi

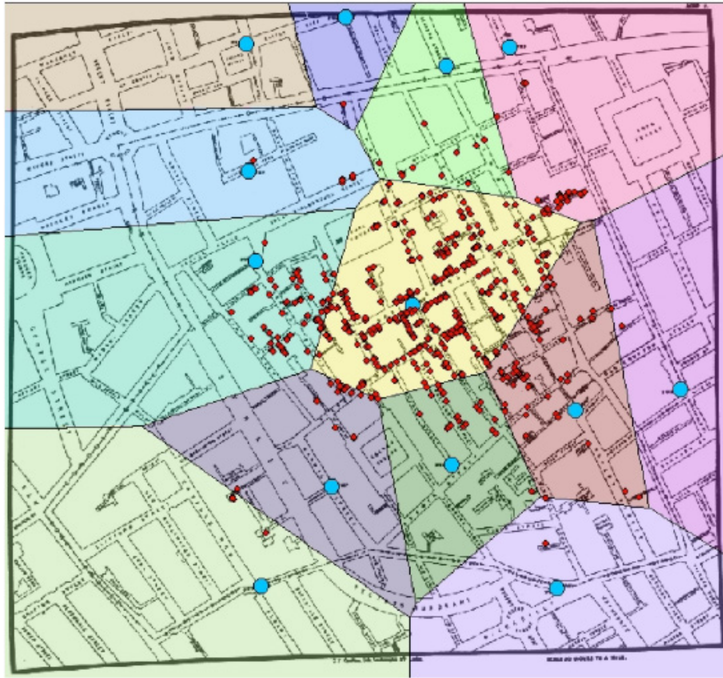

Figura 2.3: Estudio sobre la fuente de infección de cólera en Londres, siglo XIX. Los puntos azules corresponden a fuentes de agua potable, mientras los rojos son los casos de cólera que se presentaron

asociado.

- Finalmente, cada arista del diagrama corresponderá a la unión de los incentros de los triángulos adjacentes, formando así los polígonos asociados a la teselación de Voronoi.

Este proceso se ve reflejado en las Figuras 2.8, 2.9 y 2.10.

Esto nos da la posibilidad de crear un diagrama de Voronoi luego de haber calculado la triangulación de Delaunay, por lo que también debemos analizar los algoritmos que generan este tipo de triangulaciones. Para esta transformación existen algoritmos secuenciales que lo resuelven, pero no hay una implementación en paralelo basada en GPU.

2.2. Programación en GPU

2.2.1. CUDA

Compute Unified Device Architecture o CUDA es una plataforma de computación paralela y modelo de programación inventados por NVIDIA [8]. A través de esta herramienta es posible mejorar de manera significativa el rendimiento de un algoritmo utilizando la GPU.

La estructura que utiliza este modelo está definido por un sistema de grillas, en las cuales cada una tiene bloques de hilos. Cada hilo está identificado de manera única, a través de

Figura 2.4: Diagrama de Voronoi de los aeropuertos en Estados Unidos. Cada punto representa un aeropuerto

Figura 2.5: Estudio de cadenas de comida rápida reconocidas y sus áreas de alcance. Los puntos rojos corresponden a los locales

una componente de 3 elementos y son capaces de realizar operaciones de punto flotante de manera independiente. A su vez, cada bloque también se identifica de manera única, haciendo posible el acceso a cada hilo y cada bloque en una grilla, como aparece en la Figura 2.11.

Para programar en CUDA, es necesario levantar comunicaciones entre el *Host* (asociado

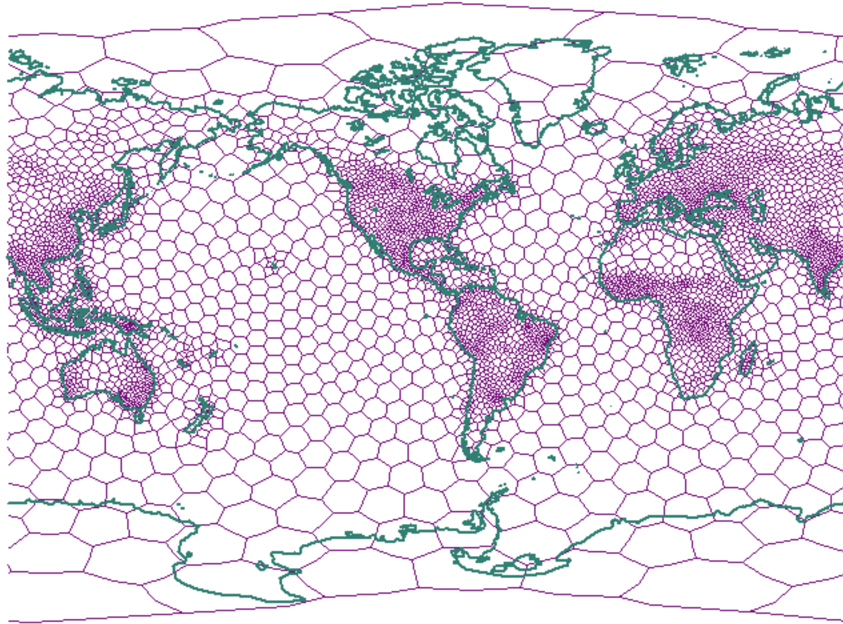

Figura 2.6: Diagrama de Voronoi de densidad poblacional

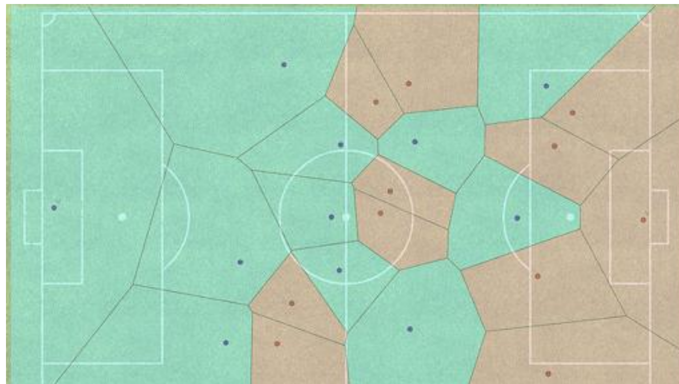

Figura 2.7: Teselación de Voronoi asociada a las áreas que cubre cada jugador en un partido de fútbol

en general a la CPU) y el *Device* (la GPU como tal). Todo código implementado se basa en un esquema de la siguiente forma:

- *Host*, de manera secuencial, realiza la copia de los datos a trabajar, destinando espacios de memoria en GPU para guardarlos.
- *Host* ordena a *Device* a ejecutar una función específica conocida por él, bajo ciertas limitantes de bloques e hilos a utilizar.
- *Device* por su parte, realiza el llamado en paralelo, enviando a cada hilo en los bloques anteriormente especificados a realizar algún cálculo (basado en operaciones de puntos flotantes) en conjunto con los datos previamente guardados en la memoria de la GPU.
- *Device* recompone la información, realizando una sincronización de todos los hilos operando (en espera de que finalicen). Una vez se logre, retorna la información en los espacios destinados de memoria en GPU.

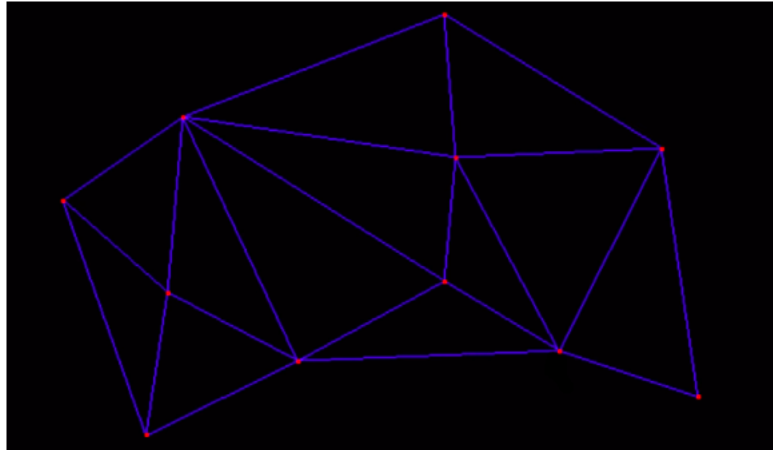

Figura 2.8: Triangulación de Delaunay de una nube de puntos

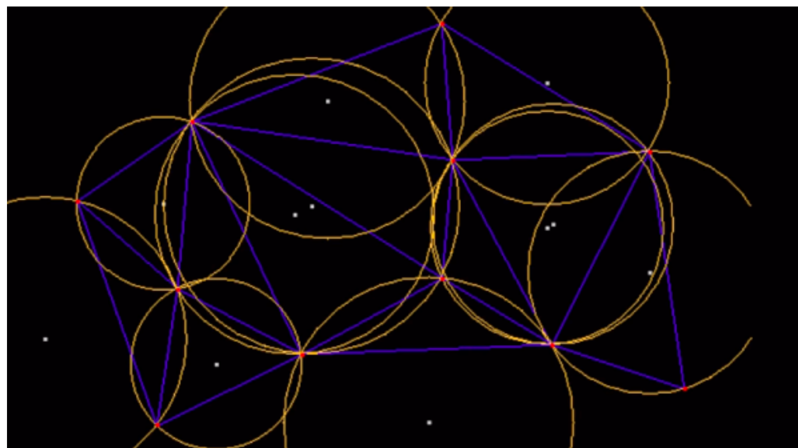

Figura 2.9: Circumcentros asociados a la triangulación de Delaunay

- Finalmente, *Host* recupera esta data desde GPU hacia la memoria en CPU.

Esto puede ser mejor visualizado en la Figura 2.12 a continuación.

Es importante destacar que al momento de trabajar con GPU no es posible pedir de manera dinámica la memoria a utilizar. Toda estructura de datos debe ser especificada y solicitada antes de correr cualquier kernel.

2.2.2. OpenGL y lenguaje de Shaders

OpenGL es una especificación[2] que define una API multilenguaje y multiplataforma que proporciona un conjunto amplio de funciones para la manipulación de gráficos e imágenes en 2D y 3D a través de elementos simples como puntos, líneas y triángulos. Su uso ha aumentado con creces a lo largo del mantenimiento de la misma, con rubros en diseño asistido

[2]Una especificación es un documento que describe y define el comportamiento exacto que debe tener los elementos asociados a esta

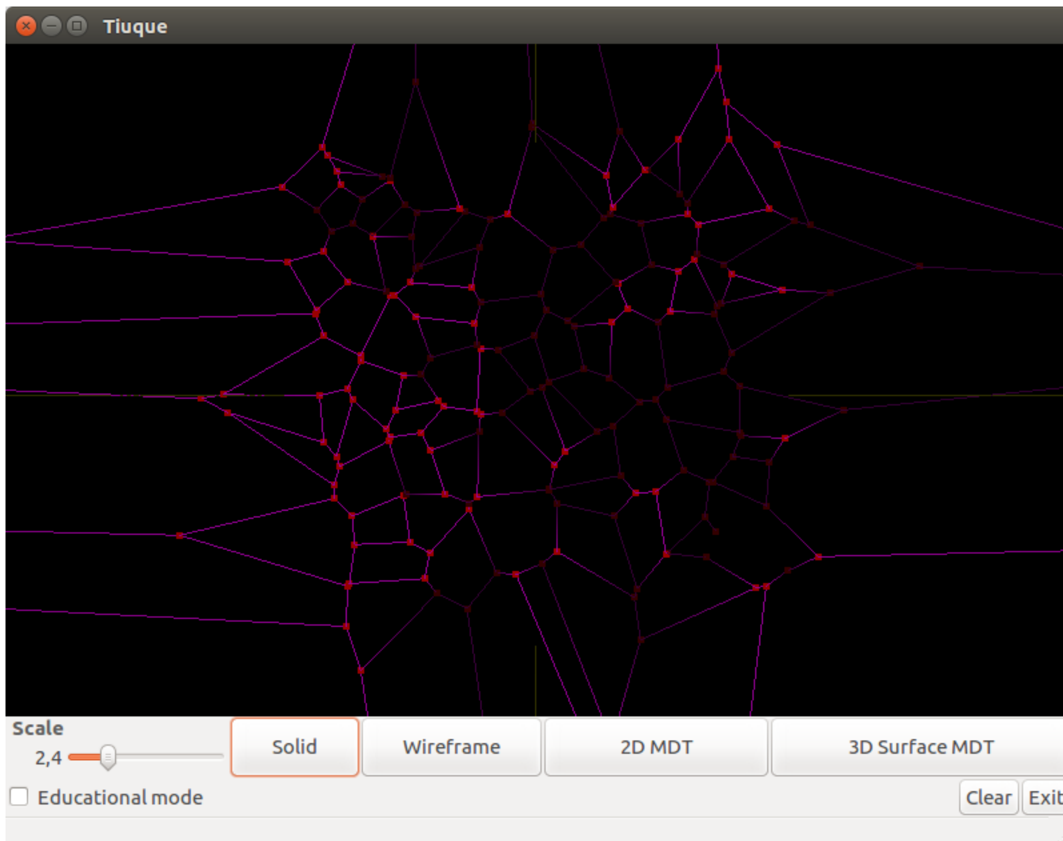

Figura 2.10: Unión de los circumcentros y visualización de Diagrama de Voronoi

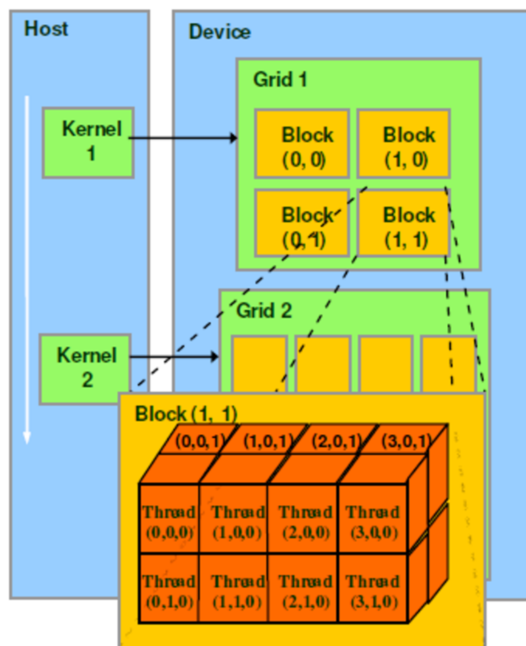

Figura 2.11: Estructuración interna del modelo de programación CUDA

por computador (Computer-aided design o CAD), realidad virtual, simulaciones científicas,

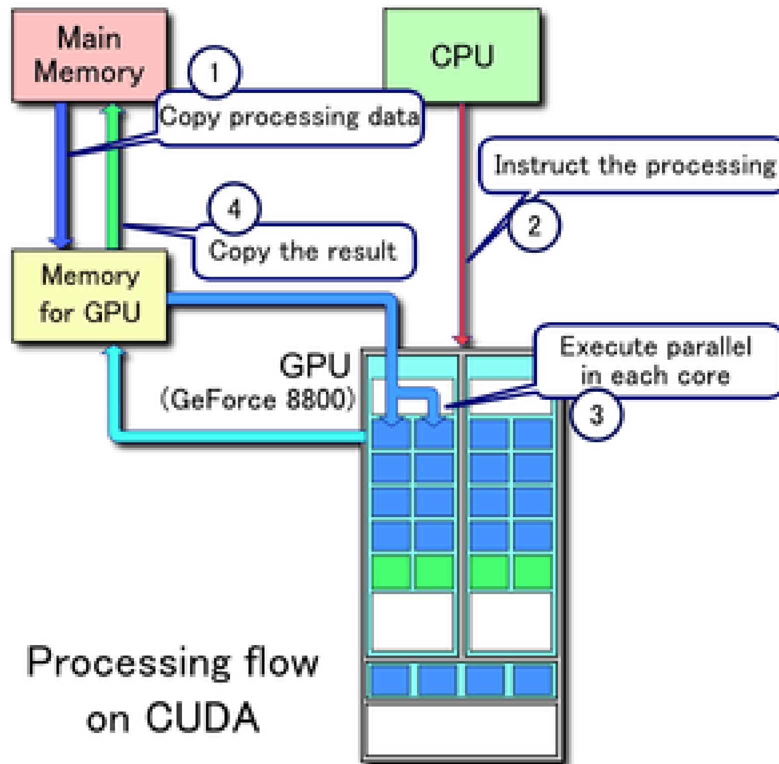

Figura 2.12: Flujo de procesos y datos en CUDA

visualizaciones de información, desarrollo de videojuegos, entre otros.

Tiene dos objetivos principales:

- Por el lado de los programadores busca eliminar la complejidad de trabajar con diferentes tarjetas gráficas y sus formas de utilizar a través de una API uniforme y única.
- Por el lado del hardware obliga a las diversas plataformas a que tengan una serie de requerimientos mínimos que soporten la funcionalidad completa de OpenGL, pudiendo hacer uso de emulaciones de ser necesario.

Tal como se nombra anteriormente, OpenGL funciona a través de primitivas como puntos, líneas y polígonos que a través de un pipeline gráfico conocido como Máquina de estados de OpenGL. De manera simplificada, es posible explicar el proceso con las siguientes etapas:

- **Evaluación**: De ser necesario, se evaluan funciones polinomiales que definirán elementos en la visualización, tratando de igualar las curvas y la geometría de las superficies.
- **Operaciones por vértices**: En esta fase se realiza el ensamblado de primitivas, realizando las transformaciones, ilumnaciones y discernimiento entre elementos que se mostrarán y no.
- **Rasterización**: Acá los polígonos son representados en conjuntos de pixeles, de manera que puedan ser desplegados en un medio de salida digital.
- **Operaciones por fragmentos**: Se realizan actualizaciones de valores que estaban almacenados junto a los nuevos valores, combinaciones de colores, entre otros.

- **Frame Buffer**[3]: En esta última etapa se almacenan los elementos en pantalla calculados previamente en memoria.

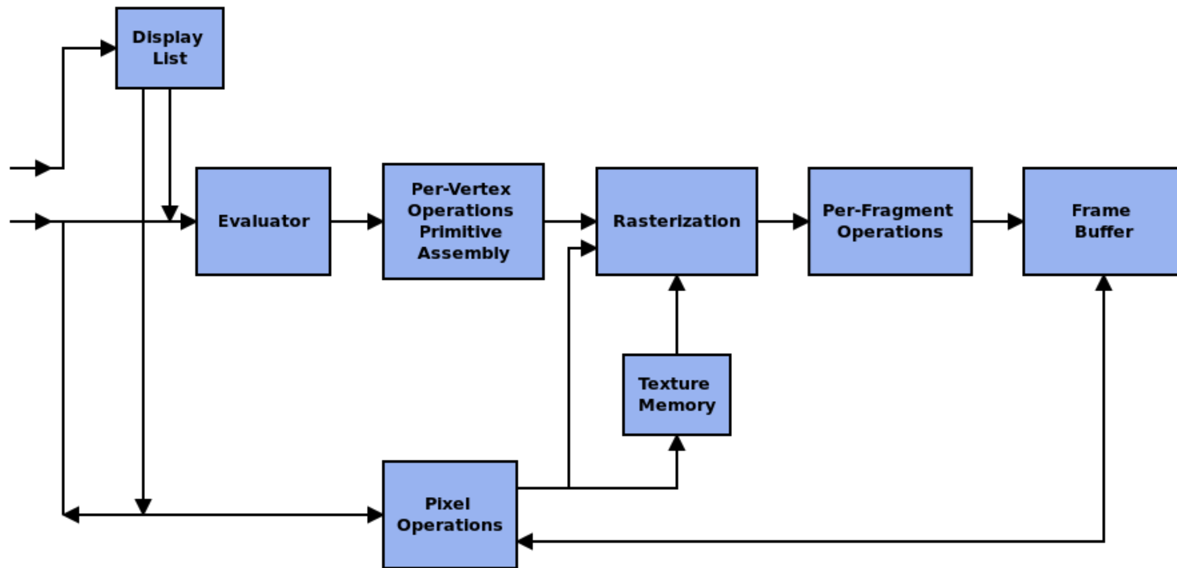

Figura 2.13: Máquina de estados de OpenGL

En la Figura 2.13 es posible ver el proceso descrito previamente.

Todo lo explicado anteriormente hace entender de que OpenGL es una API basada en procedimientos de bajo nivel, obligando al usuario a especificar los pasos exactos para renderizar los elementos a cambio de poder implementar cosas de niveles bastante complejos.

Esto funciona en conjunto con los **Shaders**, que están encargados de calcular transformaciones de vértices, coloreados y sombreados entre otras cosas. GLSL o OpenGL Shading Language es un lenguaje de alto nivel basado en una sintaxis similar a C. Con este lenguaje es posible calcular de manera más eficiente y veloz a través de GPU ciertas transformaciones gráficas a nivel de vértices, fragmentos, geometrías, teselaciones, entre otros. Es importante destacar que este lenguaje es compatible con muchos sistemas operativos y hardware de diferentes marcas pero no puede funcionar de manera solitaria, debe ir en conjunto con OpenGL.

2.2.3. Interoperabilidad entre CUDA y OpenGL

Existe la capacidad de utilizar CUDA y OpenGL en conjunto para hacer procesamientos gráficos, permitiendo que CUDA lea la información escrita por OpenGL o escriba información para el consumo de este. Para ello debe establecerse un protocolo de comunicación entre ambas partes.

Primero, por el lado de CUDA es necesario registrar un recurso antes de poder realizar los enlaces. Esta operación de por sí es costosa, por lo que se registra solo una vez por elemento.

[3] Un Frame Buffer es una categoría de dispostivos gráficos en la cual se representan cada uno de los pixeles de la pantalla como ubicaciones en la memoria RAM

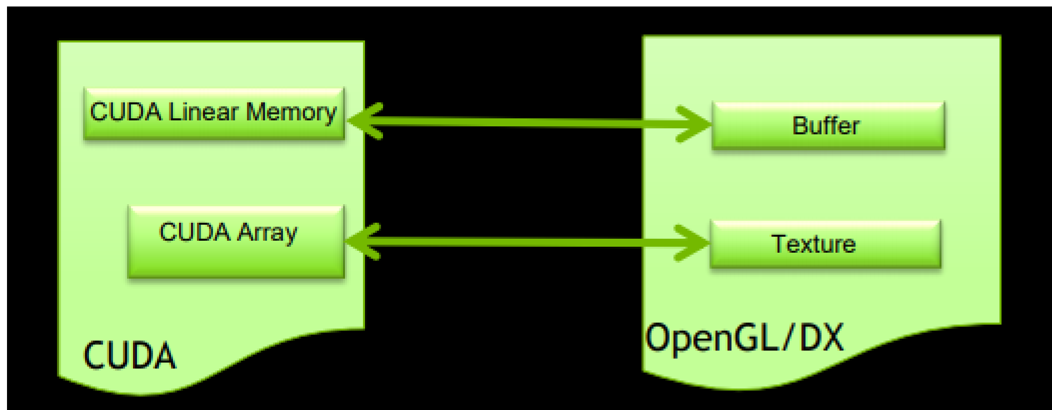

Figura 2.14: Asociación de data entre CUDA y OpenGL

Una vez registrado el recurso, es posible enlazar (mapear) los diferentes objetos a compartir entre ambos contextos por cada cálculo de imagen o frame, siguiendo un patrón como el que se presenta a continuación:

- Se registra el recurso a utilizar (en el contexto actual).
- Por cada elemento de OpenGL que se necesite para realizar los cálculos en CUDA se realiza un mapeo entre ambas instancias, entregando (a través de una copia de memoria) la data de un lado a otro.
- Se realizan los cálculos en CUDA. En caso de que se hagan cálculos en paralelo es necesario realizar una sincronización de los hilos utilizados. Esto hace que el resultado sea efectivo siempre y cuando todos hayan terminado.
- Se entregan los resultados desde CUDA a OpenGL copiando en memoria la data. Luego se desmapean las instancias pertinentes.
- Se realizan los cálculos (usualmente a nivel gráfico) en OpenGL.

Esto puede ser ilustrado en la Figura 2.15

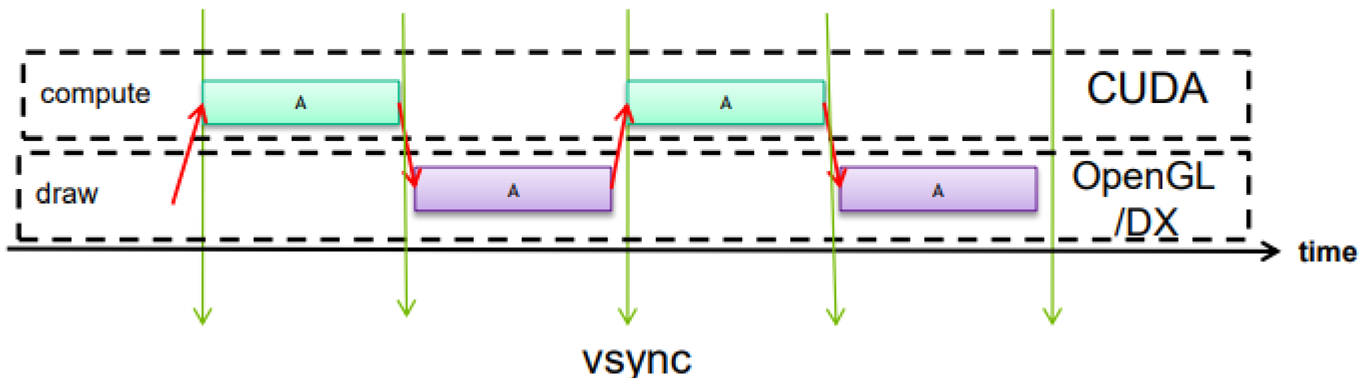

Figura 2.15: Interoperabilidad entre CUDA y OpenGL serial

Capítulo 3

Análisis y diseño de las librerías CLEAP y TIUQUE

En este capítulo se hablará de las librerías CLEAP y TIUQUE, su funcionamiento, estructuras de datos y comunicación con el fin de dar una base para explicar el diseño e implementación del nuevo algoritmo en función de lo existente.

3.1. Introducción

CLEAP[9] es una librería open source desarrollada en C++, que utiliza los principios de la programación en paralelo en GPU para mejorar los tiempos de reestructuración de las mallas de triángulos en 2D para que sean Delaunay a través del método *Edge-Flip* (o al menos localmente Delaunay, vale decir, tiene triángulos que no cumplen con la condición Delaunay pero el conjunto lo hace). Para ello, utiliza CUDA como platafoma de paralelismo.

Todas las funciones que *Device* opera en paralelo en los hilos que hayan sido asignados se conocen como *Kernel*. Dicho esto, CLEAP se basa principalmente en 2 conjuntos de kernel: el primero corresponde a **kernel utils**, el cual contiene operaciones del tipo producto cruz, producto punto, cálculos vectoriales, entre otros, para realizar los algoritmos necesarios; y el segundo corresponde a **kernel delaunay transformation**, conjunto que realiza las verificaciones de que un triángulo cumpla con las condiciones de Delaunay, realiza los *flips* en las aristas y mantiene los índices y valores asociados en el cálculo paralelo del diagrama.

Esta información es contenida en varias estructuras, pero son tres las principales para el manejo de la malla de triángulos en *Host*. El primero es el vértice, que contiene vectores con la ubicación espacial, normal y color asociados a un punto de la nube. La siguiente es una arista, que mantiene el índice de los vértices desde donde comienza hasta donde termina, los triángulos a los cuales pertenece y el índice del vértice opuesto en cada triángulo para facilitar la operación de *flipping* en caso de que no cumpla las condiciones. La última estructura principal es la malla, que mantiene un arreglo de vértices y de aristas, la cantidad de vértices, aristas y triángulos en la malla, y la referencia a la estructura simil de la malla

pero por el lado de *Device*. Por el lado de este último, existe una única estructura que contiene arreglos con toda la información contenida en las estructuras de *Host*.

3.2. Estructuras de datos

CLEAP se basa principalmente en cuatro estructuras que especificaremos a continuación:

3.2.1. Vértices

Una de las primeras estructuras y más básicas dentro de CLEAP es la que corresponde a los vértices. En ella se almacena la información obtenida del archivo .obj, ordenándola en:

- **v**: contiene las coordenadas (x, y, z, α) de un vértice. Es representado a través de un elemento de tipo float4.
- **n**: representa el vector normal de un vértice. Se almacena a través de un elemento de tipo float4.
- **c**: guarda la representación RGB del color asociado al vértice. De igual manera que los otros elementos, está contenido en un espacio tipo float4.

Estos elementos componen la estructura llamada **cleap_vnc_data**. Esta forma de almacenamiento simplifica la interoperabilidad con OpenGL y Shaders ya que son compatibles con los mismos.

3.2.2. Aristas

Esta tiene como nombre **cleap_edge_data** y contiene información con respecto a las aristas que forman los triángulos en la malla. Se basa en cuatro elementos:

- **n**: este elemento referencia el índice donde se encuentran las coordenadas de los vértices de ambos extremos de la arista. Es representado a través del tipo int2.
- **a**: Índice referencial a uno de los triángulos de los cuales la arista pertenece. Esta asociación apunta al índice de ambos vértices en el arreglo de triángulos completo. Está representado por un elemento de tipo int2.
- **b**: Índice referencial a uno de los triángulos de los cuales la arista pertenece. De igual manera que **a**, esta asociación es hacia los índices donde se encuentran ambos vértices en el arreglo de triángulos, pero este valor puede ser -1, haciendo alusión a que uno de los lados de la arista no está contenido en un triángulo, vale decir, es una arista externa de la malla. Se representa mediante un int2.
- **op**: Indica el vértice opuesto de cada uno de los triángulos asociados a la arista. A través de un tipo int2 utliza el primer elemento para referenciar el vértice opuesto del triángulo A y el segundo para el triángulo B.

3.2.3. Triángulos

Esta última es la encargada de agrupar y almacenar los índices de los vértices que conforman cada uno de los triángulos de la malla. Si bien no está definida como una estructura a nivel de código, es fácilmente representable a través de un arreglo de largo $3n$ con n siendo la cantidad de triánglulos existentes en la superficie, de manera tal que el acceso a los vértices de uno de los elementos es equivalente tomar tres elementos continuos desde el índice del triángulo triplicado.

Esto hace que la estructura y los datos sean compatibles con OpenGL.

3.2.4. Mallas

Todas las bases nombradas anteriormente son contenidas en una gran estructura llamada **cleap_mesh** y son manejadas a nivel de CPU, manejando de manera centralizada la información. Por otro lado, al trabajar con CUDA estas estructuras no son entendibles por el procesador, por lo que existe una estructura paralela a la existente en *Host* llamada **cleap_device_mesh**, que maneja estructuras de tipos básicas para operar a nivel de *Device*. En la Figura 3.1 es posible observar el intercambio de información entre *Host* y *Device* con el fin de compartir los datos a trabajar, destacando en rojo el manejo de los vértices y en verde el de las aristas.

3.3. Algoritmos en CLEAP

Tal como se dijo anteriormente, CLEAP es una librería que busca mejorar los tiempos de obtención de las triangulaciones de Delaunay en mallas de triángulos 2D a través del uso de GPU. Por simplicidad, se separará el proceso en dos fases de desarrollo:

- **Creación de estructura y fill-in de datos**, donde se hablará de los tipos de archivos que recibe, cómo los procesa y arma las instancias de las estructuras previamente nombradas. También se expondrán las conexiones con los buffers de OpenGL y traspaso de memoria con CUDA.
- **Transformación a Delaunay** en la cual se utilizarán los datos previamente armados para obtener la triangulación de Delaunay (o la más cercana a ella) de la nube de puntos entregada, haciendo uso de multiples hilos de la GPU.

3.3.1. Creación de estructuras y fill-in de datos

Para comenzar es necesario explicar el tipo de archivo que recibe, el cuál es de formato **OFF**.

```
struct _cleap_mesh {
                cleap_vnc_data vnc_data;
                cleap_edge_data edge_data;
                GLuint* triangles;
                int vertex_count, edge_count, face_count;
                float3 max_coords, min_coords;
                int processed_edges, wireframe, solid;
                cleap_device_mesh *dm;
                CLEAP_RESULT status;    // important flag!!
};
```

```
struct cleap_device_mesh {
        struct cudaGraphicsResource *vbo_v_cuda, *vbo_n_cuda, *vbo_c_cuda, *eab_cuda;
        GLuint vbo_v, vbo_n, vbo_c, eab;
        int2 *d_edges_n, *d_edges_a, *d_edges_b, *d_edges_op;
        int *d_trirel, *d_trireservs, *d_listo;
        CLEAP_RESULT status;
};
```

Figura 3.1: Comparación de estructuras asociadas a la malla. En el recuadro superior se encuentra la estructura a nivel de CPU, mientras que en el inferior es a nivel GPU

OFF (Object File Format) es un formato orientado a describir objetos en 2 y 3 dimensiones. Su contenido consiste en una primera linea donde se especifica la cantidad de vértices, caras y aristas que componen el polihedro, seguida de la lista de coordenadas de los vértices (x, y, z, w). Cuando la lista de vértices termina, se describen las caras de la malla, especificando la cantidad de vértices que utiliza y cuáles son. En estos último es opcional agregar un color asociado a la cara.

Toda esta información es procesada por CLEAP en 3 métodos, dejando la información estructurada como se muestra en la 3.2:

Carga de malla CPU

Tal como su nombre lo indica, el método **_cleap_host_load_mesh** se encarga de recolectar la información de la malla y almacenarla en el dispositivo *Host* (CPU). El método recibe un archivo .off que contiene la malla y una estructura **_cleap_mesh** donde se indexarán los datos. El proceso consiste en una lectura línea a línea del archivo recibido de la siguiente forma:

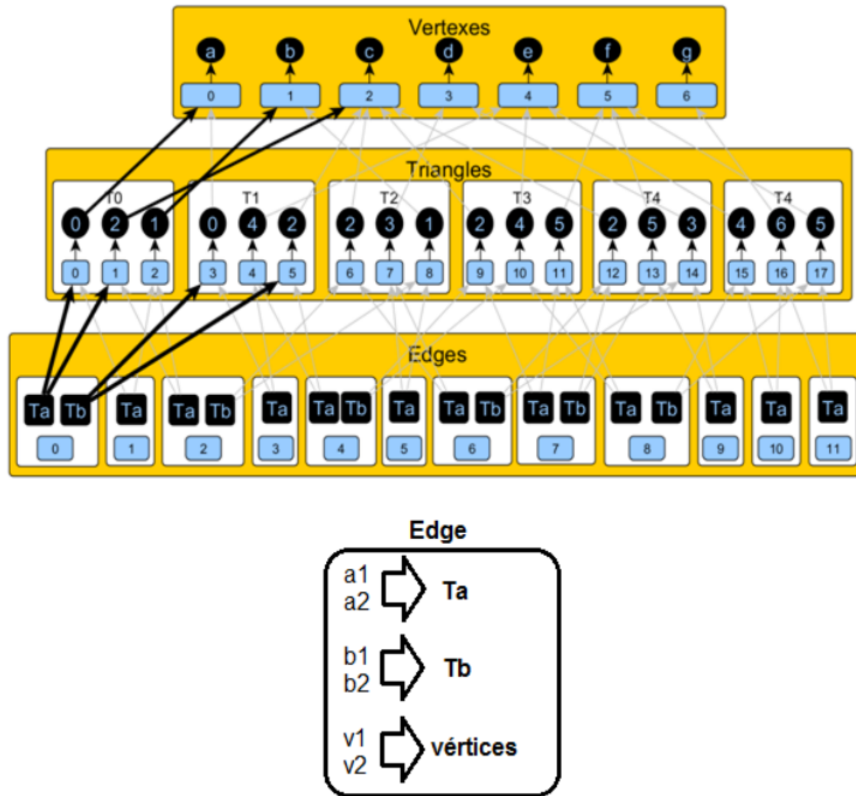

Figura 3.2: Ejemplo de estructuras, arreglos y punteros de una malla de triángulos.

- Se revisa si el tipo de archivo es el correcto, de lo contrario se levanta un error.
- Se busca la primera línea del archivo que no sea un comentario. Esta contiene, tal como se explicó anteriormente, la cantidad de vértices, caras y aristas. Esto se guarda en la estructura como **vertex_count**, **face_count** y **edge_count** respectivamente.
- Se inicializan los arreglos que en la estructura contendrán la infomación de triángulos, así como los arreglos de la estructura **cleap_vnc_data** contenida en la de la malla.
- Cada línea siguiente del archivo es integrada a la lista de vértices de la estructura de malla. Además, se verifica si este vértice corresponde en alguno de los ejes a ser el máximo o mínimo. Esto se repite una cantidad de veces igual a la cantidad de vértices descrita en el comienzo.
- Finalmente, se llama al proceso de generación de aristas que se explica más adelante.

Al finalizar, retorna una señal de éxito, de manera tal que se pueda continuar con la ejecución desde donde se llamó a la función de carga.

Generación de aristas

En este método la librería obtiene la información con respecto a los triángulos y aristas existentes en la malla por cargar, almacenando la información a nivel únicamente de *Host* (CPU). Éste recibe como parámetros la malla de tipo **_cleap_mesh** y el archivo **.OFF** e

inicializa un hash con hashes como valores (ver 3.3). A este hash lo llamaremos **root_hash** y a cada valor de este **hit**.

El proceso consiste en un ciclo de una cantidad de veces equivalente a la cantidad de caras (**face_count**) especificadas en la malla. Por cada ciclo se lee la línea correspondiente desde el archivo, realizando lo siguiente:

- Se verifica si el polígono a describir corresponde a un triángulo, vale decir, el primer número en la línea que se lee debe corresponder a 3. Si esto no se cumple el método arroja un error informando de que existen otro tipo de polígonos en la malla.
- Se almacenan en el arreglo **triangles** de la malla los índices que se leen desde la línea. Estos se guardan de manera que los índices estén ubicados en las posiciones $(i * 3)$, $(i * 3) + 1$ e $(i * 3) + 2$, con i siendo el número del ciclo (o el índice del triángulo).
- En este punto se genera un subciclo de 3 repeticiones donde por cada combinación entre los índices especificados anteriormente:
  - Se verifica cuál indice es mayor. Llamaremos a este valor como **i_mayor**. A su vez, su contraparte que compone a la arista en cuestión se le llamará **i_menor** y el índice que queda fuera de la arista pero forma parte del triángulo como **opuesto**.
  - Se obtiene el el hash **hit** correspondiente desde **root_hash** utilizando como llave el valor **i_mayor**.
  - Se verifica que exista una arista temporal en **hit** asociada a la llave **i_menor**.
    - En caso de que exista entonces rellenamos la información con respecto a segundo triángulo que contienen la arista procesada, vale decir, los índices que marcan la ubicación en el arreglo **triangles** de los 3 vértices.
    - En caso de que no exista, se genera una arista temporal que contiene el índice de los vértices, los índices de ubicación en el arreglo **triangle** del triángulo en cuestión, así como se inicializan como -1 los mismos elementos nombrados previamente pero correspondientes al segundo triángulo que contiene esta arista. Si esta fuese una arista externa, vale decir, no tiene un segundo triángulo que la contenga, los valores permanecerán como -1.
- Se calculan las normales correspondientes a la cara en cada vértice.

Finalmente, al terminar el ciclo completo se almacenan las aristas temporales en la malla como **edge_data**.

Carga de malla GPU

Tras haber formado toda la estructura de la malla a nivel de *Host* (CPU) es necesario entregar toda esta información al *Device* (GPU) que realizará los cálculos más complejos. Aquí se generan y enlazan los buffers, así como se reservan los espacios en memoria para operar con CUDA.

El método consiste en:

- Por cada objeto que se exponga de manera visual en TIUQUE se genera un buffer de

```
// the hash for edges
std::tr1::unordered_map<int, std::tr1::unordered_map<int, _tmp_edge> > root_hash;
std::tr1::unordered_map<int, _tmp_edge>::iterator hit;
```

Figura 3.3: Hashes utilizados para la indexación de aristas. **root_hash** almacena un hash **hit** con una llave equivalente al mayor índice de los vértices que conforman la arista. Por otro lado, **hit** es un hash que contiene aristas temporales indexadas por el menor índice de los vértices de las mismas.

OpenGl. Este buffer se enlaza tanto con el arreglo existente en *Device* como en *Host*. Es posible observar un ejemplo de esto en la 3.4, en donde se inicia creando el buffer, ligando la data y registrándolo en los buffers de CUDA.

- Por cada arreglo o elemento existente en la malla se reserva la memoria con su tamaño respectivo (a través del llamado a *cudaMalloc*).
- Luego de reservar todos los espacios, se inicializan los datos de éstos a través de una copia de los datos relacionados a nivel de *Host*. Esta operación se ejecuta a través del comando *cudaMemcpy*.
- Al terminar todo este proceso, se llama a **cleap_paint_mesh**, que se encarga de visualizar los buffers calculados previamente.

Finalmente, retorna una señal de éxito.

```
// CLEAP::DEVICE_LOAD:: vbo vertex data
glGenBuffers(1, &dmesh->vbo_v);
glBindBuffer(GL_ARRAY_BUFFER, dmesh->vbo_v);
glBufferData(GL_ARRAY_BUFFER, size, 0, GL_DYNAMIC_DRAW);
glBufferSubData(GL_ARRAY_BUFFER, 0, size, m->vnc_data.v);
glBindBuffer(GL_ARRAY_BUFFER, 0);
err = cudaGraphicsGLRegisterBuffer(&dmesh->vbo_v_cuda, dmesh->vbo_v, cudaGraphicsMapFlagsNone);
if( err != cudaSuccess )
        printf("CLEAP::device_load_mesh::cudaGraphicsRegisterBuffer::vbo_p:: %s\n", cudaGetErrorString(err));
```

Figura 3.4: Generación de buffer de los vértices de la malla. En verde se destacan los arreglos asociados a *Device* y en rojo a *Host*

3.3.2. Transformación de Delaunay

Una vez cargados los datos en *Host* y *Device* es posible proceder a transformar la malla de triángulos a una triangulación de Delaunay. Dado que este proceso afecta a la parte visual y datos en ambos niveles es necesario primero ligar los recursos y mapear los punteros. Para ello se utlizan las funciones **cudaGraphicsMapResources** y **cudaGraphicsResourceGetMappedPointer**. Luego, se calcula el tamaño de las grillas, hilos y bloques que se usarán en GPU en función de la cantidad de caras y aristas que tiene la malla, como se puede observar en 3.5.

```
int block_size = CLEAP_CUDA_BLOCKSIZE;
dim3 dimBlock(block_size);
dim3 dimGrid((cleap_get_edge_count(m)+block_size-1) / dimBlock.x);
dim3 dimBlockInit(block_size);
dim3 dimGridInit((cleap_get_face_count(m)+block_size-1) / dimBlock.x);
```

Figura 3.5: Cantidad de bloques, grillas e hilos que se utilizarán para la transformación de Delaunay

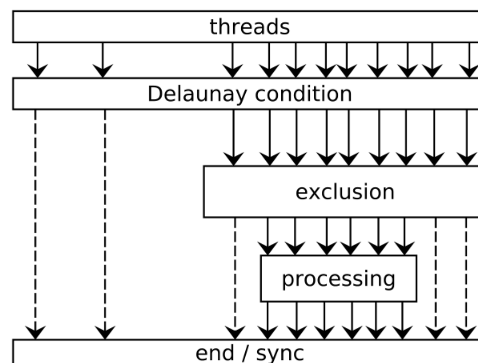

Figura 3.6: Hilos utilizados en la transformación de Delaunay. Figura tomada del artículo ”A Parallel GPU-based Algorithm for Delaunay Edge-flips” [1]

Una vez seteadas las bases para comenzar, se procede a realizar un ciclo que finalizará una vez la malla haya sido transformada a Delaunay. En cada ciclo es posible dividir la ejecución en dos fases de manera serial, donde cada una realiza procesos en paralelo: **Detección, Exclusión y Procesamiento** y **Reparación**. Al comenzar y finalizar cada una de estas es necesario realizar una sincronización de los hilos que CUDA está manejando, de manera que si alguno de estos aún está realizando cálculos se le espere hasta que termine. En la 3.7 es posible observar el estado inicial de una malla de triángulos, pasando por los pasos descritos a continuación en la 3.9 y 3.10

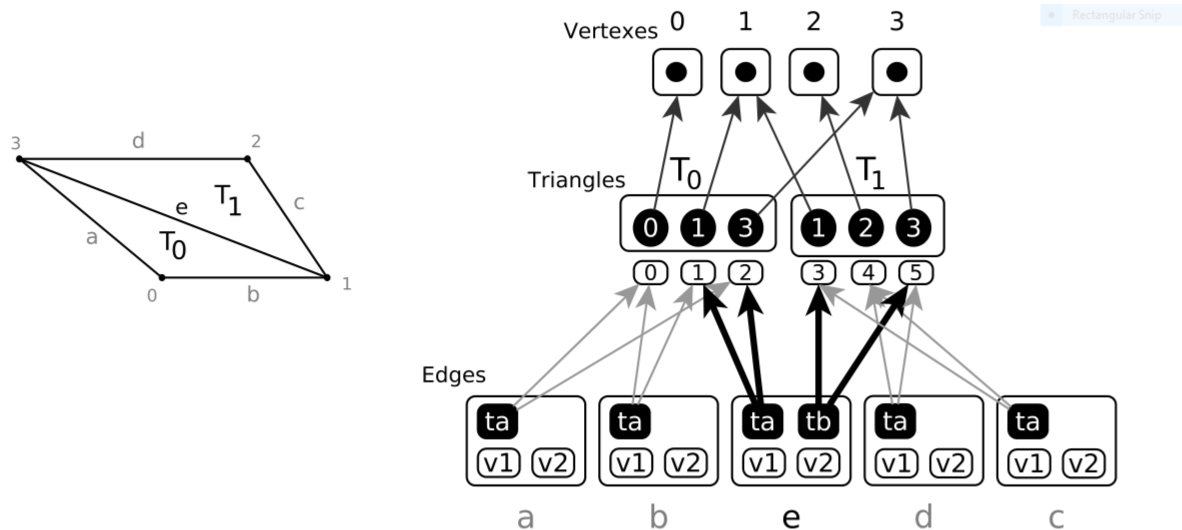

Figura 3.7: Malla de triángulos previa y su estructura previa al proceso de transformación de Delaunay. Se puede apreciar que la suma de $\measuredangle ab$ y $\measuredangle c$d es mayor que π, por lo que no cumple con la condición de Delaunay.

Detección, Exclusión y Procesamiento

Esta fase contiene tres pasos: detección de las aristas que no cumplen con la condición de Delaunay, exclusión de las aristas que no pueden ser modificadas en paralelo y procesamiento de las aristas que sí pueden.

Durante la detección a cada hilo en CUDA se le asigna una arista de la malla completa. Si la arista es externa, vale decir, tiene solo un triángulo que la contiene es ignorada. De lo contrario, se procede a verificar que cumpla con la condición de Delaunay calculando los ángulos opuestos a la arista. Si la suma de ambos es mayor a π entonces es necesario modificarlo, de lo contrario el proceso termina.

Luego, durante el paso de exclusión se marcan todos los triángulos que van a ser modificados. Este paso se hace asegurando atomicidad en la ejecución a través del uso del método **atomicExch**, tomando el control del arreglo que guardará las flags de que un triángulo ha de cambiar. Si durante este proceso otro hilo trata de marcar un excluir un triángulo que previamente fue marcado su ejecución se verá interrumpida, terminando el proceso para éste como es el caso del hilo a en la 3.8.

Finalmente, para todos aquellos hilos que siguen en ejecución (vale decir, no cumplen la condición de Delaunay y ambos triángulos que lo contienen fueron debidamente excluídos) se procede a realizar la modificación. Esta consiste en un edge-flip, cambiando los vértices de las aristas, así como la indexación de los vértices del triángulo, como es posible ver en la 3.9.

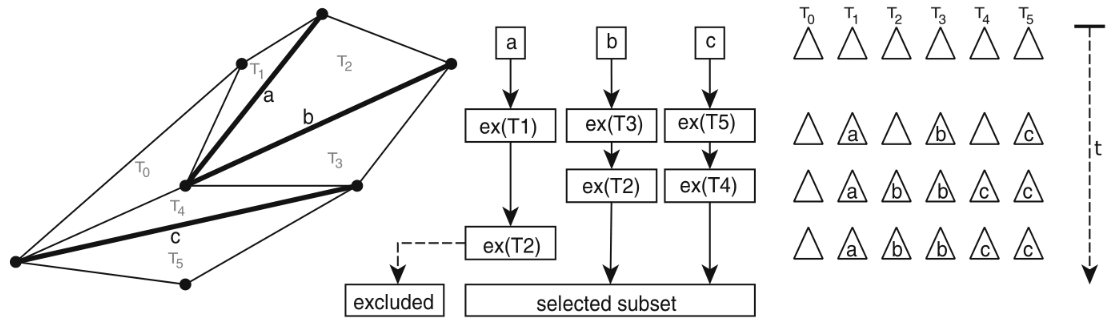

Figura 3.8: Ejemplificación de exclusión de triángulos en GPU

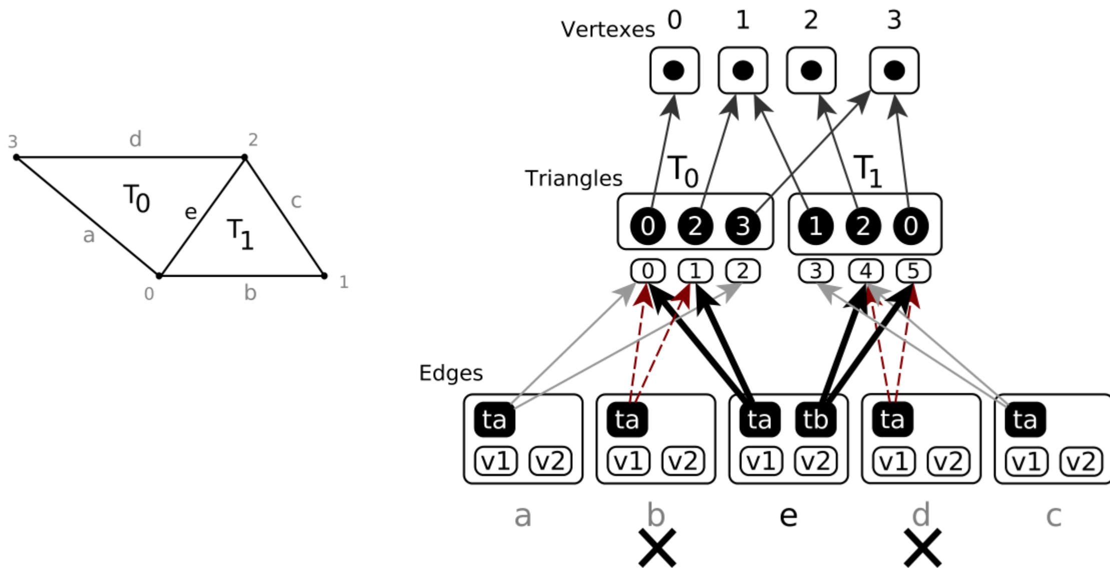

Figura 3.9: Malla de triángulos luego de ejecutar el paso de Detección, Exclusión y Procesamiento. Es posible observar que la arista e ha sido girada, modificando su información. A su vez, las aristas b y d han quedado inconsistentes.

Reparación

Tras la ejecución del paso anterior es posible que algunas referencias o información almacenada en las aristas sea inconsistente, apuntando a pares de triángulos que no necesariamente existen. Esto es fácilmente identificable verificando si los vértices indicados a través de la indexación de triángulos que contienen a la arista en cuestión y los vértices que realmente conforman la arista son equivalentes. En caso de que no lo sean, es necesario realizar una reparación de la información contenida modificando lo índices almacenados en arreglo de triángulos.

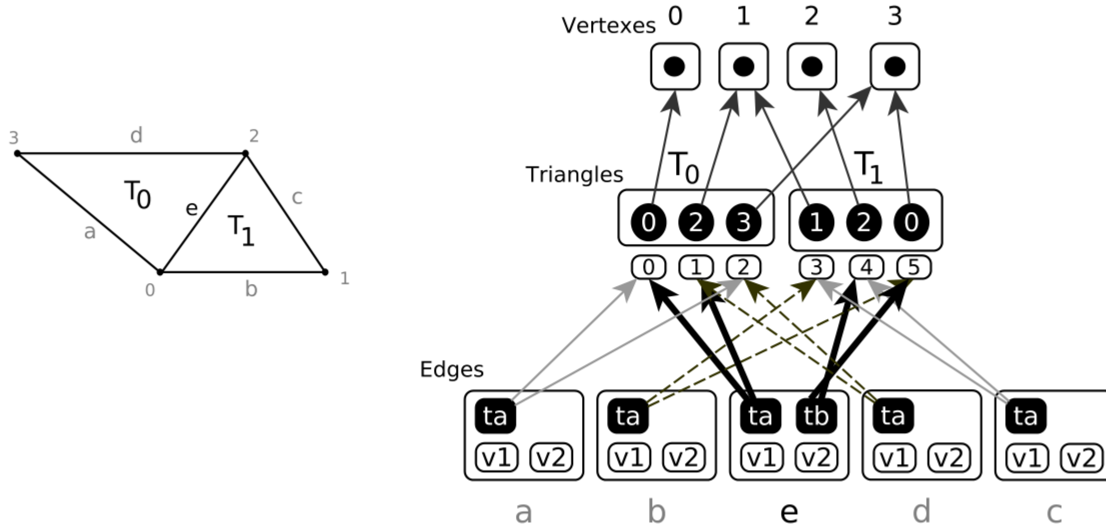

Figura 3.10: Malla de triángulos tras realizar la reparación de las aristas b y d

3.4. Funcionalidades de TIUQUE

TIUQUE es la interfaz gráfica de CLEAP programada en C++. Con esta, es posible visualizar y ejecutar de manera más interactiva los procesos que la librería contiene. La interfaz consiste principalmente en un lienzo donde se mostrará la malla de triángulos y los resultados de las transformaciones de Delaunay, una barra superior donde es posible abrir y guardar una malla y una barra inferior que permite modificar la visualización y forma de transformar la malla en una de Delaunay, como se puede observar en 3.11.

Para poder hacer uso de esta herramienta es necesario comenzar cargando un archivo de tipo **OFF**. Al hacer esto, Tiuque generará una instancia de la clase **Mesh** que internamente manejará un puntero a la malla en host de CLEAP **cleap_mesh** e inicializará el proceso de carga de datos explicado anteriormente. Una vez este proceso finaliza, Tiuque establece una ubicación para la malla y la cámara en el lienzo y redibuja la información contenida en los buffers de OpenGL. En la 3.12 se puede observar una malla cargada en Tiuque.

Los botones "Solid" y "Wireframe" cambian la visualización de la malla coloreando o haciendo invisibles las caras o aristas respectivamente, como en 3.13 se muestra.

Por otro lado, se encuentran los botones "2D MDT" y "3D Surface MDT" , que comienzan el proceso de transformación de la malla de triángulos a una que cumpla con la condición de Delaunay. El resultado del proceso se dibujará en el lienzo una vez se termine de calcular, como se observa en 3.14. En conjunto con esto, existe un checkbox etiquetado como "Educational Mode" que al estar activo la transformación se mostrará ciclo a ciclo, mostrando los cambios en la malla (de manera visual) a medida que se vuelve a presionar el botón correspondiente (2D MDT o 3D Surface MDT).

Finalmente, tiene las opciones de limpiar los buffers eliminando la visualización existente y de salir del programa, donde la librería libera toda la memoria solicitada previamente y cierra la ventana.

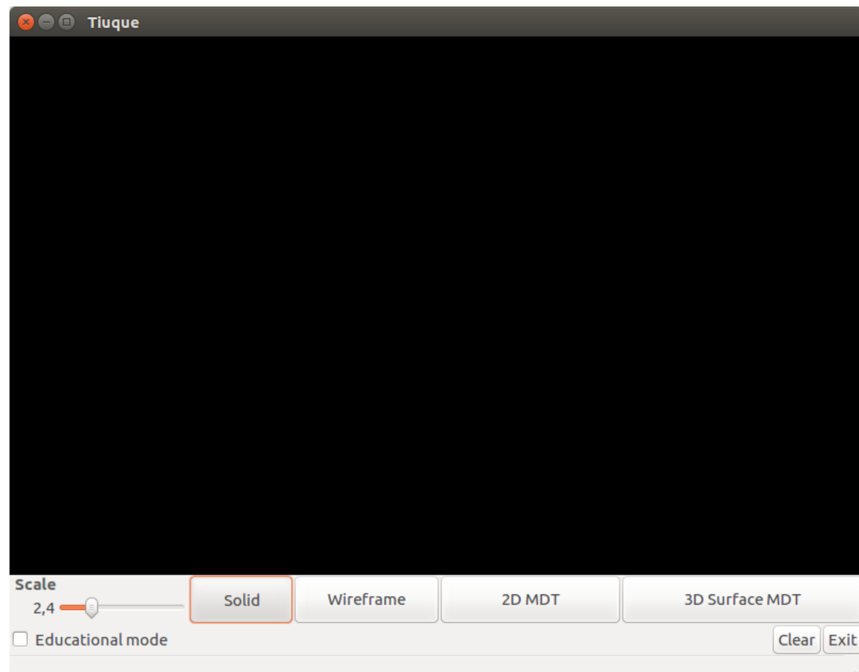

Figura 3.11: TIUQUE: interfaz gráfica que permite utilizar los métodos de CLEAP

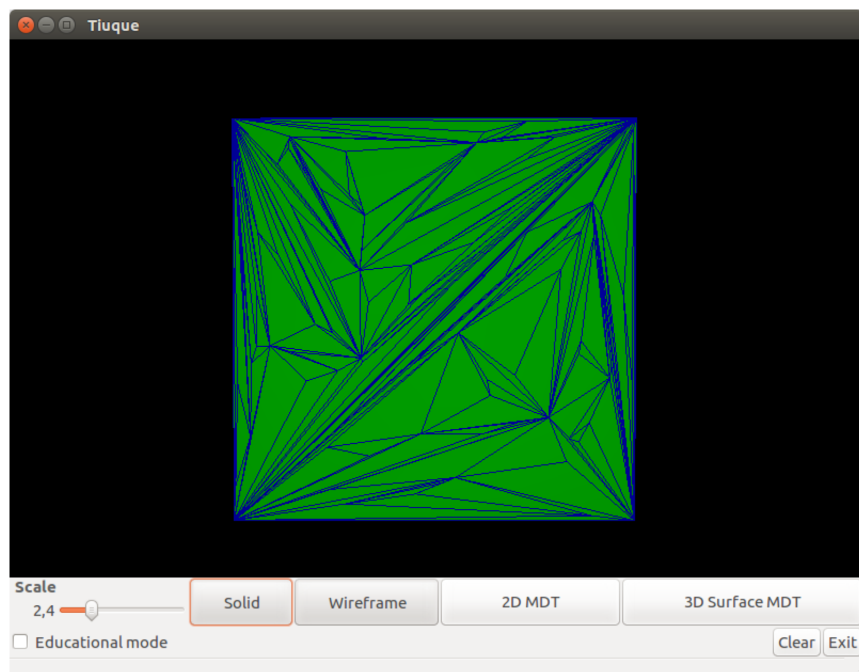

Figura 3.12: Malla de triángulos cargada en TIUQUE

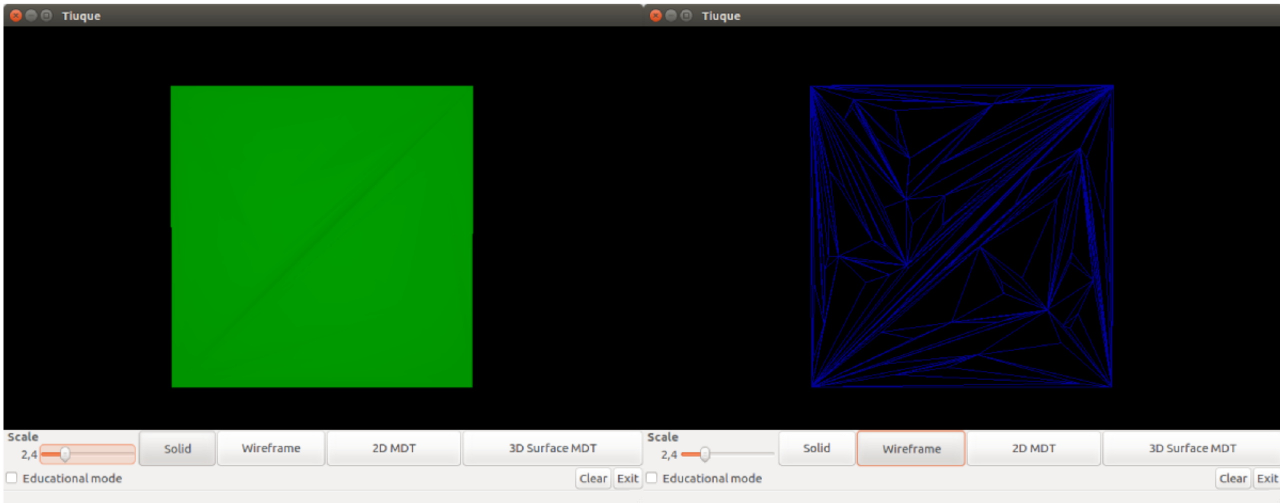

Figura 3.13: Tipos de visualización en TIUQUE. A la izquierda se observa solo la visualización de tipo sólida, mientras que en la derecha se exponen los *wireframes* de la malla.

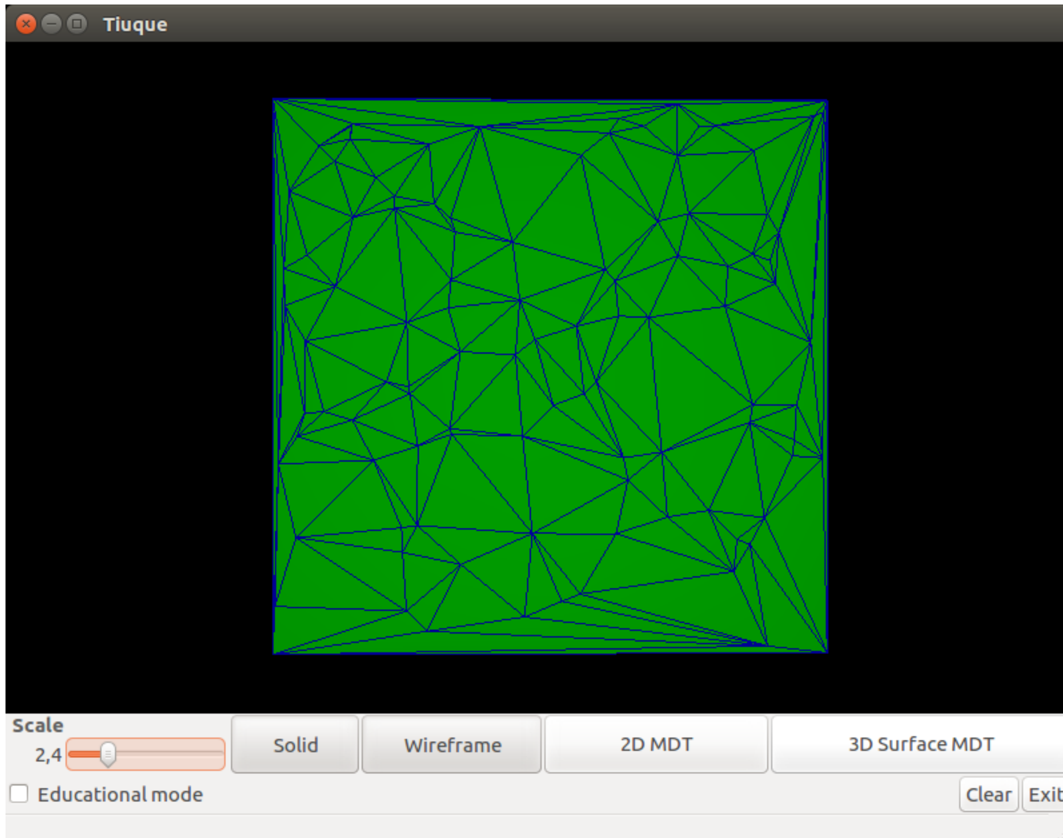

Figura 3.14: Malla de triángulos tras realizar la reparación de las aristas b y d

Capítulo 4

Diseño e implementación del cálculo del Diagrama de Voronoi

4.1. Integración

Dado que existe una dualidad entre una triangulación de Delaunay y su diagrama de Voronoi, es posible utilizar las estructuras ya existentes en la librería CLEAP junto a sus métodos para generar, almacenar y visualizar el diagrama de Voronoi. En la figura 4.1 se muestra de manera simplificada las estructuras agregadas y cómo serán implementadas en la librería existente. Es posible ver que las estructuras de Vértices y Aristas de Delaunay no han cambiado, pero ambas mallas deben ser modificadas para agregar el soporte de las nuevas estructuras.

4.2. Estructuras

Tal como se habló en las secciones anteriores, un diagrama de Voronoi se estructura a través de un conjunto de polígonos que tienen como centro uno de los vértices de la triangulación de Delaunay.

Para realizar la implementación del nuevo algoritmo en CLEAP es necesario construir nuevas estructuras que alberguen la información asociadada al diagrama de Voronoi. Es por ello que se han formado cuatro estructuras nuevas, las cuales se explican brevemente a continuación.

4.2.1. Circuncentros

Esta estructura define, tal como dice su nombre, los circuncentros de los triángulos de una malla de Delaunay. Esto forma los vértices del nuevo grafo, manteniendo la normal y color

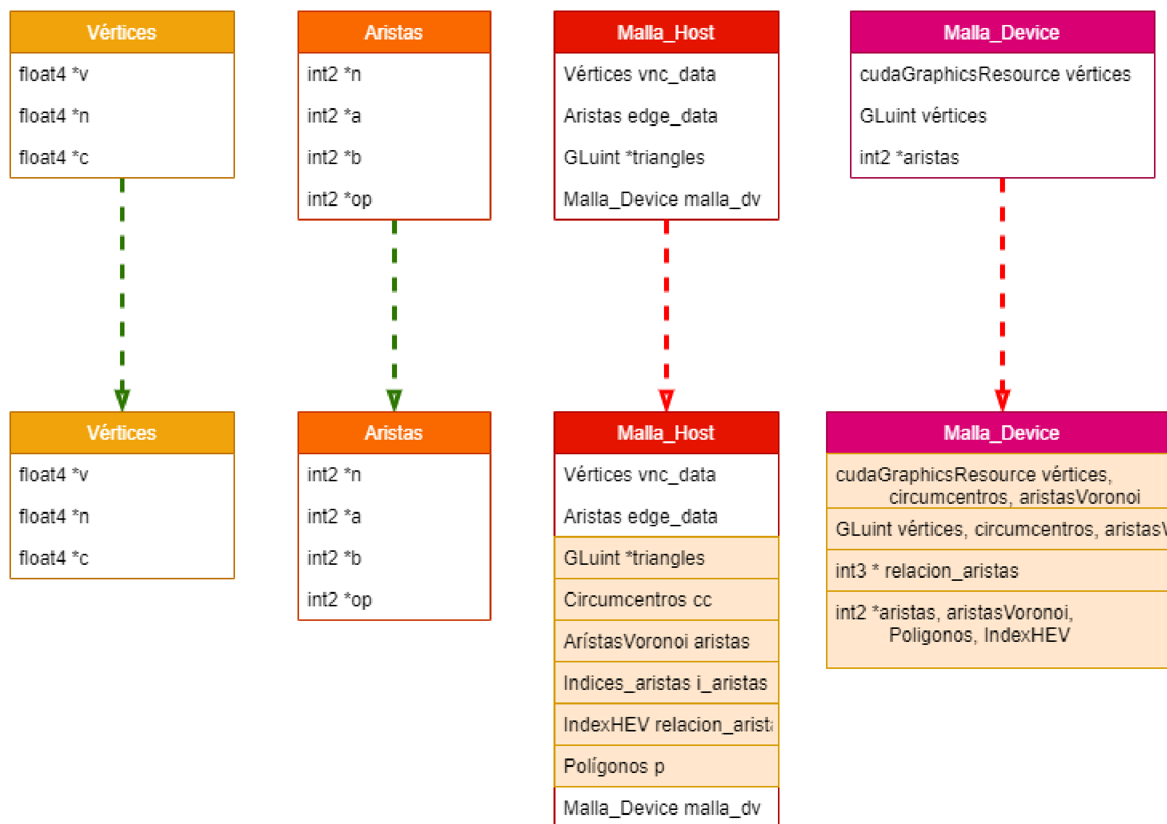

Figura 4.1: Simplificación de cambios en implementación a nivel de estructuras. Los elementos destacados son las estructuras y datos que se agregarán en la implementación

asociado a los vértices del triángulo original que lo rodean.

Se conforma a través de tres arreglos de tipo Float 4, uno para su ubicación espacial, otro para su normal y el último como color. Cada uno con un largo de la cantidad de triángulos existentes.

Por otro lado, contiene la posición, normal y color de información sobre los puntos medios (o sus proyecciones) de las aristas que no tienen dos triángulos asociados, vale decir, colinda con el exterior de la malla. Al igual que lo anterior, se conforma con tres arreglos de tipo Float 4, pero con un largo de la cantidad de aristas.

Por simplicidad, se asume que toda normal va hacia el exterior de la estructura o malla.

4.2.2. Aristas de Voronoi

En esta, se guardan las aristas que formarán los polígonos de Voronoi. Está directamente relacionada con las aristas de Delaunay en una razón de 1:1.

Se subdivide en dos tipos de aristas:

- Arista interna-interna: es aquella en que ambos extremos corresponden a un cirucun-

centro de Voronoi. Existe una por cada arista entre dos triángulos en el grafo original y conectan los cirucuncentros de los mismos.

- Arista interna-externa: en este tipo sucede que uno de los extremos "termina en el infinito". Esto sucede cuando se intenta unir el cirucuncentro de un triángulo que pertenece al borde de la malla. Se tienen tantos como aristas que no tienen asociado dos triángulos en una malla existan.

Cada uno de estos tipos de aristas están representados como arreglos de tipo Int2 de largo equivalente al número de aristas en el grafo inicial.

4.2.3. Índices-Aristas

Esta estructura relaciona los vértices de la teselación de Voronoi (cirucuncentros) con las aristas que terminan en ellos. En el caso de las aristas externas, solo las contabiliza para el vértice interno correspondiente.

Está basada en un arreglo de tipo Int3 del largo equivalente a la cantidad de cirucuncentros (o triángulos de Delaunay) presentes.

4.2.4. Relación aristas de Voronoi

Aquí, tal como indica su nombre, se busca conectar las aristas de Voronoi entre sí de manera que se le otorgue un sentido al polígono a formar. Esto se basa en la estructura Half Edge [10], de manera que una arista tiene dos sentidos (desde el vértice A al B y viceversa).

Está conformada como un arreglo que representa los índices de las aristas previas y siguientes de la analizada. Una arista se asocia a dos índices: uno se corresponde con el índice de Voronoi y el otro corresponde al mismo más la cantidad de aristas de la teselación. Si una arista viene o llega desde el "infinito" se considera como índice -1. Esta numeración la llamaremos IndexHEV.

Es un elemento de tipo Int2 con una cardinalidad del doble de la cantidad de aristas de Voronoi.

4.2.5. Polígonos

Esta última estructura relaciona la cantidad de aristas que conforman un polígono de Voronoi y la arista inicial (con su IndexHEV). En caso de ser un polígono cerrado se toma cualquiera de los índices como inicial, mientras que si es un polígono abierto se toma el índice de una de las aristas que llevan hacia el exterior.

Se basa en un arreglo de tipo Int2 de tamaño equivalente al número de polígonos de Voronoi que se forman, vale decir, el número de vértices de la teselación de Delaunay inicial.

4.3. Algoritmo

Una vez definidas las estructuras a utilizar, es posible explicar el algoritmo implementado como extensión de CLEAP. Es importante destacar que la implementación de esto no afecta los procesos que existen anteriormente, por lo que se opta por crear un kernel para GPU independiente del que maneja Delaunay.

A continuación se presentan 5 procedimientos que darán como resultado un diagrama de Voronoi utilizando la información ya existente en CLEAP para armarlo. De estos, es posible identificar dos secciones importantes:

Los primeros 3 procedimientos (identificación de cirucuncentros, indexación de aristas y proyección de puntos medios) son aquellos métodos encargados de calcular los vértices y aristas del nuevo diagrama, así como de almacenarlo en las estructuras correspondientes. Con esto ya es posible visualizar a través de TIUQUE el diagrama de Voronoi asociado a una malla.

El resto de métodos (Relación cirucuncentro-arista, half-edges y reconocimiento de polígonos) están enfocados en la identificación de los polígonos del diagrama y su exportación a un archivo **.OFF**.

4.3.1. Identificación de Cirucuncentros

El primer método dentro del algoritmo esta relacionado con los cirucuncentros. Dado que se tienen los vértices que corresponden a cada triángulo indexados es posible calcular el cirucuncentro asociado para cada uno. Como parámetros, se reciben los arreglos de vértices, triángulos que indica los vértices que corresponden a cada uno, el de cirucuncentros y la cantidad de triángulos en la malla.

Luego, de manera paralela en tantos hilos como triángulos hay se ejecuta lo siguiente se calcula el cirucuncentro de cada uno a través de la fórmula en la figura 4.1.

$$circumcenter = \frac{(c+a)}{2} + \frac{A \cdot B(C \times (A \times B))}{8\,(\|A \times B\|)^2} \tag{4.1}$$

Una vez obtenido el vector que representa la ubicación del cirucuncentro se almacena en el mismo índice del triángulo al que corresponde en la estructura de cirucuncentros.

4.3.2. Indexación de aristas

Tras calcular los cirucuncentros del diagrama se verifica cuales son aristas internas y externas mientras se indexan para su uso futuro. Usando los vértices, cirucuncentros, número de aristas e información de las aristas en sí se procede a ejecutar los siguientes pasos de manera paralela en tantos hilos como aristas existen en la malla:

- Por cada arista se verifica si tiene triángulos a ambos lados.
- Si existen dos triángulos que contengan la arista:
  - Se crea una arista de Voronoi **interna-interna**, espeficicando los índices de los cirucuncentros que une (procurando que el primer índice siempre sea el menor de ambos).
- En caso de no existan dos triángulos que contengan la arista, vale decir, es una arista externa:
  - Se contabiliza de manera atómica.
  - Se calcula el punto medio de la arista a la cual pertenece utilizando los vértices de la misma.
  - Se almacenan la ubicación espacial del cirucuncentro del triángulo al que pertenece y el punto medio en un arreglo donde los índices son el del cirucuncentro y la cantidad de triángulos más el número de la arista respectivamente.
  - Finalmente se crea una arista de Voronoi **interna-externa** la cual tiene como índices los lugares donde se almacenaron los datos anteriormente calculados.

4.3.3. Proyección de puntos medios

Como se había explicado anteriormente, las aristas internas-externas de un diagrama de Voronoi van desde un cirucuncentro de la triangulación de Delaunay hasta el infinito pasando por el punto medio de la arista del triángulo que no tiene otro triángulo asociado. Esto no es posible de replicar de manera simple al momento de trabajar con OpenGL ni almacenar la información, por lo que se opta por realizar proyección del punto medio. Esta proyección se realiza desde el cirucuncentro de la arista externa hasta el punto medio de la arista en cuestión, pero destacándola de manera visual con otro color, de manera que se entienda que esta es solo una representación finita, pero en la realidad es la recta que continúa desde el punto medio pasando por el cirucuncentro hasta el infinito. Esto también permite conservar los valores de índices, aristas y vértices de Voronoi para los cálculos siguientes del proceso.

4.3.4. Relación cirucuncentro-arista

Una vez se conocen las aristas internas, externas y cirucuncentros ya es posible armar visualmente el diagrama de Voronoi, pero no se puede almacenar ni diferenciar los polígonos a nivel de código.

El procedimiento de Relación cirucuncentro-arista tal como dice su nombre relaciona las aristas que llegan a un cirucuncentro en común. Para ello, utilizando las aristas internas y externas del diagrama de Voronoi, se asginan tantos hilos como aristas tenga. Cada uno de estos hilos inyectará información a la estructura **Índices-Aristas** de la siguiente forma:

Siendo i el índice del hilo que está operando (que es equivalente al índice de la arista), de manera atómica se realiza una operación de swap entre lo que está en el arreglo en la posición (i, x) y el primer cirucuncentro que une la arista.

- Si el valor que se recibe del arreglo es distinto de -1 (que es el valor con el que se inicializa), se repite la operación pero tomando el índice en la segunda posición, vale decir, (i, y). Si este tampoco es -1, se procede a intercambiar una última vez con lo que esté en la tercera posición (i, z).
- Si en cualquier momento el retorno es distinto de un índice la ejecución continúa.

Si la arista es de tipo interna, se repite el proceso pero usando el segundo cirucuncentro que une la arista. En cambio, si es de tipo externa la ejecución del hilo se detiene.

Todo esto se ve en las figuras 4.2 y 4.3.

La primera es el caso inicial del ejemplo en la cual es posible observar la triangulación de Delaunay en negro, la existencia de dos cirucuncentros en color rojo, cuatro proyecciones de punto medio en color verde y las aristas del diagrama de Voronoi en morado.

En la segunda figura tenemos el estado de la estructura de relación arista-cirucuncentro inicial y final, la asignación de hilos por cada arista (especificando en azul las aristas que son interna-interna y en verde las interna-externa) y el proceso de cada hilo, donde **swap** corresponde a la función atómica que toma como parámetros le índice de la arista que está trabajando (o que recibió del último swap), el índice del cirucuncentro en el cual debe agregar la información y la posición donde debería operar. El valor de retorno de la función está representado en un círculo. Dado que es una operación de exclusión las ejecuciones en otros hilos deben esperar a que se habilite la estructura a utilizar antes de operar (representado a través de líneas punteadas cuando está en espera y sólidas si está operando).

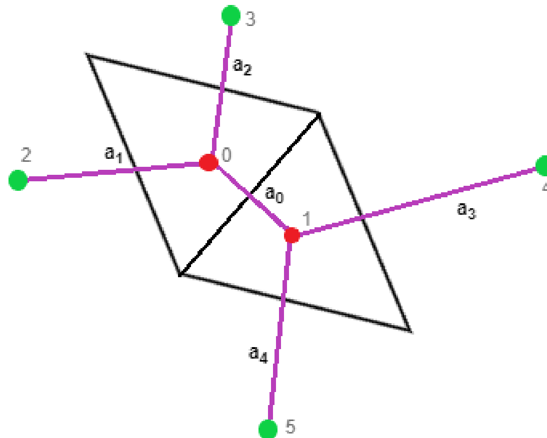

Figura 4.2: Triangulación de Delaunay y Diagrama de Voronoi para ejemplificación de proceso.

4.3.5. Half-edges y Reconocimiento de polígonos

Existen muchas formas de representar polígonos, dependiendo del enfoque que se quiera utilizar para ello. Dado que el objetivo es identificar los polígonos, poder exportarlo como texto y que el formato a entregar está enfocado en los vértices y aristas del objeto se tiene que

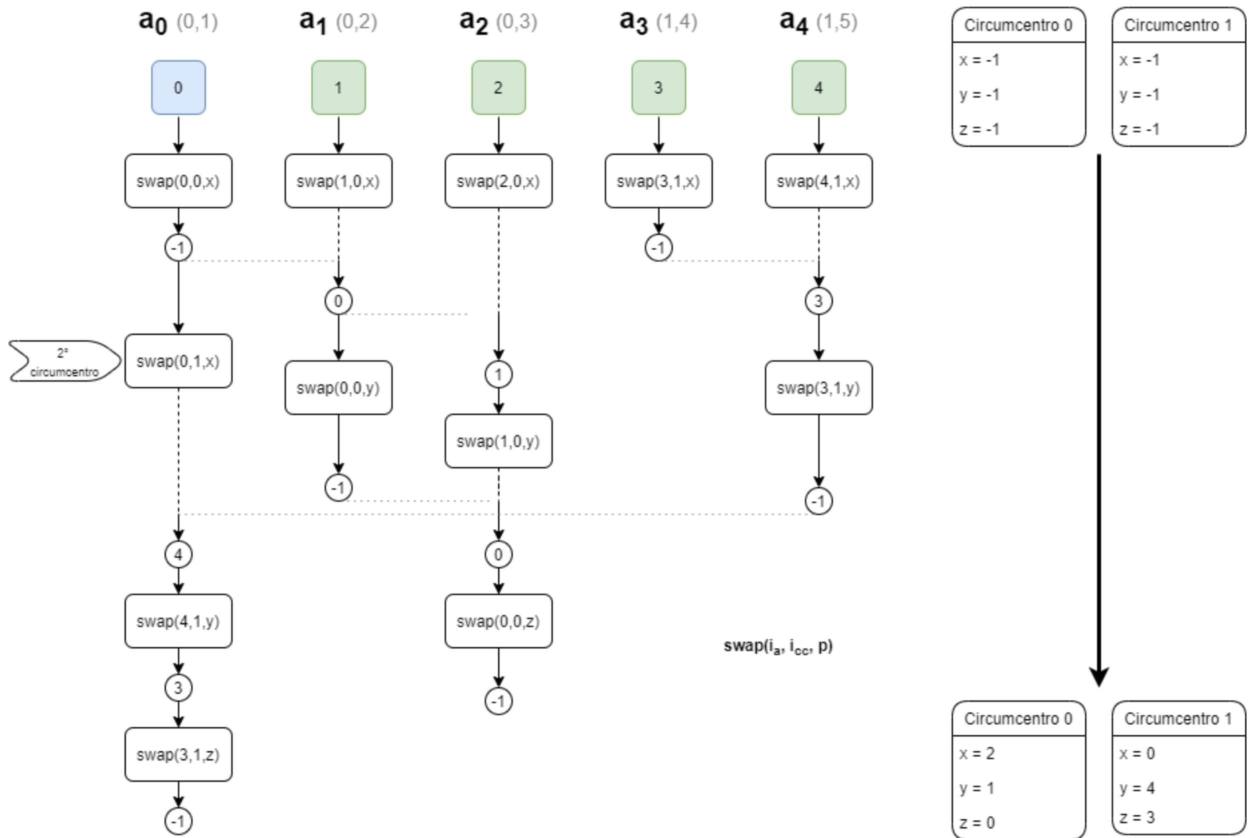

Figura 4.3: Proceso de swap en relación cirucuncentro-arista

utilizar algo más enfocado en estos elementos. Una de estas representaciones son los *Half-edges*. Estos consisten en subdividir las aristas en dos "a lo largo", haciendo que ambas aristas compartan los vértices desde y hasta donde llegan. Esta estructura permite darle un sentido de orientación y armar una cadena de *half-edges* que generan un polígono. Tal como se explicó anteriormente, la estructura **IndexHEV** busca conectar las *half-edges* del diagrama de Voronoi, de manera que una arista conoce su sucesor y previo, así como mantiene una relación de índice con su arista opuesta (aquella que tiene los mismo vértices pero sentido contrario).

Sumado a todo lo anterior, esta representación requiere de un espacio fijo, pudiendo solicitarse de manera anticipada a cualquier cálculo y volviéndola compatible con el trabajo en GPU que se está realizando.

Tras haber identificado las aristas que tiene cada uno de los cirucuncentros tenemos que asignar el sentido a cada una de estas mitades de manera que sea posible rearmar el polígono. Por simplicidad, para este método se busca que todos estos polígonos tengan la misma normal. Cada polígono tendrá contenido un vértice de la triangulación de Delaunay, por lo que tendrán asociados los índices. De igual forma, una arista estará asociada a su contraparte (su otra mitad) ya que cumple con que su índice es $i \mod n$ con n siendo el número de aristas en el modelo completo.

Para una mejor explicación, se presenta la figura 4.4 como un caso de ejemplo sobre el

cuál se explicará y ejecutará el método. En este se presenta una triangulación de Delaunay (en color negro) y su diagrama de Voronoi (en rojo y verde). Los círculos representan los vértices y cirucuncentros en colores negros y rojos respectivamente, mientras las aristas rojas y verdes son aristas internas-internas y aristas internas-externas respectivamente. Es importante señalar que las aristas de Delaunay y de Voronoi que se cruzan comparten índice pero su información está almacenada en diferentes estructuras.

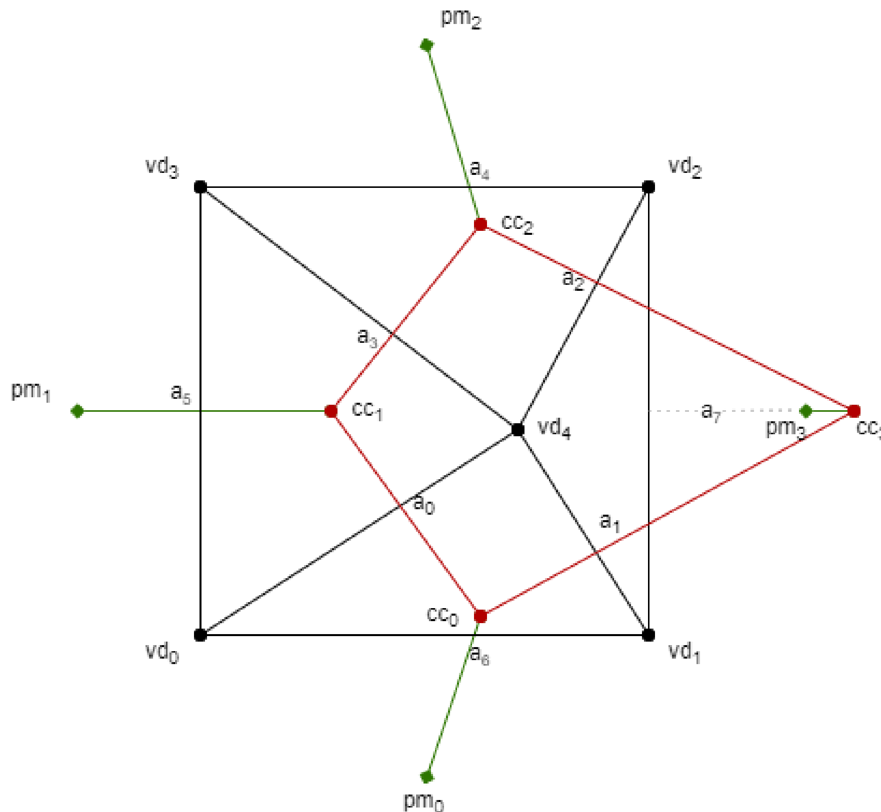

Figura 4.4: Caso inicial de ejemplificación de reconocimiento de polígonos

Utilizando los triángulos de Delaunay, aristas internas y externas de Voronoi, cirucuncentros y puntos medios el método calcula de manera paralela las relaciones entre aristas y los polígonos formados. Se asignan tantos hilos como el doble de aristas, ya que cada una equivale por dos partes. Cada hilo:

- Se le asigna un polígono. Si el hilo es menor al número de aristas entonces se busca el menor índice entre los vértices de Delaunay que tiene se oponen. De lo contrario se asigna el mayor. Esto es posible de ver en la figura 4.5, donde se muestran los $half$-$edges$ asociados a la arista 0. Se le asignará al hilo 0 como h_e_0 perteneciente al polígono del vértice 0, mientras que h_e_8 pertenecerá al vértice 4.
- Luego de saber el polígono, es necesario reconocer el sentido de la media arista y el cirucuncentro "pivot", vale decir, cuál será el cirucuncentro que comparten la arista y su siguiente. Para ello se hace una serie de productos cruz para tener conocimiento de la ubicación del cirucuncentro que son:
  - Se identifica si es una arista interna o externa. En caso de ser interna, es necesario verificar si el cirucuncentro está dentro o fuera del triángulo, primero conociendo

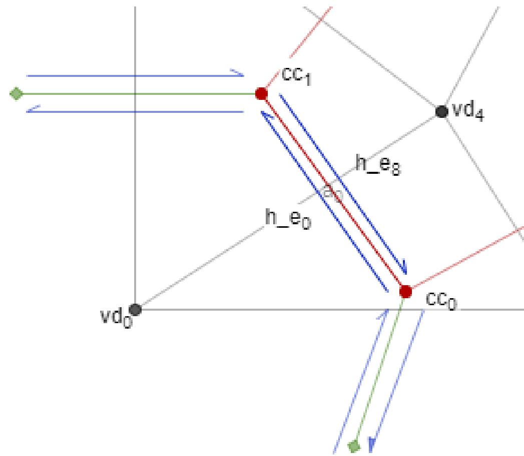

Figura 4.5: Caso inicial de ejemplificación de reconocimiento de polígonos

el sentido de las aristas del triángulo para luego hacer un producto cruz con el cirucuncentro. Si este producto cruz da negativo, significa que el cirucuncentro está afuera y se invertirá el orden de los circuncentros.

- Se calculan los vectores de diferencia entre circuncentros y uno de ellos con el vértice de Delaunay con el que se asignó el polígono
- Se hace un producto cruz entre estos vectores. Si el resultado es positivo entonces el cirucuncentro pivot será el que tienen ambos vectores en común. Además si es una arista externa significa que es la primera arista del polígono, agregando el polígono en el índice del vértice de Delaunay a la estructura como un par (x, y) donde x será el número de aristas que tiene (más adelante se entrega) e y el índice de la arista. Sino, procedemos a ver cuál es la arista que le sigue.
- Si el producto cruz fue negativo entonces el cirucuncentro pivot será el que no comparte ambos vectores. Esto se puede ver en figura 4.6, donde cualquiera de los dos casos entrega como cirucuncentro pivot, vale decir, cirucuncentro que tienen en común con la arista siguiente del poligono a cc_1)

- Se contabiliza la arista para el polígono en índice del vértice de Delaunay.
- Finalmente se identifica la arista siguiente de la que se está trabajando en un par de **IndexHEV**, con valor x el polígono al que pertence e y la siguiente arista. Si el producto cruz anterior fue menor a 0 y es una arista externa, entonces la identificamos como una arista final del polígono (valorando como -1 la siguiente arista). De lo contrario, se verifica cuál arista de las relacionadas con el cirucuncentro pivot también tiene como polígono el índice del vértice de Delaunay. Es importante destacar que si en la arista siguiente el cirucuncentro que comparten corresponde al menor de ambos (en términos de índice) se almacenará directamente, mientras que si es el mayor se almacenará su arista opuesta (equivalenta al índice de la arista más la cantidad de aristas).

Se puede observar este proceso completo en la figura 4.7, donde se identifica como pivot el circuncentro al cual llega la arista h_e_0. Desde este es posible recononcer como siguiente arista a h_e_3 o h_e_5, siendo esta última la que comparte el polígono que se está formando.

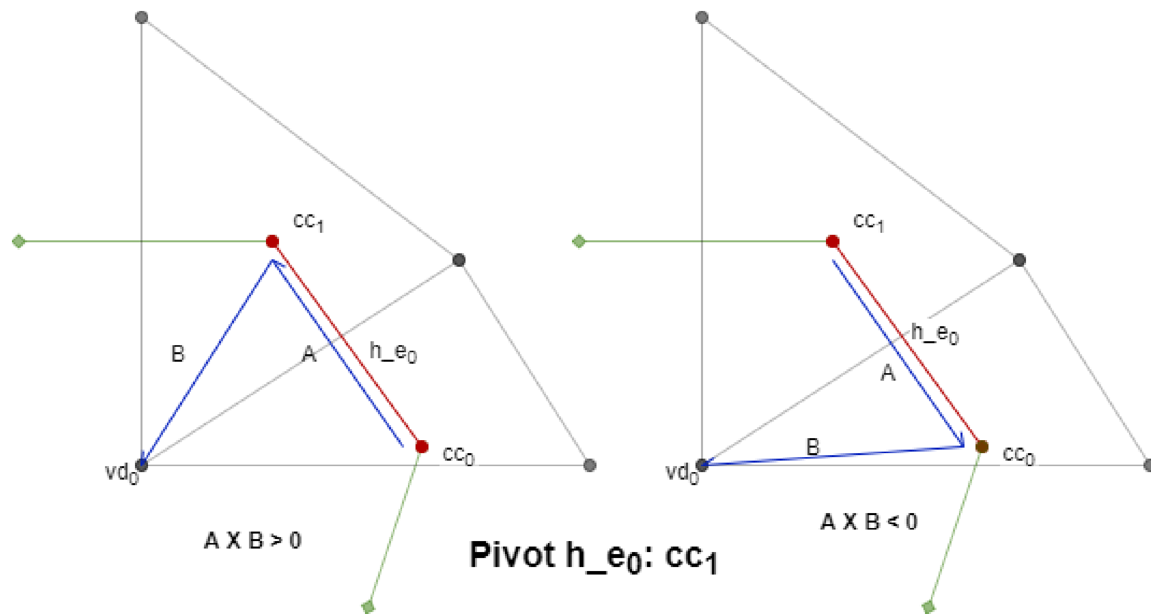

Figura 4.6: Producto cruz entre vector formado por circuncentros y cirucuncentro-vértice de Delaunay.

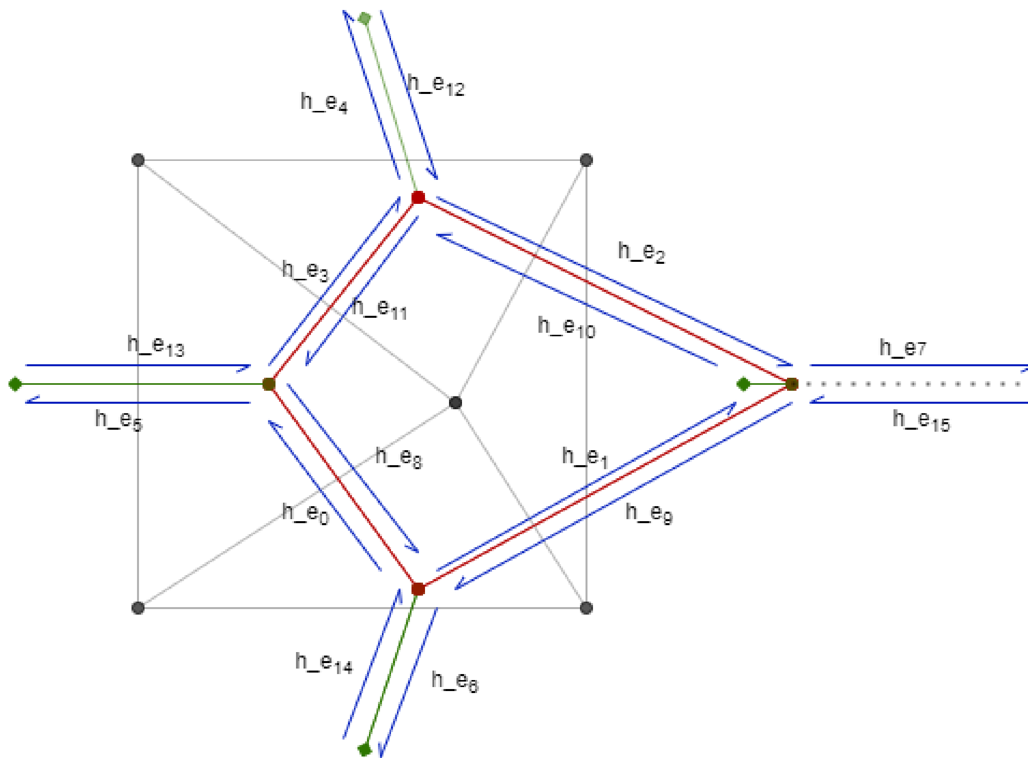

Figura 4.7: Half edges y polígonos formados

4.4. Visualización de Voronoi

Una vez se han calculado todos los pasos anteriores es posible dibujar el diagrama de Voronoi y exportarlo como un archivo **.OFF**.

Cuando las aristas de Voronoi han sido identificadas y almacenadas en las estructuras, un flag perteneciente a la malla (llamada voronoi) es activada. Es posible desactivarla con un botón en TIUQUE como se muestra más adelante. Cabe destacar que este booleano solo es activable siempre y cuando el diagrama de Delaunay y de Voronoi hayan sido calculados.

Una vez activo, durante la fase de renderización de la malla se agregan los vértices y aristas de Voronoi como parte de los buffers que se dibujarán. Junto a ello, se especifican los colores, formas y transformaciones que deben ejecutarse antes de mostrarse de manera visual (todo esto a través del uso de Shaders para la comunicación con GPU).

```
if (m->voronoi) {
    glBindBuffer(GL_ARRAY_BUFFER, m->dm->circumcenters);
    glVertexPointer(3, GL_FLOAT, 4 * sizeof(float), 0);
    glDisableClientState(GL_COLOR_ARRAY);
    glEnable(GL_PROGRAM_POINT_SIZE);
    if(m->circumcenters_draw) glPointSize(5);
    glColor3f(1.0f, 0.0f, 0.0f);
    glPolygonMode(GL_FRONT_AND_BACK, GL_POINT);
```

Figura 4.8: Fragmento de código dedicado a la visualización del diagrama de Voronoi usando OpenGL y Shaders

En la figura 4.8 se ve un trozo de la renderización del diagrama de Voronoi, donde se dibujan los circuncentros asignando un color y tamaño específicos.

En particular, se le asignan el color rojo a los circuncentros, rosado a las aristas internas de Voronoi y amarillo a las externas, de manera que sea posible diferenciar aquellas que debiesen diverger al infinito pero por temas de visualización se proyectan hacia el punto medio.

Capítulo 5

Pruebas y Resultados

A continuación se presentan algunos ejemplos de los resultados obtenidos con la implementación realizada, comparaciones del tiempo de ejecución del algoritmo en conjunto con el existente previamente y cambios en las interfaz de TIUQUE para añadir los diagramas de Voronoi y posibles cambios y/o arreglos que pueden mejorar el desempeño del sistema.

Para las pruebas se utilizó un computador con procesador Intel®Core™i5-8250U Quad Core de 1.6GHz en conjunto a una tarjeta gráfica NVIDIA GeForce MX130 de 2Gb de memoria y 16 Gb de memoria RAM con Ubuntu 16.04LTS como sistema operativo.

5.1. Cambios de interfaz

Antes de plantear los resultados de estos experimentos se explicarán los cambios a la interfaz de TIUQUE que permiten el correcto uso del sistema. En la figura 5.1 se muestra la nueva funcionalidad del sistema a través de un botón, ubicado en la zona inferior derecha de la GUI. Éste tiene una funcionalidad similar a los botones "Solid" y "Wireframe", la cual al presionar permite la visualización del diagrama de Voronoi de la malla cargada en TIUQUE siempre y cuando haya pasado por el proceso de transformación de Delaunay y los cálculos de Voronoi. De esta forma el usuario puede complementar y seleccionar las formas de visualización que estime convenientes para sus necesidades.

5.2. Definición de pruebas

Tras realizar el diseño e implementación del algoritmo y la implantación y modificación de las librería CLEAP y TIUQUE respectivamente es necesario realizar pruebas para medir el desempeño en tiempo de ejecución y uso de memoria.

Para ello, se crearon mallas de triángulos con una cantidad de puntos desde 10^1 hasta 10^6

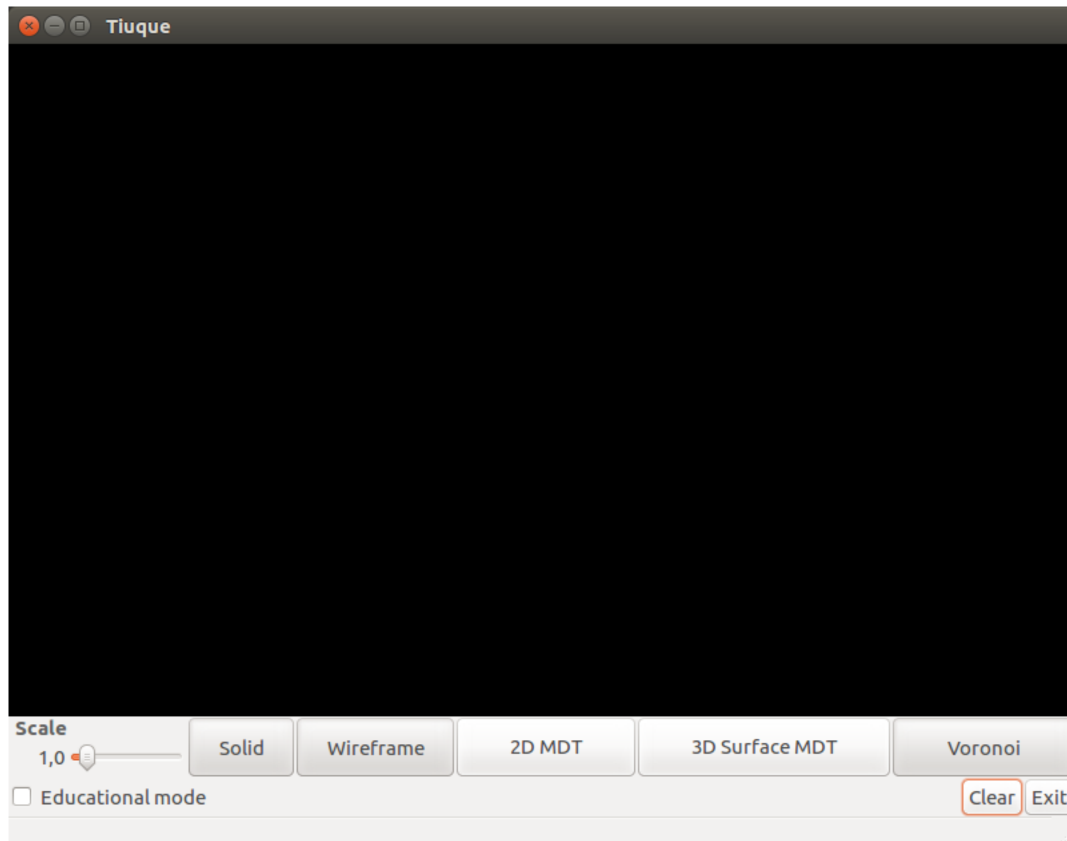

Figura 5.1: Nueva interfaz gráfica de TIUQUE

con saltos de una potencia. Junto a estas, se crearon mallas con un equispaciado de 500.000 puntos que van desde 1.0×10^6 hasta 4.0×10^6, siendo este último el máximo permitido para realizar pruebas en la máquina descrita previamente (en la siguiente sección se explicará el por qué de esta situación). Todas estas mallas serán utilizadas para medir los tiempos de ejecución, uso de memoria, comparaciones entre versiones de librería y con otros algoritmos con el mismo objetivo. Junto a estas, se agregan 3 mallas que representan un conejo, una espada y una manzana con el objetivo de visibilizar los diagramas de Voronoi formados en superficies.

5.3. Uso de memoria

Inicialmente se realizó el análisis a nivel de uso de memoria en las estructuras requeridas. Es importante destacar que las mediciones realizadas en Qhull corresponden al uso de memoria en CPU, mientras que las de CLEAP son en GPU. Esto se debe a que CLEAP crea las estructuras en ambas, pero la memoria en GPU es una limitante mayor en comparación a la memoria RAM.

5.3.1. CLEAP: Delaunay vs Delaunay-Voronoi

La primera comparación corresponde a la librería CLEAP junto a la nueva versión con el algoritmo para calcular el Diagrama de Voronoi implementado. Por simplicidad (y desde ahora en adelante) se llamará CLEAP D-V a la implementación realizada en este trabajo de título, mientras que la versión previa será CLEAP D.

En la tabla 5.1 se puede observar el uso de memoria de ambas versiones para las distintas mallas de prueba. Existe un aumento en la cantidad de memoria solicitada de más de un 250 % para las mallas desde 10^5 en adelante, esto debido a todas las nuevas estructuras que debe utilizar el algoritmo al momento de visualizar Diagrama de Voronoi.

En particular, el último caso de prueba (4.0×10^6) utiliza un aproximado de 1.7 GB de memoria en GPU siendo esto equivalente al $88,5\,\%$ de la memoria máxima (en GPU) del computador de pruebas sólo en el almacenamiento de los datos y buffers de renderización. Al agregar a esto a los procesos que normalmente utilizan la tarjeta gráfica el uso de esta llega a su límite (e incluso en ciertas ocasiones lo sobrepasa), haciendo imposible realizar pruebas con mallas más grandes en esta máquina.

N^o Puntos	CLEAP D-V(MB)	CLEAP D(MB)
10^1	1.50	1.50
10^2	2.50	2.00
10^3	2.84	2.29
10^4	3.31	2.50
10^5	40.94	15.13
10^6	464.56	173.00

N^o Puntos	CLEAP D-V(MB)	CLEAP D(MB)
1.5×10^6	695.63	260.31
2.0×10^6	929.75	348.38
2.5×10^6	1161.63	435.69
3.0×10^6	1396.94	525.75
3.5×10^6	1628.81	611.06
4.0×10^6	1770.50	699.13

Tabla 5.1: Comparación de uso de memoria entre el cálculo original de la librería y el de la nueva implementación

5.3.2. Delaunay-Voronoi: CLEAP vs Qhull

En la tabla 5.2 se muestra el uso de memoria para ambos algoritmos diferentes. Es posible notar que la medición para valores menores o iguales a 10^3 es imprecisa para CLEAP debido a que la cantidad de memoria usada en GPU para almacenar estos datos es mucho menor a las variaciones constantes que existen al renderizar la misma interfaz gráfica.

A partir de 10^4 y en la tabla 5.3 se observa un crecimiento lineal para ambos casos, en

donde CLEAP D-V presenta un menor uso de memoria que Qhull D-V. Sin embargo, el verdadero uso de memoria de CLEAP se muestra en las figuras 5.2 y 5.3 con la línea azul, en el cual también se contabiliza la memoria utilizada en CPU.

Por otro lado, la tendencia de crecimiento lineal en ambas gráficas permite proyectar y calcular la cantidad de memoria necesaria para mallas de mayor tamaño, siendo que por cada punto es necesario 4.62×10^{-4} MB como mínimo en GPU (y en CPU).

N^o Puntos	CLEAP D-V(MB)	Qhull D-V(MB)
10^1	1.50	0.02
10^2	2.50	0.07
10^3	2.84	0.60
10^4	3.31	5.88
10^5	40.94	58.76
10^6	464.56	587.47

Tabla 5.2: Comparación de uso de memoria de CLEAP y Qhull para mallas con potencias de 10 puntos

N^o Puntos	CLEAP D-V(MB)	Qhull D-V(MB)
1.0×10^6	464.56	587.47
1.5×10^6	695.63	881.20
2.0×10^6	929.75	1174.94
2.5×10^6	1161.63	1468.67
3.0×10^6	1396.94	1762.40
3.5×10^6	1628.81	2056.13
4.0×10^6	1770.50	2349.86

Tabla 5.3: Comparación de uso de memoria de CLEAP y Qhull para mallas desde 1.000.000 hasta 4.000.000 de puntos

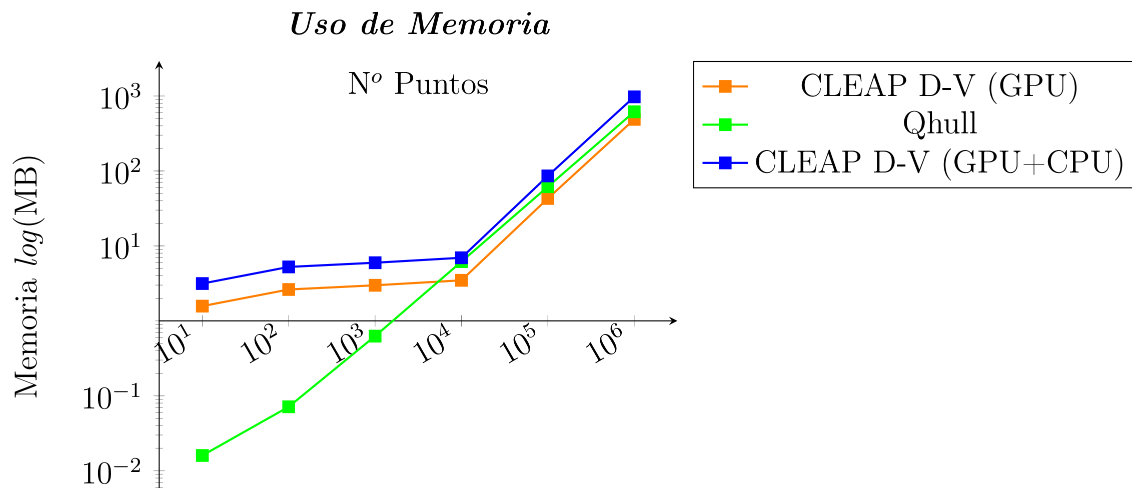

Figura 5.2: Uso de memoria de CLEAP D-V y Qhull para mallas con potencias de 10 puntos

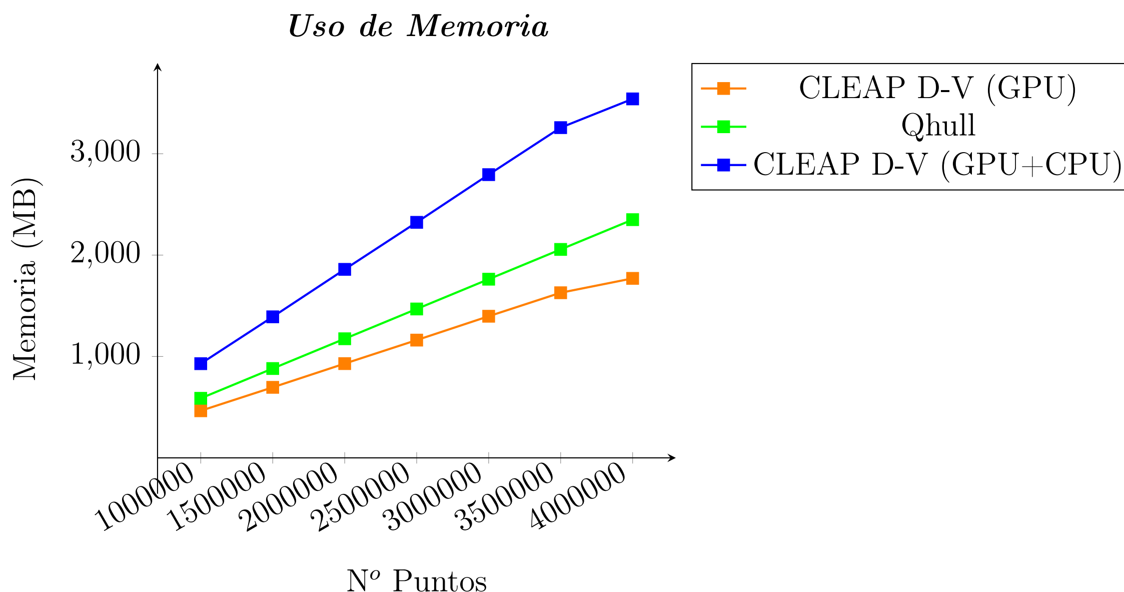

Figura 5.3: Uso de memoria de CLEAP y Qhull para mallas desde 1.000.000 hasta 4.000.000 de puntos

5.4. Tiempos de ejecución

En esta sección se muestran los tiempos de ejecución de los algoritmos, realizando la comparación entre CLEAP D y CLEAP D-V, así como Qhull D-V con este último. Este tiempo ha sido contabilizado desde después de la carga de la malla, restringiendose netamente el cálculo de los diagramas correspondientes.

5.4.1. CLEAP: Delaunay vs Delaunay-Voronoi

En la tabla 5.4 se observan los tiempos utilizados por CLEAP D y CLEAP D-V para calcular los diagramas de Delaunay y Vornoi correspondientes. Es importante destacar el hecho de que CLEAP D-V calcula ambos elementos.

Es posible observar que el tiempo de cálculo de Voronoi en comparación con el cálculo inicial es alto para mallas más pequeñas con un 50,68 % en la malla de 10 puntos, pero va disminuyendo hacia mallas más grandes siendo un 3,93 % para la última pontencia y un 2 % para la malla de 4.000.000 (equivalente a 9 ms).

5.4.2. Delaunay-Voronoi: CLEAP vs Qhull

En las tablas 5.5 y 5.6 se exponen los resultados para las pruebas realizadas. Para mallas de 10^3 o menos puntos Qhull D-V presenta mejores resultados que CLEAP D-V. Esto se debe al costo inherente de tiempo al realizar el traspaso de información entre GPU y CPU.

A partir de 10^3 CLEAP D-V muestra un tiempo de procesamiento menor que Qhull. En las figuras 5.4 y 5.5 se muestra el comportamiento de $O(nlog(n))$ de Qhull D-V, mientras que CLEAP D-V $O(n)$ en función de la cantidad de puntos, con una mejora en tiempo de 547 % en el último caso de prueba.

N^o Puntos	CLEAP D-V(s)	CLEAP D(s)
10^1	3.68×10^{-4}	2.36×10^{-4}
10^2	6.08×10^{-4}	4.91×10^{-4}
10^3	1.58×10^{-3}	1.12×10^{-3}
10^4	7.70×10^{-3}	6.95×10^{-3}
10^5	8.63×10^{-2}	8.16×10^{-2}
10^6	1.08	1.04

N^o Puntos	CLEAP D-V(s)	CLEAP D(s)
1.5×10^6	1.66	1.61
2.0×10^6	2.27	2.24
2.5×10^6	3.08	3.05
3.0×10^6	3.77	3.70
3.5×10^6	4.35	4.29
4.0×10^6	5.04	4.95

Tabla 5.4: Comparación de tiempos de ejecución entre el cálculo original de la librería y el de la nueva implementación

N^o Puntos	CLEAP D-V(s)	Qhull D-V(s)
10^1	3.68×10^{-4}	7.2×10^{-5}
10^2	6.08×10^{-4}	4.36×10^{-4}
10^3	1.58×10^{-3}	3.99×10^{-3}
10^4	7.70×10^{-3}	5.42×10^{-2}
10^5	8.63×10^{-2}	6.55×10^{-1}
10^6	1.08	6.77

Tabla 5.5: Tiempos de ejecución para mallas de potencias de 10 puntos en CLEAP y Qhull

N^o Puntos	CLEAP D-V(s)	Qhull D-V(s)
1.0×10^6	1.08	6.77
1.5×10^6	1.66	10.32
2.0×10^6	2.27	13.83
2.5×10^6	3.07	16.96
3.0×10^6	3.77	20.49
3.5×10^6	4.35	23.76
4.0×10^6	5.04	27.58

Tabla 5.6: Comparación de tiempos de ejecución para mallas desde 1.000.000 hasta 4.000.000 de puntos entre CLEAP y Qhull

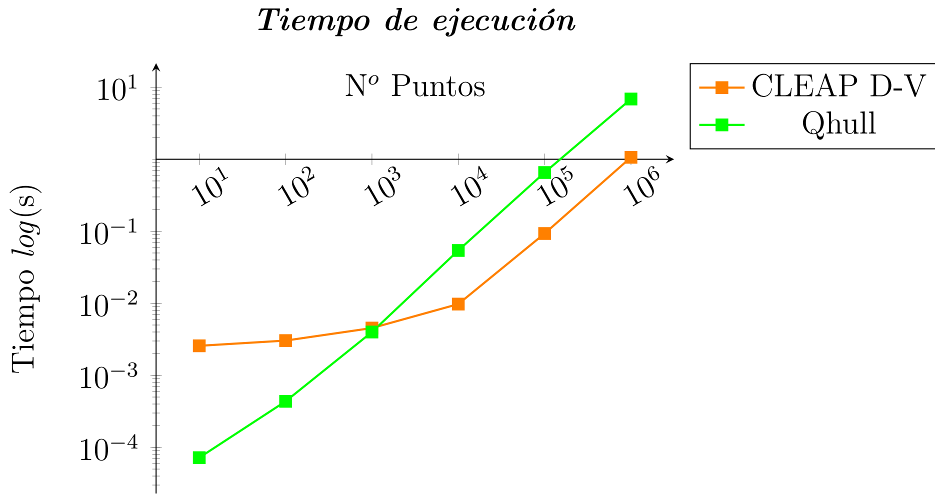

Figura 5.4: Tiempos de ejecución para mallas de potencias de 10 puntos

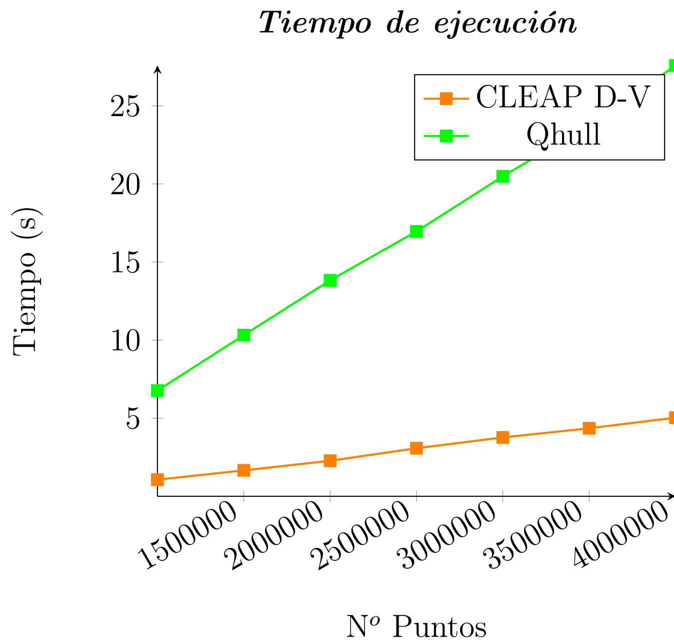

Figura 5.5: Tiempos de ejecución para mallas desde 1.000.000 hasta 4.000.000

5.5. Representaciones y correctitud

En las figuras 5.6 y 5.7 se puede observar una malla de 10 vértices antes y después de ejecutar la transformación de Delaunay. Es posible notar que la funcionalidad anterior sigue de igual forma. Al hacer click en "Voronoi" la teselación de Voronoi se renderiza en el lienzo, complementándose con la triagulación dual como se aprecia en la figura 5.8. A su vez, es posible solo observar el diagrama de Voronoi como se muestra en la 5.9. La visualización gráfica presenta algunos defectos al no mostrar todas las aristas de Voronoi (tanto internas como externas). Sin embargo, el archivo de formato **OFF** (figura 5.10) que genera el algoritmo si contiene la información necesaria para armar el diagrama de manera completa, como se

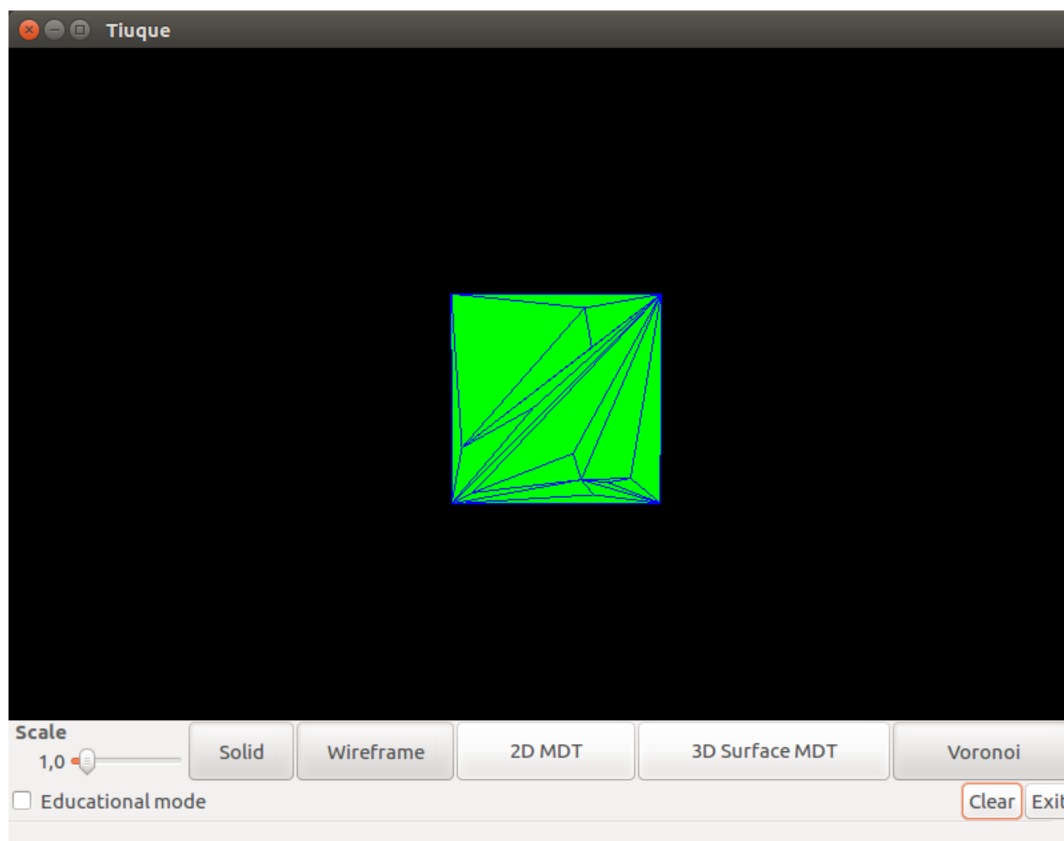

Figura 5.6: Malla de triángulos del orden de 10 vértices previa a ser procesada

muestra en la figura 5.11. Esta última imagen corresponde al software llamado RoffView[1], que es un programa que permite la visualización de archivos del formato OFF. Cabe destacar que esta visualización, al igual que las que se realizan en TIUQUE proyecta las aristas externas del diagrama de Voronoi (aquellas que divergen al infinito) hacia los puntos medios de la arista de Delaunay correspondiente.

En las figuras 5.12, 5.13, 5.14 y 5.15 es posible observar el mismo proceso y sus resultados.

Por otro lado, las mallas de superficie de objetos conocidos también presentan buenos resultados al momento de calcular su diagrama de Voronoi, como se puede observar con el caso de la manzana en las figuras 5.16 y 5.17.

Todos los ejemplos que no se presentan en este capítulo se encuentran disponibles en el capítulo de anexos al final del documento.

[1]http://www.holmes3d.net/graphics/roffview/

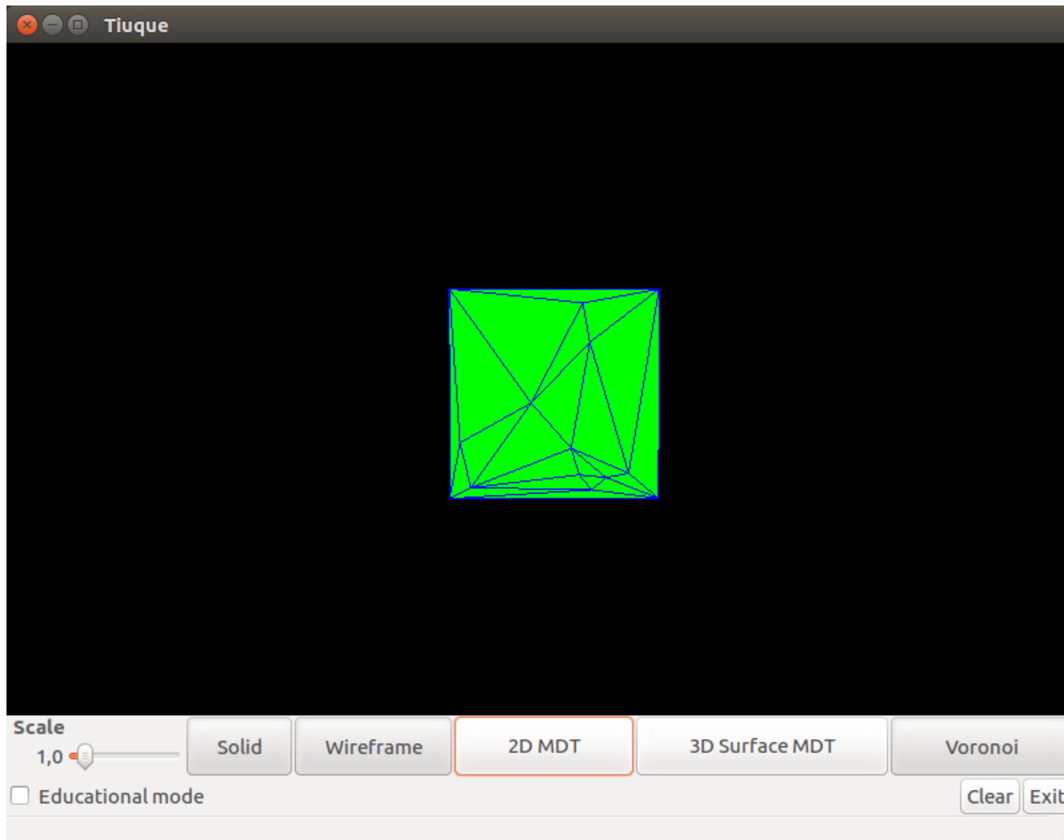

Figura 5.7: Triangulación de Delaunay obtenida por la librería

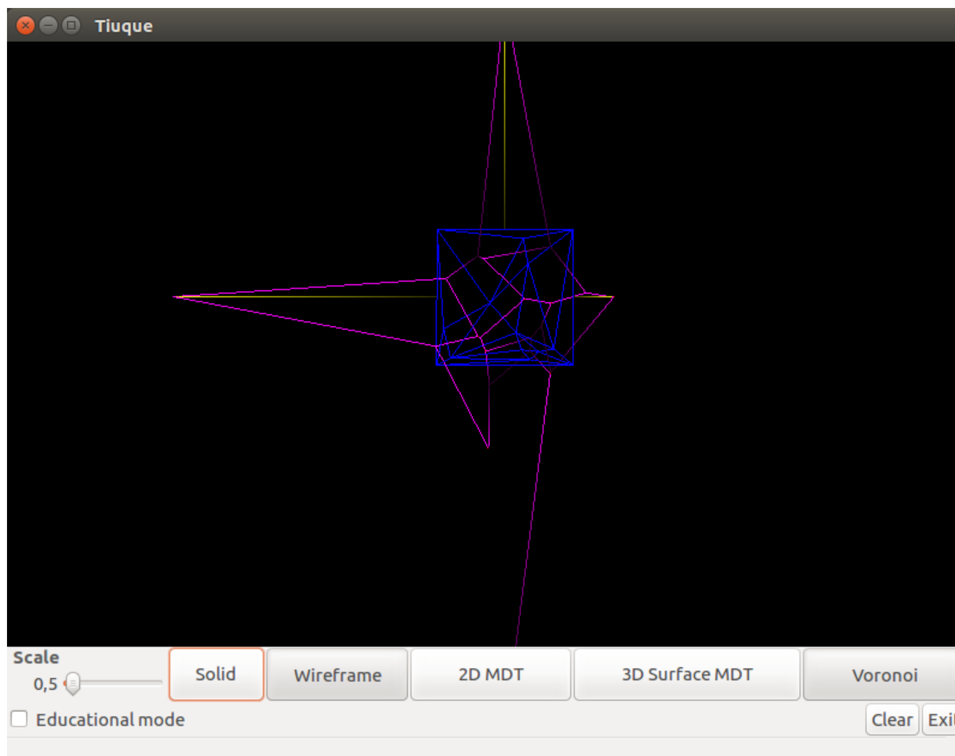

Figura 5.8: Visualización de la Triangulación de Delaunay y Diagrama de Voronoi asociados a la malla

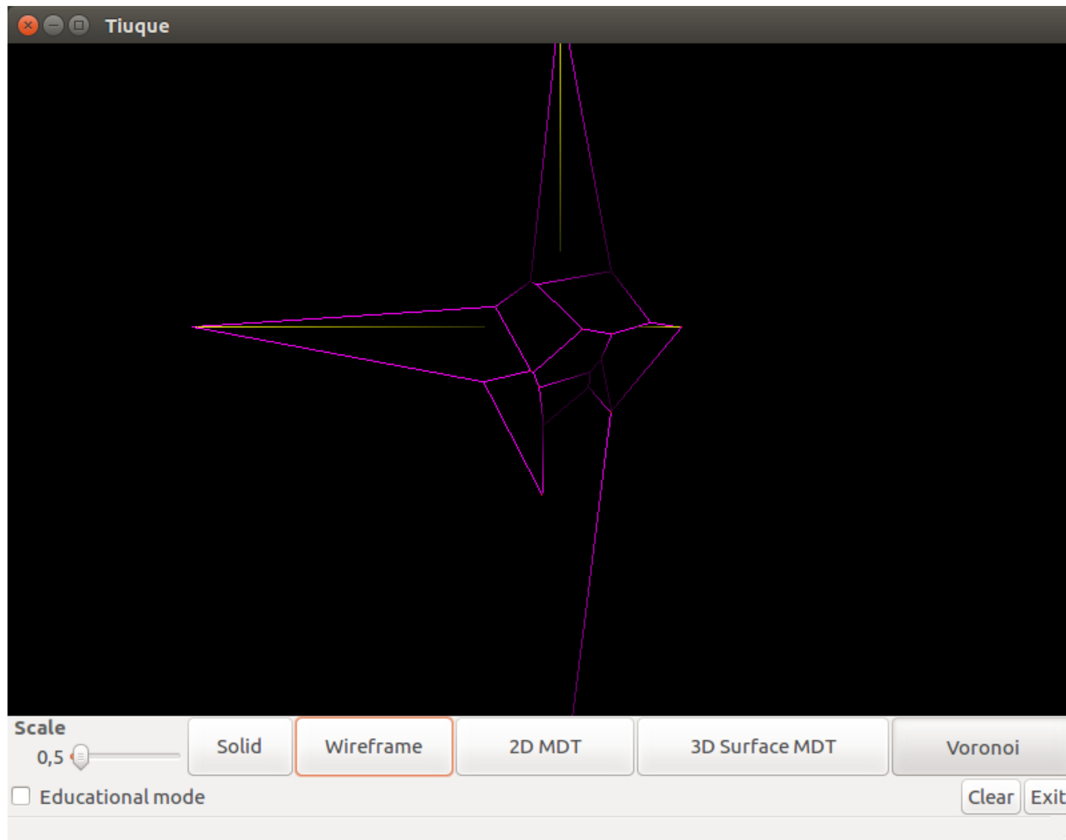

Figura 5.9: Diagrama de Voronoi visualizado de manera independiente

```
26 14 0
0.797220 -0.000000 0.000000
-0.508533 -0.359670 0.000000
0.595265 0.033047 0.000000
0.329063 -0.563137 0.000000
0.337668 -0.043841 0.000000
-0.122165 -1.103625 0.000000
-0.180709 -0.297918 0.000000
-0.142020 -0.397071 0.000000
0.146025 -0.010409 0.000000
-2.417976 0.000000 0.000000
-0.432510 0.136883 0.000000
-0.197256 -0.286101 0.000000
0.000000 2.216802 0.000000
-0.159818 0.284212 0.000000
0.334904 0.373046 0.000000
-0.199060 0.304429 0.000000
0.193707 -0.295051 0.000000
0.333288 -0.552455 0.000000
0.266761 -0.207984 0.000000
0.183339 -0.394274 0.000000
-0.113170 -0.643576 0.000000
0.000000 -3.112932 0.000000
1.500000 0.000000 0.000000
-1.500000 0.000000 0.000000
0.000000 1.500000 0.000000
0.000000 -1.500000 0.000000
6 25 21 5 1 9 23
6 22 0 17 3 21 25
6 24 12 14 2 0 22
6 23 9 10 15 12 24
4 19 16 7 20
6 1 5 20 7 6 11
6 4 8 6 7 16 18
4 10 9 1 11
6 8 13 10 11 6 8
4 13 12 15 13
5 2 14 13 8 4
5 17 0 2 4 18
```

Figura 5.10: Archivo .off resultante del proceso

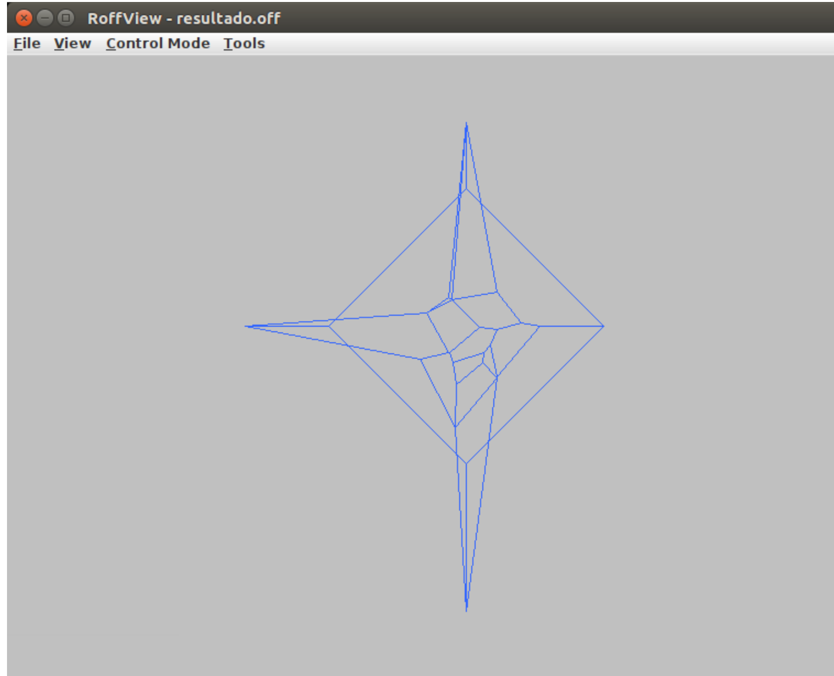

Figura 5.11: Visualización del archivo de exportación del diagrama de Voronoi para malla de orden de 10 puntos

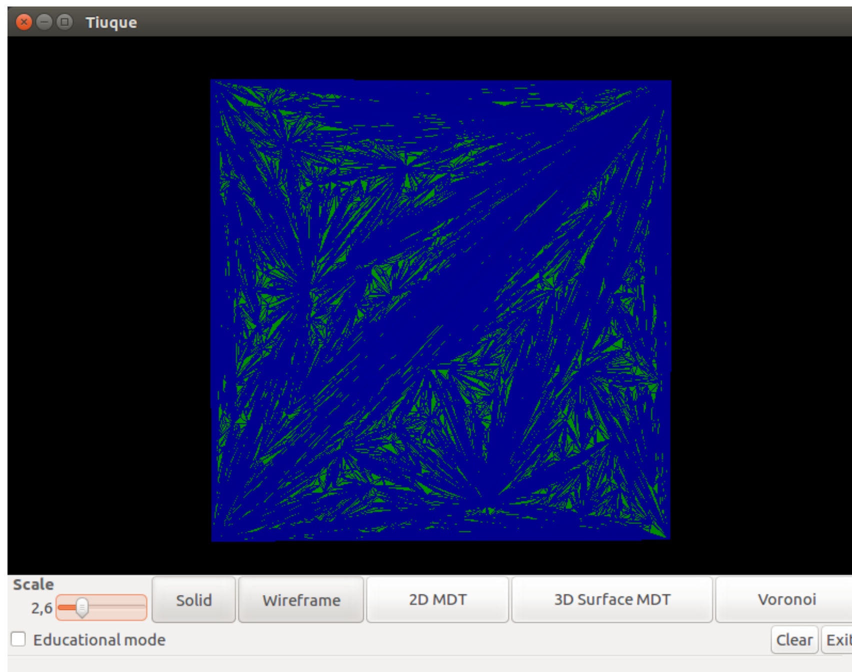

Figura 5.12: Malla de triángulos del orden de 10^4 previa a procesamiento

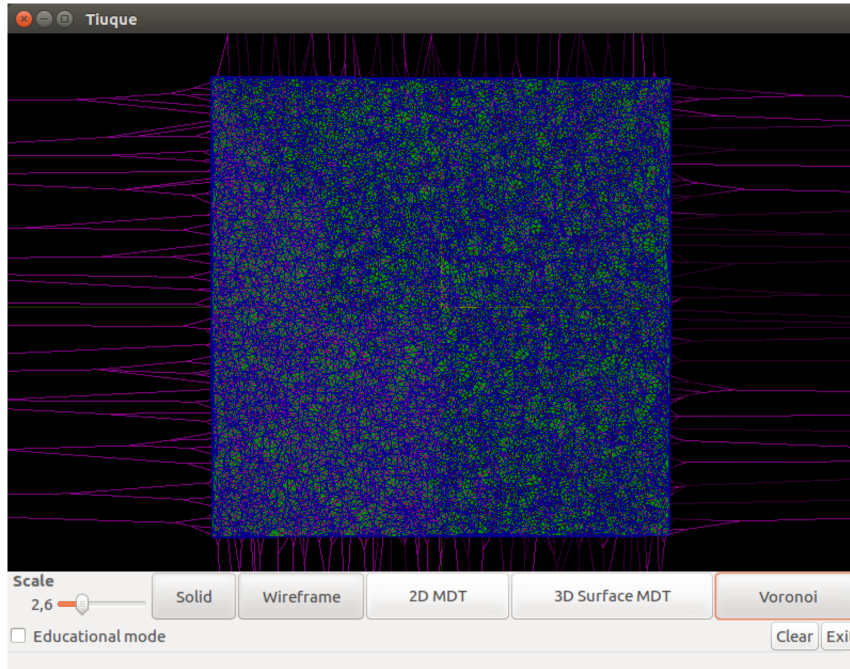

Figura 5.13: Visualización de diagramas de Delaunay y Voronoi para malla del orden de 10^4 puntos

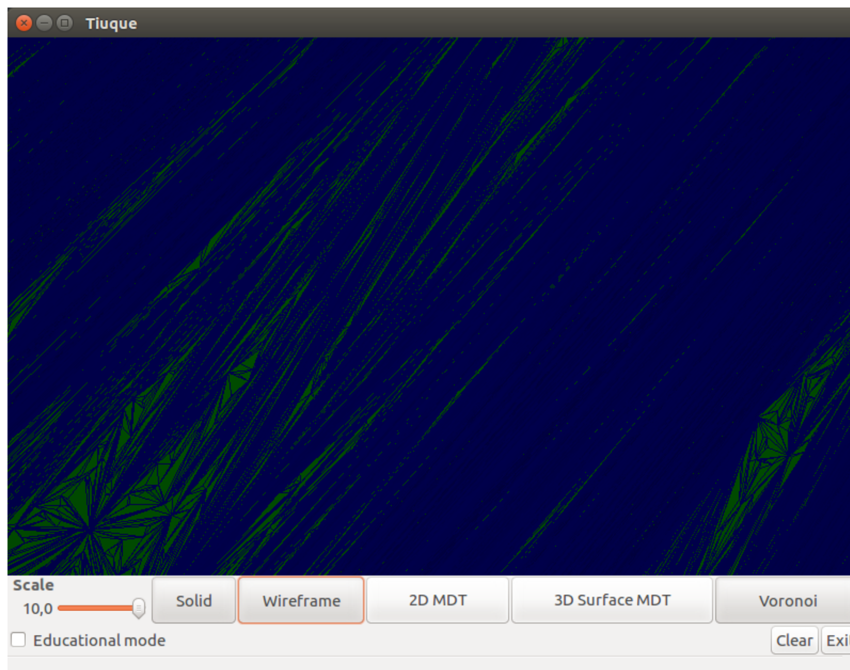

Figura 5.14: Fragmento de malla de triángulos del orden de 10^6 vértices

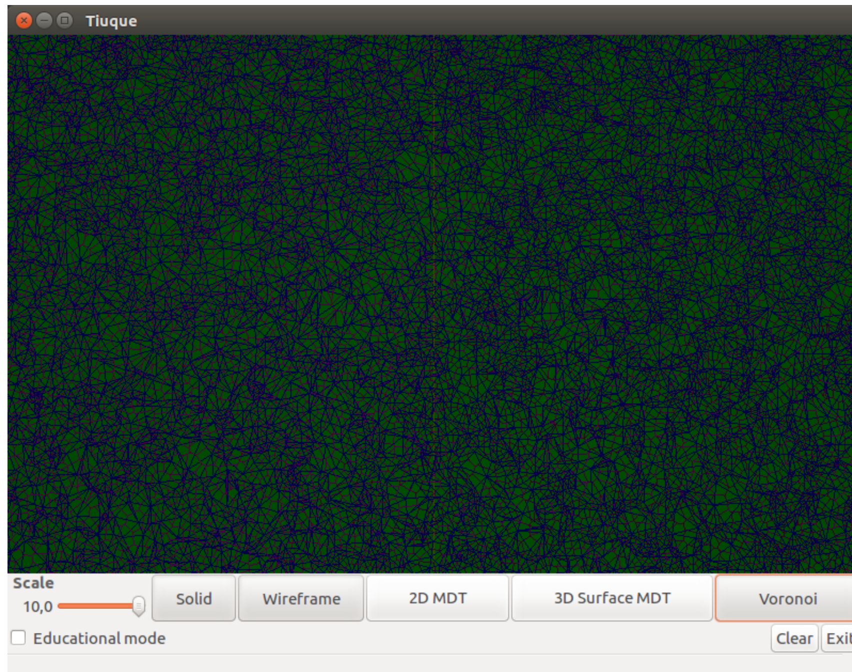

Figura 5.15: Fragmento de visualización de Delaunay-Voronoi para 10^6 puntos

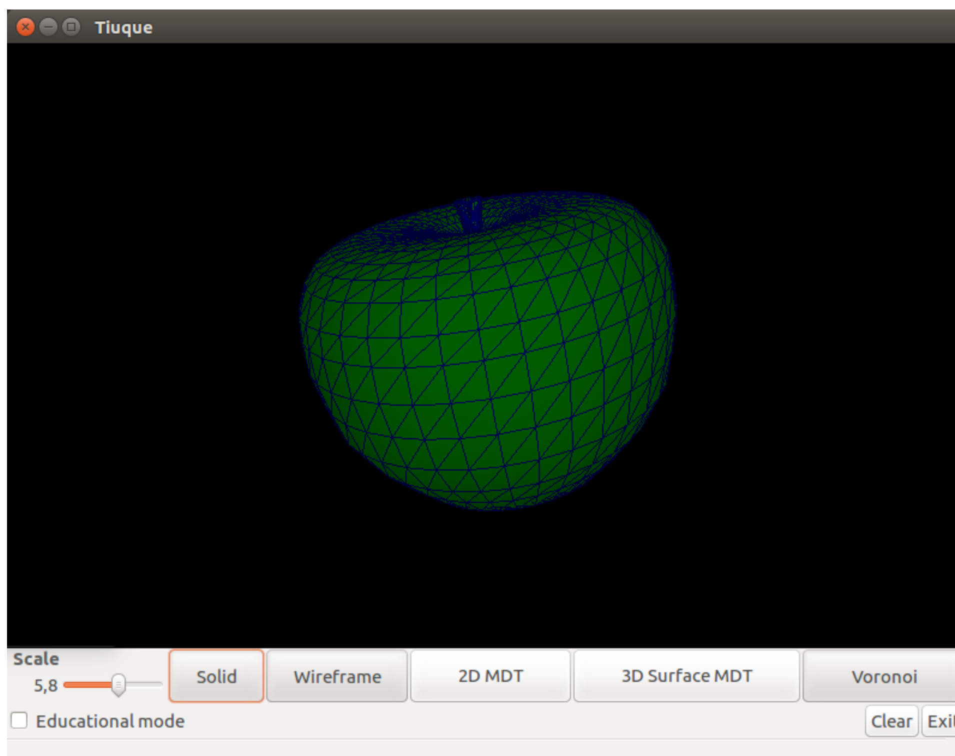

Figura 5.16: Malla de triángulos que representa la superficie de una manzana de 891 puntos

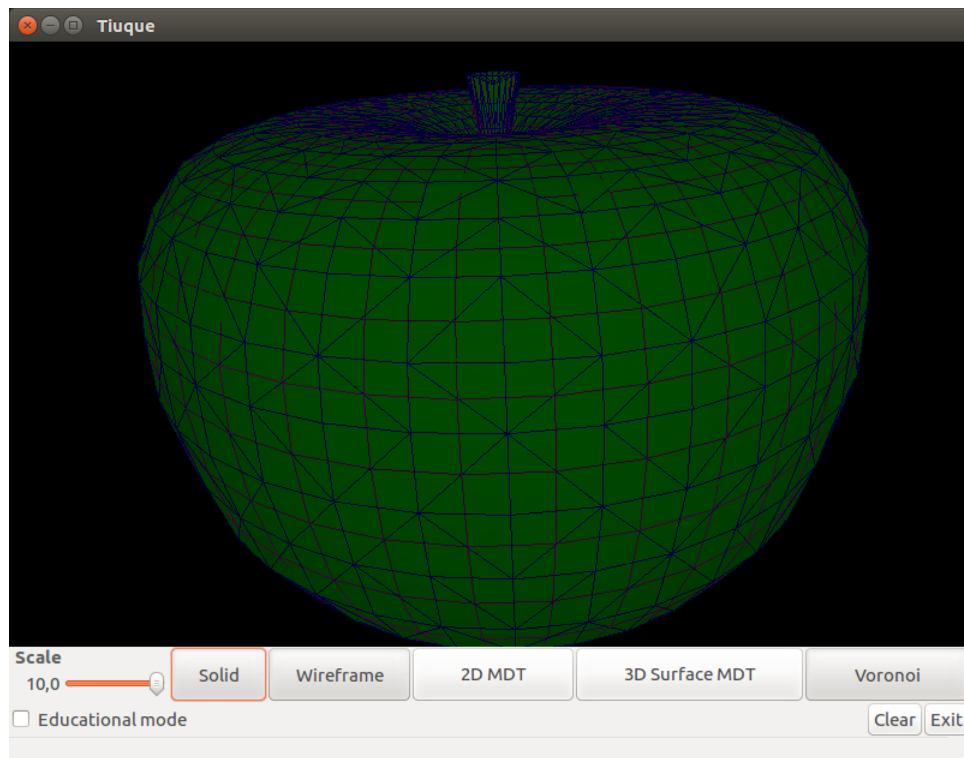

Figura 5.17: Visualización de Voronoi y Delaunay en la superficie de una manzana

Capítulo 6

Conclusiones

En primer lugar, fue posible diseñar un algoritmo paralelo para generar Diagramas de Voronoi recibiendo como información una Triangulación de Delaunay sin más restricciones. Junto a esto, la implantación en la librería CLEAP, en conjunto con los cambios de esta misma y de TIUQUE no alteraron la orgánica de la librería ni su forma de operar.

Con respecto a la utilidad del software, se puede afirmar que un 3 % de tiempo extra para un cálculo que da más flexibilidad de uso al sistema es netamente un resultado positivo, pudiendo abrir esta librería a un campo más amplio de casos de uso. La comparación con otros algoritmos con el mismo objetivo entrega de igual manera un buen comportamiento, llegando a ser 5 veces más rápido en GPU que CPU.

Por otro lado, los análisis de memoria muestran que si bien los espacios reservados en CLEAP son menores a los de otros algoritmos, el hecho de que utilice CPU y GPU (siendo este último más restrictivo) hace que para cálculos de mayores magnitudes sea necesario tener una máquina dedicada. El aumento en uso de memoria entre la librería original y la nueva implementación es ciertamente preocupante, por lo que queda propuesto como trabajo futuro realizar optimizaciones de espacio sin alterar los tiempos del algoritmo. Varias de las estructuras creadas en esta implementación terminan almacenando valores basura o simplemente vacíos, pudiendo ser modificadas para reducir la memoria utilizada.

Considerando los puntos explicados, cada uno de los objetivos específicos se cumplen, haciendo que el objetivo general del trabajo de título también se cumpla.

Bibliografía

[1] Cristobal Navarro, Nancy Hitschfeld, and Eliana Scheihing. A parallel gpu-based algorithm for delaunay edge-flips. In Michael Hoffmann, editor, *27th European Workshop on Computational Geometry (EuroCG)*, pages 75–78, Morschach, Switzerland, Mar 2011.

[2] Joseph O'Rourke. *Computational Geometry in C*. Cambridge University Press, New York, NY, USA, 2nd edition, 1998.

[3] Cristóbal A. Navarro, Nancy Hitschfeld-Kahler, and Luis Mateu. A survey on parallel computing and its applications in data-parallel problems using gpu architectures. *Communications in Computational Physics*, 15(2):285–329, 2014.

[4] A. Aggarwal, B. Chazelle, L. Guibas, C. Ó'Dúnlaing, and C. Yap. Parallel computational geometry. *Algorithmica*, 3(1):293–327, Nov 1988.

[5] D. T. Lee and B. J. Schachter. Two algorithms for constructing a delaunay triangulation. *International Journal of Computer & Information Sciences*, 9(3):219–242, Jun 1980.

[6] S. Rebay. Efficient unstructured mesh generation by means of delaunay triangulation and bowyer-watson algorithm. *Journal of Computational Physics*, 106(1):125 – 138, 1993.

[7] F Hurtado, M Noy, and J. Urrutia. Flipping edges in triangulations. *Discrete & Computational Geometry*, 15(2):333–346, 1999.

[8] NVIDIA CUDA Compute Unified N. Corporation. *Device Architecture - Programming Guide*. NVIDIA Corporation, 2007.

[9] Cristobal A. Navarro. *Cleap:A gpu-based library for handling and processing 3D meshes.* Cristobal A. Navarro, 2011.

[10] T. Alumbaugh and X. Jiao. Compact array-based mesh data structures. In *IMR*, 2005.

Anexos

Anexo A Capítulo 2

En las figuras 6.1, 6.2 y 6.3 se exponen diferentes tipos de mallas de polígonos utilizadas, expuestas en el capítulo 2.

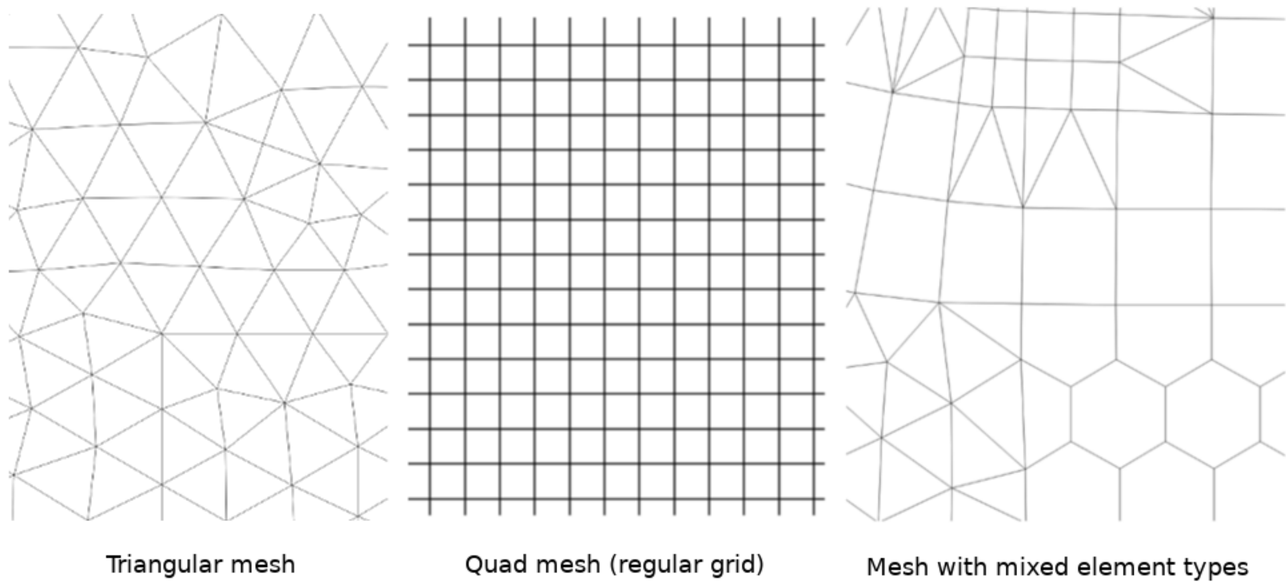

Figura 6.1: Exposición de mallas con distintos polígonos

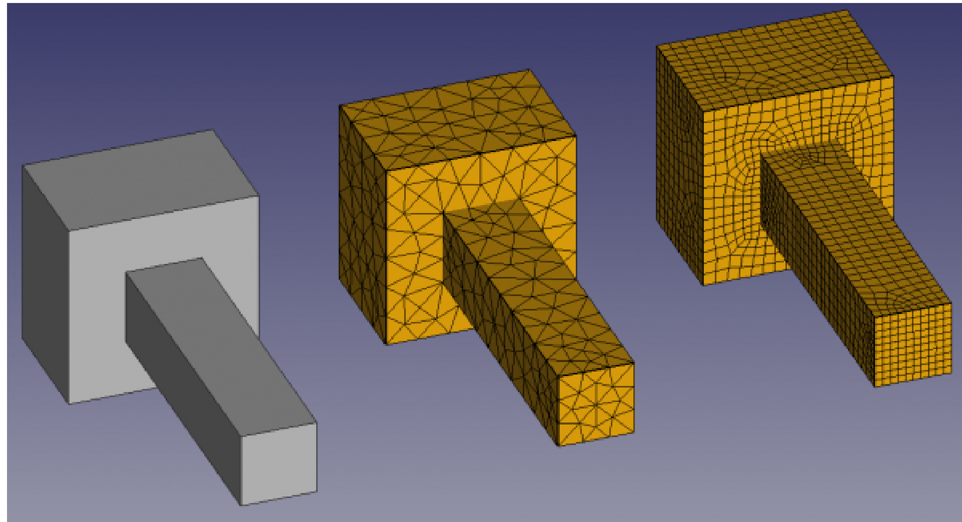

Figura 6.2: Diferentes tipos de malla sobre la superficie de un poliedro

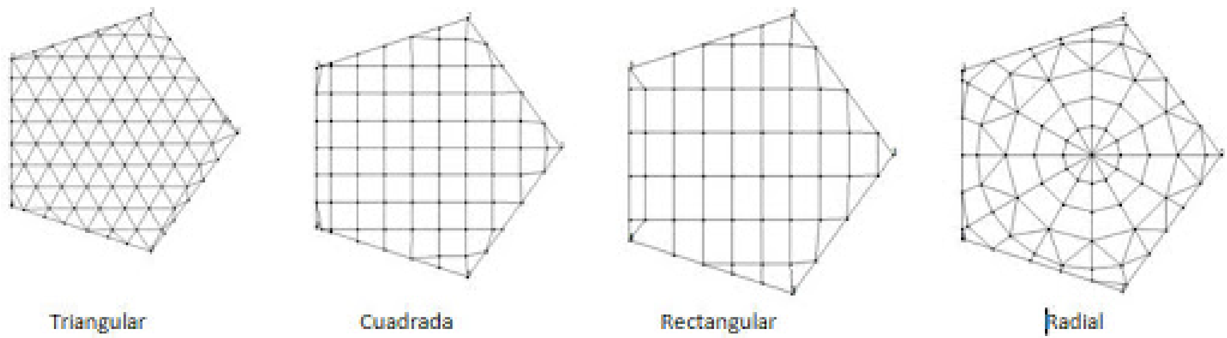

Figura 6.3: Teselaciones con polígonos variados

Anexo B Capítulo 4

Las siguientes figuras son imágenes del proceso de implementación del algoritmo, mostrando diferentes visualizaciones. La interfaz gráfica en ese entonces no había sido modificada.

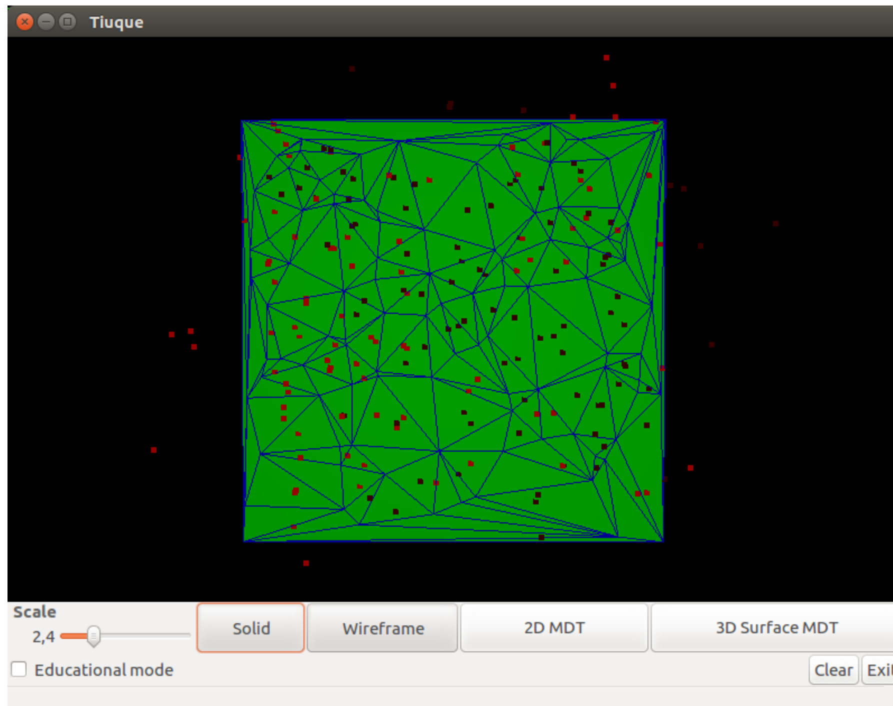

Figura 6.4: Visualización de circumentros de una triangulación

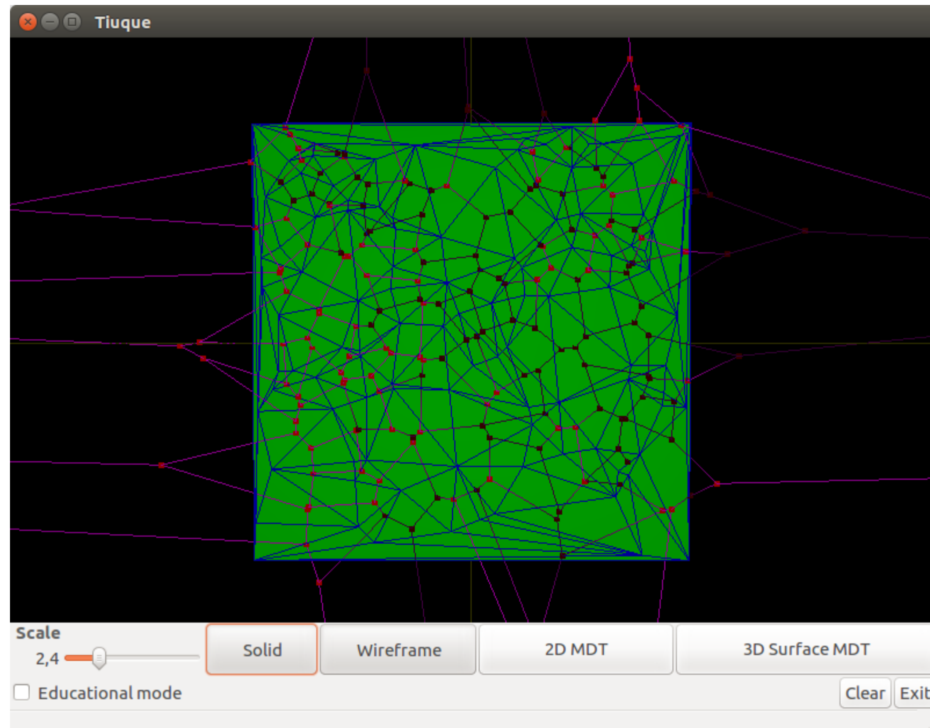

Figura 6.5: Diagrama de Voronoi y Triangulación de Delaunay de una malla de triángulos

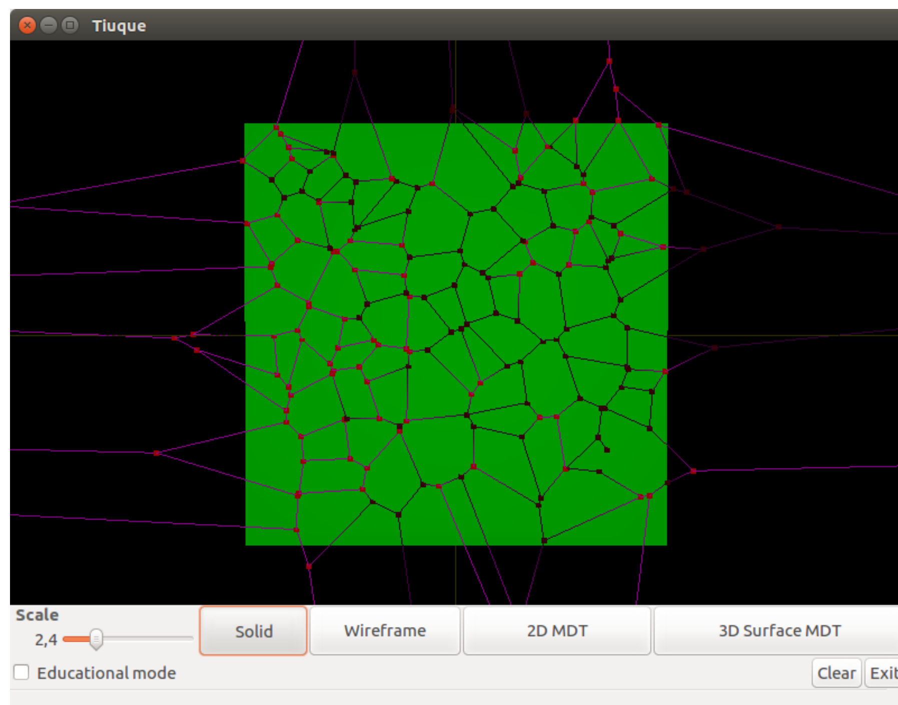

Figura 6.6: Variación de visualización 6.5 ocultando las aristas de la triangulación

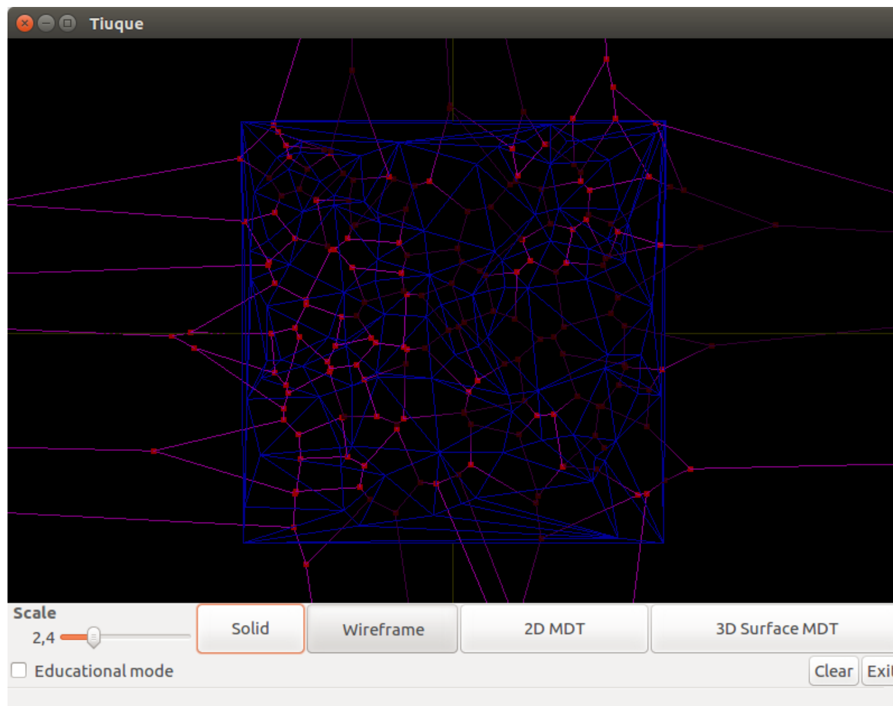

Figura 6.7: Variación de visualización 6.5, exponiendo aristas de ambos diagramas simultaneamente

Anexo C Capítulo 5

Las figuras a continuación son en su mayoría resultados de pruebas realizadas con la librería.

```
OFF
16 9 0
0.868072 0.000000 0.000000
-0.575605 -0.000000 0.000000
-0.096753 0.383708 0.000000
-0.000000 -1.186695 0.000000
-0.242767 0.221155 0.000000
-0.531379 0.024551 0.000000
0.000000 0.708096 0.000000
0.549586 -0.081993 0.000000
0.090878 -0.238627 0.000000
0.113841 -0.209305 0.000000
0.396988 -0.067976 0.000000
-0.095489 -0.622675 0.000000
1.500000 0.000000 0.000000
-1.500000 0.000000 0.000000
0.000000 1.500000 0.000000
0.000000 -1.500000 0.000000
5 15 3 11 1 13
5 12 0 7 3 15
4 14 6 0 12
7 13 1 5 4 2 6 14
4 1 11 8 5
4 4 5 8 9
5 2 10 7 0 6
4 2 4 9 10
6 3 7 9 8 11 3
```

Figura 6.8: Archivo .off resultante del proceso. Se puede ver el número de vértices y aristas que componen el diagrama, así como algunas definiciones

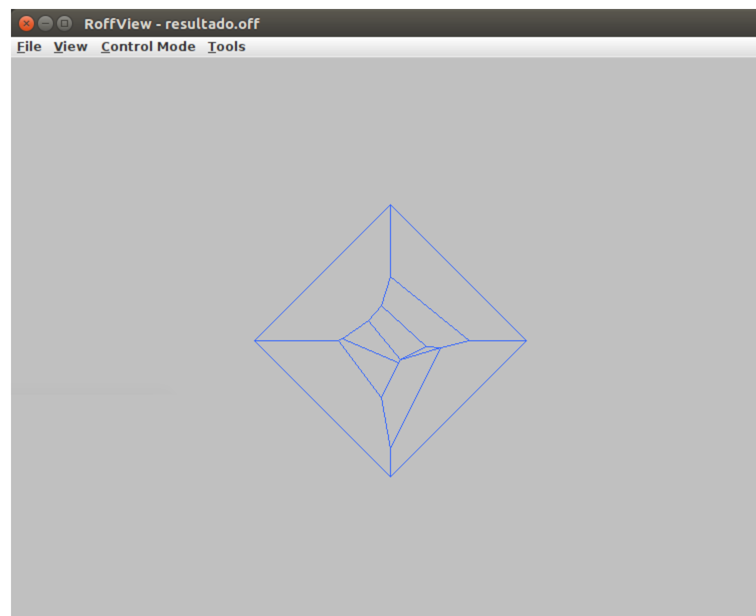

Figura 6.9: Visualización del diagrama de Voronoi expuesto en 6.8 a través de RoffView

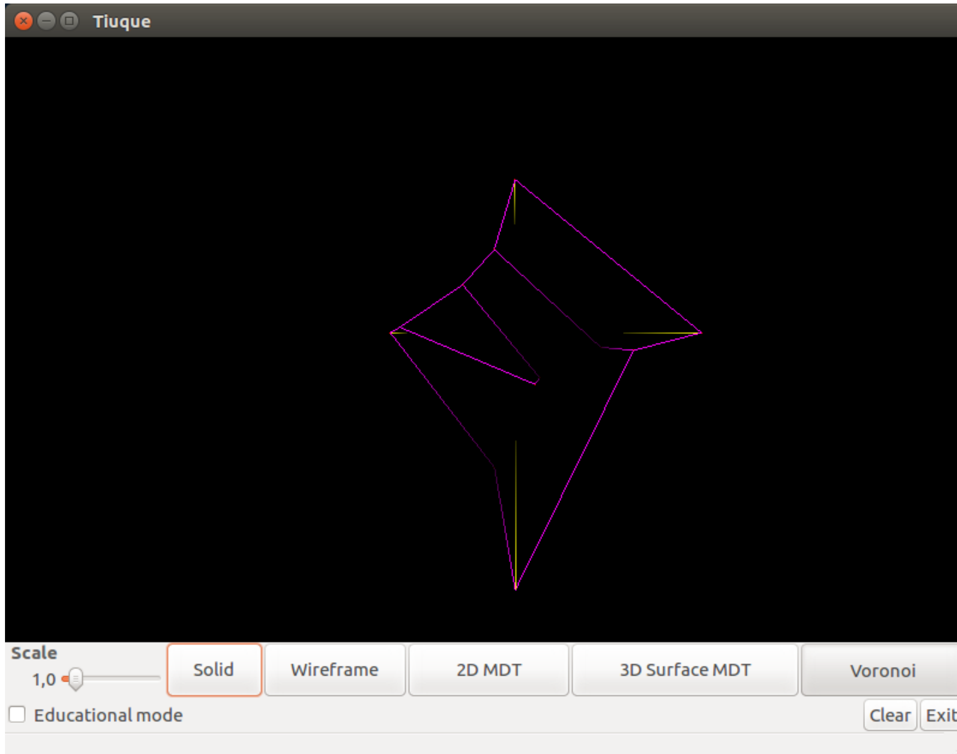

Figura 6.10: Visualización del diagrama de Voronoi expuesto en 6.8 a través de TIUQUE

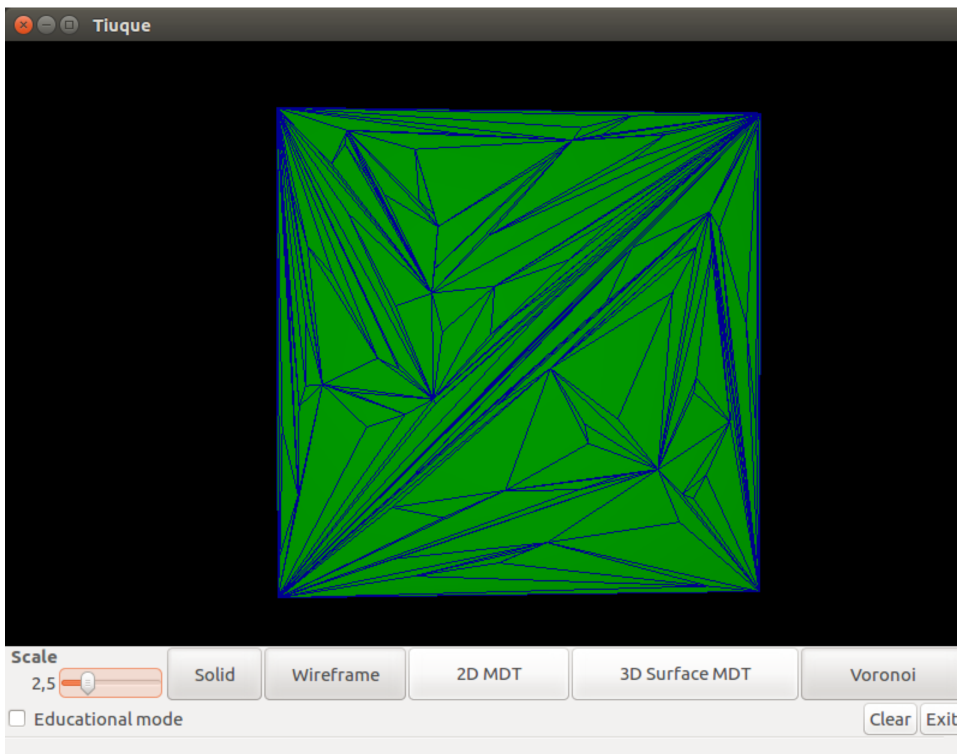

Figura 6.11: Malla de triángulo de orden de 10^2 puntos

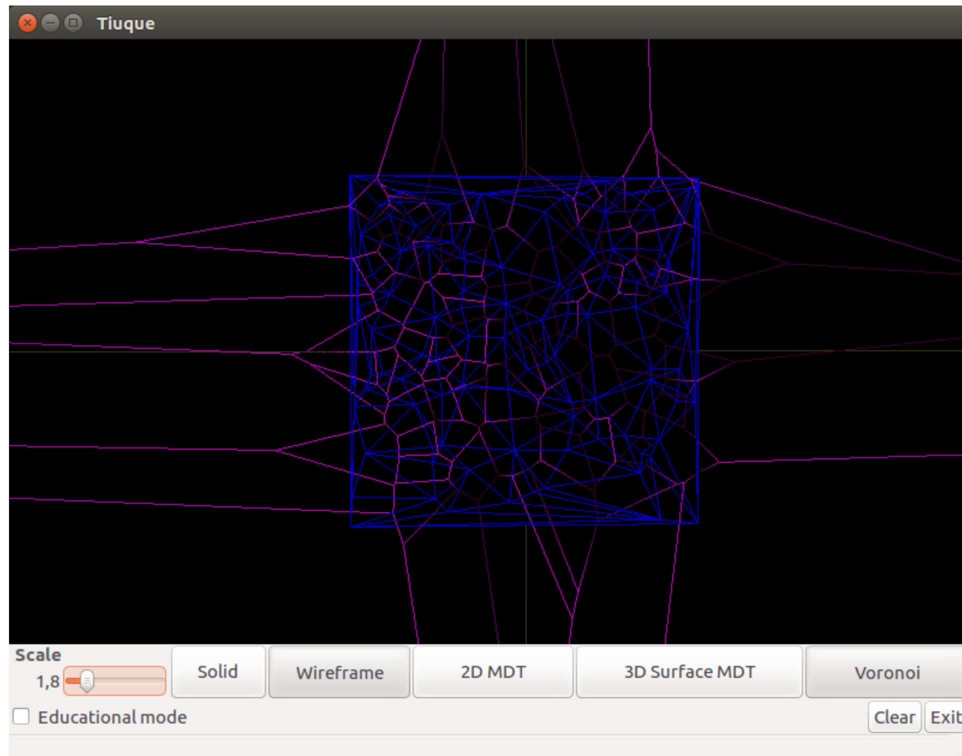

Figura 6.12: Delaunay y Voronoi de malla 6.11

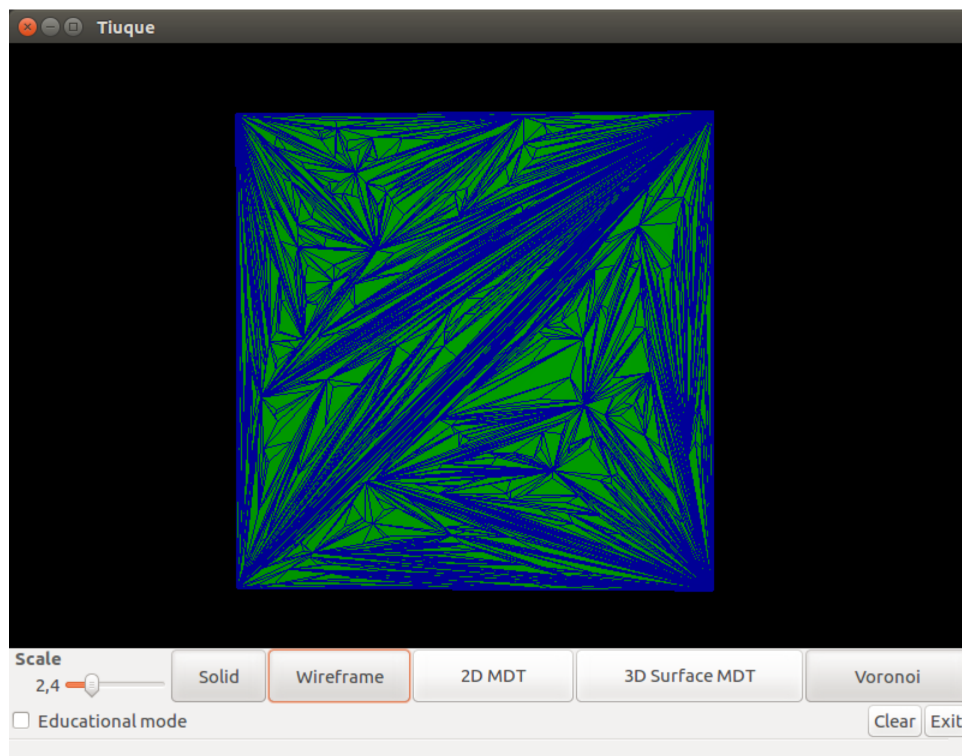

Figura 6.13: Malla de triángulo de orden de 10^3 puntos

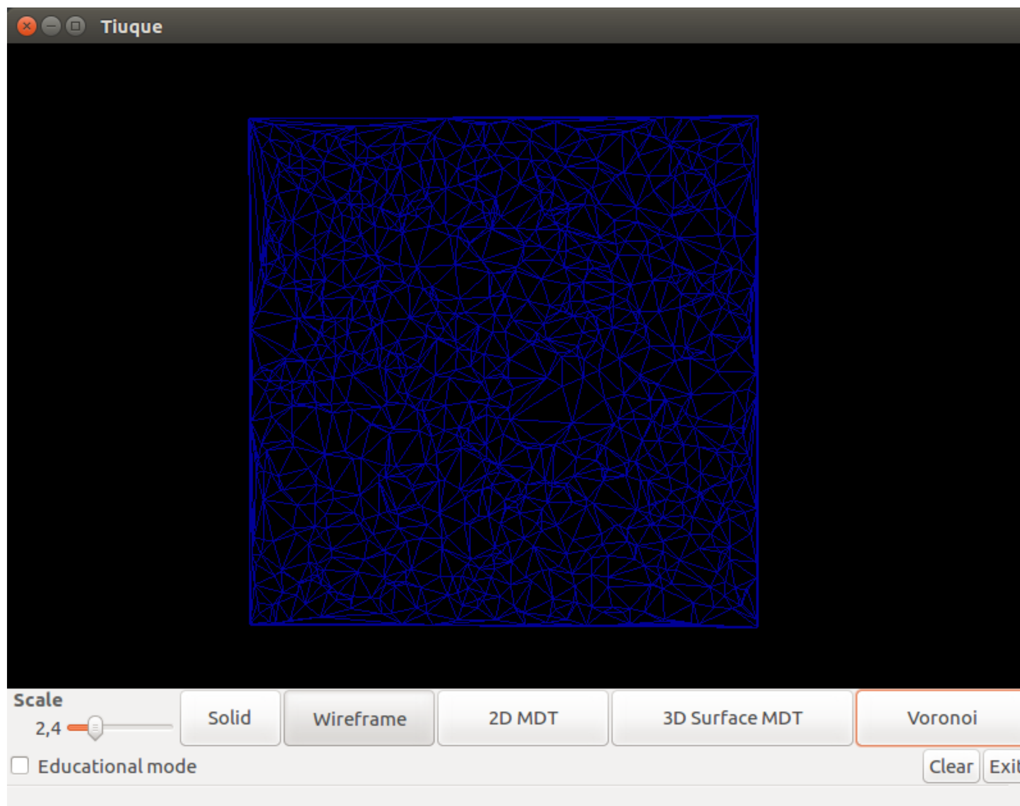

Figura 6.14: Triangulación de Delaunay de malla 6.13

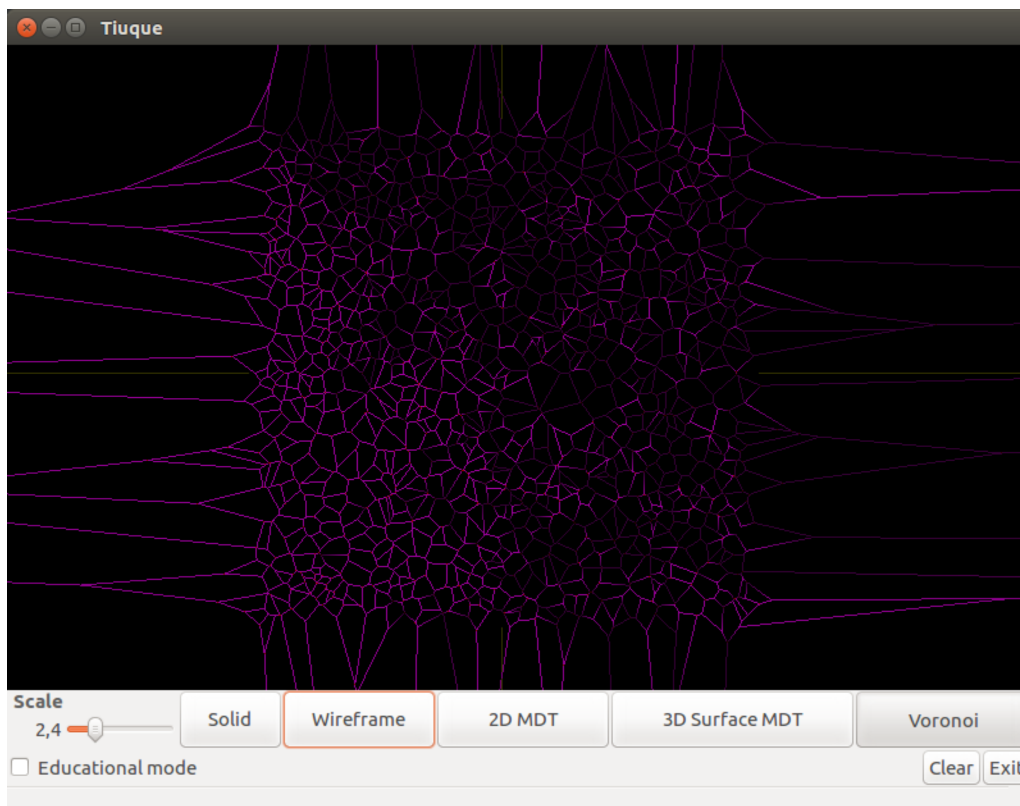

Figura 6.15: Teselación de Voronoi de malla 6.13

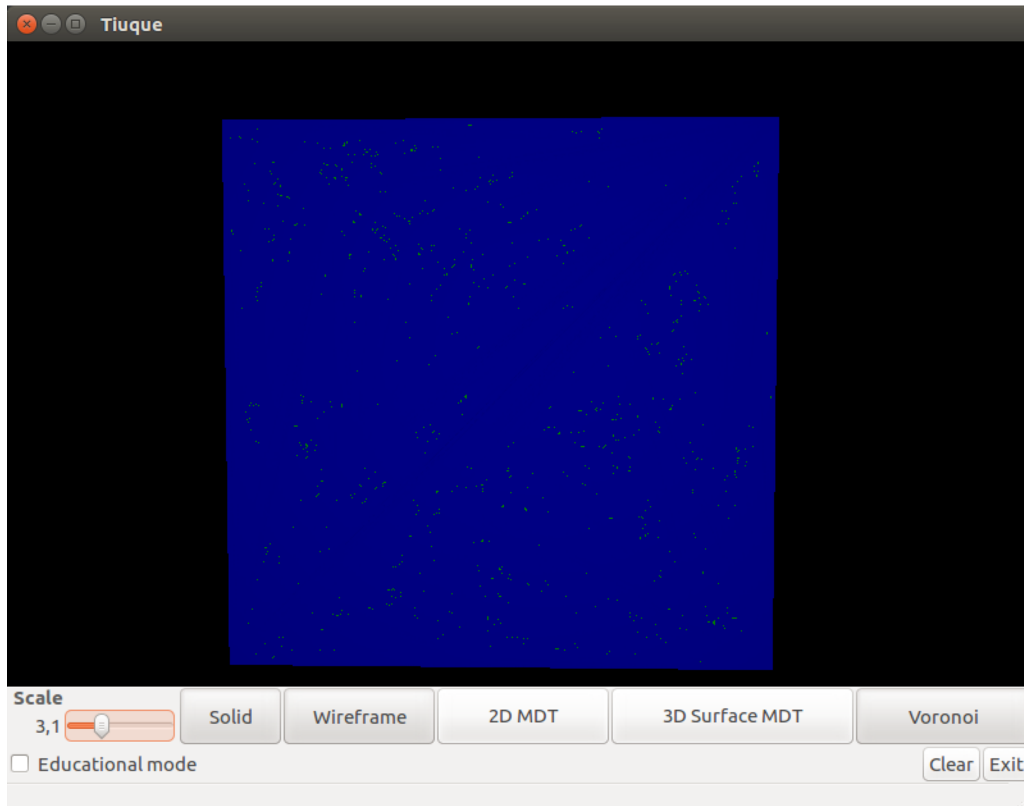

Figura 6.16: Malla de triángulo de orden de 10^5 puntos

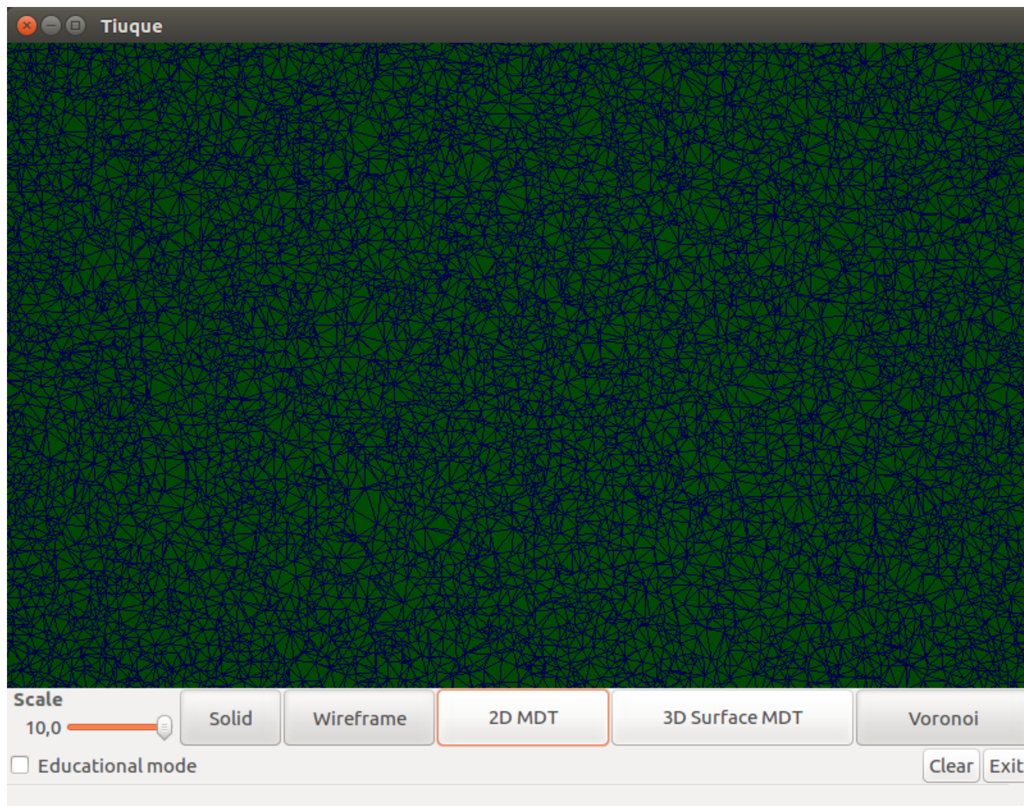

Figura 6.17: Zoom a Triangulación de Delaunay de malla 6.16

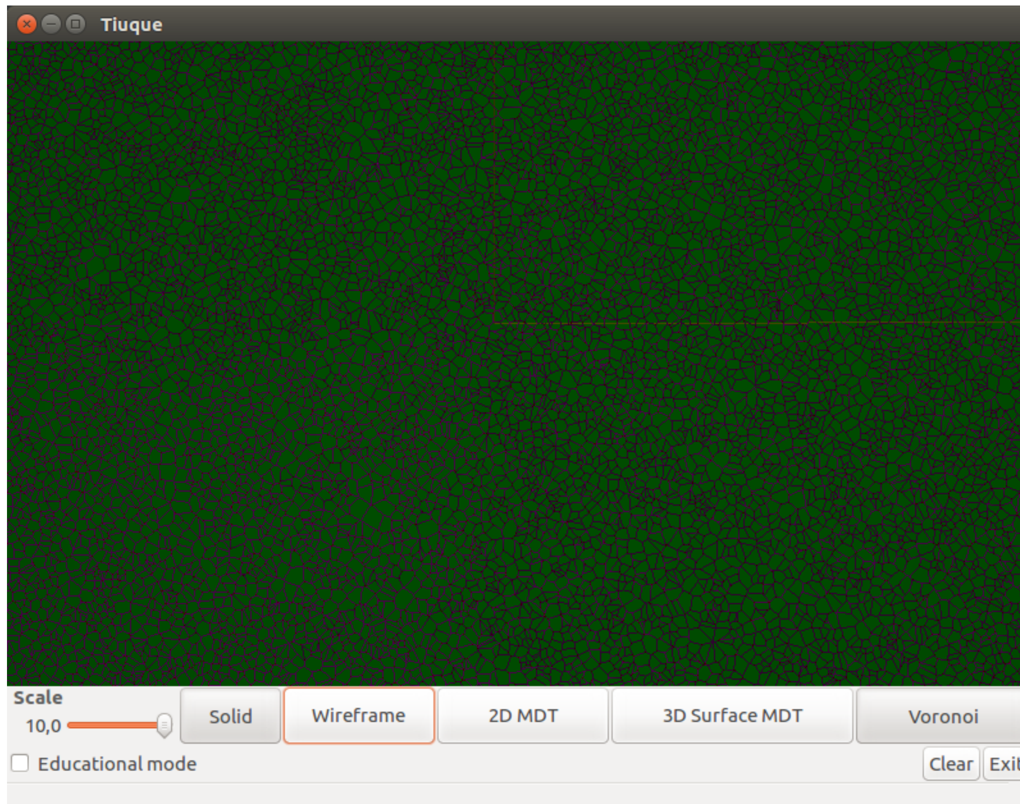

Figura 6.18: Zoom a Teselación de Voronoi de malla 6.16

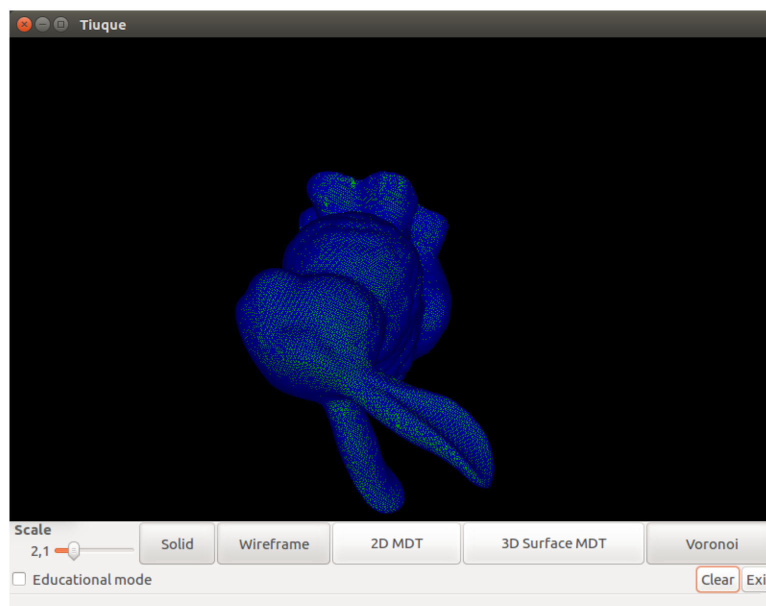

Figura 6.19: Malla de triángulo con 38.000 puntos, representando la superficie de un conejo

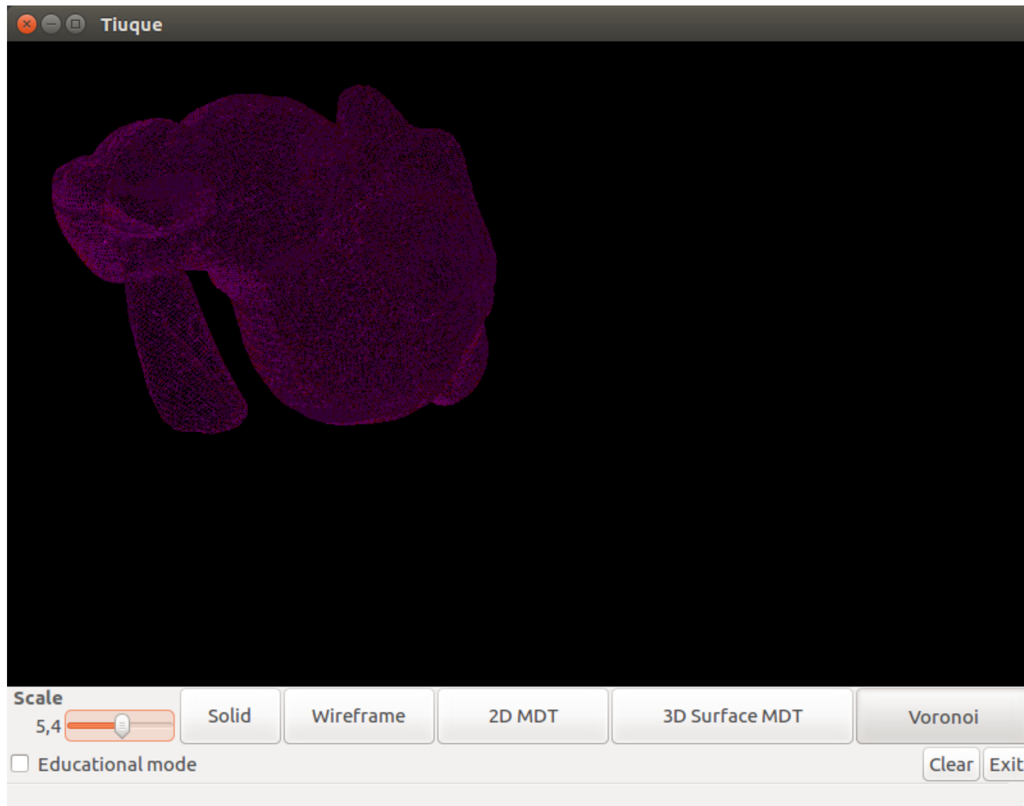

Figura 6.20: Teselación de Voronoi de malla 6.19

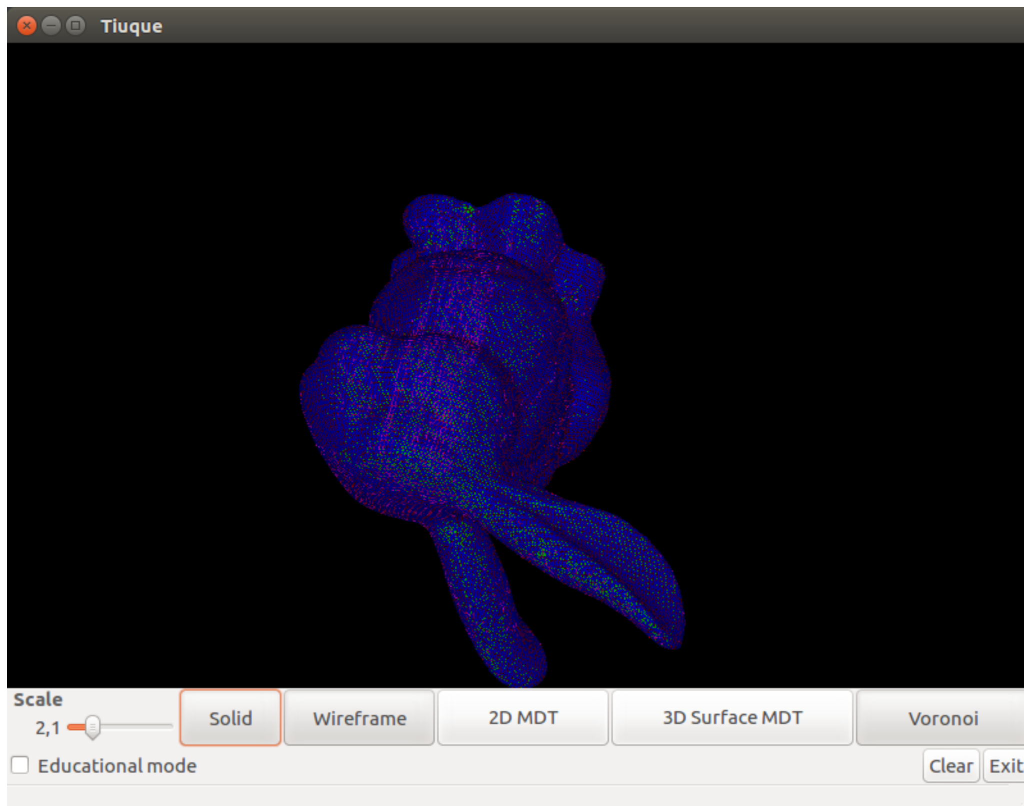

Figura 6.21: Delaunay y Voronoi de malla 6.19

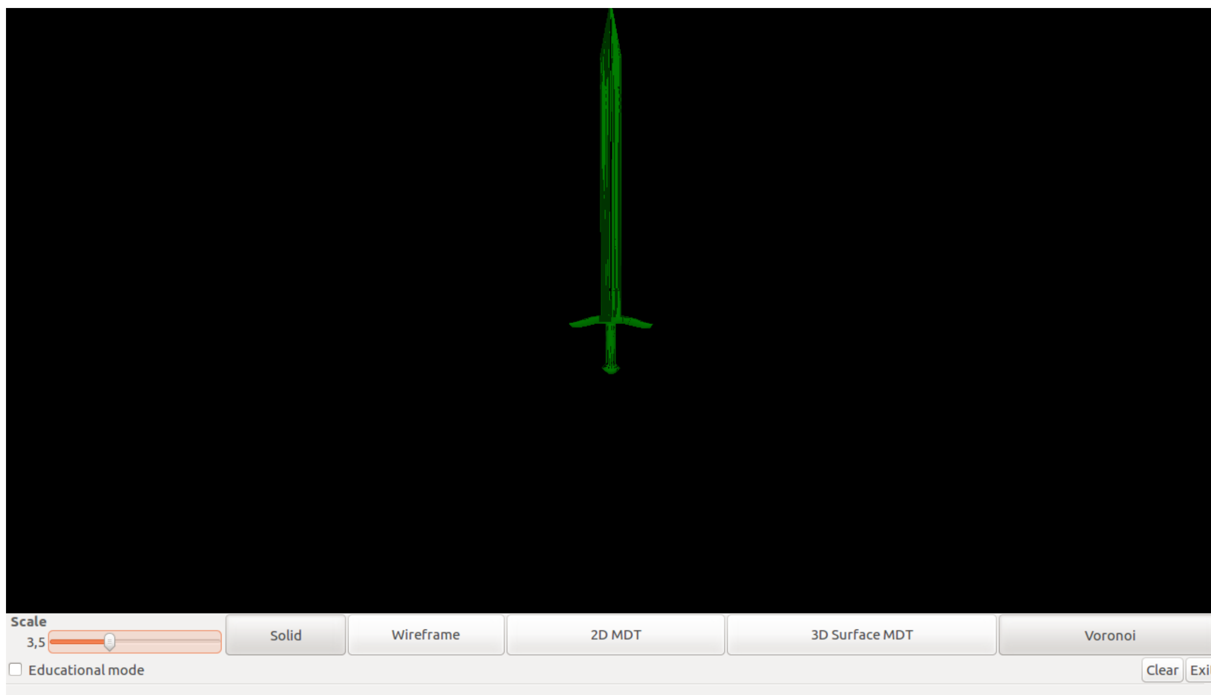

Figura 6.22: Malla de triángulo con 526 puntos, representando la superficie de una espada

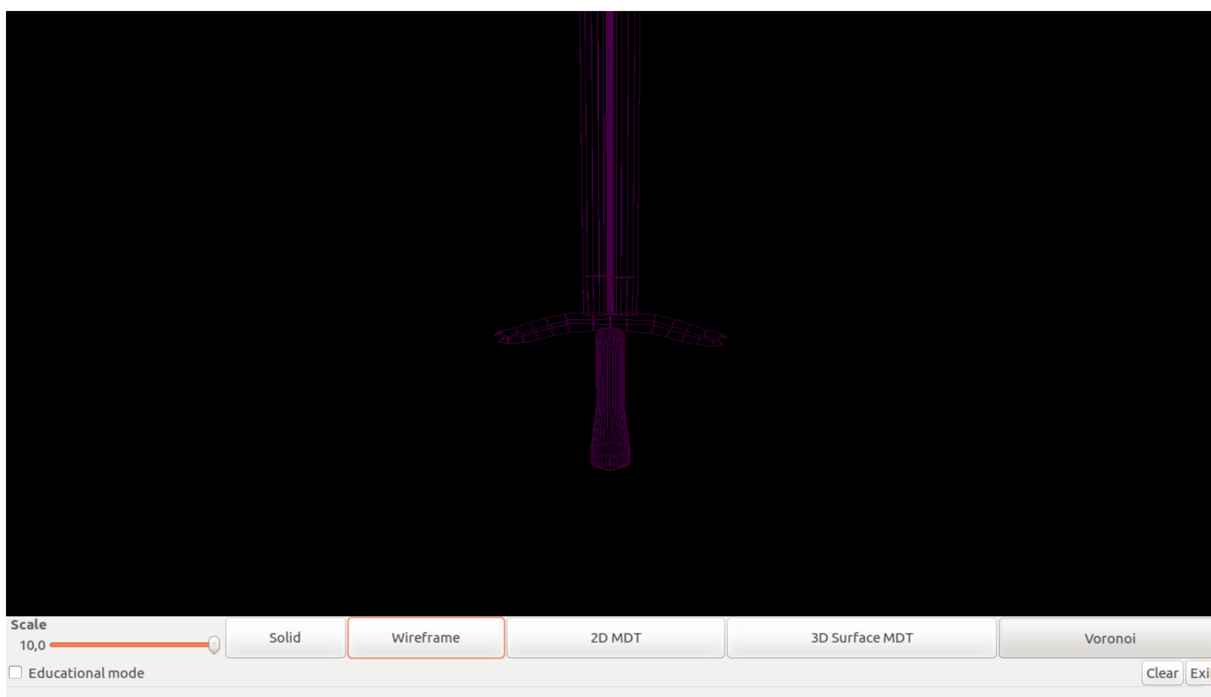

Figura 6.23: Teselación de Voronoi de malla 6.22

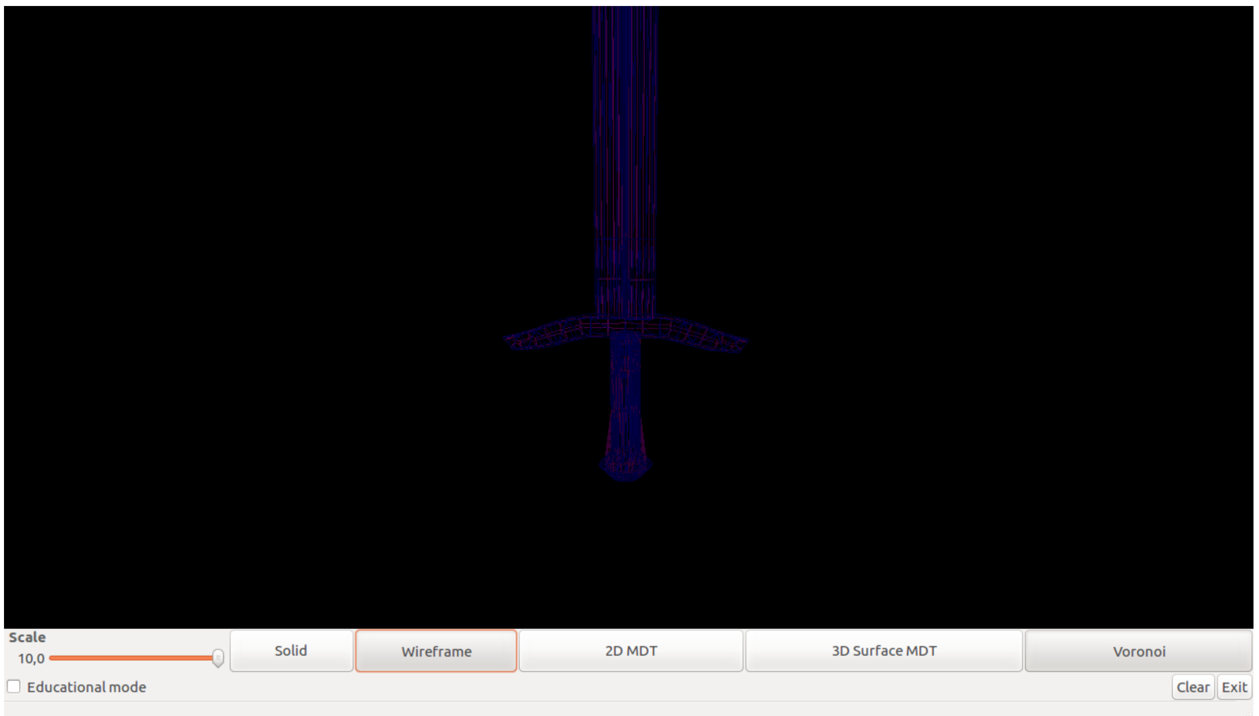

Figura 6.24: Delaunay y Voronoi de malla 6.22

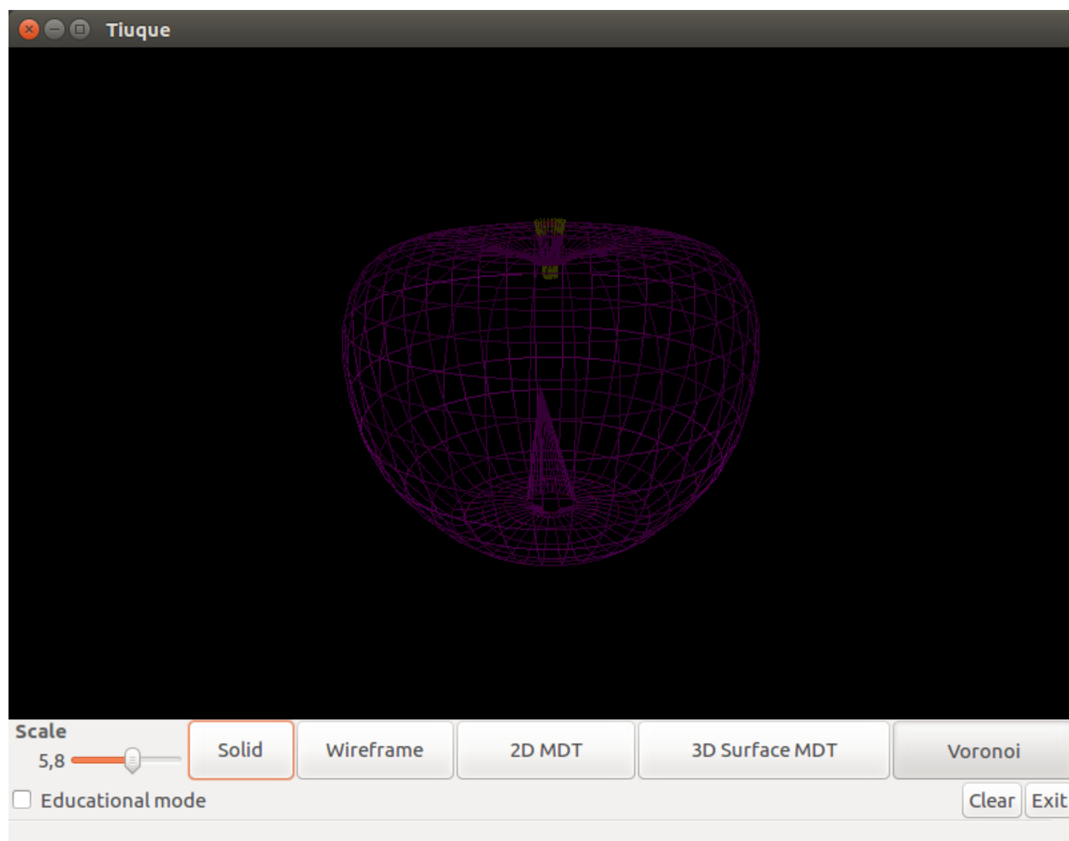

Figura 6.25: Teselación de Voronoi de malla 5.16