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

PARALELIZACIÓN DE ALGORITMOS DE MALLAS GEOMÉTRICAS EN GPU

MEMORIA PARA OPTAR AL TÍTULO DE INGENIERO CIVIL EN COMPUTACIÓN

VALENTIN LEONARDO MUÑOZ APABLAZA

PROFESOR GUÍA:
NANCY HITSCHFELD KAHLER

MIEMBROS DE LA COMISIÓN:
JEREMY BARBAY
MAURICIO PALMA LIZANA

Este trabajo ha sido parcialmente financiado por el proyecto Fondecyt 1120495

SANTIAGO DE CHILE
2014

RESUMEN DE LA MEMORIA PARA OPTAR AL
TÍTULO DE: INGENIERO CIVIL EN COMPUTACIÓN
POR: VALENTIN LEONARDO MUÑOZ APABLAZA
FECHA: 2014
PROF. GUÍA: SR. NANCY HITSCHFELD KAHLER

PARALELIZACIÓN DE ALGORITMOS DE MALLAS GEOMÉTRICAS EN GPU

La resolución de diversos problemas en ciencia e ingeniería, requiere el apoyo de soluciones y herramientas computacionales que permitan representar, visualizar y modelar sus objetos de estudio, como superficies, terrenos o células. Una forma de representar estos objetos es mediante el uso de mallas geométricas, sobre las cuales se realizan operaciones y simulaciones para modelar los problemas inherentes a cada disciplina.

Uno de los principales problemas asociados a trabajar con mallas geométricas, es el tiempo que demoran en ser procesadas. Con el auge de las tarjetas y procesadores gráficos (GPU), se han investigado nuevas técnicas que permitan usar el poder de computo de estas unidades, para desarrollar e implementar estos algoritmos.

Actualmente se cuenta con una librería (llamada Cleap), la cual permite realizar la operación de triangulación de Delaunay en Paralelo usando GPU's de marca Nvidia. A ella, se desea integrar otros algoritmos que trabajen con mallas geométricas, como algoritmos de suavizado y simplificación, además de comparar su rendimiento y calidad con otras implementaciones ya existentes.

En este trabajo, se investigó sobre algoritmos de suavizado, triangulación y simplificación de mallas geométricas, y luego se implementaron versiones de los dos primeros, los cuales fueron integrados en Cleap, y se comparó el rendimiento y calidad de sus soluciones. Con respecto al algoritmo de simplificación, solo se llegó hasta la fase de investigación teórica, pero se obtuvo la información y conocimientos necesarios para implementar e integrar una versión de este algoritmo.

Los resultados muestran que el uso de la GPU permite reducir considerablemente los tiempos de ejecución, cuando se trabaja con mallas de gran tamaño, en comparación a sus contrapartes secuenciales, y que la calidad de sus resultados es similar o incluso mejor a la de las implementaciones conocidas actualmente. Estos resultados también muestran que no siempre lo que se espera teóricamente, ocurre en la práctica, debido a problemas y fallos que ocurren al realizar cálculos con error asociado, y detalles particulares asociados a una arquitectura o plataforma determinada.

A mi familia, amigos, y mascotas.

Agradecimientos

En primer lugar, quisiera agradecer a la Profesora Nancy Hitschfeld y a Cristóbal Navarro por todo el apoyo y ayuda brindada durante el desarrollo de este trabajo de titulo y la vida académica.

También, agradecer a Ivan Rojas, por la ayuda y documentación inicial brindada, que permitió que conociera rápidamente como funciona y se programa en el "mundo" de la GPU.

Además, quisiera agradecer el financiamiento de este trabajo, a la Comisión Nacional de Investigación Científica y Tecnológica CONICYT, a través del proyecto Fondecyt 1120495.

Por supuesto, agradecer a mi Familia, en especial a mi Madre y Padre, que sin ellos y el apoyo y ayuda brindados a lo largo de mi vida, probablemente no estaría escribiendo este documento.

Un apartado especial, se lo llevan mis perros y mis gatos, que me han brindado momentos de alegrías y ventanas de distracción durante mi vida universitaria, en especial a mi gato *Toffee*(Q.E.P.D), el mas dorado, tierno, cariñoso e inspirador de todos.

Tampoco puedo olvidar agradecer a mis amigos y compañeros, que que me ayudaron, ya sea con una palabra, una sonrisa o una alegría, durante todo este camino recorrido a lo largo de mi historia. No los nombro a todos, simplemente porque la lista sería muy larga, y solo tengo una plana para escribir los agradecimientos.

También agradecer a los docentes y funcionarios del DCC, en especial a Sandra Gaez y Angélica Aguirre, por toda la ayuda académica y administrativa brindada en el transcurso de esta aventura.

Finalmente, agradecer al Centro de Alumnos del Departamento de Ciencias de la Computación (CaDCC), por brindarme un espacio en su oficina para trabajar durante el desarrollo de este trabajo.

Tabla de Contenido

Índice de tablas

Índice de figuras

Índice de algoritmos

Capítulo 1

Introducción

1.1. Antecedentes Generales

La resolución de diversos problemas en ciencia e ingeniería, en especial, los que necesitan representar terrenos o superficies de objetos, requieren el apoyo de soluciones computacionales. Estas soluciones deben permitir modelar, procesar y visualizar el dominio del problema, y ser un apoyo en la resolución de los problemas propios de estas disciplinas. Una manera de abordar estos problemas es mediante el uso de mallas geométricas las cuales permiten modelar de manera aproximada el dominio del problema, para posteriormente realizar operaciones o simulaciones sobre estas.

La temática a desarrollar durante la memoria se enmarcara en el ámbito de investigar diversos algoritmos que operan sobre mallas geométricas, en particular:

1. El algoritmo de triangulación de Delaunay[1], en particular el que convierte una triangulación cualquiera en una que satisface la condición de Delaunay.
2. Algoritmo de simplificación de mallas basado en *edge-collapse*[9], el cual consiste en colapsar arcos que no satisfacen algún criterio de calidad. Cada uno de estos arcos se reduce a un vértice en la malla.
3. Algoritmo de suavizado de mallas de Taubin [15], el cual utiliza técnicas de procesamiento de señales (como filtros e interpolación), aplicados a mallas geométricas.

Para estos algoritmos, se busca implementar versiones paralelas de ellos, y comparar su rendimiento con otras implementaciones, o con sus contrapartes secuenciales.

[1]La condición de Delaunay dice que el circuncírculo de cada triángulo de la malla no contiene en su interior a ningún otro vértice de la malla, lo cual asegura que los ángulos al interior de los triángulos son lo más grande posible.

1.2. Motivación

Las mallas geométricas son una potente herramienta que permite modelar fenómenos reales, y con esto, resolver numéricamente problemas que no tienen resolución analítica. Sin embargo, el procesamiento de mallas geométricas que manejan y representan grandes volúmenes de datos, tales como el modelamiento de superficies de terrenos para la Geofísica, se ha tornado inmanejable o lento cuando se usan algoritmos secuenciales. Entonces, se hace necesario investigar e implementar versiones de estos algoritmos que permita el procesamiento en paralelo de estos datos, de tal manera de mejorar tiempos y procesos.

Junto a esto, está la aparición en el último tiempo de tarjetas gráficas (de ahora en adelante, GPU) de alto poder de procesamiento paralelo y bajo costo, las que permitirán eventualmente que una implementación de estos algoritmos en paralelo pueda ser utilizada en una gran gama de computadores y aplicaciones. Además los principales fabricantes de estas GPU's proveen poderosas herramientas para el desarrollo de aplicaciones en paralelo, como Nvidia CUDA [2] o ATI Heterogeneous Computing [1].

Actualmente, existe una implementación del algoritmo de Delaunay [12] en GPU, disponible como una librería llamada Cleap [11], implementada por un alumno de la Universidad Austral como parte de su memoria de título, y por otro lado, existe otra implementación del algoritmo de Delaunay en CPU multi-core [7], desarrollada por Josef Kohout [8]. Junto con estar implementadas en arquitecturas paralelas distintas, otra diferencia entre ambas, es que en Cleap se utiliza el criterio de los ángulos opuestos para determinar cuándo un arco debe hacer *flip*, mientras que en Kohout se utiliza el criterio del Determinante. Esto se explica con más detalle en el Capítulo 4. Además, existen otros algoritmos, como los de simplificación y suavizado de mallas, sobre los cuales se desea tener una versión paralela que funcione en GPU, e integrarlas en la librería *opensource* Cleap.

1.3. Objetivos

1.3.1. Objetivo General

Paralelizar algoritmos de des-refinamiento y suavizado de mallas triangulares en planos y superficies, y comparar diferentes implementaciones en paralelo, del algoritmo de triangulación de Delaunay.

1.3.2. Objetivos Específicos

1. Adaptar el algoritmo de triangulación de mallas de Delaunay sobre GPU [12], para compararlo con la implementación multi-core en CPU[8].
2. Diseñar una estrategia para paralelizar el algoritmo de simplificación basado en *edge-collapse*[9] en GPU.

3. Diseñar e implementar una estrategia para paralelizar el algoritmo de Taubin [4] de suavizado de mallas sobre GPU.
4. Comparar el desempeño de las distintas implementaciones en paralelo del algoritmo de Delaunay.
5. Comparar el desempeño del algoritmo de Taubin en sus versiones secuencial y paralela.
6. Validar las fases del algoritmo de *edge-collapse* en paralelo.

1.4. Contenido de la Memoria

En el Capítulo 1 se presentan los antecedentes generales del trabajo, la motivación de su desarrollo y sus objetivos. En el Capítulo 2 se revisan las tecnologías y recursos que son transversales a estos tres algoritmos. En el Capítulo 3, se aborda el algoritmo de Taubin de suavizado de mallas geométricas, y su implementación. En el Capítulo 4, se aborda el algoritmo de triangulación de Delaunay, su implementación en paralelo, y su comparación con la implementación de Kohout. Luego, en el Capítulo 5, se aborda el algoritmo de *edge-collapse* en paralelo, y su propuesta de implementación. Finalmente, en el Capítulo 6, se presentan las principales conclusiones de este trabajo, junto al trabajo futuro que se puede realizar a partir de él.

Capítulo 2

Antecedentes

En este capítulo, se explicará el trasfondo teórico asociado a los algoritmos paralelos en GPU, que son transversales a los tres algoritmos a diseñar e implementar.

2.1. Arquitectura de la GPU

Para entender como está estructurada la arquitectura de la GPU, realizaremos una comparación entre CPU y GPU.

Por un lado se tiene que en la CPU, gran parte del espacio físico destinado se utiliza para colocar memoria caché, y poca parte del espacio para colocar ALU's[1]. Debido a esto, la latencia de la memoria disminuye considerablemente al tener una jerarquía de memoria caché. También se tiene que el procesamiento en paralelo vía hardware se encuentra limitado por la baja cantidad de núcleos presentes en una CPU.

Por otro lado, gran parte del espacio físico destinado a la GPU se utiliza para colocar ALU's, pero a cambio se obtienen memorias cache más pequeñas. Esto produce que la latencia de la memoria sea alta, y sea necesario tener muchos hilos de ejecución activos para disminuirla. Además se tiene que los hilos de ejecución se activan por bloques, por lo que las aplicaciones a realizar en GPU deben considerar este factor para obtener una mayor eficiencia de la arquitectura. En la Figura 2.1 se muestra una comparación de la distribución de espacio en ambos tipos de arquitectura.

Como se adelantó anteriormente, y como se puede apreciar en detalle de la Figura 2.2, la GPU está compuesta por bloques de hilos de ejecución.

Aún más en detalle, en la Figura 2.3 se presenta en detalle una abstracción de estos bloques, en el cual se puede apreciar los distintos niveles de memoria a los que tiene acceso cada hilo de ejecución. Se tiene que cada hilo de ejecución tiene acceso de lectura y escritura

[1] Arithmethic Logic Unit (Unidad Aritmético Lógica, en español), la cual se encarga de realizar operaciones aritméticas, como sumas o multiplicaciones, y operaciones lógicas, como or o and.

a registros y memoria local, la cual es limitada, pero rápida. Luego por cada bloque, todos los hilos que pertenezcan a él tienen acceso de lectura y escritura a un módulo de memoria compartida, el cual es rápido, siempre y cuando las lecturas sean sobre un mismo sector y las escrituras sobre sectores distintos. Finalmente todos los hilos tienen acceso de lectura y escritura a la memoria global y acceso solo lectura a un módulo de memoria constante.

Ambos tipos de memoria son relativamente lentas. En particular, si cada hilo de ejecución realiza una operación de lectura sobre la memoria global, los tiempos de ejecución se pueden ver impactados considerablemente. Sin embargo, si cada bloque realiza una lectura completa de un sector de la memoria, y la copia a la memoria compartida, entonces se puede obtener una optimización en la lectura y uso de datos para los hilos de ese bloque.

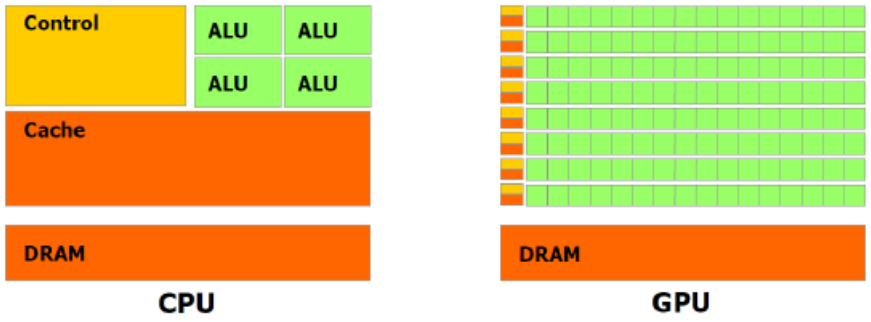


Figura 2.1: Comparación de la utilización de espacio en las arquitecturas de CPU y GPU.

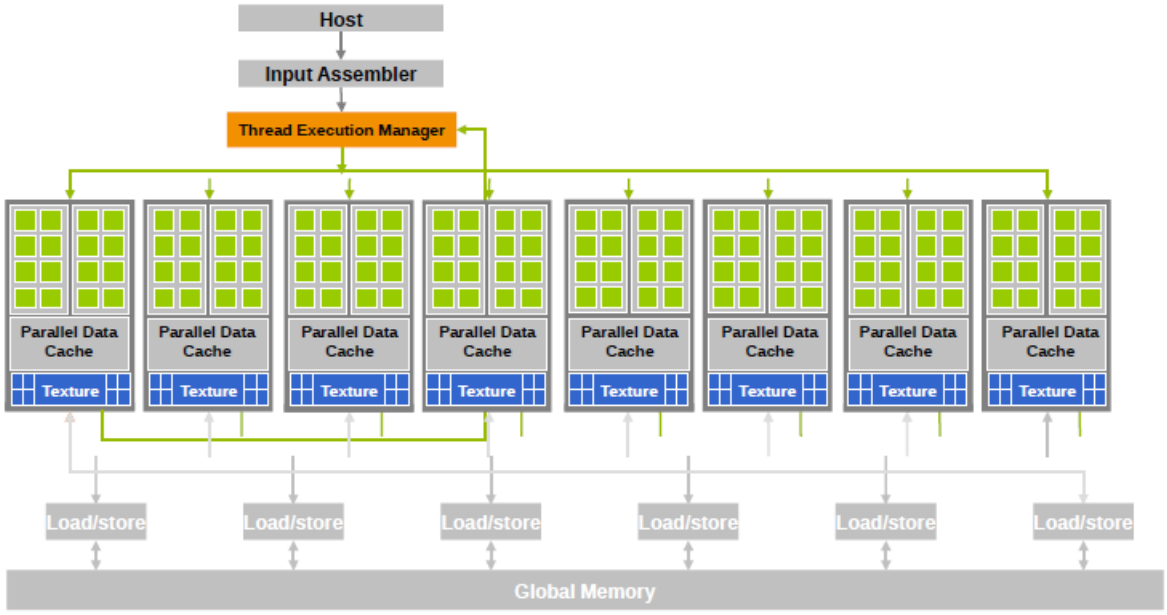


Figura 2.2: Configuración típica de una GPU, en la cual cada bloque de hilos de ejecución tiene acceso a memoria compartida entre ellos, y además una memoria global, a la cual pueden acceder todos los hilos.

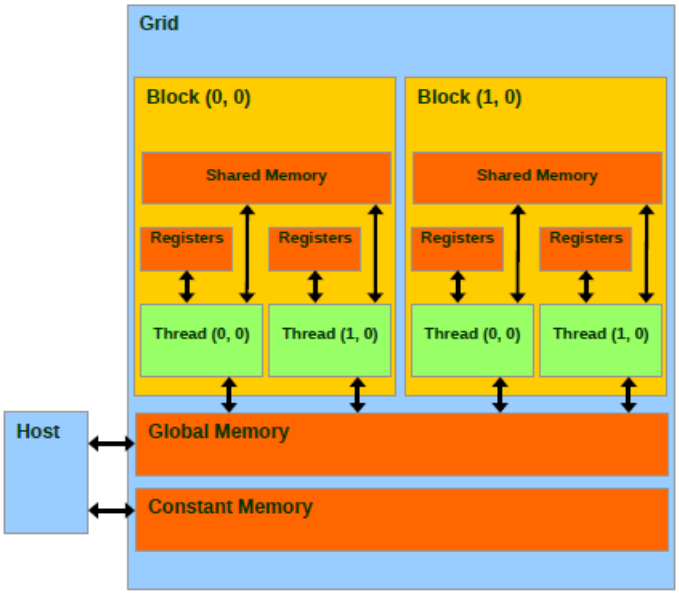


Figura 2.3: Abstracción de un bloque de ejecución de una GPU, en el cual se aprecian los distintos niveles de memoria a los que los hilos de ejecución tienen acceso.

2.2. Programación en GPU

Una decisión importante a tomar, al momento de programar en GPU, es el lenguaje que se utilizará. Esto, debido a que algunos lenguajes requieren hardware exclusivo y/o dedicado, mientras que otros son más flexibles. Por otro lado, algunos están optimizados para una arquitectura en particular, mientras que otros pueden funcionar en diversas variedades de hardware. A continuación, se revisarán los tres lenguajes/frameworks de programación en GPU más importantes.

2.2.1. Nvidia CUDA

NVIDIA *Compute Unified Device Architecture* (CUDA)[2], es una plataforma de computación paralela y un modelo de programación que permite incrementar el procesamiento de las aplicaciones, debido al paralelismo que se obtiene al utilizar una gran cantidad de hilos de ejecución provistos por esta plataforma. Las ventajas son que la plataforma se encarga de paralelizar y el cómo paralelizar, por lo que el programador no debe preocuparse demasiado de este aspecto. La desventaja es que solo funciona en dispositivos que tengan una tarjeta de video marca NVIDIA, pero debido a esto, la plataforma se encuentra optimizada para este tipo de hardware.

2.2.2. AMD APP SDK

AMD *Accelerated Parallel Processing* (APP)[1], es un kit de funciones de software y hardware, que permiten usar en conjunto el poder de cómputo de la CPU y la GPU de una manera eficiente para esta arquitectura. Esto es posible, debido a que en esta arquitectura en particular, la CPU y la GPU se encuentran en un mismo chip, conocido como APU[2], lo que da la ventaja de reducir las latencias asociadas a la comunicación entre CPU y GPU, y permite tener un bloque de memoria compartida entre ambas unidades. La desventaja, es que esto solo está disponible para dispositivos que cuenten con un procesador, y una tarjeta gráfica de marca AMD. Además, su poder de cómputo es ligeramente menor en comparación a la competencia.

2.2.3. OpenCL

Open Computing Language (OpenCL)[3], es una plataforma de computación paralela, libre y abierta, que tiene la ventaja de funcionar en una mayor gamma de dispositivos, en comparación con las anteriores alternativas. Esto debido a que no está acoplado a una arquitectura o hardware específico. Por lo mismo, la desventaja es que no se encuentra optimizado, ni puede aprovechar todos los recursos disponibles de una arquitectura en particular.

2.2.4. Tipo de Programación Elegida

Luego de revisar estas tres alternativas, se decide escoger la opción de NVIDIA CUDA, debido a dos principales razones. La primera, es que el computador donde se realizarán las implementaciones y las pruebas, cuenta con una tarjeta de vídeo marca NVIDIA con soporte CUDA. La segunda, es que los algoritmos de la librería Cleap, ya se encuentran implementados en CUDA. Con esto, es esperable que sea más sencillo implementar e integrar los algoritmos a desarrollar en esta librería, y no se deba re-implementar todo lo que ya está hecho en otro lenguaje.

2.3. Representación de matrices en GPU

Una problemática transversal a los algoritmos mencionados anteriormente, es la elección de las estructuras de datos que utilizarán los algoritmos y su posterior representación en GPU. En particular, para una malla geométrica es necesario para diversos cálculos, que cada vértice conozca cuantos vértices vecinos tiene, y quienes son estos vértices vecinos. Esto se puede representar a través de una Matriz de Grados y una Matriz de Adyacencia respectivamente. Se estudiaron diversas alternativas de representación de matrices *sparse*[3], ya que es usual que en

[2]Accelerated Processing Unit, serie de microprocesadores marca AMD, que poseen una CPU y una GPU en el mismo circuito integrado.

[3]Matrices poco densas, es decir, que gran parte del contenido en sus celdas es cero.

una malla geométrica, cada vértice tenga una cantidad acotada de vecinos, considerablemente menor al número total de vértices de la malla. Estas alternativas estudiadas [4] se listan a continuación:

2.3.1. Formato Diagonal (DIA)

En esta representación, la matriz se descompone en una matriz y un arreglo. La matriz *data*, almacena en cada columna, los valores de cada diagonal de la matriz que tenga al menos un valor distinto de cero, y el arreglo de *offsets*, que indica a que distancia, o bien, con cuanto desfase se encuentra la diagonal almacenada en la columna con respecto a la diagonal principal. El valor del *offset* es positivo para las súper-diagonales y negativo para las sub-diagonales. Como las diagonales son de distinto tamaño, se utiliza un valor arbitrario (que no esté presente en la matriz) para marcar que ese casillero no se encuentra en la matriz original. En particular, cuando se está representando una Matriz de Adyacencia o de Grados, este valor arbitrario puede ser cualquier numero negativo. En la Figura 2.4 se puede ver un ejemplo de esta representación.

$$\mathtt{A} = \begin{bmatrix} 1 & 7 & 0 & 0 \\ 0 & 2 & 8 & 0 \\ 5 & 0 & 3 & 9 \\ 0 & 6 & 0 & 4 \end{bmatrix}$$

$$\mathtt{data} = \begin{bmatrix} * & 1 & 7 \\ * & 2 & 8 \\ 5 & 3 & 9 \\ 6 & 4 & * \end{bmatrix} \qquad \mathtt{offsets} = \begin{bmatrix} -2 & 0 & 1 \end{bmatrix}$$

Figura 2.4: Representación en el formato DIA de una matriz A.

Los beneficios de esta representación es que los accesos a memoria a la matriz *data* son contiguos, lo que mejora la eficiencia de lecturas y escrituras en memoria, y cuando los valores distintos de cero se encuentran agrupados en un numero acotado de diagonales. Por otro lado, tiene las desventajas de que almacena los valores cero que estén presentes en la diagonal. Además, cuando la matriz presenta pocas diagonales densas, como se puede apreciar en la Figura 2.5, esta representación es ineficiente en la memoria utilizada.

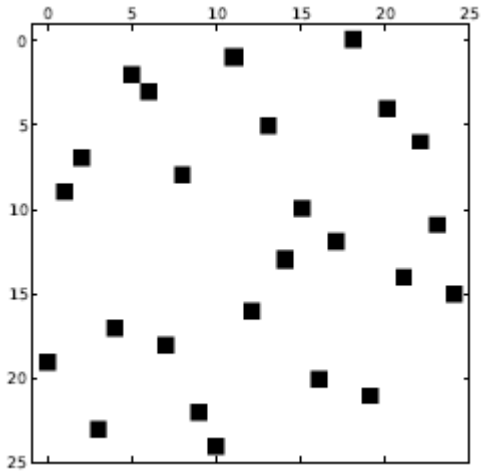


Figura 2.5: Matriz poco densa, que en formato DIA seria representada ineficientemente.

2.3.2. Formato ELLPACK (ELL)

En esta representación, para una matriz de $M \times N$,se cuentan para cada fila, el número de los valores distintos de cero. El mayor de estos números de denominará K. Luego, la matriz de $M \times N$ se descompone en una matriz *data* de $M \times K$ y una matriz de *índices*. En *data* se almacenan los valores de cada fila distintos de cero, mientras que en la matriz índices se almacena el índice de la columna de cada valor de la matriz *data* que le correspondía en la matriz original. En la Figura 2.6 se puede ver un ejemplo de esta representación.

$$\mathtt{A} = \begin{bmatrix} 1 & 7 & 0 & 0 \\ 0 & 2 & 8 & 0 \\ 5 & 0 & 3 & 9 \\ 0 & 6 & 0 & 4 \end{bmatrix}$$

$$\mathtt{data} = \begin{bmatrix} 1 & 7 & * \\ 2 & 8 & * \\ 5 & 3 & 9 \\ 6 & 4 & * \end{bmatrix} \qquad \mathtt{indices} = \begin{bmatrix} 0 & 1 & * \\ 1 & 2 & * \\ 0 & 2 & 3 \\ 1 & 3 & * \end{bmatrix}$$

Figura 2.6: Representación en el formato ELL de la matriz A.

La ventaja de este formato se produce cuando la cantidad de valores distintos de cero por fila es cercano al promedio de estos, lo que produce una representación eficiente en uso de memoria. La desventaja se produce cuando la diferencia entre la fila con mayor cantidad de valores distintos de cero y el promedio es arbitrariamente alta, lo que genera un gasto en memoria innecesario. En la Figura 2.7 se puede apreciar un caso en el que se presenta este fenómeno.

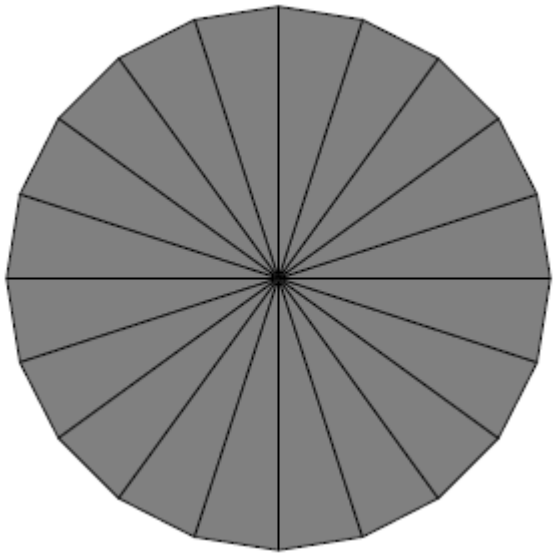


Figura 2.7: Dado que el vértice central tiene un alto grado de vecinos con respecto a los demás vértices, en formato ELL esta figura sería ineficientemente representada.

2.3.3. Formato Coordinado (COO)

En esta representación, la matriz es almacenada en tres arreglos, llamados *columnas*, *filas* y data. Por cada celda de la matriz, en estos tres arreglos, se almacena el valor de la celda en data, y el índice de la fila y la columna de esta celda en *filas* y *columnas*. En la Figura 2.8 se puede ver un ejemplo de esta representación.

$$\mathtt{A} = \begin{bmatrix} 1 & 7 & 0 & 0 \\ 0 & 2 & 8 & 0 \\ 5 & 0 & 3 & 9 \\ 0 & 6 & 0 & 4 \end{bmatrix}$$

$$\begin{aligned} \mathtt{row} &= [0 \quad 0 \quad 1 \quad 1 \quad 2 \quad 2 \quad 2 \quad 3 \quad 3] \\ \mathtt{col} &= [0 \quad 1 \quad 1 \quad 2 \quad 0 \quad 2 \quad 3 \quad 1 \quad 3] \\ \mathtt{data} &= [1 \quad 7 \quad 2 \quad 8 \quad 5 \quad 3 \quad 9 \quad 6 \quad 4] \end{aligned}$$

Figura 2.8: Representación en el formato COO de la matriz A.

La ventaja de esta representación es que el almacenamiento requerido es proporcional al número de valores distintos de cero en la matriz original. La desventaja es que, salvo que se haga un ordenamiento por filas o columnas previamente, no se garantiza que valores que antes eran contiguos se almacenen contiguamente, lo que puede producir problemas al momento de implementar los algoritmos o requerir un acceso a memoria.

2.3.4. Formato Híbrido (HYB)

Esta representación consiste en una combinación del formato ELL y el formato COO, en el cual a diferencia del formato ELL tradicional, el valor de K corresponde a la menor cantidad o al valor promedio de celdas distintas de cero en una fila. Luego la matriz se almacena en el formato ELL, y cuando un valor no alcanza a ser almacenado en este formato, es almacenado en la representación COO. De esta manera, se almacena de manera eficiente el número típico de valores distintos de cero en el formato ELL, mientras que las celdas excepcionales por fila se almacenan en el formato COO.

La ventaja es el uso eficiente de ambos formatos, y que corrige las desventajas del formato ELL, y que permite aprovechar de mejor manera algunas topologías de mallas geométricas. Además la matriz ELL en este caso será mucho más densa que en el caso tradicional, lo que aumenta su eficiencia y optimización en memoria. Las desventajas ocurren cuando se deben almacenar demasiados valores excepcionales en la matriz COO, puesto que hay que manejar y procesar estos valores en el algoritmo final, lo que agrega tiempo de procesamiento adicional al algoritmo, lo que puede ser indicador de que el uso de otro formato puede ser más eficiente.

2.3.5. Elección de Formato

Finalmente, luego de analizar todas estas alternativas, se toma la decisión de utilizar el Formato ELLPACK (ELL) para la representación de matrices en los algoritmos que lo requieran. La motivación principal es que es una estructura de datos para matrices aplicable a diversos casos de mallas geométricas. Esto permitirá abordar este problema de manera eficiente, puesto que a priori las mallas a procesar presentan un comportamiento favorable para este formato, en comparación a los demás.

2.4. Estructura de Datos

Otro aspecto importante a considerar, es la estructura de datos a utilizar, en donde se almacenará la información relevante de las mallas geométricas que los algoritmos trabajaran y procesaran. En este aspecto en particular, se decidió extender la estructura de datos usada por la librería Cleap [12]. Esta estructura consiste en una representación de vértices, triángulos y arcos de una malla geométrica. Los vértices se representan como un arreglo unidimensional en el que cada posición almacena un vector con las coordenadas (x, y, z) del vértice en el espacio cartesiano. Por otro lado, los triángulos se representan como un arreglo unidimensional que contiene índices al arreglo de vértices, y donde se tiene que tres puntos consecutivos en este arreglo, definen un triángulo. Finalmente, los arcos se representan como un arreglo unidimensional de estructuras de arcos. Esta estructura contiene un par de índices a los vértices que componen este arco, y otro par de índices al arreglo de triángulos, que indica los triángulos que utilizan a este arco. Esta información redundante en el arreglo de arcos, permite comprobar la integridad y consistencia de la malla luego de las operaciones sobre esta. La composición gráfica de esta estructura de datos, se puede ver en la Figura 2.9.

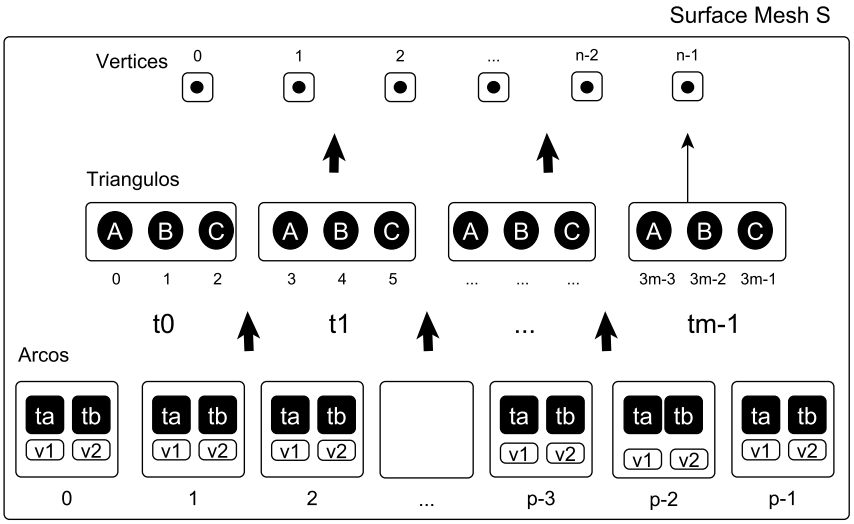


Figura 2.9: Estructura de Datos original de la librería Cleap.

Las razones principales por la que se decidió extender esta estructura de datos, en vez de empezar de cero con una nueva estructura, son:

1. Esta estructura de datos maneja toda la información de vértices, triángulos y arcos, que requieren los tres algoritmos con los que se trabajará.
2. Esta estructura ya se encuentra integrada en la arquitectura de Cleap, por lo que esta librería está optimizada para ella, y debiese ser más sencillo operar a partir de ella.

Como se mencionó anteriormente, es necesario añadir una extensión a esta estructura de datos para añadir soporte a las necesidades particulares de los algoritmos a integrar en la librería. Por el lado del Algoritmo de Taubin, es necesario añadir lo siguiente:

1. Un arreglo V de tamaño $O(vertices)$, que permita almacenar el numero de vértices vecinos que tiene cada vértice.
2. Una matriz de adyacencia A, de tamaño $N \times K$, que en cada fila almacena índices a los vértices vecinos correspondientes al índice de la fila en cuestión.

Estos agregados se pueden apreciar en la Figura 2.10. En particular, para el segundo caso, se decidió que la matriz A, fuese implementada en el formato ELLPACK, donde N es el número de vértices total, y K es el máximo número de vecinos que puede poseer un vértice de la malla. Así, es posible comprimir y ahorrar espacio en memoria, y tanto N como K se pueden obtener fácilmente de las estructuras ya existentes. Finalmente, es posible representar esta matriz como un arreglo unidimensional de tamaño $N \times K$, en donde cada K posiciones, se almacena la siguiente fila de la matriz original. Esto ultimo se realiza, pues es más sencillo y rápido de manejar y cargar en la memoria de la GPU, un arreglo, que una matriz. En caso de que un vértice tenga una cantidad de vecinos menores a K, los espacios restantes en este arreglo se rellenan con un valor arbitrario tal que, no represente ningún índice valido en la malla (como -1, por ejemplo).

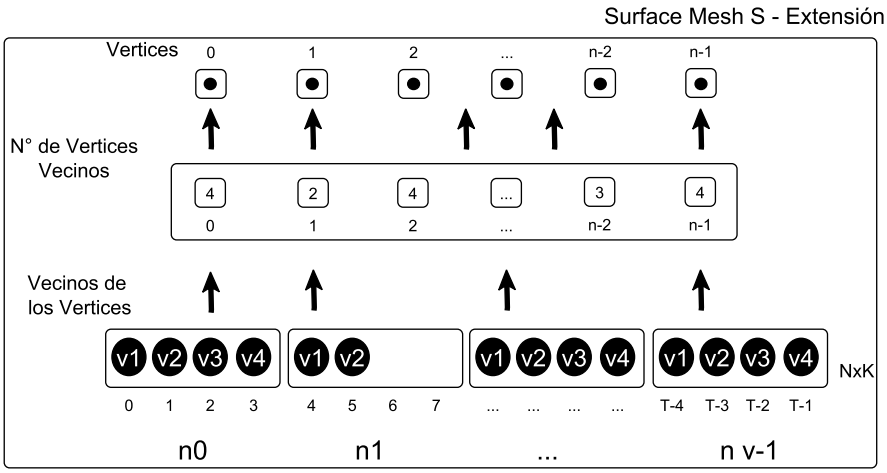


Figura 2.10: Extensión de la estructura de datos de la librería Cleap.

Capítulo 3

Algoritmo de Suavizado de Taubin

En este capítulo, se abordará el algoritmo de suavizado de mallas de Taubin [15]. Primero se explorará la teoría que existe detrás de este algoritmo. Luego se estudiaran sus implementaciones secuencial y paralela. Finalmente se compararán ambas implementaciones en pruebas cuantitativas y cualitativas.

3.1. Trasfondo Teórico

El algoritmo de Taubin para suavizado de superficies[15], consiste en utilizar técnicas de procesamiento de señales, en particular, técnicas de filtros de frecuencia, con el objetivo de disminuir o suavizar el ruido y distorsiones presentes en una señal. Luego, la base teórica de este algoritmo postula que es posible disminuir las perturbaciones y distorsiones existentes en una superficie geométrica, si se le aplica un tratamiento mediante filtros, de manera de realizar algo similar a lo que se hace en el campo de trabajo con señales para eliminar el ruido en ellas.

Para esto, Taubin postula que aplicar técnicas relacionadas con el Análisis de Fourier de estas señales, permite identificar las componentes de la señal que son datos y las que son ruido. Con esto, y haciendo la analogía al campo de las mallas geométricas, es posible descomponer las coordenadas de los vértices de la superficie de la malla en los tres ejes cartesianos principales (x, y, z), por lo que el análisis a partir de ahora se puede realizar para una coordenada arbitraria, y posteriormente se puede replicar para las demás.

Antes de describir el algoritmo de Taubin, es necesario describir algunos conceptos provenientes del campo de Procesamiento y Análisis de Señales, lo cual se realiza en la siguiente sección.

3.1.1. Señales y Filtros

Una señal es un tipo de onda generada por algún fenómeno electromagnético, la cual es representable por una función matemática, en donde sus variables son el espacio (amplitud) y tiempo (periodo). Dado que generalmente, estas señales provienen de mediciones realizadas a fenómenos reales, también es necesario considerar un ruido aleatorio, asociado a esta señal. La siguiente ecuación describe matemáticamente una señal.

$$x_s = x(x,t) + n(x,t)$$

Para eliminar este ruido, se utilizan filtros de frecuencia, los cuales permiten eliminar bandas en el espectro de frecuencias. Un filtro de frecuencia es una función $f(w)$ en el dominio de frecuencias, tal que:

$$x'(w) = x_s(w) * f(w)$$

Donde $x'(w)$ corresponde a la señal original, luego de aplicado el filtro $f(w)$ Un método de filtrado ideal, es conocido como descriptores de Fourier, el cual consiste en descomponer la señal original en una base de vectores propios $u_1, ..., u_n$, asociados a las frecuencias presentes en la señal:

$$x(w) = \sum_{\mathrm{i}=1} s_{\mathrm{i}} u_{\mathrm{i}}(w)$$

Esta representación, es equivalente a la Transformada Rápida de Fourier[1], y de esta descomposición, se eliminan las componentes asociadas a las frecuencias y ruidos que se desean eliminar. Usualmente el ruido se encuentra ligado a las perturbaciones de alta frecuencia, por lo que lo ideal es eliminar estas componentes de alta frecuencia.

Sin embargo, el cálculo asociado al Análisis de Fourier es costoso computacionalmente (lo cual es reafirmado por el autor del algoritmo), por lo que se ofrece la alternativa de utilizar filtros pasa-bajo[2] para lograr un resultado aproximado. Dado que el filtro pasa-bajo elimina las componentes de alta frecuencia de una señal, en teoría es una alternativa viable y sencilla de utilizar para resolver este problema en particular.

3.1.2. Algoritmo de Taubin

Antes de presentar la construcción teórica del algoritmo, es necesario introducir algunas definiciones, para comprender mejor lo que se quiere construir:

[1] FFT, Fast Fourier Transform

[2] Un filtro pasa-bajo, es uno que deja pasar las frecuencias bajas y atenúa las frecuencias altas en un espacio muestral determinado

- Superficie Poliedral: Se define como un par de listas $S = V, F$, donde V es una lista que contiene a los vértices del poliedro, y F es una lista que contiene las caras del poliedro.
- Vecindad: Se define como el conjunto i^* de vértices, que comparten un arco con un vértice i particular.
- Señal discreta superficial: Se define como una función, que se aplica sobre los vértices de una superficie poliedral.
- Laplaciano Discreto: Se define como la discretización del operador diferencial continuo Δ, para abarcar el caso de dominios finitos y grillas.

Con esto, primero definimos el Laplaciano discreto de una señal discreta superficial, como:

$$\Delta x_{\mathrm{i}} = \sum_j \omega_{\mathrm{i}j}(x_j - x_{\mathrm{i}})$$

En donde x_{i} es la coordenada del vértice a procesar, x_j representa a los vértices vecinos de x_{i}, y $\omega_{\mathrm{i}j}$ se denominan pesos, que representan cuanta ponderación e importancia tiene el vértice vecino sobre el vértice en cuestión. Una buena medida de pesos, es que todos los vecinos ponderen lo mismo sobre este vértice ($\omega_{\mathrm{i}j} = 1/|\mathrm{i}^*|$), pero existen otras medidas más sofisticadas, que consideran el largo del arco que los une, a los vecinos de los vecinos, etc, las cuales son mas costosas de computar, pero en teoría dan resultados mas exactos. La ecuación anterior se puede expresar de forma matricial de la siguiente manera:

$$\Delta x = (W - I)x$$

En donde W es la matriz de pesos (representable por una matriz de adyacencia), I es la matriz identidad, y x es la señal discreta superficial. Con esto, se aplica una aproximación del filtro pasa-bajo, representada por la siguiente expresión:

$$x' = f(K)^N x$$

Donde x' es la señal luego de aplicado el filtro, K representa la matriz circulante[3] del problema, $f(K)$ representa la función de transferencia del filtro, y N el número de iteraciones que se aplica el filtro. La función de transferencia propuesta por el autor, es la siguiente:

$$f(k) = (1 - \lambda k)(1 - \mu k)$$

Finalmente, todos estos pasos, se pueden reducir a aplicar consecutivamente las siguientes ecuaciones, que representan dos etapas de filtrado, para cada punto de la malla.

$$x'_{\mathrm{i}} = x_{\mathrm{i}} + \lambda \Delta x_{\mathrm{i}}$$

[3]Una matriz circulante corresponde a la matriz de grados de un grafo, restándole la matriz de adyacencia del mismo grafo.

$$x'_{\mathrm{i}} = x'_{\mathrm{i}} + \mu \Delta x_{\mathrm{i}}$$
$$\mu < -\lambda$$

En donde λ representa el factor de crecimiento, y μ representa el factor de decrecimiento, los cuales se utilizan para evitar que aplicaciones sucesivas del filtro compriman o expandan demasiado la superficie de la malla. La aplicación de este par de fórmulas se puede apreciar en la Figura 3.1, un ejemplo de funcionamiento esperado de este algoritmo se puede apreciar en la Figura 3.2, y en los próximos capítulos se abordara tanto la implementación secuencial en CPU, como la implementación en paralelo en GPU de este algoritmo.

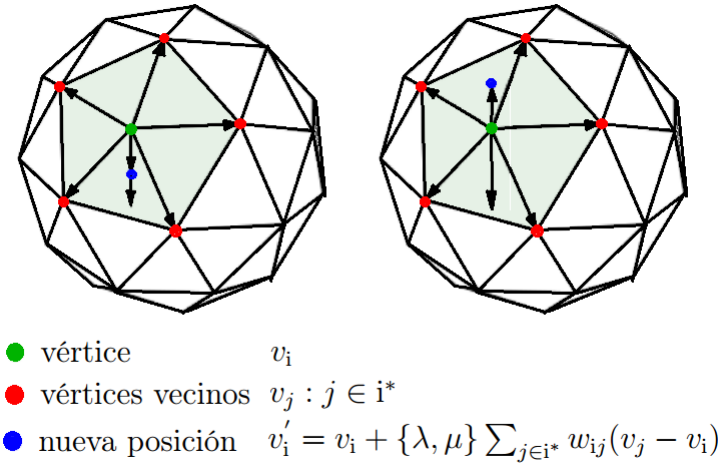
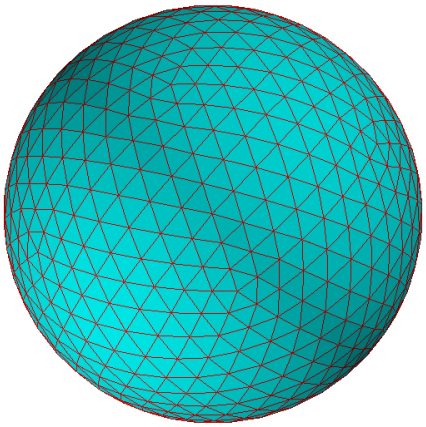
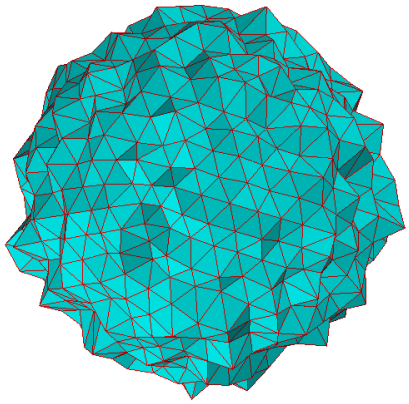


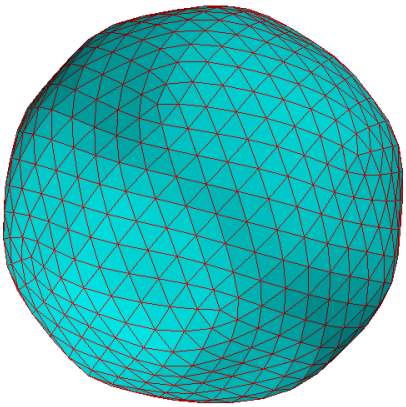
Figura 3.1: Los 2 pasos de la aplicación de este algoritmo, la primera, con un factor λ positivo, y la segunda con un factor μ negativo, se aplica a todos los vértices de la malla.



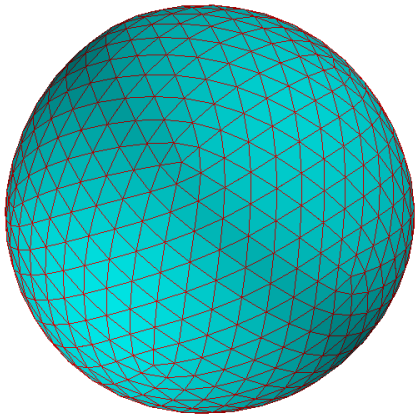
(a) Esfera sin deformaciones.



(b) Esfera con deformaciones.



(c) Esfera deformada luego de 10 iteraciones del algoritmo.



(d) Esfera deformada luego de 100 iteraciones del algoritmo.

Figura 3.2: Resultados del algoritmo de suavizado

3.2. Versión Secuencial en CPU

Luego de conocer el trasfondo teórico necesario para poder implementar de buena manera el algoritmo, procedemos a analizar las componentes principales del algoritmo a implementar. De la sección anterior, se deduce que el algoritmo se compone de 4 etapas principales:

1. Calcular el Laplaciano discreto para un vértice x (suma ponderada)
2. Desplazar (expandir) el vértice, en función del resultado anterior, y la ponderación λ
3. Calcular nuevamente el el Laplaciano discreto para el vértice desplazado (suma ponderada)
4. Desplazar (contraer) nuevamente el vértice, en función del resultado anterior, y la ponderación μ

Estas cuatro etapas se repiten por una cantidad determinada de veces o *Iteraciones*, hasta lograr el resultado esperado. También tenemos que notar que las etapas 1 y 3 son similares, y las etapas 2 y 4 son iguales, salvo el factor de ponderación utilizado. Para poder implementar estas etapas sin mayores dificultades, es necesario utilizar la extensión a la estructura de datos de Cleap, descrita en la Figura 2.10, la cual nos provee la información de la cantidad de vecinos de un vértice, y quienes son estos vecinos.

Con esto, podemos implementar las etapas 1 y 3, descritas en el Algoritmo 1:

Algoritmo 1: Taubin: etapas 1 y 3, suma ponderada de vértices vecinos

```
Data: Arreglos: Vertices[N], NumeroVecinos[N], Vecinos[N×K]
Result: Arreglo: SumaPonderada[N]
SumaPonderada.x = 0;
SumaPonderada.y = 0;
SumaPonderada.z = 0;
for i = cada vértice en el arreglo Vertices[N] do
    for j = cada vértice vecino de i en el arreglo Vecinos[N×K] do
        SumaPonderada[i].x += Vertices[j].x / NumeroVecinos[i];
        SumaPonderada[i].y += Vertices[j].y / NumeroVecinos[i];
        SumaPonderada[i].z += Vertices[j].z / NumeroVecinos[i];
    end
end
```

Para este algoritmo, tenemos que para cada coordenada x, y, z de cada vértice, se genera una suma ponderada, en función de sus vértices vecinos, que representa la aplicación del laplaciano discreto para cada vértice. Conociendo estos valores, ahora podemos proceder a

implementar las etapas 2 y 4, descritas en el Algoritmo 2:

Algoritmo 2: Taubin: etapas 2 y 4, Desplazamiento del vértice

```
Data: Arreglos: Vertices[N], SumaPonderada[N]
Result: Arreglo: Vertices[N]
p = Ponderador: {λ, μ};
for i = cada vértice en el arreglo Vertices[N] do
    Vertices[i].x = Vertices[i].x + p*SumaPonderada[i];
    Vertices[i].y = Vertices[i].y + p*SumaPonderada[i];
    Vertices[i].z = Vertices[i].z + p*SumaPonderada[i];
end
```

Para este algoritmo, tenemos que, se aplica el desplazamiento del vértice, en función del valor de la suma ponderada calculada anteriormente para cada vértice. Este desplazamiento puede ser de expansión o contracción, dependiendo si se usa el ponderador λ o μ. Con esto, podemos finalmente tener el algoritmo secuencial de Taubin, descrito en el Algoritmo 3:

Algoritmo 3: Pseudo-codigo Algoritmo Secuencial de Taubin

```
Data: Archivo .OFF con la malla a suavizar
Result: Arreglo: Vertices[N]
Leer y almacenar puntos en la estructura de datos de Cleap;
Generar Arreglo de Vecinos[N×K] y Numero de vecinos[N];
iter = iteraciones a realizar;
while i <iter do
    Calcular sumaPonderada con Algoritmo 1.;
    Calcular expansión de los vértices con Algoritmo 2 y factor λ;
    Calcular sumaPonderada con Algoritmo 1.;
    Calcular contracción de los vértices con Algoritmo 2 y factor μ;
    i++;
end
```

Cabe mencionar que dada esta construcción del algoritmo, tenemos que teóricamente demora aproximadamente tiempo $O(N \times K \times Iteraciones) \approx O(N^3)$, debido a que existen 3 ciclos anidados dentro de este algoritmo, y el largo de estos ciclos depende de la cantidad de Vértices N, el numero máximo de vecinos que puede tener un vértice K, y la cantidad de *Iteraciones* que se desea aplicar el suavizado. Por lo que en un principio, no seria una alternativa eficiente para resolver este problema. En los capítulos siguientes se estudiará la implementación paralela de este algoritmo, y posteriormente se compararan los resultados de ambas implementaciones.

3.3. Versión Paralela en GPU

Para implementar una versión paralela de este algoritmo, primero tenemos que analizar que etapas del algoritmo se pueden realizar en paralelo. Para ello, recordemos las etapas, ya descritas en el capitulo anterior:

1. Calcular el Laplaciano discreto para un vértice x (suma ponderada)
2. Desplazar (expandir) el vértice, en función del resultado anterior, y la ponderación λ
3. Calcular nuevamente el el Laplaciano discreto para el vértice desplazado (suma ponderada)
4. Desplazar (contraer) nuevamente el vértice, en función del resultado anterior, y la ponderación μ

Dado que para cada una de estas etapas, es necesario tener el calculo realizado en la etapa anterior, no es posible realizar una paralelización que involucre realizar cada una de estas etapas de manera simultanea. Lo interesante esta en cada una de estas etapas por si sola, en donde se realiza un ciclo que recorre todos los vértices de la malla. Con esto, es natural pensar en una optimización tal que en cada etapa del algoritmo, se realice un procedimiento paralelo en GPU, en el cual cada thread de la GPU procesa un vértice de la malla, lo cual permite realizar en un ciclo de GPU, la totalidad de ciclos que se deben realizar en CPU para procesar todos los vértices.

Con esto en mente, procedemos a presentar la implementación de las etapas 1 y 3 en GPU, descritas en el Algoritmo 4

Algoritmo 4: Taubin: etapa 1 y 3 en paralelo, suma ponderada de vértices vecinos

```
Data: Arreglos: Vertices[N], NumeroVecinos[N], Vecinos[N×K]
Result: Arreglo: SumaPonderada[N]
i = thread.id if i <N then
    SumaPonderada.y = 0;
    SumaPonderada.y = 0;
    SumaPonderada.z = 0;
    for j = cada vértice vecino de i en el arreglo Vecinos[N×K] do
        SumaPonderada[i].x += Vertices[j].x / NumeroVecinos[i];
        SumaPonderada[i].y += Vertices[j].y / NumeroVecinos[i];
        SumaPonderada[i].z += Vertices[j].z / NumeroVecinos[i];
    end
end
```

Aquí notamos algunos elementos distintos con respecto al algoritmo secuencial 1. Primero notar que se removió el ciclo que recorre todos los vértices, ya que este código solo procesa un solo vértice, y este se ejecuta simultáneamente para todos ellos. Además, dado que cada thread en la GPU se identifica con un ID, es necesario realizar una comprobación para que los threads que tengan una ID superior al numero de Vértices, no realicen ningún cálculo.

Ahora procedemos a presentar la implementación de las etapas 2 y 4 en GPU, descritas en el Algoritmo 5:

Algoritmo 5: Taubin: etapa 2 y 4 en paralelo, Desplazamiento del vértice

```
Data: Arreglos: Vertices[N], SumaPonderada[N], p = Ponderador: {λ, μ}
Result: Arreglo: Vertices[N]
i = thread.id if i <N then
    Vertices[i].x = Vertices[i].x + p*SumaPonderada[i];
    Vertices[i].y = Vertices[i].y + p*SumaPonderada[i];
    Vertices[i].z = Vertices[i].z + p*SumaPonderada[i];
end
```

Al igual que en el algoritmo anterior, es necesario comprobar que los threads que tengan una ID superior al numero de Vértices, no realicen ningún calculo. Por lo mismo, y dado que cada thread solo procesa un vértice, podemos eliminar el ciclo que recorre todos los vértices.

Finalmente podemos presentar el algoritmo de Taubin en Paralelo, descrito en el Algoritmo 6:

Algoritmo 6: Pseudo-código Algoritmo de Taubin en Paralelo

```
Data: Archivo .OFF con la malla a suavizar
Result: Arreglo: Vertices[N]
Leer y almacenar puntos en la estructura de datos de Cleap;
Generar Arreglo de Vecinos[N×K] y Numero de vecinos[N];
iter = iteraciones a realizar;
Copiar todos estos datos y estructuras a la memoria de la GPU while i <iter do
    Calcular sumaPonderada con Algoritmo 4.;
    SincronizarGPU();
    Calcular expansión de los vértices con Algoritmo 5 y factor λ;
    SincronizarGPU();
    Calcular sumaPonderada con Algoritmo 4.;
    SincronizarGPU();
    Calcular contracción de los vértices con Algoritmo 5 y factor μ;
    SincronizarGPU();
    i++;
end
Copiar los datos y resultados de la memoria de la GPU a la memoria RAM principal.
```

Con esta construcción del algoritmo, teóricamente tenemos que demora aproximadamente tiempo $O(K \times Iteraciones) \approx O(N^2)$, debido a que se disminuye un orden de magnitud, al reemplazar el ciclo que recorre todos los vértices de forma secuencial, por un procedimiento en paralelo que los recorre todos simultáneamente. Este tiempo es mejor que su contraparte secuencial, pero también tenemos que tener en cuenta dos factores importantes, que añaden un costo constante importante. Estos son, inicializar la GPU con los datos a procesar, y sincronizar los threads en la GPU. Realizar estos pasos es necesario, ya que, por un lado, la GPU no puede leer la memoria principal (RAM), por lo que es necesario copiar todos estos

datos a la memoria de la GPU, para que sean procesados allí. Por otro lado, es necesario sincronizar los threads luego de ejecutar cada Algoritmo en paralelo, ya que, como vimos al principio de esta sección, no se puede iniciar una etapa sin tener los resultados de la etapa anterior. Por lo que para evitar *Data-Races*[4], se debe poner una condición de espera, de tal manera de iniciar la siguiente etapa cuando todos los threads de la GPU hayan terminado de procesar.

En el siguiente capitulo de resultados, se compararán y estudiarán los resultados obtenidos con ambas implementaciones.

3.4. Resultados

Para probar el funcionamiento de los algoritmos, se realizó una batería de pruebas, que se describe a continuación:

- Se generaron 9 Ico-esferas[5] de radio 2, con distinto número de vértices, cuyas características se pueden ver en la Tabla 3.1
- Para cada una de estas esferas, se aplicaron 5 niveles distintos de deformación en su superficie.Esta deformación consiste en desplazar el vértice con respecto a su centro, una distancia no mayor al radio de la esfera, alejándose o acercándose a su centro (aleatorio).
- Para cada una de estas esferas, se ejecutó el algoritmo, con distinta cantidad de iteraciones: 10, 50, 100, 500, 1000 y 5000.
- En total, se cuentan con 45 Mallas de prueba, y cada una de ellas, produce 6 mallas resultantes.

Ico-esfera	# Vértices	# Triángulos
2	42	80
3	162	320
4	642	1.280
5	2.562	5.120
6	1.0242	20.480
7	40.962	81.920
8	163.842	327.680
9	655.362	1.310.720
10	2.621.442	5.242.880

Tabla 3.1: Número de vértices para cada ico-esfera

[4]Múltiples procesos o hilos de ejecución se encuentran en un *Data-Race*, si el resultado final de todos ellos depende de su orden de ejecución. No identificar y manejar estas condiciones, puede producir resultados incoherentes o corruptos.

[5]Una Ico-Esfera es una esfera formada solo por triángulos equiláteros del mismo tamaño. Para aumentar el tamaño de una Ico-Esfera, se reemplaza un arco por un triangulo del mismo tamaño

Por otro lado, las características del Equipo donde se realizaron estas pruebas, son las siguientes:

- Procesador Intel Core® i7™-4700MQ @ 2.40GHz.
- 12GB de RAM DDR3 @ 1600MHz.
- Chip Gráfico NVIDIA Geforce 740M con 2048MB de memoria.
- Sistema Operativo Ubuntu 13.10 x64 con Ubuntu Linux kernel versión 3.11.0-12.19.

A continuación se presentan los resultados de las dos implementaciones del algoritmo. Estos resultados se separan en dos: Resultados Cuantitativos y Resultados Cualitativos.

3.4.1. Resultados Cuantitativos

En este apartado, se muestran los resultados obtenidos por ambos algoritmos. Se midieron los tiempos que demoran ambas implementaciones bajo distintos escenarios. Dado que se realizo una gran cantidad de pruebas, solo se muestran los resultados mas relevantes para obtener conclusiones.

Tiempo[s] Taubin Paralelo con 500 iteraciones

Malla		Deformación				
Ico-esfera	Vértices	Nivel 0	Nivel 1	Nivel 3	Nivel 5	Nivel 10
2	42	0,0460	0,0459	0,0470	0,0467	0,0464
3	162	0,0483	0,0484	0,0486	0,0488	0,0482
4	642	0,0516	0,0525	0,0524	0,0524	0,0518
5	2562	0,0809	0,0795	0,0780	0,0824	0,0785
6	10242	0,2316	0,2370	0,2317	0,2384	0,2307
7	40962	0,9000	0,8982	0,8990	0,9013	0,8984
8	163842	3,4781	3,4745	3,4745	3,4795	3,4740
9	655362	13,7630	13,7530	13,7600	13,7690	13,7620
10	2621442	54,8000	54,7960	54,7970	54,8120	54,8100

Tabla 3.2: Medición de tiempo que demora el Algoritmo de Taubin en GPU para 500 iteraciones. Se aprecia que los tiempos son independientes del nivel de deformación aplicado.

Tiempo[s] Taubin Paralelo con 5000 iteraciones

Malla		Deformación				
Ico-esfera	Vértices	Nivel 0	Nivel 1	Nivel 3	Nivel 5	Nivel10
2	42	0,4357	0,4538	0,4497	0,4511	0,4509
3	162	0,4566	0,4663	0,4615	0,4589	0,4602
4	642	0,4904	0,4985	0,5000	0,4939	0,4972
5	2562	0,7744	0,7789	0,7796	0,7738	0,7792
6	10242	2,3144	2,3383	2,3227	2,3180	2,3187
7	40962	8,9894	9,0015	9,0079	8,9940	9,0035
8	163842	34,7690	34,8170	34,8030	34,7970	34,7830
9	655362	137,5000	137,5400	137,5700	137,5800	137,5200
10	2621442	548,0200	548,0400	548,2000	548,2300	547,9200

Tabla 3.3: Medición de tiempo que demora el Algoritmo de Taubin en GPU para 5000 iteraciones. Se observa que los tiempos aumentan proporcionalmente al numero de iteraciones.

Tiempo[s] Taubin Secuencial con 500 iteraciones

Malla		Deformación				
Ico-esfera	Vértices	Nivel 0	Nivel 1	Nivel 3	Nivel 5	Nivel10
2	42	0,0064	0,0064	0,0064	0,0064	0,0064
3	162	0,0272	0,0272	0,0272	0,0272	0,0272
4	642	0,1004	0,1004	0,1004	0,1004	0,1004
5	2562	0,4054	0,4054	0,4054	0,4054	0,4054
6	10242	1,6447	1,6447	1,6447	1,6447	1,6447
7	40962	6,5102	6,5102	6,5102	6,5102	6,5102
8	163842	25,9690	25,9690	25,9690	25,9690	25,9690
9	655362	104,4900	104,4900	104,4900	104,4900	104,4900
10	2621442	416,3900	416,3900	416,3900	416,3900	416,3900

Tabla 3.4: Medición de tiempo que demora el Algoritmo de Taubin en CPU para 500 iteraciones. Se observa que los tiempos son superiores en comparación a la misma prueba sobre la GPU.

Tiempo[s] Taubin Secuencial con 5000 iteraciones

Malla		Deformación				
Ico-esfera	Vértices	Nivel 0	Nivel 1	Nivel 3	Nivel 5	Nivel10
2	42	0,0628	0,0628	0,0628	0,0628	0,0628
3	162	0,2472	0,2472	0,2472	0,2472	0,2472
4	642	0,9895	0,9895	0,9895	0,9895	0,9895
5	2562	3,9837	3,9837	3,9837	3,9837	3,9837
6	10242	16,2370	16,2370	16,2370	16,2370	16,2370
7	40962	64,1520	64,1520	64,1520	64,1520	64,1520
8	163842	251,2400	251,2400	251,2400	251,2400	251,2400
9	655362	1013,1000	1013,1000	1013,1000	1013,1000	1013,1000
10	2621442	4246,3000	4246,3000	4246,3000	4246,3000	4246,3000

Tabla 3.5: Medición de tiempo que demora el Algoritmo de Taubin en CPU para 5000 iteraciones.

Tiempo[s] Taubin Paralelo para Esfera deformada Nivel 1

Deformación Nivel 1		Iteraciones					
Ico-esfera	Vértices	10	50	100	500	1000	5000
2	42	0,001	0,005	0,009	0,046	0,092	0,454
3	162	0,001	0,005	0,010	0,048	0,095	0,466
4	642	0,001	0,005	0,011	0,053	0,099	0,498
5	2562	0,002	0,008	0,017	0,080	0,155	0,779
6	10242	0,005	0,025	0,050	0,237	0,461	2,338
7	40962	0,019	0,096	0,190	0,898	1,801	9,002
8	163842	0,074	0,361	0,695	3,475	6,961	34,817
9	655362	0,287	1,383	2,758	13,753	27,511	137,540
10	2621442	1,103	5,481	10,961	54,796	109,600	548,040

Tabla 3.6: Medición de tiempo que demora el Algoritmo de Taubin en GPU para Ico-Esfera con deformación de nivel 1. Se observa a grandes rasgos que existe una relación proporcional entre el tiempo que demora y el numero de iteraciones.

Tiempo[s] Taubin Secuencial para Esfera deformada Nivel 1

Deformación Nivel 1		Iteraciones					
Ico-esfera	Vértices	10	50	100	500	1000	5000
2	42	0,0001	0,0006	0,0013	0,0064	0,0128	0,0628
3	162	0,0006	0,0026	0,0053	0,0272	0,0508	0,2472
4	642	0,0021	0,0101	0,0208	0,1004	0,2050	0,9895
5	2562	0,0079	0,0399	0,0838	0,4054	0,8159	3,9837
6	10242	0,0345	0,1573	0,3300	1,6447	3,2532	16,2370
7	40962	0,1286	0,6318	1,3047	6,5102	12,9140	64,1520
8	163842	0,5125	2,5396	5,2060	25,9690	51,6830	251,2400
9	655362	2,1034	10,4010	20,8890	104,4900	210,9600	1013,1000
10	2621442	8,4466	41,1080	82,6180	416,3900	847,7200	4246,3000

Tabla 3.7: Medición de tiempo que demora el Algoritmo de Taubin en CPU para Ico-Esfera con deformación de nivel 1. Se aprecia que los tiempos son superiores en comparación a la GPU.

Tiempo[s] Taubin Paralelo para Esfera deformada Nivel 5

Deformación Nivel 5		Iteraciones					
Ico-esfera	Vértices	10	50	100	500	1000	5000
2	42	0,001	0,005	0,009	0,047	0,093	0,451
3	162	0,001	0,005	0,010	0,049	0,097	0,459
4	642	0,001	0,005	0,011	0,052	0,105	0,494
5	2562	0,002	0,008	0,017	0,082	0,158	0,774
6	10242	0,005	0,025	0,050	0,238	0,465	2,318
7	40962	0,019	0,096	0,188	0,901	1,802	8,994
8	163842	0,074	0,360	0,696	3,480	6,963	34,797
9	655362	0,286	1,390	2,761	13,769	27,535	137,580
10	2621442	1,097	5,482	10,963	54,812	109,630	548,230

Tabla 3.8: Medición de tiempo que demora el Algoritmo de Taubin en GPU para Ico-Esfera con deformación de nivel 5.

Tiempo[s] Taubin Secuencial para Esfera deformada Nivel 5

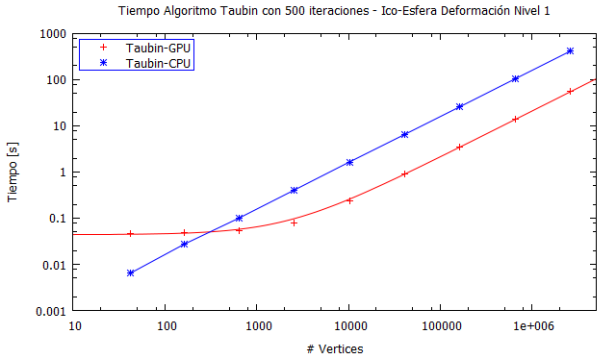
Deformación Nivel 5		Iteraciones					
Ico-esfera	Vértices	10	50	100	500	1000	5000
2	42	0,0001	0,0006	0,0013	0,0064	0,0128	0,0628
3	162	0,0006	0,0026	0,0053	0,0272	0,0508	0,2472
4	642	0,0021	0,0101	0,0208	0,1004	0,2050	0,9895
5	2562	0,0079	0,0399	0,0838	0,4054	0,8159	3,9837
6	10242	0,0345	0,1573	0,3300	1,6447	3,2532	16,2370
7	40962	0,1286	0,6318	1,3047	6,5102	12,9140	64,1520
8	163842	0,5125	2,5396	5,2060	25,9690	51,6830	251,2400
9	655362	2,1034	10,4010	20,8890	104,4900	210,9600	1013,1000
10	2621442	8,4466	41,1080	82,6180	416,3900	847,7200	4246,3000

Tabla 3.9: Medición de tiempo que demora el Algoritmo de Taubin en CPU para Ico-Esfera con deformación de nivel 5.

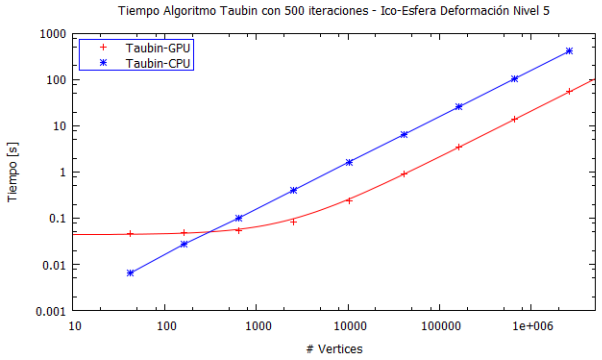
Speed-up Algoritmo Paralelo/Secuencial.

		Iteraciones					
Ico-esfera	Vértices	10	50	100	500	1000	5000
2	42	0,1	0,1	0,1	0,1	0,1	0,1
3	162	0,5	0,5	0,5	0,6	0,5	0,5
4	642	2,0	1,9	2,0	1,9	2,0	2,0
5	2562	4,7	4,8	5,1	4,9	5,2	5,1
6	10242	6,9	6,3	6,7	6,9	7,0	7,0
7	40962	6,7	6,6	7,0	7,2	7,2	7,1
8	163842	6,9	7,1	7,5	7,5	7,4	7,2
9	655362	7,3	7,5	7,6	7,6	7,7	7,4
10	2621442	7,7	7,5	7,5	7,6	7,7	7,7

Tabla 3.10: *speed-up* del algoritmo paralelo con respecto a su versión secuencial. Se observa que en promedio, el algoritmo paralelo es 6-7 veces mas rápido que su contraparte secuencial.

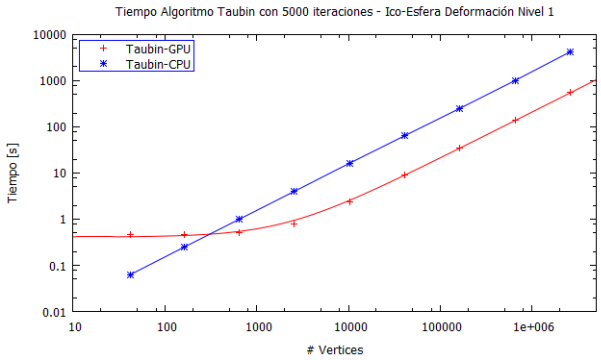


(a) Ico-Esfera Deformación Nivel 1

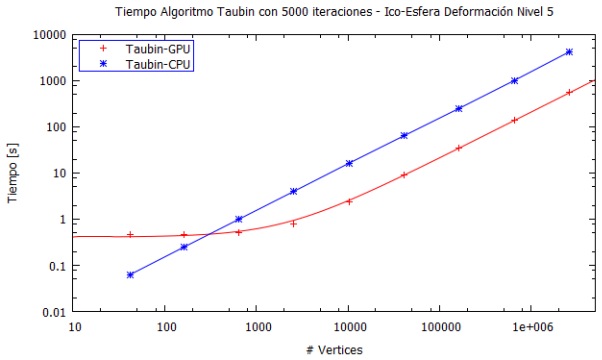


(b) Ico-Esfera Deformación Nivel 5

Figura 3.3: Gráfico tiempo[s] Algoritmo Taubin con 500 Iteraciones. Independiente del nivel de deformación, los tiempos del algoritmo secuencial tienden a ser un orden superior en comparación al algoritmo paralelo.

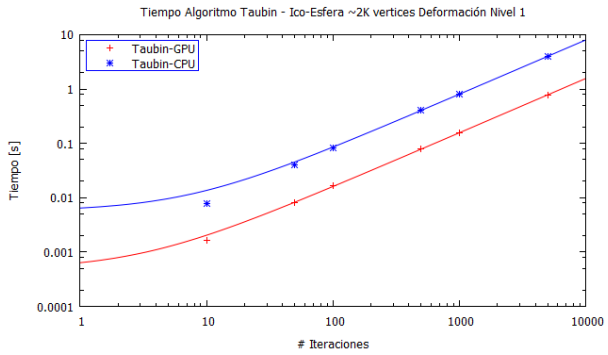


(a) Ico-Esfera Deformación Nivel 1

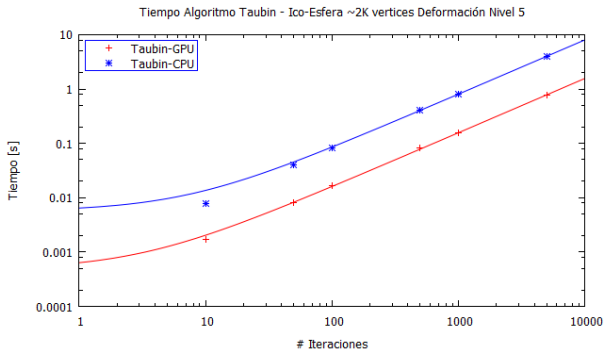


(b) Ico-Esfera Deformación Nivel 5

Figura 3.4: Gráfico tiempo[s] Algoritmo Taubin con 5000 Iteraciones. Se observa que la tendencia se mantiene al aumentar el numero de iteraciones.

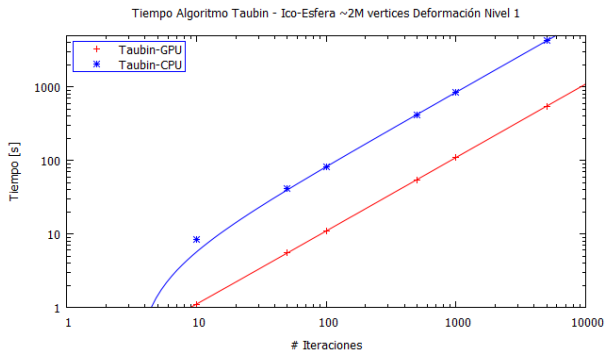


(a) Ico-Esfera Deformación Nivel 1

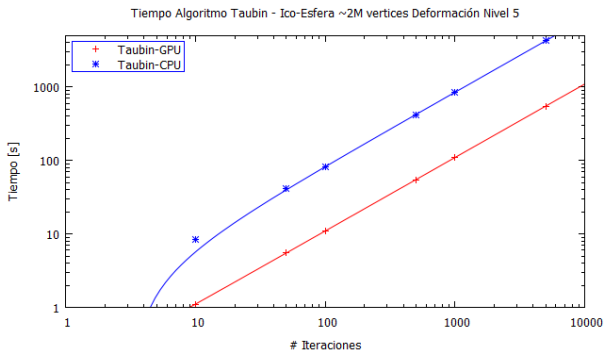


(b) Ico-Esfera Deformación Nivel 5

Figura 3.5: Gráfico numero de iteraciones para Ico-Esfera de $\approx 2K$ Vértices. Se aprecia que los tiempos tienen una tendencia lineal, en función del numero de iteraciones.



(a) Ico-Esfera Deformación Nivel 1

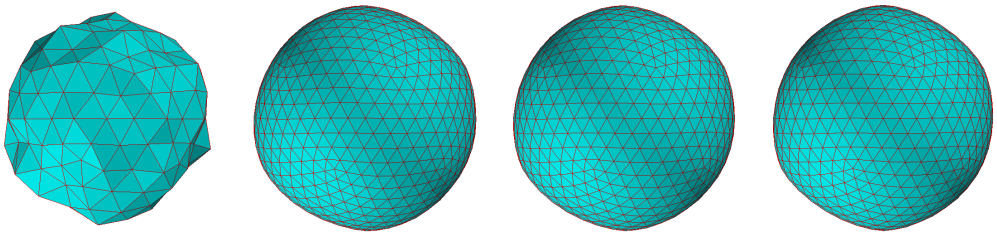


(b) Ico-Esfera Deformación Nivel 5

Figura 3.6: Gráfico numero de iteraciones para Ico-Esfera de $\approx 2M$ Vértices. La tendencia se mantiene al aumentar el numero de vértices.

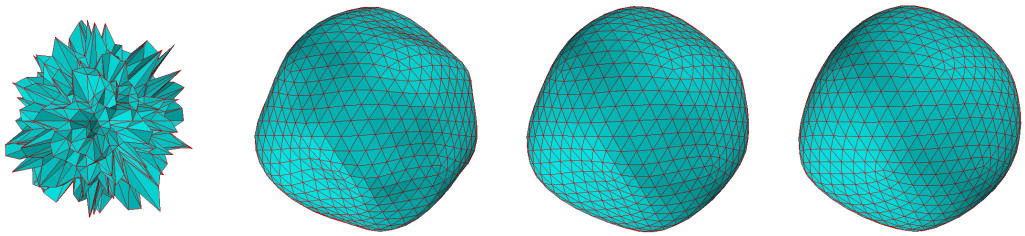
3.4.2. Resultados Cualitativos

En este apartado, se muestra el resultado final de las mallas luego de aplicar el Algoritmo. Dado que el espíritu del Algoritmo Paralelo y Secuencial es el mismo, y no se aprecian visualmente diferencias entre uno y otro, no se realiza diferenciación entre los resultados producidos por ambos algoritmos.



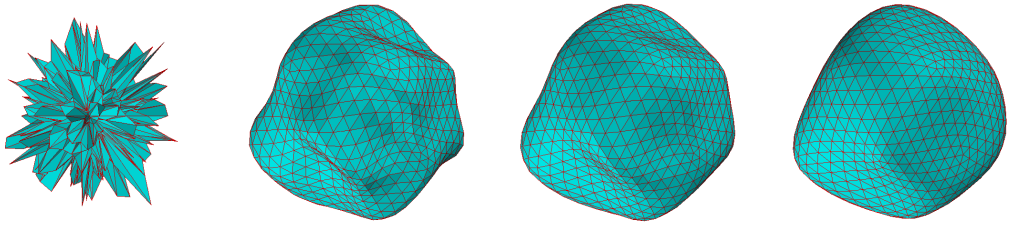
(a) Esfera con Deformación Nivel 1. (b) Esfera luego de 50 iteraciones. (c) Esfera luego de 100 iteraciones. (d) Esfera luego de 500 iteraciones.

Figura 3.7: Resultados Ico-Esfera de ≈ 500 vértices con deformación nivel 1. Se observa que la esfera recupera su forma original al paso de unas pocas iteraciones.



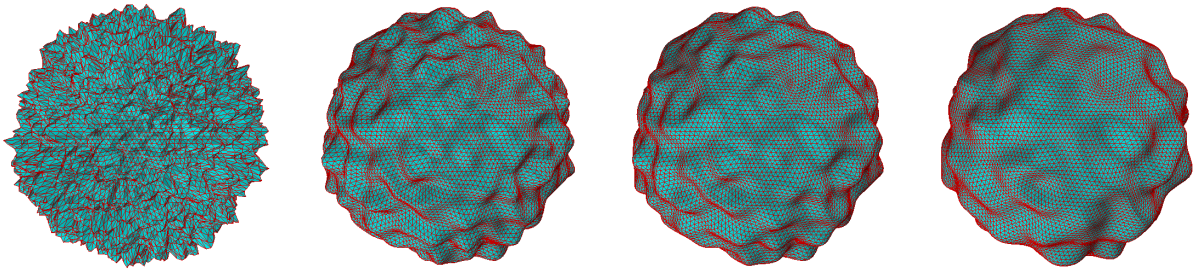
(a) Esfera con Deformación Nivel 5. (b) Esfera luego de 50 iteraciones. (c) Esfera luego de 100 iteraciones. (d) Esfera luego de 500 iteraciones.

Figura 3.8: Resultados Ico-Esfera de ≈ 500 vértices con deformación nivel 5. Se observa que la esfera converge a una forma similar a la esfera original.



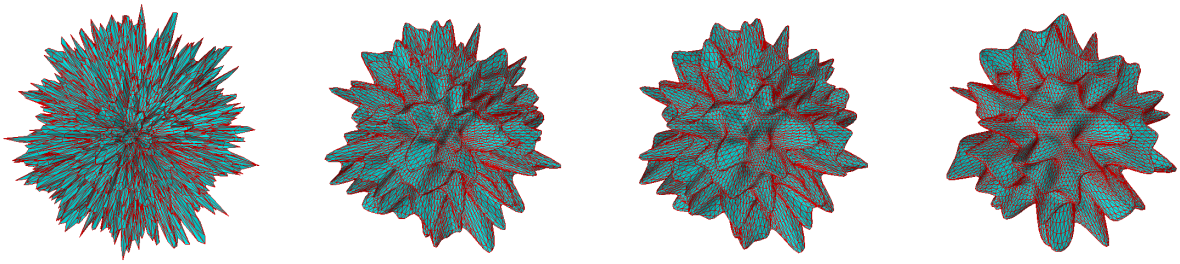
(a) Esfera con Deformación Nivel 10. (b) Esfera luego de 50 iteraciones. (c) Esfera luego de 100 iteraciones. (d) Esfera luego de 500 iteraciones.

Figura 3.9: Resultados Ico-Esfera de ≈ 500 vértices con deformación nivel 10. Se observa que luego de varias iteraciones, el resultado no converge a la esfera original.



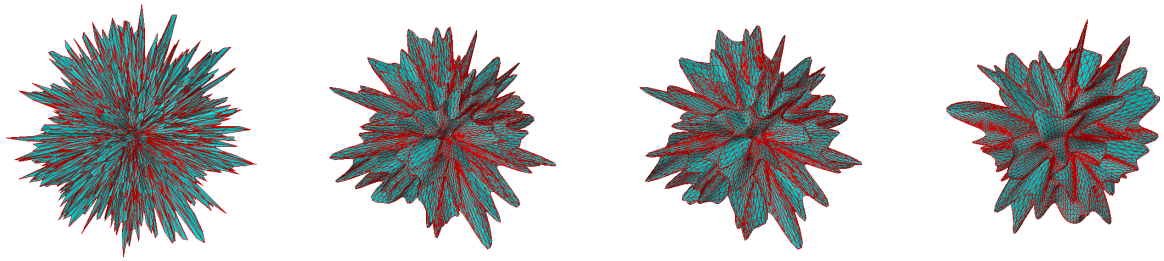
(a) Esfera con Deformación Nivel 1. (b) Esfera luego de 50 iteraciones. (c) Esfera luego de 100 iteraciones. (d) Esfera luego de 500 iteraciones.

Figura 3.10: Resultados Ico-Esfera de $\approx$ 10000 vértices con deformación nivel 1. La esfera tiende a una figura parecida a la original.



(a) Esfera con Deformación Nivel 5. (b) Esfera luego de 50 iteraciones. (c) Esfera luego de 100 iteraciones. (d) Esfera luego de 500 iteraciones.

Figura 3.11: Resultados Ico-Esfera de $\approx$ 10000 vértices con deformación nivel 5. La esfera converge a una figura tipo estrella, distinta de la esfera original.



(a) Esfera con Deformación Nivel 10. (b) Esfera luego de 50 iteraciones. (c) Esfera luego de 100 iteraciones. (d) Esfera luego de 500 iteraciones.

Figura 3.12: Resultados Ico-Esfera de $\approx$ 10000 vértices con deformación nivel 10. Se observa que la figura no converge a la esfera original.

3.5. Conclusiones

Luego de analizar los resultados cuantitativos, podemos obtener las siguientes conclusiones:

A partir de las Tablas 3.4 y 3.5 para el caso de procesamiento en CPU, y de las Tablas 3.2 y 3.3 para el caso de la GPU, se puede concluir que, independientemente del nivel de deformación presente en la malla, el tiempo que demora cada uno de los algoritmos, es similar en cada plataforma. Por otro lado, de estas mismas tablas, es posible deducir que aumentar en un orden de magnitud el numero de iteraciones (de 500 a 5000), el tiempo que demora el algoritmo en cada caso particular también aumenta en 1 orden de magnitud aproximadamente. Estos puntos se reafirman el los gráficos de la Figuras 3.3 y 3.4, en los cuales se aprecia que, independientemente del nivel de deformación, las curvas de tiempo generadas por los algoritmos son la misma. También, dado que los gráficos están en escala logarítmica en ambos ejes, se puede apreciar que para un mismo punto, el algoritmo de Taubin en CPU supera por 1 orden de magnitud en tiempo de ejecución al algoritmo en GPU, a partir de un cierto numero de vértices(≈ 400). También de la tabla 3.10 se puede apreciar que el algoritmo paralelo tiende a ser entre 2.0 y 7.5 veces mas rápido que su contraparte secuencial.

De los mismos gráficos podemos deducir que para mallas con un numero pequeño de vértices, el algoritmo en CPU es mas eficiente que su contraparte en GPU. Esto se debe a que, como se explicó anteriormente, existe un sobrecosto asociado al inicializar y utilizar la GPU, lo cual hace que esta alternativa no sea del todo viable, cuando la cantidad de elementos a procesar en paralelo es muy pequeña, ya que se pierde el enorme poder de paralelización que brinda la GPU.

Adicionalmente, los gráficos de las Figuras 3.5 y 3.6 reafirman que el tiempo que demoran los algoritmos es directamente proporcional al numero de iteraciones que se ejecuta el algoritmo, en cualquiera de las dos plataformas.

Por otro lado, a partir de los resultados cualitativos, se pueden obtener las siguientes conclusiones:

Tenemos que para las Figuras 3.7 y 3.8, que representan una esfera de ≈ 500 vértices, con 1 y 3 niveles de deformación respectivamente, se aprecia que ya con pocas iteraciones la esfera deformada empieza a recuperar su forma original, y esta va mejorando al aumentar el numero de iteraciones. Por otro lado, para la Figura 3.9 se aprecia que la esfera con 10 niveles de deformación mejora bastante su suavidad a medida que aumenta el numero de iteraciones, pero a priori se aprecia que no es posible reconstruir la forma de la esfera original a partir de este nivel de deformación.

Esta ultima teoría se reafirma con las Figuras 3.10, 3.11 y 3.12, en la cual se puede apreciar que solo en el primer caso, la esfera deformada empieza a retomar parte de su forma original. Para el resto de las esferas, se tiene que la suavidad de la esfera deformada mejora bastante, pero la figura a la cual empieza a converger dista bastante de la esfera original.

Para explicar estos últimos resultados, se tienen dos posibles explicaciones. La primera es,

que el nivel de deformación en la malla para un número de puntos dado, es tan alta, que el algoritmo no es capaz de reconstruir la esfera original, a tal punto que la malla deformada no se parece a una esfera. También tenemos que recordar que este método es una analogía al procesamiento de señales, campo en el cual también es complicado reconstruir una señal cuando el ruido presente en ella es muy grande. La segunda explicación es el valor utilizado en los factores de expansión y contracción, λ y μ, ya que en la implementación se decidió utilizar los valores por defecto propuestos por el autor. Puede ser que al usar un par de valores de λ y μ distintos, se puedan obtener otros resultados que presenten una mejor solución para estos casos particulares.

Capítulo 4

Algoritmo de Triangulación de Delaunay

En este capítulo, se abordará el algoritmo de triangulación de Delaunay en paralelo. Primero se explorará la teoría que existe detrás de este tipo de triangulación. Luego se estudiarán implementaciones paralelas, tanto en CPU, como en GPU, y se propondrá una nueva implementación paralela del algoritmo en GPU, que utiliza un criterio ligeramente distinto. Finalmente se realizaran pruebas cualitativas y cuantitativas entre estas tres implementaciones y se presentarán conclusiones a partir de estas pruebas.

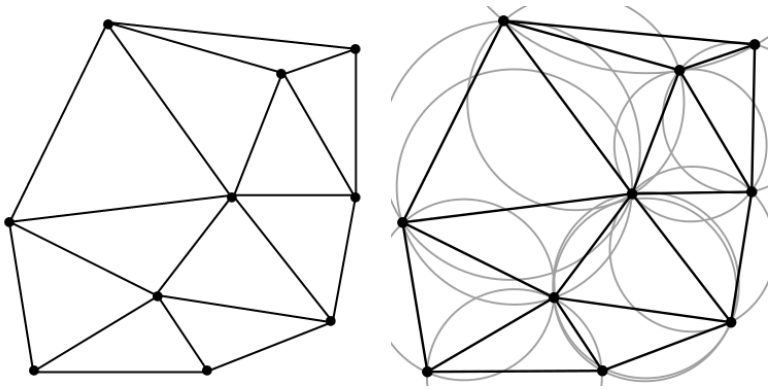
4.1. Trasfondo Teórico

La triangulación de Delaunay de un conjunto de N vértices, es una triangulación que cumple la condición de que ningún vértice de este conjunto N está en el interior del circuncírculo de cualquier triangulo de esta triangulación. Esto se puede apreciar en la Figura 4.1. Además, estas triangulaciones maximizan el ángulo mínimo de los triángulos que la componen, lo que evita que existan triángulos tales que sus vértices sean casi colineales en la malla. Esta propiedad es útil cuando se tiene aplicaciones de modelamiento en tiempo real, o modelamiento de superficies con mucho detalle en algunas de sus regiones, ya que permite reducir eventuales distorsiones producidas por otros algoritmos que usen como entrada, estas triangulaciones.

Existen diversos algoritmos para generar una triangulación de Delaunay a partir de un conjunto de N vértices, pero solo se abordarán dos de ellos, El algoritmo basado en *edge-flips*, y el algoritmo *incremental*, que son los que están implementados en la librería Cleap y el software de Kohout, respectivamente.

Antes de explicar en qué consiste cada uno, primero hay que describir que es un *edge-flip*. Dados dos triángulos que tienen un arco en común, un *edge-flip* consiste en intercambiar el arco que tienen en común. Esto se puede ver gráficamente en la Figura 4.2. La técnica del *edge-flip* es importante, pues permite que dos triángulos que no satisfacen la condición de Delaunay, la satisfagan al intercambiar el arco que comparten.

Para determinar cuándo realizar un *edge-flip*, existen dos criterios. El primero es el criterio



(a) Triangulación de un conjunto de vertices

(b) Triangulación, con los circuncírculos asociados a cada triangulo.

Figura 4.1: Con los circuncírculos se puede evaluar si una triangulación satisface o no la condición de Delaunay

de los *ángulos opuestos*, y el segundo es el criterio del *determinante*. Cabe señalar que ambos criterios son equivalentes, y sirven para determinar si un vértice está dentro o no del circuncírculo de un triángulo de la triangulación, pero la forma y eficiencia en su computación es variable.

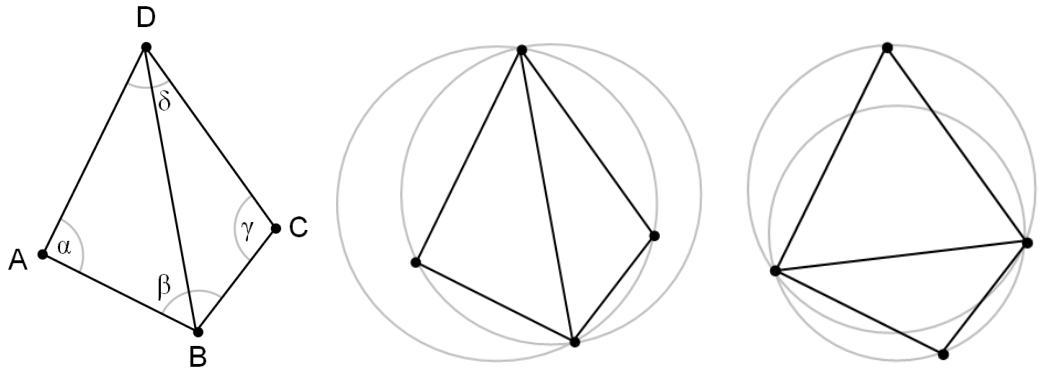
Para explicar el criterio de los ángulos opuestos, tomemos como referencia la Figura 4.2a. Este criterio indica que dos triángulos unidos por un arco, no cumplen la condición de Delaunay, si la suma de los ángulos opuestos a través de este arco (en este ejemplo, α y γ), es mayor a 180°. Por lo tanto, en este caso es necesario computar el valor de los ángulos α y γ, mediante operaciones geométricas y vectoriales.

Por otro lado, el criterio del determinante indica que para determinar si un punto D, se encuentra dentro del circuncírculo formado por los puntos A, B, C, se puede calcular el siguiente determinante:

$$\begin{vmatrix} A_x & A_y & A_x^2 + A_y^2 & 1 \\ B_x & B_y & B_x^2 + B_y^2 & 1 \\ C_x & C_y & C_x^2 + C_y^2 & 1 \\ D_x & D_y & D_x^2 + D_y^2 & 1 \end{vmatrix} = \begin{vmatrix} A_x - D_x & A_y - D_y & (A_x - D_x)^2 + (A_y - D_y)^2 \\ B_x - D_x & B_y - D_y & (B_x - D_x)^2 + (B_y - D_y)^2 \\ C_x - D_x & C_y - D_y & (C_x - D_x)^2 + (C_y - D_y)^2 \end{vmatrix}$$

Luego, dependiendo de cómo estén ordenados los puntos A, B, C en la estructura de datos, el signo de este determinante determina si el punto D está fuera o dentro del circuncírculo. Para este cálculo, es necesario realizar operaciones de multiplicaciones, y si no se tiene el supuesto de un buen orden en los puntos en la estructura de datos, además es necesario calcular otro determinante más pequeño, que es equivalente a calcular el producto cruz de los puntos A, B, C, y así determinar su orientación relativa.

Ahora que definimos que es un *edge-flip*, podemos describir los algoritmos de triangulación. El algoritmo basado en *edge-flips*, primero construye una triangulación cualquiera de los



(a) Triangulación de los puntos ABCD.

(b) Con los circuncírculos es evidente que no se cumple la condición de Delaunay.

(c) Luego de aplicar un *edge-flip* del arco BD por el arco AC, esta triangulación satisface la condición de Delaunay.

Figura 4.2: Etapas de un *edge-flip*

puntos y luego realiza la operación de *edge-flips* hasta que no existan triángulos que no cumplan la condición de Delaunay.

Por otro lado, el algoritmo incremental consiste en, dada una triangulación inicial de tres vértices de la lista de vértices, se agrega un nuevo vértice de esta lista a esta triangulación, y luego utilizando *edge-flips*, se corrige para que sea una triangulación de Delaunay válida. Este proceso se repite hasta que no queden vértices por agregar a la triangulación.

4.2. Versión Paralela Cleap

El algoritmo implementado en Cleap[12], consiste de cuatro etapas paralelas consecutivas:

- Detección
- Exclusión
- Procesamiento
- Reparación

En la etapa de detección, se identifican que arcos compartidos por un par de triángulos, no satisfacen la condición de Delaunay, de acuerdo a la Figura 4.2. Para esto, cada *thread* computa y verifica si se cumple o no la condición de los ángulos opuestos. La verificación de esta condición se realiza a través de un *test* para cada arco en la malla. Este algoritmo, se implementa y ejecuta a nivel de la GPU, y se muestra en el Algoritmo 7

Algoritmo 7: Delaunay: *test* del Ángulo en GPU

Data: Vértices A,B,C,D de los triángulos que comparten el arco
Result: Condición: True o False
vect1,vect2 = obtener los demás arcos que forman el primer triangulo;
α = calcular ángulo entre vect1 y vect2;
vect3,vect4 = obtener los demas arcos que forman el segundo triangulo;
β = calcular ángulo entre vect3 y vect4;
retornar el valor de verdad de la condición: $\alpha + \beta \leq \pi$;

En caso de que este *test* entregue que la condición no se cumple, entonces nos encontramos en el caso en que para este arco es cuestión, es necesario realizar un *edge-flip* para que los triángulos que lo contienen satisfagan la condición de Delaunay.

En la etapa de Exclusión, se debe verificar que la serie de *edge-flip* a realizar en paralelo, no produzca inconsistencias en la malla. Esto, debido a que la operación de *edge-flip* en paralelo, puede producir errores e inconsistencias en la malla, si se procesan al mismo tiempo arcos que pertenecen a una misma vecindad. Notar que este problema no ocurre en las implementaciones secuenciales del algoritmo, ya que en esos casos solo se procesa de un arco por iteración.

Debido a esto, es necesario generar una condición de exclusión entre arcos, de tal manera de procesar un sub-conjunto de arcos, tal que se encuentren en vecindades distintas, y así, sea posible realizar una operación de *edge-flip* en paralelo sin comprometer la integridad de la malla. Para esto, cada arco necesita tener control exclusivo sobre los triángulos que lo utilizan. Para implementar esto, se cuenta con un arreglo de *Flags*, de tamaño igual al numero de triángulos de la malla, el cual indica mediante un valor binario, si el triangulo representado por el *i-esimo* índice de este arreglo, ha sido tomado por algún arco o no. De esta manera, cada *thread* en la GPU debe verificar que ambos triángulos que comparten el arco en cuestión están disponibles o no, y solo proseguir con el algoritmo si tiene control exclusivo sobre ambos

triángulos. Para evitar *Data-races* y *Dead-locks*[1], la operación de verificar la disponibilidad de ambos triángulos se realiza de manera *atómica*, es decir, cada thread verifica en un solo paso, que ambos triángulos estén disponibles, lo que produce que esa porción de código se ejecute de manera secuencial en el caso que existan dos o mas *threads* que quieran acceder a un punto de memoria compartida al mismo tiempo. Estos pasos se describen en el Algoritmo 8

Algoritmo 8: Delaunay: *test* de Exclusión Mutua en GPU

```
Data: Arco e, Arreglo compartido Flags[]
Result: Condición: True o False
i,j = Índices triángulos que comparten arco e;
atomicLock(i,j);
if Flags[i] == 0 and Flags[j] == 0 then
    Flags[i] == 1;
    Flags[j] == 1;
    atomicUnlock(i,j);
    retornar True;
end
atomicUnlock(i,j);
retornar False;
```

Por lo tanto, este *test* indica sobre que subconjunto de arcos se puede realizar *edge-flip* en paralelo, sin comprometer la integridad ni consistencia de la malla.

En la etapa de procesamiento, se realiza el procedimiento de *edge-flip* para los *threads* que sobrevivieron a las etapas anteriores. Para esto, se realiza un intercambio de índices de arcos en el arreglo de triángulos, entre los triángulos involucrados en la operación de *edge-flip* en la estructura de datos de Cleap (Figura 2.9). Esta operación se puede ver gráficamente en la Figura 4.3, y se explica en el Algoritmo 9

Algoritmo 9: Delaunay: procedimiento de *edge-flip* en GPU

```
Data: Arreglos: Opuestos[], Triangulos[], Arcos[]
Result: Triangulos[], Arcos[]
o1 = Opuestos[i][0];
o2 = Opuestos[i][2];
c1 = Arcos[i].t_a1;
c2 = Arcos[i].t_b2;
Triangulos[c2] = Triangulos[o1];
Triangulos[c1] = Triangulos[o2];
Arcos[i].t_a=[o1,c1];
Arcos[i].t_b=[c2,o1];
Arcos[i].v1 = Arcos[i].t_a1;
Arcos[i].v2 = Arcos[i].t_b2;
```

[1]Cuando dos o mas procesos o hilos de ejecución, se estan esperando mutuamente para terminar su ejecución, se dice que estan en una condición de *Dead-lock*

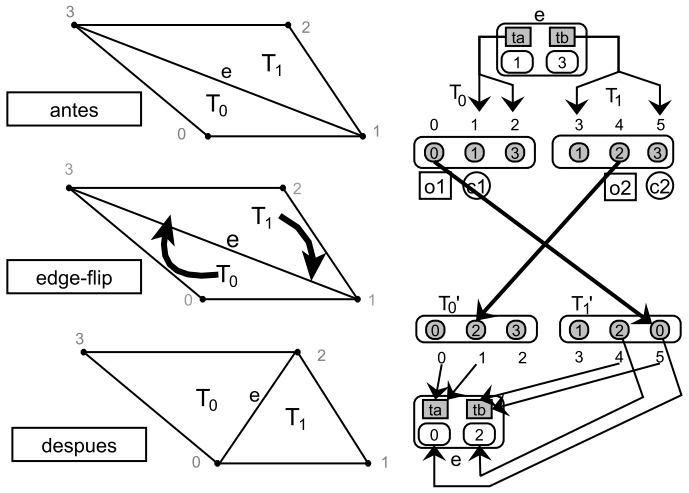


Figura 4.3: Ejemplo del procedimiento de *edge-flip* en Cleap.

Para realizar este intercambio de índices, es necesario utilizar la información redundante que maneja la estructura de datos de Cleap, la cual permite identificar rápidamente los vértices que pertenecen al arco sobre el que se realizará *edge-flip*, y los vértices que corresponderán al arco luego de realizar el *edge-flip*. Así es posible realizar rápidamente el intercambio de vértices en los triángulos, y luego actualizar al arco con estas nuevas coordenadas.

Finalmente, en la etapa de reparación, se deben actualizar las referencias de los arcos que no participaron de este procedimiento paralelo, los cuales ahora pueden almacenar información desactualizada de los triángulos que lo contienen. Un ejemplo de esto, se puede apreciar en la Figura 4.4. Para detectar estos casos, se realiza una verificación utilizando las coordenadas de los vértices de los triángulos:

$$q = |v_1 - t_{a1}| + |v_2 - t_{a2}|$$
$$w = |v_1 - t_{b1}| + |v_2 - t_{b2}|$$

En la cual, si alguna de las dos expresiones es mayor a 0, entonces el arco presenta una referencia desactualizada, que debe ser reparada. Esto quiere decir que la referencia que tiene el arco del triángulo que supuestamente lo contiene, no es consistente con la información de los vértices que componen a este arco. Para realizar esta reparación, se cuenta con un arreglo de Rotaciones, de largo igual al numero de triángulos, que indica con que triángulo se realizó la rotación, en función de la fase de procesamiento anterior. Este procedimiento se describe

en el Algoritmo 10.

Algoritmo 10: Delaunay: fase de Reparación en GPU

Data: Arreglos: Triangulos[], Rotaciones[]
Result: Triangulos[], Arcos[]
T_a = Índice al triángulo t_a del arco actual;
T_b = Índice al triángulo t_b del arco actual;
V_1,V_2 = Vértices que forman el arco actual;
if V_1 *!=* T_{a1} *o* V_2 *!=* T_{a2} **then**
  R_a = Rotaciones[T_a];
  T_a = R_a;
end
if V_1 *!=* T_{b1} *o* V_2 *!=* T_{b2} **then**
  R_b = Rotaciones[T_b];
  T_a = R_b;
end
Arcos[i].T = [T_a,T_b];

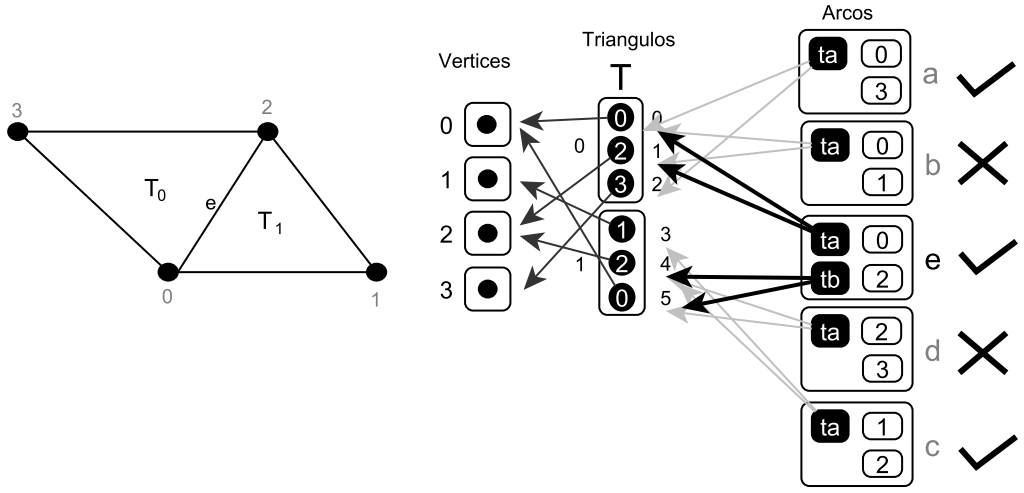


Figura 4.4: Etapa de reparación en Cleap. Los arcos marcados con una cruz, deben actualizar sus referencias a triángulos.

Finalmente, con ayuda de estas cuatro fases, se compone el algoritmo de Delaunay en Paralelo, y su funcionamiento se puede describir gráficamente en la Figura 4.5.

Con esta implementación, se debe tener en cuenta el siguiente detalle. La verificación de la condición de los ángulos opuestos, considera un pequeño factor de tolerancia ε, para manejar errores de calculo de punto flotante. Esto produce que se ignore una cierta cantidad de *edge-flips* que si se deberían haber realizado, por lo que este algoritmo produce mallas consideradas *quasi-Delaunay*. De todos modos, estas mallas producidas presentan un porcentaje de error cercano al 0,05 %, en comparación con una malla completamente Delaunay, por lo que esta diferencia en la practica es imperceptible.

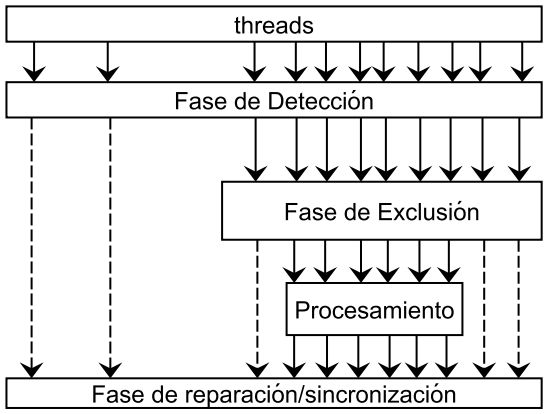


Figura 4.5: Vista general de las 4 Fases del algoritmo de Delaunay en Cleap.

4.3. Versión Paralela Kohout

En el articulo de Kohout[8], y el software que sé genero a partir de él, se proponen diversos métodos y combinaciones de paralelización del algoritmo de Delaunay en CPU, en función de la investigación realizada en su paper.

Para esto, primero describe dos variantes del algoritmo incremental de Delaunay, la Construcción incremental, y la Inserción incremental. La primera describe la manera clásica del algoritmo incremental, que consiste en los siguientes pasos:

1. Se inicia triangulando tres vértices cualquiera (ojala consecutivos), de tal manera que no exista un cuarto vértice en su interior.
2. Se agrega un nuevo punto a la triangulación, tal que este nuevo vértice no se encuentre dentro del circuncírculo de alguno de los triángulos de la triangulación actual.
3. Se triangula este punto, y se cuenta con una triangulación parcial que cumple la condición de Delaunay.
4. Se repiten los pasos 2 y 3, hasta agregar todos los vértices de la malla

Por otro lado, la segunda variante consiste en lo siguientes pasos:

1. Del conjunto de vértices, se obtienen los vértices que conforman su *Convex-hull*[2], y a partir de estos vértices, se forma una triangulación inicial.
2. Para cada punto que aun no pertenezca a la triangulación, se realiza su inserción en la triangulación, para lo cual es necesario subdividir el triangulo donde se inserta este vértice.
3. Luego, es necesario revisar recursivamente en la triangulación el criterio del circuncírculo, y en caso que no se cumpla para un par de triángulos, se debe realizar un *edge-flip* para cumplir la condición de Delaunay.

[2]La Cerradura Convexa de un conjunto de vértices, consiste en el conjunto mas pequeño que contiene a todos los vértices en su interior o contorno.

4. Se repiten los pasos 2 y 3, hasta agregar todos los vértices de la malla.

Ambas variantes del algoritmo incremental se pueden apreciar gráficamente en las Figuras 4.6 y 4.7 respectivamente. Cada una tiene sus ventajas y contratiempos, pero Kohout, luego de realizar una exhaustiva comparación, se inclina por la variante de la inserción incremental, principalmente, porque a diferencia de la otra variante, esta ofrece robustez en el caso de que ocurran errores de cálculo de punto flotante en el cálculo asociado al criterio del circuncírculo, ya que no realizará operaciones de *edge-flip* que comprometan la integridad de la triangulación.

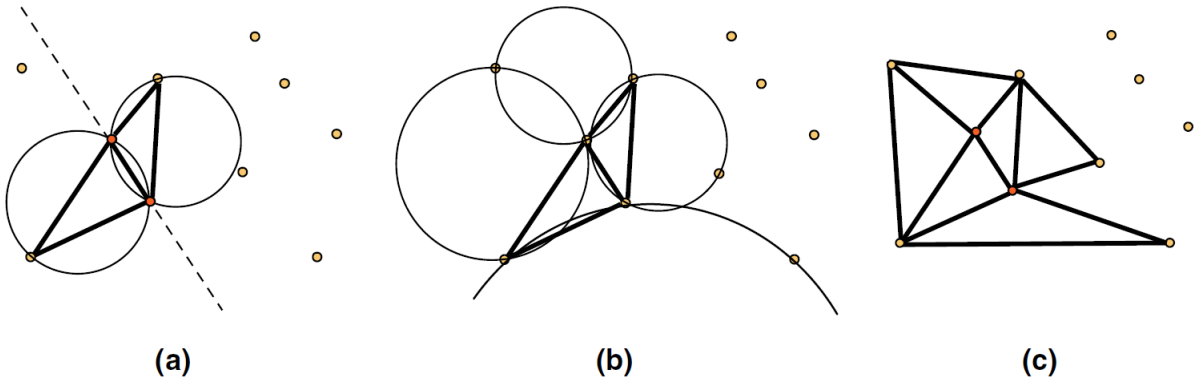


Figura 4.6: (a) Triangulación inicial. (b) Se utiliza el circuncírculo para ver que nuevos vértices agregar. (c) Triangulación parcial luego de agregar nuevos vértices.

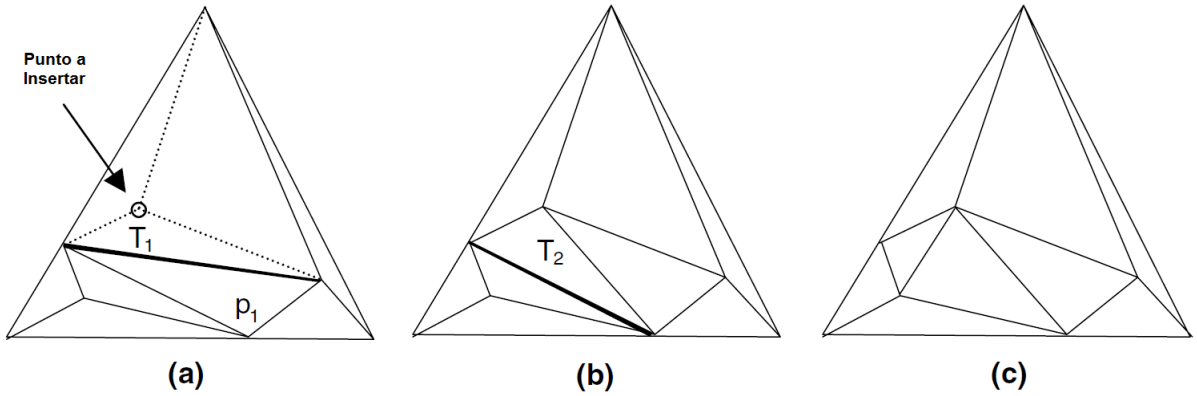


Figura 4.7: (a) Se inserta un nuevo vértice a la triangulación, subdividiendo la malla. (b) se revisa el criterio del circuncírculo, y se realiza el procedimiento de *edge-flip* en caso de ser necesario. (c) Triangulación parcial luego del procedimiento.

Con esto, la paralelización del algoritmo de Delaunay propuesta por Kohout consiste en paralelizar el Algoritmo incremental por inserción. Para esto, se identifican tres Fases:

1. Fase de Localización, en la cual se identifica el triangulo afectado por la inserción del nuevo vértice.
2. Fase de Subdivisión, en la cual se realiza la inserción del nuevo vértice y se subdivide el triangulo afectado.
3. Fase de Legalización, en la cual se aplica el criterio del circuncírculo para verificar la condición de Delaunay, y realizar *edge-flips* de ser necesario.

La principal estructura de datos utilizada por Kohout, llamada DAG[3], consiste a grandes rasgos en un árbol, que almacena en sus nodos los triángulos de la triangulación. Luego, cada vez que un triangulo se subdivide, se le agregan 2 o 3 nodos hijos a este nodo (dependiendo del caso de subdivisión), y los *edge-flips* se modelan como intercambio de punteros entre los nodos de este árbol. Finalmente, los nodos hoja de este árbol, corresponderán a los triángulos finales de la triangulación. El uso y forma de esta estructura de datos se puede apreciar en la Figura 4.8, y mas detalles de esta estructura, se pueden ver en el capitulo 2 del *paper* de Kohout[8].

Con esto, tenemos que cada una de las tres Fases del algoritmo, requiere acceso de manera concurrente a la estructura DAG, ya sea de lectura o escritura. Por la naturaleza del algoritmo, solo es necesario manejar condiciones de exclusión mutua solo cuando se realizan operaciones de escritura de manera simultánea en la estructura DAG. Estas operaciones de escritura solo son necesarias para las fases de Subdivisión y Legalización, que es donde se modifica la triangulación. Para esto, el autor propone tres posibilidades de exclusión mutua:

1. Lote o *Batch*: Todos los *threads* disponibles (salvo uno) realizan la fase de localización, pero solo uno de ellos (el que no realizo localización) realiza las fases de subdivisión y legalización.
2. Pesimista: Todos los *threads* realizan las tres fases, pero solo un *thread* a la vez puede estar realizando la fase de subdivisión y legalización.
3. Optimista: Todos los *threads* realizan las tres fases, pero cuando un *thread* necesita modificar la estructura DAG, necesita tener acceso exclusivo al nodo a modificar, junto a todos sus hijos.

De estas tres posibilidades, y de acuerdo a las pruebas realizadas y lo expresado por el autor, la Optimista es la menos restrictiva, y la que presenta mejores resultados en la mayoría de los casos.

Finalmente, para el manejo de acceso concurrente, y así evitar *dead-locks* en los *threads*, se tienen tres estrategias:

1. Detección: Cada *thread* mide el tiempo que lleva esperando para entrar a la sección crítica. Si supera un cierto umbral (experimental) de espera, entonces no inserta el vértice, y busca otro para insertar.
2. Prioridad: Cada *thread* maneja un campo de prioridad. Cada vez que un *thread* quiere

[3]Directed Acyclic Graph, Grafo Dirigido Acíclico

ingresar a una sección critica, revisa si la prioridad del *thread* que esta adentro es menor que su prioridad. Si es menor, se queda esperando, en caso contrario, busca insertar otro vértice, y aumenta su prioridad en 1.

3. Bandido: Cuando un *thread* necesita ingresar a una sección critica, se otorga acceso inmediato, y bloquea a los demás *threads* que puedan estar trabajando en ella. Cuando el *thread* termina de trabajar, desbloquea a los demás *threads*. En este caso, los *threads* desbloqueados además tienen que verificar si pueden seguir trabajando sin que se produzcan inconsistencias. En caso que no puedan, simplemente liberan la sección critica e intentan insertar otro vértice.

De estos tres métodos, y de acuerdo a las pruebas realizadas y lo expresado por el autor, el de Detección es la que presenta mejores resultados teóricos y experimentales. Con esto, la combinación a utilizar para comparar con la implementación de la librería Cleap, sera usando la condición de exclusión mutua Optimista, con el método de Detección.

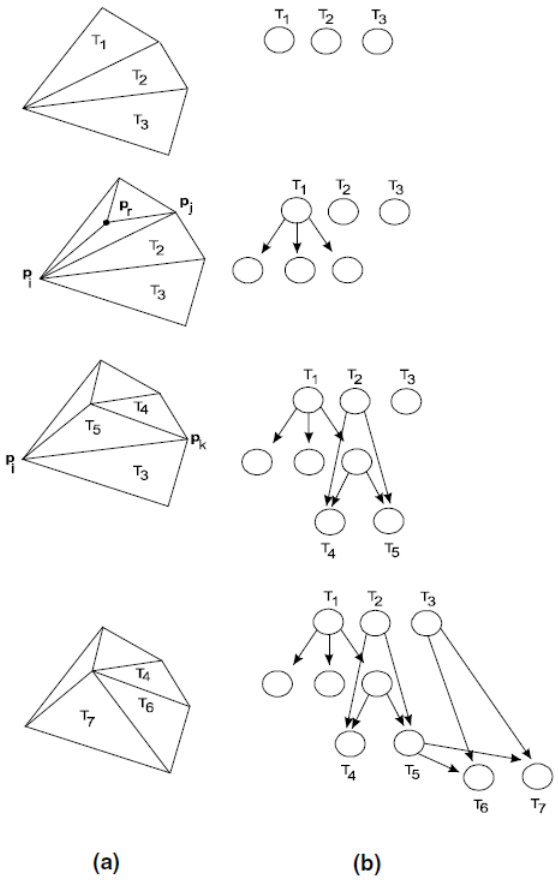


Figura 4.8: (a) Triangulación bajo diversas operaciones. (b) Estado de la estructura DAG luego de cada operación en la triangulación.

4.4. Nueva Versión Paralela para Cleap

Debido a lo explicado al final del capitulo 4.2, se desea implementar el criterio del Determinante en este algoritmo, de manera de evitar la introducción del factor de tolerancia ε, y así ver si es posible obtener resultados mas precisos, o bien, obtener los mismos resultados que produce el algoritmo actual, con mayor rapidez.

Tenemos que buena parte de la estructura del algoritmo original se puede conservar para la implementación del criterio del Determinante, ya que para integrar este criterio, solo se deben realizar cambios a nivel de la Fase de Detección, reemplazando el criterio de los ángulos opuestos, por este nuevo criterio, y así se puede integrar sin mayores problemas con el resto de las fases.

Para realizar este *test* sobre el conjunto de los cuatro vértices que componen a los triángulos compartidos por el arco a revisar, es necesario realizar dos cálculos. El primero, es calcular el siguiente determinante:

$$M = \begin{vmatrix} A_x - D_x & A_y - D_y & (A_x - D_x)^2 + (A_y - D_y)^2 \\ B_x - D_x & B_y - D_y & (B_x - D_x)^2 + (B_y - D_y)^2 \\ C_x - D_x & C_y - D_y & (C_x - D_x)^2 + (C_y - D_y)^2 \end{vmatrix}$$

Para el cual, los vértices A, B, C forman un triangulo, y se desea verificar si el punto D esta en el interior o no del circuncirculo formado por los vértices A, B, C. En teoría, este cálculo del determinante asume que los vértices A, B, C están ordenados en el sentido opuesto a las agujas del reloj (*counter-clockwise*). Dado que no se puede asumir este supuesto en el orden de los puntos dispuesto por la estructura de datos de Cleap, es necesario realizar un cálculo adicional, para determinar la orientación relativa de los vértices A, B, C. Esto se puede realizar a través de un producto cruz entre los arcos que forman este triángulo, lo cual también se puede realizar calculando un determinante más pequeño:

$$O = \begin{vmatrix} A_x - c_x & A_y - C_y \\ B_x - C_x & B_y - C_y \end{vmatrix}$$

Con esto, Finalmente la condición de si es necesario realizar un *edge-flip* o no sobre este arco, viene dada por comprobar el signo de la expresión:

$$M \times O > 0$$

Si este signo es mayor a 0, entonces el punto D se encuentra dentro de la circunferencia circunscrita a los vértices A, B, C. Con esto en mente, podemos pasar a revisar el Algoritmo

11, que implementa este criterio.

Algoritmo 11: Delaunay: *test* del Determinante en GPU

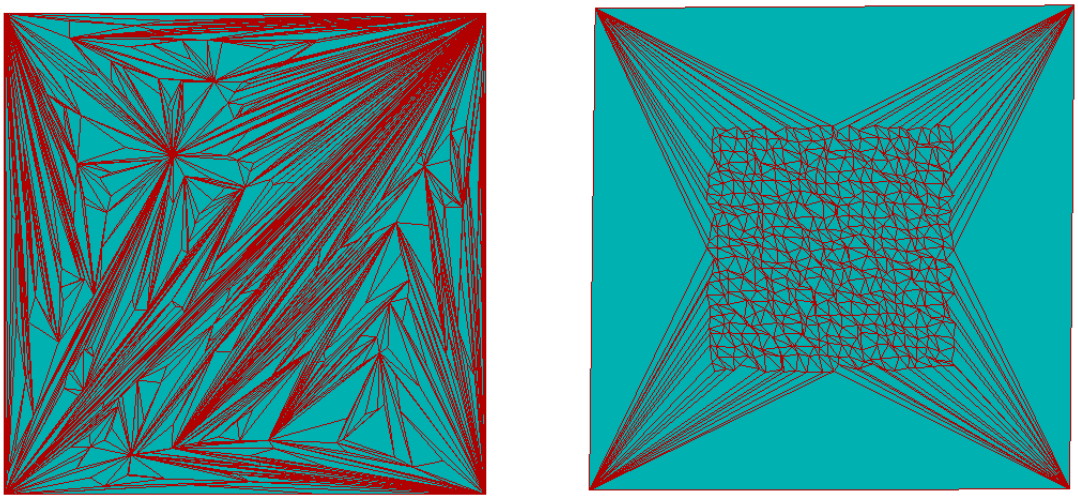
Data: Vértices A,B,C,D de los triángulos que comparten el arco
Result: Condición: True o False
A_{11} = A.x - D.x;
A_{12} = A.y - D.y;
A_{13} = (A.x* - D.x)*(A.x - D.x) + (A.y - D.y)*(A.y - D.y);
A_{21} = B.x - D.x;
A_{22} = B.y - D.y;
A_{23} = (B.x*B.x - D.x*D.x) + (B.y*B.y - D.y*D.y);
A_{31} = C.x - D.x;
A_{32} = C.y - D.y;
A_{33} = (C.x - D.x)*(C.x - D.x) + (C.y - D.y)*(C.y - D.y);
det = A_{11}*(A_{22}*A_{33} - A_{23}*A_{32}) - A_{12}*(A_{21}*A_{33} - A_{23}*A_{31}) + A_{13}*(A_{21}*A_{32} - A_{22}*A_{31});
B_{11} = A.x - C.x;
B_{12} = A.y - C.y;
B_{21} = B.x - C.x;
B_{22} = B.y - C.y;
order = B_{11}*B_{22} - B_{12}*B_{21};
retornar el valor de verdad de la condición: det*order>0;

Con esta implementación, se espera obtener mejores resultados que utilizando el criterio del ángulo, ya que en teoria y hasta el momento, no es necesario considerar un factor de tolerancia ε. Además la cantidad de operaciones matemáticas realizadas es ligeramente menor que su contraparte, por lo que se espera que sea ligeramente mas rápido. En el siguiente cápitulo se realizaran pruebas y comparaciones con las demás implementaciones revisadas en los capítulos anteriores.

4.5. Resultados

Para probar el funcionamiento de los algoritmos, se realizó una batería de pruebas, que se describe a continuación:

- Se generaron 30 mallas en 2D con un numero de vértices variable entre 100.000 y 1.500.000 . 15 de ellas corresponden a mallas denominadas tipo *random*, y las otras 15, corresponden a mallas tipo *noise*. Las características particulares de cada tipo de malla se pueden ver en la Figura 4.9.
- Para cada una de estas mallas, se midió el tiempo, que demoran en realizar la triangulación (solo tiempo efectivo de ejecución del algoritmo, sin considerar otros factores, como tiempo de carga de la malla en memoria).
- Para el caso de los algoritmos en Cleap, además se midió la cantidad de iteraciones en paralelo realizadas.
- Cada uno de las mallas resultantes producidas por estos algoritmos, se comparó con el resultado producido por la librería CGAL[4], que se considera una de las mejores librerías de algoritmos secuenciales y que produce los resultados más exactos.



(a) Malla tipo *Random*.

(b) Malla tipo *Noise*.

Figura 4.9: Ejemplos de mallas sobre las que se realizaron las pruebas.

Por otro lado, las características del Equipo donde se realizaron estas pruebas, son las siguientes:

- Procesador Intel Core® i7™-4700MQ @ 2.40GHz.
- 12GB de RAM DDR3 @ 1600MHz.
- Chip Gráfico NVIDIA Geforce 740M con 2048MB de memoria.
- Sistema Operativo Ubuntu 13.10 x64 con Ubuntu Linux kernel versión 3.11.0-12.19, para las implementaciones de Cleap.

[4]Computational Geometry Algorithms Library

- Sistema Operativo Windows 8.1 x64 kernel versión 6.3 build 9600, para la implementación de Kohout.

A continuación se presentan los resultados de las dos implementaciones del algoritmo. Estos resultados se separan en dos: Resultados Cuantitativos y Resultados Cualitativos.

4.5.1. Resultados Cuantitativos

En este apartado, se muestran los resultados obtenidos por los tres algoritmos. Se midieron los tiempos que demoran las implementaciones, cantidad de iteraciones realizadas, y porcentaje de error en el resultado generado, bajo distintos escenarios.

	Tiempo de Ejecución [s]			
# Vértices (*10^6)	Kohout	Cleap-Ángulo	Cleap-Determinante	CGAL
0.1	0,03	0,24	0,22	0,11
0.2	0,06	0,52	0,47	0,23
0.3	0,09	0,79	0,76	0,34
0.4	0,12	1,19	1,04	0,47
0.5	0,15	1,38	1,35	0,58
0.6	0,18	1,68	2,14	0,70
0.7	0,21	2,15	1,96	0,82
0.8	0,24	2,41	2,20	0,93
0.9	0,27	2,87	2,57	1,05
1.0	0,31	3,10	2,77	1,18
1.1	0,35	3,63	3,55	1,27
1.2	0,37	3,80	3,65	1,40
1.3	0,40	4,79	3,98	1,53
1.4	0,44	4,57	5,13	1,64
1.5	0,46	4,75	5,50	1,78

Tabla 4.1: Medición de tiempo que demoran los Algoritmos de triangulación para mallas tipo *random*. Se observan que los tiempos de los algoritmos de Cleap son un orden superior a los demás.

	Tiempo de Ejecución [s]			
# Vértices (*10^6)	Kohout	Cleap-Ángulo	Cleap-Determinante	CGAL
0.1	0,04	0,05	0,10	0,11
0.2	0,09	0,09	0,18	0,23
0.3	0,14	0,16	0,22	0,34
0.4	0,18	0,19	0,38	0,47
0.5	0,24	0,21	0,40	0,58
0.6	0,29	0,36	0,51	0,70
0.7	0,34	0,32	0,61	0,82
0.8	0,39	0,41	0,79	0,93
0.9	0,45	0,53	0,79	1,05
1.0	0,50	0,51	1,00	1,18
1.1	0,54	0,54	0,99	1,27
1.2	0,61	0,56	1,15	1,40
1.3	0,66	0,56	1,61	1,53
1.4	0,73	0,67	1,83	1,64
1.5	0,74	0,69	1,81	1,78

Tabla 4.2: Medición de tiempo que demoran los Algoritmos de triangulación para mallas tipo *noise*. Se observan tiempos similares entre Kohout con Cleap-Ángulo, y Cleap-Determinante con CGAL.

	# de Iteraciones en Cleap			
	Mallas Random		Mallas Noise	
# Vértices	Ángulo	Determinante	Ángulo	Determinante
0.1	32	32	10	23
0.2	34	33	9	20
0.3	34	35	12	17
0.4	39	36	10	22
0.5	38	37	9	18
0.6	36	51	13	20
0.7	39	38	10	20
0.8	38	37	11	23
0.9	40	38	13	20
1.0	39	37	11	23
1.1	41	43	11	21
1.2	40	42	10	22
1.3	46	41	9	29
1.4	41	51	10	31
1.5	40	51	10	28

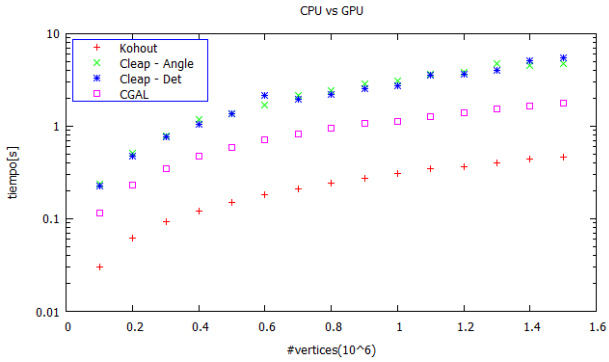
Tabla 4.3: Cantidad de iteraciones realizadas por los Algoritmos de Cleap para distintos escenarios. Para mallas *random* es similar, mientras que para mallas *noise*, Cleap-Ángulo toma casi la mitad de iteraciones que Cleap-Determinante.

	% error resultados mallas Noise con respecto a CGAL		
# Vertices (*10^6)	Cleap-Angulo	Cleap-Determinante	Kohout
0.1	0,0010	1,5011	0,0000
0.2	0,0000	1,3557	0,0000
0.3	0,0010	1,3070	0,0000
0.4	0,0026	1,2955	0,0000
0.5	0,0013	1,3169	0,0000
0.6	0,0013	1,2631	0,0000
0.7	0,0017	1,2970	0,0000
0.8	0,0021	1,2755	0,0000
0.9	0,0027	1,2689	0,0000
1.0	0,0013	1,2588	0,0000
1.1	0,0023	1,2511	0,0000
1.2	0,0017	1,2373	0,0000
1.3	0,0015	1,2746	0,0000
1.4	0,0012	1,2740	0,0000
1.5	0,0015	1,2827	0,0000

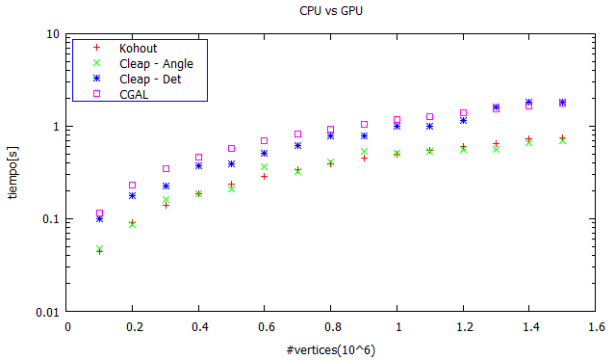
Tabla 4.4: Porcentaje de error en los resultados de mallas tipo *random*, en comparación a CGAL. El porcentaje de error de Cleap-Determinante es superior a los demás.

	% error resultados mallas Random con respecto a CGAL		
# Vertices (*10^6)	Cleap-Angulo	Cleap-Determinante	Kohout
0.1	0,000	1,748	0,044
0.2	0,000	4,377	0,027
0.3	0,001	6,617	0,021
0.4	0,003	8,426	0,016
0.5	0,002	10,320	0,013
0.6	0,003	10,480	0,013
0.7	0,008	14,134	0,011
0.8	0,002	14,530	0,011
0.9	0,019	16,572	0,010
1.0	0,003	16,717	0,009
1.1	0,012	19,725	0,009
1.2	0,003	17,227	0,008
1.3	0,002	21,231	0,008
1.4	0,004	19,031	0,007
1.5	0,018	19,649	0,007

Tabla 4.5: Porcentaje de error en los resultados en mallas tipo *noise*, en comparación a CGAL. El porcentaje de error de Cleap-Determinante es superior a los demás.

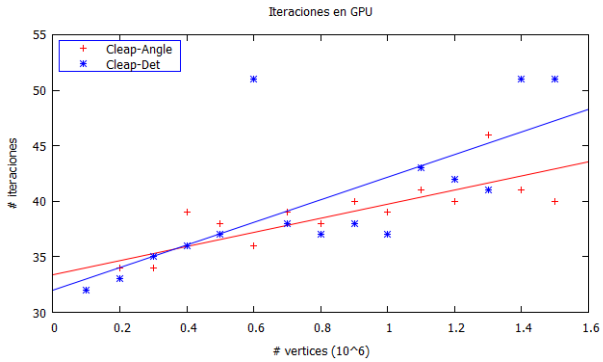


(a) Resultados para malla tipo *Random*.

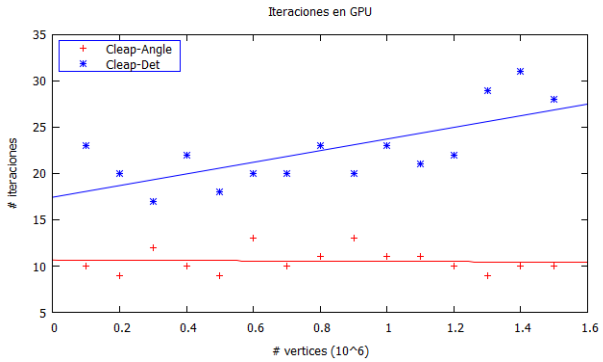


(b) Resultados para malla tipo *Noise*.

Figura 4.10: Gráficos de tiempos de Ejecución para los algoritmos. Se observa que para mallas *random*, los algoritmos de Cleap tienen tiempos similares, pero superiores a los demas.

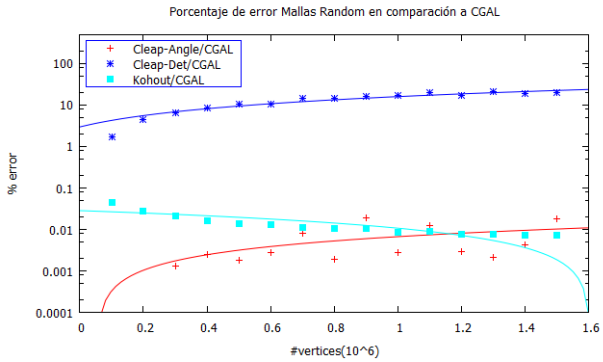


(a) Resultados para malla tipo *Random*.

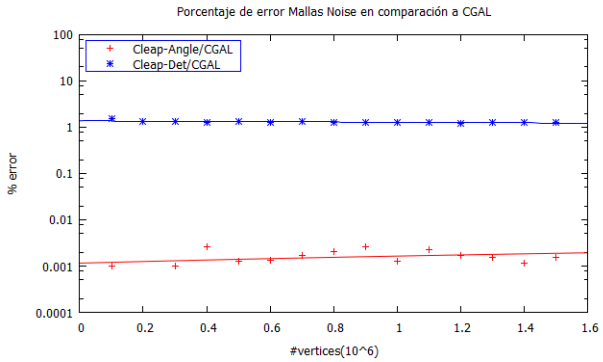


(b) Resultados para malla tipo *Noise*.

Figura 4.11: Gráficos de numero de iteraciones para los algoritmos de Cleap. Se observa que la tendencia es que el algoritmo Cleap-Determinante tenga mas iteraciones que Cleap-Ángulo, para ambos tipos de malla.



(a) Resultados para malla tipo *Random*.

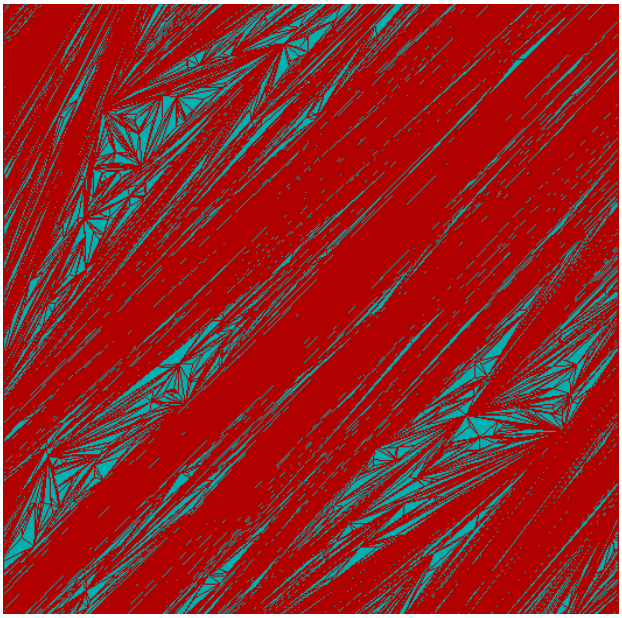


(b) Resultados para malla tipo *Noise*.

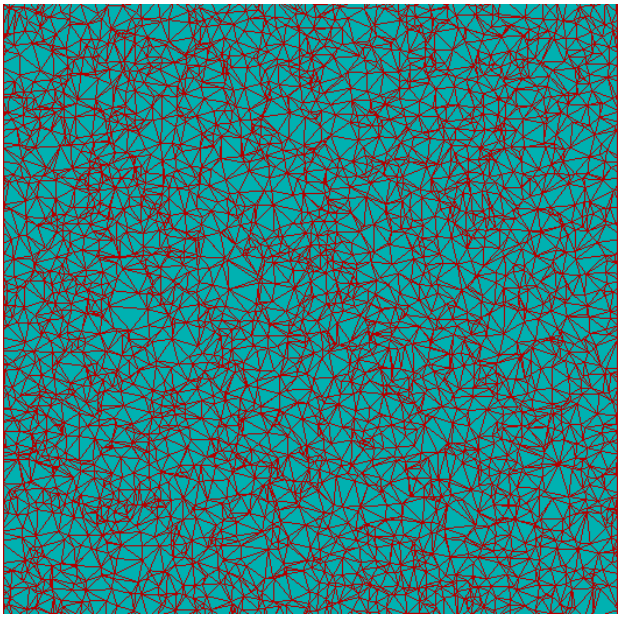
Figura 4.12: Gráficos de porcentaje de error para los resultados de los algoritmos, en comparación a los resultados de CGAL. El error asociado es considerablemente superior para el Algoritmo Cleap-Determinante, para ambos tipos de malla.

4.5.2. Resultados Cualitativos

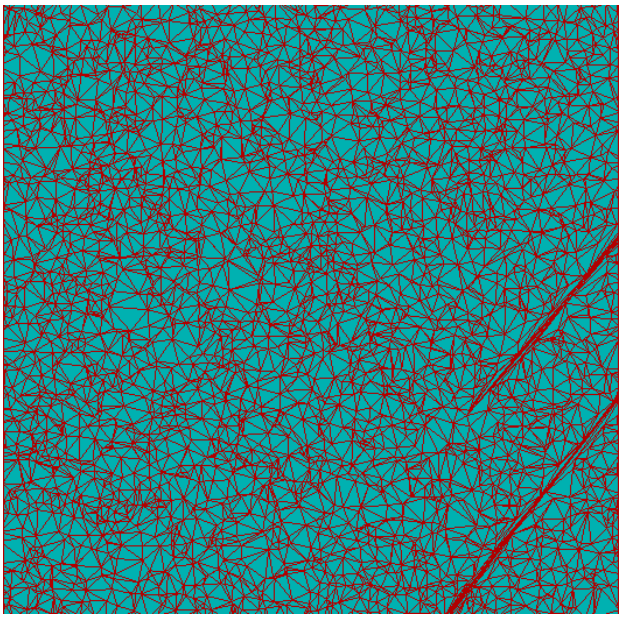
En este apartado, se muestra el resultado final de las mallas luego de aplicar cada Algoritmo. Debido a la gran cantidad de vértices que poseen las mallas, no es posible apreciar cambios y diferencias en la resolución de una hoja de papel para las mallas en su tamaño original, por lo que se mostraran imágenes con un nivel de acercamiento suficiente para apreciar los detalles.



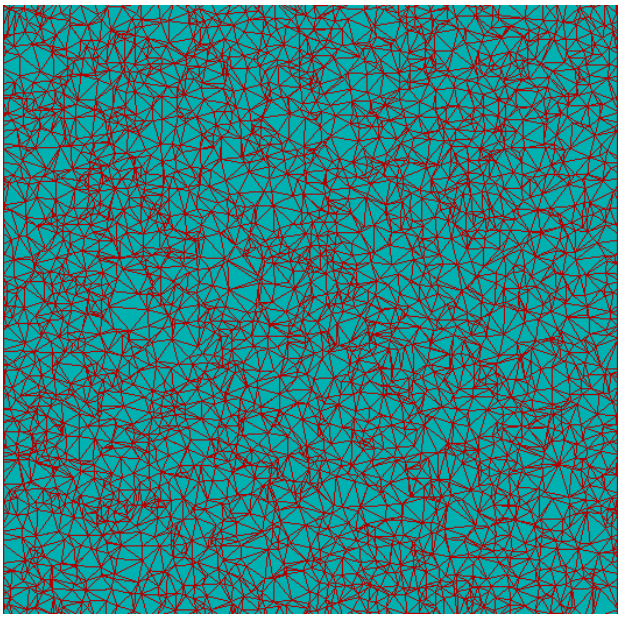
(a) Malla original



(b) Resultado Cleap-Ángulo

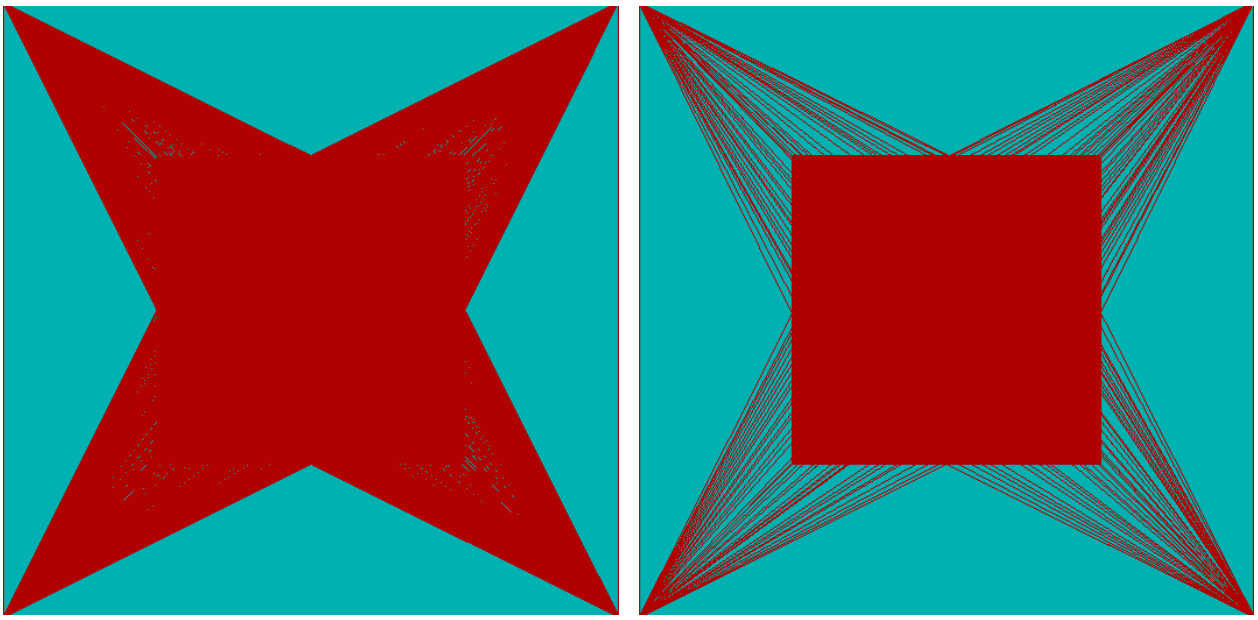


(c) Resultado Cleap-Determinante



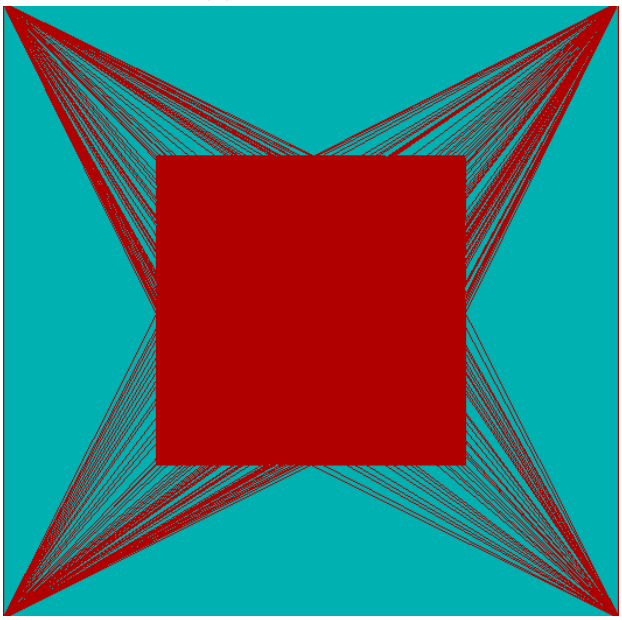
(d) Resultado Kohout

Figura 4.13: Resultados para Triangulación de Malla tipo *random* de 100.000 vértices. Se observan ciertas imperfecciones en el resultado de Cleap-Determinante. Por otro lado, Cleap-Ángulo y Kohout producen un resultado similar.

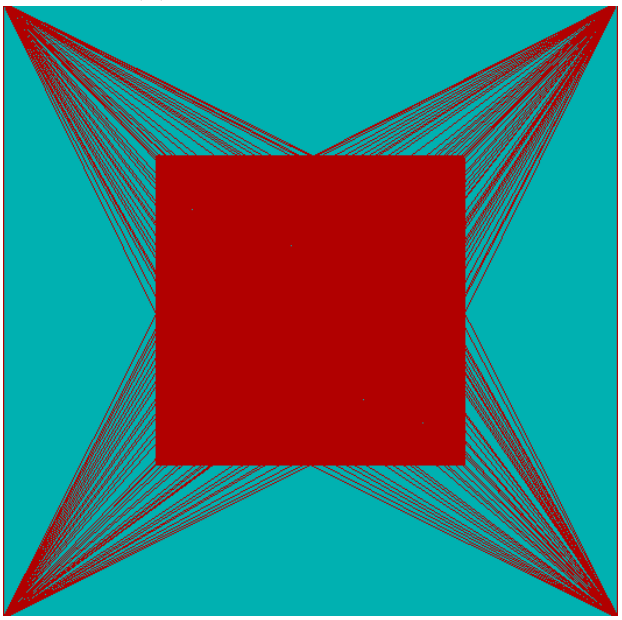


(a) Malla original

(b) Resultado Cleap-Ángulo

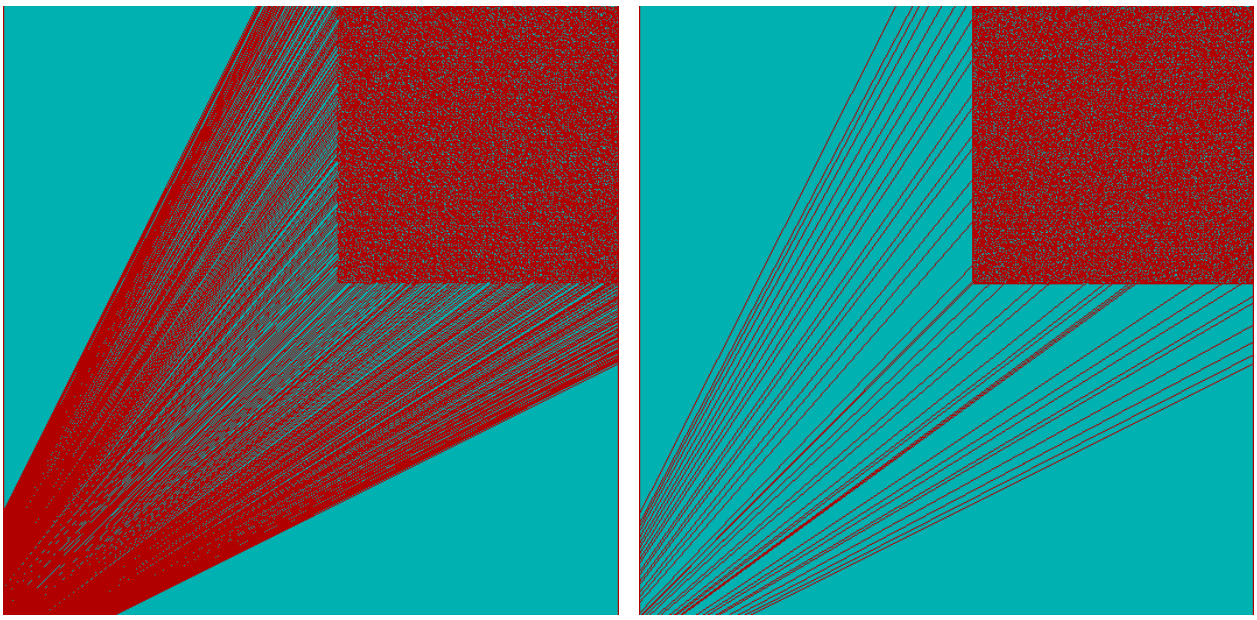


(c) Resultado Cleap-Determinante



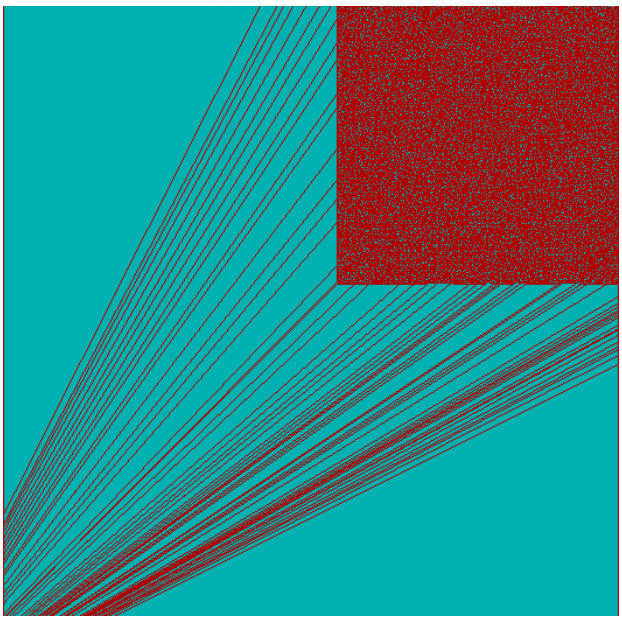
(d) Resultado Kohout

Figura 4.14: Resultados para Triangulación de Malla tipo *noise* de 100.000 vértices, sin acercamiento. Se observan resultados similares para los tres algoritmos, pero Cleap-Determinante posee algunas imperfecciones en las lineas laterales.

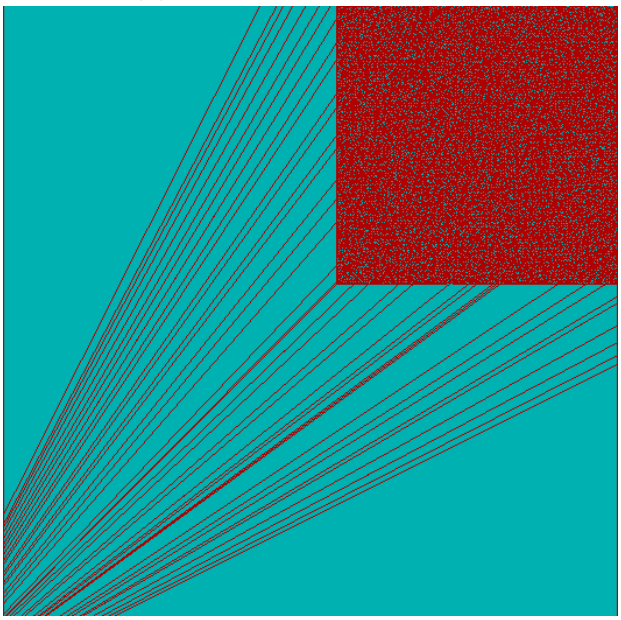


(a) Malla original

(b) Resultado Cleap-Ángulo



(c) Resultado Cleap-Determinante



(d) Resultado Kohout

Figura 4.15: Resultados para Triangulación de Malla tipo *noise* de 100.000 vértices, con acercamiento a uno de sus extremos. Se observa que Cleap-Determinante posee algunos arcos bastante juntos entre si, que no poseen los demas.

4.6. Conclusiones

Luego de analizar los resultados cuantitativos, se obtuvieron las siguientes conclusiones:

Por un lado, a partir de los gráficos de la Figura 4.10, se deduce que el algoritmo de Kohout supera por al menos un orden de magnitud en tiempo de ejecución, a los demás algoritmos para mallas de tipo *random*. Y en el caso de mallas *noise*, el tiempo de ejecución entre Cleap-ángulo y Kohout son similares, mientras que el rendimiento del algoritmo de Cleap-determinante y CGAL también son similares entre si. Acá se deben destacar dos resultados que no se esperaban. Primero, que el algoritmo de Cleap-determinante no tuviese mejor rendimiento que su contraparte con el criterio de los ángulos opuestos. Y segundo, que los algoritmos en GPU de Cleap, no presentaran un rendimiento superior al del algoritmo de Kohout en CPU.

Por otro lado, al comparar los resultados obtenidos, y calcular que tanto difieren con respecto a los resultados producidos por CGAL (Figura 4.12), se observa que el error asociado a los resultados producidos por el algoritmo de Cleap-determinante es demasiado alto en comparación a los demás (al menos tres ordenes de magnitud), para ambos tipos de malla. Además se observan dos tendencias interesantes. La primera es que el error asociado a los algoritmos de Cleap tiende a crecer lentamente a medida que aumenta el tamaño de la malla, pero solo el algoritmo de Cleap-angulo, se mantiene en un rango razonable ($\approx 0{,}001\,\%$), mientras que su contraparte con el criterio del determinante, posee un error en torno al $15 - 30\,\%$. La segunda es que el error del algoritmo de Kohout, tiende a disminuir a medida que aumenta el tamaño de la malla. En particular para las mallas de tipo *noise*, este error es 0 (lo cual se reafirma con la Tabla 4.5, lo que quiere decir que los resultados producidos, son iguales a los producidos por CGAL.

Finalmente, del gráfico de la Figura 4.11, se observa que el numero de iteraciones realizada por el algoritmo de Cleap-determinante, tiende a ser superior con respecto al algoritmo Cleap-ángulo. Esto nos dice que algo esta ocurriendo con el algoritmo de Cleap-determinante, que en teoría no se esperaba. Antes de presentar explicaciones para este fenómeno, observemos los resultados cualitativos.

Por un lado, de la Figura 4.13, se observa que para mallas tipo *random*, los resultados obtenidos por Cleap-ángulo y Kohout, son prácticamente iguales, mientras que el resultado obtenido por Cleap-Determinante, presenta ciertas imperfecciones, en forma de segmentos transversales, con triángulos que claramente no respetan la condición de Delaunay. Estas conclusiones se mantienen al observar las Figuras 4.14 y 4.15, con los resultados para mallas tipo *noise*.

Para analizar este problema, tomemos como referencia, la Figura 4.16, en donde se observa a grandes rasgos el principal problema, que consiste en los segmentos longitudinales, que no respetan la condición de Delaunay. Lo curioso es que si en teoría, bastaba con cambiar un criterio por otro para que el algoritmo funcionara, entonces porque solo uno de los criterios producía estas imperfecciones.

Luego de analizar el problema, en conjunto con el creador de la librería Cleap, en el cual

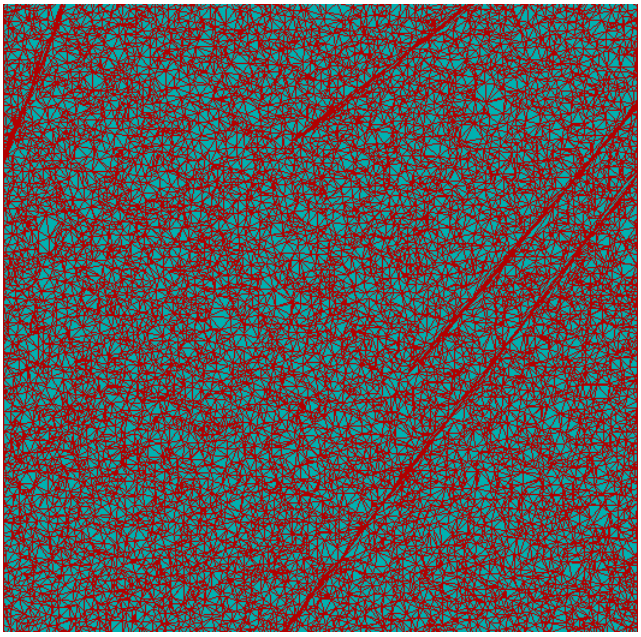


Figura 4.16: Detalle Cleap-Determinante. Se observan imperfecciones en la malla resultante.

se analizaron las fases del algoritmo. En particular, se desacoplaron las fases de detección y exclusión (es decir, no puede iniciar la fase de exclusión si todos los *threads* no han terminado la fase de detección), y así confirmar si el problema estaba a nivel de la fase donde se aplica el criterio de detección o no. De acá, se lograron deducir dos principales problemas.

El primero es que a nivel de la fase de Detección y Exclusión, puede ocurrir una *race-condition*, que consiste en que un *thread* pasa la fase de Detección, pero otro *thread* realiza una modificación (*edge-flip*) sobre la vecindad del *thread* anterior, por lo que este *thread* luego estaría procesando sobre una estructura de datos aun no actualizada (lo cual se arregla en la fase de reparación). Esto se evidencia a nivel de la fase de detección, en donde se encuentran vértices repetidos para la verificación del criterio, lo que en teoría compromete la lógica de estos *test*.

Lo segundo, es que a nivel del *test* del determinante para cuatro vértices, existen casos particulares de falsos positivos cuando el valor de este determinante es cercano a cero, lo que es equivalente a que la suma de los ángulos opuestos sea cercana a 0, π o 2π para el caso del criterio del ángulo. Estos falsos positivos, producen que se realice un *edge-flip* cuando no se debe realizar, lo cual corrompe la estructura de datos, y a partir de ese momento la malla también queda corrupta, produciendo singularidades como las de la Figura 4.16. Por otro lado, los falsos negativos no causan problemas, ya que mantienen la triangulación, salvo que algunos puntos no respetaran la condición de Delaunay.

Para el primer problema, se concluyó que esta condición no afecta la integridad del algoritmo como un todo, debido a que si un *thread* realizó la fase de detección con información inconsistente, entonces significa que otro *thread* ya paso la fase de exclusión en esa vecindad, por lo que el primer *thread* quedara excluido de hacer *edge-flip*, y su inconsistencia sera

reparada en la fase de reparación.

Para el segundo problema, se determinó que para estos casos de borde, el criterio del determinante genera un falso positivo, mientras que el criterio del ángulo genera un falso negativo. Así se tiene que debido a estos falsos positivos, se generan inconsistencias en la estructura y consistencia de la malla al usar el criterio del determinante. Esto nos hace pensar que es necesario agregar filtros de punto flotante al calculo del determinante, debido a que se compone de muchas multiplicaciones y calculo de potencias entre números decimales, y en particular cuando las coordenadas de los vértices están entre 0 y 1, los resultados convergen mucho mas rápido a cero, por lo que dependiendo de la precisión utilizada en estos números, se puede perder valiosa información en los decimales menos significativos de ellos. El principal problema de esto, es que agregar filtros de punto flotante a nivel de la GPU, puede encarecer bastante el tiempo de ejecución del algoritmo, por lo que en principio, no seria una alternativa viable como si lo es al procesar en la CPU.

Como conclusión adicional, podemos asegurar que a pesar de que los tiempos de ejecución entre los algoritmos de Kohout, y Cleap-ángulo son ligeramente distintos, los resultados producidos por ambos son similares. También hay un detalle, que no fue considerado en las mediciones de tiempo originales, es que no se consideraron los tiempos que demora en cargar la malla geométrica en memoria los algoritmos. Para esto ultimo, se observo que ocurre todo lo contrario a los tiempos de ejecución del algoritmo en si, es decir, el algoritmo de Kohout demora al menos un orden de magnitud adicional en tiempo que su contraparte en Cleap al cargar los datos de la malla en memoria. Esto solo fue observado al momento de realizar las pruebas, y no se considero relevante para medirlo, puesto que el foco era medir los tiempos de ejecución del algoritmo de triangulación de Delaunay, sin considerar factores que no estuviesen directamente relacionados con el algoritmo y los resultados que este produce. Así, puede que en algunos casos sea mejor considerar el uso de un algoritmo por sobre otro, en función del tiempo total que demora en ejecutarse como un todo (desde que se ingresa la malla, hasta que sale la malla resultante), y este factor debería ser considerado en futuras mediciones.

Además, se observo que la implementación de Kohout, no logró cargar mallas que superaban los 1.500.000 vértices, la cual es la razón principal por la cual la batería de test no considera mallas de mayor tamaño. Esto, por un lado, es una limitante importante en esta implementación, y por otro lado, no permite ver si la tendencia de estos resultados se mantiene para mallas de tamaño superior.

Finalmente, la GPU utilizada para las pruebas, corresponde a un modelo de *notebook*, los cuales están focalizados en movilidad y ahorro de energía, y son menos poderosas que sus contrapartes para computadores de escritorio (Nvidia GeForce X) y maquinas especializadas en calculo científico (Nvidia Tesla). Por lo que se espera que al realizar estas pruebas en alguna de estas plataformas, debería mejorar bastante los resultados de los algoritmos de Cleap, y así obtener resultados mas similares a los obtenidos en el articulo donde se describe el algoritmo en la librería Cleap[12].

Capítulo 5

Algoritmo de Simplificación Edge-collapse

En este capítulo, se abordará la revisión bibliográfica realizada, en miras de proponer un algoritmo de simplificación de mallas basado en *edge-collapse*, en paralelo en la GPU. Primero se explorará la teoría que existe detrás de este tipo de simplificación. Luego se estudiaran más a fondo, diversas formas de realizar algunas fases de este algoritmo, y finalmente se presentaran análisis y conclusiones con respecto a esta investigación, cuyo principal objetivo es integrar a Cleap un algoritmo de simplificación de mallas geométricas, que funcione bien para el caso particular de mallas de superficie que representan terrenos y superficies geograficos.

5.1. Trasfondo Teórico

La simplificación de mallas geométricas, es una operación que permite disminuir el tamaño y complejidad de una malla (principalmente en cantidad de vértices, reduciendo su tamaño en memoria), manteniendo un nivel de calidad aceptable para que los resultados obtenidos y procedimientos realizados con esta malla simplificada, sigan considerándose válidos y/o útiles. Así, es posible mejorar la eficiencia de otros algoritmos y procedimientos que utilicen esta malla simplificada, debido a que por su tamaño reducido, es más sencillo y rápido de procesar. También eventualmente puede permitir que estas mallas simplificadas puedan ser visualizadas en dispositivos, que por limitaciones de hardware, era complicado o lento lograr una buena *renderización*, como en *tablets* o celulares.

Una de las técnicas dentro de la simplificación de mallas es conocida como *edge-collapse*, que consiste en eliminar un arco de la malla, reemplazando los vértices que lo componen, por un solo vértice, lo que también elimina a los triángulos que contienen a este arco. La operación inversa, se conoce como *vertex-split*. Ambas operaciones se describen gráficamente en la Figura 5.1, y en la Figura 5.2 se puede apreciar un ejemplo en el cual se aplica este procedimiento, con diferentes grados de detalle en el resultado final, en función del criterio elegido para colapsar un arco.

En general, los pasos a realizar para un procedimiento de *edge-collapse*, son los siguientes:

1. Establecer un criterio o métrica para determinar si un arco debe ser colapsado o no. Algunos criterios pueden ser tan simples como el orden en que se encuentran en la estructura de datos, o bien pueden ser más complejos, como utilizar métricas de errores para determinar que arcos producen menos deformación de la geometría de la malla al terminar el proceso.
2. Determinar en qué orden se procesaran los arcos que superaron el criterio anterior.
    - En el caso de implementaciones paralelas, además es necesario contar con condiciones de exclusión mutua entre hilos de ejecución, para evitar problemas asociados a la modificación concurrente sobre la malla.
3. Procesar estos arcos, removiéndolos de la malla, junto a todos sus datos asociados. Además es necesario actualizar la estructura de datos para ver reflejados estos cambios, en caso de ser necesario.

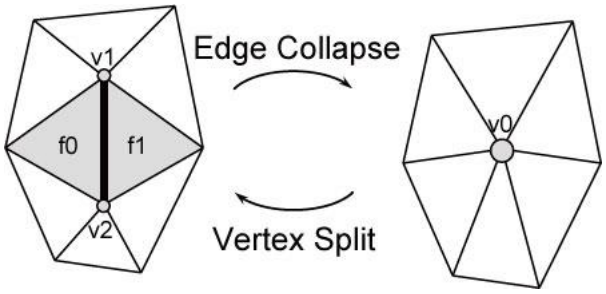


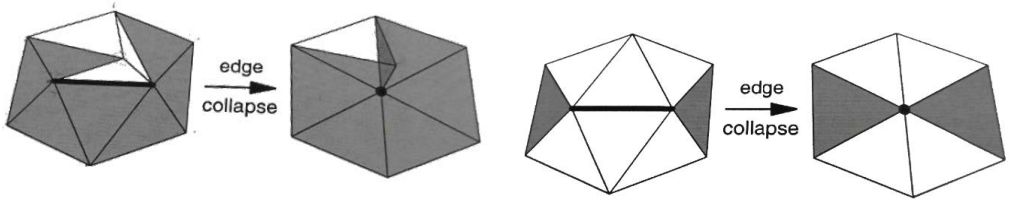
Figura 5.1: Operación de *edge-collapse* y vertex-split en una malla 2D.

Figura 5.2: Ejemplo de como se ve una malla afectada por el procedimiento de *edge-collapse*, con distintos grados de calidad. Se puede apreciar que la calidad disminuye a medida que disminuye la cantidad de puntos que componen la malla.

Uno de los principales problemas que presenta esta operación, es que, dependiendo del tamaño y complejidad de la malla, la operación de simplificado puede demorar bastante tiempo (del orden de horas, o incluso días). Debido a esto, se han investigado diversas formas de mejorar el rendimiento, sin perder demasiada calidad, desarrollando optimizaciones a los algoritmos secuenciales, o desarrollando algoritmos que funcionen en arquitecturas multi-core.

La forma de comparar los resultado de las soluciones producidas por las diversas implementaciones de este método de simplificado, se miden principalmente en:

- El tiempo de ejecución del algoritmo.
- La cantidad de memoria utilizada por el algoritmo.
- Que el resultado de la operación preserve la topología y la geometría de la malla original, lo cual se puede apreciar gráficamente en la Figura 5.3.



(a) Operación de *edge-collapse*, que intersecta dos caras de la malla, lo que no preserva la geometría.

(b) Operación de *edge-collapse*, que conecta dos regiones que antes no estaban conectadas, lo que no preserva la topología.

Figura 5.3: Problemas que pueden ocurrir al no preservar la topología o la geometría al realizar un *edge-collapse*

Por otro lado, no es trivial paralelizar un algoritmo basado en este procedimiento, debido a que, como se vio en los capítulos anteriores, no se puede contar con estructuras de datos muy complejas en la GPU (lo que también limita enfoques basados en orientación a objetos). Ademas, cada hilo de ejecución debe mantener control sobre una vecindad en torno al arco candidato a ser colapsado, lo que requiere agregar fases de exclusión mutua y sincronización entre estos hilos. Esto, para evitar que se produzcan fallas en la geometría y/o la topología de la malla, luego de realizar varios *edge-colapse* en paralelo.

Además, en un articulo de investigación [9], se describen diversos problemas que pueden aparecer al realizar un procedimiento de *edge-collapse*, como por ejemplo:

- Intersección de áreas (2D) o volúmenes (3D) al interior de la malla.
- Problemas cuando el perímetro de la malla no es una curva convexa.
- Perdida de la topología y/o geometría de la malla.

Junto a esto, proponen ciertos procedimientos para evitar o solucionar estos problemas, que se basan principalmente en realizar una serie de *test*, que verifican que el eventual *edge-collapse* no provoque alguno de los problemas antes listados. Estos *test* son lo suficientemente robustos, como para tener una idea general de lo que es necesario para realizar un buen procedimiento de *edge-collapse*, y tener la precaución de considerarlos al momento de paralelizar el algoritmo.

Con esto, se investigaron diversas propuestas e implementaciones de algoritmos de simplificación basados en *edge-collapse*, las cuales proponen diversas formas de elegir el criterio con el cual se colapsarán los arcos, y en el caso de implementaciones paralelas, el criterio de exclusión mutua entre los hilos de ejecución y como funcionan en la teoría. La revisión bibliográfica realizada se expone en la siguiente sección, con miras a proponer un algoritmo de simplificación que sea útil para los propósitos antes mencionados.

5.2. Revisión Bibliográfica

En primer lugar, se revisaron algunas implementaciones secuenciales en CPU. Dado que son implementaciones secuenciales, no es necesario introducir condiciones de exclusión mutua, por lo que la revisión se enmarcará en identificar el criterio para realizar un *edge-collapse* sobre un arco o no.

Una de las métricas propuestas[10], consiste en utilizar el largo de los arcos, ponderado por la curvatura formada por las caras que comparten el arco en el caso 3D, como una medida, y el criterio consiste en colapsar todos los arcos cuya métrica sea menor a un largo arbitrario, lo cual tiene la ventaja de ser sencillo de computar, lo que permite tener diversas aplicaciones practicas para aplicaciones en tiempo real. La principal desventaja es que al ser una métrica simple, no es lo suficientemente apropiada para todos los casos particulares que puedan poseer las mallas, en particular, cuando estas poseen muchas curvaturas, o son muy suaves.

También se revisó un concepto más amplio que *edge-collapse*, conocido como *vertex-contraction*[6], que consiste en contraer dos vértices, que no necesariamente están unidos por un arco, lo que permite unir regiones de la malla que anteriormente no estaban unidas, lo cual produce que se pierda la topología de la malla. Lo interesante es el criterio utilizado para determinar si contraer o no un par de vértices, que utiliza una medida llamada *Quadric Error Metrics*. Esta medida consiste en calcular el error que se producirá en la malla al contraer un par de vértices, en función de sus eventuales nuevas posiciones, lo cual permite medir cuanta calidad se pierde en la malla al realizar esta operación con un determinado par de vértices, y así encontrar una cadena de operaciones que produzca un buen resultado, en función de la calidad que se desea. Con esto, también notamos que a grandes rasgos, *edge-collapse* es un caso particular de *vertex-contraction*, en el cual solo se pueden compartir vértices que sean compartidos por el mismo arco. Una de las desventajas que posee esta métrica, es que requiere ser computada cada vez que se realiza una operación, lo cual puede ser costoso en tiempo de ejecución del algoritmo como un todo.

En segundo lugar, se revisaron algunas implementaciones paralelas en GPU. Dado que son implementaciones que funcionan en paralelo, es necesario además considerar el criterio de exclusión mutua utilizado para evitar problemas de acceso concurrente entre los *threads*.

Una de las soluciones revisadas[13], utiliza como criterio para realizar el *edge-collapse*, la misma métrica utilizada en el algoritmo secuencial anterior, *Quadric Error Metrics*. Como criterio de exclusión mutua, se introduce el concepto de *áreas de independencia*. Dos arcos

se encuentran en *áreas de independencia* distintas, si los triángulos que son vecinos a los triángulos que contienen a este arco, no poseen vértices en común. Con esto, la condición de exclusión mutua consiste en colapsar solo los arcos que pertenecen a *áreas de independencia* distintas. Ademas, los autores realizaron la implementación en el lenguaje *OpenCL*, lo cual en teoría permite que sea compatible con una gran cantidad de arquitecturas y dispositivos (GPU's), a cambio de que no se aproveche el potencial y características de una arquitectura en particular, como *Nvidia-Cuda* o *ATI-APP*. Por otro lado, en este documento no queda claro como se implementaron estas partes del algoritmo, o bien quedan muy poco claros. Sin embargo, esto sirve como punto de partida para la búsqueda de otros criterios, y compararlos teóricamente.

Otra solución revisada[5], utiliza como criterio para realizar un *edge-collapse* o no, una métrica conocida como *Volume Error Metrics*, la cual consiste en asignar una pequeña esfera de radio arbitrario a cada vértice, y la idea es minimizar la pérdida de volumen de estas esferas en los vértices que no fueron eliminados tras la operación de *edge-collapse*. Con respecto a la condición de exclusión mutua, se propone una alternativa que cumple el mismo propósito, que consiste en subdividir la malla en pequeños segmentos, y cada uno de estos segmentos es procesado por un *thread*. Junto a esto, es necesario marcar como "intocables" a los arcos que son compartidos entre estos segmentos, los cuales no pueden ser candidatos a ser colapsados. Por un lado, esta propuesta de exclusión mutua es bastante interesante, ya que no se debe lidiar con demasiadas herramientas de sincronización, que pueden disminuir el rendimiento del algoritmo. Por otro lado, la calidad visual de la solución producida, no es del todo satisfactoria para algunos casos, debido a que la restricción de los arcos "intocables", que produce un patrón de arcos que no puede ser colapsado, como se aprecia en la figura 5.4.

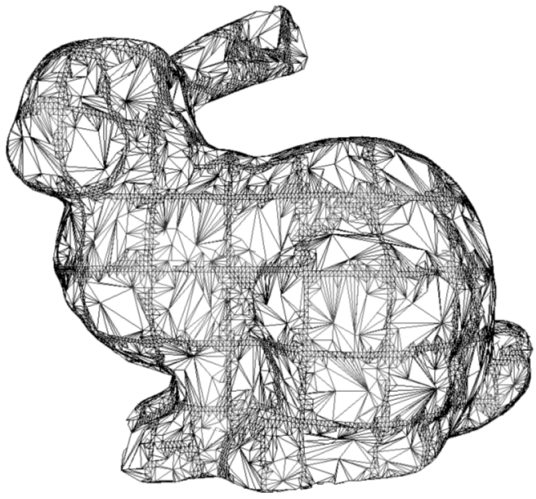


Figura 5.4: Malla simplificada, en la cual se aprecian áreas poco homogéneas, debido a los sectores protegidos de la malla que no pueden ser colapsados.

En último lugar, se revisó una implementación particular[14], la cual propone utilizar en conjunto las capacidades de la CPU y la GPU (enfoque *hibrido*), buscando un balance óptimo en las cargas de trabajo de ambas unidades, aprovechando el hecho de que para mallas pequeñas, un algoritmo secuencial tiende a ser mas rápido que sus contrapartes en

paralelo. Primero, se debe determinar cuantos y cuales arcos procesará cada unidad. Luego, para el caso de la CPU, se procede con un algoritmo tradicional, que recorra todos los vértices asignados y determine que arcos colapsar. Para el caso de la GPU, se presentan dos posibles condiciones de exclusión mutua.

En la primera, cada vértice maneja un campo adicional, que indica si el vértice ha sido marcado por algún *thread*. Con esto, la idea es que cada *thread* debe revisar los vértices de los triángulos afectados por la eventual operación de *edge-collapse*, y tener control exclusivo sobre todos ellos. Si un *thread* detecta que algún vértice ya esta tomado, entonces libera los que eventualmente pudo haber tomado, y pasa al siguiente arco. En la segunda, solo se realiza una operación de *edge-collapse*, si no se ha realizado otro *edge-colapse* en la vecindad del arco a colapsar, durante la actual iteración en paralelo. Este concepto de vecindad, puede ser apreciado en detalle en la Figura 5.5.

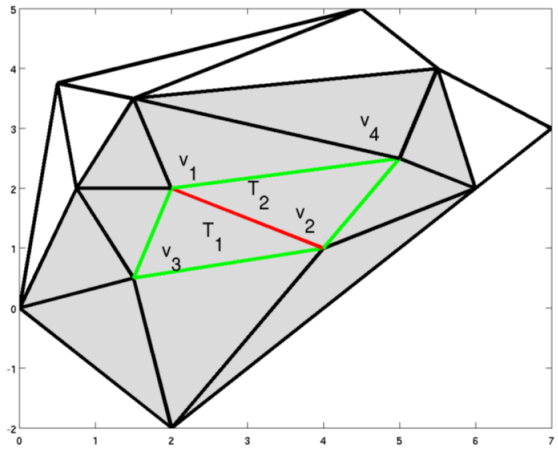


Figura 5.5: La vecindad del arco (en rojo), consiste en identificar los triángulos que serán eliminados por el eventual *edge-collapse*(en verde), y luego la vecindad corresponde a todos los triángulos que comparten al menos un vértice con los triángulos que serán eliminados (área en gris)

Sin embargo, uno de los detalles encontrados en esta implementación, es que, al parecer, el criterio de elección de los arcos a colapsar, consiste simplemente en el orden en que se encuentran los arcos en la estructura de datos. Esto produce que en algunos casos, existan regiones de la malla que sean colapsadas mas de la cuenta, por lo que la malla pierde bastante calidad al paso de pocas iteraciones.

5.3. Propuesta de Algoritmo y Conclusiones

Con la revisión bibliográfica del capitulo anterior, ya se cuenta con el conocimiento necesario para proponer una versión propia del algoritmo, utilizando elementos vistos en esta revisión.

Para contextualizar lo que necesita realizar, revisamos la Figura 5.6, que muestra un caso de malla geométrica que representa una superficie de terrenos. Se puede apreciar que existe una gran cantidad de puntos, que le brindan detalle a la malla, y lo que se busca es identificar los relieves y los puntos importantes de este tipo de mallas, para los cuales no se requiere gran cantidad de detalles. Ademas, en la Figura 5.7 se puede apreciar una pequeña comparación, entre el detalle actual de una malla, y el resultado se que espera obtener al aplicar este procedimiento.

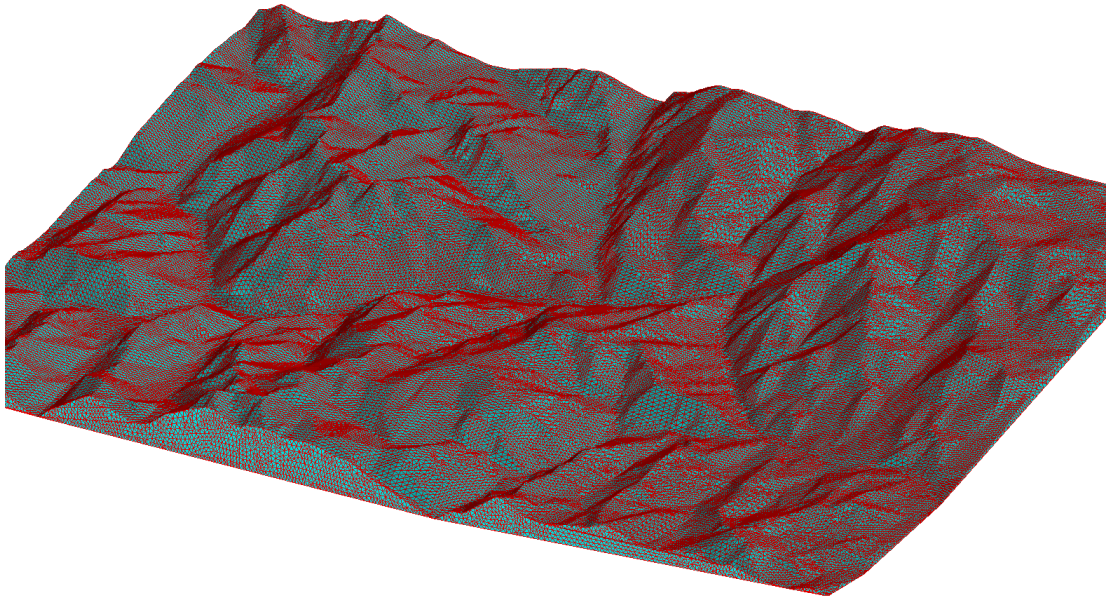
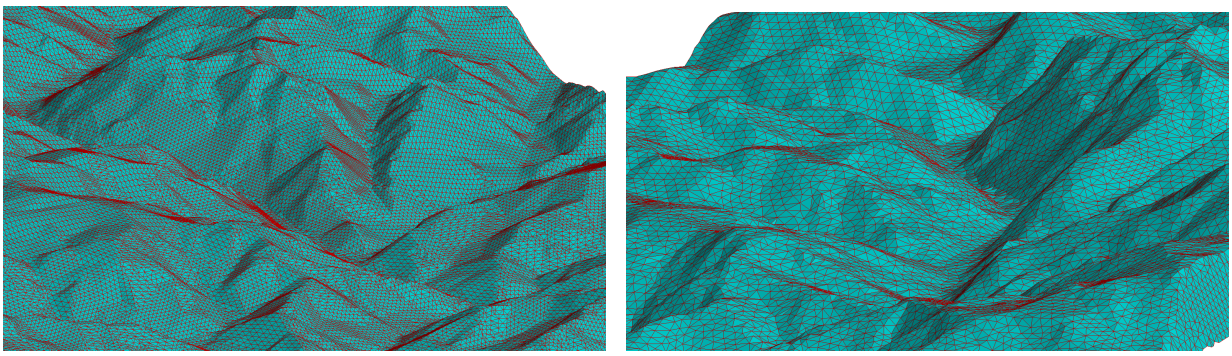


Figura 5.6: Ejemplo de malla que representa un terreno. La concentración de rojo indica que hay una mayor densidad de vértices y arcos representando esa área en particular.

Con esto, tenemos que definir que criterios utilizaremos para las principales partes del algoritmo. Para el criterio de definir que arcos deben ser colapsados, se estima que la mejor opción es usar un criterio que consiste en colapsar los arcos tal que su largo, sea menor a una cierta distancia determinada[10]. Esto se cree, debido a que como se puede apreciar en la Figura 5.6, las zonas con mayor densidad (de color rojo), indican que hay una mayor cantidad de vértices y arcos representando esta zona, por lo que es esperable que los arcos y vértices estén a corta distancia entre si, por lo que en teoría es posible representar esta superficie con una menor cantidad de vértices, sin perder demasiada calidad.

Para el criterio de exclusión mutua, consideramos que la mejor opción, es el criterio utilizado por los autores del algoritmo de *edge-collapse* híbrido[14], que consiste en que un arco solo puede ser colapsado, si en su vecindad no ha sido colapsado otro arco. Esto, debido a que en teoría, solo se requiere mantener una estructura de datos compartida a nivel de la GPU,



(a) Ejemplo de malla original con gran nivel de detalle.

(b) Ejemplo de malla luego de realizar una simplificación.

Figura 5.7: Numero de Iteraciones para Iso-Esfera de $\approx 2K$ Vértices

que mantenga información por triángulos, que indique si este triangulo ha desaparecido o pertenece a una vecindad donde se colapsó un arco o no. Así, se puede revisar y realizar atómicamente la verificación si la vecindad presenta un arco colapsado o no, y luego actualizar esta estructura dentro de la misma iteración.

Finalmente, un punto que no fue abordado en esta composición del algoritmo, es el como reparar y actualizar la estructura de datos principal que almacena la malla, luego de realizar una iteración de *edge-colapse* en paralelo. De acuerdo a la revisión bibliográfica realizada, se tiene que esta fase del algoritmo está ligada y es dependiente de como funcionen las demás partes del algoritmo. Por lo que esto debe ser evaluado, cuando se tenga claro como se implementarán en especifico las demás partes del algoritmo en Cleap.

Capítulo 6

Conclusiones y Trabajo Futuro

6.1. Conclusiones

En este trabajo de título se realizó una investigación relacionada con algoritmos de suavizado, triangulación y suavizado sobre mallas geométricas, con el fin de integrar implementaciones de estos algoritmos en la librería Cleap, desarrollada por Cristóbal Navarro el año 2013.

Con respecto al algoritmo de suavizado de mallas, se logró implementar e integrar el algoritmo de suavizado de Taubin, tanto en una versión secuencial en CPU, como una versión paralela en GPU. Los resultados de estas implementaciones muestran que ambas implementaciones generan las mismas mallas de salida, pero la implementación paralela es aproximadamente 7 veces mas rápida que su contraparte secuencial. Por otro lado, se observó que para ciertos casos, el algoritmo no es capaz de reconstruir la malla original, a partir de la malla deformada, principalmente debido a que la malla tiene un nivel de deformación tal, que el algoritmo no tiene la suficiente información para reconstruir la forma original de la malla. Para las pruebas realizadas, si el nivel de deformación es relativamente bajo (aproximadamente entre los niveles 1 y 3 de las pruebas realizadas), entonces en la mayoría de los casos era posible reconstruir una figura bastante similar a la original.

Con respecto al algoritmo de triangulación, se investigaron dos implementaciones, de Kohout en CPU, y Cleap en GPU (con el criterio del ángulo). Basándonos en esta ultima, se implementó una segunda versión paralela en GPU, que utiliza un criterio distinto para determinar cuando se debe realizar un *edge-flip* o no (criterio del determinante). Los resultados obtenidos entre estas tres implementaciones, muestran que las implementaciones de Kohout y Cleap-Ángulo obtienen resultados muy similares, pero la implementación de Kohout es ligeramente mas rápida en la ejecución del algoritmo en sí. Por otro lado, la implementación de Cleap-Determinante, tuvo un rendimiento inferior a los demás, y sus resultados presentaban un grado de error considerable, en comparación a los demás. Luego de revisar el porqué, se concluyó que este criterio introduce casos de falsos-positivos, debido a errores asociados al cálculo de punto flotante, lo que produce que el algoritmo realice operaciones que no debe hacer, y comprometen la integridad de la malla. Los resultados obtenidos even-

tualmente podrían haber sido mas consistentes si se hubiese utilizado un valor de tolerancia e*psilon*, como en el caso del calculo del ángulo. De todos modos, el calculo del determinante tiene una cantidad superior de multiplicaciones de punto flotante en comparación al calculo del ángulo, por lo que agregar filtros de tolerancia para cada una de estas multiplicaciones, pueden disminuir bastante el rendimiento del algoritmo.

Con respecto al algoritmo de simplificación, se investigaron diversas implementaciones basadas en la operación de *edge-collapse*, tanto secuenciales en CPU, como paralelas en CPU y GPU. Con esto, se descubrieron diversos problemas que pueden ocurrir y precauciones que se deben tomar al momento de realizar esta operación, en especial al trabajar en paralelo. Para este algoritmo en particular, no se llego a la implementación, pero se dejaron los cimientos teóricos necesarios para llegar a proponer una implementación que sea compatible con Cleap.

6.2. Trabajo Futuro

Como se pudo apreciar a lo largo de este trabajo, en especial con los resultados obtenidos, hay muchos tareas interesantes que se pueden desarrollar.

Con respecto al Algoritmo de suavizado de Taubin, un aspecto interesante a desarrollar, es si es posible mejorar los resultados cualitativos de la malla, al variar los factores de expansión λ y contracción μ, y ver si es posible reparar la malla y llegar a su forma original, para casos con altos niveles de deformación. Además se puede ver si es posible realizar una optimización en memoria utilizada, para las estructuras de datos adicionales utilizadas por el algoritmo, ya que se evidenció que para ciertos casos particulares, el espacio utilizado crece considerablemente, cuando la malla no se representa de manera óptima en el formato de compresión de matrices elegido, ELLPACK.

Con respecto al Algoritmo de triangulación de Delaunay, por un lado, queda ver si es posible reparar la implementación de Cleap-Determinante, para considerar los casos de borde mencionados que producen falsos-positivos, sin disminuir demasiado su rendimiento. Por otro lado, queda ver si la tendencia de estos resultados se mantiene o no, al repetir la batería de *test*, en una plataforma similar o superior a la cual se hicieron las pruebas originales con Cleap-Ángulo[12]

Finalmente, con respecto al Algoritmo de simplificación basado en *edge-collapse*, queda completar la propuesta del algoritmo, su implementación y su integración en la librería Cleap. Esto no es una tarea sencilla, ya que, como se vió durante el trabajo realizado, el uso de distintos criterios o estrategias, puede variar bastante los resultados, y en algunos casos, el como implementarlos para un lenguaje de programación particular, no quedan del todo claros. En particular, este es un tema que me gustaría retomar, y considerar como base para la Tesis de Magíster a desarrollar a futuro.

Capítulo 7

Bibliografía

[1] ATI Heterogeneous Computing. `http://developer.amd.com/tools-and-sdks/heterogeneous-computing/`, 2014. [Online; ultimo acceso 10-Junio-2014].

[2] Nvidia CUDA. `http://www.nvidia.com/object/cuda_home_new.html`, 2014. [Online; ultimo acceso 10-Junio-2014].

[3] OpenCL. `http://www.khronos.org/opencl/`, 2014. [Online; ultimo acceso 10-Junio-2014].

[4] Nathan Bell and Michael Garland. Efficient sparse matrix-vector multiplication on CUDA. NVIDIA Technical Report NVR-2008-004, NVIDIA Corporation, December 2008.

[5] Martin Franc and Václav Skala. Parallel triangular mesh decimation without sorting. In *Proceedings of the 17th Spring Conference on Computer Graphics*, SCCG '01, pages 22–, Washington, DC, USA, 2001. IEEE Computer Society.

[6] Michael Garland and Paul S. Heckbert. Surface simplification using quadric error metrics. In *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '97, pages 209–216, New York, NY, USA, 1997. ACM Press/Addison-Wesley Publishing Co.

[7] Josef Kohout. Software de Triangulación de Kohout. `http://herakles.zcu.cz/~besoft/download/PDT.zip`, 2005. [Online; ultimo acceso 10-Junio-2014].

[8] Josef Kohout, Ivana Kolingerová, and Jirí Zára. Parallel delaunay triangulation in e2 and e3 for computers with shared memory. *Parallel Computing*, 31(5):491–522, 2005.

[9] Martin Kraus and Thomas Ertl. Simplification of nonconvex tetrahedral meshes. In *Hierarchical and Geometrical Methods in Scientific Visualization*, pages 185–196. Springer-Verlag, 2002.

[10] S. Melax. A simple, fast, and effective polygon reduction algorithm, November 1998.

[11] Cristobal Navarro. Librería Cleap. `http://users.dcc.uchile.cl/~crinavar/doc/cleap/`, 2011. [Online; ultimo acceso 10-Junio-2014].

[12] Cristobal A. Navarro, Nancy Hitschfeld-Kahler, and Eliana Scheihing. A gpu-based method for generating quasi-delaunay triangulations based on edge-flips. In Sabine Coquillart, Carlos Andújar, Robert S. Laramee, Andreas Kerren, and José Braz, editors, *GRAPP/IVAPP*, pages 27–34. SciTePress, 2013.

[13] A. Papageorgiou and N. Platis. Triangular mesh simplification on the gpu. In *NASAGEM Geometry Processing Workshop*, CGI2013, 2013.

[14] S. Shontz and D. Nistor. Cpu-gpu algorithms for triangular surface mesh simplification. In *Proceedings of the 21st International Meshing Roundtable*, pages 475–492. Springer Berlin Heidelberg, 2013.

[15] Gabriel Taubin. A signal processing approach to fair surface design. In *SIGGRAPH*, pages 351–358, 1995.