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

MEJORAMIENTO DE MALLAS DE TRIÁNGULOS USANDO GPU

MEMORIA PARA OPTAR AL TÍTULO DE INGENIERO CIVIL EN COMPUTACIÓN

GABRIEL ALFONSO SANHUEZA SANHUEZA

PROFESORA GUÍA:
NANCY HITSCHFELD KAHLER

MIEMBROS DE LA COMISIÓN:
MARÍA CECILIA RIVARA ZÚÑIGA
ÉRIC TANTER

Esta memoria fue apoyada parcialmente por Proyecto Fondecyt N° 1181506

SANTIAGO DE CHILE
2018

Resumen

Una malla triangular es un modelo de representación de objetos que consiste en un conjunto de vértices, caras (triángulos) y lados (aristas). Estas mallas son muy usadas en computación gráfica y en simulaciones numéricas. Estas mallas pueden no ser suficientemente buenas como para poder realizar cálculos sobre ellas, dado el riesgo de propagar errores, cuando los triángulos que conforman estas mallas son demasiado delgados.

Para mejorar/refinar los triángulos que se consideran de mala calidad existen distintos algoritmos, tales como Lepp-Bisección, Lepp-Delaunay o Lepp-Centroide, que hasta ahora solo tienen implementaciones en CPU (ya sean secuenciales o multi-core). El concepto de Lepp (*Longest-Edge Propagation Path*) consiste en comenzar con un triángulo de la malla, ir hacia el triángulo vecino por su arista más larga sucesivamente, hasta llegar a la última arista más larga que tienen los triángulos en común, llamada arista terminal.

El objetivo de esta memoria es diseñar e implementar una solución mixta que usa tanto GPU (parte paralela) y CPU (parte secuencial) para el algoritmo Lepp-Centroide, de tal forma de poder explorar las posibilidades que ofrece la GPU en el mejoramiento y refinamiento de mallas.

Para cumplir con este objetivo se implementó la aplicación QLepp2D, la cual consiste en la unión de una biblioteca (QLepp2D-lib) y una interfaz gráfica (QLepp2D-gui) que permiten al usuario cargar una triangulación y mejorarla/refinarla de manera rápida y simple, utilizando el poder que ofrece la biblioteca de OpenCL para paralelizar este procesamiento.

QLepp2D-lib está escrita en C++ y OpenCL 1.2. Esta biblioteca puede ser instalada en el sistema y ser usada por otros programas. QLepp2D-gui está escrita usando Qt y utiliza QLepp2D-lib como demostración de la facilidad de uso de esta biblioteca. Adicionalmente, es posible portar la interfaz a distintos idiomas, por lo que es posible utilizarla en distintos sistemas, traduciendo la aplicación de forma automática.

Con esta aplicación es posible conseguir el mejoramiento/refinamiento de mallas de triángulos con una velocidad más de 2 veces superior para el procesamiento completo y más de 400 veces superior para el algoritmo del método más rápido, además de demostrar el poder de OpenCL para adaptarse a cualquier dispositivo que posea una implementación disponible.

Agradecimientos

A mi familia, que me empujó a mejorar en todos mis años de formación, y me apoyó en todo lo que requiriera para poder seguir mis estudios.

A mis amigos, que me acompañaron y apoyaron en las buenas y en las malas, haciendo más llevadera mi vida.

A mis compañeros de universidad, que me abrieron la mente a nuevas perspectivas.

A mi profesora guía, Nancy Hitschfeld, que se tomó todo el tiempo que requerí para orientarme en todo momento.

A mi profesora co-guía, María Cecilia Rivara, cuyas clases me dieron la idea de realizar este proyecto.

Tabla de Contenido

1. Introducción **1**

1.1. Motivación . . . 2

1.2. Objetivos . . . 3

1.2.1. Objetivo General . . . 3

1.2.2. Objetivos Específicos . . . 3

1.3. Contenido de la Memoria . . . 4

2. Antecedentes **7**

2.1. Triangulaciones . . . 7

2.1.1. Triangulación en abanico . . . 8

2.1.2. Triangulación de peso mínimo . . . 8

2.1.3. Triangulación de Delaunay . . . 9

2.2. Mejoramiento de una triangulación . . . 10

2.2.1. Algoritmo de *edge-flipping* . . . 12

2.2.2. Algoritmos de tipo Lepp . . . 12

2.3. Patrones de Arquitectura de Software . . . 15

2.3.1. MVC . . . 15

2.3.2. ModelView . . . 15

2.4. Patrones de Diseño de Software . . . 15

2.4.1. Factory Pattern . . . 16

2.4.2. Singleton Pattern . . . 16

2.4.3. Strategy Pattern . . . 17

2.4.4. Façade Pattern . . . 17

2.4.5. Bridge Pattern . . . 18

2.5. Herramientas disponibles . . . 19

2.5.1. CUDA vs OpenCL . . . 19

2.5.2. Gtk+ vs Qt . . . 21

2.6. Opciones elegidas . . . 22

2.6.1. Biblioteca de cómputo . . . 22

2.6.2. Biblioteca gráfica . . . 22

3. Análisis y Diseño **23**
3.1. Requisitos 23
3.1.1. Requisitos de Usuario 23
3.1.2. Requisitos de Software 24
3.2. Arquitectura 25
3.3. Algoritmos 25
3.4. Biblioteca 29
3.4.1. Diseño de las Estructuras 30
3.4.2. Diseño del Modelo 30
3.4.3. Módulo de Carga y Guardado de archivos 31
3.4.4. Módulo de Procesamiento 33
3.5. Interfaz Gráfica 34
3.5.1. Módulo MainWindow 37
3.5.2. Módulo OpenGLWidget 38
3.5.3. Módulo About 39
3.5.4. Módulo Tutorial 39

4. Implementación **41**
4.1. Ambiente de desarrollo 41
4.2. Biblioteca 42
4.2.1. Estructuras de Datos 42
4.2.2. Carga de Archivo 44
4.2.3. Guardado de Archivo 47
4.2.4. File Managers 48
4.2.5. Procesamiento 49
4.2.6. Motor de CPU 51
4.2.7. Motor de GPU 63
4.2.8. API 69
4.3. Interfaz Gráfica 71
4.3.1. Ventana principal 72
4.3.2. Menús 74
4.3.3. Carga y guardado 74
4.3.4. Renderizado con OpenGL 77
4.3.5. Tutorial 85
4.3.6. Acerca de 85
4.3.7. Acerca de Qt 86
4.3.8. Internacionalización 86
4.3.9. Keybindings 88
4.3.10. Drag-and-Drop 88
4.3.11. Historial 89
4.4. Instalación y uso de la biblioteca 91
4.5. Instalación y uso de la interfaz 92

5. Mediciones y análisis **93**
5.1. Ambiente de pruebas . . . 93
5.2. Generación de archivos de prueba . . . 94
5.3. Número de elementos de los archivos de prueba . . . 94
5.4. Metodología de obtención de resultados . . . 97
5.5. Resultados . . . 98
5.5.1. Comparación de procesamiento . . . 98
5.6. Análisis . . . 102
5.6.1. Speedup: Nvidia (GPU) . . . 102
5.6.2. Speedup: Intel (Multi-core) . . . 106

6. Conclusiones y Trabajo Futuro **111**
6.1. Aplicación . . . 111
6.2. Resultados obtenidos . . . 112
6.3. Trabajo futuro . . . 112
6.3.1. Biblioteca . . . 112
6.3.2. Interfaz . . . 112

Bibliografía **114**

Anexos **115**
A. Estructura de un archivo OFF . . . 115
B. Patrón NSC y SNC . . . 116
C. Generación de triangulaciones de prueba . . . 118
D. Extendiendo la biblioteca . . . 120
D.1. Añadir un nuevo motor de procesamiento . . . 120
D.2. Añadir capacidad para leer más tipos de archivos . . . 122

Índice de figuras

2.1. Triangulación en abanico. 8
2.2. Triangulación de peso mínimo. 8
2.3. Triangulación de Delaunay. 9
2.4. Triangulación no-Delaunay. 10
2.5. Geometría, malla de Delaunay y refinamiento 11
2.6. *Flip* de una arista 12
2.7. Lista Lepp. 13
2.8. Lepp-Bisección. 13
2.9. Lepp-Centroide. 14
2.10. UML del patrón de diseño Factory. 16
2.11. UML del patrón de diseño Singleton. 16
2.12. UML del patrón de diseño Strategy. 17
2.13. UML del patrón de diseño Façade. 17
2.14. UML del patrón de diseño Bridge 18

3.1. UML básico de las capacidades de la aplicación. 25
3.2. UML simplificado de la biblioteca. 29
3.3. UML de la API. 31
3.4. UML del módulo de carga y guardado de archivos. 32
3.5. UML del módulo de procesamiento. 34
3.6. UML de la Interfaz (Sección principal). 35
3.7. UML de la Interfaz (Sección secundaria). 36
3.8. UML del módulo MainWindow. 37
3.9. UML del módulo OpenGLWidget. 38
3.10. UML del módulo About. 39
3.11. UML del módulo Tutorial. 39

4.1. Estructuras usadas en CPU. 43
4.2. Estructuras usadas en GPU (OpenCL). 43
4.3. Diseño de la interfaz con Qt Designer. 71
4.4. Interfaz de la aplicación. 72
4.5. Ejemplo de menú. 75
4.6. Cargar archivo OFF. 76
4.7. Malla cargada. 81

4.8. Ventana Tutorial. 85
4.9. Ventana Acerca de. 86
4.10. Ventana Acerca de Qt. 86
4.11. Qt Linguist traduciendo QLepp2D. 87
4.12. Ventana en Inglés. 87
4.13. Ventana en Español. 87
4.14. *Keybindings* vía Qt Designer. 88
4.15. Archivos recientes. 90

5.1. 100 triángulos. 95
5.2. 1.000 triángulos. 95
5.3. 10.000 triángulos. 95
5.4. 100.000 triángulos. 95
5.5. 1.000.000 triángulos. 96
5.6. Tiempos para detectBadTriangles. 99
5.7. Tiempos para detectTerminalEdges. 100
5.8. Tiempos totales. 101
5.9. Speedup de detectBadTriangles (Nvidia). 103
5.10. Speedup de detectTerminalEdges (Nvidia). 104
5.11. Speedup total (Nvidia). 106
5.12. Speedup de detectBadTriangles (Intel). 107
5.13. Speedup de detectTerminalEdges (Intel). 108
5.14. Speedup total (Intel). 109

Índice de tablas

5.1. Número de elementos contenidos en cada archivo. 94
5.2. Tiempo de procesamiento para **insertCentroids** (nanosegundos). 98
5.3. Tiempo de procesamiento para **detectBadTriangles** (nanosegundos). . . . 99
5.4. Tiempo de procesamiento para **detectTerminalEdges** (nanosegundos). . . 100
5.5. Tiempo total de procesamiento (nanosegundos). 101

Capítulo 1

Introducción

A lo largo de la historia, y en particular en el siglo pasado y este, los computadores han sido usados cada vez más para cálculos matemáticos, desarrollo de aplicaciones, juegos, etc. Un caso particular, pero importante del uso de computadores está en el procesamiento de figuras geométricas, tanto una por una, como en un conjunto de éstas, como lo son las mallas poligonales.

Una malla poligonal es un modelo de representación de objetos que consiste en un conjunto de vértices, caras (polígonos) y lados (aristas). Los objetos pueden ser tanto del mundo real como del mundo virtual. Esto significa que se pueden representar superficies de terrenos, ciudades, caminos, y también personajes ficticios de videojuegos, entre otros.

Las mallas más ocupadas son las mallas de triángulos, que tienen la particularidad de representar prácticamente cualquier figura geométrica, simplemente descomponiendo estas figuras en dos o más triángulos. Por otro lado, no cualquier malla es útil porque pueden aparecer triángulos con ángulos demasiado agudos u obtusos, los cuales son conocidos por aumentar los errores numéricos en el cálculo de ángulos, vectores normales o áreas, al usar números de punto flotante (*float*) en el computador.

Una de las triangulaciones más conocidas es la triangulación de Delaunay, que es la triangulación que "maximiza el ángulo mínimo" [6] de sus triángulos, lo que la convierte en una de las mejores triangulaciones posibles para un conjunto de vértices, respecto al cálculo numérico sobre los triángulos.

La triangulación de Delaunay es especialmente útil, porque al hacer que se mejore el ángulo mínimo de los triángulos se elimina en gran parte la imprecisión de los cálculos que se requieren hacer con números del tipo *float*, evitando así errores de arrastre. Sin embargo, una malla de Delaunay inicial puede no ser lo suficientemente fina para una simulación numérica. Esto nos lleva a considerar el diseñar e implementar un algoritmo de refinamiento y mejora de triangulaciones.

Los algoritmos de refinamiento consisten en dividir triángulos de una malla con el objetivo de reducir el tamaño de los triángulos y obtener una mejor resolución. Los algoritmos de mejoramiento modifican los elementos geométricos (puntos, aristas, triángulos) que componen la malla en base a una métrica usada hasta poder determinar que la malla es de "calidad".

El proceso de mejorar una triangulación se hace cada vez más lento en la medida que se aumenta el número de puntos de la triangulación. Es por esto que en esta memoria se quiere estudiar la manera de acelerar estos procesos mediante la paralelización de los algoritmos existentes.

Dentro de las arquitecturas de paralelización se pueden considerar las siguientes opciones:

- Multi-core: Usar la CPU (Central Processing Unit).
- Many-core: Usar la GPU (Graphics Processing Unit).

Usar la CPU implica tomar el procesador (o procesadores) de un sistema computacional y asignar tareas independientes para procesamiento, pero actualmente el poder de procesamiento de un computador de escritorio raramente supera los 16 *cores*, los que, además, están orientados a paralelismo de tareas (*task paralell*). Pero sí hay un dispositivo particular que tiene un poder de procesamiento mucho mayor, conocido como GPU. Éste es un dispositivo de *hardware* que posee una cantidad de *cores* muy superiores a una CPU. En promedio, una GPU actual tiene alrededor de 640 *cores*, orientados al procesamiento gráfico y al paralelismo de datos (*data paralell*), que para esta aplicación es mucho más prometedora que una CPU.

1.1. Motivación

En la actualidad existen ciertas bibliotecas que permiten trabajar con mallas 2D y 3D, tales como CGAL[1] y Triangle[2].

Si bien estas bibliotecas son bastante poderosas, tienen algunos detalles importantes que las hacen poco aptas para trabajar con ellas de forma rápida y fácil. Por ejemplo, CGAL tiene demasiadas funcionalidades, por lo que es difícil llegar y usar directamente. Triangle, por otro lado, es más simple y fácil de usar, pero utiliza formatos de archivos inusuales, y no permite un refinamiento fácilmente controlable. Además, ninguna de estas dos bibliotecas utiliza GPU.
Existe una biblioteca llamada Cleap[3], que permite tomar una malla y mejorarla utilizando GPU, pero no la puede refinar (i.e., agregar triángulos), y dado que está hecha en CUDA, solo es posible usarla en computadores que tengan tarjetas gráficas Nvidia, impidiendo su uso en otro tipo de hardware.

[1] https://www.cgal.org/
[2] https://www.cs.cmu.edu/~quake/triangle.html
[3] https://users.dcc.uchile.cl/~crinavar/doc/cleap/

Por otro lado, para poder visualizar mallas de triángulos, existen aplicaciones dedicadas como Geomview[4] o Tetview[5], pero estas interfaces utilizan mayormente bibliotecas gráficas obsoletas, son poco intuitivas y no tienen ningún tipo de internacionalización (idiomas), además de tener código difícil de entender y extender.

La motivación principal de crear esta aplicación es la de poder tener una biblioteca simple y extensible que permita mejorar mallas usando el algoritmo "Lepp-Centroide" [13], cuya implementación pueda utilizar GPU, pero sin limitar su uso a solo una marca de tarjetas gráficas, con lo cual se puedan lograr *speedups* superiores a implementaciones secuenciales. Además, se busca tener un visualizador fácil de usar, atractivo e intuitivo, que pueda utilizar esta biblioteca cuyo código sea fácil de extender, y que tenga el potencial de utilizarse en varios idiomas.

1.2. Objetivos

1.2.1. Objetivo General

Diseñar y desarrollar una aplicación que permita al usuario visualizar y mejorar triangulaciones en 2D, implementando algoritmos que usen la GPU. Además, la herramienta debe ser capaz de exportar las mallas para poder usarlas en otras aplicaciones que trabajen con ellas.

1.2.2. Objetivos Específicos

- Diseñar e implementar un algoritmo de mejoramiento/refinamiento de mallas basado en la técnica Lepp-Centroide, que use GPU.
- Diseñar e implementar un visualizador de mallas triangulares que permita importar y exportar archivos con información de las mallas. (Preferentemente en formato OFF).
- Comparar esta implementación con versiones secuenciales y paralelas en multi-core.
- Crear una biblioteca que se pueda usar en otras implementaciones de visualizadores.
- Crear código legible y extensible a futuros desarrollos.

[4] `http://www.geomview.org/`
[5] `http://wias-berlin.de/software/tetgen/tetview.html`

1.3. Contenido de la Memoria

Introducción

Se presentan los aspectos generales del trabajo, la motivación y los objetivos del trabajo, tanto general como específicos.

Antecedentes

Se presentan con mayor detalle los temas a tratar, los conceptos referentes al problema planteado, los algoritmos existentes que pueden ayudar a resolver dicho problema, las nociones de patrones de software y arquitectura de la aplicación, el análisis de las herramientas a utilizar y los motivos para utilizar la herramienta elegida.

Análisis y Diseño

Se presentan los requisitos que se esperan que cumpla la aplicación, el planteamiento de la arquitectura de la aplicación, el diseño de los algoritmos utilizados se muestran conceptos de ingeniería de software a utilizar en el desarrollo de la aplicación y los diagramas UML de cada módulo.

Implementación

Se presenta una revisión del ambiente de desarrollo de la aplicación, utilidades adicionales usadas para el soporte del desarrollo, detalles sobre la implementación de la biblioteca de la aplicación, la interfaz de programación de la biblioteca (API) para su uso en otras aplicaciones y la implementación de la interfaz gráfica, con enfoque en las secciones más relevantes.

Mediciones y Análisis

Se presenta un ambiente de pruebas, la generación de los archivos de prueba, el tamaño de los archivos de prueba (peso de los archivos y cantidad de triángulos), los resultados de las mediciones de rendimiento para cada uno de los motores de procesamiento, y se exponen los resultados, tanto a nivel general como con los detalles más específicos de cada método usado.

Conclusiones y Trabajo Futuro

Se muestran las conclusiones obtenidas a partir del trabajo realizado, se revisa si se cumplieron los objetivos y se analiza el posible trabajo futuro que se puede llevar a cabo al extender esta aplicación.

Bibliografía

Se presentan las fuentes de información bibliográfica utilizadas tanto en el marco teórico (antecedentes) como en el diseño e implementación.

Anexos

Se presenta información adicional y de interés sobre el desarrollo de la memoria, como por ejemplo, la estructura de los archivos de mallas, detalles importantes dentro de la implementación, la forma de generar las mallas a usar y ejemplos de cómo extender la biblioteca.

Capítulo 2

Antecedentes

En este capítulo se presentan los temas que anteceden al desarrollo de este trabajo de título. Estos temas son requisitos básicos para poder comprender las razones y justificaciones del desarrollo de esta aplicación, junto con las ideas utilizadas y desarrolladas en el diseño del mismo.

Primero, se presenta la noción de triangulación y malla triangular en detalle, los distintos tipos de triangulaciones relevantes disponibles, y la explicación de algunos de los algoritmos existentes que permiten mejorar estas triangulaciones.

Luego, se exponen patrones de arquitectura de software aplicables, tales como MVC y ModelView, cómo éstos se pueden ajustar a la aplicación, y se explican los patrones de software que se incluyeron en la aplicación.

Finalmente, se analizan las herramientas disponibles para la construcción y desarrollo de la aplicación, junto con la explicación de por qué se eligieron las herramientas utilizadas.

2.1. Triangulaciones

Una malla triangular (triangulación) es una especialización de una malla de polígonos, la cual es capaz de definir la forma de un objeto dividiéndolo en un conjunto de muchos triángulos unidos por sus aristas, que forman así una superficie.

Existen varios tipos de triangulaciones que se pueden utilizar:

- Triangulación en abanico.
- Triangulación de peso mínimo.
- Triangulación de Delaunay.

2.1.1. Triangulación en abanico

La triangulación en abanico es una forma simple de triangular un polígono eligiendo un vértice y dibujando diagonales al resto de los vértices. No es posible triangular con este método con todos los polígonos, por lo que solo se usa para triangular polígonos convexos.

Aunque esta triangulación puede ayudar a resolver ciertos problemas de computación gráfica, como la rasterización o la detección de colisiones, puede no ser útil para otras tareas, porque el vértice origen acumula muchos vecinos, y los ángulos internos no están uniformemente distribuídos. [5]

2.1.2. Triangulación de peso mínimo

La triangulación de peso mínimo es aquella donde cada arista tiene un costo o "peso": Los pesos pueden ser arbitrarios, pero "la elección natural es la de usar la distancia euclideana entre los puntos".[6] Por lo tanto, el peso de la triangulación es la suma de los pesos de las aristas usadas, y la triangulación de peso mínimo es aquella con el menor peso posible.

El problema con la triangulación de peso mínimo es que es un problema *NP-hard* (probado con reducción PLANAR 1-IN-3-SAT) [12], por lo que para generar una triangulación de este tipo es necesario utilizar heurísticas, como por ejemplo, con algoritmos avaros [6] que solo pueden aproximarse a la verdadera triangulación.

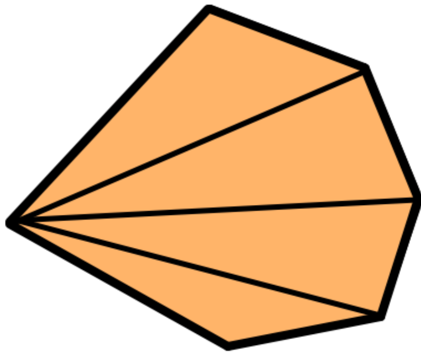

Figura 2.1: Triangulación en abanico.

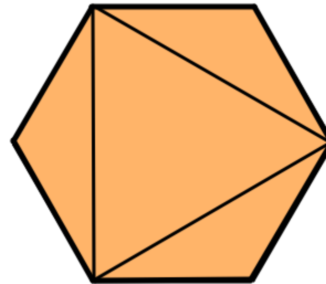

Figura 2.2: Triangulación de peso mínimo.

2.1.3. Triangulación de Delaunay

La triangulación de Delaunay es la triangulación que "minimiza el máximo circunradio, maximiza el mínimo circunradio y maximiza el ángulo mínimo de los triángulos que la componen." [6]

La triangulación de Delaunay es especialmente útil, porque al hacer que se mejore el ángulo mínimo de los triángulos se elimina en gran parte la imprecisión de los cálculos que se requieren hacer con números del tipo *float*, evitando así errores de arrastre.

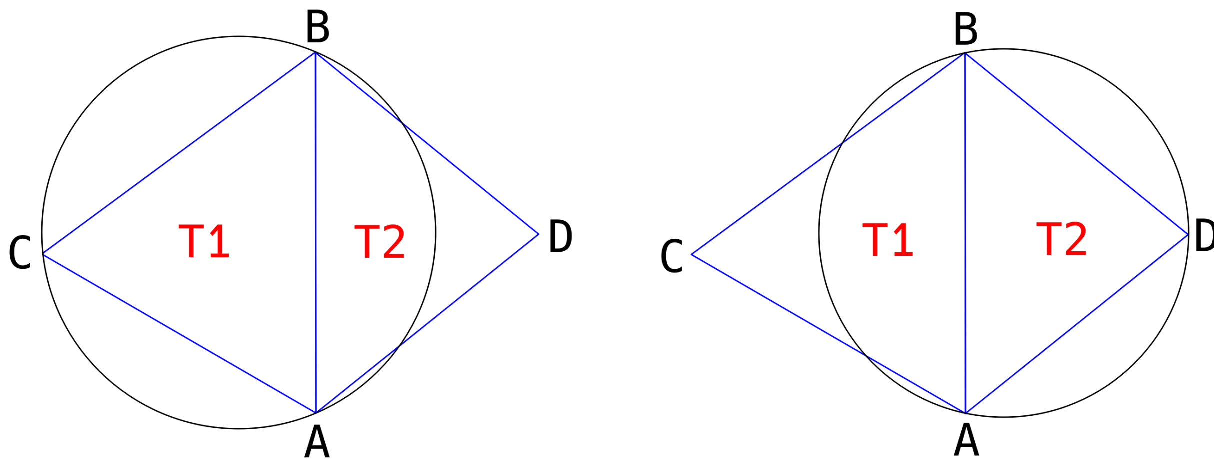

Figura 2.3: Triangulación de Delaunay.

Consideremos como ejemplo el caso de una triangulación compuesta por 2 triángulos. La triangulación de la Figura 2.3 está compuesta por los triángulos ABC y ADB. Al lado izquierdo se traza el circuncírculo del triángulo ABC, y se observa que el punto D se encuentra en el exterior del circuncírculo. Análogamente, a la derecha se traza el circuncírculo del triángulo ADB, y se observa que el punto C también se encuentra en el exterior del circuncírculo.
Como en la triangulación completa no existen puntos dentro de los circuncírculos de cada triángulo, se considera que es una triangulación de Delaunay.

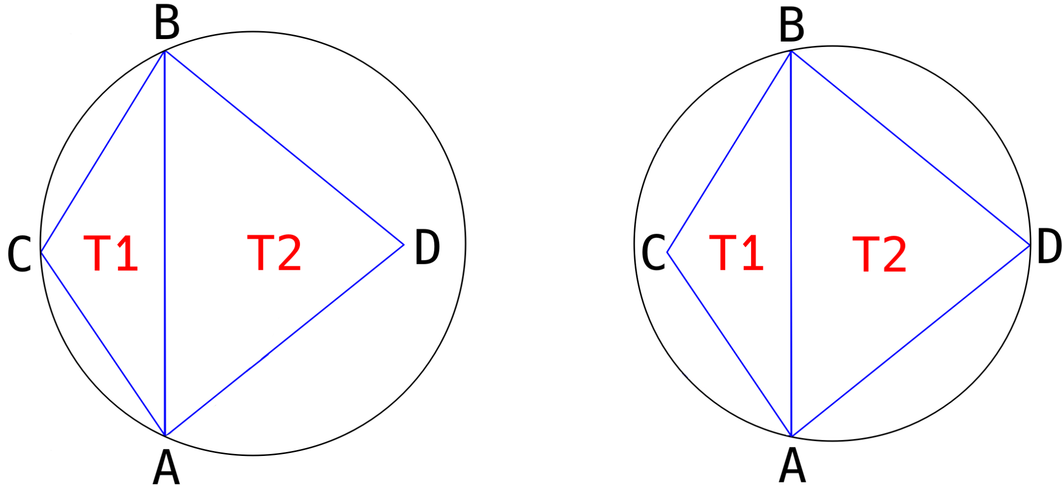

Figura 2.4: Triangulación no-Delaunay.

Por otro lado, al lado izquierdo de la Figura 2.4, al trazar el circuncírculo del triángulo ABC se observa que el punto D sí está al interior, y al lado derecho, al trazar el circuncírculo del triángulo ADB también se observa que el punto C se encuentra al interior. Por lo tanto, esta triangulación no es de Delaunay.

2.2. Mejoramiento de una triangulación

Cuando se tiene una malla triangular cualquiera, normalmente se espera trabajar haciendo cálculos sobre ella. Pero esta malla inicial puede no ser lo suficientemente adecuada para una simulación numérica, puesto que existe la posibilidad de acumular pequeños errores de precisión que en conjunto entregarán resultados erróneos. Esto lleva a buscar formas de mejorar esta triangulación.

Es posible saber que una malla es "mala" cuando en los triángulos que la componen existen *slivers*[1] que hacen difícil el cálculo correcto en esta triangulación por errores de precisión.

Para poder mejorar una triangulación, normalmente se busca que ésta se convierta en una triangulación de Delaunay, por lo que cada triángulo que compone esta triangulación debe tener un ángulo mínimo lo suficientemente grande como para cumplir con la métrica de calidad de Delaunay, la cual es lograr que ningún vértice que no sea parte del triángulo esté dentro del circuncírculo del mismo.

[1]Un triángulo *sliver* es aquel triángulo que posee un área muy pequeña debido a que sus ángulos son muy cercanos a cero.[16]

Para ello existen varios acercamientos. Por ejemplo, se puede convertir una malla cualquiera en una triangulación de Delaunay cambiando los triángulos que la componen y manteniendo sus vértices, o agregando nuevos vértices para crear triángulos en posiciones estratégicas que ayuden a mejorar la malla.

Los algoritmos de refinamiento consisten en dividir un triángulo en triángulos más pequeños hasta, por ejemplo, que todos los lados tengan un largo menor a un cierto valor. Los algoritmos de mejoramiento insertan nuevos puntos para crear triángulos, o simplemente mueven los triángulos existentes hasta que la métrica usada verifique que la triangulación tiene suficiente "calidad".

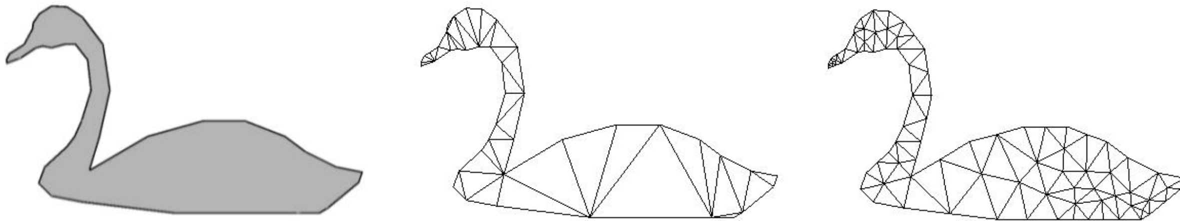

Figura 2.5: Geometría, malla de Delaunay y refinamiento

En la Figura 2.5 se puede apreciar que a partir de una geometría dada se puede generar una triangulación, y que ésta se puede refinar respecto a un ángulo mínimo requerido.

En las siguientes subsecciones se mostrarán y explicarán distintos algoritmos que pueden ser usados para mejorar una triangulación.

2.2.1. Algoritmo de *edge-flipping*

El algoritmo de *edge-flipping*, originalmente propuesto por Lawson [10] es un algoritmo que toma una triangulación cualquiera y la transforma a una triangulación de Delaunay, tomando dos triángulos vecinos e intercambiando la posición de su arista común; esto se repite para todos los triángulos que requieran un *flip*, i.e., que no cumplen con la Condición de Delaunay[2].

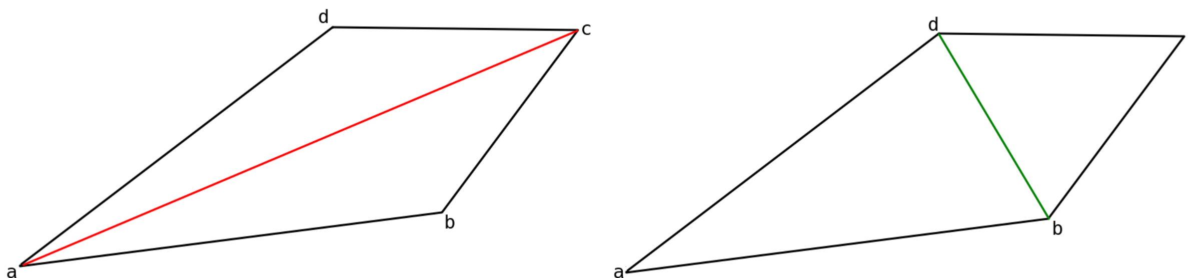

Figura 2.6: *Flip* de una arista

Como se puede ver en la Figura 2.6, el triángulo ABC y el triángulo ACD comparten la arista AC. Al hacer un *flip*, la arista AC se convierte en la arista BD, convirtiendo los triángulos anteriores en los nuevos triángulos ABD y BCD.

Hay que considerar que este algoritmo permite mejorar una triangulación, pero no agrega nuevos triángulos a ésta.

2.2.2. Algoritmos de tipo Lepp

El algoritmo Lepp (*Longest-Edge Propagation Path*) es un algoritmo que consiste en detectar triángulos de mala calidad[3], recorrerlos creando una lista de triángulos accesibles a través de su arista más larga, hasta llegar a la *arista terminal* [4] y modificar el o los últimos triángulos que comparten esta arista. Este proceso se repite hasta lograr modificar el triángulo inicial.

Supondremos que el triángulo ABC de la Figura 2.7 es de mala calidad. Para crear la lista Lepp, se busca la arista más larga del triángulo "malo", la cual para el ejemplo es la arista BC. A partir de esta arista, detectaremos al vecino de ABC, el cual es el triángulo BDC. Buscaremos su arista más larga, la cual es la arista BC, y volveremos a buscar a su vecino, y así sucesivamente.

[2] La suma de los ángulos opuestos a la arista es menor o igual a 180°.

[3] Estos son los triángulos que no cumplen la Condición de Delaunay por ser demasiado agudos u obtusos.

[4] Última arista más larga, dado que el resto de las aristas son más pequeñas.

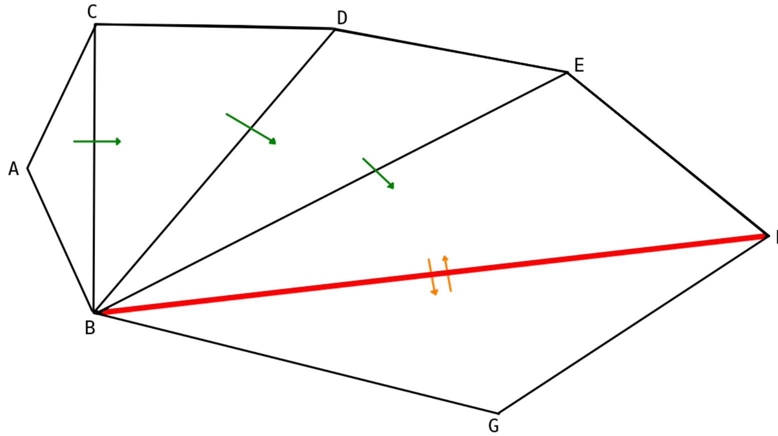

Figura 2.7: Lista Lepp.

Este algoritmo termina de crear la lista cuando se alcanza la arista BF, porque el vecino de arista más larga del triángulo BFE es el triángulo BGF, y el vecino de arista más larga del triángulo BGF vuelve a ser el triángulo BFE. Esto significa que para estos dos triángulos vecinos, su arista más larga es la misma, lo que significa que esta será la arista a modificar.

Lepp-Bisección

Para modificar la arista terminal, el algoritmo Lepp-Bisección[14], divide esta arista en su punto medio, creando 4 triángulos donde antes habían dos.

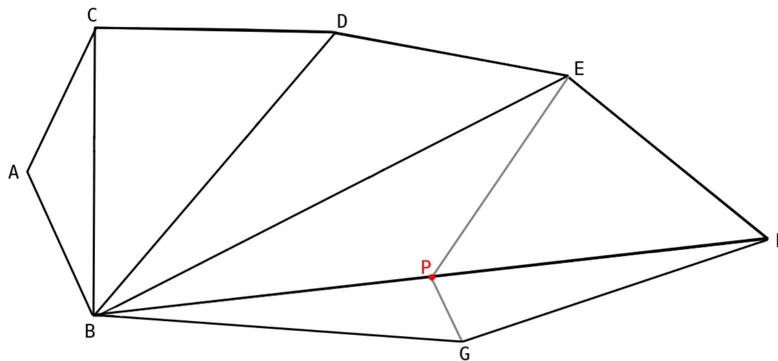

Figura 2.8: Lepp-Bisección.

En la Figura 2.8, la arista terminal BF se divide en su punto medio, esto es, sumando las coordenadas de los puntos B y F y dividiéndolas por 2, para conseguir el punto P. A partir de los puntos B, G, F, E y P, se crean los 4 triángulos BGP, GFP, FEP y EBP, consiguiendo un refinamiento de la triangulación.

Lepp-Centroide

El algoritmo Lepp-Centroide sigue la misma idea base de la lista Lepp, pero introduce una mejora al momento de alterar la arista terminal: En lugar de dividir la arista por su punto medio, la divide buscando el centroide de los puntos que componen a los dos triángulos que están compartiendo dicha arista.

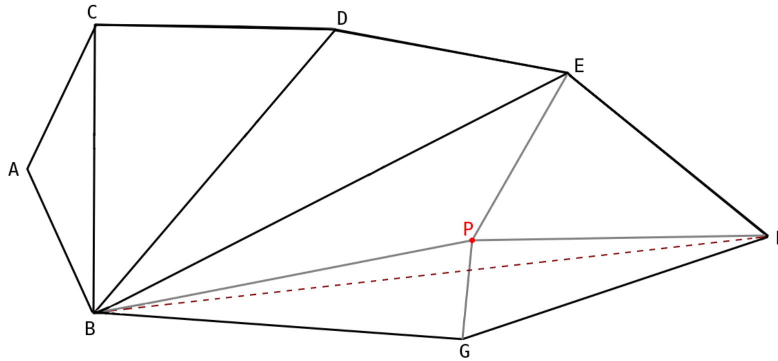

Figura 2.9: Lepp-Centroide.

En la Figura 2.9, la arista terminal BF (representada por la línea entrecortada) se elimina completamente, quedando solo los puntos B, G, F y E. A partir de las coordenadas de estos puntos se crea un nuevo punto P, el centroide del cuadrilátero BGFE. Entonces, con estos 5 puntos se crean los 4 triángulos BGP, GFP, FEP y EBP, consiguiendo un refinamiento de la triangulación mejor que usando Lepp-Bisección, puesto que los ángulos mínimos de los triángulos cambian gracias al uso del centroide.

2.3. Patrones de Arquitectura de Software

2.3.1. MVC

MVC (Model-View-Controller) es un patrón de arquitectura de software que consiste en 3 objetos: El Modelo, que es el objeto de la aplicación, la Vista, que es su presentación, y el Controlador, que define la forma en la que la interfaz de usuario reacciona al input del mismo. [9]

MVC desacopla las vistas de los modelos al establecer un protocolo de "suscripción" entre ellos. La Vista debe asegurarse que su apariencia refleja el estado del Modelo. Cuando el Modelo cambia, debe notificarle a la(s) vista(s) que depende(n) de él, lo que permite que las vistas se actualicen a sí mismas. De esta forma, se pueden asociar múltiples vistas a un modelo sin tener que reescribirlo. [9]

2.3.2. ModelView

ModelView es un patrón de arquitectura de software que consiste en 2 objetos: El Modelo, que es el objeto de la aplicación, y la Vista que es tanto su presentación como la forma en la que interfaz de usuario reacciona al input del mismo.

Como se vio en la sección anterior, se puede separar una aplicación en su Modelo, Vista y Controlador. Si la Vista y el Controlador se combinan, se obtiene como resultado la arquitectura ModelView. Esta arquitectura aún separa la forma de manejar los datos respecto a la forma de presentarlos al usuario, pero provee un *framework* más simple que se basa en los mismos principios. Esta separación hace posible mostrar los mismos datos en muchos tipos de vistas distintos, e implementar nuevas vistas sin tener que cambiar la estructura de datos interna. [11]

2.4. Patrones de Diseño de Software

Para poder presentar los patrones de software más importantes, es necesario primero definir qué es un patrón. Un patrón "describe un problema que ocurre a menudo en nuestro ambiente, y describe el núcleo de la solución al problema, de forma de poder usar esta solución un millón de veces [...]" [4].

Para el caso aplicado al diseño de software, un patrón de diseño "nombra, abstrae, e identifica los aspectos claves de una estructura común de diseño, que la hace útil para crear un diseño reutilizable orientado a objetos. El patrón de diseño identifica la participación de clases e instancias, sus roles y colaboraciones, y la distribución de sus responsabilidades. [...] Describe cuándo es aplicable y que las consecuencias, ventajas y desventajas de su uso."[9]

Dentro de los patrones más relevantes para este contexto se encuentran: Factory Pattern, Singleton Pattern, Strategy Pattern, Façade Pattern y Bridge Pattern.

2.4.1. Factory Pattern

El patrón de diseño "Factory" permite crear objetos sin tener que especificar la clase exacta del objeto que se requiere crear, evitando duplicación de código al momento de inicializar un objeto. Además, facilita el trabajo del desarrollador al ofrecer una interfaz simple para crear objetos.

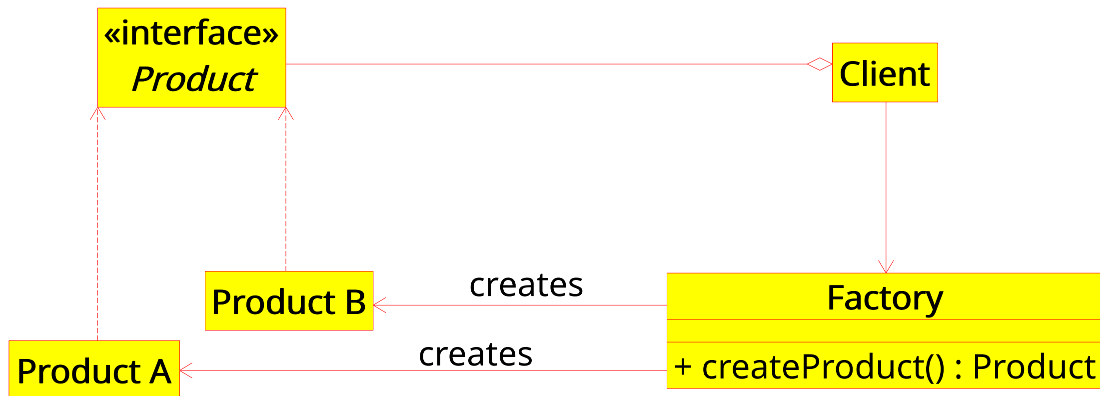

Figura 2.10: UML del patrón de diseño Factory.

2.4.2. Singleton Pattern

El patrón de diseño "Singleton" asegura que solamente exista una instancia de una clase y provee un acceso global hacia esta instancia. Se utiliza para evitar que otras clases generen instancias de la clase Singleton que sean distintas entre sí.

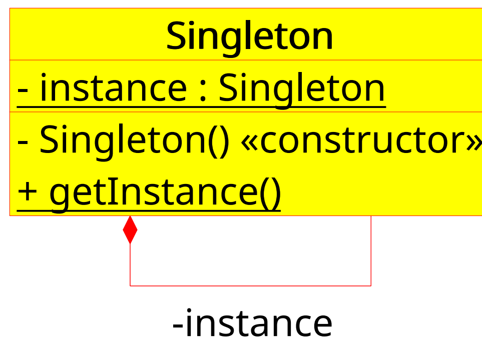

Figura 2.11: UML del patrón de diseño Singleton.

2.4.3. Strategy Pattern

El patrón de diseño "Strategy" permite definir una familia de algoritmos, encapsular cada uno de ellos y permitir elegir entre éstos en tiempo de ejecución, logrando así ejecutar variantes de un mismo algoritmo.

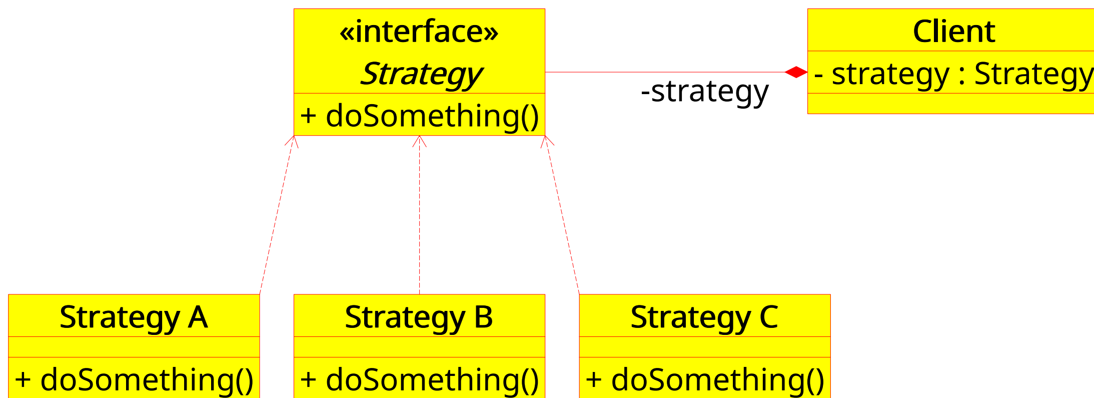

Figura 2.12: UML del patrón de diseño Strategy.

2.4.4. Façade Pattern

El patrón de diseño Façade permite proveer una interfaz simplificada a una gran cantidad de clases, como lo es una biblioteca. Se utliza para reducir la complejidad al trabajar con subsistemas, al minimizar las comunicaciones y dependencias entre éstos.

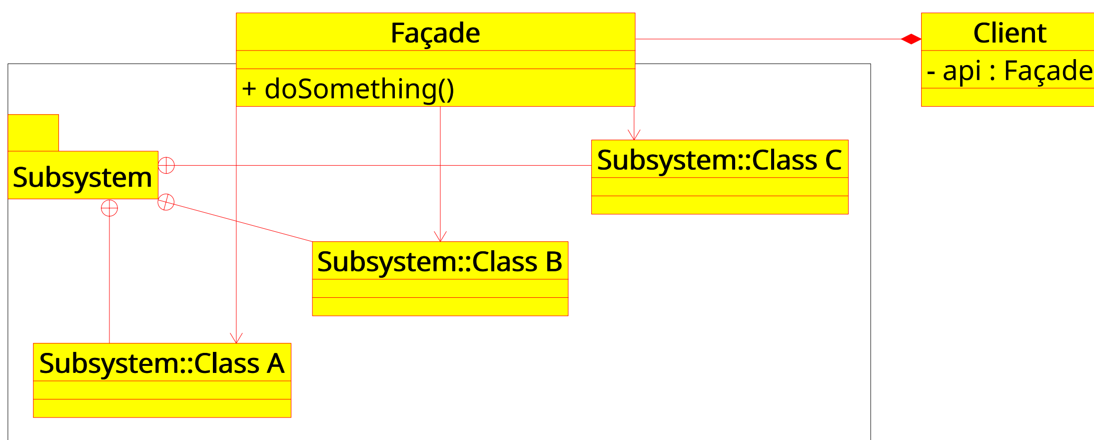

Figura 2.13: UML del patrón de diseño Façade.

2.4.5. Bridge Pattern

El patrón de diseño Bridge (también conocido como "puntero opaco" [5]) permite separar una abstracción de una implementación, de forma que ambas puedan variar de forma independiente. En el caso de C++ se lo conoce como *PIMPL* (Pointer-to-implementation), y permite acelerar el proceso de compilación, al separar en distintos archivos cada una de las partes involucradas.

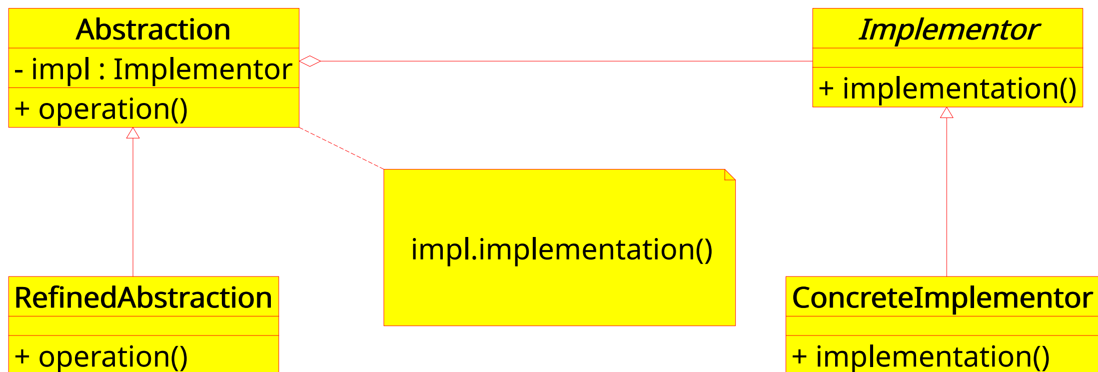

Figura 2.14: UML del patrón de diseño Bridge

[5] https://en.wikipedia.org/wiki/Opaque_pointer

2.5. Herramientas disponibles

En esta sección se presentan las herramientas disponibles para desarrollar la aplicación, tales como bibliotecas de cómputo en GPU y bibliotecas gráficas.

Para poder trabajar con GPU existen dos alternativas: CUDA y OpenCL, y para poder mostrar los resultados en la pantalla existen varias bibliotecas, siendo Gtk+ y Qt las dos más importantes en el mundo de las interfaces.

CUDA [6] es un modelo de programación y plataforma de computación paralela creado y desarrollado por NVIDIA que permite hacer computación general [15] usando una tarjeta gráfica, lo que permite incrementar la velocidad de procesamiento de una aplicación.

OpenCL [7] es un estándar abierto multiplataforma de programación paralela usado en computadores personales, servidores, dispositivos móviles y plataformas embebidas, el cual incrementa en gran manera la velocidad y responsividad de una gran cantidad de aplicaciones de todo tipo.

Gtk+ [8] (*The GIMP Toolkit*), es un *toolkit* multiplataforma escrito en C, hecho para crear interfaces gráficas. Ofrece un conjunto de widgets que permite trabajar tanto en proyectos pequeños como en suites completas.

Qt [9] es un *framework* multiplataforma escrito en C++, usado para desarrollar software de aplicaciones que pueden correr en distintas plataformas con poco o cero cambios en la base, y aún así tener capacidades nativas y velocidad nativa.

2.5.1. CUDA vs OpenCL

Nvidia es la empresa propietaria de CUDA, por lo que solamente esta empresa puede ofrecer esta implementación. Por otro lado, existen las siguientes empresas que ofrecen implementaciones de OpenCL:

- AMD: Ofrece hasta OpenCL 2.0.
- Nvidia: Ofrece hasta OpenCL 1.2.
- Intel: Ofrece hasta OpenCL 2.0.

[6] `https://developer.nvidia.com/cuda-zone`
[7] `https://www.khronos.org/opencl/`
[8] `https://www.gtk.org/`
[9] `https://www.qt.io/`

Ambas herramientas poseen interoperabilidad con OpenGL y DirectX [3], lo que corresponde a una característica bastante relevante. Por otro lado, para poder comparar las demás características, se presentan sus pros y contras en la siguiente lista.

Portabilidad

CUDA solo funciona con tarjetas Nvidia, mientras que OpenCL puede funcionar con una gran gama de dispositivos, sacrificando velocidad a cambio de portabilidad. [1]

Capacidades

- Tanto CUDA como OpenCL soportan mapeo de memoria al *host*.
- Las características de sincronización de CUDA no son tan flexibles como en OpenCL. Por ejemplo, en CUDA no es posible encolar instrucciones a la GPU en desorden.
- CUDA tiene un *profiler* nativo (*nvprof* [10]), mientras que OpenCL depende de que el implementador haya hecho herramientas de *profiling* (por ejemplo, Intel® SDK for OpenCL™ Applications [11]).

Flexibilidad

- Para un programador de C, la *runtime API* de CUDA es más fácil de usar que la *runtime API* de OpenCL.
- Para un programador de C++, OpenCL provee *bindings* de C++, lo que lo hace más fácil para orientación a objetos.

Desempeño

Dado que CUDA está desarrollado por y para Nvidia, es ligeramente más rápida que OpenCL.

Licenciamiento

CUDA es una biblioteca propietaria, mientras que OpenCL es un estándar *open-source*.

[10] http://docs.nvidia.com/cuda/profiler-users-guide/index.html
[11] https://software.intel.com/en-us/intel-opencl

2.5.2. Gtk+ vs Qt

Tipo de biblioteca

Gtk+ es una *toolkit* que permite crear interfaces gráficas, mientras que Qt es un *framework* para aplicaciones que contiene un *toolkit* de interfaz gráfica como parte de sus componentes.

Retro-compatibilidad

Qt permite retrocompatibilidad entre versiones menores (Ej., 5.8 y 5.9). Por otro lado, Gtk+ ha recibido muchas críticas respecto a API y ABI inestables entre versiones menores (Ej., 3.12 y 3.14). [18]

Entornos de desarrollo

- Gtk+ no posee un IDE propio, pero posee un editor gráfico de interfaces llamado Glade [12] que permite diseñar la interfaz de una aplicación sin necesidad de programarla.
- Qt posee un IDE propio llamado Qt Creator [13] que permite crear desde cero una aplicación de escritorio, y trae integrado un editor gráfico de interfaces llamado Qt Designer [14] que cumple la misma funcionalidad que Glade.

Paradigma de programación

Gtk+ está desarrollado en C, usando GObject para poder conseguir orientación a objetos. Qt está desarrollado en C++, permitiendo naturalmente orientación a objetos.

Lenguajes soportados

Gtk+ soporta una mayor cantidad de lenguajes, tales como C, C++, D, Genie, Go, Haskell, Java, Julia, JavaScript, Lua, Pascal, Perl, Prolog, Python, Ruby, Rust y Vala. [15].

Qt soporta los lenguajes C++, C#, D, Crystal, Go, Haskell, Java, JavaScript, Julia, Pascal, Python, OCaml, QML, Ring, Ruby y Rust. [16]

[12] `https://glade.gnome.org`
[13] `https://wiki.qt.io/Qt_Creator`
[14] `https://wiki.qt.io/Tools#Qt_Designer`
[15] `https://en.wikipedia.org/wiki/List_of_language_bindings_for_GTK+`
[16] `https://en.wikipedia.org/wiki/List_of_language_bindings_for_Qt_5`

Licenciamiento

Gtk+ usa la licencia GNU LGPL 2.1, que permite desarrollo tanto público como privado, mientras que Qt posee un sistema de licenciamiento dual de código abierto (GPL 3.0, LGPL 2.1 y LGPL 3.0) y de código privativo (Qt Commercial License).

2.6. Opciones elegidas

2.6.1. Biblioteca de cómputo

Se decidió utilizar la biblioteca de OpenCL por dos motivos principales: Portabilidad del código y *bindings* nativos. Esto es, el código creado con OpenCL se puede correr en cualquier dispositivo que tenga una implementación de OpenCL, lo que es una gran ventaja respecto al tener que estar forzado a usar solamente tarjetas Nvidia, en caso de haber elegido CUDA.

Por otro lado, OpenCL tiene *bindings* nativos para C++ sin necesidad de usar otras bibliotecas complementarias, mientras que CUDA posee una API de C, por lo que se requiere usar bibliotecas extras, o aceptar que se debe programar en dos lenguajes.

2.6.2. Biblioteca gráfica

Se decidió utilizar la biblioteca de Qt, también por dos motivos: Conveniencia de tener un *framework* completo de trabajo bajo una sola API y *bindings* nativos de C++.

El *framework* de Qt permite no solamente generar una buena interfaz gráfica, sino que además facilita notablemente el desarrollo del modelo de la aplicación, usando APIs intuitivas de lectura y escritura de archivos, y de modularización de componentes. Además, el IDE que trae (Qt Creator) es más fácil de usar, dada la documentación integrada y el editor gráfico de interfaces que posee.

Sumado a esto está el hecho de que Qt está nativamente hecho en C++, lo que permite trabajar con orientación a objetos sin tener que usar *hacks*, como GObject de Gtk+.

Finalmente, Qt tiene *bindings* hacia OpenGL, que ofrecen una capa de abstracción manejable, mientras que GTK+ depende de las herramientas nativas de OpenGL para mostrarse en pantalla.

Capítulo 3

Análisis y Diseño

En este capítulo se mostrarán los requisitos planteados e implementados en la aplicación, junto con el diseño modularizado de los componentes, las estructuras de datos que fueron usadas y el diseño de las soluciones propuestas.

3.1. Requisitos

3.1.1. Requisitos de Usuario

Interfaz simple

La aplicación debe ser suficientemente simple e intuitiva de usar.

Visualización

La aplicación debe poder mostrar la malla de triángulos al momento de cargarla desde memoria secundaria.

Zoom

La aplicación debe permitir acercar y alejar la vista de la malla.

Traslación

La aplicación debe permitir trasladar la malla visualmente.

Rotación

La aplicación debe permitir rotar la malla, o en su defecto, moverla en más de un eje a fin de simular rotación.

Restaurar vista

La aplicación debe permitir restaurar la vista en caso de mover la malla.

Ayuda

La aplicación debe proveer de instrucciones de uso suficientemente claras.

Traducciones

La aplicación debe permitir su utilización en sistemas que tengan un idioma distinto al inglés y aplicar automáticamente su traducción.

Keybindings (Opcional) Las funcionalidades de la aplicación deben funcionar tanto con el mouse (menús y *toolbars*) como con el teclado (accesos directos de teclado).

Drag-and-Drop (Opcional) La aplicación debe permitir cargar archivos arrastrándolos hacia ésta, y filtrar archivos que no correspondan a los tipos que la aplicación puede manejar.

Historial (Opcional) La aplicación debe permitir guardar un historial de archivos recientemente cargados. Además, debe permitir limpiar dicho historial.

3.1.2. Requisitos de Software

Uso de GPU

La aplicación debe ser capaz de utilizar GPU en la ejecución del algoritmo de mejoramiento de mallas triangulares.

Uso de CPU

La aplicación debe proveer una implementación CPU a modo de *fallback*, o para ser comparada con la implementación GPU.

API

La aplicación debe proveer una API en case de usarse sin interfaz gráfica.

Carga de archivos

La aplicación debe poder leer archivos OFF desde memoria secundaria y cargar la malla.

Guardado de archivos

La aplicación debe poder guardar archivos OFF a memoria secundaria respecto a la malla, tanto al momento de cargarla como de mejorarla.

3.2. Arquitectura

Se decidió que la aplicación debía separarse en modelo y vista, conectarlos fácilmente, además de permitir que cualquier desarrollador pudiera crear su propia vista sin tener que desarrollar la aplicación desde cero, por lo que se separó la aplicación en dos componentes: Una biblioteca y una interfaz gráfica. De esta forma, se hace posible usar los recursos que provee la aplicación usando la interfaz de programación de aplicación (API) propuesta para la biblioteca, la cual es demostrada al ser usada por la interfaz.

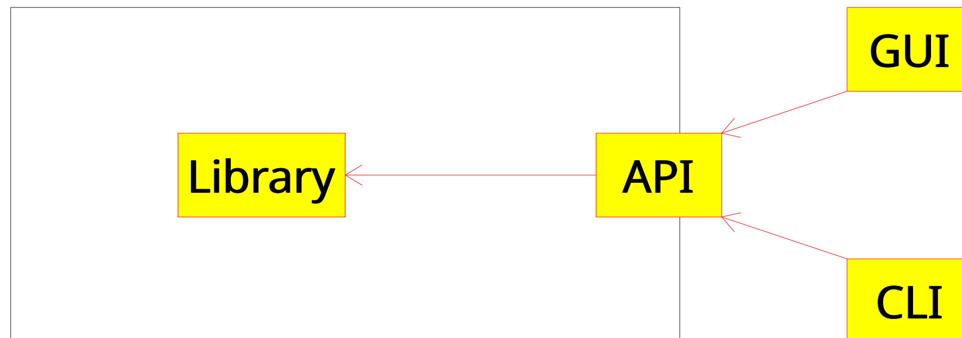

Figura 3.1: UML básico de las capacidades de la aplicación.

Como se puede ver en la Figura 3.1, la arquitectura ModelView expuesta en los Antecedentes se acopla perfectamente a las necesidades expuestas en el párrafo anterior, y dada esta arquitectura, se separará la descripción de la aplicación en Biblioteca e Interfaz.

3.3. Algoritmos

El algoritmo Lepp-Centroide base tiene la siguiente estructura:

```
T = Triangulación
S = Conjunto de triángulos de mala calidad de T

for t in S do
    Encontrar el Lepp asociado a t
    Tomar los triángulos vecinos a la arista terminal
    Inserción (simple) del centroide
    Recalcular S
end for
```

Se diseñó una variante a partir de la estructura general de manera que pudiera funcionar tanto en CPU como en GPU, basándose en la idea de:

- Detectar los triángulos de mala calidad.
- Mejorar la triangulación respecto a estos triángulos:
    - Detectar las aristas terminales de todos los triángulos de mala calidad.
    - Insertar los centroides en cada par de triángulos vecinos que fueran parte de las aristas terminales.
    - Recalcular (volver a detectar) triángulos de mala calidad.

Para el caso de este desarrollo particular, solo se consideró mejorar la triangulación respecto a las aristas terminales que no fuesen de borde, a fin de evitar problemas en la GPU al momento de detectar una arista terminal que además fuera de borde. En este caso, puesto que el centroide sería el punto medio de la arista terminal, subdividir el triángulo implicaría realizar una bisección, lo que crea la necesidad de crear un método particular de inserción para este caso. Se decidió dejar fuera este caso para concentrarse solamente en el algoritmo Lepp-Centroide.

Algoritmo en CPU

```
// Este diseño está descrito para 1 ejecución
T = Triangulación
E = Lista de aristas de T

// Detección de triángulos de mala calidad
for t in T do
    if t no cumple con ángulo mínimo
        Marcar como triángulo malo
    end if
end for

// Detección de aristas terminales
for t in T do
    if t es triángulo malo
        Encontrar el Lepp asociado a t, marcando las aristas terminales
    end if
end for

// Inserción de centroide
for e in E do
    if e es arista terminal y no es arista de borde
        Insertar centroide en los triángulos compartidos
    end if
end for
```

```
Recalcular todos los triángulos de mala calidad en T
```

Algoritmo en GPU

```
// Este diseño está descrito para 1 ejecución
T = Triangulación
E = Lista de aristas de T

// Detección de triángulos de mala calidad - GPU
paralell for t in T do
    if t no cumple con ángulo mínimo
        Marcar como triángulo malo
    end if
end for

// Detección de aristas terminales - GPU
paralell for t in T do
    if t es triángulo malo
        Encontrar el Lepp asociado a t, marcando las aristas terminales
    end if
end paralell for

// Inserción de centroides - CPU (Secuencial)
for e in E do
    if e es arista terminal y no es arista de borde
        Insertar centroide en los triángulos compartidos
    end if
end for

paralell Recalcular todos los triángulos de mala calidad en T
```

Como se puede observar en el pseudocódigo (ver comentarios), el algoritmo en GPU tiene un *approach* mixto, donde se detectan los triángulos de mala calidad y se encuentran sus aristas terminales usando la GPU[1], pero se insertan los centroides en CPU[2], dado que no es nativamente posible crear elementos en GPU que puedan ser devueltos a la CPU, porque para el caso de la inserción, se convierten los dos triángulos de la arista compartida en cuatro nuevos triángulos con una inserción simple del centroide.

[1]Secciones `paralell for` del pseudocódigo.
[2]Sección `for` del pseudocódigo.

En base al diseño realizado y a las restricciones impuestas para el algoritmo, la ejecución se repite hasta que no queden triángulos de mala calidad, o hasta que solo queden aristas terminales de borde.

3.4. Biblioteca

El diseño de la biblioteca fue pensado con énfasis en la simplicidad de uso por otros programas, y priorizando su facilidad para extenderla. Para ello, se comenzó a programar desde la API que se espera que ofrezca la biblioteca, donde internamente están los módulos que permiten su funcionamiento.

Además, gracias al polimorfismo que permite la orientación a objetos, es posible cargar dinámicamente las clases necesarias para utilizar distintos motores de procesamiento y distintas clases que se encarguen de la carga y guardado de archivos de distinta extensión.

Finalmente, utilizando herencia y aplicando patrones de diseño de software convenientes para cada módulo, es posible extender la aplicación para poder agregar más motores de procesamiento, o arreglar partes de cada motor simplemente extendiendo la clase original, y también es posible extender las clases que se encargan de cargar y guardar archivos para poder incrementar los tipos de archivos manejados por la biblioteca.

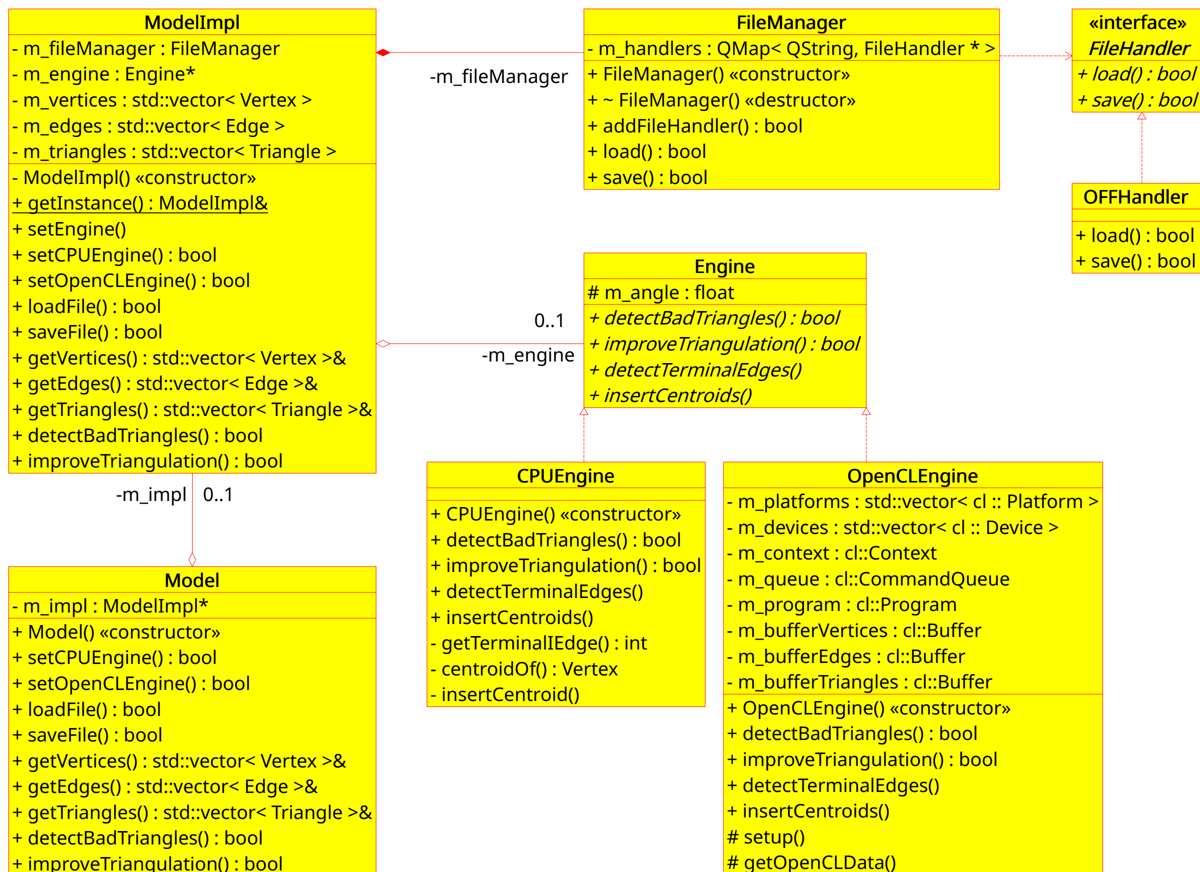

Figura 3.2: UML simplificado de la biblioteca.

Como se puede ver en la Figura 3.2, que muestra una simplificación del UML real de la biblioteca (en realidad la mayoría de los métodos sí reciben argumentos), se ha diseñado la biblioteca de tal forma que sea tan fácil usarla como extenderla[3]. Las clases que la componen serán explicadas en las siguientes secciones.

3.4.1. Diseño de las Estructuras

Antes de iniciar la explicación del modelo, es necesario mostrar cuáles son las estructuras base utilizadas y su semántica en la aplicación. Estas estructuras fueron pensadas para darle flexibilidad y simplicidad a los algoritmos que la ocupan.

- **Vertex**: Estructura que define a un vértice. Posee coordenadas x, y, z.
- **Edge**: Estructura que define a una arista. Posee punteros (índices) a los dos triángulos que describe, índices a los dos vértices que lo componen, y un flag *isTE* que marca si esta arista es terminal o no.
- **Triangle**: Estructura que define a un triángulo. Posee índices a los vértices que lo componen, índices a las aristas que lo componen (donde la arista a se corresponde con el vértice a opuesto a ésta) y un flag *bad* que marca si el triángulo no cumple con la calidad que se busca.

3.4.2. Diseño del Modelo

Dado que el modelo debe ser capaz de exponer una API para ser usada, se decidió aplicar aquí el patrón de diseño de software Façade, con lo que además de simplificar la abstracción de la biblioteca permitió crear la API directamente.

Por otro lado, dado que la biblioteca necesita tener acceso garantizado a la GPU sin tener que competir entre instancias de ella misma, se decidió que el modelo usaría el patrón de diseño de software Singleton, por lo que el modelo siempre es el mismo.

Finalmente, para poder ocultar los detalles de implementación y acelerar el proceso de compilado de la aplicación, se utilizó el patrón de diseño de software Bridge, que permitió desacoplar la API de la implementación del façade y separar los headers públicos de los headers privados, utilizando el concepto de *pointer-to-implementation*[4].

[3] Para más información, ver Anexo D.

[4] `https://en.wikipedia.org/wiki/Opaque_pointer`

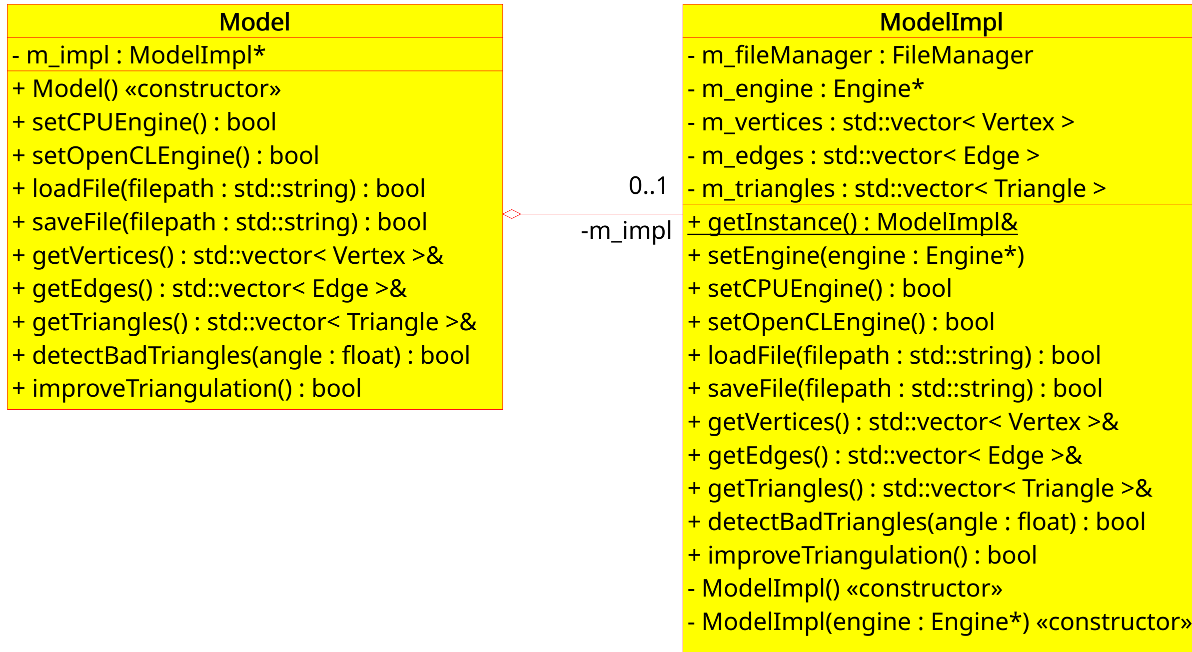

Figura 3.3: UML de la API.

Como se puede ver en el UML de la Figura 3.3, en la clase Model se define la API pública que expone la biblioteca al resto de las aplicaciones que la usen. Normalmente podría haberse expuesto ModelImpl, pues ésta es la clase façade original, pero dado que todavía hay detalles de implementación en los atributos de ModelImpl, se usó el concepto de *pointer-to-implementation* (aplicando el patrón de diseño de software Bridge), con el cual la API solo expone un puntero a la implementación real del façade de la biblioteca.

Como se puede observar en la clase ModelImpl, se maneja el gestor de archivos (FileManager), el motor de procesamiento (Engine) y se almacenan los datos correspondientes a las triangulaciones.

Finalmente, se puede observar que en la API todos los métodos (exceptuando los métodos *getters*) retornan un *bool*. Este booleano se puede usar en la interfaz para avisar si ese método pudo ser ejecutado correctamente.

3.4.3. Módulo de Carga y Guardado de archivos

Para poder manejar los archivos que tiene que leer la biblioteca y permitir la extensibilidad de esta funcionalidad, se utilizó el patrón de diseño de software Strategy, que a partir de una interfaz mínima con métodos *load()* y *save()*, pueda cargar en memoria principal el contenido de un archivo de mallas, donde cada clase encargada de un formato específico

pueda sobreescribir estos métodos convenientemente. Este patrón fue utilizado en la interfaz *FileHandler* y en la implementación *OFFHandler*.

Adicionalmente, se utilizó el patrón de diseño de software Factory en el gestor de archivos (*FileManager*), permitiendo agregar aquí nuevas clases que pudieran manejar archivos, sin que el cliente (que está usando *FileManager*) tenga que preocuparse de la extensión del archivo para crear una nueva instancia del *handler* adecuado para cada carga o guardado.

De esta forma, para los *handlers* de archivos, el método *load()* debe ser capaz de crear los objetos en memoria descritos en el archivo, y el método *save()* debe ser capaz de rescatar el estado actual de los objetos en memoria y escribirlos a un archivo.

Como se observa en la Figura 3.4, *FileManager* sabe cuál es el *FileHandler* a usar respecto a la extensión del archivo cargado (e.g.: mesh**.off**), por lo que es responsabilidad del *handler* responder ante el mensaje de cargar/ guardar un archivo.

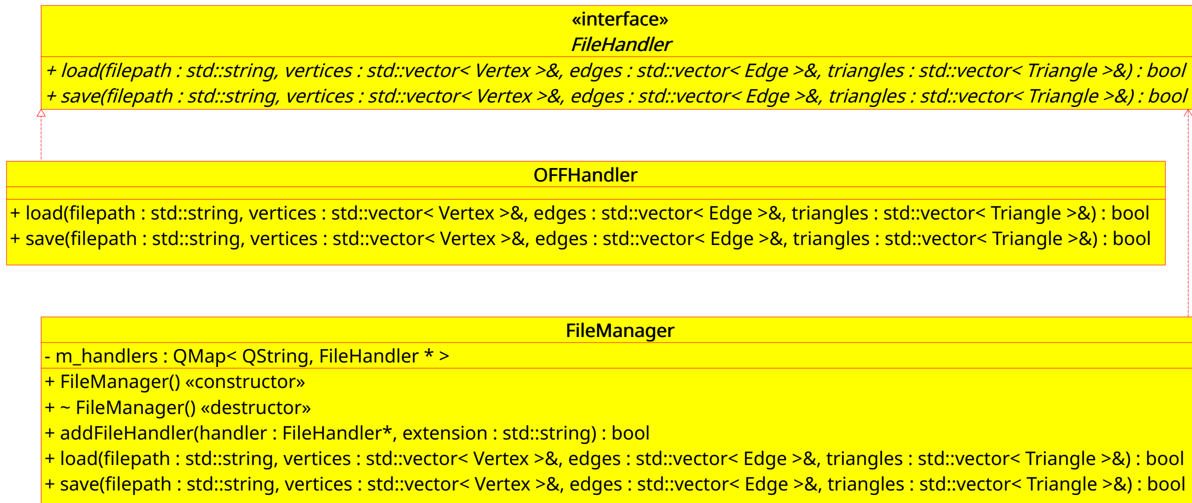

Figura 3.4: UML del módulo de carga y guardado de archivos.

3.4.4. Módulo de Procesamiento

De manera similar al diseño del módulo de cargado y guardado de archivos, se buscó que fuera posible integrar un motor de procesamiento a la biblioteca de forma modular, y que fuera posible extender esta funcionalidad para poder crear y ejecutar nuevos motores fácilmente, por lo que también se utilizó el patrón de diseño de software Strategy, que a partir de una interfaz que contiene los métodos principales *detectBadTriangles()* y *improveTriangulation()*, y los métodos secundarios *detectTerminalEdges()* y *insertCentroids()* se pudieran crear varios motores de procesamiento que los implementaran.

En este módulo se implementan las clases que realmente hacen el mejoramiento de la malla usando el algoritmo Lepp-Centroide, adaptado para que utilice tanto la CPU (*CPUEngine*) como la GPU *OpenCLEngine*. Como el patrón de diseño de software utilizado permite crear fácilmente un nuevo motor extendiendo la clase abstracta *Engine*, es posible integrar no solo una variante CUDA (por ejemplo), sino que también se pueden agregar distintos algoritmos, simplemente al *overridear* los dos métodos principales expuestos en el párrafo anterior.

Adicionalmente, este módulo se diseñó de tal forma que permitiera el intercambio dinámico entre los motores en tiempo de ejecución, por lo que es posible utilizar un motor para un método secundario, cambiar el motor y aplicar el segundo método secundario con el nuevo motor de forma transparente.

Como se ve en la Figura 3.5, la clase abstracta *Engine* se comporta como la interfaz que el cliente de este módulo puede usar, donde *CPUEngine* y *OpenCLEngine* son las implementaciones. Para el caso de *CPUEngine*, basta con recibir los vectores de datos para trabajar con ellos, mientras que para *OpenCLEngine* se crearon atributos de clase que permitieran trabajar con OpenCL sin tener que generarlos por cada ejecución.

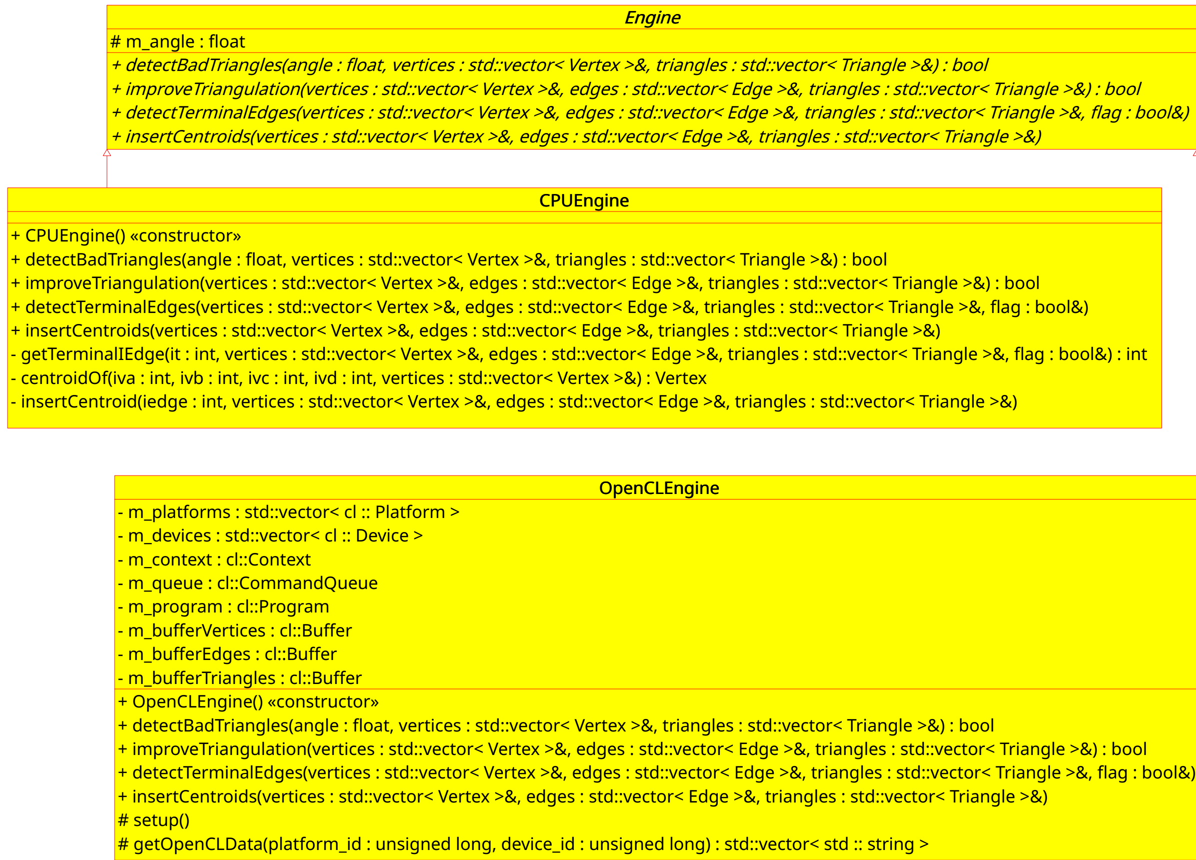

Figura 3.5: UML del módulo de procesamiento.

3.5. Interfaz Gráfica

El diseño de la interfaz fue pensado con énfasis en la simplicidad y facilidad de uso, pero permitiendo utilidades que se adapten a usuarios que trabajan solo con el *mouse* y también a usuarios que trabajan mayormente con el teclado (integrando *keybindings* a la aplicación).

Además, la interfaz se diseñó pensando en que tendría que integrarse con una biblioteca que ya existía, por lo que las conexiones hechas entre la biblioteca y la interfaz están directamente relacionadas entre sí.

Puesto que Qt trabaja nativamente con el patrón de arquitectura ModelView, se aplicó este mismo patrón en la conexión entre interfaz y biblioteca. Además, como se puede ver en la Figura 3.6, que muestra la parte más relevante del UML de la interfaz, se ha diseñado la interfaz de tal forma que sea posible intercambiar el *widget* utilizado para mostrar gráficos en OpenGL.

Por otro lado, como lo muestra la Figura 3.7, se consideró además el ofrecer la posibilidad de agregar ventanas, para mostrar ayuda de uso del programa, y para mostrar una ventana de Acerca de.

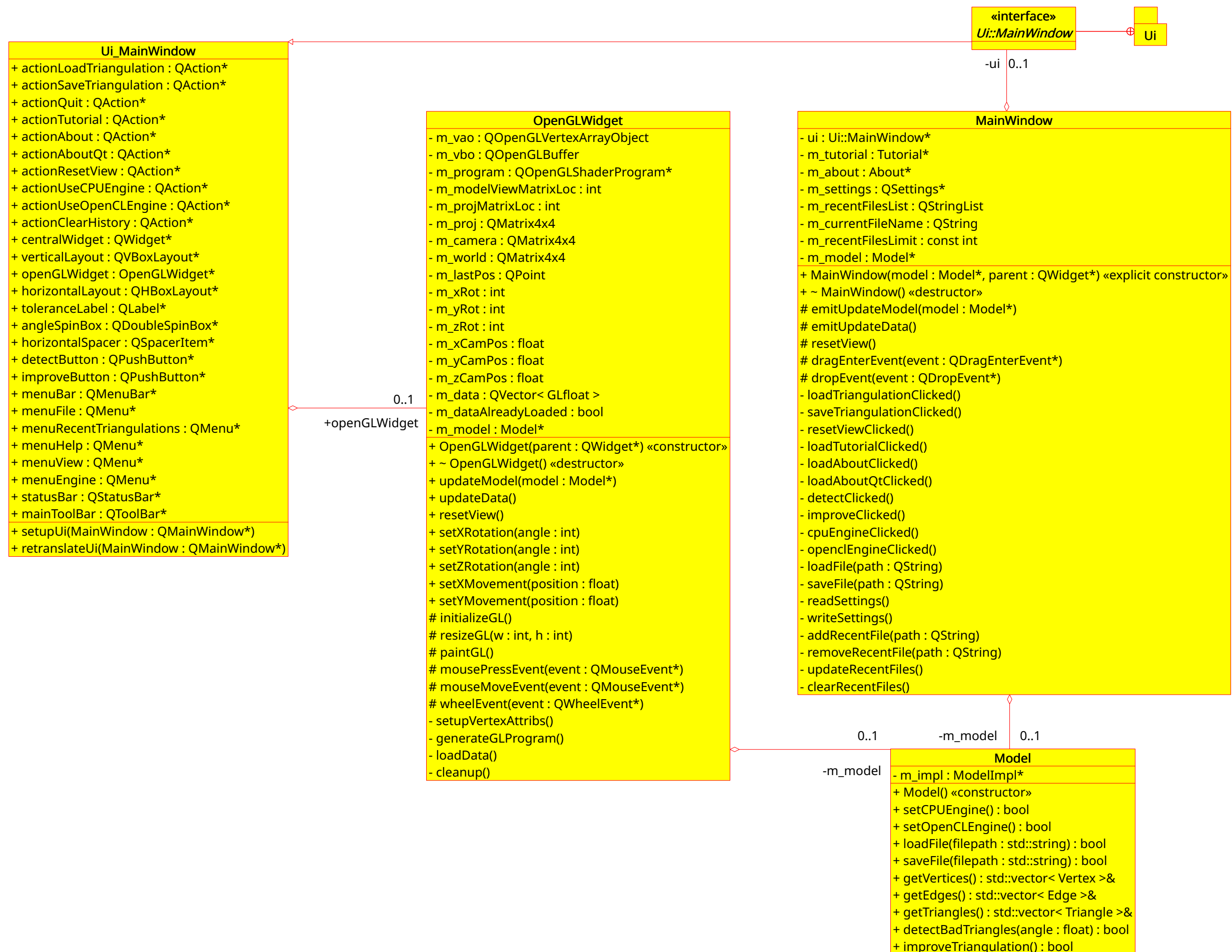

Figura 3.6: UML de la Interfaz (Sección principal).

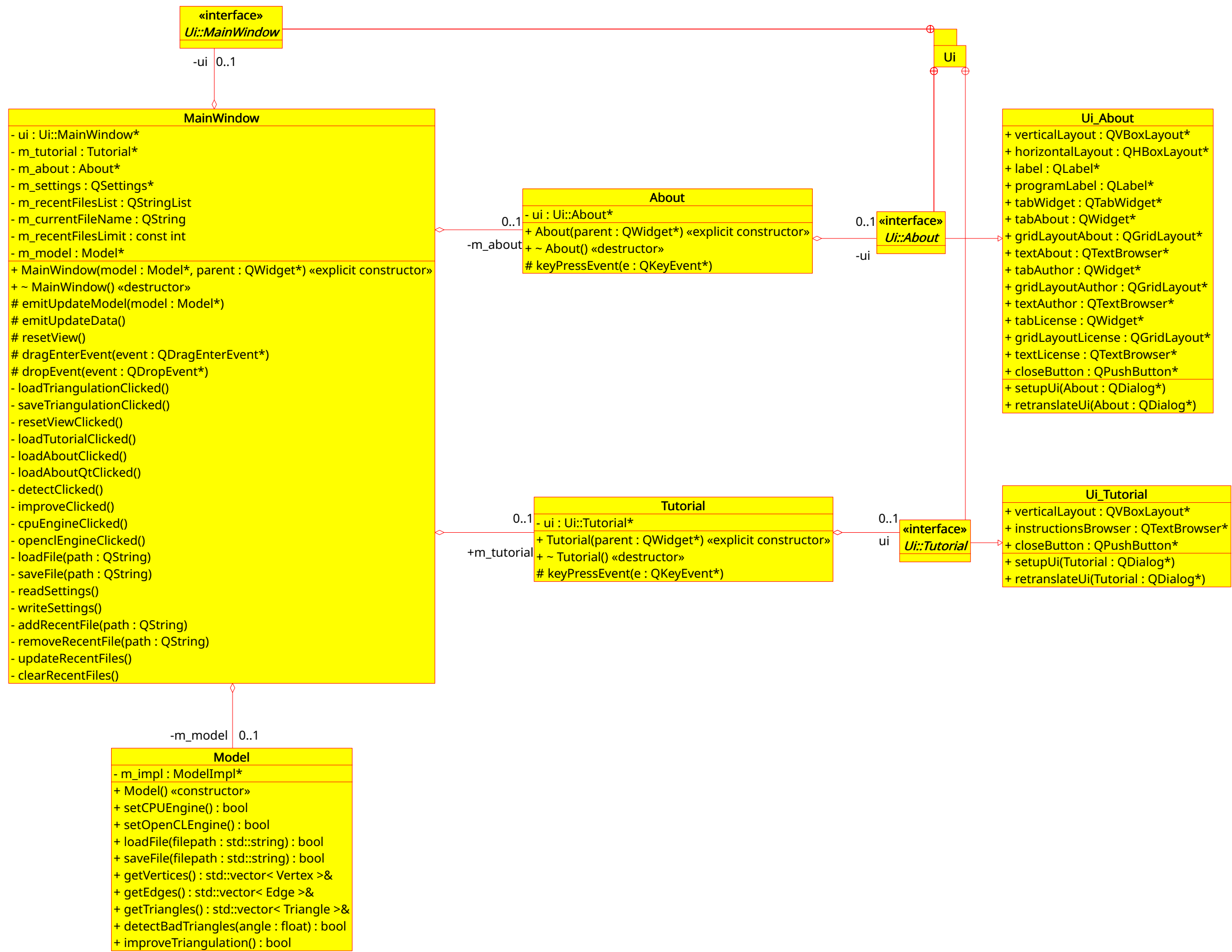

Figura 3.7: UML de la Interfaz (Sección secundaria).

3.5.1. Módulo MainWindow

El módulo principal de la interfaz de esta aplicación debe ser capaz de ofrecer todas las capacidades principales que ofrece la biblioteca de una forma concisa y simple de usar, de forma similar a lo que hace el façade de la biblioteca. Se decidió que este módulo tendría la referencia al modelo, por lo que para compartirlo con otros módulos se utiliza el mecanismo *Signal/Slot* que ofrece Qt.

Como se puede ver en la Figura 3.8, la interfaz fue creada pensando en permitir cargar y guardar triangulaciones, detectar y mejorarlas y cargar distintos motores de la biblioteca. Adicionalmente, se diseñaron formas de mejorar la experiencia de usuario, tales como permitir recordar los archivos recientes o cargar un archivo arrastrándolo a la ventana.

El uso del *namespace* UI::MainWindow lo genera Qt automáticamente, lo que permite separar la lógica de manejo de la vista respecto a cada uno de los botones utilizados en la aplicación. De esta forma, la clase *MainWindow* se puede usar como *façade*, mientras que la clase *UI_MainWindow* contiene las posiciones de cada botón en la interfaz.

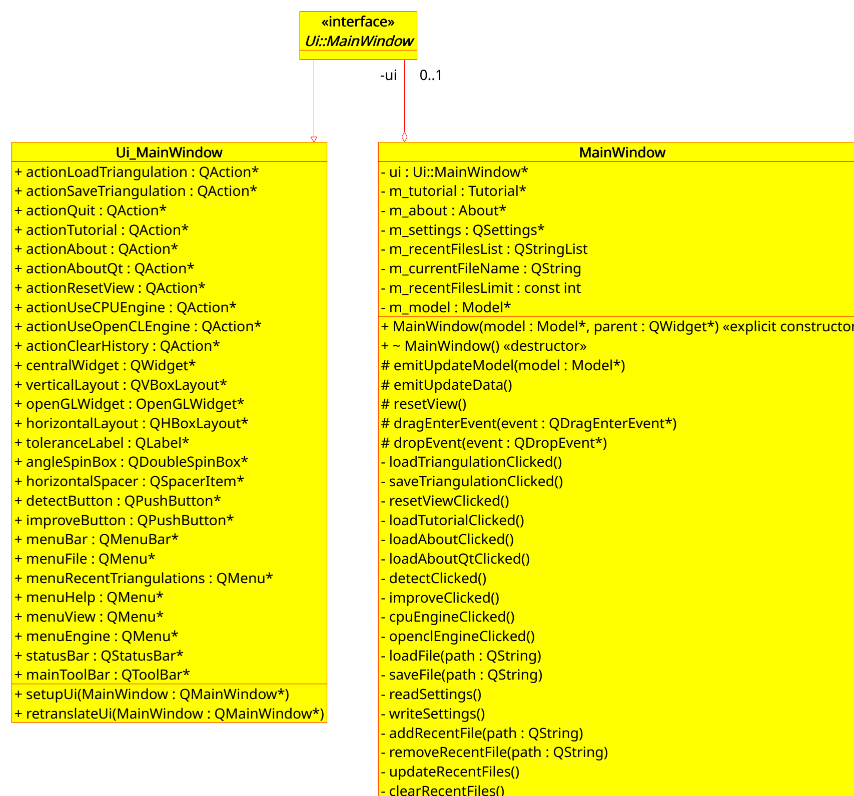

Figura 3.8: UML del módulo MainWindow.

3.5.2. Módulo OpenGLWidget

Este módulo dibuja la triangulación cargada por la biblioteca usando OpenGL. Dado que la referencia al modelo está en el módulo MainWindow, se utiliza el mecanismo *Signal/Slot* de Qt al momento de cargar el archivo de la malla y al momento de calcular los triángulos "malos" y el mejoramiento de la triangulación, enviando la dirección de memoria del modelo al momento de iniciar la aplicación.

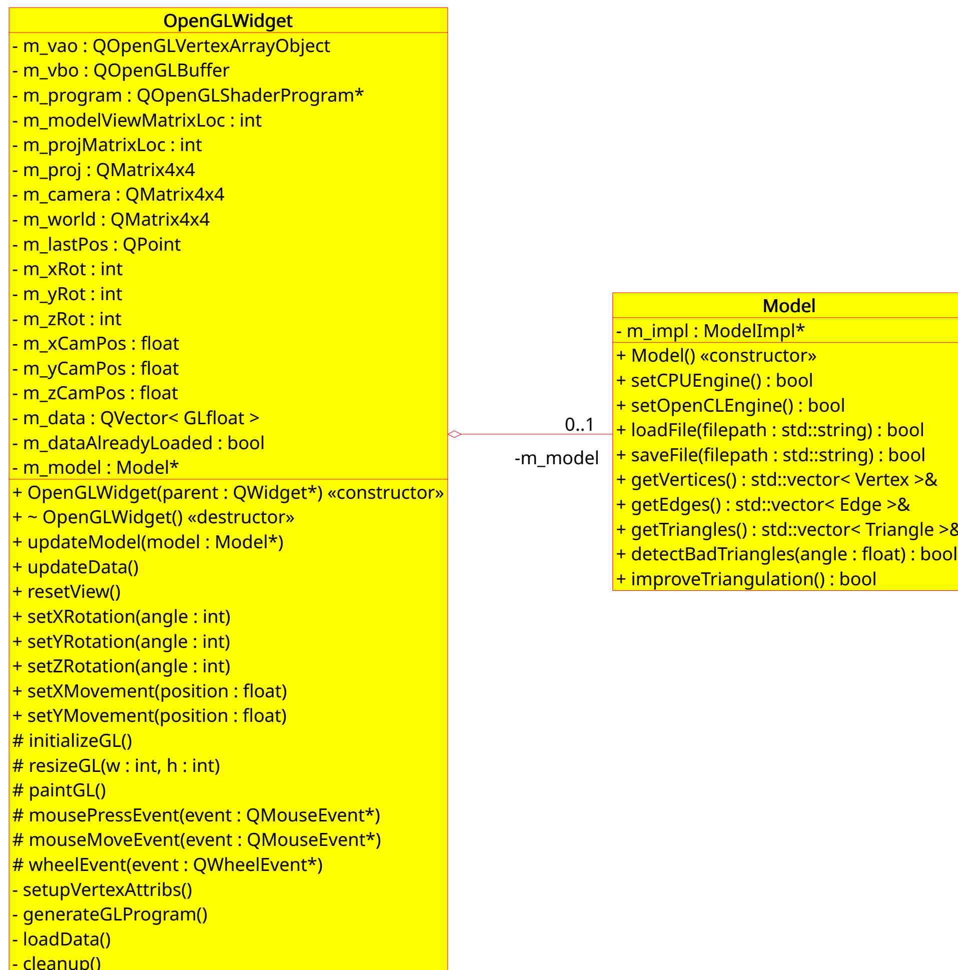

Figura 3.9: UML del módulo OpenGLWidget.

3.5.3. Módulo About

Este módulo muestra una pantalla del creador de la aplicación, la versión de la aplicación y un acceso a la licencia correspondiente. Este módulo fue creado como demostración de la posibilidad de agregar ventanas a la aplicación.

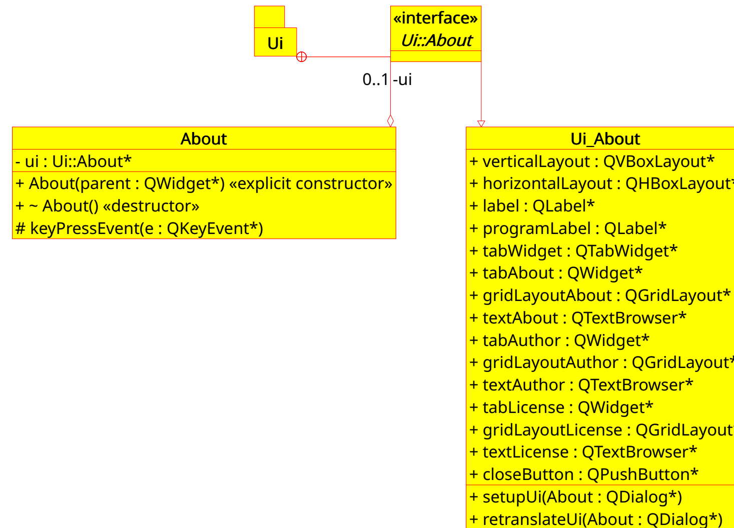

Figura 3.10: UML del módulo About.

3.5.4. Módulo Tutorial

Este módulo muestra una pantalla que le explica al usuario la forma de usar la interfaz de la aplicación. Al igual que en el módulo About, este módulo fue creado como demostración de la posibilidad de agregar ventanas a la aplicación y adicionalmente demostrar las capacidades de internacionalización de Qt.

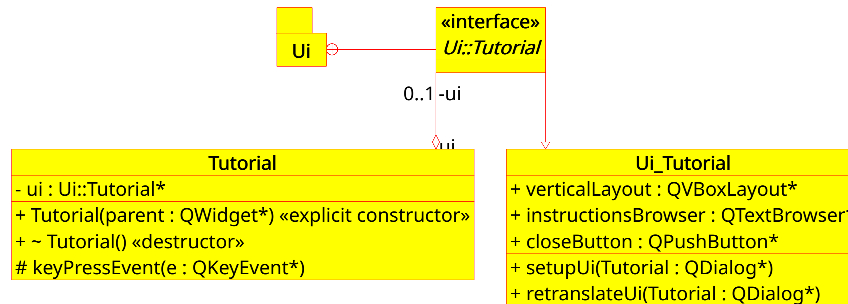

Figura 3.11: UML del módulo Tutorial.

Capítulo 4

Implementación

En este capítulo se presentará el ambiente utilizado para el desarrollo de la aplicación, los detalles de implementación de la biblioteca y los detalles de implementación de la interfaz, además de mostrar cómo instalar ambos componentes y cómo usarlos.

En el caso de la biblioteca, se explican detalles tales como las estructuras de datos usadas, cómo se manejan los archivos y cómo fueron implementados los algoritmos que se usaron para procesar las mallas de triángulos.

En el caso de la interfaz, se explica cómo se implementó la pantalla principal, los menús, el paso del archivo de malla de la interfaz a la biblioteca, el renderizado con OpenGL, las ventanas de "Acerca de" y del "Tutorial", la internacionalización ofrecida, los *keybindings* y el *drag-and-drop*.

4.1. Ambiente de desarrollo

Esta aplicación fue desarrollada usando en un sistema GNU/Linux de 64 bits (Arch Linux), utilizando el entorno de desarrollo integrado (IDE) QtCreator 4.7 para la biblioteca y la interfaz respectivamente.

Las bibliotecas utilizadas fueron Qt 5.11.1, GCC 8.1.1 y OpenCL 1.2. Se utilizó el sistema de compilación *qmake*, que genera el archivo Makefile a utilizar durante la compilación. Se puede instruir a *qmake* a hacer distintos pasos al compilar dependiendo del sistema operativo donde se instalará la aplicación.

4.2. Biblioteca

Para poder implementar la biblioteca se utilizó el template *lib* que usa qmake y se eliminó la bibloteca GUI que ofrece Qt por defecto, escribiendo estas opciones en el archivo `QLepp2DLib.pro`, que maneja qmake.

Adicionalmente, en el archivo .pro se definió la forma en la cual serían instalados los headers al momento de instalar la biblioteca, para no exponer la implementación privada de la biblioteca.

Para incluir la biblioteca de OpenCL, se agregó el *kernel*[1] usado como Recurso de Qt, a fin de evitar problemas de ubicación del archivo y no tener que depender de *paths* relativos.

4.2.1. Estructuras de Datos

A fin de reutilizar la mayoría de las herramientas ya existentes en el lenguaje de programación y el *framework* utilizado, se exponen las siguientes estructuras principales a considerar:

C++

- **std::vector**: Es un *template* de contenedores que representan arreglos que pueden cambiar dinámicamente de tamaño.
- **std::string**: Es la estandarización de C++ para una secuencia de caracteres, que facilita el tratamiento de strings, en lugar de usar `char *`.
- **std::min**: Es un template que permite detectar el mínimo entre dos valores.
- **std::max**: Es un template que permite detectar el máximo entre dos valores.

Qt

- **QVector**: Contenedor de Qt optimizado para accesos aleatorios rápidos.
- **QString**: Implementación de Qt de strings, con una API más rica y fácil de usar.
- **QFile**: Interfaz para leer y escribir archivos.
- **QFileInfo**: Provee información de archivos independiente del sistema operativo.
- **QMap**: Provee un diccionario basado en un árbol *red-black*. Similar a QHash, pero provee orden por llaves.
- **QTextStream**: Provee una interfaz de lectura y escritura de texto.

[1] El *kernel* es el archivo que contiene el código ejecutado por OpenCL

Por otro lado, las estructuras de datos Vertex, Edge y Triangle expuestas en la sección de Análisis y Diseño fueron creadas como *structs* en lugar de clases, porque OpenCL solo permite estructuras personalizadas, por motivos de manejo de memoria de la biblioteca de OpenCL. Existen otras estructuras conocidas que se pudieron haber usado (*half-edge*, por ejemplo), pero se decidió usar las estructuras que se exponen aquí, pues son las más fáciles y directas de manejar en GPU.

OpenCL no garantiza que los tamaños de los tipos primitivos [2] sean del mismo tamaño en CPU y GPU, pero ofrece sus propios tipos *cl_int* y *cl_double* (entre otros), los cuales se usan en el *host* para obtener el mismo tamaño de cada tipo primitivo de la GPU.

```
typedef struct {
    cl_float x;
    cl_float y;
    cl_float z;
} Vertex;

typedef struct {
    cl_int ita;
    cl_int itb;
    cl_int iv1;
    cl_int iv2;
    cl_int isTE;
} Edge;

typedef struct {
    cl_int iv1;
    cl_int iv2;
    cl_int iv3;
    cl_int ie1;
    cl_int ie2;
    cl_int ie3;
    cl_int bad;
} Triangle;
```

Figura 4.1: Estructuras usadas en CPU.

```
typedef struct {
    float x;
    float y;
    float z;
} Vertex;

typedef struct {
    int ita;
    int itb;
    int iv1;
    int iv2;
    int isTE;
} Edge;

typedef struct {
    int iv1;
    int iv2;
    int iv3;
    int ie1;
    int ie2;
    int ie3;
    int bad;
} Triangle;
```

Figura 4.2: Estructuras usadas en GPU (OpenCL).

[2] *int*, *float*, *double*, etc.

4.2.2. Carga de Archivo

Dado que esta aplicación está originalmente creada para cargar archivos OFF, se mostrará el proceso de carga de este tipo de archivos específicamente. (Para la explicación de la estructura de un archivo OFF, ver Anexo A.)

Al momento de recibir el archivo que contiene la malla, primero se verifica que se pueda cargar. Si se puede cargar, se revisa que la primera línea diga `OFF`, y se omiten todas las siguientes líneas que comiencen con `#` (puesto que éstas se consideran comentarios).

La primera línea que contiene información de la malla es la que muestra la cantidad de vértices, caras y aristas (en ese orden) contenidas en el archivo. Por lo tanto, se asignan como limitadores de lectura para los vértices y los triángulos.

Para cada línea dentro del límite de vértices existentes en el archivo, se escanean los 3 números que representan las 3 coordenadas, se crea un nuevo *Vertex* y se añade al arreglo de vértices.

Para cada línea dentro del límite de triángulos existentes en el archivo (i.e., **para cada triángulo**) se realizan las siguientes acciones:

Inicialización de triángulo

Se crea un triángulo con información incompleta. Esto es, se leen los índices y se agregan al triángulo, pero se deja la información de las aristas en -1, puesto que aún no se sabe en qué posición del arreglo de aristas se encontrará cada una de éstas. Después de crearlo, se inserta al arreglo tal como está (incompleto).

```
line = in.readLine();
QStringList mappedIndices = line.split(" ", QString::SkipEmptyParts);
Triangle t;

t.iv1 = mappedIndices.at(1).toInt();
t.iv2 = mappedIndices.at(2).toInt();
t.iv3 = mappedIndices.at(3).toInt();
t.ie1 = -1;
t.ie2 = -1;
t.ie3 = -1;
t.bad = 0;

triangles.push_back(t);
```

Generación de aristas e inserción en QMap

Luego, se detectan los 3 índices a vértices del triángulo, se crean las 3 aristas correspondientes a [3]:

```
QString("%1-%2").arg(std::min(t.iv1, t.iv2))).arg(std::max(t.iv1, t.iv2)));
QString("%1-%2").arg(std::min(t.iv2, t.iv3))).arg(std::max(t.iv2, t.iv3)));
QString("%1-%2").arg(std::min(t.iv3, t.iv1))).arg(std::max(t.iv3, t.iv1)));
```

Si en el *QMap* no existe una arista con ese identificador, significa que recién se está leyendo este triángulo. Por lo tanto, se crea una arista nueva con los siguientes valores: (Considérese e el nombre de la arista)

```
// Ejemplo con iv1 = índice a vértice 1, iv2 = índice a vértice 2
e.iv1 = std::min(iv1, iv2);
e.iv2 = std::max(iv1, iv2);
e.ita = i; // Índice del triángulo actual
e.itb = -1; // Aún no hemos encontrado un triángulo vecino
e.isTE = 0; // Al inicio no se sabe si es triángulo terminal
```

En cambio, si en el *QMap* ya existe una arista con ese identificador, significa que ya existe un triángulo anterior con esa arista, por lo que este triángulo será vecino del anterior. Por lo tanto, la arista simplemente actualiza el segundo triángulo que contiene la arista, eliminando el -1 de esta arista.[4]

```
e = map.value(key); // Se rescata el edge del QMap
e.itb = i; // Índice del triángulo actual, vecino de uno encontrado antes
```

Actualización de aristas

Posteriormente a este proceso, se itera sobre cada arista creada en el paso anterior[5], rescatando las posiciones de los triángulos (que están en *e.ita* y *e.itb*). Adicionalmente, se inserta la arista en el arreglo de aristas y se guarda el número de esta arista. [6].

```
int k = 0; // Contador de aristas procesadas
for (QMap<QString, Edge>::iterator i(map.begin()); i != map.end(); i++,k++)
{
```

[3]Nótese que estos identificadores son QString, donde los dos números son separados con un guión, no confundir con una resta de números. Además, se garantiza la unicidad de la llave utilizando *min* y *max*.

[4]Nótese que como esta acción se realiza iterando índices, siempre resulta que el índice del primer vértice es menor que el del segundo vértice.

[5]En estos momentos, las aristas se encuentran solo en el *QMap*, pero ya poseen información completa.

[6]Como hay más aristas que triángulos, se usa un iterador sobre el *QMap*, por lo que hay que mantener manualmente un contador de aristas procesadas.

```
    Edge e(i.value());
    edges.push_back(e);
    (...)
```

Respecto al triángulo rescatado para cada caso, si el vértice n-ésimo del triángulo difiere de ambos vértices de la arista, éste se asigna como vértice opuesto a la arista n-ésima sobre la que se está trabajando.

Esto se hace tanto para el primer triángulo (que **debe** existir, por construcción) como para el segundo triángulo (si existe).

```
    // Triángulo A
    Triangle &ta = triangles.at(e.ita);
    /* Si el vértice "a" del triángulo no está en e.iv1 o e.iv2,
    * entonces este vértice "a" es el opuesto de la arista actual.
    */
    if (ta.iv1 != e.iv1 and ta.iv1 != e.iv2)
    {
        ta.ie1 = k;
    }
    else if (ta.iv2 != e.iv1 and ta.iv2 != e.iv2)
    {
        ta.ie2 = k;
    }
    else if (ta.iv3 != e.iv1 and ta.iv3 != e.iv2)
    {
        ta.ie3 = k;
    }
    else
    {
        qFatal("Inconsistent data (A)!");
    }

    // Triángulo B
    if (e.itb < 0)
    {
        continue; // Quizá no existe un vecino.
    }
    (...) /* Código análogo al Triángulo A */
}
```

De esta forma, los vértices, las aristas y los triángulos se cargan en memoria, cada uno de ellos con información completa.

4.2.3. Guardado de Archivo

Dado que esta aplicación está originalmente creada para cargar archivos OFF, se mostrará el proceso de guardado a este tipo de archivos específicamente.

Para guardar la información que está contenida en memoria, primero se abre un nuevo archivo de texto, donde se escribe “OFF” (que valida que el archivo es del tipo OFF), se agrega un comentario de “Archivo creado con QLepp2D” y se escribe el número de vértices, triángulos y aristas que contienen cada uno de los arreglos.

```
QTextStream out(&outputFile);
out << "OFF" << endl;
out << "# File created by QLepp2D." << endl;

out << numVertices << " " << numTriangles << " " << numEdges;
```

Después, se escriben las coordenadas de los vértices que contiene el arreglo de vértices, agregando un salto de línea por cada 3 coordenadas escritas.

```
unsigned long coordinatesPerVertex = vertices.size() / numVertices;
for (unsigned long i(0); i < vertices.size(); i++)
{
    if (i % coordinatesPerVertex == 0)
    {
        out << endl;
    }
    out << vertices.at(i).x << " ";
    out << vertices.at(i).y << " ";
    out << vertices.at(i).z;
}
```

Finalmente, para cada triángulo, se escribe primero un “3” (que muestra la cantidad de lados de esta figura), y luego se escriben los índices a los vertices a los que apuntan cada uno de los triángulos.

```
int verticesPerFace = 3;
for (const Triangle &t : triangles)
{
    out << verticesPerFace << " ";
    out << t.iv1 << " ";
    out << t.iv2 << " ";
    out << t.iv3 << endl;
}
```

4.2.4. File Managers

Dado que esta aplicación fue pensada para ser extensible, se requirió que hubiese un administrador de clases que supieran cargar y guardar archivos [7]. Por lo tanto se creó este *FileManager* que pudiese:

Agregar *handlers* Se implementó un sistema muy simple que pudiese agregar un *handler*, asociándole su extensión correspondiente utilizando un *QMap* con la extensión como llave y el *handler* como su valor.

```
bool FileManager::addFileHandler(FileHandler *handler,
                                 std::string extension)
{
    if (extension.size() > 0)
    {
        m_handlers.insert(QString::fromStdString(extension), handler);
        return true;
    }
    return false;
}
```

Cargar archivos

A partir del nombre del archivo a cargar, se extrae su extensión, se busca en el *QMap* su *handler* asociado, y se ejecuta su método *load()*.

```
bool FileManager::load(std::string filepath,
                    std::vector<Vertex> &vertices,
                    std::vector<Edge> &edges,
                    std::vector<Triangle> &triangles)
{
    QFileInfo fileinfo(QString::fromStdString(filepath));
    QString ext = fileinfo.suffix();

    FileHandler *handler = m_handlers.value(ext);
    return (handler != nullptr and
            handler->load(filepath, vertices, edges, triangles));
}
```

[7]Estas clases las llamaremos *File Handlers* o *handlers*.

Guardar archivos

A partir del nombre del archivo a guardar, se extrae su extensión, se busca en el *QMap* su *handler* asociado, y se ejecuta su método *save()*.

```
bool FileManager::save(std::string filepath,
                    std::vector<Vertex> &vertices,
                    std::vector<Edge> &edges,
                    std::vector<Triangle> &triangles)
{
    QFileInfo fileinfo(QString::fromStdString(filepath));
    QString ext = fileinfo.suffix();

    FileHandler *handler = m_handlers.value(ext);
    return (handler != nullptr and
            handler->save(filepath, vertices, edges, triangles));
}
```

Gracias a la simplicidad de la implementación, es posible agregar nuevos *handlers* al *FileManager* y ejecutarlos dinámicamente.

4.2.5. Procesamiento

Para poder incluir varios motores que pudieran procesar el mejoramiento de una triangulación, se diseñó una interfaz *engine.h* que posee los siguentes métodos.

```
Engine() = default;
virtual ~Engine() = default;

virtual bool detectBadTriangles(double angle,
                                std::vector<Vertex> &vertices,
                                std::vector<Triangle> &triangles) = 0;

virtual void detectTerminalEdges(std::vector<Vertex> &vertices,
                                 std::vector<Edge> &edges,
                                 std::vector<Triangle> &triangles,
                                 bool &flag) = 0;

virtual void insertCentroids(std::vector<Vertex> &vertices,
                             std::vector<Edge> &edges,
                             std::vector<Triangle> &triangles) = 0;

virtual bool improveTriangulation(std::vector<Vertex> &vertices,
```

```
                                    std::vector<Edge> &edges,
                                    std::vector<Triangle> &triangles) = 0;
```

Estos 4 métodos son los que deben ofrecer los motores que quieran implementar su versión de mejoramiento de triangulaciones usando Lepp-Centroide.

detectBadTriangles

Recibe 3 argumentos: El ángulo que hace de *threshold* entre los triángulos buenos y los malos, el vector de vértices y el vector de triángulos.

Este método se encarga de detectar todos los triángulos malos de la triangulación, esto es, cada uno de los triángulos cuyo ángulo mínimo sea menor que *angle* será marcado para su procesamiento por los siguentes métodos que se ejecuten.

detectTerminalEdges

Recibe 4 argumentos: El vector de vértices, el vector de aristas, el vector de triángulos y un *flag*.

Este método se encarga de tomar todos los triángulos que hayan sido marcados por *detectBadTriangles* y genera una lista Lepp para cada uno de ellos, para recorrerlos hasta llegar a su arista terminal.[8]

Al momento de llegar a las aristas terminales, éstas son marcadas como tales para su procesamiento al momento de insertar los centroides. Adicionalmente, se tiene un parámetro *flag*, que marca si aún existen aristas terminales que no sean de borde. Puesto que esta aplicación no divide aristas que sean de borde, se requiere esta *flag* para saber si todavía quedan aristas terminales que no sean de borde.

insertCentroids

Recibe 3 argumentos: El vector de vértices, el vector de aristas y el vector de triángulos.

Este método se encarga de revisar los edges marcados por el método *detectTerminalEdges*, calcular el centroide de los 4 vértices que componen los 2 triángulos compartidos, insertar el centroide y reemplazar estos 2 triángulos en 4 nuevos triángulos que contengan dicho centroide.

[8] Es posible que dos listas Lepp distintas lleguen a la misma arista terminal, pero eso no genera problemas de colisión, pues se trabajará en las aristas.

improveTriangulation

Este método reúne los 3 métodos anteriores en uno solo, de forma que sea posible ejecutar un mejoramiento de la malla directamente, en lugar de llamar manualmente a los 3 métodos. Se usa como acceso directo para los botones de la interfaz (GUI), los cuales pueden llamar a este método usando la API ofrecida por la biblioteca.

4.2.6. Motor de CPU

Aquí serán explicadas en detalle las responsabilidades de cada método expuesto en la sección de "Procesamiento" cuando se implementa en CPU.

Detección de triángulos malos (detectBadTriangles)

Para cada uno de los triángulos del vector, se toman sus vértices (usando los índices dentro del triángulo y el vector de vértices).

```
Vertex A, B, C;
A = vertices.at(t.iv1);
B = vertices.at(t.iv2);
C = vertices.at(t.iv3);
```

Luego, se calcula el largo de cada arista, tomando la diferencia entre las coordenadas de dos vértices, para cada arista.

```
float length_A = pow(B.x - C.x, 2) + pow(B.y - C.y, 2) + pow(B.z - C.z, 2);
float length_B = pow(A.x - C.x, 2) + pow(A.y - C.y, 2) + pow(A.z - C.z, 2);
float length_C = pow(A.x - B.x, 2) + pow(A.y - B.y, 2) + pow(A.z - B.z, 2);

float length_a = sqrt(length_A);
float length_b = sqrt(length_B);
float length_c = sqrt(length_C);
```

Finalmente, se utiliza la Ley del Coseno[17] para detectar los ángulos internos del triángulo, para marcarlo si sus ángulos son menores que el *threshold* dado por *angle*.

```
float angle_opp_A = std::acos((length_B + length_C - length_A)
                              / (2 * length_b * length_c));
float angle_opp_B = std::acos((length_A + length_C - length_B)
                              / (2 * length_a * length_c));
float angle_opp_C = std::acos((length_A + length_B - length_C)
                              / (2 * length_a * length_b));
```

Detección de aristas terminales (detectTerminalEdges)

Para cada uno de los triángulos del vector, se detecta si el triángulo es malo, y se llama a un método *helper* `getTerminalIEdge`, el cual es el encargado de retornar el índice al vector de aristas donde se encuentra la arista terminal de este triángulo, para poder marcarlo.

```
for (int i(0); i < static_cast<int>(triangles.size()); i++)
{
    Triangle &t(triangles.at(i));
    if (t.bad)
    {
        int longestIEdge = getTerminalIEdge(i,
                                            vertices,
                                            edges,
                                            triangles,
                                            flag);
        edges.at(longestIEdge).isTE = 1;
    }
}
```

Como el método *helper* `getTerminalIEdge` es muy importante, será explicado en el párrafo inmediatamente posterior.

getTerminalIEdge

Recibe 5 argumentos: El índice al triángulo inicial, el vector de vértices, el vector de aristas, el vector de triángulos y el *flag* mencionado anteriormente.

Este método crea un historial simplificado de triángulos, donde almacena los índices de los últimos 3 triángulos que se visitaron, por lo que se inicializa con {-1, -1, -1}.

```
QVector<int> triangleHistory;

// Índice of triangleHistory
int k = 0;

triangleHistory.resize(3);
for (int j(0); j < 3; j++)
{
    triangleHistory.push_back(-1);
}
```

Luego, se ejecuta un *loop* infinito (que se romperá al encontrar la arista terminal), donde se agrega el triángulo al historial.

```
// Triángulo obtenido por copia, no por referencia
Triangle t(triangles.at(it));

while (true)
{
    // Agregar triángulo al historial
    triangleHistory[k] = it;
    (...)
```

Se obtienen los vértices del triángulo y el largo (al cuadrado) de sus aristas.

```
Vertex A, B, C;
A = vertices.at(t.iv1);
B = vertices.at(t.iv2);
C = vertices.at(t.iv3);

float length_A = pow(B.x - C.x, 2) + pow(B.y - C.y, 2) + pow(B.z - C.z, 2);
float length_B = pow(A.x - C.x, 2) + pow(A.y - C.y, 2) + pow(A.z - C.z, 2);
float length_C = pow(A.x - B.x, 2) + pow(A.y - B.y, 2) + pow(A.z - B.z, 2);
```

Luego, se busca cuál de estas aristas es la más larga, para conocer cuál es su triángulo vecino.

```
int neighbourIT;
int longestIE;
Edge longestEdge;

if (length_A > length_B and length_A > length_C)
{
    longestEdge = edges.at(t.ie1);
    longestIE = t.ie1;
}
else if (length_B > length_A and length_B > length_C)
{
    longestEdge = edges.at(t.ie2);
    longestIE = t.ie2;
}
else // Unicidad de arista garantizada aquí, para equiláteros o isósceles
{
    longestEdge = edges.at(t.ie3);
```

```
    longestIE = t.ie3;
}

// Detectar índice a mi vecino.
neighbourIT = (longestEdge.ita == it) ? longestEdge.itb : longestEdge.ita;
```

Si el vecino es arista terminal, simplemente se retorna el índice de la arista más larga encontrada hasta aquí.

```
// Triángulo de borde
if (neighbourIT < 0)
{
    return longestIE;
}
```

Si no, se verifica en el historial si el triángulo actual ya fue visitado anteriormente.

```
// Si este triángulo está en el historial, entonces
// ésta (longestIE) es la arista terminal del Lepp.
if (it == triangleHistory.at((k + 1) % 3))
{
    flag = true;
    return longestIE;
}
```

Finalmente, si no fue ninguno de los casos anteriores, se actualiza el historial para la siguiente iteración del *loop*.

```
// Actualizar al triángulo vecino de arista más larga
it = neighbourIT;
t = triangles.at(neighbourIT);
k = (k + 1) % 3; // Actualizar puntero del historial
```

Inserción de centroides (insertCentroids)

Para cada una de las aristas del vector, se detecta si fue marcada como arista terminal por el método `detectTerminalEdges` y si no es una arista de borde (i.e., el índice de su triángulo vecino es -1), para llamar a un método *helper* `insertCentroid`, el cual es el encargado de insertar el centroide de los dos triángulos marcados por la arista correspondiente.

```
for (unsigned int ie(0); ie < edges.size(); ie++)
{
    Edge &e(edges.at(ie));
```

```
    // Si e.itb == -1, es arista de borde, por lo que no
    // insertaremos un centroide ahí
    if (e.isTE and e.itb != -1)
    {
        insertCentroid(ie, vertices, edges, triangles);
    }
}
```

Como el método *helper* `insertCentroid` es el método más importante del algoritmo, será explicado en el párrafo inmediatamente posterior.

insertCentroid

Recibe 4 argumentos: El índice a la arista terminal ya detectada, el vector de vértices, el vector de aristas y el vector de triángulos.

Este algoritmo es muy largo, por lo cual se dividirá en 1 "fase" inicial y 7 "fases" de desarrollo. [9]

Fase 0

Se respaldan copias de los triángulos compartidos por la arista compartida, y se respalda la misma arista compartida.

```
// Copias, no referencias
Edge oldE(edges.at(iedge));
Triangle oldTA(triangles.at(oldE.ita));
Triangle oldTB(triangles.at(oldE.itb));
```

Fase 1

Se obtienen los 4 vértices de los 2 triángulos, y se crea un vector de vértices con patrones predefinidos, a fin de evitar crear triángulos *slivers*. Los patrones disponibles para usar fueron denominados "Patrón NSC" y "Patrón SNC". (Para ver detalles de estos patrones, ver Anexo B):

- N: Vértice que **no** se encuentra en la arista compartida.
- S: Vértice que **sí** se encuentra en la arista compartida.
- C: Centroide.

[9] La fase inicial será llamada "Fase 0".

```
// Arbitrariamente comenzaremos con el patrón NSC aquí.
QVector<int> iVertexPattern;
iVertexPattern.push_back(-1);
iVertexPattern.push_back(oldE.iv1);
iVertexPattern.push_back(-1);
iVertexPattern.push_back(oldE.iv2);

int i = 0;
for (int iv : {oldTA.iv1, oldTA.iv2, oldTA.iv3,
               oldTB.iv1, oldTB.iv2, oldTB.iv3})
{
    if (not iVertexPattern.contains(iv))
    {
        iVertexPattern[i] = iv;
        i = 2;
    }
}
```

Como se puede ver en el fragmento de código anterior, *iVertexPattern* comienza con el patrón NSC, por lo que el primer y el tercer vértice son temporalmente inválidos. Para poder cambiarlos, se busca en 6 vértices de los dos triángulos (incluyendo duplicados), y se añaden los que no estaban ya en *iVertexPattern*.

Una vez que se crea el vector de patrones, se chequea si los futuros triángulos serán CCW. Si no lo son, se invierten los vértices de clase S para cambiar la orientación de todos los futuros triángulos a CCW. Este chequeo se realiza con la fórmula del área de Gauss[10].

```
float area(0.0f);
int n(3);

int j(n - 1);
for (int i(0); i < n; i++)
{
    area += (vertices.at(iVertexPattern.at(j)).x +
             vertices.at(iVertexPattern.at(i)).x) *
            (vertices.at(iVertexPattern.at(j)).y -
             vertices.at(iVertexPattern.at(i)).y);
    j = i;
}
```

[10]https://en.wikipedia.org/wiki/Shoelace_formula

```
// Area < 0 ==> CCW
if (area > 0)
{
    int tmp(iVertexPattern.at(1));
    iVertexPattern[1] = iVertexPattern.at(3);
    iVertexPattern[3] = tmp;
}
```

Después, se crea el centroide, con el método *helper* `centroidOf`, y se añade al vector de vértices.

```
// Detectamos las coordenadas del centroide y lo creamos
Vertex CPUEngine::centroidOf(int iva,
                             int ivb,
                             int ivc,
                             int ivd,
                             std::vector<Vertex> &vertices)
{
    Vertex centroid;
    centroid.x = (vertices.at(iva).x +
                  vertices.at(ivb).x +
                  vertices.at(ivc).x +
                  vertices.at(ivd).x) / 4.0f;
    centroid.y = (vertices.at(iva).y +
                  vertices.at(ivb).y +
                  vertices.at(ivc).y +
                  vertices.at(ivd).y) / 4.0f;
    centroid.z = (vertices.at(iva).z +
                  vertices.at(ivb).z +
                  vertices.at(ivc).z +
                  vertices.at(ivd).z) / 4.0f;

    return centroid;
}
(...)
// Añadimos el centroide
Vertex centroid = centroidOf(iVertexPattern.at(0),
                             iVertexPattern.at(1),
                             iVertexPattern.at(2),
                             iVertexPattern.at(3),
                             vertices);
vertices.push_back(centroid);
int iCentroid = vertices.size() - 1; // Index of new Vertex
```

Fase 2

Se detectan todas las aristas que no corresponden a la arista compartida. Se puede filtrar utilizando las 6 aristas (incluyendo duplicadas), puesto que el índice de la arista compartida es el mismo, por lo que quedarán 4 aristas no compartidas, las que se usarán posteriormente.

```
QVector<int> nonSharedIEdges;
for (int ie : {oldTA.ie1, oldTA.ie2, oldTA.ie3,
               oldTB.ie1, oldTB.ie2, oldTB.ie3})
{
    if (ie != iedge)
    {
        nonSharedIEdges.append(ie);
    }
}
```

Se crean 4 aristas nuevas, que serán las que lleven al centroide. Dado que no hay información suficiente en esta etapa del proceso, todos sus valores serán inicializados en -1 (y el marcador de arista terminal en 0).

```
QVector<Edge> newEdges;
for (int ie(0); ie < 4; ie++)
{
    Edge e;
    e.ita = e.itb = e.iv1 = e.iv2 = -1;
    e.isTE = 0;

    newEdges.append(e);
}
```

Fase 3

Se crearán 4 triángulos nuevos, utilizando el vector de patrones y se añadirán a un vector temporal de triángulos. Como la información en esta etapa es incompleta, los punteros a aristas serán inicializados en -1.

```
QVector<Triangle> newTriangles;
for (int it(0); it < 4; it++)
{
    // Crearemos triángulos usando iVertexPattern[i],
    // iVertexPattern[(i + 1) % 4] y Centroide.
    Triangle t;
    t.iv1 = iVertexPattern.at(it);
```

```
    t.iv2 = iVertexPattern.at((it + 1) % 4);
    t.iv3 = iCentroid;
    t.ie1 = t.ie2 = t.ie3 = -1;
    t.bad = 0;

    newTriangles.append(t);
}
```

Fase 4

Se rescatan los índices de los 2 triángulos antiguos, y se reciclan sus posiciones, sobreescribiendo los triángulos originales. Adicionalmente, se guardan los 2 nuevos índices de los triángulos nuevos que quedan, y se añaden al vector original de triángulos.

```
// Indices de triángulos (A y B se reciclan)
QVector<int> newITriangles;

newITriangles.append(oldE.ita);
newITriangles.append(oldE.itb);
triangles.at(newITriangles.at(0)) = newTriangles.at(0);
triangles.at(newITriangles.at(1)) = newTriangles.at(1);

// Se agregan los nuevos que quedan disponibles
triangles.push_back(newTriangles.at(2));
newITriangles.append(triangles.size() - 1);
triangles.push_back(newTriangles.at(3));
newITriangles.append(triangles.size() - 1);
```

Fase 5

Se toman dos triángulos vecinos según el vector temporal *newTriangles*, y se asignarán a uno de las nuevas aristas. Si se estudia con detención el orden de los vértices en *iVertexPattern*, todos los triángulos del vector temporal son vecinos entre sí, ya sea en una orientación o en la otra. [11] De esta forma, el triángulo n y el triángulo $n+1$ siempre serán vecinos que comparten 1 de las aristas recién creadas.

Actualizaremos la información de esta arista, de forma que quede con información completa, pero los triángulos todavía tienen información incompleta en esta fase, puesto que hay que insertar las nuevas aristas en el vector original de aristas. [12]

[11] Los triángulos siguen el patrón NSC, SNC, NSC y SNC respectivamente, por lo que no hay forma de que queden enfrentados entre sí.

[12] Recordemos que al crear las aristas (cuando se carga el archivo), siempre el primer índice a vértice de la arista es menor (en número) al segundo índice. Por eso en el código se usa *min* y *max*.

```
for (int i(0); i < 4; i++)
{
    newEdges[i].ita = newITriangles.at(i);
    newEdges[i].itb = newITriangles.at((i + 1) % 4);
    newEdges[i].iv1 = std::min(newTriangles.at(i).iv2,
                               newTriangles.at(i).iv3);
    newEdges[i].iv2 = std::max(newTriangles.at(i).iv2,
                               newTriangles.at(i).iv3);
}
```

Todavía falta actualizar la información de las aristas no compartidas, por lo que se reemplazará el índice a triángulos de estas aristas, utilizando la información (actualizada) del vector temporal de índices *newITriangles*.

```
for (int ie : nonSharedIEdges)
{
    Edge &e(edges.at(ie));

    for (int it(0); it < 4; it++)
    {
        if (e.iv1 == std::min(triangles.at(newITriangles.at(it)).iv1,
                              triangles.at(newITriangles.at(it)).iv2)
        and e.iv2 == std::max(triangles.at(newITriangles.at(it)).iv1,
                              triangles.at(newITriangles.at(it)).iv2))
        {
            triangles.at(newITriangles.at(it)).ie3 = ie;
            if (e.ita == oldE.ita or e.ita == oldE.itb)
            {
                e.ita = newITriangles.at(it);
            }
            else if (e.itb == oldE.ita or e.itb == oldE.itb)
            {
                e.itb = newITriangles.at(it);
            }
        }
    }
}
```

Si nos fijamos en el fragmento que dice `triangles.at(newITriangles.at(it)).ie3 = ie;`, se puede ver que se garantiza el índice a arista de una de las aristas de los nuevos triángulos. Esto es correcto porque en la creación de un nuevo triángulo con patrón SNC o NSC, el centroide es siempre el último vértice. Puesto que los índices a aristas del triángulo

representan la arista opuesta, entonces la arista no compartida correspondiente siempre coincidirá con el tercer vértice del nuevo triángulo, que es el centroide.

Fase 6

Se crea un vector temporal de índices a las nuevas aristas, y se agrega el índice de la arista original (porque se reciclará). Posteriormente, se reemplaza la arista real, y se agregan los índices de las 3 aristas nuevas restantes, y se agregan las 3 aristas nuevas reales al vector original de aristas.

```
QVector<int> newIEdges;
newIEdges.append(iedge);
edges.at(newIEdges.at(0)) = newEdges.at(0);
edges.push_back(newEdges.at(1));
newIEdges.append(edges.size() - 1);
edges.push_back(newEdges.at(2));
newIEdges.append(edges.size() - 1);
edges.push_back(newEdges.at(3));
newIEdges.append(edges.size() - 1);
```

Fase 7

Finalmente, terminamos de actualizar la información de los nuevos triángulos [13].

Como se explicó al final de la Fase 5, en un triángulo hay un índice de arista ya garantizado, por lo que solamente queda actualizar el primer y el segundo índice a arista.

```
for (int ie : newIEdges)
{
    Edge &e(edges.at(ie));

    for (int it(0); it < 4; it++)
    {
        // Triángulo "it" (ie1)
        if (e.iv1 == std::min(triangles.at(newITriangles.at(it)).iv2,
                              triangles.at(newITriangles.at(it)).iv3) and
            e.iv2 == std::max(triangles.at(newITriangles.at(it)).iv2,
                              triangles.at(newITriangles.at(it)).iv3))
        {
            triangles.at(newITriangles.at(it)).ie1 = ie;
```

[13] Recordemos que al final de la Fase 1 el vector de vértices está actualizado, y al final de la Fase 6 el vector de aristas está actualizado.

```
        }

        // Triángulo "it" (ie2)
        if (e.iv1 == std::min(triangles.at(newITriangles.at(it)).iv1,
                              triangles.at(newITriangles.at(it)).iv3) and
            e.iv2 == std::max(triangles.at(newITriangles.at(it)).iv1,
                              triangles.at(newITriangles.at(it)).iv3))
        {
            triangles.at(newITriangles.at(it)).ie2 = ie;
        }
    }
}
```

Nuevamente se aprovecha del hecho de saber que numéricamente que el primer índice a vértice de una arista es menor que el del segundo, por lo que es posible encontrar los triángulos correspondientes eficazmente.

Mejoramiento completo (improveTriangulation)

Finalmente, en este método se unen los 3 métodos anteriores, donde se usa el *flag* del método *detectTerminalEdges* para saber si vale la pena continuar o no.

```
try
{
    // Flag que muestra si quedan aristas terminales que NO son de borde
    bool nonBTERemaining = false;

    detectTerminalEdges(vertices, edges, triangles, nonBTERemaining);
    if (not nonBTERemaining)
    {
        return true; // Finalización correcta
    }

    insertCentroids(vertices, edges, triangles);
    detectBadTriangles(m_angle, vertices, triangles);

    return true;
}
catch (std::exception &e)
{
    qWarning() << e.what();
    return false;
}
```

4.2.7. Motor de GPU

Aquí serán explicados en detalle cada uno de los pasos previos al procesamiento de una triangulación usando GPU, la forma de pasar los datos a la GPU, cómo se procesan estos datos y cómo se usan posteriormente.

Inicialización

En el host es necesario hacer varios pasos antes de poder usar la GPU. El primer paso es detectar las plataformas[14] y dispositivos[15] disponibles. Luego, se crea el contexto y la cola de comandos (*Command Queue*). Finalmente, es necesario cargar el kernel correspondiente, crear el programa de OpenCL y "construirlo" para el dispositivo que se está usando.

Detección de plataformas y dispositivos

A partir de un identificador numérico de la plataforma y dispositivo a usar, OpenCL identifica cuál de ellas usar. Para esta implementación se asume que solo hay un *driver* de OpenCL, por lo que se usa la primera plataforma y dispositivo que encuentra.

```
unsigned long platform_id(0);
unsigned long device_id(0);

cl::Platform::get(&m_platforms);
m_platforms.at(platform_id)
    .getDevices(CL_DEVICE_TYPE_GPU | CL_DEVICE_TYPE_CPU, &m_devices);
```

Creación de contexto y *command queue*

inmediatamente después, se crea el contexto correspondiente y la cola de comandos, donde se puede habilitar el *profiling* de OpenCL.

```
m_context = cl::Context(m_devices);
m_queue = cl::CommandQueue(m_context,
                           m_devices.at(device_id),
                           CL_QUEUE_PROFILING_ENABLE);
```

Carga de Kernel

Para cargar el kernel, es necesario tenerlo como un *string* inserto en el código o como un archivo externo que se pueda leer, por lo que se decidió tenerlo como archivo para reducir la interferencia con el código del host. Sin embargo, al dejarlo como un archivo suelto no es fácilmente posible detectar dónde está el archivo relativo al código. Para solucionar este

[14] La plataforma se refiere a la marca, como Intel, AMD o Nvidia.
[15] El dispositivo se refiere al modelo, como Nvidia GTX960M, Nvidia GT650, Intel Pro Graphics, etc.

problema, se decidió añadirlo como un "Recurso de Qt", que permite añadirlo como un recurso virtual, por lo que el kernel queda compilado dentro de la biblioteca, y deja de ser necesario acceder al archivo directamente.

```
QFile kernelfile(":/kernels/kernel.cl");
kernelfile.open(QIODevice::ReadOnly);
std::string kernel_code = kernelfile.readAll().toStdString();
kernelfile.close();
```

Como se puede ver, en lugar de tener "`../kernels/kernel.cl`", que depende de un *path* relativo, se compila el kernel directamente a la biblioteca, usando el *path* virtual "`:/`".

Para hacer que Qt sepa que se está usando un Recurso, es necesario tener un archivo especial con extensión **.qrc**, que se agrega a QLepp2DLib.pro para poder usarlo.

```
<RCC>
    <qresource prefix="/">
        <file>kernels/kernel.cl</file>
    </qresource>
</RCC>
```

Construcción del programa de OpenCL

Finalmente, se toma el código cargado del kernel y se introduce en el programa a crear, el cual está asociado al contexto que se generó anteriormente.

```
cl::Program::Sources source;
source.push_back({kernel_code.c_str(), kernel_code.length()});

m_program = cl::Program(m_context, source);
m_program.build(m_devices);
```

Detección de información

Aunque no es necesario en la inicialización, se decidió agregar código que permitiera detectar la información del dispositivo que está siendo ocupado actualmente para la aplicación de la implementación GPU.

Para ello, se extrae de la plataforma la información reference al nombre, empresa y versión de OpenCL.

```
m_platforms.at(platform_id).getInfo(CL_PLATFORM_NAME, &s);
m_platforms.at(platform_id).getInfo(CL_PLATFORM_VENDOR, &s);
m_platforms.at(platform_id).getInfo(CL_PLATFORM_VERSION, &s);
```

Adicionalmente, se extrae información del dispositivo que maneja la plataforma.

```
m_devices.at(device_id).getInfo(CL_DEVICE_NAME, &s);
m_devices.at(device_id).getInfo(CL_DEVICE_OPENCL_C_VERSION, &s);
m_devices.at(device_id).getInfo(CL_DEVICE_MAX_COMPUTE_UNITS, &i);
m_devices.at(device_id).getInfo(CL_DEVICE_LOCAL_MEM_SIZE, &size);
m_devices.at(device_id).getInfo(CL_DEVICE_GLOBAL_MEM_SIZE, &size);
m_devices.at(device_id).getInfo(CL_DEVICE_MAX_MEM_ALLOC_SIZE, &size);
m_devices.at(device_id).getInfo(CL_DEVICE_MAX_WORK_GROUP_SIZE, &size);
m_devices.at(device_id).getInfo(CL_DEVICE_MAX_WORK_ITEM_SIZES, &d);
```

Intercambio de información con la GPU

Para enviar información a la GPU es necesario crear *buffers* de OpenCL que contengan una copia de los datos de CPU. La forma más fácil de lograrlo es usando los iteradores de `std::vector`. De esta forma, se crean los buffers y se copian a éste automáticamente.

Desde aquí y para el resto de la explicación se tomará como ejemplo la función *detectBadTriangles*.

```
const bool USE_HOST_PTR(true);
m_bufferTriangles = cl::Buffer(m_context,
                               triangles.begin(),
                               triangles.end(),
                               false,
                               USE_HOST_PTR);
m_bufferVertices = cl::Buffer(m_context,
                              vertices.begin(),
                              vertices.end(),
                              false,
                              USE_HOST_PTR);
```

Luego, se crea el kernel específico para ser ejecutado por OpenCL. El nombre de la función en el kernel y los parámetros deben coincidir para que la ejecución pueda realizarse. En el ejemplo, se enviará el ángulo, el buffer de vértices y el buffer de triángulos.

```
cl::make_kernel<float&, cl::Buffer&, cl::Buffer&>
    detect_kernel(m_program, "detectBadTriangles");
```

Para poder regular la cantidad de *threads* que se deben usar para la ejecución de la función en GPU, se explicita el "tamaño global". OpenCL tiene la ventaja de que no es necesario explicitar el "tamaño local" de cada bloque[16], por lo que se puede omitir.

[16] Si se conoce el tamaño específico de los datos independientes entre sí, y este tamaño es igual para todos, es mejor especificarlo. Este no es el caso, lamentablemente.

```
// Se usan threads por triángulo
cl::NDRange global(triangles.size());
cl::EnqueueArgs eargs(m_queue, global);
```

Una vez rellenada la información, es posible ejecutar el kernel y recibir información de éste, como el tiempo de ejecución en GPU.

```
cl::Event event = detect_kernel(eargs,
                                angle,
                                m_bufferTriangles,
                                m_bufferVertices);
event.wait();

cl_ulong time_start(0);
cl_ulong time_end(0);
event.getProfilingInfo(CL_PROFILING_COMMAND_START, &time_start);
event.getProfilingInfo(CL_PROFILING_COMMAND_END, &time_end);
```

Finalmente, para recuperar la información desde la GPU, se copia explícitamente el resultado almacenado en los buffers de OpenCL hacia los vectores de información que mantiene el modelo.

```
cl::copy(m_queue, m_bufferTriangles, triangles.begin(), triangles.end());
```

Detección de triángulos malos y aristas terminales

A partir del diseño realizado para el motor de GPU (ver sección Diseño), la implementación de estos dos métodos es análogo al de CPU, con las siguientes diferencias:

- Eliminación del `for`.
- Posición en el vector dependiente del *thread*.
- Distinto acceso a un elemento de un vector.

La diferencia más evidente es que se eliminó el *for* que está contenido en la implementación CPU de *detectBadTriangles*, pues para este método los datos son independientes entre sí. Para el caso *detectTerminalEdges*, se permite la escritura concurrente, puesto que para dos listas Lepp distintas, si éstas llegan a detectar la misma arista terminal, ambas marcarán solamente la arista.

Para conseguir eliminar el *for*, se detecta la posición de un elemento en los vectores utilizando el *thread* actual en un ámbito global. Así, si por ejemplo existen 10.000 triángulos y los *threads* son por triángulo, habrán 10.000 *threads* en total.

```
int idx = get_global_id(0);
Triangle t = triangles[idx];
```

Finalmente, a partir del código anterior se puede verificar que el acceso a un elemento se hace directamente, en lugar de utilizar una funcion `at`, como se hace en CPU.

A modo de ejemplo, se expone el código del *kernel* para la detección de triángulos malos.

```
kernel void detectBadTriangles(const float angle,
                               global Triangle *triangles,
                               global Vertex *vertices)
{
    int idx = get_global_id(0);

    Vertex A, B, C;
    A = vertices[triangles[idx].iv1];
    B = vertices[triangles[idx].iv2];
    C = vertices[triangles[idx].iv3];

    float length_A = pown(B.x-C.x, 2) + pown(B.y-C.y, 2) + pown(B.z-C.z, 2);
    float length_B = pown(A.x-C.x, 2) + pown(A.y-C.y, 2) + pown(A.z-C.z, 2);
    float length_C = pown(A.x-B.x, 2) + pown(A.y-B.y, 2) + pown(A.z-B.z, 2);

    float length_a = sqrt(length_A);
    float length_b = sqrt(length_B);
    float length_c = sqrt(length_C);

    float angle_opp_A = acos((length_B + length_C - length_A)
                            / (2 * length_b * length_c));
    float angle_opp_B = acos((length_A + length_C - length_B)
                            / (2 * length_a * length_c));
    float angle_opp_C = acos((length_A + length_B - length_C)
                            / (2 * length_a * length_b));

    float rad_angle = angle * M_PI / 180.0;
    triangles[idx].bad = (angle_opp_A < rad_angle ||
                          angle_opp_B < rad_angle ||
                          angle_opp_C < rad_angle);
}
```

Inserción de centroides

Este es el único método de la implementación del motor de GPU que en realidad se hace en CPU, por lo que el algoritmo en su totalidad se considera mixto.

```
void OpenCLEngine::insertCentroids(std::vector<Vertex> &vertices,
                                   std::vector<Edge> &edges,
                                   std::vector<Triangle> &triangles)
{
    CPUEngine cpuengine;
    cpuengine.insertCentroids(vertices, edges, triangles);

    // Se actualiza la información en GPU para
    // evitar inconsistencias.
    const bool USE_HOST_PTR = true;
    m_bufferEdges = cl::Buffer(m_context,
                               edges.begin(),
                               edges.end(),
                               false,
                               USE_HOST_PTR);
}
```

Como se observa en el código, se necesita actualizar el buffer de aristas para que cuando se llame nuevamente al método *detectBadTriangles*, éste solo necesite encargarse de actualizar los buffers de vértices y triángulos.

4.2.8. API

La API privada de la biblioteca posee los siguientes métodos:

```
public:
    static ModelImpl& getInstance();
    void setEngine(Engine *engine);
    bool setCPUEngine();
    bool setOpenCLEngine();

    bool loadFile(std::string filepath);
    bool saveFile(std::string filepath);

    std::vector<Vertex>& getVertices();
    std::vector<Edge>& getEdges();
    std::vector<Triangle>& getTriangles();

    bool detectBadTriangles(double angle);
    bool improveTriangulation();

private:
    ModelImpl();

private:
    FileManager m_fileManager;
    Engine *m_engine;
    std::vector<Vertex> m_vertices;
    std::vector<Edge> m_edges;
    std::vector<Triangle> m_triangles;
```

ModelImpl es la clase *façade* de la biblioteca de la aplicación, que ofrece sus funcionalidades sin entrar en detalles de cada componente. De esta forma, de forma simple es posible:

- Obtener la instancia (única) de la biblioteca.
- Intercambiar los motores.
- Cargar un archivo.
- Guardar un archivo.
- Obtener los vectores de información.
- Detectar los triángulos malos.
- Mejorar la triangulación.

Se puede observar que, salvo la obtención de la instancia y de los vectores, todos los métodos devuelven un *bool*. Esto se hace para verificar que el método haya hecho su función correctamente, o permitir que el desarrollador tenga la libertad de definir el mensaje de error que desee en caso de que el método no haya podido ejecutarse como corresponde.

Adicionalmente, para poder ofrecer estas funcionalidades sin exponer los detalles de implementación, utilizando el patrón de diseño de software *Bridge* se puede ofrecer esta API pública al desarrollador:

```
public:
    Model();
    bool setCPUEngine();
    bool setOpenCLEngine();

    bool loadFile(std::string filepath);
    bool saveFile(std::string filepath);

    std::vector<Vertex>& getVertices();
    std::vector<Edge>& getEdges();
    std::vector<Triangle>& getTriangles();

    bool detectBadTriangles(double angle);
    bool improveTriangulation();

private:
    ModelImpl *m_impl;
```

Estos métodos ofrecen lo mismo que la clase *façade* ModelImpl, pero se pueden observar dos diferencias importantes entre la API privada y la API pública. Primero, esta API pública no muestra las variables privadas que hacen funcionar la aplicación, tales como *FileManager*, *Engine* o los vectores que poseen la información. Segundo, que ofrece un constructor público, a pesar de que la API privada solo ofrece el método *getInstance*. Se decidió hacerlo así para simplificar el uso de la API a un desarrollador, de forma de que no tenga que preocuparse de llamar directamente a *getInstance*, y aún así poder beneficiarse del patrón *Singleton*.

4.3. Interfaz Gráfica

Para poder implementar la interfaz se utilizó el template *app* que usa qmake y se agregó la biblioteca *Widgets* que ofrece Qt, escribiendo estas opciones en el archivo QLepp2DGui.pro, que maneja qmake.

En el archivo .pro se definió el nombre del ejecutable de la aplicación, los archivos de diseño que fueron usados, las ubicaciones de las traducciones de la interfaz, la forma de cargar las bibliotecas, tanto las internas (la biblioteca de esta misma aplicación) como externas, que dependen del sistema operativo[17], y la forma de instalar la aplicación en el sistema.

Adicionalmente, para poder diseñar la vista de las ventanas a usar de forma rápida y eficiente, se utilizó Qt Designer, el cual crea archivos con extensión *.ui*[18], los cuales son incluídos posteriormente en la aplicación en tiempo de compilación.

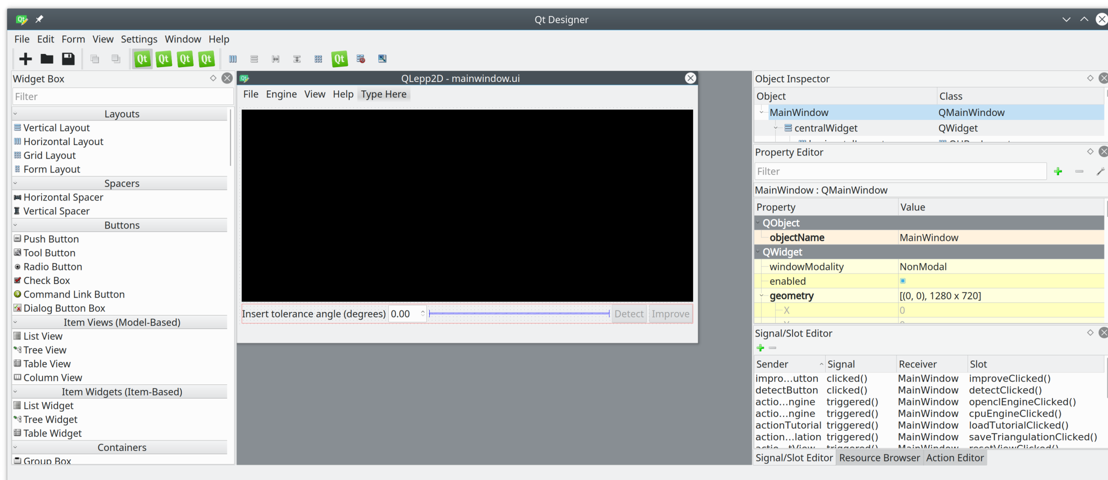

Figura 4.3: Diseño de la interfaz con Qt Designer.

Para poder incorporar los archivos *.ui* a la aplicación, qmake ofrece agregar los nombres de los archivos en el archivo *.pro*. De esta forma, Qt puede convertirlo a código C++ y compilarlo automáticamente, usando el *approach* de puntero-a-implementación de forma similar a usar el patrón de diseño de software *Bridge*.

Si bien es posible programar cada uno de los botones y menús por separado, el uso de Qt Designer simplifica el proceso de diseño de la interfaz del código usado para manejar las acciones.

[17]Por ejemplo, las ubicaciones de OpenCL son distintas en Mac que en Linux.
[18]Estos archivos internamente son archivos XML creados por Qt Designer.

Finalmente, para poder controlar el Modelo, éste se inicializa en el archivo principal *main.cpp* y se pasa a la ventana principal como argumento al momento de crearla.

```
Model *model(new Model);
model->setCPUEngine();

MainWindow window(model);

window.show();
```

4.3.1. Ventana principal

En la ventana principal (*MainWindow*) están todas las conexiones que se hacen hacia la biblioteca al momento de recibir un evento de parte del usuario. Esto es, recibir los *clicks* hechos a botones, *toolboxes* y menús. Dado el diseño de la biblioteca, es posible conectar estos elementos usando su API, por lo que solo se necesita implementar lógica de procesamiento para el manejo de archivos y de configuraciones.

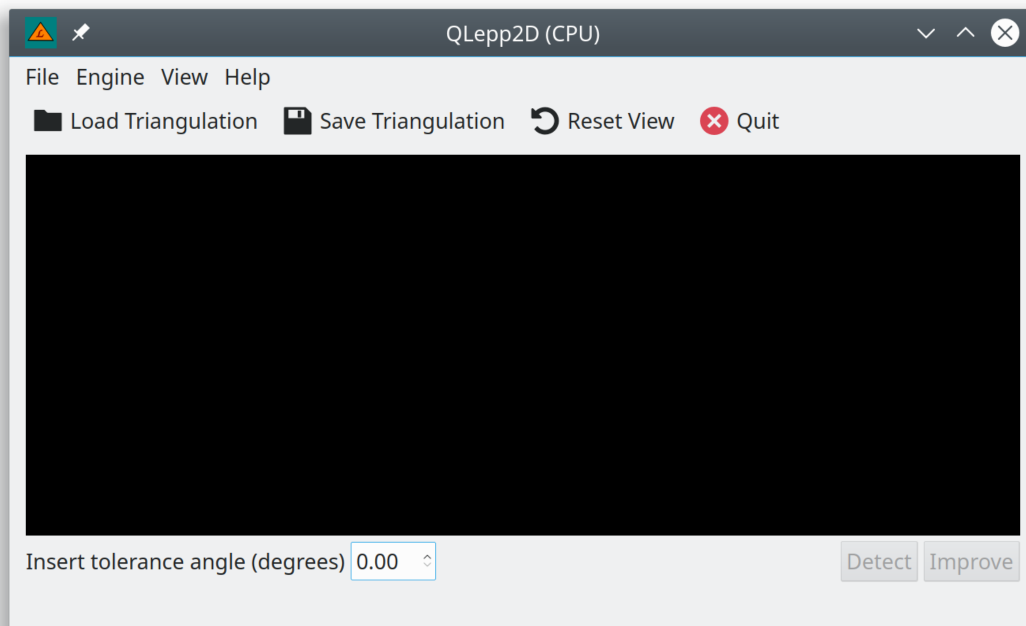

Figura 4.4: Interfaz de la aplicación.

El manejo de eventos del usuario se maneja con el siguiente código:

```
private slots:
    void loadTriangulationClicked();
```

```
void saveTriangulationClicked();
void resetViewClicked();
void loadTutorialClicked();
void loadAboutClicked();
void loadAboutQtClicked();
void detectClicked();
void improveClicked();
void cpuEngineClicked();
void openclEngineClicked();
```

Cada uno de estos métodos es un *slot* en el Framework de Qt. De esta forma, cada vez que un botón (o menú) es clickeado, éste envía una señal *clicked*, la cual está conectada a uno de los *slots* que se muestran en este código.

Gracias a Qt Designer, las conexiones de *signals* a *slots* se pueden hacer usando su interfaz gráfica, por lo que cuando *qmake* realiza la conversión automática de diseño a código, se crea el siguiente código, el cual conecta las señales con los *slots*:

```
QObject::connect(actionSaveTriangulation,
                SIGNAL(triggered()),
                MainWindow,
                SLOT(saveTriangulationClicked()));
QObject::connect(actionTutorial,
                SIGNAL(triggered()),
                MainWindow,
                SLOT(loadTutorialClicked()));
QObject::connect(actionAbout,
                SIGNAL(triggered()),
                MainWindow,
                SLOT(loadAboutClicked()));
(...)
QMetaObject::connectSlotsByName(MainWindow);
```

La interfaz habilita y deshabilita botones dependiendo del estado actual de la aplicación, de tal forma que, por ejemplo, no se permite mejorar una malla cuyos triángulos malos no han sido detectados aún. Por ejemplo:

```
ui->improveButton->setDisabled(true);
ui->detectButton->setEnabled(true);
```

Además, se limita el ángulo que se puede ingresar desde la interfaz al rango de 0° a 60°, dado que es incoherente ingresar otro rango. El limitador fue incorporado al diseño de la aplicación con Qt Designer, y el código que se crea es el siguiente:

```
angleSpinBox = new QDoubleSpinBox(centralWidget);
angleSpinBox->setObjectName(QStringLiteral("angleSpinBox"));
angleSpinBox->setMaximum(60);
```

4.3.2. Menús

Para que la interfaz de la aplicación tuviera menús, éstos se agregaron al diseño hecho en Qt Designer, por lo que el código generado es el siguiente:

```
menuBar = new QMenuBar(MainWindow);
menuBar->setObjectName(QStringLiteral("menuBar"));
menuBar->setGeometry(QRect(0, 0, 1280, 49));
menuFile = new QMenu(menuBar);
menuFile->setObjectName(QStringLiteral("menuFile"));
menuFile->addAction(actionLoadTriangulation);
menuFile->addAction(menuRecentTriangulations->menuAction());
menuFile->addSeparator();
menuFile->addAction(actionSaveTriangulation);
menuBar->addAction(menuFile->menuAction());

QObject::connect(actionLoadTriangulation, SIGNAL(triggered()),
                MainWindow, SLOT(loadTriangulationClicked()));
QObject::connect(actionSaveTriangulation, SIGNAL(triggered()),
                MainWindow, SLOT(saveTriangulationClicked()));
(...)
```

Para conectarlos a la funcionalidad ofrecida por la biblioteca, se adaptó el sistema de *signal / slot* de Qt, de forma de que cada menú que emitiera la señal *clicked* tuviera su *slot* correspondiente en el código de MainWindow.

```
// Ejemplo de slot cuando el botón Acerca de... recibe una señal clicked()
void MainWindow::loadAboutClicked()
{
    qDebug() << "Load About button clicked";
    m_about->show();
}
```

4.3.3. Carga y guardado

Para poder cargar y guardar archivos usando la interfaz, se utilizó la utilidad QFileDialog ofrecida por Qt, que permite mostrar un diálogo de cargado o guardado de archivos, y obtiene el *path* absoluto del archivo. Luego, se entrega este *path* al modelo, que carga (o guarda)

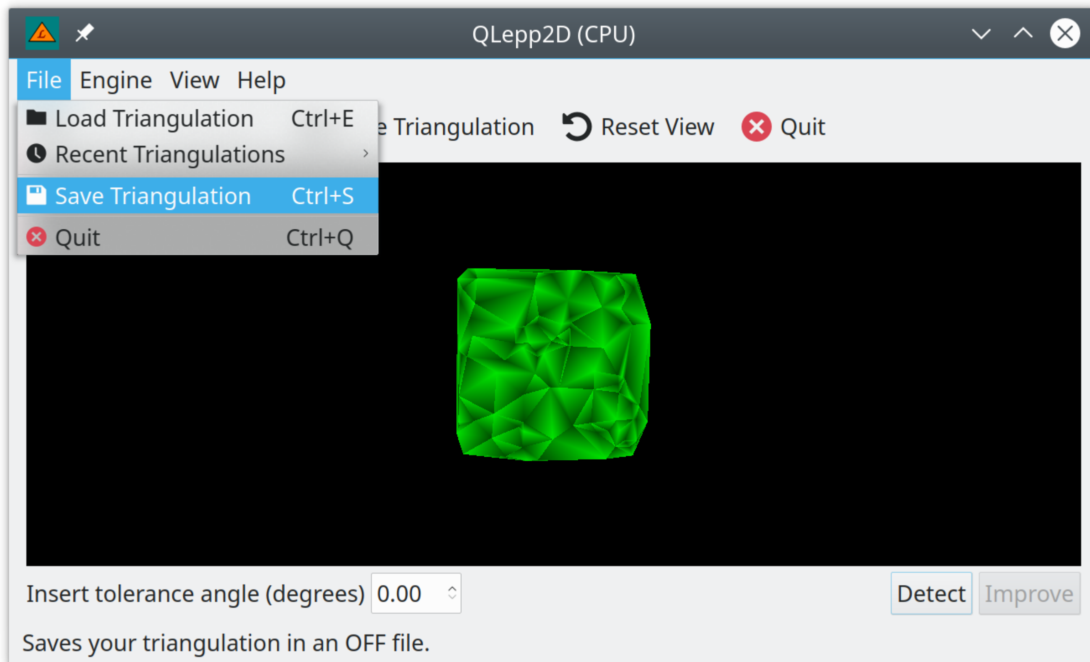

Figura 4.5: Ejemplo de menú.

el archivo correspondiente. En el caso de la carga, *MainWindow* emite además una señal *emitUpdateData* que recibe el *QWidget* encargado de renderizar la malla usando OpenGL.

```
// Ejemplo para loadFile()
void MainWindow::loadTriangulationClicked()
{
    QString filepath = QFileDialog::getOpenFileName(
                        this,
                        tr("OFF files"),
                        m_settings->value("lastDir", ".").toString(),
                        tr("OFF Files (*.off)"));
    loadFile(filepath);
}

void MainWindow::loadFile(QString path)
{
    ui->statusBar->showMessage(tr("Loading..."));
    std::string spath = path.toStdString();
    if (m_model->loadFile(spath))
    {
        QFileInfo fileinfo(path);
        m_currentFileName = fileinfo.completeBaseName();
```

```
        addRecentFile(path);

        qDebug() << m_currentFileName << "triangulation loaded." << endl;
        ui->statusBar->showMessage(tr("Loaded."));
        ui->improveButton->setDisabled(true);
        ui->detectButton->setEnabled(true);

        emit emitUpdateData();
    }
    else
    {
        qWarning() << "Could not open file" << path;
        removeRecentFile(path);
        ui->statusBar->showMessage(tr("Unable to load."));
    }
    updateRecentFiles();
}
```

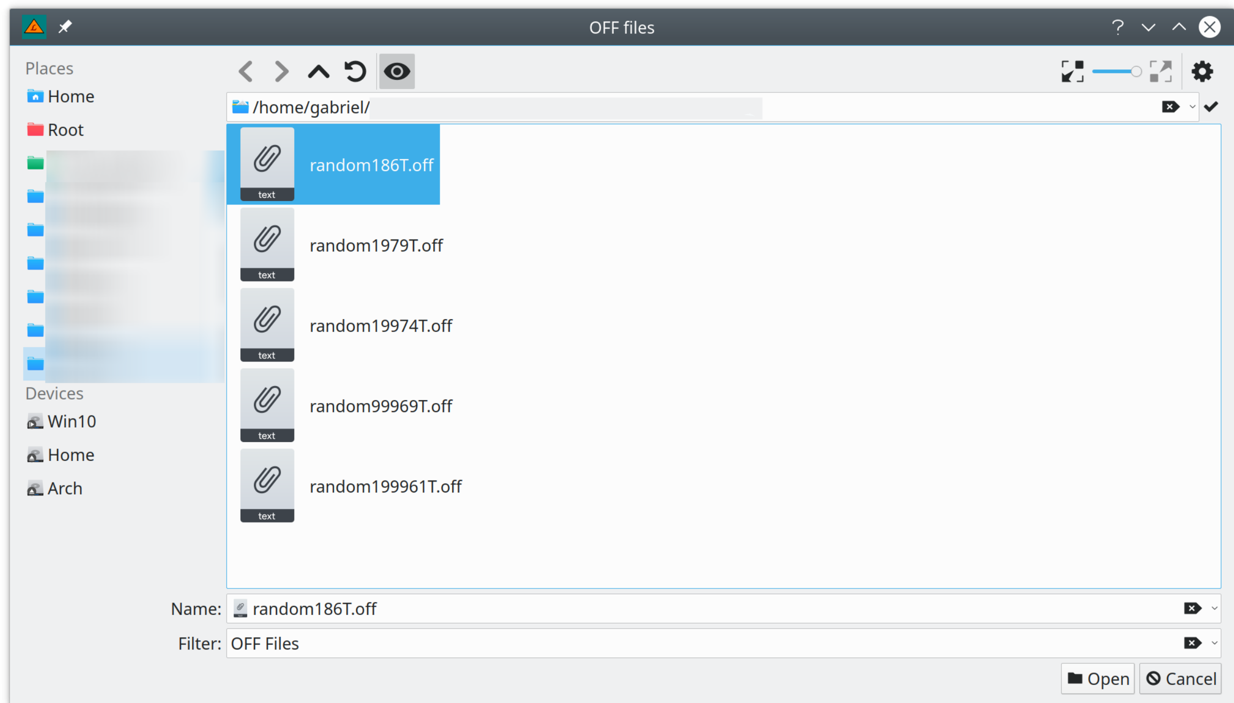

Figura 4.6: Cargar archivo OFF.

4.3.4. Renderizado con OpenGL

Qt ofrece un *widget* para trabajar de forma menos engorrosa con OpenGL, llamado *QOpenGLWidget*, el cual funciona como puente (*bridge*) entre Qt y OpenGL. Se heredó de este widget y se extendió para utilizarlo específicamente para la aplicación.

Este widget, que llamaremos *OpenGLWidget*, está inserto en MainWindow, pero a diferencia de los otros widgets (ej., botones y menús), se decidió que la interacción entre MainWindow y OpenGLWidget no sería directa, sino que se utilizaría el sistema de *signals* y *slots* de Qt, lo que permite reemplazar OpenGLWidget por otro widget, sin cambiar el código de MainWindow. La utilización de este sistema será explicada en el siguiente párrafo.

Intercambio de señales

Para poder renderizar la malla de triángulos, OpenGLWidget necesita tener información de los vértices y los triángulos que contiene el modelo. [19]
La forma más fácil de trabajar sin tener que enviar la información a cada momento es la de permitir que OpenGLWidget tenga la dirección de memoria del modelo, por lo que al momento de crear la interfaz, el constructor de MainWindow emite una señal llamada `emitUpdateModel(Model *)`, que recibe OpenGLWidget, donde el parámetro enviado es la dirección de memoria del modelo. De esta forma, OpenGLWidget tiene acceso al modelo sin que MainWindow tenga que controlar directamente a OpenGLWidget.

Adicionalmente, puesto que es MainWindow quien recibe las acciones de los botones y menús, envía una señal `emitUpdateData()` a OpenGLWidget cada vez que hay que redibujar, por lo que OpenGLWidget toma los vértices y triángulos del modelo (usando la API) y redibuja la triangulación.

Inicialización

OpenGLWidget requiere de 3 metodos básicos para trabajar (los cuales se heredan de QOpenGLWidget).

```
void initializeGL() override;
void resizeGL(int w, int h) override;
void paintGL() override;
```

El método *initializeGL* se encarga de inicializar las funciones de OpenGL (recordemos que QOpenGLWidget es un *bridge*) y de crear el programa usado por OpenGL.

[19] Recordemos que la biblioteca tiene una API pública, la que será usada aquí.

```
void OpenGLWidget::initializeGL()
{
    initializeOpenGLFunctions();
    generateGLProgram();
}
```

Dentro de las funciones más importantes de *generateGLProgram()* están el cargar y conectar los *shaders* que muestran y colorean la malla, y conectarlos al programa de OpenGL.

```
m_program = new QOpenGLShaderProgram;
m_program->addShaderFromSourceFile(QOpenGLShader::Vertex,
                                   ":/shaders/vertex.glsl");
m_program->addShaderFromSourceFile(QOpenGLShader::Fragment,
                                   ":/shaders/fragment.glsl");
m_program->bindAttributeLocation("vertex", 0);
m_program->bindAttributeLocation("color", 1);
m_program->link();
(...)
```

Posteriormente, se llama al método *paintGL()*, el cual se encarga de limpiar la pantalla, ajustar las posiciones de modelo (*world*), cargar los datos del modelo y dibujar la triangulación.

Para hacer más eficiente la carga de datos, en lugar de dibujarlos a cada momento, se dibujan solamente si OpenGLWidget recibe una señal que accione el slot *updateData()*. (La carga de datos será explicada en detalle en la siguiente parte.)

```
// Se cargan datos solo si se modificó el modelo
if (not m_dataAlreadyLoaded)
{
    // Carga de datos
    loadData();
    setupVertexAttribs();
    m_dataAlreadyLoaded = true;
}

glDrawArrays(GL_TRIANGLES,
             0,
             3 * static_cast<int>(m_model->getTriangles().size()));

void OpenGLWidget::updateData()
{
    m_dataAlreadyLoaded = false;
```

```
    update(); // Llama automáticamente a paintGL()
}
```

Finalmente, *resizeGL()* se usa solamente si la ventana (y por lo tanto, OpenGLWidget) cambian de forma.

```
void OpenGLWidget::resizeGL(int w, int h)
{
    m_proj.setToIdentity();
    m_proj.perspective(45.0f, GLfloat(w) / h, 0.01f, 10000.0f);
}
```

Renderizado de la triangulación

Para dibujar la triangulación que está actualmente en el modelo, se requieren obtener los vértices y los triángulos, lo que se puede lograr simplemente usando la API.

```
std::vector<Vertex> &vertices(m_model->getVertices());
std::vector<Triangle> &triangles(m_model->getTriangles());
```

Pero antes de renderizar la triangulación, es necesario conectar la posición de los vértices y triángulos que se cargan, con la posición que OpenGL espera de cada uno de estos atributos. Para ello,se hacen calzar las posiciones ya declaradas en *generateGLProgram()* con las posiciones de los atributos.

```
m_vbo.bind();
QOpenGLFunctions *f = QOpenGLContext::currentContext()->functions();
f->glEnableVertexAttribArray(0);                        // Vertex
f->glEnableVertexAttribArray(1);                        // Color
f->glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, nullptr);
f->glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 0,
    reinterpret_cast<void *>(3 *
                             sizeof(Vertex) *
                             m_model->getTriangles().size()));
m_vbo.release();
```

Puesto que la información finalmente llegará al VBO[20], se decidió que primero serían cargadas todas las posiciones de los vértices, donde cada 3 posiciones habría un nuevo vértice.

```
VBO = ([x][y][z][x][y][z]...)
```

[20] Vertex Buffer Object

Luego, la información referente a los colores estaría inmediatamente después de cargar todos los vértices en el VBO. (Ver en el código anterior la sección de *reinterpret_cast*).

```
VBO = (...[x][y][z][c1][c2][c3]...)
```

Por lo tanto, para rellenar los vértices en el VBO, se escanean uno por uno y se escriben sus coordenadas por separado.

```
for (Triangle &t : triangles)
{
    unsigned long t_iv1(static_cast<unsigned long>(t.iv1));
    unsigned long t_iv2(static_cast<unsigned long>(t.iv2));
    unsigned long t_iv3(static_cast<unsigned long>(t.iv3));

    vertexData.append(vertices.at(t_iv1).x);
    vertexData.append(vertices.at(t_iv1).y);
    vertexData.append(vertices.at(t_iv1).z);

    vertexData.append(vertices.at(t_iv2).x);
    vertexData.append(vertices.at(t_iv2).y);
    vertexData.append(vertices.at(t_iv2).z);

    vertexData.append(vertices.at(t_iv3).x);
    vertexData.append(vertices.at(t_iv3).y);
    vertexData.append(vertices.at(t_iv3).z);
}
```

Dado que OpenGLWidget se diseñó sin un *geometry shader*, para poder diferenciar los triángulos entre sí, se creó un sistema simple que pudiera garantizar que los triángulos fueran discernibles, al menos en 2 de sus 3 vértices.

```
// Generamos el color
for (Triangle &t : triangles)
{
    for (int i(1); i <= 3; i++)
    {
        vertexData.append( t.bad * 0.3f * i);
        vertexData.append(!t.bad * 0.3f * i);
        vertexData.append(0.0);
    }
}
```

Como se puede ver en el código, el color de cada vértice es ligeramente distinto entre sí, y dado que los triángulos tienen vértices referenciados como punteros en forma de número, no existe posibilidad de colisiones múltiples que impidan diferenciar entre triángulos.

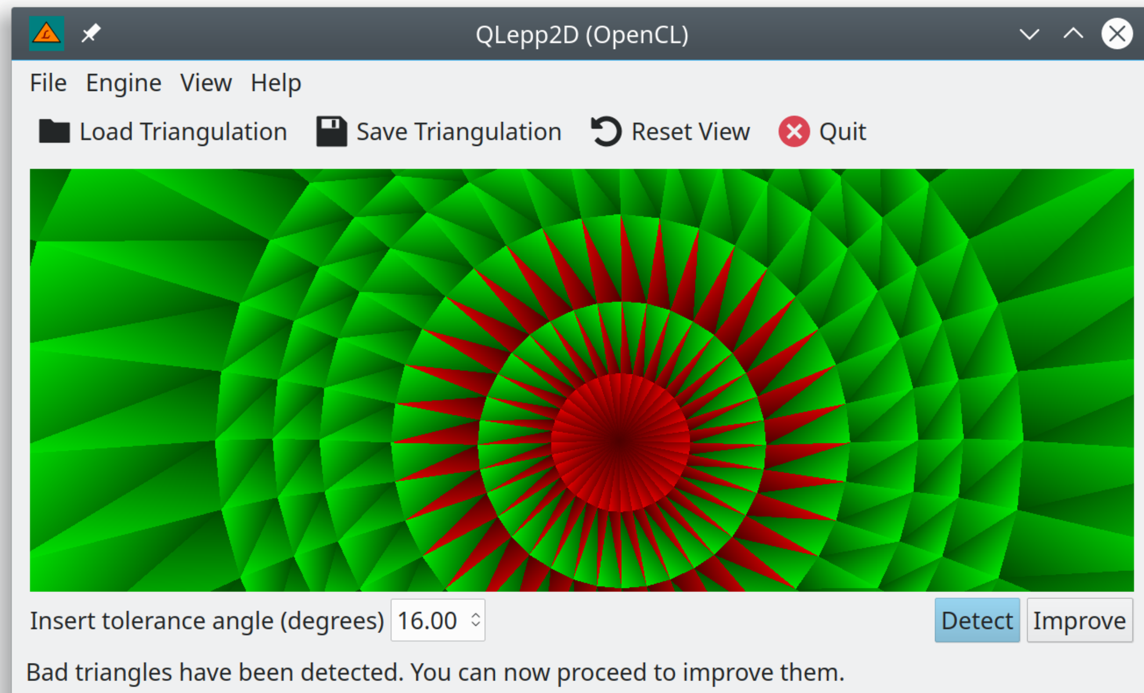

Figura 4.7: Malla cargada.

Shaders

Dado que los colores ya fueron diseñados en la carga de datos a OpenGLWidget, los shaders simplemente cargan los puntos y los dibujan con el color creado.

```
// Vertex shader
#version 120

attribute vec3 vertex;
attribute vec3 color;

varying vec3 f_color;

uniform mat4 projMatrix;
uniform mat4 modelViewMatrix;
```

```
void main(){
    f_color = color;
    gl_Position = projMatrix * modelViewMatrix * vec4(vertex, 1.0);
}

// Fragment Shader
#version 120

varying highp vec3 f_color;

void main() {
    gl_FragColor = vec4(f_color, 0.0);
}
```

Zoom, rotación y traslación

El zoom, la rotación y la traslación son señales que OpenGLWidget recibe directamente, por lo que basta filtrar las señales para saber cuál de las tres opciones aplicar.

Puesto que estas señales se emiten al "mover" la pantalla de OpenGLWidget, se decidió que cada botón tendría una respuesta distinta al ser accionado.

Zoom

El zoom se modifica al poner el cursor sobre la triangulación y mover la rueda del mouse hacia adelante o atrás, y se implementa de la siguiente manera.

```
void OpenGLWidget::wheelEvent(QWheelEvent *event)
{
    m_zCamPos += (event->delta() / 120);
    m_camera.setToIdentity();
    m_camera.translate(-m_xCamPos, m_yCamPos, m_zCamPos);
    update();
}
```

Aquí, se detecta el cambio correspondiente al movimiento de la rueda del mouse, y como `event->delta()` puede ser positivo o negativo, es posible aumentar y disminuir el zoom directamente.

Rotación

La rotación se detecta a partir del botón izquierdo y derecho del mouse, por lo que es necesario detectar la posición del momento en el que se hace click en la pantalla.

```
void OpenGLWidget::mousePressEvent(QMouseEvent *event)
{
    m_lastPos = event->pos();
}
```

Luego, se detecta la diferencia entre la posición recién detectada con la posición hacia donde se dirige el mouse. El botón izquierdo se encarga de la rotación en los ejes X e Y, y el derecho entre los ejes X y Z.

```
void OpenGLWidget::mouseMoveEvent(QMouseEvent *event)
{
    int dx = event->x() - m_lastPos.x();
    int dy = event->y() - m_lastPos.y();

    if (event->buttons() & Qt::LeftButton)
    {
        setXRotation(m_xRot + 8 * dy);
        setYRotation(m_yRot + 8 * dx);
    }
    else if (event->buttons() & Qt::MiddleButton)
    {
        setXMovement(m_xCamPos + (static_cast<float>(dx) / 200.0f));
        setYMovement(m_yCamPos - (static_cast<float>(dy) / 200.0f));
    }
    else if (event->buttons() & Qt::RightButton)
    {
        setXRotation(m_xRot + 8 * dy);
        setZRotation(m_zRot + 8 * dx);
    }
    m_lastPos = event->pos();
}
```

A modo de ejemplo, se muestra el método *setXRotation*. Los otros dos métodos son análogos.

```
void OpenGLWidget::setXRotation(int angle)
{
    qNormalizeAngle(angle);
```

```
    if (angle != m_xRot)
    {
        m_xRot = angle;
        update();
    }
}
```

Traslación

La traslación se detecta a partir del botón medio del mouse, por lo que, al igual que con la rotación, es necesario detectar la posición del momento en el que se hace click en la pantalla[21].

La diferencia está en que se mueve la cámara en lugar del modelo, a fin de simular movimiento de la triangulación. A modo de ejemplo, se muestra el código usado para la traslación en el eje X. (El eje Y es análogo).

```
void OpenGLWidget::setXMovement(float position)
{
    if (qAbs(position - m_xCamPos) > 0.01f)
    {
        m_xCamPos = position;
        m_camera.setToIdentity();
        m_camera.translate(m_xCamPos, m_yCamPos, m_zCamPos);
        update();
    }
}
```

Restaurar Vista

Se puede restaurar la vista al estado inicial presionando el botón "Restaurar Vista" de la interfaz. Puesto que esta señal es recibida por MainWindow, OpenGLWidget la recibe en forma de señal emitida por MainWindow.

```
void MainWindow::resetViewClicked()
{
    qDebug() << "Reset View button clicked";
    emit resetView();
}
```

[21] El código utilizado es el mismo que en la sección de rotación, pero solo tomando en cuenta la sección `Qt::MiddleButton`.

Cuando OpenGLWidget la recibe, reinicia las posiciones del modelo, la cámara, mueve la camara y vuelve a actualizar la imagen que está renderizando OpenGL.

```
void OpenGLWidget::resetView()
{
    m_xRot = m_yRot = m_zRot = 0;
    m_xCamPos = m_yCamPos = 0;
    m_zCamPos = -5;
    m_camera.setToIdentity();
    m_camera.translate(-m_xCamPos, m_yCamPos, m_zCamPos);
    update();
}
```

4.3.5. Tutorial

La ventana de Tutorial fue creada para facilitar la curva de aprendizaje del usuario al momento de usar la aplicación, y como conveniencia para que no se viera obligado a buscar el *README* que trae la aplicación.

Por otro lado, se ocupó como entorno de pruebas para incluir internacionalización en la aplicación, por lo que si, por ejemplo, el usuario ejecuta la aplicación en un sistema en Español en lugar de Inglés, verá este tutorial en Español.

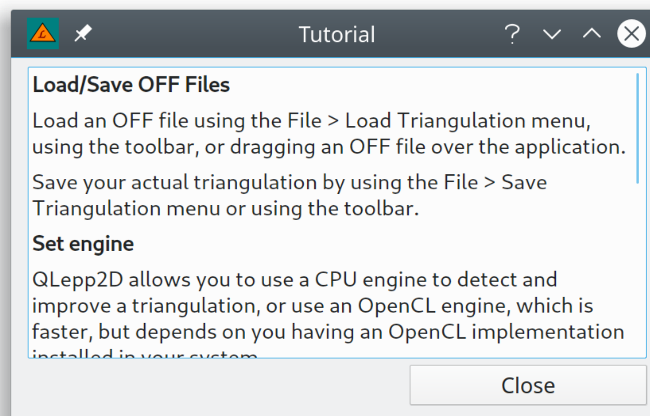

Figura 4.8: Ventana Tutorial.

4.3.6. Acerca de

La ventana “Acerca de” muestra la versión de la aplicación, al autor y la licencia (GPL) con la cual se puede utilizar esta aplicación.

4.3.7. Acerca de Qt

La ventana "Acerca de Qt" muestra las características del framework Qt, junto con cumplir con la licencia de Qt, que pide mostrar su uso en caso de usar la versión de código abierto.

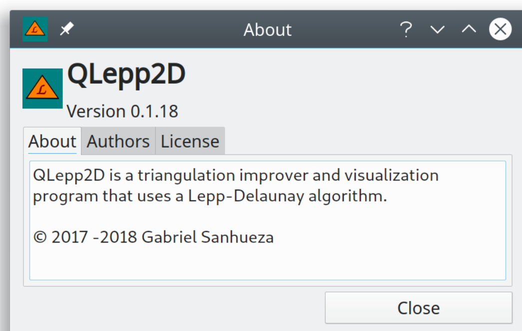

Figura 4.9: Ventana Acerca de.

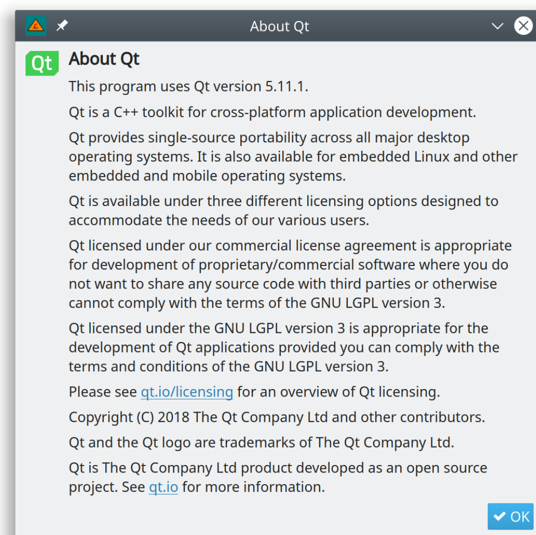

Figura 4.10: Ventana Acerca de Qt.

4.3.8. Internacionalización

Qt provee un sistema de internacionalización[22] llamado Qt Linguist, el cual es una aplicación que lee archivos con extensión *.ts*, los cuales llevan punteros a las líneas de código que se pueden traducir.

Para poder crear este archivo TS, se requiere correr el comando `lupdate QLepp2DGui.pro`. De esta forma, se obtienen los *strings* a traducir. Estos *strings* son declarados de forma especial en el código, para poder ser marcados:

```
// Ejemplo de internacionalización
ui->statusBar->showMessage(tr("OpenCL Engine has been set."));
```

Como se puede ver en este código, el *string* está rodeado por una función *tr*. Esta es la función que marca los *strings* que captura el comando `lupdate`.

Por otro lado, Qt Linguist es capaz de crear un archivo compilado con las traducciones que usa la aplicación, con `lrelease qlepp2d_es.ts -qm res/i18n/qlepp2d_es.qm`.

[22]Se refiere a que es posible cambiar el idioma de la aplicación dependiendo del idioma del sistema operativo.

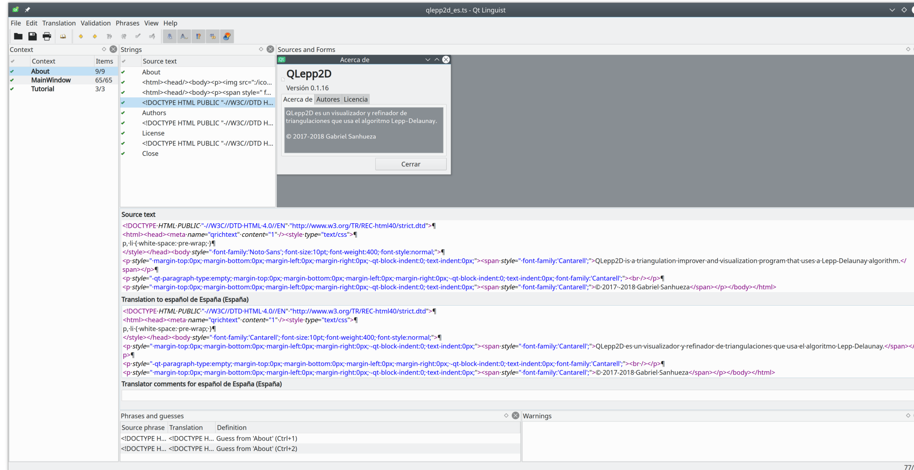

Figura 4.11: Qt Linguist traduciendo QLepp2D.

Finalmente, para poder instalar el traductor, se modifica el archivo *main.cpp* para incluir el siguiente código:

```
QApplication app(argc, argv);

QTranslator translator;
translator.load(QString("qlepp2d_%1")
            .arg(QLocale::system().name().split("_").at(0)), ":/i18n/");
app.installTranslator(&translator);
```

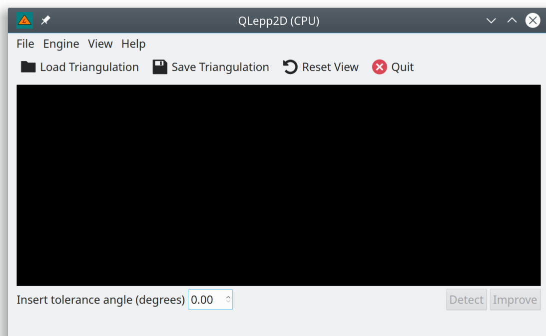

Figura 4.12: Ventana en Inglés.

Figura 4.13: Ventana en Español.

4.3.9. Keybindings

Para incluir *keybindings*[23] en la aplicación, se puede usar Qt Designer para editar las acciones de cada menú, e incluir el *keybinding* deseado.

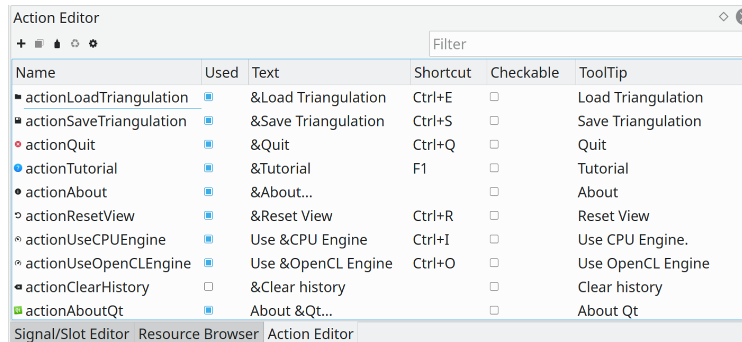

Figura 4.14: *Keybindings* vía Qt Designer.

Alternativamente, para las ventanas Tutorial y Acerca de, se añadieron los *keybindings* manualmente:

```
// Keybinding para ventana Acerca de...
void About::keyPressEvent(QKeyEvent *e)
{
    if (e->key() == Qt::Key_Escape)
    {
        close();
    }
}
```

4.3.10. Drag-and-Drop

Para poder habilitar *drag-and-drop* en la aplicación, se hizo un *override* de los métodos de Qt *dragEnterEvent(QDragEnterEvent)* y *dropEvent(QDropEvent)*. De esta forma, se hace posible aceptar que se pueda arrastrar un archivo a la aplicación para poder cargarlo de forma directa, al igual que si se hubiese cargado usando el menú o la barra de herramientas.

Adicionalmente, la aplicación puede filtrar el tipo de archivos usando su información *MIME*. Para este caso, es posible cargar cualquier archivo de texto plano, y la biblioteca se encarga de verificar si realmente es un archivo que puede manejar o no.

[23] Se los conoce como “Accesos directos de teclado”.

```
void MainWindow::dragEnterEvent(QDragEnterEvent *event)
{
    if (event->mimeData()->hasFormat("text/plain"))
    {
        event->acceptProposedAction();
    }
}
void MainWindow::dropEvent(QDropEvent *event)
{
    QString filepath = event->mimeData()->urls().at(0).toLocalFile();
    loadFile(filepath);
}
```

4.3.11. Historial

Para incluir un historial de archivos recientes a la aplicación se hizo uso de QSettings, el cual utiliza distintas formas de guardar información dependiendo del sistema operativo. En el caso de Linux, QSettings genera un archivo personalizado, donde se guarda la lista de archivos recientes y el último directorio accedido, utilizando un sistema parecido al de QMap.

```
(...)
m_settings = new QSettings("QLepp2D", "qlepp2d", this);
(...)
void MainWindow::readSettings()
{
    // Lee archivos recientes de la última sesión
    m_recentFilesList = m_settings->value("recentFiles", "").toStringList();
}

void MainWindow::writeSettings()
{
    // Guarda los últimos archivos recientes antes de cerrar
    m_settings->setValue("recentFiles", m_recentFilesList);
}

void MainWindow::addRecentFile(QString path)
{
    QFileInfo info(path);

    m_settings->setValue("lastDir", info.absoluteDir().absolutePath());
    // Limpieza de duplicados
    removeRecentFile(path);
```

```
    m_recentFilesList.push_front(path);

    while (m_recentFilesList.size() > m_recentFilesLimit)
    {
        m_recentFilesList.removeLast();
    }
}
(...)
```

Como se ve en el código, *readSettings* y *writeSettings* leen y escriben información de los archivos abiertos anteriormente, mientras que *addRecentFile* obtiene la información del archivo abierto, elimina duplicados (i.e., abierto más de una vez en la misma sesión), lo agrega a la lista, y al final se reduce la lista para no tener exceso de archivos abiertos recientemente.

Para poder mostrarlos en la interfaz, se crean manualmente acciones asociadas a cada archivo, para posteriormente mostrarlos en el menú de Archivo ¿Triangulaciones recientes:

```
for (int i(0); i < m_recentFilesList.size(); i++)
{
    QString recentFile(m_recentFilesList.at(i));
    QAction *recentFileAction = ui->menuRecentTriangulations
        ->addAction(QString("&%1 %2").arg(i + 1).arg(recentFile));
    connect(recentFileAction, &QAction::triggered,
            this, [recentFile, this] () {loadFile(recentFile);});
}
```

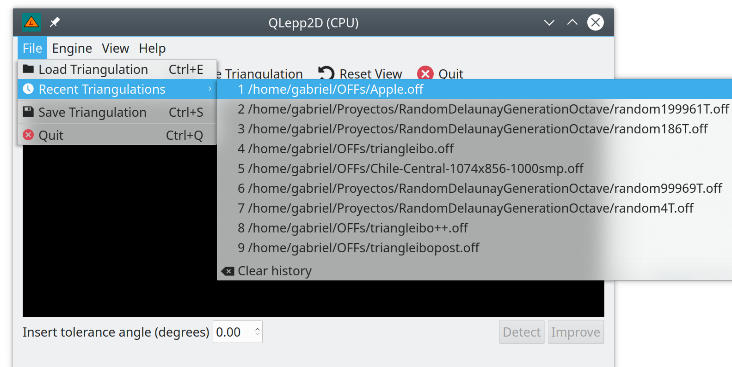

Figura 4.15: Archivos recientes.

4.4. Instalación y uso de la biblioteca

Esta biblioteca tiene como dependencias Qt (Base) y OpenCL, por lo que hay que conseguirlas antes de instalarla. Una vez que las dependencias están resueltas, basta con clonar el código de la aplicación y ejecutar los siguientes comandos:

```
cd QLepp2DLib
mkdir build
cd build
qmake CONFIG+=release ..
make
sudo make install
```

Una vez instalada en el sistema, es posible utilizarla con la interfaz de esta aplicación, o incluirla en un desarrollo distinto. Para este último caso, se debe:

- Incluir el *header* **/usr/include/QLepp2D/model.h**.
- Enlazar a la biblioteca de OpenCL.
- Enlazar a la biblioteca **qlepp2d-lib**.

Un ejemplo de Makefile es el siguiente:

```
all:
	g++ lepp.cpp -o lepp -I/usr/include/QLepp2D -lOpenCL -lqlepp2d-lib
```

Un ejemplo de desarrollo personal usando la biblioteca es el siguiente:

```
#include <model.h>

int main()
{
    Model model;
    model.setOpenCLEngine();
    model.loadFile("A.off");
    model.detectBadTriangles(50.0);
    model.improveTriangulation();
    model.saveFile("B.off");
    return 0;
}
```

4.5. Instalación y uso de la interfaz

Esta interfaz tiene como dependencias Qt (Widgets) y la biblioteca expuesta en este trabajo, por lo que hay que conseguirlas antes de instalarla. Una vez que las dependencias están resueltas, basta con clonar el código de la aplicación y ejecutar los siguientes comandos:

```
cd QLepp2DLib
mkdir build
cd build
qmake CONFIG+=release ..
make
sudo make install
```

Una vez instalada en el sistema, es posible utilizarla directamente cargando el ícono QLepp2D creado, o ejecutarla desde una consola con el comando `qlepp2d`.

Puesto que esta aplicación cambia de idioma respecto al sistema operativo, es posible probarla manualmente con los comandos:

```
LANGUAGE=en qlepp2d
LANGUAGE=es qlepp2d
```

Una vez que está instalada, se recomienda ir al menú Ayuda > Tutorial para poder entender cómo usar la aplicación.

Capítulo 5

Mediciones y análisis

En este capítulo se presentará el sistema utilizado para las mediciones, la forma de generar los archivos de prueba, el aspecto de las mallas, los resultados de las mediciones en formato de tabla, gráficos de los tiempos y análisis de los resultados, incluyendo *speedups*.

Las mediciones se realizaron comparando la implementación secuencial con la implementación paralela de esta biblioteca. Puesto que con OpenCL es posible utilizar más de una sola implementación, la implementación paralela se reportará utilizando multi-core (Intel) y GPU (Nvidia).

5.1. Ambiente de pruebas

El sistema utilizado para las mediciones tiene las siguientes características:

- Sistema Operativo: Arch Linux™
- Procesador: Intel® Core™ i7-6700HQ CPU @ 2.60GHz
- GPU: Nvidia® GeForce GTX960M (4 GB)
- RAM: 12 GB DDR4 @ 2133 MHz

Las implementaciones de OpenCL utilizadas son las siguientes:

- Nvidia: OpenCL 1.2 CUDA 9.2.101
- Intel: OpenCL 2.0 LINUX

La implementación de OpenCL fue hecha con OpenCL 1.2, para que ambos dispositivos pudiesen ser usados para las comparaciones.

5.2. Generación de archivos de prueba

Se creó un *script* especial `generator.m` (incluído en el proyecto) programado en Octave que permite generar triangulaciones Delaunay aleatorias, con el fin de poder refinarlas usando esta aplicación.

Este archivo recibe como argumento el número de puntos a utilizar en la triangulación a crear, y la genera utilizando internamente la biblioteca de QHull[1], la cual genera mallas convexas de triángulos. (Para más detalles acerca de la implementación del *script*, ver Anexo C.)

5.3. Número de elementos de los archivos de prueba

Utilizando el script `generator.m` se generaron triangulaciones tan cercanas a un múltiplo de 10 como fuesen posibles. Se hicieron varias iteraciones con el mismo comando hasta alcanzar el número deseado [2].

Número de puntos	Número de triángulos	Número de aristas
55	100	154
509	1.000	1.508
5.012	10.000	15.011
50.014	100.000	150.013
100.015	200.000	300.014
150.015	300.000	450.014
200.015	400.000	600.014
250.015	500.000	750.014
300.018	600.000	900.017
350.016	700.000	1.050.015
400.016	800.000	1.200.015
450.016	900.000	1.350.015
500.018	1.000.000	1.500.017

Tabla 5.1: Número de elementos contenidos en cada archivo.

[1] `http://www.qhull.org`

[2] Como la generación de puntos es aleatoria, correr el comando mostrado solo garantiza que el número de triángulos será cercano al reportado, por lo que se debe correr el comando más de una vez para conseguirlo.

Las mallas tienen las siguientes visualizaciones (se muestran solo algunas):

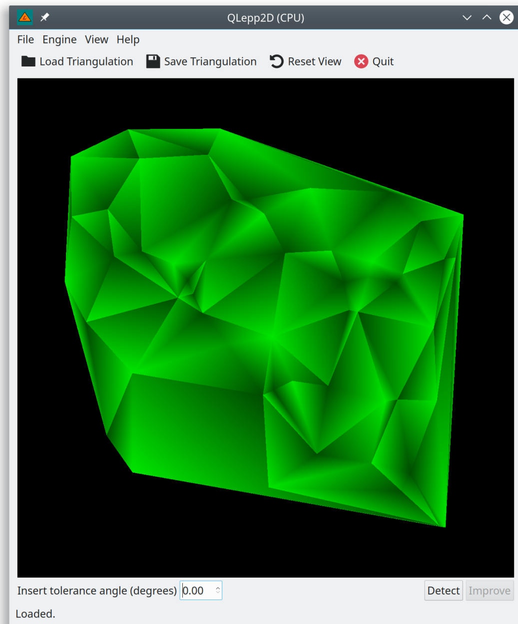

Figura 5.1: 100 triángulos.

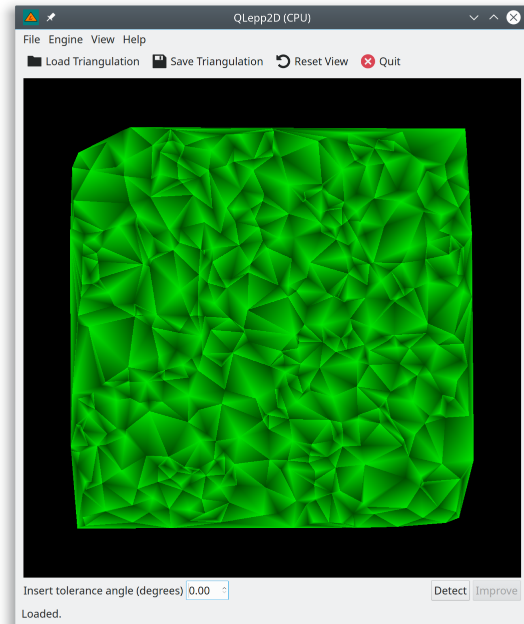

Figura 5.2: 1.000 triángulos.

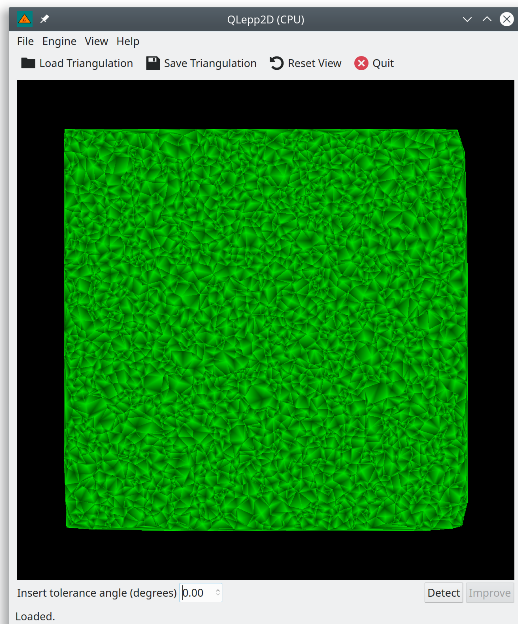

Figura 5.3: 10.000 triángulos.

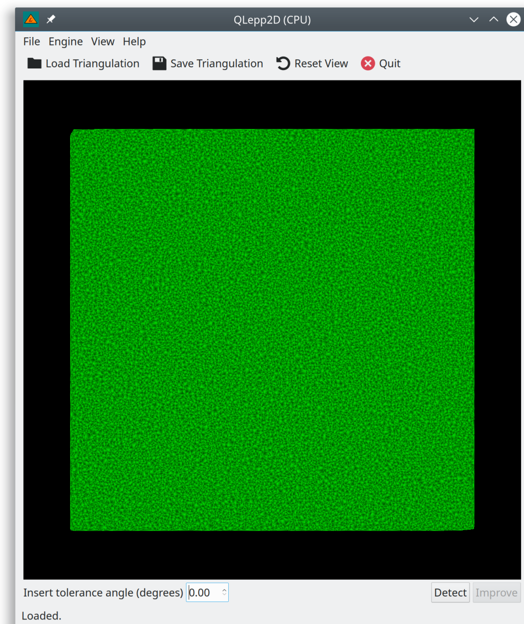

Figura 5.4: 100.000 triángulos.

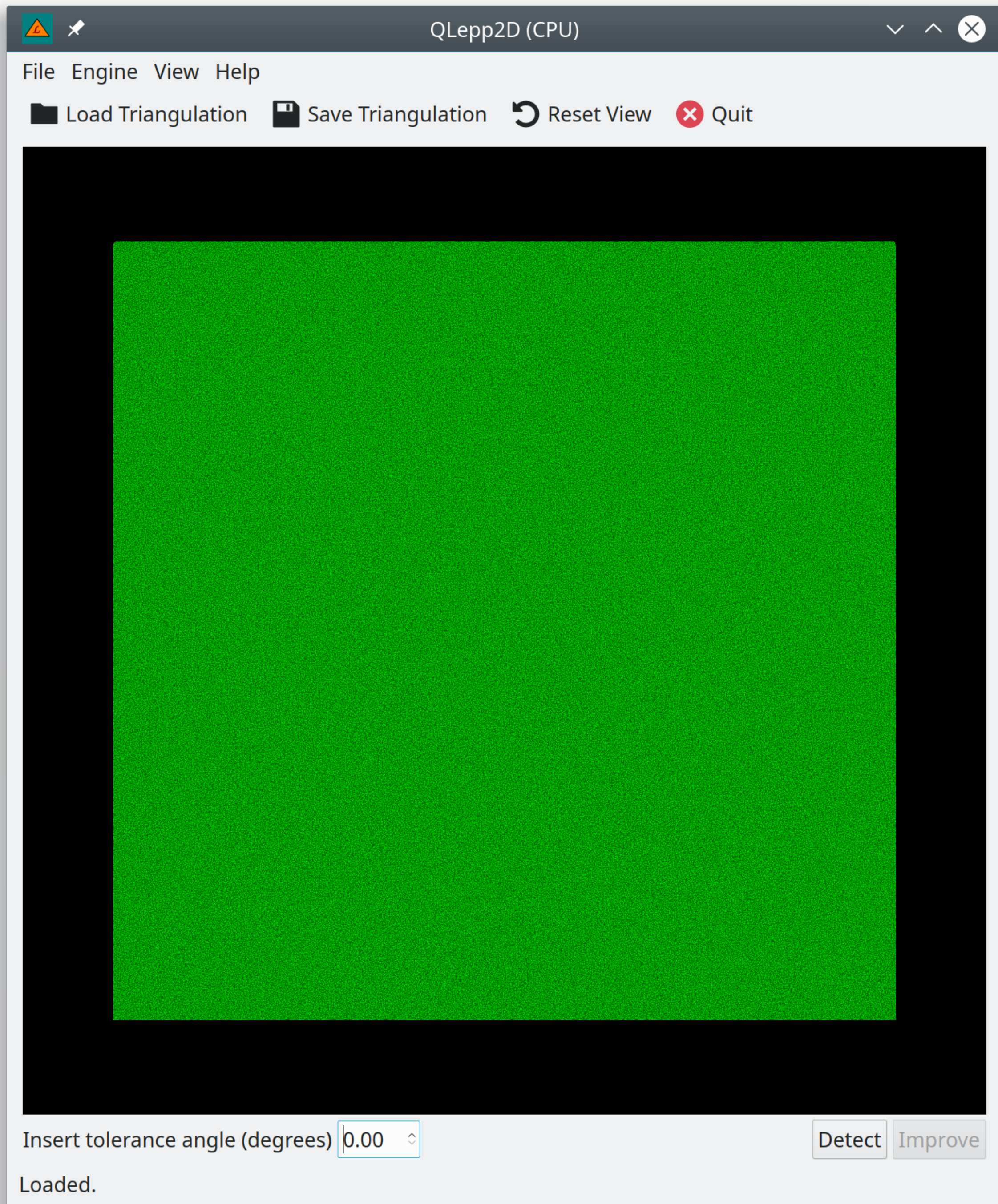

Figura 5.5: 1.000.000 triángulos.

5.4. Metodología de obtención de resultados

Los cálculos fueron hechos detectando y mejorando las triangulaciones, tomando un ángulo *threshold* de 25°.

Los tiempos de procesamiento completo fueron medidos usando `QElapsedTimer`, donde se inició la medición en la línea inmediatamente anterior al inicio del procesamiento, y se detuvo en la línea inmediatamente posterior.

Los tiempos de procesamiento del algoritmo (desde ahora, "tiempos de algoritmo") fueron medidos usando la utilidad de OpenCL que mide el tiempo utilizado por la GPU en procesar las instrucciones:

- `event.getProfilingInfo(CL_PROFILING_COMMAND_START, &time_start)`
- `event.getProfilingInfo(CL_PROFILING_COMMAND_END, &time_end)`

La precisión conseguida depende de la implementación interna de los métodos usados para tomar el tiempo: *QElapsedTimer* por parte de Qt (para la medición de la implementación del algoritmo en CPU y los tiempos de copias para ambas implementaciones) y *event.getProfilingInfo()* por parte de OpenCL.

Para ambos casos, se ejecutaron los métodos 10 veces cada uno y se sacó un valor promedio, cada uno de los cuales se encuentran en las tablas que se reportan a continuación.

5.5. Resultados

Hay dos tipos de resultados que aparecen en esta sección. El primero es el tiempo que toma detectar y mejorar las mallas, asumiendo el tiempo de copia a GPU como parte del proceso (i.e., el tiempo completo), mientras que el segundo solo toma el tiempo de procesamiento de las mallas, sin considerar el tiempo requerido para hacer la copia de los datos a GPU. De esta forma es posible ver tanto el número de triángulos mínimos necesarios para hacer que valga la pena usar GPU como el *speedup* que se obtiene al usar una mayor cantidad de *cores*.

5.5.1. Comparación de procesamiento

Esta comparación toma en cuenta la suma del tiempo de copia de los vectores a *buffers* de OpenCL más con el tiempo que toma el algoritmo, y el tiempo del algoritmo por sí solo, para las implementación de OpenCL de Nvidia (GPU) e Intel (CPU Multi-core[3]). Los ítemes marcados con **(A)** representan el tiempo (en nanosegundos) que tomó el algoritmo, mientras que los ítemes marcados con **(C)** representan el tiempo (en nanosegundos) que tomó todo el procesamiento, esto es, el algoritmo más el tiempo de copia de vectores a *buffers*.

Puesto que el algoritmo es mixto, la inserción de centroides se hace en CPU (secuencial) para todos los casos. Recordemos que `insertCentroids` inserta centroides en cada par de triángulos donde se detectó que su arista compartida era la más larga de la lista Lepp.

Triángulos	CPU
100	22.546
1.000	161.806
10.000	1.226.042
100.000	15.575.211
200.000	28.780.089
300.000	42.272.069
400.000	56.390.903
500.000	80.519.144
600.000	86.894.156
700.000	107.763.173
800.000	120.061.330
900.000	130.287.829
1.000.000	143.246.926

Tabla 5.2: Tiempo de procesamiento para **insertCentroids** (nanosegundos).

Para el resto de los métodos, se muestran las tablas y gráficos correspondientes.

[3]El algoritmo "multi-core" es en realidad la implementación hecha para GPU, pero corriendo sobre el procesador de Intel, gracias a que Intel tiene una implementación de OpenCL para CPU.

`detectBadTriangles`: Recorre los triángulos y los marca en caso de ser de mala calidad.

Triángulos	CPU	Intel (A)	Nvidia (A)	Intel (C)	Nvidia (C)
100	10.982	1.321.365	32.995	4.469.367	3.077.153
1.000	111.195	1.598.978	36.387	5.338.864	2.786.959
10.000	918.008	1.377.264	42.490	4.731.628	2.883.924
100.000	20.750.212	5.916.211	165.792	7.970.716	4.807.947
200.000	32.920.085	6.907.678	341.290	8.992.686	6.294.794
300.000	36.732.433	9.351.466	677.763	11.123.264	7.517.337
400.000	48.447.641	10.166.641	1.055.642	11.911.017	9.365.263
500.000	58.052.892	13.581.818	1.428.954	15.270.958	14.885.444
600.000	71.792.611	17.248.677	1.810.192	18.811.920	14.628.290
700.000	80.600.076	17.357.047	2.199.603	18.972.779	16.022.511
800.000	92.309.624	19.141.816	2.608.019	20.949.096	17.161.205
900.000	100.927.253	23.294.059	3.049.162	24.930.735	21.807.571
1.000.000	115.009.704	27.327.124	3.485.731	29.239.383	22.402.844

Tabla 5.3: Tiempo de procesamiento para **detectBadTriangles** (nanosegundos).

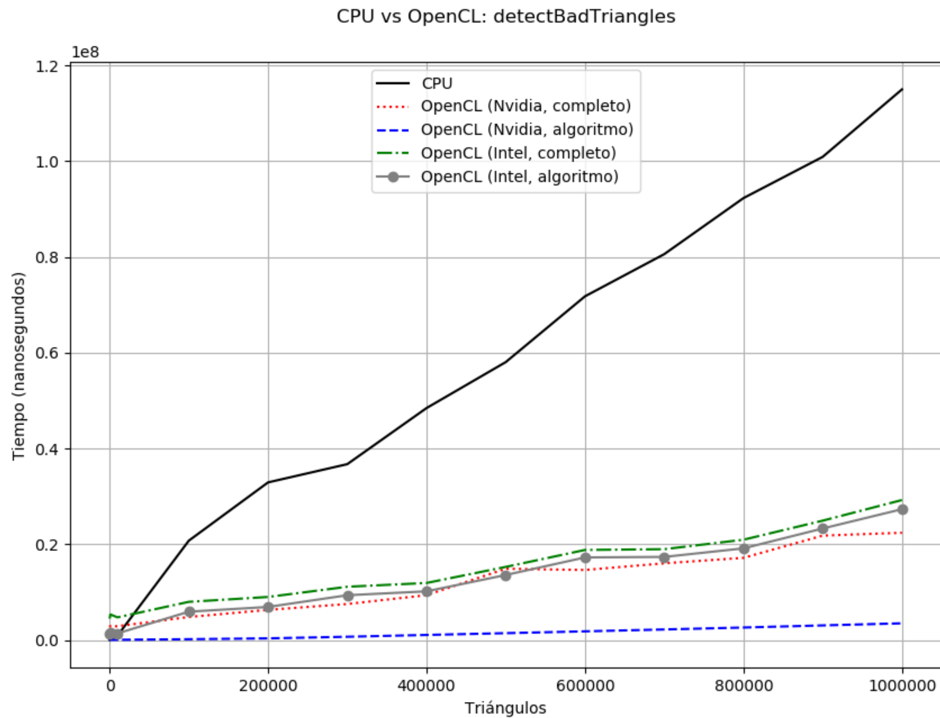

Figura 5.6: Tiempos para detectBadTriangles.

`detectTerminalEdges`: Recorre los triángulos hasta encontrar la arista terminal.

Triángulos	CPU	Intel (A)	Nvidia (A)	Intel (C)	Nvidia (C)
100	9.786	842.374	31.811	4.508.499	3.435.983
1.000	117.751	1.594.165	42.842	5.690.117	3.803.543
10.000	1.707.376	4.854.151	77.005	7.301.545	4.212.455
100.000	30.795.228	32.582.416	650.982	34.554.563	7.269.493
200.000	38.723.963	38.238.016	1.571.466	40.008.390	9.920.233
300.000	64.856.464	47.640.971	2.963.459	49.466.873	14.799.777
400.000	79.313.069	63.603.574	4.529.965	65.445.044	18.287.834
500.000	116.136.757	77.408.889	6.101.571	79.388.016	22.692.774
600.000	128.806.741	85.135.562	7.694.797	86.834.756	29.077.708
700.000	158.335.981	107.506.641	9.319.232	109.698.306	29.708.419
800.000	182.202.869	110.432.175	10.901.248	112.199.172	37.028.105
900.000	189.333.339	128.307.412	12.579.965	130.033.407	40.391.006
1.000.000	208.480.405	149.036.108	14.112.723	150.961.419	42.299.784

Tabla 5.4: Tiempo de procesamiento para **detectTerminalEdges** (nanosegundos).

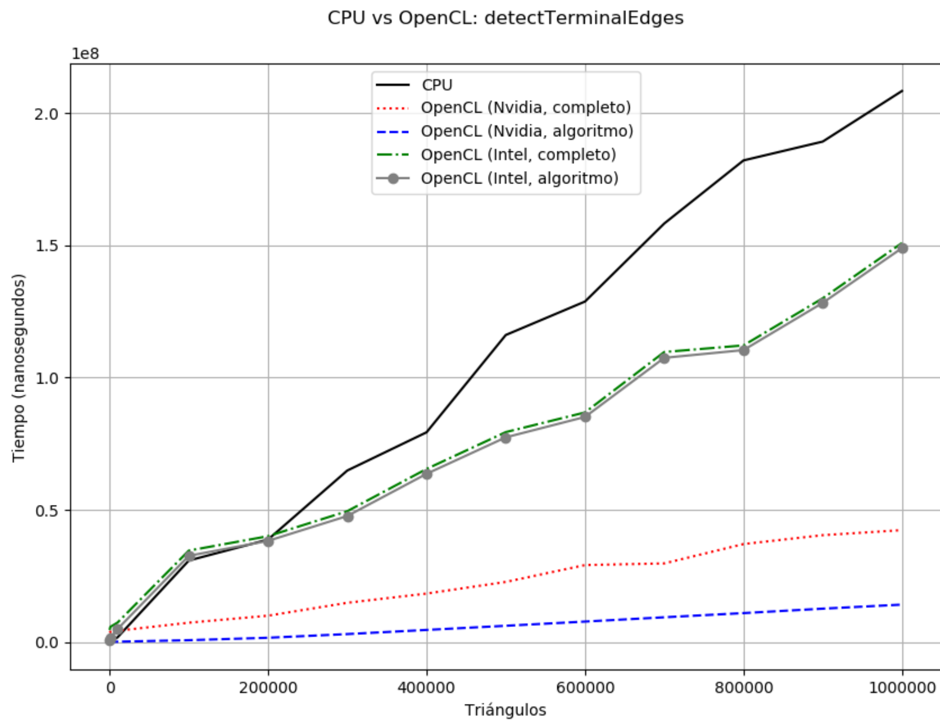

Figura 5.7: Tiempos para detectTerminalEdges.

Triángulos	CPU	Intel (A)	Nvidia (A)	Intel (C)	Nvidia (C)
100	43.313	2.186.284	87.352	9.000.411	6.535.682
1.000	390.752	3.354.949	241.035	11.190.787	6.752.308
10.000	3.851.427	7.457.458	1.345.537	13.259.215	8.322.421
100.000	67.120.651	54.073.838	16.391.986	58.100.490	27.652.652
200.000	100.424.137	73.925.784	30.692.844	77.781.165	44.995.116
300.000	143.860.966	99.264.506	45.913.292	102.862.206	64.589.183
400.000	184.151.613	130.161.119	61.976.510	133.746.964	84.044.001
500.000	254.708.794	171.509.851	88.049.669	175.178.118	118.097.363
600.000	287.493.508	189.278.396	96.399.145	192.540.833	130.600.155
700.000	346.699.230	232.626.860	119.282.008	236.434.258	153.494.103
800.000	394.573.823	249.635.321	133.570.597	253.209.598	174.250.640
900.000	420.548.421	281.889.300	145.916.955	285.251.970	192.486.406
1.000.000	466.737.035	319.610.159	160.845.380	323.447.728	207.949.554

Tabla 5.5: Tiempo total de procesamiento (nanosegundos).

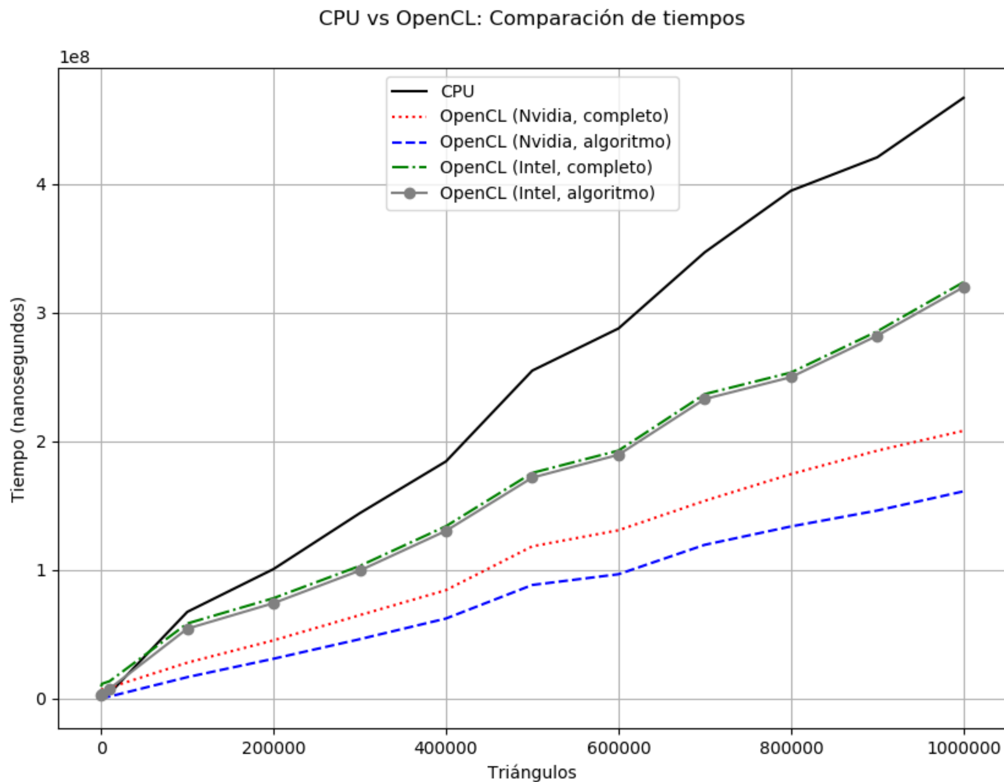

Figura 5.8: Tiempos totales.

5.6. Análisis

En esta sección se presentan los speedups conseguidos y se plantean posibles respuestas a los resultados encontrados usando la aplicación.

5.6.1. Speedup: Nvidia (GPU)

Detección de triángulos malos

El mejor resultado se dio con el método *detectBadTriangles*, donde el algoritmo de detección de triangulos malos (sin considerar la copia a buffers) fue siempre más rápido que en CPU, donde en particular, para 100.000 triángulos se encontró un *speedup* de 404.8 veces superior a la implementación en CPU.

Curiosamente, al probar con mallas mayores a 100.000 triángulos, el *speedup* del algoritmo comenzó a bajar considerablemente (hasta 133.9 veces más rápido solamente). Se investigó la posible razón de esta pérdida de *speedup*, y se encontró que el tamaño en bytes de los componentes de la malla (vértices, aristas y triángulos) supera la memoria mapeable (*pinned memory*) disponible que el sistema le da a OpenCL[2], por lo que se pierde ancho de banda al momento de traspasar información.

Tomando en cuenta el tiempo en el algoritmo completo (incluyendo copia a buffers), el *speedup* estuvo debajo del de CPU hasta antes de los 10.000 triángulos, donde el *speedup* fue en ese momento de 1.33, y subió paulatinamente hasta 20.8 para 1.000.000 triángulos.

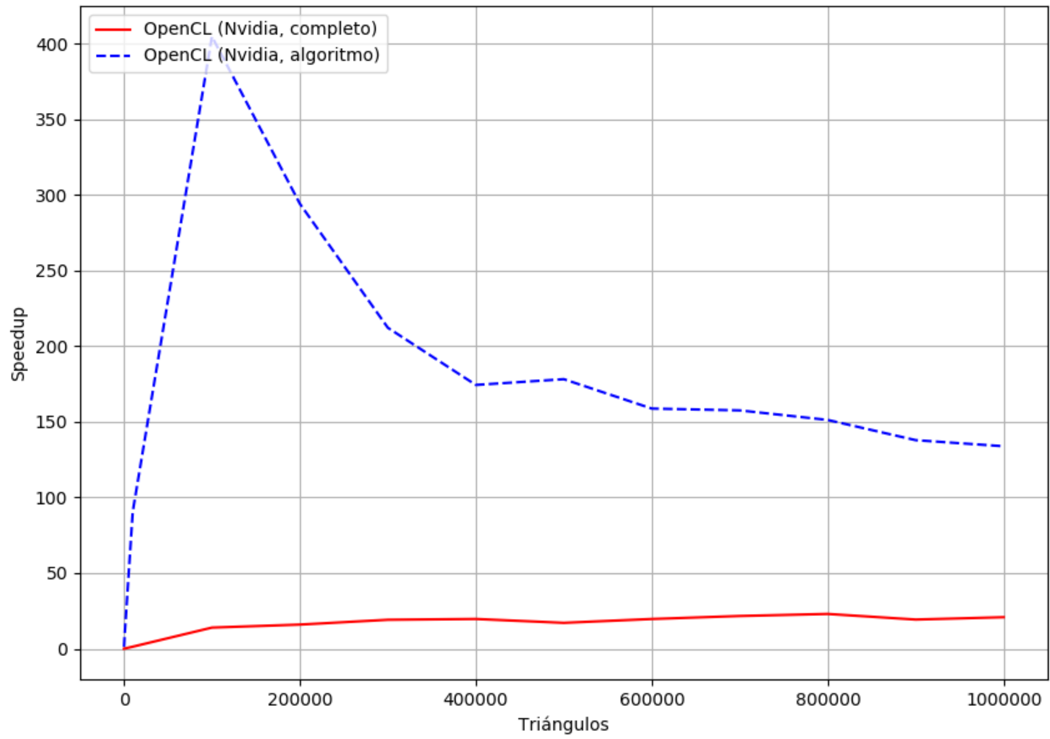

Figura 5.9: Speedup de detectBadTriangles (Nvidia).

Detección de aristas terminales

La misma subida (y repentina bajada) de *speedup* ocurrió con la detección de aristas terminales: La implementación del algoritmo (sin considerar la copia a buffers) fue siempre superior, con un *speedup* mínimo de 1.36 para 100 triángulos y un *speedup* máximo de 103.11 para 100.000 triángulos, obteniendo un *speedup* de 33.1 para 1.000.000 triángulos.

Tomando en cuenta el tiempo en el algoritmo completo (incluyendo copia a buffers), el *speedup* estuvo debajo del de CPU hasta ligeramente después de los 10.000 triángulos, donde el *speedup* fue en ese momento de 0.92, y subió paulatinamente hasta 11.03 para 1.000.000 triángulos.

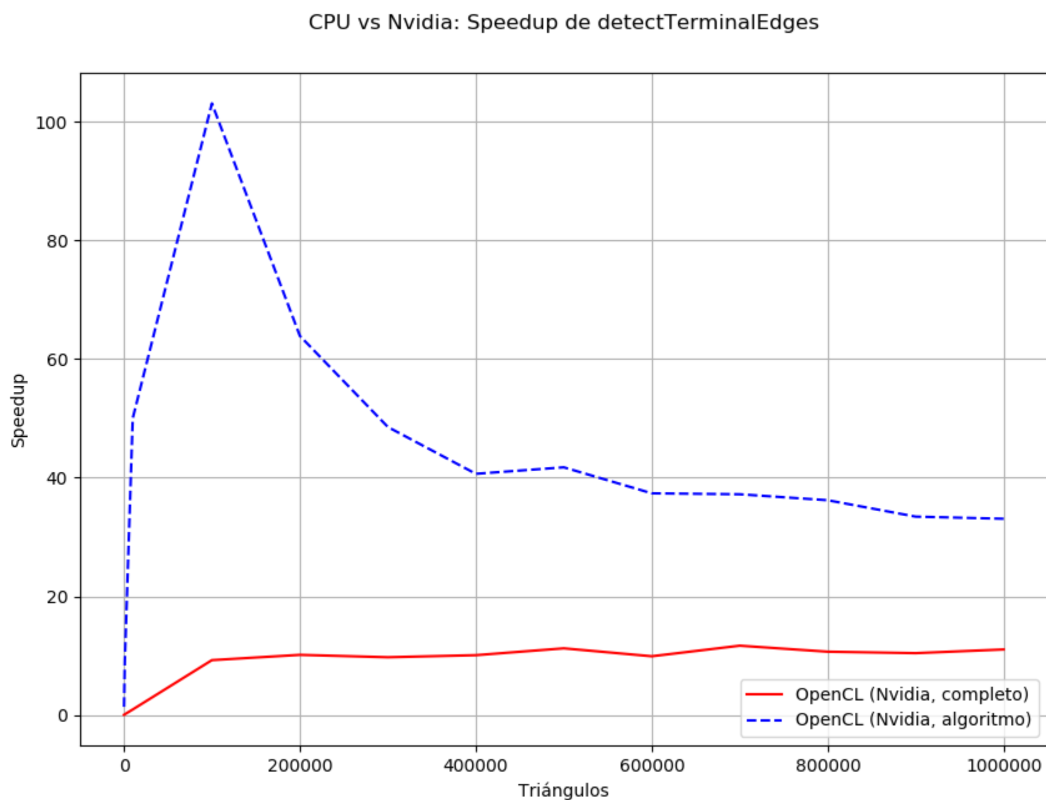

Figura 5.10: Speedup de detectTerminalEdges (Nvidia).

Tiempos totales

Recordando que esta es una implementación mixta, donde la inserción de centroides se realiza 100 % en CPU, se sumaron los tiempos de la detección de triángulos malos, la detección de aristas terminales y la inserción de centroides, para los casos "solo-algoritmo" y "algoritmo-más-copia-a-buffer".

Al considerar solo los algoritmos, se encontró un *speedup* de 0.5 para 100 triángulos, lo que hace suponer que la caché de CPU le da ventaja respecto a la implementación GPU. Sin embargo, esta ventaja se pierde rápidamente desde 1.000 triángulos en adelante, donde el *speedup* conseguido llega hasta 4.1 veces más rápido que en CPU para 100.000 triángulos.

A juzgar por la forma de implementar los algoritmos de la aplicación, es muy probable que el *speedup* tienda a $1.0 + \varepsilon$ para una cantidad infinita de triángulos, puesto que, a pesar de que los algoritmos en GPU siempre son más rápidos que los de CPU, el tiempo tomado por la inserción de centroides (que se hace siempre en CPU) convertirá en insignificante al resto de los tiempos que consigue la implementación en GPU.

Al considerar los algoritmos y los tiempos de copia, se encontró que el rendimiento en GPU era inferior al de CPU hasta los 10.000 triángulos, donde el *speedup* de ese momento era 0.46. Sin embargo, al aumentar el número de triángulos, el *speedup* es superior al de CPU, donde para 100.000 triángulos es de 2.43 y para 1.000.000 triángulos es de 2.24 (aludiendo a las razones anteriormente expuestas del por qué se pierde *speedup*.)

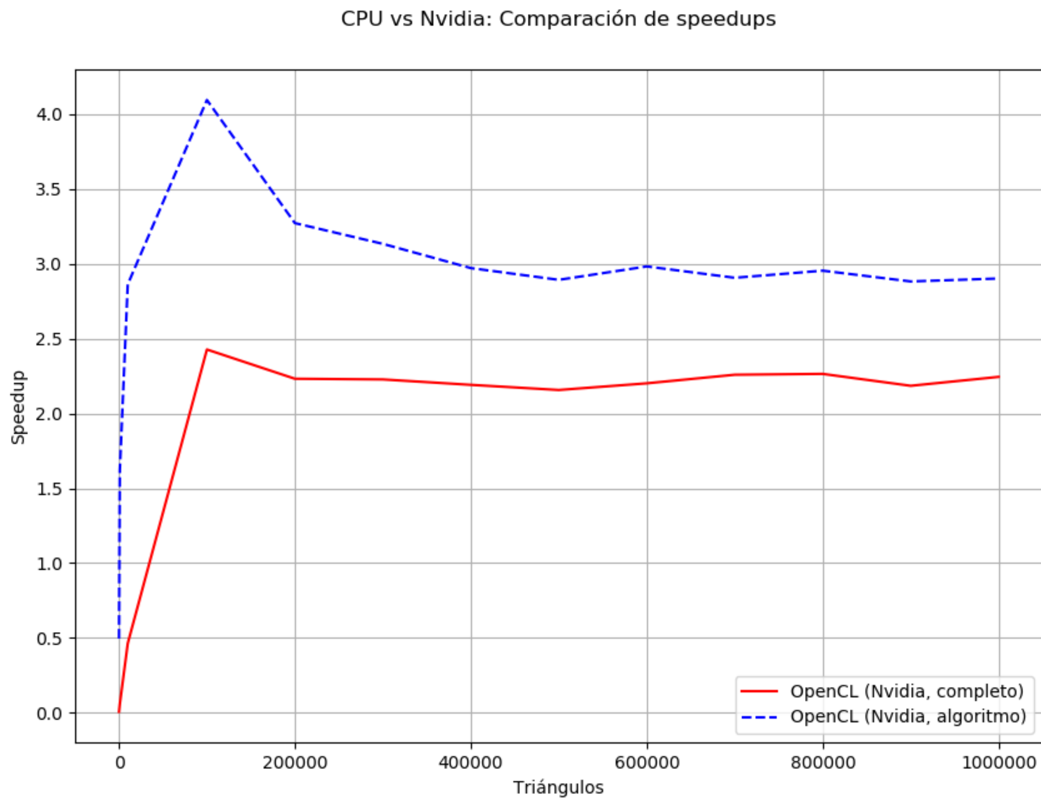

Figura 5.11: Speedup total (Nvidia).

5.6.2. Speedup: Intel (Multi-core)

Dado que OpenCL permite trabajar con cualquier dispositivo que tenga una implementación de OpenCL disponible, se utilizó la implementación para Intel CPU, convirtiendo los algoritmos originalmente diseñados para GPU, en algoritmos *multi-core* automáticamente.

Detección de triángulos malos

El mejor resultado se dio con el método *detectBadTriangles*, donde el algoritmo de detección de triangulos malos (sin considerar la copia a buffers) fue más rápido que en una implementación secuencial para 10.000 triángulos y más, donde se encontró un *speedup* de hasta 20.6 veces superior para 800.000 triángulos.

Es muy probable que OpenCL para Intel haga optimizaciones internamente cuando tiene una gran cantidad de threads, puesto que se tiene un *speedup* de 20.6 para 800.000 triángulos, siendo que solo hay 8 *threads* en el procesador.

Curiosamente, el *speedup* fue inferior a la implementación secuencial para 100 y 1.000 triángulos, donde se encontró un *speedup* de 0.03 y 0.24, respectivamente, lo que hace pensar que OpenCL no logra optimizar bien el paralelismo en CPU para pocos triángulos.

Como no se encontró mucha pérdida de *speedup* para 1.000.000 triángulos, la falta de memoria es una razón con mayor verosimilitud para explicar esa pérdida cuando se utilizó la implementación de OpenCL para Nvidia.

Tomando en cuenta el tiempo en el algoritmo completo (incluyendo copia a buffers), el *speedup* estuvo debajo del de CPU hasta los 10.000 triángulos, donde el *speedup* fue en ese momento de 0.81, y subió paulatinamente hasta 18.83 para 800.000 triángulos.

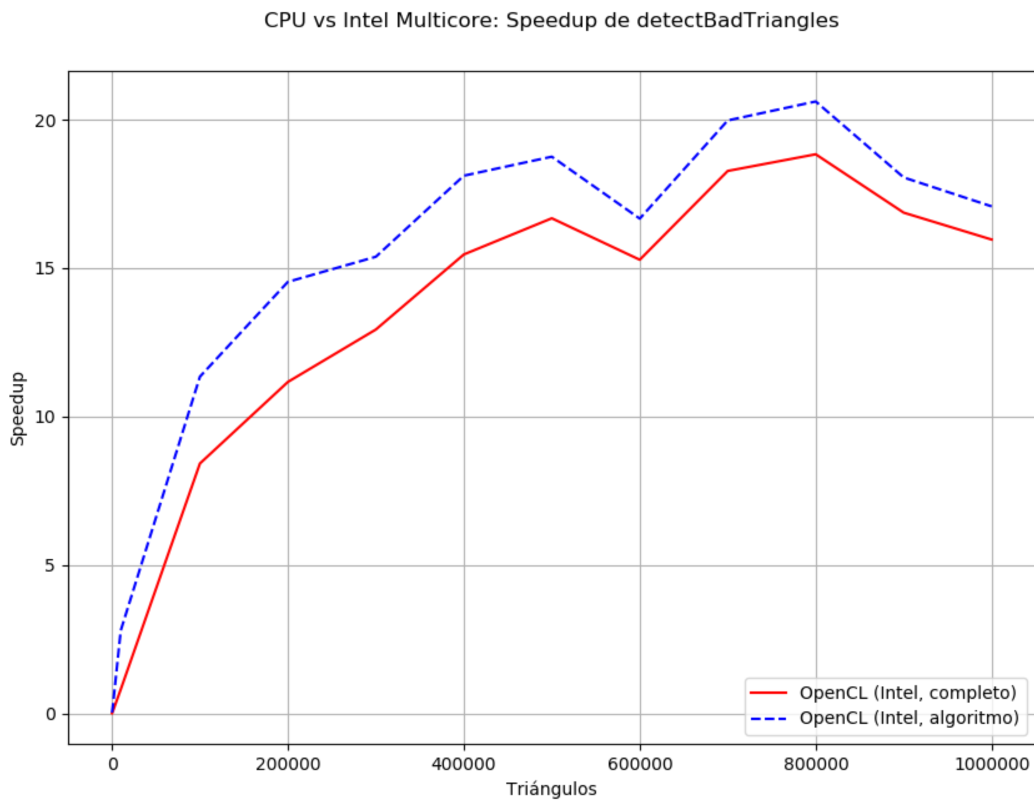

Figura 5.12: Speedup de detectBadTriangles (Intel).

Detección de aristas terminales

La implementación del algoritmo (sin considerar la copia a buffers) fue inferior a la implementación secuencial, donde el mejor *speedup* menor que 1 fue de 0.79 para 10.000 triángulos. Sin embargo, la implementación de OpenCL superó a la implementación secuencial en los 100.000 triángulos, con un *speedup* de 2.06, y de 3.57 para 800.000 triángulos.

Tomando en cuenta el tiempo en el algoritmo completo (incluyendo copia a buffers), el *speedup* estuvo debajo del de CPU hasta después de los 10.000 triángulos, donde el *speedup* fue en ese momento de 0.52. Sin embargo, la implementación de OpenCL superó a la implementación secuencial en los 100.000 triángulos, con un *speedup* de 1.94, y de 3.51 para 800.000 triángulos.

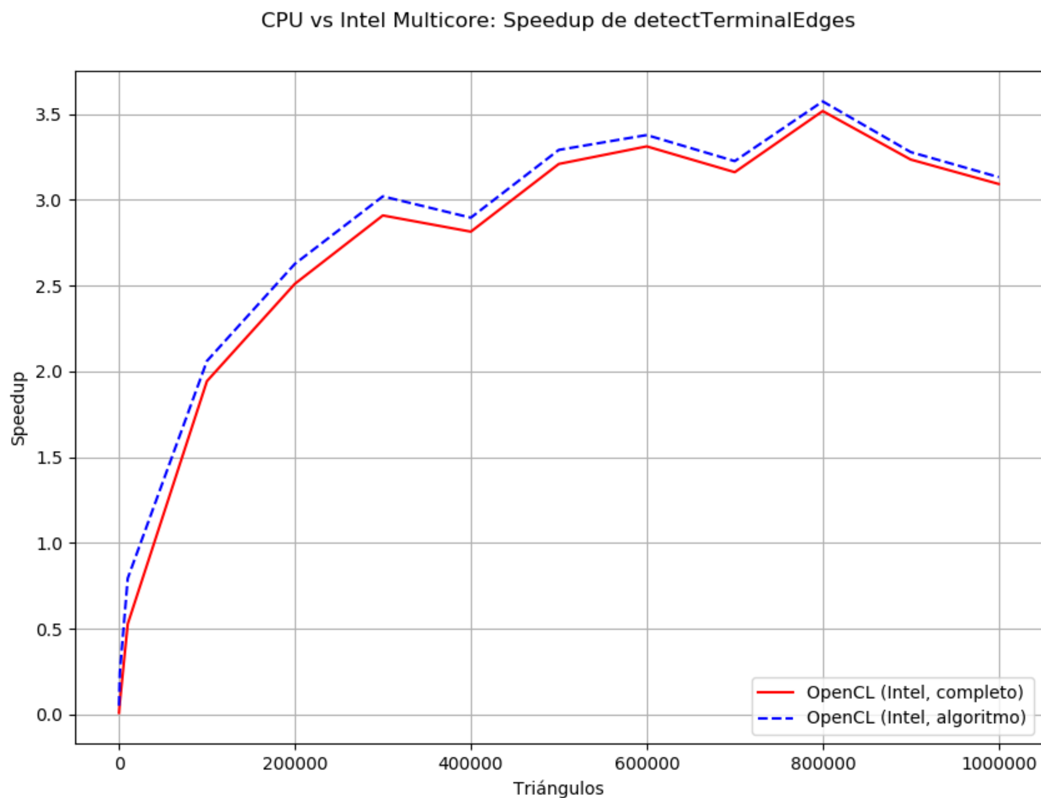

Figura 5.13: Speedup de detectTerminalEdges (Intel).

Tiempos totales

Al considerar solo los algoritmos, se encontró un *speedup* inferior que en la implementación secuencial, de hasta 0.51 para 10.000 triángulos, lo que hace suponer que el uso de caché del procesador en la implementación secuencial fue más eficiente para una cantidad reducida de triángulos que las posibles optimizaciones que pueda estar haciendo OpenCL internamente.

Se observa un *speedup* muy ligeramente superior para 100.000 y 800.000 triángulos, de 1.24 y 1.58 veces más rapido, respectivamente.

Al considerar los algoritmos y los tiempos de copia, se encontró un *speedup* inferior que en la implementación secuencial, de hasta 0.29 para 10.000 triángulos, pero que remonta a 1.16 y 1.56 para 100.000 y 800.000 triángulos, respectivamente.

El que no se pueda observar una estrepitosa caída de *speedup* respecto a lo que ocurrió con la implementación de OpenCL para Nvidia hace suponer que mientras haya memoria, es posible conseguir subir el *speedup* para más y más triángulos.

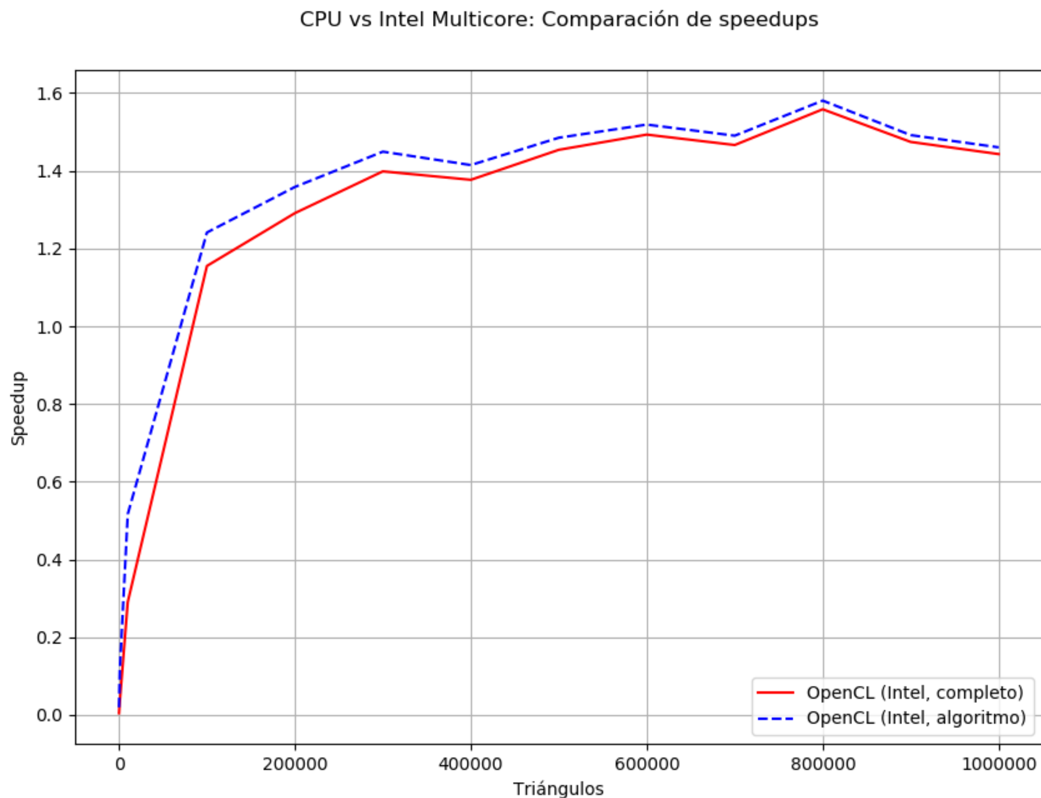

Figura 5.14: Speedup total (Intel).

Capítulo 6

Conclusiones y Trabajo Futuro

6.1. Aplicación

En este trabajo de título se presentó el diseño, implementación y rendimiento de la suma de algoritmos en GPU y algoritmos en CPU en una biblioteca, y un visualizador de mallas triangulares en una interfaz gráfica, que concluyo en la aplicación llamada QLepp2D.

Esta aplicación permite visualizar y mejorar mallas de triángulos en 2D utilizando el algoritmo Lepp-Centroide, con una implementación secuencial y una implementación mixta GPU + CPU, además de permitir guardar las mejoras en archivos OFF, para su posterior uso por otras aplicaciones.

Utilizando patrones de diseño y orientación a objetos se logró implementar esta aplicación de tal manera que es posible extender su funcionalidad a nuevas implementaciones de algoritmos, al manejo de nuevos tipos de archivos y a la creación de nuevas interfaces gráficas sin necesidad de modificar el código directamente.

Para el caso de la biblioteca, es muy sencillo utilizarla en otros programas, simplemente cargando el archivo OFF y manejándolo con la API pública ofrecida. Además, en la biblioteca se ofrece un ejemplo que usa todos los métodos básicos.

Para el caso de la interfaz, se diseñó de tal forma que fuera intuitivo utilizarla (dejando visible las opciones más evidentes, como cargar/guardar una triangulación, detectar triángulos malos y mejorar la triangulación), atendiendo a distintos casos de uso, como *keybindings* y arrastre de archivos sobre la interfaz, y agregando un historial de archivos que facilita la carga de archivos anteriormente procesados.

Adicionalmente, la interfaz posee capacidades de internacionalización, lo que la diferencia de la mayoría de los visualizadores, los cuales solo están en inglés, y agrega un tutorial para poder usar la aplicación (que también es traducible).

6.2. Resultados obtenidos

Dentro de los resultados obtenidos por la biblioteca de la aplicación, se puede observar que para la implementación de OpenCL en GPU el *speedup* es notablemente bueno para triangulaciones de tamaño mediano (50.000 a 1.000.000 triángulos), donde se pueden mejorar triangulaciones en un tiempo bastante más corto que con una implementación secuencial, aunque se recomienda tener una tarjeta gráfica con una gran capacidad de memoria para que ésta no sea un cuello de botella.

Para la implementación multi-core (OpenCL para Intel), se puede conseguir mayor velocidad que la implementación secuencial para triangulaciones de mayor tamaño (100.000 a 1.000.000+), siempre y cuando la memoria *pinneable* del sistema sea suficientemente alta.

6.3. Trabajo futuro

Se han identificado los siguientes puntos que pueden suponer una mejora a la aplicación, tanto en la biblioteca como en la interfaz.

6.3.1. Biblioteca

- Mejorar la implementación de OpenCL para disminuir la necesidad de la parte mixta del algoritmo, incrementando el uso de GPU.
- Agregar soporte para poder leer y escribir una mayor cantidad de tipos de archivo.
- Agregar un nuevo motor de procesamiento específicamente diseñado para multi-core.
- Disminuir la dependencia de Qt para la biblioteca, a fin de poder instalarla en sistemas que no posean este *framework*.
- Agregar *enum*s de códigos de error para poder mostrar mayor detalle en caso que una parte de la biblioteca no funcione correctamente.
- Exponer una forma de detectar si hay más de una implementación de OpenCL instalada en el sistema, en lugar de utilizar la primera que se encuentra.

6.3.2. Interfaz

- Agregar soporte para incluir mayor cantidad de tipos de archivos.
- Mejorar el *widget* de OpenGL para permitir visualizar aristas por sobre los triángulos, evitando usar mallas “artísticas” para poder diferenciar sus componentes.
- Exponer una interfaz gráfica para que el usuario elija la implementación de OpenCL a usar, solo en caso de tener más de una (dependiente de las capacidades de la biblioteca).

Bibliografía

[1] CUDA vs OpenCL; Which should I use? https://wiki.tiker.net/CudaVsOpenCL. Accessed: 2017-11-13.

[2] NVIDIA OpenCL Best Practices Guide. pages 10–10, 2009.

[3] Mixing graphics and compute with multiple GPUs. http://on-demand.gputechconf.com/gtc/2012/presentations/S0267A-GTC2012-Mixing-Graphics-Compute.pdf, 2013. Accessed: 2017-11-14.

[4] Christopher Alexander, Sara Ishikawa, Murray Silverstein, Max Jacobson, Ingrid Fiksdahl-King, and Shlomo Angel. *A pattern language*. Oxford University Press, 1977.

[5] Wikipedia contributors. Fan triangulation — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Fan_triangulation&oldid=828677588, 2018. Accessed: 2018-06-21.

[6] Jesus A. De Loera, Jorg Rambau, and Francisco Santos. *Triangulations: Structures for Algorithms and Applications*. Springer Publishing Company, Incorporated, 1st edition, 2010.

[7] Alan Ezust and Paul Ezust. *An Introduction to Design Patterns in C++ with Qt, Second Edition*. Prentice Hall Press, Upper Saddle River, NJ, USA, second edition, 2011.

[8] James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes. *Computer Graphics: Principles and Practice (2Nd Ed.)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1990.

[9] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[10] Charles L. Lawson. Transforming triangulations. *Discrete Math.*, 3(4):365–372, January 1972.

[11] The Qt Company Ltd. ModelView Programming. http://doc.qt.io/qt-5/model-view-programming.html, 2018. Accessed: 2018-06-19.

[12] Wolfgang Mulzer and Günter Rote. Minimum-weight triangulation is np-hard. *J. ACM*, 55(2):11:1–11:29, May 2008.

[13] Maria-Cecilia Rivara and Nancy Hitschfeld. Lepp-delaunay algorithm: a robust tool for producing size-optimal quality triangulations. In *Proc. of the 8th Int. Meshing Roundtable*, pages 205–220, 1999.

[14] María Cecilia Rivara. New mathematical tools and techniques for the refinement and/or improvement of unstructured triangulations. In *Proceedings of 5th International Meshing Roundtable, Sandia National Laboratories*, pages 77–86, 1996.

[15] Jason Sanders and Edward Kandrot. *CUDA by Example, An Introduction to General-Purpose GPU Programming*. Addison-Wesley, 1st edition, 2011.

[16] R.T. Stevens. *Computer Graphics Dictionary*. Advances in computer graphics and game development. Charles River Media, 2002.

[17] Eric Weisstein. Law of cosines - from mathworld–a wolfram web resource. `http://mathworld.wolfram.com/LawofCosines.html`. Accessed: 2018-06-22.

[18] Morten Welinder. How Does One Create A Gtk+ Application? `https://blogs.gnome.org/mortenw/2014/06/23/how-does-one-create-a-gtk-application/`, 2014. Accessed: 2017-11-14.

Anexos

A. Estructura de un archivo OFF

Un archivo con extensión OFF tiene la siguiente estructura.

- La palabra “OFF” al inicio del archivo.
- Comentarios (opcionales) que comienzan con $\sharp$.
- 3 números separados por espacios que representan el número de vértices, caras y aristas de la figura.
- Una lista de coordenadas X, Y y Z de cada vértice, separadas por espacios.
- Una lista de caras, donde cada elemento de la lista contiene n números. El primer número representa la cantidad de vértices que usa una cara, y los $n-1$ números restantes representan los punteros a los vértices que componen esa cara.

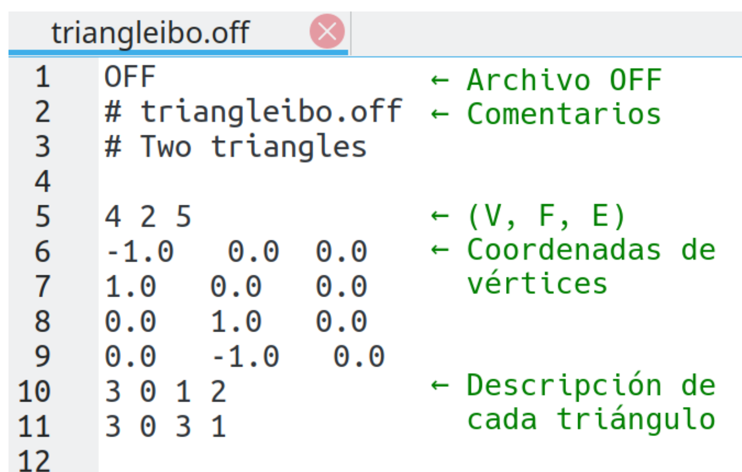

Figura 1: Ejemplo de un archivo OFF.

B. Patrón NSC y SNC

NSC y SNC son los nombres que se les dieron a los patrones de vértices, de tal forma que al crear los 4 triángulos necesarios respecto al centroide, éstos fueran mejores que los 2 triángulos anteriores.

Las siglas N, S y C representan clases de vértices respecto a la arista compartida:

- N: Vértices que **no** se encuentran en la arista compartida.
- S: Vértices que **sí** se encuentran en la arista compartida.
- C: Centroide.

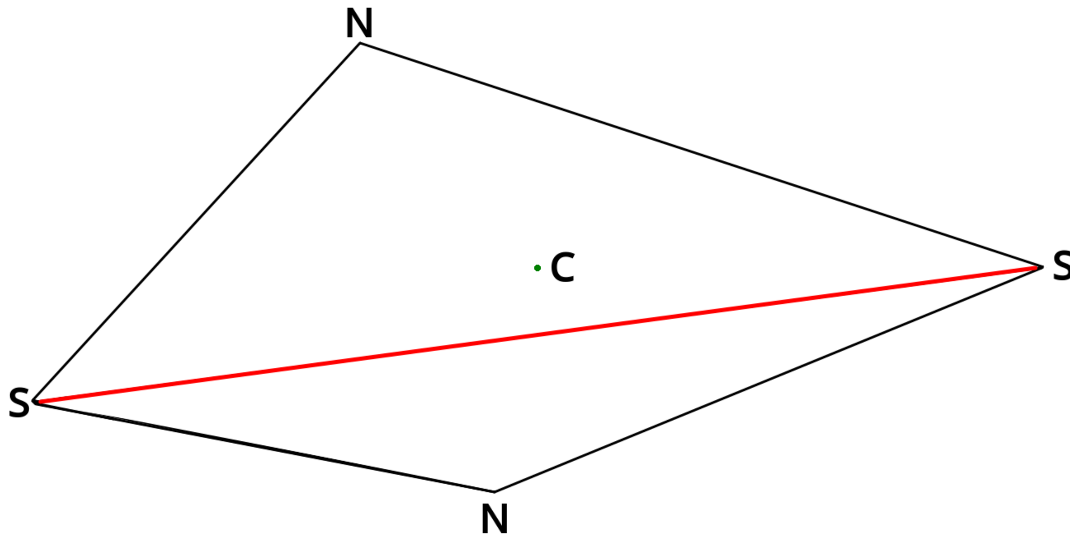

Figura 2: Triángulos terminales (arista compartida en rojo).

Como se puede ver en la Figura 3, T1 y T3 siguen el patrón NSC, porque para poder crearlos, se tomó primero un vértice de clase N, después un vértice de clase S y finalmente el centroide. Análogamente, T2 y T4 siguen el patrón SNC.

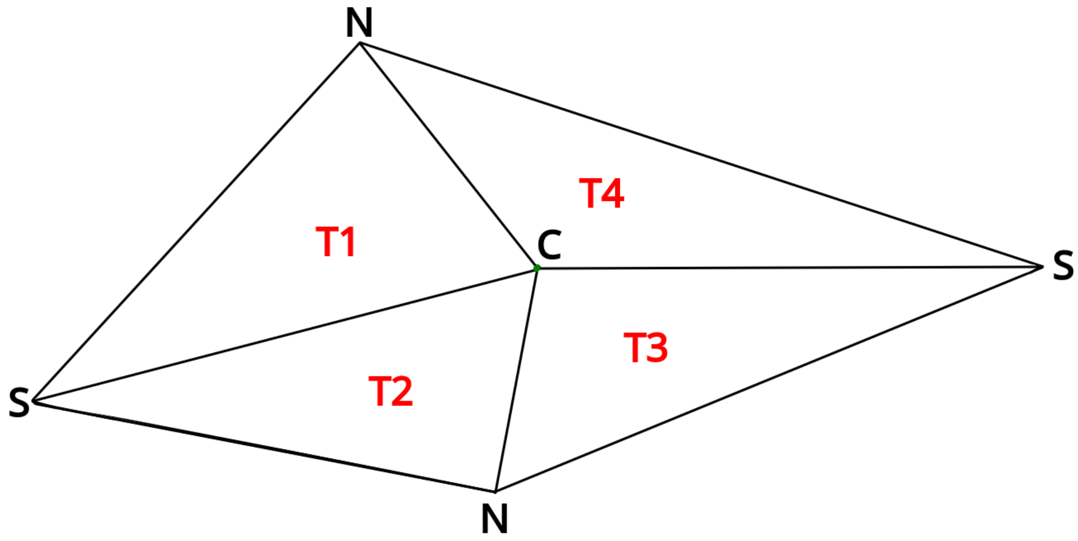

Figura 3: Triángulos correctamente creados.

Por otro lado, no seguir estos patrones puede llevar a que los triángulos creados sean peores que los originales y que se crucen entre ellos.

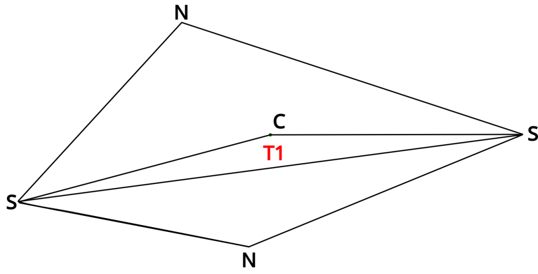

Figura 4: Triángulo mal creado (SSC).

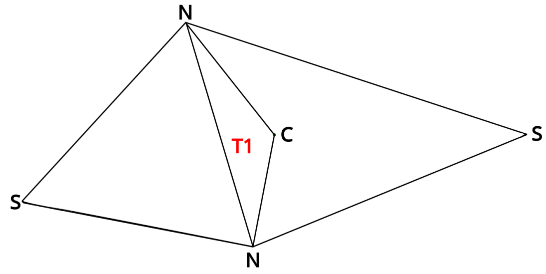

Figura 5: Triángulo mal creado (NNC).

C. Generación de triangulaciones de prueba

Se creó un *script* especial `generator.m` (incluído en el proyecto) programado en Octave que permite generar triangulaciones Delaunay aleatorias, con el fin de poder refinarlas usando esta aplicación.

El *script* recibe como argumento el número de puntos a usar.

```
#!/usr/bin/octave -qf
# This script creates a random Delaunay mesh.

# Parse command line arguments
arg_list = argv();

if (size(arg_list) > 0)
    V = str2num(arg_list{1});
else
    printf("Usage: generator.m N\n");
    printf("  N = Number of points\n");
    exit(1);
endif
```

A partir del número de puntos recibidos, se crean números aleatorios en el rango $[-1, 1]$ para las coordenadas X e Y, y 0 para la coordenada Z.

```
printf("Creating %d random points...\n", V);

x = 2 * (rand(1, V) - 0.5);
y = 2 * (rand(1, V) - 0.5);
z = zeros(1, V);
```

Luego, Octave usa la implementación de QHull para poder crear una triangulación de Delaunay a partir de los puntos aleatorios.

```
printf("Creating Delaunay mesh...\n");
tri = delaunay(x, y);

F = size(tri)(1);

printf("%d triangles have been created!\n", F);
```

Finalmente, se calculan las aristas que debería tener la triangulación, y se escribe a un archivo de extensión OFF.

```
# Euler's formula: V + F - E = 2
E = V + F - 2;

printf("Writing file...\n");

# Now we write our OFF file.
FILENAME = strcat("random", num2str(F), "T.off");
fid = fopen(FILENAME, "w");
fprintf(fid, "OFF\n");
fprintf(fid, "# File randomly generated using Octave's delaunay method,\n");
fprintf(fid, "%d %d %d\n", V, F, E);

# First we write the vertices' coordinates.
for (i = 1:V)
    fprintf(fid, "%f %f %f\n", x(i), y(i), z(i));
endfor

# Then, we write the pointers to vertices so we can get our triangles.
# Remember to offset the numbers, because Octave's arrays start at 1.
for (i = 1:F)
    fprintf(fid, "%d %d %d %d\n", 3, tri(i, 1) - 1,tri(i, 2) - 1, tri(i, 3) - 1);
endfor

fclose(fid);

printf("Finished!\n");
```

D. Extendiendo la biblioteca

La biblioteca fue pensada para poder ser modificada y extendida sin la necesidad de pasar a llevar código de módulos distintos al que se pretende modificar. Adicionalmente, se agregó documentación para cada método de cada clase. A modo de ejemplo, se muestran dos formas de extender la biblioteca.

D.1. Añadir un nuevo motor de procesamiento

Un nuevo motor de procesamiento debe heredar de la clase *Engine* y hacer *override* de cada uno de sus métodos. De esta forma, la clase *ModelImpl* puede agregar y usar el nuevo motor, y la clase *Model* puede exponer este nuevo motor, agregando el método a la API.

El método `detectBadTriangles`, encargado de detectar y marcar los triángulos de mala calidad, requiere conocer el ángulo *threshold* para decidir si el triángulo es bueno o no, el arreglo de vértices (para conocer las coordenadas) y el arreglo de triángulos (para revisarlos).

```
/**
 * @brief Detects every bad triangle in the vector of triangles.
 *
 * @param angle p_angle: Tolerance angle.
 * @param vertices Vector of vertices.
 * @param triangles Vector of triangles.
 * @return True if detected without issues.
 */
virtual bool detectBadTriangles(float angle,
                                std::vector<Vertex> &vertices,
                                std::vector<Triangle> &triangles) = 0;
```

El método `detectTerminalEdges`, encargado de detectar y marcar las aristas terminales, requiere el arreglo de vértices (para conocer las coordenadas), el arreglo de aristas (para conocer los índices a los triángulos vecinos) y el arreglo de triángulos (para revisar cuál arista de cada triángulo es la más larga). Adicionalmente, existe un argumento *flag*, que avisa si quedan aristas terminales internas o solo quedan aristas terminales de borde.

```
/**
 * @brief Detects terminal edges for each bad triangle in the "triangles"
 * vector.
 *
 * @param vertices Vector of vertices.
 * @param edges Vector of edges.
 * @param triangles Vector of triangles.
 * @param flag Flag that marks if a non-border terminal edge still
 * exists.
 */
virtual void detectTerminalEdges(std::vector<Vertex> &vertices,
                                 std::vector<Edge> &edges,
                                 std::vector<Triangle> &triangles,
                                 bool &flag) = 0;
```

El método `insertCentroids`, encargado de insertar el centroide y actualizar los vectores de información, requiere el arreglo de vértices (para conocer las coordenadas e insertar el centroide), el arreglo de aristas (para conocer los punteros a las antiguas aristas no compartidas e insertar las nuevas aristas internas) y el arreglo de triángulos (para insertar los nuevos triángulos).

```
/**
 * @brief Inserts centroids on every region that has a terminal edge.
 *
 * @param vertices Vector of vertices.
 * @param edges Vector of edges.
 * @param triangles Vector of triangles.
 */
virtual void insertCentroids(std::vector<Vertex> &vertices,
                             std::vector<Edge> &edges,
                             std::vector<Triangle> &triangles) = 0;
```

Finalmente, el método `improveTriangulation`, que engloba los tres métodos anteriores, requiere todos los arreglos de elementos geométricos, para entregárselos a estos métodos.

```
/**
 * @brief Improves the actual triangulation from the vector of triangles.
 *
 * @param vertices Vector of vertices.
 * @param edges Vector of edges.
 * @param triangles Vector of triangles.
 * @return True if improved without issues.
 */
virtual bool improveTriangulation(std::vector<Vertex> &vertices,
                                  std::vector<Edge> &edges,
                                  std::vector<Triangle> &triangles) = 0;
```

D.2. Añadir capacidad para leer más tipos de archivos

Para poder leer un nuevo tipo de archivos, se requiere extender la clase *FileHandler* e implementar sus dos métodos. Luego, se debe registrar el nuevo motor en el atributo *m_fileManager* de la clase *ModelImpl* [1]. Alternativamente, se puede registrar el nuevo motor en el constructor de la clase *FileManager*.

El método `load` se encarga de cargar la triangulación, y para ello requiere conocer el *path* del archivo a cargar, y acceso a los tres vectores de elementos geométricos para guardar ahí la información. Retorna un valor booleano que indica si se cargó correctamente o no el archivo.

```
/**
* @brief Method that loads a mesh file and modifies the parameters
* according to the loaded triangulation.
*
* @param filepath Path of the file.
* @param vertices Vector of vertices.
* @param edges Vector of edges.
* @param triangles Vector of triangles.
* @return True if correctly loaded.
*/
bool load(std::string filepath,
          std::vector<Vertex> &vertices,
          std::vector<Edge> &edges,
          std::vector<Triangle> &triangles) override;
```

[1] `m_fileManager.addFileHandler(handler, extension)`

Por otro lado, el método `save` se encarga de guardar la triangulación, y para ello requiere conocer el *path* del archivo a guardar, y acceso a los tres vectores de elementos geométricos para leer la información y escribirla en el archivo. Retorna un valor booleano que indica si se guardó correctamente o no el archivo.

```
/**
* @brief Method that saves a mesh file according to the actual parameters.
*
* @param filepath Path of the file.
* @param vertices Vector of vertices.
* @param edges Vector of edges.
* @param indices Vector of triangles.
* @return True if correctly saved.
*/
bool save(std::string filepath,
          std::vector<Vertex> &vertices,
          std::vector<Edge> &edges,
          std::vector<Triangle> &triangles) override;
```