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

REINGENIERÍA DE CAMARON: UN VISUALIZADOR DE MALLAS DE POLÍGONOS Y POLIEDROS

MEMORIA PARA OPTAR AL TÍTULO DE
INGENIERO CIVIL EN COMPUTACIÓN

JULIO ESTEBAN ALBORNOZ VALENCIA

PROFESOR GUÍA:
NANCY HITSCHFELD KAHLER

MIEMBROS DE LA COMISIÓN:
BENJAMÍN BUSTOS CÁRDENAS
JOSÉ SAAVEDRA RONDO

Este trabajo ha sido parcialmente financiado por Proyecto Fondecyt N°1211484

SANTIAGO DE CHILE
2022

Resumen

Camaron es una herramienta de visualización enfocado en la manipulación de mallas tridimensionales mixtas, permitiendo trabajar sobre una gama de diferentes formatos de archivos, otorgando la capacidad de poder evaluar el contenido de estos en tiempo real a través de diferentes operaciones disponibles al usuario. Este software fue desarrollado por alumnos del departamento del DCC, enfocado en la extensibilidad de sus componentes internos y en la velocidad de respuesta en las interacciones entre usuarios. Debido al numero importante de cambios realizados desde el año 2012 a la fecha, el software ha alcanzado un nivel de complejidad alto, acumulando deuda técnica que ha dificultado su desarrollo, permaneciendo el proyecto inactivo desde el año 2017.

El objetivo de esta memoria se centra en la realización de un análisis exhaustivo que permita diagnosticar el estado del proyecto, recopilando información relacionada a su funcionamiento así como a fallas presentes en el sistema, permitiendo la implementación de mejoras en el rendimiento de la aplicación, la corrección de falencias y la actualización de la documentación de *Camaron*. Por otro lado este proyecto busca aportar con el trabajo futuro sobre este sistema, otorgando herramientas de desarrollo en forma de tests unitarios y documentación técnica que permitan guiar el desarrollo de la aplicación tras este trabajo.

Durante el transcurso de este proyecto, se realizaron modificaciones estructurales al sistema, se logro compilar una lista de 52 fallas presentes dentro del sistema, de las cuales 30 pudieron ser corregidas. Al mismo tiempo se construyo un conjunto de tests compuesto por 98 rutinas individuales, las cuales permiten asegurar el funcionamiento correcto de una parte significativa del proyecto, así como hacer visible ciertas falencias aun existentes dentro del sistema. El proyecto incluyo modificaciones importantes sobre la arquitectura interna de *Camaron*, la cuales permitieron una reducción en el consumo de memoria RAM promedio de un 18 %, mas una mejora en la velocidad de ejecución promedio correspondiente a un 75 % con respecto a la versión original.

A mi familia.

Agradecimientos

A mi familia que me ha dado un soporte incondicional durante mi formación académica y a los amigos que he tenido dentro y fuera del ambiente académico, en especial a las personas del departamento de computación.

Tabla de Contenido

Índice de Tablas

Índice de Ilustraciones

Capítulo 1

Introducción

En los últimos años, la tecnología asociada a la digitalización de objetos tridimensionales ha realizado avances importantes dentro del área científica, permitiendo capturar la complejidad de estos objetos con un alto nivel de precisión, a través de diferentes técnicas de escaneo 3D (Structured-Light-Scanning (SLS) y Structure from Motion (SfM), entre otras), permitiendo a investigadores la capacidad de poder trabajar sobre estos objetos virtuales sin comprometer la integridad de los objetos originales.

Si bien estos avances han mejorado la capacidad de recuperación de estos objetos, el procesar estos modelos ha significado un desafío técnico importante, ya que en algunos casos el nivel de resolución exigido puede llegar a ser demasiado alto (ej: obtención de altura en mapeo digital, reconstrucción de objetos arqueológicos [1], etc), generado mallas geométricas de gran tamaño que hagan dificultoso su procesamiento. Dependiendo del caso es posible reducir la calidad de los modelos para poder permitir su procesamiento, sin embargo esta reducción no puede aplicarse de forma exhaustiva, debido a que este puede modificar de forma irreparable al modelo, así como incurrir en la pérdida de detalles importantes los cuales estuviesen presentes dentro del objeto original (ej: huellas dactilares [2], caracteres pequeños, etc).

Ante esta necesidad es que surgen diferentes herramientas especializadas en la visualización científica, con el objetivo de asistir en la investigación y en el diseño a través de sistemas computacionales, permitiendo la visualización y manipulación de objetos tridimensionales. Estos objetos pueden ser representados por mallas de polígonos que describan su superficie o por modelos sólidos (volumen) descritos por poliedros, adaptándose a los recursos disponibles por el usuario (memoria RAM, cores de CPU, GPU), para así lograr un buen nivel de performance durante la ejecución. Dentro de este tipo de software se encuentra el visualizador *Camaron*, que entre sus funcionalidades principales posee:

- Visualización de mallas de polígonos (convexas o no), que representan la superficie de objetos, que pueden ser cerradas o no.
- Visualización de mallas sólidos de hasta 2 millones de poliedros, tales como tetraedros y hexaedros.

- Evaluación y cálculo de estadísticas para diferentes métricas, tales como: ángulo mínimo, ángulo máximo y área, entre otras, aplicadas a cada elemento que compone al objeto 2D o 3D según correspondan.
- Diferentes métodos de renderizado, permitiendo combinar filtros sobre un mismo modelo
- Importación y exportación de modelos para diversos formatos (e.g: .off, .m3d, .visf, .ply, etc.)
- Multiplataforma (Linux, Windows)
- Capacidad de visualizar campos de propiedades intrínsecas a través de isolíneas e isosuperficies

La primera versión de este software fue desarrollado por el ex-alumno del Departamento de Ciencias de la Computación (DCC) Aldo Canepa como parte de su trabajo de título[3] en el año 2012. Se encuentra desarrollado en el lenguaje C++ y utiliza para la visualización de las mallas la API gráfica OpenGL, en conjunto con el framework para aplicaciones visuales Qt5. El objetivo principal de este software es permitir a un usuario visualizar mallas de polígonos y poliedros, visualizar los valores obtenidos en simulaciones y generar estadísticas de los elementos de la malla.

Durante el año 2016 el ex-alumno del DCC Gonzalo Infante Lombardo realizó una serie de mejoras al programa como parte de su memoria de título [4], enfocándose principalmente en la representación de campos de propiedades ya codificadas en los archivos, así como la implementación de Isolineas e Isosuperficies basadas en esta información y la extensión del software para representar ejes adicionales al modelo. En segunda instancia se inició un proceso de refactorización del código fuente, implementando funcionalidades del estándar C++11 como punteros inteligentes y streams de datos.

1.1. Motivación

Si bien *Camaron* posee las capacidades necesarias para poder competir en el área de la visualización gráfica, una de las mayores desventajas presente en el software corresponde al consumo de memoria RAM para modelos de gran tamaño.

Esta limitación fue presentada por los trabajos anteriores como una decisión de diseño para priorizar la velocidad de respuesta de la aplicación por sobre los recursos utilizados, lo cual se puede evidenciar en la velocidad de respuesta tras el proceso de carga de modelos. Lamentablemente estos beneficios no pueden ser aplicados en el caso de modelos de gran tamaño, ya que la cantidad de recursos utilizados evitan que la aplicación logre terminar el proceso de carga en cuestión.

Si bien esta es una limitante que siempre puede aparecer al aumentar el tamaño de los modelos, actualmente algunas de las alternativas mostradas anteriormente son capaces de

cargar estos modelos de gran tamaño, por lo que existe precedente para evaluar posibles mejoras que permitan corregir esta falencia.

Cabe notar que dentro del código de *Camaron* existen variantes de ciertos componentes con el objetivo de sobrepasar esta limitante, pero que al ser utilizadas resultan en una reducción significativa de la funcionalidad de *Camaron*, por lo que tampoco logran entregar una respuesta completa a esta problemática.

En base a lo anterior, este trabajo plantea desarrollar un análisis exhaustivo del código fuente existente, con el objetivo de determinar que componentes consumen la mayor cantidad de memoria y mayor tiempo de CPU, para así poder mejorar el performance de Camaron sin comprometer la funcionalidad que diferencia a *Camaron* como herramienta de visualización científica.

Finalmente, el resultado de este análisis busca facilitar el desarrollo futuro de la aplicación, otorgando documentación actualizada de componentes claves, tests unitarios para validar componentes y el registro de las falencias existentes en el sistema, así como sus posibles soluciones.

1.2. Objetivos

Objetivo General

Realizar un proceso de refactorización para robustecer la funcionalidad existente de Camaron, restaurar funcionalidad que ya no se encuentre utilizable, así como implementar cambios en la arquitectura del proyecto para hacer un uso eficiente de la memoria por parte del software, sin comprometer la eficiencia en tiempo de forma significativa.

Objetivos Específicos

1. Robustecer la implementación actual, permitiendo que la aplicación trabaje de forma correcta ante diferentes casos de uso y entregando errores legibles que mejoren el proceso de debugging.

2. Mejorar el rendimiento de la aplicación, encontrando un balance óptimo entre consumo de memoria RAM y de velocidad de procesamiento.

3. Implementar estándares de calidad y pruebas automatizadas para facilitar la legibilidad y extensibilidad del código a nuevos desarrolladores

4. Crear documentación técnica detallada para asistir a desarrolladores que utilicen la aplicación de forma efectiva

1.3. Metodología

Durante el transcurso del proyecto, se realizaron reuniones semanales con la profesora guía, con el objeto de documentar el comportamiento esperado de los componentes de *Camaron*, así como discutir posibles optimizaciones, cursos de acción y deprecaciones de cierto contenido. Se desarrolló el proyecto a través de un trabajo de análisis, documentación y testing sobre los componentes principales de el software, enfocándose en la resolución de las falencias mas importantes dentro del sistema.

1.3.1. Ambiente de Trabajo

El desarrollo de este trabajo fue realizado en un computador portátil con las siguientes características técnicas:

- OS: Windows 10 64bits
- Memoria Ram: 8GB
- Procesador: Intel i5 (8CPUs)
- GPU: NVIDIA GeForce GTX 1050

Al mismo tiempo, el proyecto fue compilado en los siguientes ambientes de desarrollo para asegurar la portabilidad entre sistemas operativos:

- Windows: Mingw64 10.0, g++ 10.2
- Linux: Maquina virtual con imagen de Lubuntu v20.10, g++ 10.2

Pór ultimo, en la primera mitad del proyecto se utilizó la versión Qt5.4 para coincidir con la versión de Qt utilizada en la iteración anterior, mas tarde se actualizó esta dependencia a su version LTS Qt5.15.2, la cual puede ser descargada de forma directa a través del paquete *mingw-w64-x86_64-qt5-static* en caso de Windows.

1.3.2. Repositorio

Se realizó un fork al repositorio utilizado en la iteración anterior, llevando registro de las modificaciones principales en la branch central *refactor*. Las modificaciones de cada componente individual fueron desarrollados dentro de branchs separadas, las cuales son unidas al branch central tras completar ciertos hitos. El código fuente puede ser encontrado en el siguiente link: (`https://sourceforge.net/u/jalbornozv/camaron/ci/refactor/tree/`)

1.4. Contenido de la Memoria

En el capítulo de *Antecedentes* se entrega información sobre alternativas existentes a Camaron, para luego realizar una descripción detallada del estado original de *Camaron* previo al trabajo de refactorizado.

En el capítulo de *Análisis* se realiza un análisis técnico sobre la implementación de *Camaron*, documentando falencias y anomalías específicas de cada componente dentro de *Camaron* e incluyendo mediciones de performance sobre el software original.

En el capítulo de *Refactorizacion de Camaron* se detallan las modificaciones y correcciones realizadas durante el transcurso del proyecto. Posterior a estas correcciones se incluye un resumen de los tests implementados para asegurar el funcionamiento del sistema.

En el capítulo de *Resultados* se detallan los efectos de estas modificaciones sobre el software, incluyendo comparaciones de rendimiento entre versiones, mejoras en mantenibilidad y portabilidad del sistema entre otros

En el capítulo de *Recomendaciones* se describe una serie de modificaciones que fueron consideradas durante el desarrollo pero que no pudieron ser implementadas en el tiempo disponible. Posterior a esto se listan las falencias detectadas durante el capítulo de *Análisis y Metodología* que no pudieron ser corregidas durante el transcurso del proyecto, organizadas según el conocimiento presente a la fecha.

Por último el trabajo concluye con un capítulo de *Conclusiones*, el cual entrega una síntesis del trabajo realizado mas un listado de modificaciones a largo plazo que podrían ser utilizados en trabajos de memoria posteriores.

Capítulo 2

Antecedentes

En este capítulo se realiza una comparación inicial entre *Camaron* y otras herramientas de visualización gráfica, para luego describir el estado inicial del software previo al trabajo realizado.

La sección de estado inicial, entrega una visión general de la arquitectura del sistema, para luego ahondar sobre cada componente interno asociado a las modificaciones que serán realizadas en secciones posteriores.

2.1. Alternativas Existentes

A la fecha existe una variedad de visualizadores disponibles, especializándose cada uno en diferentes facetas dentro del espacio del problema. Si bien existen alternativas propietarias con mayores funcionalidades, nos remitiremos a comparar *Camaron* con otras herramientas de código abierto, las cuales son:

- MayaVi [5]: Software implementado en Python que permite la visualización de diferentes tipos de datos de interés científico, tales como campos escalares y vectoriales.
- VisIt [6]: Software de visualización para mallas bidimensionales, enfocado en el estudio de campos vectoriales a gran escala.
- ParaView [7]: Software para procesamiento y visualización 3D, permitiendo utilizar su funcionalidad a traves de una API en C++ para poder construir pipelines de procesamiento de datos propios.
- GigaMesh [8]: Software diseñado para la visualización de mallas bidimensionales, especializándose en la visualización de objetos arqueológicos a través de escaneo 3D.

Si bien estas herramientas permiten replicar diferentes aspectos de *Camaron*, estos no sustituyen a este de forma completa. En particular la aplicación tiene un carácter generalista, permitiendo no solo la manipulación de objetos 3D completos o incompletos, representados

por polígonos y poliedros convexos y en formatos poco comunes, sino también siendo capaz de realizar evaluaciones y selecciones dentro de estos objetos.

2.2. Comparación de Capacidades

Tipos de Modelos

	Nube de Vértices	Malla de polígonos	Malla de Poliedros	Malla mixta
Camaron	Si	Si	Si	Si
GigaMesh	Si	Si	No	No
MayaVi	No	Si	No	No
ParaView	Si	Si	Si	Si
VisIt	No	Si	Si	Si

Tabla 2.1: Soporte de diferentes tipos de modelos. Una malla mixta contiene polígonos y poliedros dentro de este.

Renderizadores

	Wireframe	Phong Shading	Transparencia	Intersección Geometrías	Vértices
Camaron	Si	Si	Si	Si	Si
GigaMesh	Si	Si	Si	Si	Si
MayaVi	Si	Si	Si	No	Si
ParaView	Si	Si	Si	Si	Si
VisIt	Si	Si	Si	Si	No

	Profundidad	Elevación	Normales	IsoLineas	IsoSuperfices
Camaron	Si	Si	Si	Si	Si
GigaMesh	No	Si	No	Si	No
MayaVi	No	No	Si	No	No
ParaView	No	Si	Si	Si	Si
VisIt	No	No	Si	Si	Si

Tabla 2.2: Renderizadores Disponibles

Calidad de modelo

	Area	Superficie	Volumen	Ángulos Internos	Ángulos Dihedros	Ángulos Solidos	Radio	Conteo Caras
Camaron	Si	Si	Si	Si	Si	Si	Si	Si
GigaMesh	No	No	Si	No	No	No	No	No
MayaVi	No	No	No	No	No	No	No	No
ParaView	Si	No	Si	Si	No	Si	Si	Si
VisIt	Si	No	No	Si	No	No	Si	Si

Tabla 2.3: Evaluación de parámetros por componentes

Soporte de formatos

	OFF	PLY	TRI	Node/Ele	Obj	Collada	Alembic
Camaron	Ambos	Importe	Importe	Importe	No	No	No
GigaMesh	No	Ambos	No	No	No	No	No
MayaVi	No	Importe	No	No	No	No	No
ParaView	No	Ambos	No	No	Ambos	Importe*	Importe*
VisIt	No	Ambos	No	No	No	No	No

Tabla 2.4: Capacidad de Importe y Exporte de cada alternativa. Notar que *Camaron* también permite la manipulación de los formatos propios VisF, TS, M3D, TQS, los cuales no son listados al no ser parte de algún estándar externo. En el caso de Paraview los formatos marcados con * pueden ser importados a través de la API ofrecida por la aplicación

2.3. Estado inicial del proyecto

2.3.1. Arquitectura general

A grandes rasgos, *Camaron* fue diseñado bajo la arquitectura de tipo modelo-vista-controlador, con los siguientes componentes principales:

Interfaz gráfica Localizada principalmente en el directorio *UI*. Cada *widget* es implementado a través de una subclase de *QWidget*, que especifica su interacción con el sistema, mas un archivo .ui, el cual describe la apariencia gráfica del *widget*. En el caso de estrategias de selección y de renderizadores, estos poseen *widgets* propios que se localizan en sus directorios respectivos.

Representación de Modelos Contenedores utilizados para representar el modelo siendo manipulado por el usuario. Estos contenedores corresponden a la clase *Model* y *RModel*, las cuales almacenan su contenido en CPU y GPU, respectivamente.

Controlador Aplicación Este componente se encuentra implementado dentro de la clase *Visualizador*, la cual recibe señales de la interfaz para iniciar la ejecución de la funcionalidad escogida por el usuario.

Componentes Corresponden a módulos del sistema los cuales aplican una operación sobre el modelo existente, manipulando principalmente a la clase *Model* y actualizando el contenido de *RModel* de ser necesario. Cada módulo posee múltiples estrategias, cada una siendo registrada dentro de un objeto de tipo *singleton*, implementados dentro del directorio *Factories*.

Renderizadores Clases encargadas de mostrar el modelo en pantalla. Cada implementación presente utiliza la pipeline gráfica de OpenGL para visualizar una característica especifica del modelo a través de shaders. Este tipo de clase opera principalmente con los buffers almacenados en GPU, utilizando *RModel* para verificar si existieron modificaciones externas.

A continuación se especificará en mayor detalle el funcionamiento de cada una de estas secciones:

2.3.2. Elementos Geométricos

Corresponden a los objetos geométricos básicos utilizados para componer un modelo 3D. Cada clase implementada hereda de una clase base *Element*, la cual es luego extendida para representar un componente específico. Actualmente se encuentran disponibles los siguientes elementos:

Vertex
Elemento mas simple dentro de la jerarquía, descrito por un vector de coordenadas tridimensional correspondiente a su posición.

Edge
Representación de una arista adicional, descrita por dos vértices y dos colores asociados a cada uno.

Polygon
Representación de un polígono convexo individual compuesto de tres o mas vértices, siendo listados en función del orden explicitado en cada formato. A su vez esta clase posee un especialización de tipo *Triangle*, la cual almacena información de cada arista del triangulo para optimizar el calculo asociado a este tipo de elemento.

Polyhedron
Representación de un poliedro individual, compuesto por cuatro o mas polígonos representando las caras de este. Originalmente *Camaron* incluía un subtipo para representar tetraedros (*Tetrahedron*), pero este no poseía una implementación completa por lo que este no era utilizada.

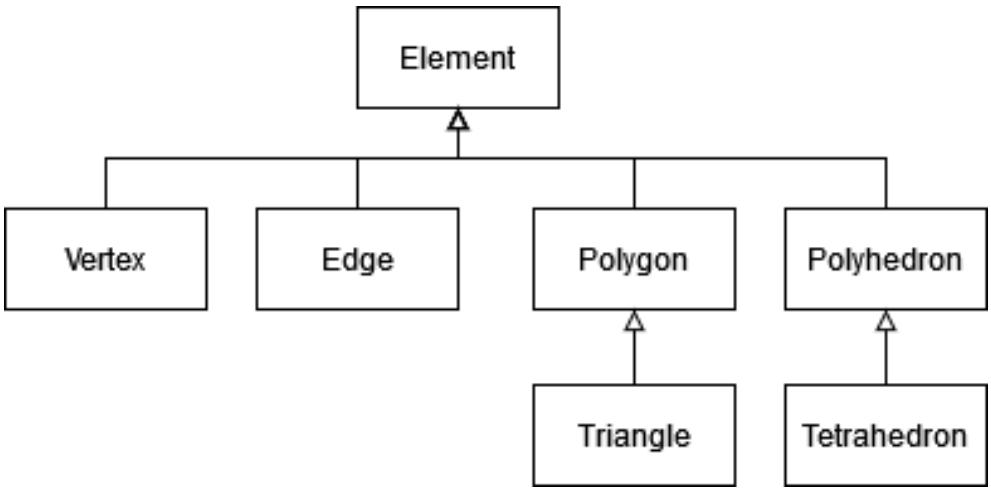


Figura 2.1: Jerarquía de clases de tipo Element

Debido a que *Camaron* fue diseñado con el objeto de reducir el trabajo realizado por la CPU al mínimo. Cada elemento posee un objeto de tipo *LightMemoryHash*, el cual actúa como cache interno para almacenar propiedades resultantes de la evaluación de un modelo o de la carga de un campo escalar. Al mismo tiempo estas clases almacenan información adicional a cada componente, así como las relaciones de vecindad entre cada componente. Estas relaciones de vecindad se listan a continuación:

- Un vértice posee referencias a cada polígono del que es parte
- Un polígono posee referencias a los polígonos vecinos a este
- Un polígono almacena las referencias sobre los poliedros del que es parte. En particular este tipo de vecindad asume que un polígono solo puede ser una cara de máximo dos poliedros.

Al mismo tiempo, se definió una jerarquía de clases similar a la mencionada en la figura 2.1, con el objeto de intentar reducir el consumo de RAM dentro del modelo. Estas clases son implementadas bajo el *namespace* LW (Lightweight), correspondiendo a las clases *LWElement, LWVertex, LWPolygon y LWPolyhedron* (Ver fig 2.2). Estas clases solo poseen información relacionada a sus componentes internos, sin cache interno ni referencias de vecindad asociada.

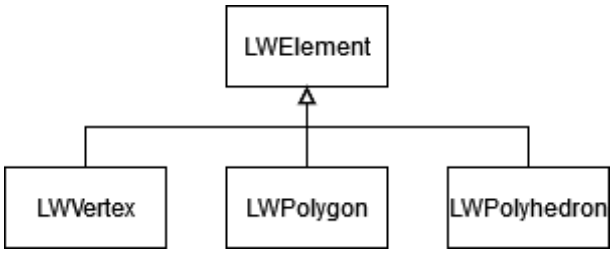


Figura 2.2: Jerarquía para variantes de bajo consumo

2.3.3. Modelo

Camaron representa un modelo como un conjunto de elementos geométricos, almacenando en su interior referencias a cada objeto alojado en memoria principal. En base a los elementos mostrados en la sección anterior, se implementaron las siguientes clases:

Vertex Cloud Modelo solo contiene vértices y ejes adicionales

Polygon Mesh Malla compuesta exclusivamente de polígonos

Polyhedron Mesh Malla compuesta exclusivamente de poliedros

Si bien en teoría una malla debiese contener solo un tipo de elemento, debido a como *Camaron* representa a estos elementos en memoria, es imposible representar elementos complejos sin la definición previa de sus elementos constitutivos (Ej: Un polígono no puede existir en el sistema sin antes crear los vértices que lo componen). Debido a esto cada tipo de malla debe incluir todos los componentes de la subclase anterior a este. Originalmente se diseño la jerarquía de clases en base a una relación de herencia en cadena la cual se muestra en la siguiente imagen:

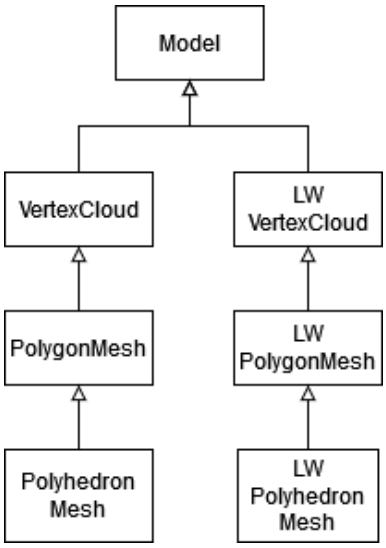


Figura 2.3: Jerarquía de clases asociada a la clase Model

La clase base *Model* posee información común asociada a otros sistemas dentro de Camaron, mientras que cada subclase definido agrega un vector con el tipo de elemento que este representa, extendiendo la interfaz para permitir el accesos a este. En base a esto un modelo complejo se define como una clase hija de un modelo mas simple.

Estas clases también poseen una jerarquía paralela asociada a implementaciones de bajo consumo, las cuales corresponden a las clases *LWModel, LWVertexCloud, LWPolygonMesh y LWPolyhedronMesh*, los cuales a su vez poseen referencias a los elementos LW asociados.

2.3.4. Visualizador

Clase principal de la aplicación, encargada de conectar la interacción del usuario en la interfaz gráfica con los componentes presentes dentro del sistema. Para lograr lo anterior, al inicio del programa el objeto creado inicializa cada *Widget* de la interfaz, conectandolas con subrutinas pertenecientes a cada funcionalidad del sistema a través del mecanismo de *slots* presente en Qt5.

Durante la ejecución del programa, el objeto creado mantiene una referencia a los objetos *Model* y *RModel* asociados al modelo 3D siendo manipulado por el usuario, así como una lista de las estrategias disponibles por cada componente del sistema, estas se encuentran almacenadas en un sistema de registros que es definido durante la etapa de compilación del programa.

A la hora de necesitar el trabajo de un componente, el objeto solicita al registro la estrategia escogida por el usuario, la cual es luego activada utilizando los puntos de acceso definidos dentro del modelo activo.

2.3.5. RModel

Esta clase tiene como función principal la de actuar como puente entre el modelo 3D almacenado en memoria principal y la representación localizada dentro de la GPU. Para ello esta clase tiene las siguientes responsabilidades:

2.3.5.1. Triangulación de modelos y transferencia de datos a GPU

Tras recibir un modelo recién ingresado a *Camaron*, *RModel* es encargado de convertir los elementos geométricos presentes en una representación de triángulos, la cual pueda ser ingresada a la GPU a través de la API OpenGL posteriormente. Para ello se sigue los siguientes casos:

1. Si el modelo corresponde a una nube de vértices o ya se encuentra descrito por triángulos, este continua el proceso de cargado.
2. Si el modelo contiene polígonos con mas de tres vértices, *RModel* utiliza el método *PolygonUtils::getTriangleVertices*, el cual utiliza el algoritmo *Fan triangulation* (ver Figura 2.4)
3. En el caso de contener poliedros con caras no triangulares, el poliedro es descompuesto en un conjunto de tetraedros.

Ambos algoritmos de triangulación son aplicados de forma automática tras completar el proceso de procesamiento de mallas (ver figura 2.6), el cual es iniciado a través del método *loadRModelData* a través de *double dispatch*. Durante este proceso, a cada elemento que debe ser triangulado se le incluye la posición de los triángulos que lo componen dentro de esta nueva

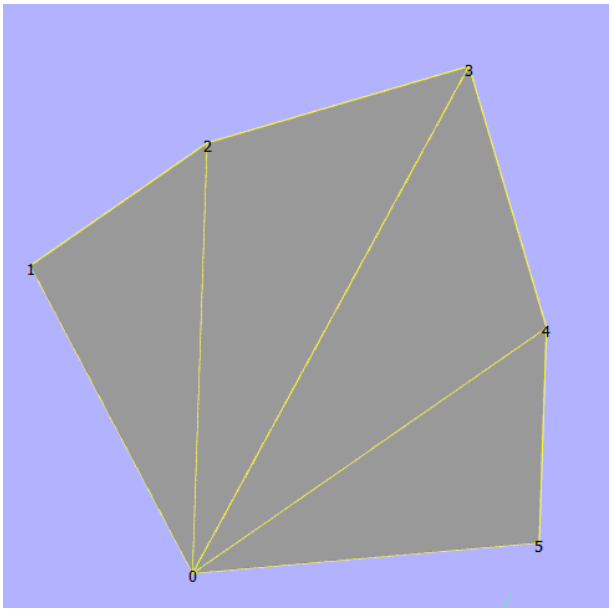


Figura 2.4: Triangulación de poliedros. Esta operación resulta en la generación de (numero de vertices - 2) triangulos, siempre partiendo desde el vértice inicial

representación. Es importante mencionar que estos algoritmos requieren que los polígonos a descomponer sean convexos, por lo que si el modelo presenta geometría no convexa, esto puede generar artefactos visuales.

Tras este proceso, *RModel* utiliza buffers de Opengl creados a través de *glBufferData*, para transferir la descomposición generada mas la información requerida por los renderizadores implementados, esta información consiste en:

- Posición de cada vértice en el modelo
- Vectores normales de cada vértice
- Posiciones y colores de cada arista adicional
- Relaciones de poliedro->polígono

2.3.5.2. Actualización de modelo en pantalla

RModel utiliza la clase interna *RVertexFlag* para almacenar atributos extra de cada vértice. Esta información se encuentra dentro de un vector propio, el cual es también almacenado dentro de la GPU, manteniendo a ambos contenedores actualizados a medida que se realizan modificaciones en el sistema.

Por otro lado, *RModel* utiliza un buffer adicional para representar el campo de propiedades activo. Este contenedor es actualizado a través del método *loadPropertyField* al momento de escoger un nuevo campo a visualizar por los renderizadores de Isolineas e IsoSuperficies.

2.3.5.3. Manipulación de Cámara

Camaron permite operar la cámara al arrastrar el cursor con click derecho, con la restricción de que la cámara siempre se encuentra orientada hacia el punto central del modelo. Para actualizar el *viewport* durante el renderizado, *RModel* almacena un conjunto de matrices de transformación, que pueden ser actualizadas al interactuar con el modelo. Estas matrices son luego utilizadas por los *Renderizadores* para ser incluidos en la pipeline de renderizado.

2.3.6. Componentes Disponibles

2.3.6.1. Carga de Modelos

Uno de los componentes principales dentro de *Camaron* corresponde a la carga de modelos externos. En particular el software entrega soporte para los formatos externos PLY, OFF y ELE/NODE, incluyendo también la lectura de los formatos experimentales TRI, TS, M3D y VisF. El detalle de como cada especificación es manejada, se especificará dentro del Apéndice A adjunto a este documento.

El registro de este módulo se encuentra implementado en la clase *ModelLoadingFactory*. Cada estrategia implementada es aplicada a un formato específico, heredando de la clase abstracta *ModelLoadingStrategy* y reimplementando los métodos *validate* y *load*. En paralelo *Camaron* incluye implementaciones de carga para modelos de tipo *Lightweight*, para los formatos PLY, ELE/NODE y OFF.

Internamente, el componente es ejecutado en un thread separado del proceso principal, al ser este una clase derivada de la clase *QThread*. Para iniciar el proceso de carga, el visualizador obtiene la estrategia de carga asociada desde la extensión del archivo, para luego utilizar el método *loadModelQThread* para iniciar el proceso de carga. (Ver proceso completo en diagrama 2.5)

El proceso de carga implementado por el método *load* se compone de las siguientes etapas:

1. Lectura de encabezado de archivo para obtener información preliminar sobre contenido
2. Creación de modelo inicial y reserva de memoria en caso de obtener el número de elementos a almacenar
3. Lectura de elementos descritos en el archivo, realizando la construcción de cada elemento dentro del *heap* como subclases de tipo *Element*.
4. Procesamiento de modelo creado

El proceso de lectura es realizado por un objeto de clase *CharArrayScanner*, el cual opera sobre un buffer de caracteres que contiene la totalidad del archivo objetivo. En particular el último paso corresponde al calculo de las relaciones de vecindad, computación de normales de polígonos, entre otros. Este proceso solo es aplicable a mallas de polígonos y poliedros, con la mayoría de estas etapas siendo aplicada de forma paralela.

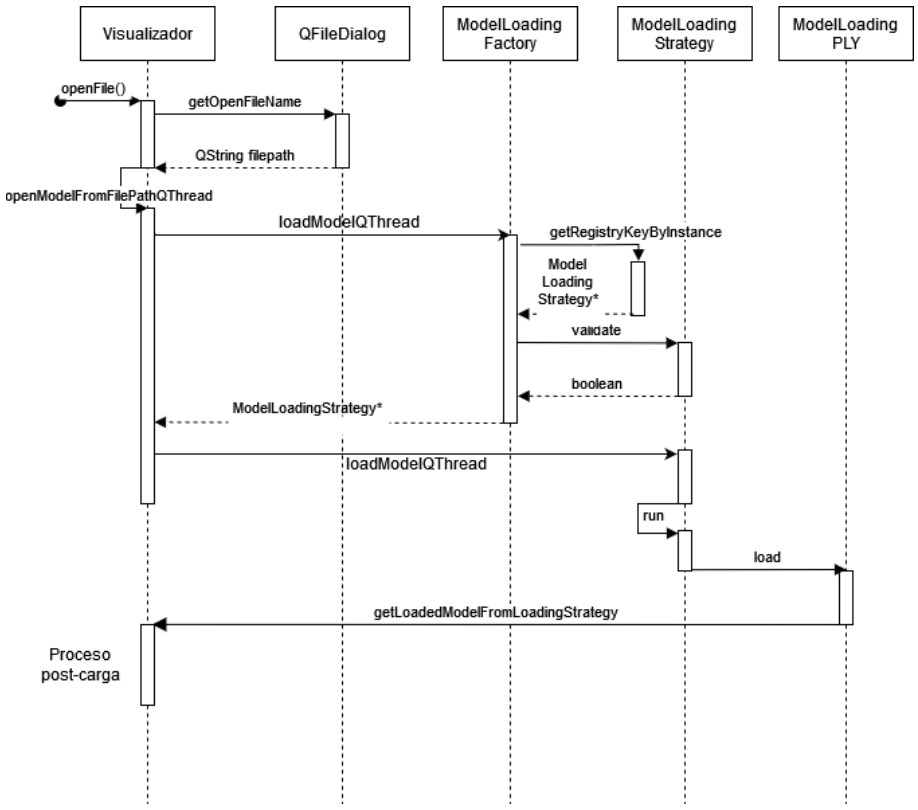


Figura 2.5: Diagrama de interacción entre Visualizador y Módulo de carga de modelos. Notar que al pedir la estrategia de carga, se realiza una validación previa al contenido del archivo

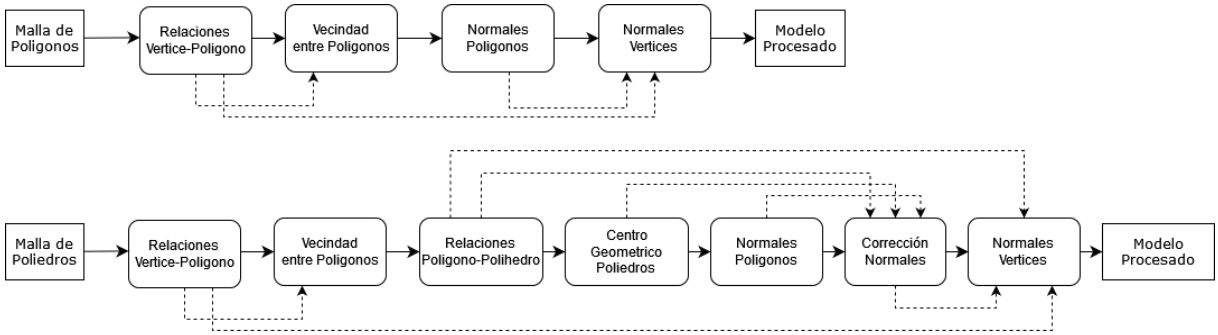


Figura 2.6: Diagrama de flujo para etapa de procesamiento de mallas. Lineas sólidas corresponden al orden en que estas etapas son ejecutadas, mientras que las lineas punteadas corresponden a dependencias entre los resultados de estas con las etapas posteriores que las necesitan

Notar que esta última etapa trabaja sobre el modelo construido y no sobre el archivo en cuestión, ya que en términos de funcionalidad el procesamiento de una malla correspondería a un proceso independiente.

2.3.6.2. Exportación de Modelos

Camaron posee la capacidad de convertir un modelo previamente cargado a un formato de almacenamiento diferente al original. Actualmente es capaz de transferir modelos a los formatos Ele/Node, VisF, OFF y TS. Al mismo tiempo la aplicación posee la capacidad de almacenar el contenido de una selección específica, lo cual solo se encuentra habilitada para transferir a formatos OFF y VisF.

Es importante mencionar que debido a las características de cada formato disponible, no es posible realizar ciertas conversiones entre archivos. En particular Camarón solo se remite a realizar operaciones hacia formatos que puedan describir los elementos específicos de cada modelo.

	PLY	Node/Ele	VisF	OFF	M3D	TS	TRI
Vertex Cloud	Si	No	Si	No	No	No	No
Polygon Mesh	Si	Si	Si	Si	No	Si	Parcial
Polyhedron Mesh	Si	Si	Si	No	Parcial	No	No

Tabla 2.5: Conversiones válidas entre tipo de modelo y formatos. Conversiones denominadas como parciales poseen requisitos adicionales descritos a continuación.

En el caso del formato TRI, solo es posible convertir mallas de poliedros compuestas exclusivamente por triángulos. Al mismo tiempo debido a que el formato M3D solo permite modelos descritos por un número limitado de prismas, ciertos modelos quedarían excluidos de esta conversión debido a su topología.

Si bien teóricamente sería posible describir un modelo complejo en base a sus elementos constituyentes (Ej: Malla de poliedros como malla de polígonos), una conversión de este estilo resultaría en la pérdida de información asociada a la identificación de los elementos mas complejos, por lo que no sería posible revertir esta operación. En vista de esto la aplicación considera a este tipo de conversión como una funcionalidad fuera del alcance del proyecto.

Otro punto a considerar es que este componente solo se remite a exportar la descripción geométrica de los elementos de cada modelo, por lo que información asociada a evaluaciones o campos de propiedades no son considerados a la hora de aplicar esta operación.

El registro de este módulo se encuentra implementado en la clase *ModelExportStrategyRegistry*. Al mismo tiempo cada estrategia implementada es aplicada a un formato especifico, las cuales heredan de la clase abstracta *ModelExportStrategy*, donde cada subtipo puede implementar los métodos *exportModel* o *exportSelection*, según el tipo de modelo que sea permitido. (Detalle interacción en diagrama 2.7)

2.3.6.3. Estrategia de Evaluación

Este componente tiene como objetivo aplicar métricas sobre elementos del mismo tipo, con el objetivo de obtener estadísticas que permitan evaluar la calidad de diferentes sectores dentro de un modelo.

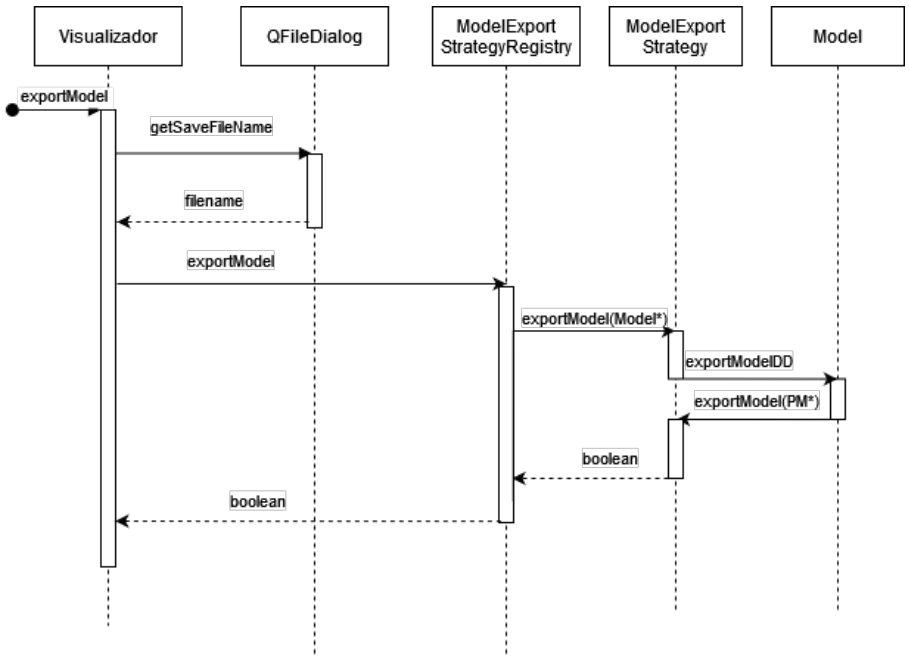


Figura 2.7: Diagrama de interacción entre Visualizador y Módulo de exporte de modelos, notar que en este caso el registro ejecuta la estrategia de forma directa

Esta funcionalidad puede ser invocada de tres formas dentro de *Camaron*:

1. Utilizar el widget de Evaluación directamente, el cual debe ser habilitado a través del menú *Windows->Evaluation Strategies*.
2. Escoger el Renderizador *Properties by Renderer*, utilizando el botón *Apply* tras escoger la estrategia a utilizar dentro del menu *Config* asociado.
3. Escoger la opción *Select by Property* dentro del *Widget* de Selección.

En los dos primeros casos, la funcionalidad esta implementada aplicando *double dispatch* sobre el modelo activo, tras identificar el conjunto de elementos que deben ser procesados. La estrategia utiliza *double dispatch* sobre cada elemento geométrico para invocar la función *value*. El valor calculado es almacenado dentro del cache interno presente en cada elemento y en un cache adicional presente dentro de la misma estrategia. (Ver detalle proceso en diagrama 2.8)

A la hora de renderizar los resultados, la estrategia obtiene los valores cacheados de cada elemento y los almacena en un buffer entregado por el renderizador, permitiendo reutilizar cómputos previos sobre un mismo modelo.

Asociada a esta funcionalidad se encuentra la posibilidad de crear un histograma de con los valores de la métrica. Para que este gráfico pueda ser generado, (a través de *Windows->Statistics->Element Statistics*) se debe aplicar la métrica previamente a través de los puntos de entrada mencionados anteriormente.

Las métricas de calidad actualmente implementados dentro de *Camaron* se listan a continuación:

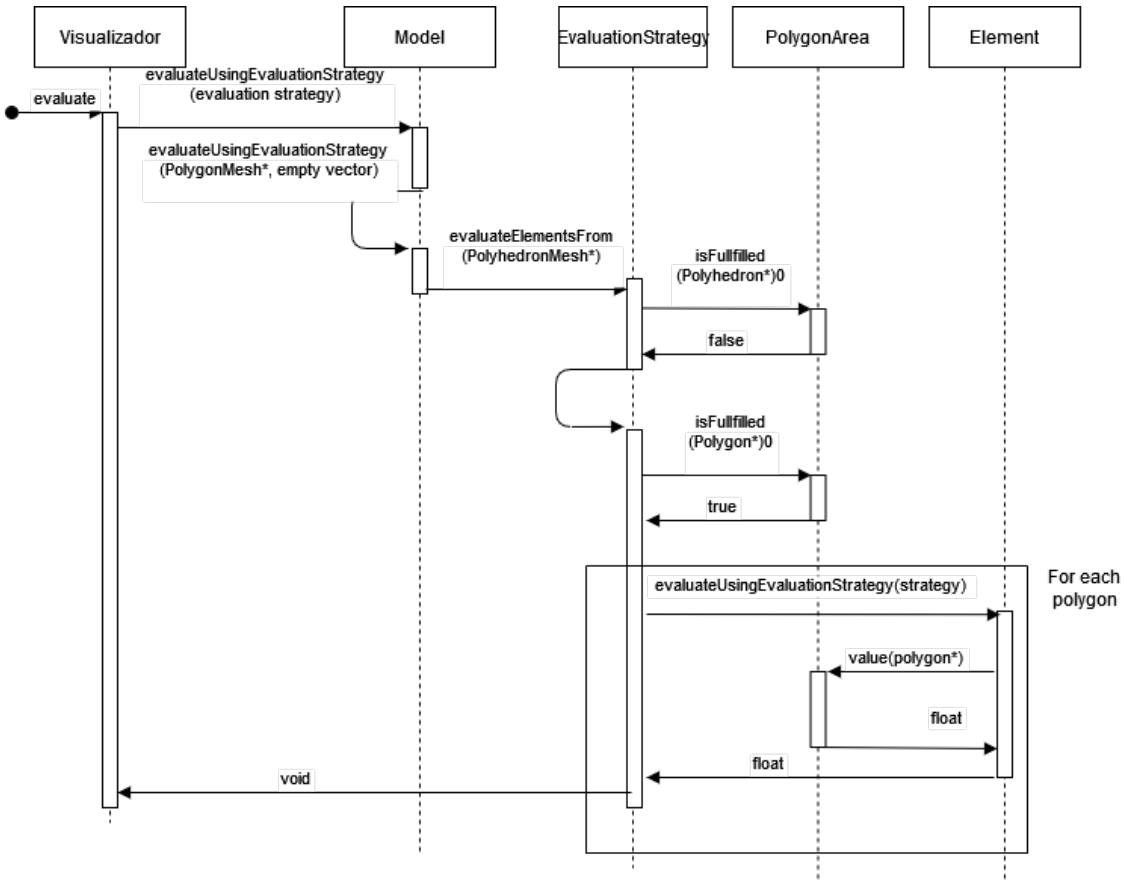


Figura 2.8: Diagrama de interacción entre Visualizador y Módulo de evaluación de modelos

- Polígonos
  - Área
  - Mínimo y Máximo ángulo dihedro
  - Mínimo y Máximo ángulo interno
  - Número de Vértices
  - Relación de aspecto
  - Relación de radio
- Poliedros
  - Superficie
  - Volumen
  - Mínimo y Máximo ángulo sólido
  - Número de Caras
  - Relación de aspecto
  - Relación de radio
  - Criterio de Bhatia & Lawrence

2.3.6.4. Carga de Campos de Propiedades

Como complemento a la información obtenida por las estrategias de evaluación, *Camaron* también permite la carga de propiedades calculadas de forma externa. Estas propiedades son almacenadas como campos escalares. Este tipo de datos pueden ser encontrados dentro de los formatos *PLY, ELE/NODE, TQS y OFF*, con *Camaron* entregando soporte para los dos primeros formatos, mas una implementación inconclusa para la extensión TQS, inicialmente excluida dentro del proceso de compilación de *Camaron*. Internamente *Camaron* representa un campo escalar a través de la clase *PropertyFieldDef*, la cual contiene la metadata de un campo específico, mientras que los valores del archivos son almacenados dentro del cache interno de cada elemento geométrico.

La carga de estos campos es ejecutada de forma automática tras la carga de un modelo o iniciada de forma manual a través del menú *File->Import Property Field*. Similar al componente de carga de archivos, este componente utiliza su propio *thread* de ejecución, al mismo tiempo el componente debe determinar el tipo de elemento geométrico que se piensa cargar, lo cual es realizado a través del patrón *visitor*. El proceso de carga de campos considera los siguientes pasos durante su ejecución:

1. Obtener nombre de archivo a cargar en base a input de usuario
2. Obtener estrategia asociada a la extensión del archivo
3. Compilar lista de campos de propiedades presentes en el archivo (*loadDefs*)
4. Pedir a usuario que escoja definiciones a cargar
5. Cargar contenido dentro de cada elemento geométrico asociado al campo
6. Almacenar definición de campo dentro de *Model* para uso posterior

La información contenida es utilizada posteriormente por los renderizadores de *Isolineas* e *IsoSuperficies*. Para permitir el traspaso de esta información, la clase *RModel* utiliza un buffer dentro de la GPU para almacenar el campo de propiedades actualmente activo, utilizando la clase *RModelPropertyFieldDef* para lograr este cometido. (Ver detalle del proceso en diagrama 2.9)

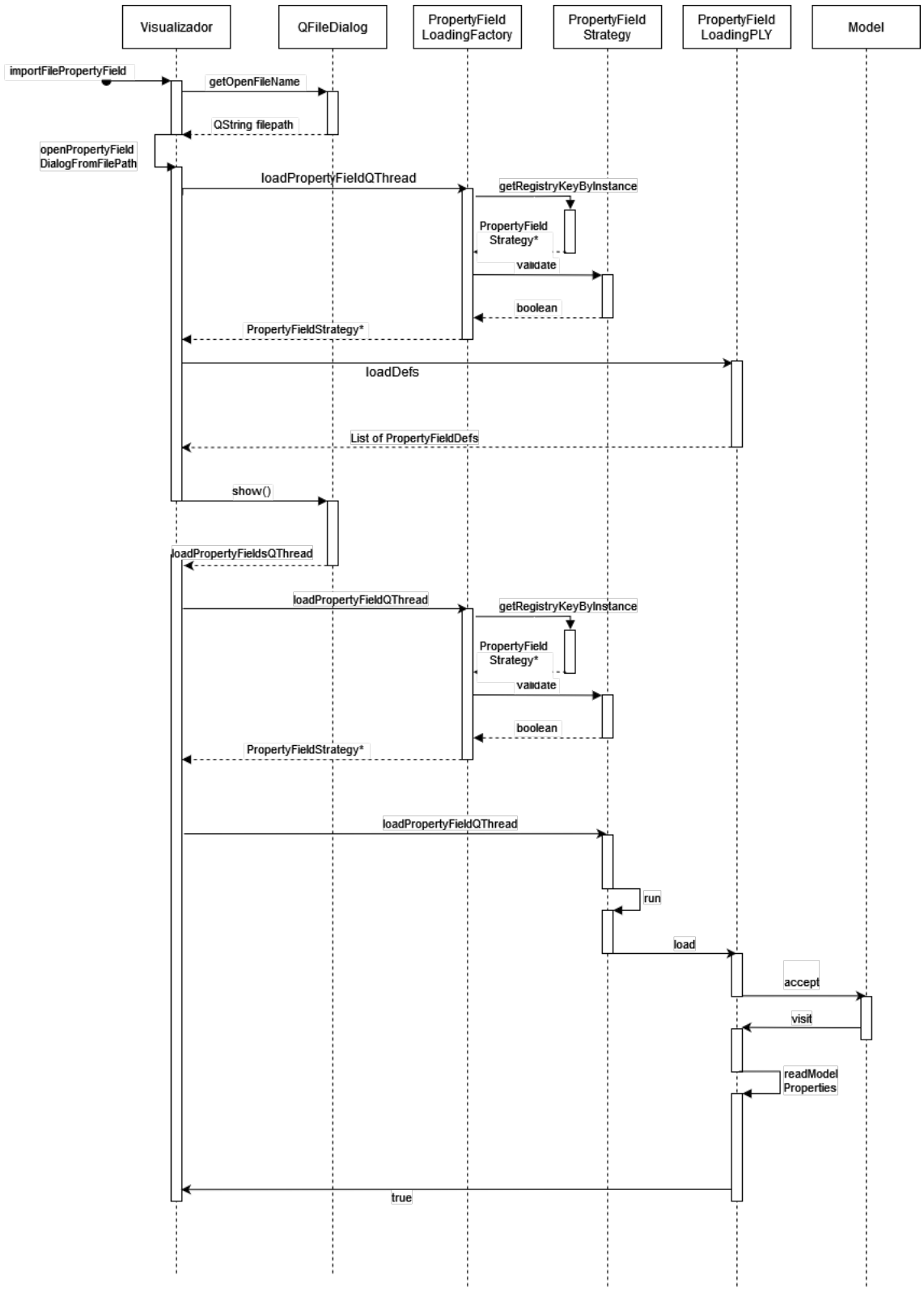


Figura 2.9: Diagrama de interacción entre Visualizador y Módulo de carga de propiedades

2.3.6.5. Estrategias de Selección

Este componente permite a un usuario escoger una región particular del modelo, lo que permite aplicar operaciones de evaluación sobre esta o su exporte posterior. *Camaron* actualmente implementa las siguientes estrategias:

1. Seleccionar área con mouse
2. Escoger elementos geométricos con identificadores específicos
3. Escoger en base a valores de propiedad de evaluación
4. Seleccionar intersección entre modelo y geometría convexa
5. Extender selección previa a elementos vecinos del mismo tipo

Cada estrategia permite escoger aplicar estas selecciones sobre polígonos o poliedros, así como poder unir, intersectar o invertir una selección previa. Esta funcionalidad puede ser accedida a través del *Widget Selection Strategies*. En el caso de la selección por mouse, el usuario debe presionar la tecla *Shift* y arrastrar el mouse sobre el área a escoger.

Estas estrategias utilizan la estructura de datos *Selection* para almacenar el contenido seleccionado por el usuario. Esta estructura utiliza un diccionario para asociar el identificador de cada componente con un puntero a este elemento. Para que *Camaron* pueda considerar un elemento como seleccionado, su campo *selected* debe ser *true* y actualizar la representación interna de *RModel*. Estas estrategias son ejecutadas de forma similar a las estrategias de evaluación, con la diferencia de que la clase *Visualizador* debe incluir un objeto de tipo *Selection* al iniciar el componente correspondiente.

2.3.7. Renderizadores

Dentro de *Camaron* se encuentran implementados un total de 13 renderizadores individuales, los cuales pueden ser accedidos a través del *Widget Renderers*.

1. Wireframe
2. Phong shading
3. Transparente (Glass)
4. Intersección de geometrías convexas
5. Normales de vértices
6. Nube de puntos
7. Ejes adicionales
8. Propiedades de evaluación

9. Elevación
10. Campos Escalares
11. Isolineas
12. Isosuperficies
13. Profundidad

Cada renderizador incluye un menú de configuraciones asociado, el cual puede ser accedido a través del botón *Config*, por otro lado se pueden combinar múltiples renderizadores utilizando el botón '+' aledaño, el cual agrega un renderizador a la lista de renderizadores secundarios. A la hora de eliminar uno de estos renderizadores, el usuario debe escoger con el mouse al renderizador y presionar la tecla *Suprimir/Delete*.

Internamente cada Renderizador es implementado como una subclase de la clase padre *Renderer*. Al mismo tiempo el menú de configuración es implementado de forma separada, incluyendo un archivo *.ui* para su generación.

2.3.8. Utilities

Dentro de *Camaron* se puede encontrar el directorio *Utils*, el cual contiene clases y métodos que son utilizados por uno o mas componentes dentro del sistema. Debido a la falta de documentación existente mas el alto número de dependencias asociado, no se pudo realizar un análisis exhaustivo sobre estos objetos, remitiéndose solo a la revisión de las herramientas que afectan a módulos estudiados en este trabajo. A continuación se listarán los componentes que serán mencionados en las secciones posteriores de este documento:

2.3.8.1. MatrixTransformationsUtils

Clase encargada de crear matrices asociadas al viewport de *Camaron*[3](Sección 4.4), utilizado principalmente por la clase *RModel* para implementar la manipulación de la cámara durante el proceso de renderizado.

2.3.8.2. FileUtils

Clase responsable de manipular archivos previo a su lectura, utilizado principalmente para almacenar el contenido completo de estos en memoria RAM como arreglo de caracteres y para operar sobre sus nombres.

2.3.8.3. Endianness

Clase estática que permite obtener el *endianness* del hardware que esta ejecutando el programa, así como la conversión de bytes entre los diferentes tipos de ordenamiento (*little*

endian o *big endian*).

2.3.8.4. CharArrayScanner

Este componente se encarga de transcribir el contenido dentro de un arreglo de caracteres al estilo C, realizando conversiones numéricas y de texto dependiendo del tipo requerido por el programador. Este componente es utilizado principalmente por las estrategias de carga para procesar el contenido de un archivo soportado, en conjunto con FileUtils y Endianness. Este lector también permite el mover la posición de este a la siguiente línea dentro del buffer, así como el mover este un número específico de bytes.

2.3.8.5. LowMemoryHashMap

Estructura de datos con una interfaz similar a un *hashmap*, implementada por el desarrollador original[3] del proyecto con objeto de reducir el consumo de memoria del sistema. Este estructura se encuentra alojada en cada elemento geométrico creado dentro del modelo, para almacenar la información computada por ambas estrategias de evaluación y estrategias de carga de propiedades. Internamente este componente define la estructura *LowMemoryHashItem*, la cual actúa como un par tipo *(key, value)* dentro del objeto.

Esta estructura internamente es implementada a través de una arreglo de estilo C, el cual posee punteros sobre cada par almacenado por esta, la clase es capaz de extender su capacidad en incrementos de tres pares a la vez, con cada par almacenado dentro de memoria *heap*.

Operaciones de inserción y acceso son realizados recorriendo el arreglo de forma secuencial, con complejidad $O(n)$. En el caso de la inserción, esta operación siempre resulta en el almacenamiento de un nuevo par, delegando la responsabilidad de prevenir repeticiones a los componentes que utilizan esta estructura.

2.3.8.6. HashingTree

Estructura de datos utilizada exclusivamente por las estrategias de carga de tipo Ele/-Node y M3D, Estos formatos solo entregan información sobre los vértices y poliedros que componen al modelos, por lo que es necesario construir las caras de forma autónoma para poder representar y dibujar el modelo.

Para evitar almacenar caras repetidas, esta estructura se encarga de llevar registro de cada secuencia de vértices leída, asociando cada secuencia única con un poliedro creado a través de un puntero. Internamente esta es implementada usando una estructura de árbol, donde cada nodo corresponde a un vértice específico, representando una secuencia a través de una cadena de nodos, la cual almacena el puntero al poliedro en el último miembro de esta.

Debido a que *Camaron* asume que una cara puede ser utilizada en dos poliedros, la

estructura optimiza su consumo de memoria eliminando un nodo cada vez que se ingrese una secuencia repetida, permitiendo reducir el tamaño del árbol durante la ejecución.

2.3.8.7. Polygon and Polyhedron Utils

Clases estáticas que incluyen rutinas para trabajar sobre los elementos geométricos respectivos, siendo su uso mas importante durante los procesos de triangulación mencionados en la sección 2.3.5

Capítulo 3

Análisis

Este capítulo corresponde al análisis realizado sobre el software. Inicia con la documentación de falencias que afectan al sistema como conjunto, para luego realizar un estudio mas a fondo sobre cada componente individual. Cada sección incluye una discusión sobre los hallazgos mas importantes realizados, entrega una lista con las falencias detectadas. Tras esto se realiza un estudio sobre el consumo de recursos de la aplicación, el cual se divide en una estimación teórica para entender el peso de cada componente en memoria principal, para luego entregar mediciones empíricas sobre modelos de prueba utilizados en trabajos previos, con el objetivo de cuantificar el rendimiento de la versión original de *Camaron* bajo condiciones de estrés.

3.1. Problemas generales

3.1.1. Optimizaciones del compilador

Una problemática importante presentada durante las etapas iniciales de indagación, consistió en la incapacidad de poder ejecutar el programa con las opciones de optimización entregadas por g++ (-O2, -O3 entre otros). Estas opciones de compilación entregan optimizaciones de performance en múltiples ámbitos, a través de modificaciones agresivas del código fuente. En el caso de *Camaron*, la compilación del proyecto puede ser completado, pero al ejecutar este el programa falla durante la inicialización de la aplicación. Este tipo de compilación no incluye símbolos de *debug*, ya que las modificaciones realizadas al código pueden dejar ciertas áreas de este drásticamente diferentes a sus contrapartes en texto. La corrección de falencias en este tipo de ambiente tiene una dificultad mayor.

3.1.2. Documentación existente

La documentación del software se encuentra principalmente disponible en los trabajos de memoria mencionados anteriormente [3]-[4]. Si bien estos documentos permiten obtener una

imagen general de cómo el proyecto se encuentra estructurado, existen incongruencias entre esta y el código fuente del software, ante la presencia de componentes internos que no se encuentran documentados.

El código fuente también carece de documentación técnica en forma de anotaciones o *comments*, dificultando la comprensión de cómo los componentes del sistema fueron implementados, ya que las decisiones relacionadas a la lógica de ciertos componentes no puede ser extraída de forma directa del código existente.

Al mismo tiempo, modificaciones posteriores al sistema generan incongruencias entre los nombres descriptivos utilizados con respecto a su implementación, siendo un caso recurrente la reutilización de interfaces en componentes con funcionalidades diferentes.

3.1.3. Estado repositorio inicial

Camaron inicialmente presentaba errores de compilación menores asociados a *#include statements* faltantes o con capitalización incorrecta (afectando la compilación en Linux). Si bien estos errores son de fácil corrección, estos se encuentran presentes en la mayoría de las revisiones registradas dentro del repositorio, haciendo más difícil el testeo de versiones previas, así como la utilización de herramientas como *git bisect* para la identificación de regresiones.

En segunda instancia, si bien el desarrollo del proyecto se encontraba bajo un sistema de control de versiones para la totalidad de la segunda iteración del proyecto [4], el desarrollo de la primera iteración no se encuentra registrada en este. La totalidad del código fuente de primera versión esta presente dentro del *commit* inicial. Si bien esta revisión incluye un *changelog* dentro de un archivo de texto (*Changes.txt*), este historial se encuentra incompleto, remitiéndose a describir a grandes rasgos ciertas modificaciones del sistema, por lo que gran parte de los componentes dentro de *Camaron* no poseen un historial de cambios detallado que permitan entender ciertas decisiones de diseño o que permitan identificar regresiones ingresadas previo a este periodo.

3.1.4. Undefined Behaviour

Se agregó como opción de compilación la posibilidad de incluir el sanitizador de undefined behavior (*ubsan*) presente en versiones recientes de g++ . Con la utilización de esta herramienta se pudo constatar la presencia de 11 instancias de este tipo de error, las cuales serán detalladas en las secciones posteriores. Esta herramienta puede ser utilizada dentro de los ejecutables disponibles a través de los siguientes comandos de consola para incluirlo dentro del proceso de compilación:

```
qmake "CONFIG += ubsan"
make
```

Las anomalías detectadas pueden ser agrupadas en dos grupos:

Utilización de variables no inicializadas

La inicialización de estas variables depende de la implementación específica de cada plataforma, lo que en general conlleva a fallas en áreas de código posterior a su acceso al intentar utilizarlas sin asignarles un valor inicial. Esto resulta en fallas asociadas al acceso de memoria fuera de rango en contenedores, o comportamiento erratico durante el renderizado de la aplicación.

División de cero flotante

Si bien el estándar IEEE 764 [9] especifica que el resultado de esta operación resulta por default en el valor *inf*, el estándar C++ 11 define esta operación como *undefined behaviour*[10](Seccion 5.6.4). Es asi como su resultado varía dependiendo del hardware en que se ejecute *Camaron*.

3.2. Análisis por componente

3.2.1. Model

La jerarquía *Model* (ver figura 2.3) presentaba falencias expuestas en el trabajp predecesor a este [4](Pag 20). Sin embargo debido a que su rol es esencial dentro de *Camaron* y su corrección implicaría modificar una gran sección del código fuente, esto no fue abordado en su tiempo. Para este trabajo se encontraba planeada la reestructuración de esta arquitectura, por lo que se estudió el efecto de este diseño sobre *Camaron*, logrando obtener los siguientes hallazgos:

1. Subclases de modelos representan conceptos ortogonales entre si, implicando la presencia de ciertos elementos geométricos y asegurando la exclusión de otros. Esto se condice con la jerarquía actual, ya que esta herencia interpreta un modelo complejo como un *tipo* de modelo simple en vez de representar composición. (Ej: En la implementación actual, una malla de poliedros *es* una nube de vértices, en vez de *contener* una instancia esta)
2. La utilización de *double dispatch* para inicializar componentes choca con la implementación de esta arquitectura. Si un modelo no contiene cierto elemento, esta clase no es capaz de articular esta situación al carecer de una interfaz que lo permita. Debido a esto cada versión del punto de entrada de estos componentes intenta operar sobre solo una sección de la interfaz, resultando en la utilización de *upcasting* de punteros para poder acceder a implementaciones alternas de cada subclase. Los componentes no son capaces de determinar el curso de acción solo con la información entregada por la subclase del modelo entregado, resultando en un flujo de operación difícil de mantener.
3. La carga de elementos de un modelo requiere el conocer de antemano el tipo de elementos a almacenar, debido a la restricción impuesta por la asimetría de interfaz entre subclases.
4. El crear nuevos componentes implica la creación de un puntos de acceso para cada tipo de modelo, la creación de una llamada de *double dispatch* para las clases *Model* y

Element

5. Cada modelo almacena un valor constante el cual representa su tipo, este es utilizado por los renderizadores de *Camaron*. Esto se debe a que la clase *RModel* solo almacena un puntero de tipo *Model* y la información asociada al tipo del modelo se pierde.

6. El número de elementos almacenados dentro de la clase y el contador asociado a cada elemento actúan de forma independiente, por lo que existe la posibilidad de causar inconsistencias entre ambos si no se tiene cuidado. (Causante de *segfaults* para modelos sin ejes adicionales)

7. Como fue mencionado en el trabajo precedente a este [4](*Trabajo Futuro*), el modelo almacena punteros a cada objeto almacenado en memoria. Esto genera una fragmentación del *heap* impidiendo que el procesador pueda aplicar optimizaciones, ya que cada elemento se encuentra en una sección no colindante de memoria.

Por otro lado se encuentra la jerarquía paralela de las variantes *Lightweight*, las cuales poseian tratamiento diferente al de la implementación principal. Si bien estas clases heredan de la clase base *Model* de forma similar a las otras clases, su implementación no permite la substitución entre variantes con el mismo contenido (Ej: VertexCloud y LWVertexCloud), violando así el principio de substitución de Liskov. Esto resulta en que cada componente que permite el uso de estas variantes, utilice clases y métodos paralelos a la implementación original para permitir su manipulación.

Estos subsistemas al estar separados de la funcionalidad principal de *Camaron*, no se encuentran al día con respecto a cambios realizados durante el desarrollo, por lo que terminan conteniendo métodos desactualizados y con errores que ya habían sido corregidos en variantes normales de *Camaron*. Como ejemplo de esto, el cargador de modelos para archivos PLY sufrió una regresión durante su desarrollo, impidiendo la carga de ciertos modelos. Sin embargo este archivo podía ser cargado usando la variante *LightWeight*, ya que la actualización no consideró a estos subsistemas, manteniendo la implementación original.

Este tipo de interfaz también dificulta el desarrollo futuro de la aplicación, ya que si se quisiera incluir una funcionalidad nueva a *Camaron*, no solo se debe implementar la estrategia objetivo, si no también se deben implementar su variante *LightWeight*, causando una duplicación de código fuente importante.

3.2.2. Carga de modelos

Este módulo en particular concentraba la mayor cantidad de falencias dentro del sistema, con situaciones que incurren en *segfaults* ante la carga de archivos con ciertas características. En estos casos es importante notar que los archivos causantes de fallas si estarían cumpliendo con casos de uso esperados por el sistema, por lo que las fallas encontradas afectaban a *Camaron* de forma significativa. (Las fallas mas graves estaban en los cargadores de formatos VisF y M3D, que se encontraban inutilízables al momento de realizar el análisis). Un resumen de la fallas detectadas se presentan en la Tabla 3.1

Dentro del código asociado a este componente se detectó un manejo inconsistente en el tratamiento de posibles fallas. En particular la carga de modelos se encuentra dentro de un bloque *try-catch* asociado a un tipo de excepción especifico a la carga de modelos, el cual nunca es lanzado por las rutinas dentro del bloque. Al mismo tiempo el flujo del proceso de carga utiliza valores booleanos para determinar si la lectura de un tipo de elemento falló durante la ejecución, lo cual se contradice con la implementación de estas rutinas ya que todas siempre retornan *true*. Por último, si la lectura del header de un archivo falla, el componente retorna un puntero nulo, que no es manejado apropiadamente por la aplicación, pues esta espera recibir una excepción congelando la interfaz de *Camaron* en el proceso.

A nivel de componente, el archivo leído es cargado en memoria principal utilizando el componente *FileUtils*, el cual a su vez utiliza *malloc* internamente para crear un buffer para el archivo. Este buffer es solo eliminado al final de la etapa procesamiento de mallas, proceso que no utiliza el buffer en ninguno de sus pasos, por lo que la memoria ocupada por el archivo reduce la cantidad de memoria disponible para almacenar componentes del modelo en si. Por otro lado, a la hora de eliminar el contenido del buffer, se utiliza el comando *delete* en vez de *free*, posibiliando una fuga de memoria, ya que al liberar memoria con el comando incorrecto, C++ no asegura la liberación efectiva de la memoria en cuestión.

En este componente también se puede observar una anomalía dentro del diseño de la jerarquía de modelos, la cual corresponde a la incapacidad de construir un modelo vació sin conocer los tipos de elementos a almacenar. Si bien en ciertos formatos es posible inferir el tipo de malla en base a la información entregada al inicio del archivo, otros formatos solo definen la existencia de ciertos elementos ya avanzada la lectura de un archivo. *Camaron* actualmente intenta resolver esta falencia creando un modelo temporal con el tipo de elemento geométrico mas complejo soportado por la estrategia. En el caso de ser un modelo mas "simple", el sistema procede a crear un nuevo modelo con el tipo apropiado, transfiriendo el contenido original a este nuevo contenedor y eliminando el modelo original.

Otro factor no menor que afecta a este componente corresponde a la gran cantidad de código copiado entre implementaciones, en particular cada implementación presente replica código de otras estrategias, incluyendo fallas y lógica que no aplica a ciertos formatos. A su vez en el caso de formatos con múltiples codificaciones, la implementación de lectura de elementos es duplicada nuevamente (Ej: loadVertices, loadVerticesBinary en PLY). Esto hace difícil la modificación de la lógica dentro de este modulo, ya que para lograrlo también se debe modificar cada implementación clonada en las demás estrategias.

3.2.2.1. Falencias específicas a cada formato

PLY

En su versión original, la estrategia para este formato realizaba tres suposiciones sobre el contenido de los archivos ingresados: (i) En formatos binarios el *endianness* sería siempre contrario al de la máquina utilizada (ii) Los campos de propiedades solo podían ser aplicados a vértices y (iii) su contenido siempre debía ser interpretado como valores de tipo punto flotante. Estas suposiciones no cumplen con la especificación del formato, produciendo que múltiples modelos descargados de forma externa terminaran

en *segfaults*.

Ele/Node

Si bien este componente no poseía fallas durante su ejecución, una anomalía presente en su implementación consistía en que su estrategia de carga utilizaba la misma interfaz que otras estrategias de carga similares pero internamente leía dos especificaciones diferentes en vez de una (utilizando las mismas extensiones). Debido a la carencia de documentación sobre este punto, no se tenía conocimiento de esta funcionalidad previo al desarrollo de este proyecto.

VisF

La causante principal de las fallas en este formato se encuentra asociada a una serie de cambios dentro de la especificación, los cuales fueron realizados de forma posterior a la definición entregada por el desarrollador original[3], no siendo documentadas. Esto implicó que los archivos de prueba disponibles no pudieran ser leídos por el sistema, resultando en *segfaults*, lo que hizo mas difícil su diagnostico inicial.

TS

La carga de modelos de este tipo se encontraba funcional originalmente, con la salvedad de que el formato en si correspondía a un tipo de archivo de prueba interno, el cual no posee una especificación asociada que permita asegurar su correcto funcionamiento. Esto en conjunto con el poco uso que se le daba a este tipo de extensión, se decidió eliminar el soporte para este tipo de formato en particular.

M3D

Como fue mencionado anteriormente, este formato se encontraba inutilizable al momento de iniciar el proyecto. Un análisis en mayor detalle reveló que la causa de la falla estaba asociada a un error dentro del proceso de tetraedrización realizado por la clase *RModel* (ver 2.3.5). En el caso de encontrar un poliedro con caras no triangulares, el algoritmo primero debía descomponer la cara en múltiples triángulos, los cuales luego eran asociados a un vértice para construir los tetraedros necesarios. En cambio, la implementación original tomaba tríos de vértices sucesivos en vez de triangularizar correctamente, por lo que caras que no fueran múltiples de tres resultaban en *segfault*, al intentar obtener tres vértices con los elementos sobrantes. Esta falla también era la causa de un error similar para modelos VisF compuestos de poliedros.

El formato tampoco poseía una especificación asociada a este. En este caso si existía motivación por preservar el soporte de esta extensión, por lo que se planifico una formalización de la especificación de este formato (Ver Anexo A).

TQS

Si bien dentro de la documentación previa se menciona la capacidad de leer archivos en este formato experimental [4](Sección 3.3.3), *Camaron* actualmente no implementa una estrategia de carga para este formato, por lo que no es posible cargar un modelo de prueba utilizando la funcionalidad existente. Este tipo de formato solo posee una estrategia para carga de propiedades dentro del código, la cual se detallará en la sección 4.2.4 de este documento.

3.2.2.2. Falencias especificas al procesamiento de mallas

En esta sección se encontraron dos fallas importantes. En primer lugar la etapa de cálculo de normales por vértice tenía un tiempo de ejecución similar a una implementación secuencial del mismo algoritmo, siendo que esta etapa había sido implementada de forma paralela. Una revisión en mas detalle detectó que esta etapa era iniciada por un la subrutina *calculateNormalsVertices*, la cual esperaba como parámetro un booleano en vez de un número como las demás etapas. Esto implicaba que si se asignaba un número de threads mayor a uno, el sistema implícitamente haría una conversión a *true*, interpretando este como el valor uno, en consecuencia actuando de forma equivalente a una implementación secuencial.

En segundo lugar, la pipeline de procesamiento incluye en su diseño original una etapa de corrección de normales. Si bien esto es una funcionalidad que podría ser valiosa como una herramienta disponible al usuario, esta implícitamente estaría modificando el contenido del archivo leído. Esto contradice el objetivo de *Camaron* de ser una herramienta de diagnóstico, ya que en el caso que un modelo presentara una falla con respecto a la orientación de sus normales, *Camaron* corregiría este error sin notificar al usuario de su presencia.

Tras conversar con la profesora guía sobre este punto, se planeo la remoción de esta etapa. Al quitar esta sección de la pipeline de procesamiento se pudo evidenciar que los formatos de tipo Ele/Node, M3D y TS estaban cargando sus componentes de manera errónea, generando vectores normales incorrectos los cuales eran corregidos por esta etapa. El error había pasado desapercibido. Una investigación en mayor detalle encontró el origen de esta falla, el cual estaba asociado a una interpretación errónea de la especificación de estos formatos afectados.

Componente afectado	Condición inicial causante de falla	Resultado falla	Causante
Formato PLY	Carga de modelos bajo codificación *little endian*	segfault	Programa asume codificación *big endian* para cualquier input
	Carga de modelo con campos de tipo uint8	segfault	Regresión removió la identificación de tipos para campos específicos
	Carga de modelo con header sin keyword "end_header"	Programa no responde	Componente lee archivo completo como header, no enviando señal de término a interfaz
	Carga de modelos con codificación *ASCII*	Lectura incompleta de polígonos	Utilización de código *ASCII* como numero en vez de su valor numérico actual.

	Carga de modelo con propiedades de poliedros	segfault	Cargador no identifica presencia de propiedad, leyendo contenido de cada cara de forma incorrecta
Formato VisF	Carga de cualquier modelo	segfault	Modificación no documentada dentro de la especificación del formato previene lectura de modelos antiguos
	Carga de modelo con codificación *big endian*	Borde de modelo inconsistente	Calculo de borde ocurre previo a la inversion de *endianness*
Formato M3D	Carga de modelos compuestos de poliedros con caras no triangulares	segfault	Regresión causada por tetraedrizacion incorrecta de poliedros complejos
	Carga de modelo de poliedros	Normales invertidas en todo los poliedros	Orden de lectura de elementos de poliedros se encuentra invertida
Formato ELE/NODE	Carga de modelo compuesto por mas de un millón de tetraedros	Aplicación no responde	Escasez de recursos causa memory thrashing
	Carga de cualquier modelo	Primer poliedro posee identificador invalido	Utilización de variable no inicializada al inicio de la carga de poliedros
	Carga de malla de poliedros	Caras con normales inconsistentes	Lectura de vértices no considera orden sugerido por especificación
Cálculo de normales para vértices	Modelo con normales invertidas	Elementos con normales NaN	Normalización de vector cero durante procesamiento
	Carga de cualquier modelo	Rendimiento similar a versión no paralelizable	Parámetro para especificar numero de threads es implícitamente convertido a boolean

Tabla 3.1: Deficiencias detectadas en carga de modelos

3.2.3. Estrategias de Evaluación

Las implementaciones específicas de este componente no poseen falencias aparentes con respecto a su lógica, sin embargo se detectó que este componente almacena la misma información en múltiples partes dentro de *Camaron* (Ver en la sección 3.3.3), por lo que el consumo de memoria podría ser optimizado.

Con respecto al diseño de este componente, como fue mencionado en la sección 2.3.6.3, el punto de entrada de cada estrategia puede ser invocado de tres formas diferentes. Estudiando con mas atención se notó que el propósito de estas llamadas correspondía a casos de uso diferentes, siendo la primera asociada a aplicar el calculo de las métricas, mientras que las otras dos corresponden al acceso de los datos previamente almacenados. Debido a esto la lógica interna del método se encuentra divida en dos áreas que son escogidas a través de parámetros de ingreso, haciendo la mantención del código mas compleja al intentar mantener ambos flujos dentro de un mismo punto de entrada.

El código presenta una complejidad alta a la hora de operar sobre un modelo, ya que su implementación intenta resolver las siguientes incógnitas:

1. El modelo recibido corresponde a una subclase que ya posee soporte? (*EvaluateElementsFrom*)
2. El modelo posee elementos que pueden ser evaluados por la estrategia? (*isFullFilled*)
3. El tipo del elemento recibido es soportado por la aplicación? (*evaluateUsingEvaluationStrategy*)

La primera pregunta es necesaria debido al diseño interno de la clase Model (ver sección 3.2.1), en cambio la segunda pregunta ya es respondida por la clase misma, considerando que estrategia solo puede operar sobre un solo subtipo de elementos geométricos, conocido durante la etapa de compilación.

La última consulta es realizada para poder determinar si un puntero de polígonos recibido es internamente un elemento de tipo *Triangle*. Esta pregunta es realizada sobre cada elemento dentro del modelo, debido a que el contenedor podría almacenar punteros a elementos de diferente tipo.

Por último, debido a que la información de cada elemento se encuentra dentro del cache interno de cada elemento geométrico, el acceso a este implica el consultar si el elemento posee la propiedad específica. Esto en conjunto con lo mencionado anteriormente conlleva a un componente con un alto nivel de complejidad ciclomática que podría ser mejorado si se realiza una reingeniería a los componentes con los que interactúa para facilitar su mantenimiento.

3.2.4. Estrategias de Selección

Con respecto a la clase *Selection*, cada objeto de clase *Element* utiliza como identificador el valor numérico provisto por el archivo leído, este identificador es único entre elementos del

mismo tipo pero no entre grupos de elementos diferentes (Ej: El id 0 puede ser utilizado por el primer vértice, polígono y poliedro dentro de un mismo modelo). Debido a que el objeto *Selection* no realiza distinciones entre subtipos de elementos, es posible causar colisiones al ingresar elementos de diferente subtipo con el mismo identificador en la estructura.

La clase *Selection* posee una referencia a *RModel* la cual es utilizada al actualizar el contenido de este componente. Esta variable debe inicializarse de forma independiente a la construcción de esta estructura, por lo que existe el riesgo de que en cambios futuros esta referencia no sea inicializada previo a su uso, lo cual podría ser causal de *undefined behaviour*.

Con respecto al diseño de las estrategias, se notó la presencia de estrategias *híbridas*, las cuales implementan la interfaz de *SelectionStrategy* y de *Renderer* para poder realizar su funcionalidad. Siendo estas *Mouse Selection* y *ConvexGeometryIntersectioRenderer*. Si bien su implementación es correcta, esta posee limitantes dentro de la arquitectura las cuales podrían ser mejoradas a través de una abstracción mas adecuada.

Con respecto a la usabilidad de las estrategias, se notó que estas selecciones son aplicadas dependiendo de la orientación de la cámara. Permitiendo selecciones no intuitivas al permitir seleccionar objetos que estén detrás de la superficie mostrada por el visor, pero que posean una orientación adecuada.

También se pudo identificar una serie de errores asociados a estrategias especificas que no pudieron ser profundizadas durante el desarrollo del proyecto, los cuales son listados en la tabla 3.2

Componente afectado	Condición inicial causante de falla	Resultado falla	Causante
Selección por Mouse	Realizar selección sobre área con agujeros	Selección inconsistente	No Identificado
	Seleccionar área	OpenGL Error	No Identificado
Expansión a vecinos	Aplicar estrategia a selección de poliedros basada en ángulo	Selección ocurre sobre elementos con ángulo mayor al escogido	Comparación implementada de forma incorrecta

Tabla 3.2: Deficiencias detectadas en estrategias de selección

3.2.5. Carga de propiedades

Como fue mencionado en la sección de antecedentes para este componente (sección 2.3.6.4. la implementación original de la estrategia TQS se encontraba inconclusa. En particular el proceso de carga de definiciones utilizaba clases aun no implementadas, de las cuales no se pudo encontrar archivos asociados dentro del historial del proyecto. En segunda instancia, el proceso de carga de estas propiedades corresponde a una versión modificada de la estrategia

para el formato Ele/Node, la cual no solo realiza la carga de propiedades, sino que además construye un objeto de tipo *Model* adicional de forma idéntica a Ele/Node. Tras la carga del modelo, el método retorna un puntero al modelo cargado, generando un error de compilación debido a que este resultado difiere de la interfaz que deben seguir las estrategia de carga de propiedades.

Dentro de los formatos funcionales, el componente posee un número bajo de fallas menores, sin embargo la interacción entre este componente y los demás sistemas podría ser mejorado para reducir el numero de llamadas necesarias. Estas deficiencias menores son resumidas en la tabla 3.3

Componente afectado	Condición inicial causante de falla	Resultado falla	Causante
Formato PLY	Sistema no emite advertencia al detectar codificación binaria	Interfaz no notifica al usuario	Regex no reconoce archivos con headers que incluyen la versión de la especificación.
Formato ELE/Node	Cargar campo de propiedades escogiendo archivo .ele	Campo no es cargado, usuario no es notificado de error	extensión .ele no es registrado en cargador durante el inicio del programa

Tabla 3.3: Deficiencias menores detectadas en componente de carga de propiedades

3.2.6. Exportación de modelos

La estrategia de exporte del formato VisF es la estrategia con mayor problemas dentro de este componente, en particular debido al número de revisiones que el formato a recibido durante el tiempo. Su implementación actual no se encuentra actualizada, causando que fuera imposible la lectura de los archivos creados. Al mismo tiempo si bien el exporte de selección se encuentra disponible, su implementación no es la correcta debido a que esta fue copiada (sin modificaciones) de la implementación encontrada en el exportador de archivos OFF.

En el caso de las otras estrategias de exporte, se detectaron inconsistencias en el orden en que son escritos los elementos geométricos que componen a los poliedros del modelo, lo cual resulta en modelos que al volver a ser leídos por la aplicación, posean caras con normales incorrectas. Estas falencias son presentadas en la tabla 3.4

Componente afectado	Condición inicial causante de falla	Resultado falla	Causante
Formato VisF	Cargar modelo de nube de puntos, recientemente exportado con este componente	Modelo no es renderizado	Archivo es escrito en base a especificación deprecada

	Exporte de selección sobre modelo	Creación de archivo con formato OFF	Implementación copiada de Exportador para formato OFF sin modificaciones
Formato Ele/Node	Exporte de mallas de poliedros	Poliedros escritos poseen orientación incorrecta	No Identificado
Formato OFF	Exporte de modelo cualquiera	Elementos escritos poseen orientación incorrecta	No Identificado
Formato TS	Exportar archivo a este formato	Formato no es reconocido	No Identificado

Tabla 3.4: Deficiencias detectadas en exporte de modelos

3.2.7. Interfaz gráfica

La tabla 3.5 resume las falencias detectadas al realizar pruebas manuales sobre la interfaz de la aplicación. Estas fallas no resultan en la interrupción de la aplicación, correspondiendo principalmente a anomalías gráficas dentro de la pipeline de OpenGL. Debido a su bajo riesgo mas al tiempo acotado del proyecto, solo se remitió a documentar a grandes rasgos su identificación.

Componente afectado	Condición inicial causante de falla	Resultado falla	Causante
Progreso de Carga	Carga de cualquier modelo ingresado	Porcentaje de carga NaN % durante inicio	División por cero flotante durante inicio de carga.
	Cancelar carga a través de botón de cerrado	Carga no es interrumpida	Funcionalidad no Implementada
Opciones de OpenGL	Habilitar *Perspective Projection*	Modelo solo es visible tras realizar zoom por varios segundos, renderizado resulta en artefactos visuales	No Identificado
	Habilitar y Desactivar *Face Culling CCW*	*Face Culling* sigue activo	No Identificado

Visualizar Identificadores	Renderizado en GPU	OpenGL Error: Uniform location not found	No identificado
	Renderizado en GPU con transparencia habilitada	Identificadores reemplazados con rectángulos negros	No Identificado
	Renderizado en GPU con transparencia deshabilitada	Identificadores detrás de la superficie del modelo son renderizados	No Identificado
	Renderizado en CPU	Identificadores renderizados de forma inconsistente dependiendo de posición de la cámara	No Identificado
	Aplicar sobre malla de poliedro con caras no triangulares	Identificadores repetidos dentro de misma cara	IDs de poliedros son obtenidas en base a relaciones polígono-poliedro, los cuales no incluyen información referente a la triangularizacion del modelo
Histograma de evaluación	Funcionalidad no puede ser utilizada sobre mallas de poliedros	Histograma no es generado	*Setup* de evaluaciones no incluye caso de uso para mallas de poliedros

Tabla 3.5: Errores a nivel de interfaz

3.2.8. Renderizadores

Los renderizadores listados en la Sección 2.3.7 fueron revisados de forma manual, utilizando modelos de prueba disponibles para identificar anomalías visuales y verificando que las opciones presentes en sus configuraciones fueran las correctas. En general los renderizadores mas importantes no presentaban fallas significativas (*Wireframe*, *Phong Shading*, *Glass*, *Normals*, etc), siendo estas concentradas en renderizadores de menor importancia. En particular dentro de este modulo se pudieron identificar dos fallas fatales importantes. Un resumen de las fallas se encuentra en la tabla 3.6

En primera instancia el visor *Ejes adicionales* siempre se encuentra disponible, sin importar el contenido del modelo. En el caso de la lectura de un modelo sin estos ejes, el proceso de carga no inicializa el contador de ejes dentro del modelo, por lo que el activar este renderizador resulta en la utilización de un valor indeterminado como índice de acceso.

El renderizador de intersección permite definir el objeto utilizado para intersectar al

modelo a través del menú *Config->Geometry*. La descripción de este objeto posee múltiples opciones, incluyendo la capacidad de cargar un archivo para cada opción. En el caso de cargar un archivo con errores la aplicación resulta en un error de tipo *segfault*. Este renderizador también posee problemas asociados a su usabilidad. En particular para definir un objeto a través de la interfaz se debe utilizar una sintaxis propia la cual es difícil de especificar (Ver 3.1). Si bien parte de esta sintaxis puede ser obtenida colocando el mouse sobre la opción escogida, esta carece de detalles suficientes para que un usuario pueda utilizarlo de forma correcta. Dentro de la documentación previa se define esta sintaxis con un poco mas de detalle [3]pag 100, pero esta es inconsistente con la sintaxis entregada por la interfaz por lo que su actualización posterior es recomendada.

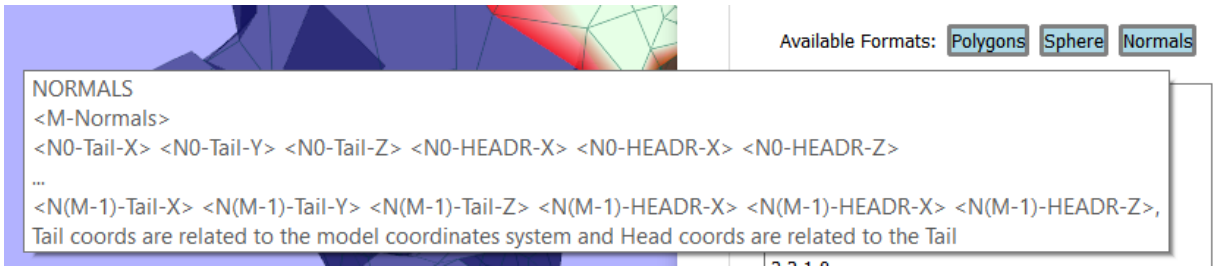


Figura 3.1: Ejemplo de sintaxis provista por renderizador

El renderizador *Visor de Profundidad* (ver figura 3.2 presenta una configuración básica, la cual al ser modificada no genera cambios dentro del modelo. La interacción con el modelo tampoco genera resultados. Esto en conjunto con la falta de documentación impide poder determinar el comportamiento esperado de este renderizador.

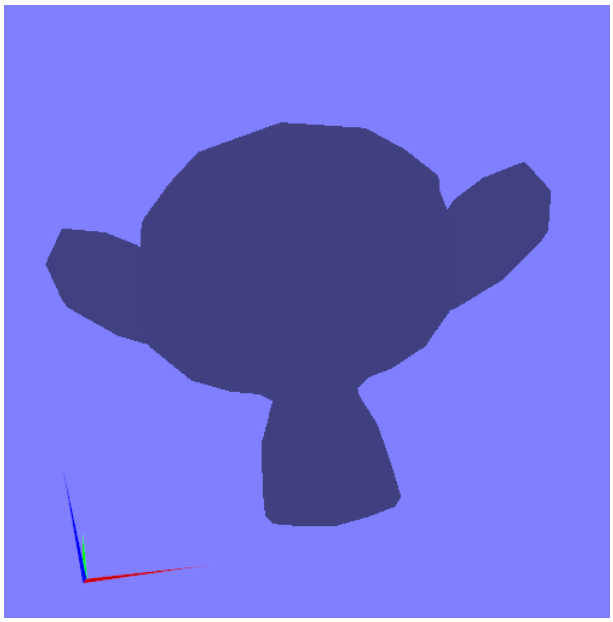


Figura 3.2: Visor de Profundidad, resultado presente es idéntico para todos los modelos probados

Por último a nivel de interfaz, se detectó la presencia de opciones asociadas a la interpolación de color (ver fig 3.3), deshabilitadas en cada una de estas instancias, para todos los

modelos testeados. Si bien la presencia de esta opción en múltiples visores puede deberse a reutilización de código, la falta de documentación no permite entregar respuesta clara sobre su caso de uso específico.

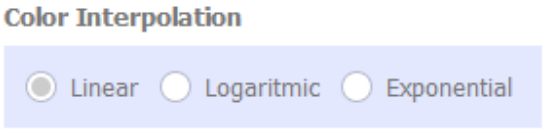


Figura 3.3: Menú de interpolación de color

Componente afectado	Condición inicial causante de falla	Resultado falla	Causante
Visor de Ejes Adicionales	Escoger renderizador en modelos sin este tipo de elemento	segfault	Conteo de ejes no es inicializado si modelo no tiene ejes adicionales, causando que programa acceda a valor indeterminado
Renderizador de Profundidad	Manipular configuración	Sin cambios en modelo	No Identificado
Visor de Intersección de objetos	Cargar segundo modelo	segfault	Lectura de geometría desconocida retorna puntero nulo como parser de geometria, el cual resulta en la dereferenciación de puntero nulo
	No Identificado	Instrucción Ilegal	Asignación de booleano a variable no inicializada
	Aplicar Intersección con geometría de tipo *Normals*	Instrucción Ilegal	No Identificado
	Interseccion de plano perpendicular sobre malla solida	Intersección inconsistente al incluir elementos no intersectados	No Identificado
Visor de Propiedades	Selección de opción *Maximum Dihedral Angle* (Inconsistente)	Modelo Desaparece	No Identificado
	Habilitar Face Culling (CCW) previo a renderizado	Renderizado incorrecto de geometrias	No Identificado

Phong por pixel	Habilitar Traslucent Mode	Triangulos con coloración incorrecta dependiendo de la posición de la camara	No Identificado

Tabla 3.6: Errores presentes dentro de renderizadores

3.2.9. Utilidades

3.2.9.1. MatrixTransformationsUtils

Durante el inicio del análisis, se constató que tras la carga de archivos, la interfaz no estaba renderizando el modelo cargado, mostrando una ventana vacía en lugar del objeto 3D. Investigando la causa del error se determinó que las matrices construidas por este componente eran inicializadas con valores basura, los cuales no eran reemplazados en su totalidad, debido a que el contenido de estos valores no se encuentra especificado, esto resulta en que la visualización solo es posible en algunos ambientes de desarrollo.

3.2.9.2. LowMemoryHash

La estructura posee múltiples errores de diseño que en conjunto socavan el propósito inicial de esta clase. En primera instancia la estructura no asegura la unicidad de las llaves, siendo posible la inserción de múltiples items utilizando el mismo identificador. Internamente el acceso a el valor de la propiedad es realizado a través de este identificador, retornando la primera ocurrencia de este, por lo que pares agregados posteriormente no pueden ser accedidos en caso de no verificar previamente su presencia.

Por otro lado, durante su construcción la estructura pide espacio dentro del *heap* para tres pares. Si ahora consideramos que los modelos son inicialmente ingresados al sistema sin evaluaciones ni campos escalares, el sistema acaba ocupando espacio para tres potenciales propiedades en cada elemento del modelo por default.

La clase fue implementada con un destructor virtual, esto resulta en la creación de un puntero a la *virtual table* de esta, el cual agrega un costo adicional de 8 bytes por elemento. Debido a estas falencias la clase acaba consumiendo el doble de los recursos reportados en su documentación, agregando un gasto importante en el caso de cargar modelos de gran tamaño. Un resumen de estos errores se encuentra en la tabla 3.7

Componente afectado	Condición inicial causante de falla	Resultado falla	Causante

MatrixTransformationsUtils	Carga de cualquier modelo ingresado	Modelos no son renderizados en pantalla	Matrices utilizadas no son inicializadas apropiadamente, resultando en matrices con valores inválidos
PolygonUtils	Carga de modelo con vértices que tengan solo una cara adyacente	Normales de estos vértices no son renderizadas	Algoritmo asigna flag para ignorar estos vértices de forma incorrecta
FileUtils	Lectura de archivos	Instrucción Ilegal	Recursos asignados con *malloc* son eliminados con *delete* posteriormente
LowMemory Hash	Ingresar par con llave existente	Estructura almacena pares repetidos	Ingreso de información no verifica si par existe
	Cargar campo de propiedades de forma reiterada	Pares ingresados con llaves incorrectas	Overflow de posibles valores de llaves sin verificación previa

Tabla 3.7: Deficiencias detectadas en clases auxiliares

3.3. Consumo esperado de memoria RAM

El consumo de memoria fue medido en función de los recursos utilizados por la clase *Model* y de sus elementos asociados. En particular es importante destacar que los valores entregados por la función *sizeof* solo incluyen el tamaño en memoria de los miembros internos de una clase, por lo que si un objeto almacena información fuera de la clase, esta información no es incluida en el resultado final entregado.

En base a lo anterior se vio la necesidad de actualizar la documentación previa sobre este tópico, ya que esta dependía de *sizeof*. A continuación se presenta un análisis en mayor detalle del consumo de memoria RAM de cada componente, con el objetivo de medir el impacto de cada uno dentro del sistema. Cada estructura listada presenta un costo inicial, el cual puede aumentar en casos de que se inserte información a estos, este tipo de coste adicional usualmente no es considerado dentro del conteo ya que la mayoría de estas estructuras son creadas sin un contenido inicial. Este tipo de costo sera representado en los conteos con su valor rodeado de paréntesis cuadrados.

Notar que este análisis solo entrega una aproximación al consumo efectivo del sistema, ya que el valor exacto también depende de sistemas que escapan al alcance de este proyecto (Sistema Operativo, Tarjeta Gráfica, Arquitectura CPU).

3.3.1. Primitivas

El lenguaje C++ no incluye dentro de su especificación el tamaño efectivo en bytes de cada primitiva o puntero del lenguaje, ya que este depende de la arquitectura en el cual el programa será compilado. Debido a esto se listará a continuación el valor de cada uno dentro de la arquitectura utilizada en el desarrollo de este proyecto como referencia.

En particular la arquitectura 64 bits duplica el consumo de memoria con respecto a punteros con respecto a su contraparte de 32 bits. Al mismo tiempo variantes de tipo *unsigned* poseen el mismo tamaño en memoria que sus variantes comunes.

- bool, char: 1 byte
- short: 2 bytes
- int, long, float: 4 bytes
- double: 8 bytes
- raw pointer: 8 bytes
- shared pointer: 16 bytes

3.3.2. Estructuras de datos

glm::vec3 12 bytes
: Estructura presente en la librería glm, corresponde a un grupo de tres floats empaquetados de forma contigua. Esta estructura se utiliza principalmente dentro de las clases *Element*, para representar posición, normales, etc.

std::vector 24 + Tamaño contenido*Numero elementos
: Esta estructura almacena 3 punteros internamente. Estos son utilizados para poder manipular el contenido del vector, el cual se encuentra en una sección de memoria contigua aparte dentro del heap.

  Cabe notar que si el vector requiere cambiar de tamaño, la capacidad final de este corresponde a un detalle de implementación que no aplica al alcance de este trabajo.

std::map 48 bytes mínimo
: La implementación interna de este tipo de contenedor es mucho mas compleja, por lo que el valor estimado solo considera el costo inicial mínimo entregado por *sizeof*.

  Independiente del posible coste tras su utilización, el costo inicial de esta estructura ya es bastante prohibitiva, por lo que su uso es muy limitado dentro de *Camaron*. Este costo fue la motivación principal del desarrollador inicial para construir una versión de bajo costo que pudiera ser agregada a cada elemento del modelo (*LowMemoryHash*).

LowMemoryHashItem 8 bytes
: Para el caso de *Camaron*, las propiedades se encontraban almacenadas en forma de (unsigned char, integer), lo cual tenía un coste de 8 bytes (1 byte de char + 3 bytes de padding + 4 bytes de integer)

LowMemoryHash 48 bytes mínimo

Durante su construcción se almacenan tres objetos de tipo *LowMemoryHashItem* en el heap de forma separada, creciendo en múltiplos de tres a medida que se realizan mas evaluaciones o se leen mas campos escalares.

- virtual table pointer: 8 bytes
- integer numItemes: 4 bytes + 4 bytes de padding
- puntero a contenido items: 8 bytes
- contenido inicial: 24 bytes

Debido a que las llaves utilizadas en Camaron poseen el tipo *unsigned char*, teóricamente la estructura tendría capacidad de almacenar un máximo de 255 pares individuales. Al mismo tiempo las fallas mencionadas en la sección 2.3.8.5 resultan en un contenedor sin un límite máximo establecido. En base a esto el consumo puede ser calculado utilizando la siguiente forma:

$$48 + (\left(8 * \left\lceil \frac{n_p}{3} \right\rceil\right) * 3) \text{ bytes} \qquad n_p := \text{Numero de items almacenados}$$

3.3.3. Elementos Geométricos

Element 88 bytes

Debido a que esta clase es la base de la jerarquía de los elementos implementados, su coste es incluido en el cálculo de cada subtipo posterior.

- virtual table pointer: 8 bytes
- int id: 4 bytes
- boolean selected: 1 byte + 3 bytes de padding
- std::vector rmodelPosition: 24 bytes [+ 4 bytes por item]
- LowMemoryHash properties: 48 bytes [+ 8 bytes por item]

El vector *RmodelPositions* es llenado durante el proceso de carga de datos a la GPU realizado por *RModel*, con el objetivo de permitir que en el caso de modificar un elemento de forma individual, este cambio se refleje en todas sus instancias dentro de la GPU.

El *LowMemoryHash* properties es llenado al aplicar una evaluación o al cargar un campo escalar. Esta información se mantiene almacenada hasta que el modelo es cerrado o la aplicación es cerrada. Si bien la cantidad de evaluaciones es limitado (15 actualmente), el número de campos escalares no tiene un limite máximo, por lo que es posible llenar la estructura cargando el mismo campo escalar de forma repetida en el peor caso.

El campo *selected* es modificado en contextos donde se tiene acceso al objeto *Selection*, por lo que su presencia en esta clase es redundante con respecto a su utilización.

Edge 128 bytes

El uso de este tipo de elemento es bastante limitado, por lo que su efecto en memoria es bastante bajo con respecto a los demás elementos.

- Coste Element: 88 bytes
- raw pointer vertex0 8 bytes
- raw pointer vertex1 8 bytes
- glm::vec3 12 bytes + 4 bytes de padding

Vertex 144 bytes

- Coste Element: 88 bytes
- glm::vec3 coords: 12 bytes
- glm::vec3 normal: 12 bytes
- int position: 4 bytes + 4 bytes de padding
- std::vector polygons: 24 bytes [+ 8 bytes por item]

Debido a que en los modelos Ele/Node, los vértices pueden empezar con 0 o 1, el programa almacena la posición efectiva del vértice en el contenedor, mas su número identificador otorgado por el archivo.

El vector *polygons* es llenado durante la etapa de procesamiento de mallas, guardando un puntero a cada polígono contiguo a este. Este tipo de relaciones tiene un costo alto en memoria ya que no poseen una cota máxima al numero de vecinos posibles.

Polygon 176 bytes

- Coste Element: 88 bytes
- float area: 4 bytes + 4 padding
- std::vector vertices: 24 bytes [+ 8 bytes por item]
- std::vector neighborPolygons: 24 bytes [+ 8 bytes por item]
- glm::vec3 normal: 12 bytes + 4 bytes de padding
- raw array polyhedrons: 16 bytes (2 punteros)

Los contenedores *vertices, neighbor polygons y polyhedrons* corresponden a las relaciones de vecindad asociadas a este elemento, las cuales son llenadas con punteros durante la fase de procesamiento de mallas. Los tamaños máximos de cada contenedor se listan a continuación:

- vertices: Máximo igual al número de aristas del polígono
- poligonos: En el caso de una malla de polígonos la cota es igual al número de aristas. En el caso de una malla de poliedros, no hay una cota superior ya que es posible conectar un numero infinito de poliedros a cada arista.
- poliedro: Para nuestro sistema, se asume que un polígono solo puede ser incluido dentro de dos poliedros como máximo.

El área del polígono inicialmente posee un valor default, el cual es reemplazado a la hora de aplicar la evaluación *PolygonArea*. Debido a esto el valor del área de un polígono se encuentra almacenado 3 veces dentro del sistema (Cache interno evaluación, cache en *LowMemoryHash* y valor de este campo).

Triangle 192 bytes

- Coste Polygon: 176 bytes
- float lado menor: 4 bytes
- float lado medio: 4 bytes
- float lado mayor: 4 bytes + 4 bytes de padding

Esta clase es utilizada para facilitar el cálculo de ciertas evaluaciones de propiedades. El largo de cada arista es inicializado durante el primer acceso a esta información.

Polyhedron 136 bytes

- Coste Element: 88 bytes
- float area: 4 bytes
- float volumen: 4 bytes
- std::vector polygons: 24 bytes [+ 8 bytes por ítem]
- glm::vec3 geocenter: 12 bytes + 4 bytes de padding

El área y volumen actúan de forma similar al campo área de *Polygon*, siendo inicializados tras el calculo de una mertrica (*PolyhedronArea* y *PolyhedronVolume* respectivamente) y también correspondiendo a campos redundantes dentro del objeto.

El vector *polygons* almacena referencias a cada cara del poliedro, por lo que no poseen una cota máxima de tamaño.

El campo *geocenter* es inicializado durante la fase de procesamiento de mallas, actuando como un valor cacheado para ser utilizado por otros componentes del sistema.

3.3.3.1. Variantes Lightweight

Debido a que esta implementación corresponde a una funcionalidad experimental de Camaron, la cual no cumple con los casos de uso de la aplicación, la glosa presentada a continuación solo corresponde a información referencial.

LWElement 16 bytes

- virtual table pointer: 8 bytes
- int id: 4 bytes + 4 bytes de padding

LWVertex 48 bytes

- Coste Element: 16 bytes
- bool usedby: 1 byte + 7 bytes de padding
- glm::vec3 coords: 12 bytes
- glm::vec3 normal: 12 bytes

LWPolygon 56 bytes

- Coste Element: 16 bytes
- std::vector vertices: 24 bytes [+ 8 bytes por item]
- raw array polyhedrons: 16 bytes

LWPolyhedrons 40 bytes

- Coste Element: 16 bytes
- std::vector polygons: 24 bytes [+ 8 bytes por item]

3.3.4. Modelo

Debido a la jerarquía especial de este grupo de clases, el tamaño de cada uno es agregado al subtipo siguiente. Como el sistema solo almacena una instancia del modelo a la vez, el costo inicial de sus miembros es despreciable comparado con el coste tras la carga del modelo.

Al mismo tiempo, si bien estas clases almacenan referencias a los elementos creados, al ser estos construidos en el heap, es necesario agregar el coste del elemento mas el coste de la referencia por cada ítem almacenado en el modelo. También es importante considerar que existe un overhead adicional causado por fragmentación de la memoria utilizada, dado que estos elementos no son almacenados en memoria contigua.

Model 168 bytes

- virtual table pointer: 8 bytes
- std::vector bounds: 24 bytes iniciales + 24 bytes de contenido (6 floats)
- int modelType: 4 bytes + 4 bytes de padding
- std::vector propertyFieldDefs: 24 bytes [+ 16 bytes por campo cargado]
- std::map propertyFieldPosition: 48 bytes mínimo
- std::string filename: 32 bytes (Depende de tamaño string e implementacion)

VertexCloud 232 bytes

- Coste Model: 168 bytes
- boolean _2D: 1 byte + 7 bytes de padding
- std::vector vertices: 24 bytes [+ 152 bytes por vértice]

- std::vector edges: 24 bytes [+ 132 bytes por arista]
- int verticesCount: 4 bytes
- int additionalEdgesCount: 4 bytes

PolygonMesh 264 bytes

- Coste VertexCloud: 232 bytes
- std::vector polygons: 24 bytes [+ 184 bytes por polígono o 200 bytes por triangulo]
- int polygonCount: 4 bytes + 4 padding

PolyhedronMesh 296 bytes

- Coste PolygonMesh: 264 bytes
- std::vector polyhedrons: 24 bytes [+ 140 bytes por poliedro]
- int polyhedronCount: 4 bytes + 4 padding

3.3.5. RModel

Esta clase realiza dos tipos de pedidos de memoria a considerar, los cuales dependerán principalmente del contenido existente en ambas representaciones del modelo. Debido a que el contenido de estos depende del tamaño de cada contenedor, se utilizara la siguiente simbología para representar estas cantidades:

$\mathbb{V}$:= Número total de vértices
$\mathbb{E}$:= Número total de ejes adicionales
$\mathbb{P}$:= Número total de polígonos
$\mathbb{H}$:= Número total de poliedros
$\mathbb{A}$:= Número de atributos almacenados tras descomposición

Heap

Si bien *RModel* posee varios miembros, esta clase es única durante el desarrollo del sistema, por lo que su coste inicial es despreciable, exceptuando el vector de atributos asociado al modelo, con su tamaño dependiendo del tipo de descomposición escogido (sección 2.3.5.1).

VertexCloud

No es necesario triangularizar el contenido, por lo que su tamaño en memoria corresponde a cuatro veces el numero de vértices en el modelo.

$$\mathbb{A} = \mathbb{V} * 4$$

PolygonMesh y PolyhedronMesh

RModel realiza una triangularizacion de tipo fan (ver Figura 2.4) sobre cada polígono disponible, por lo que su tamaño máximo puede ser calculado en base a la siguiente formula:

$$\mathbb{A} = \left(\sum_{i=0}^{\mathbb{P}} (v_i - 2) * 3 \right) * 4 \text{ bytes} \qquad v_i := \text{Número de vértices del polígono i}$$

VRAM

En el caso de la GPU, Rmodel almacenara información de utilidad para los renderizadores a través de buffers de Opengl, los cuales utilizan la información obtenida tras el proceso de triangulación respectivo.

VertexCloud $16 * \mathbb{V} + 24 * \mathbb{E}$ bytes

Al no realizarse una triangularización, el arreglo de atributos posee el mismo tamaño que el número de vértices en el modelo.

- Copia de atributos almacenados en memoria: (4*$\mathbb{V}$) bytes
- Posición de cada vértice: (12*$\mathbb{V}$) bytes
- Coordenadas de vértices asociados a cada arista adicional: (24*$\mathbb{E}$) bytes
- Color de cada vértice asociado a cada arista adicional: (24*$\mathbb{E}$) bytes

PolygonMesh $\gamma + (48 * \mathbb{E}) + (48 * \mathbb{A})$ bytes

En este caso, se agrega la información del modelo previo, con la salvedad que este se encuentra asociado al arreglo de vértices triangularizado. Actualmente existen tres buffers asociados a la relación poliedro-polígono, no documentados, por lo que serán nombrados como PPH.

- Copia de atributos almacenados en memoria (γ)
- Buffers de vértices adicionales (48 * $\mathbb{A}$)
- Posición de cada vértice: (12 * $\mathbb{A}$)
- Normales de cada vértice: (12 * $\mathbb{A}$)
- Identificador de cada vértice: (12 * $\mathbb{A}$)
- PPH_1 (4 * $\mathbb{A}$)
- PPH_2 (4 * $\mathbb{A}$)
- PPH_3 (4 * $\mathbb{A}$)

PolyhedronMesh Este tipo de modelo utiliza la misma cantidad de buffers que la malla de polígonos, agregado un buffer extra para almacenar el resultado de la tetraedralización de poliedros, con un coste equivalente a 4 veces el número de tetraedros generados tras la descomposición.

3.3.6. Estrategias de evaluación

Cada estrategia de evaluación posee un cache de valores ordenados, el cual es utilizado para la confección de histogramas bajo el rango de valores presentes dentro del modelo cargado. Debido a su ordenamiento, este buffer no actúa como reemplazo a la información presente en cada elemento.

El costo de cada cache corresponde a cuatro veces el numero de elementos a evaluar en bytes. Este cache utiliza el idioma *shrink-to-fit* para destruir el cache en caso de que el modelo sea eliminado por lo que no se estaría incurriendo en una fuga de memoria.

3.4. Consumo efectivo de la aplicación

En paralelo al análisis teórico, se procedió a realizar mediciones de consumo para el proceso de carga de modelos, documentando el coste promedio de cada etapa. Los modelos de prueba fueron otorgados por la profesora guía, los cuales fueron utilizados para realizar mediciones en los trabajos previos [3]. De este grupo se escogieron los modelos de mayor tamaño, para así poder testear la aplicación bajo condiciones de estrés (Ver detalle contenido en tabla 3.8. Se realizaron cinco mediciones de memoria utilizada y tiempo de ejecución para cada modelo, obteniéndose los valores promedio de cada etapa, los cuales son mostrados en las tablas 3.9 y 3.10.

Modelo	Formato	Numero de Vértices	Numero de Polígonos	Numero de Poliedros
xyz_figurine	PLY	4.999.996	9.999.764	0
Esfera 8	Ele/Node	524.288	7.085.413	3.542.559
Esfera 9	Ele/Node	1.048.576	14.176.211	7.087.920
Esfera 10	Ele/Node	2.097.152	28.365.692	14.182.610
lucy	PLY	14.027.872	28.055.742	0

Tabla 3.8: Número de elementos geométricos contenidos en modelos de prueba

Modelo	Carga [GB]	Procesamiento [GB]	RModel [GB]	Estable [GB]
xyz_figurine	3,81	4,72	7,15	6,96
Esfera 8	3,49	4,55	6,28	6,3
Esfera 9	6,62	8,9	12,56	12,53
Esfera 10	12,06	Sin Datos	Sin Datos	Sin Datos
lucy	Sin Datos	Sin Datos	Sin Datos	Sin Datos

Tabla 3.9: Alocación promedio de memoria RAM en GB, durante cada etapa del proceso, mas el consumo final tras el proceso (Estable). Previo a la carga la aplicación consume en promedio 174MB, mientras que tras liberar la memoria utilizada del modelo el programa consume 328MB

Estas mediciones son obtenidas al observar el crecimiento de memoria tipo *Commit*, obtenidas del monitor de recursos entregados por Microsoft. En el caso de los últimos dos modelos,

la memoria necesaria supera la capacidad de almacenamiento disponible, por lo que el sistema operativo procede a almacenar parte de la memoria RAM en disco, causando *memory thrashing* en el proceso, lo cual reduce el performance de la aplicación de forma significativa.

Modelo	Carga [seg]	Procesamiento [seg]	RModel [seg]	Total [seg]	Eliminación [seg]
xyz_fig	13	22	74	109	28
Esfera 8	43	77	44	164	15
Esfera 9	117	1380	1452	2949	1200
Esfera 10	1800	Sin Datos	Sin Datos	Sin Datos	Sin Datos
lucy	Sin Datos	Sin Datos	Sin Datos	Sin Datos	Sin Datos

Tabla 3.10: Tiempo de ejecución de cada etapa en segundos, también se incluye el tiempo que toma el proceso de eliminación del modelo.

En el caso de los modelos *Esfera 10* y *lucy*, el programa no es capaz de terminar el proceso de carga, resultando en uno de los siguientes escenarios:

1. *Camaron* recibe la excepción *std::bad_ alloc* lo cual interrumpe el proceso , esto permite al usuario continuar utilizando el programa sin resultar en un error fatal. Esto generalmente ocurre si el sistema se queda sin recursos durante la primera fase del proceso.
2. La aplicación cierra de forma anticipada sin un mensaje en consola
3. El sistema operativo deja de responder, en el peor de los casos resultando en un error fatal de sistema ante la falta de recursos.

Tal como se muestra en la tabla 3.9, en el modelo *Esfera 9* se puede notar el punto en el que el sistema queda sin recursos durante el procesamiento de la malla. Estos tests fueron realizados con *Camaron* siendo el único programa abierto. Al mismo tiempo, los tiempos mostrados durante condiciones de *thrashing* variarán dependiendo de cómo el sistema operativo de cada sistema prioriza la utilización de los recursos disponibles.

3.5. Curso de acción

En función de lo observado en este documento, se consideró iniciar el proceso de refactorizacion con la estabilización del módulo de carga de modelos. Esto debido al alto numero de errores fatales dentro de este componente, las cuales inutilizaban secciones importantes del código.

A continuació se definió trabajar sobre la arquitectura de la clase *Model*, ya que las anomalías detectadas en este análisis limitaban la capacidad de poder realizar cambios estructurales a *Camaron*. Para esto se procedió a desarrollar tests unitarios para cada componente que interactuara con esta clase, para así asegurar que los cambios no causaran regresiones importantes.

Por último se consideró realizar modificaciones a estos componentes afectados tras la modificación de la arquitectura dentro del tiempo restante del proyecto. Debido a que la segunda fase del proyecto consideró más tiempo del prosupuestado, se decidió reducir el alcance del proyecto y no incluir dentro de este a las clases asociadas a Renderizadores y a la Interfaz gráfica.

Capítulo 4

Reingeniería de Camaron

Durante el trabajo realizado se aplicaron cambios y correcciones sobre una parte de las falencias detectadas en el capítulo de análisis. Estos cambios se pueden clasificar en las siguientes categorías:

1. Cambios estructurales: corresponden a cambios que afectan a múltiples componentes dentro de *Camaron*, como cambios de arquitectura y de interacciones entre módulos.
2. Modificaciones sobre componentes individuales: estos incluyen correcciones de errores y cambios en la interfaz de estos, los cuales se ven influenciados por los cambios estructurales de la sección anterior.
3. Cambios generales menores: corresponden a modificaciones asociadas a la mantención del sistema que son locales a una clase o método dentro de *Camaron*.

Tras esto se agrega una sección la cual describe el ambiente de testing creado durante el proceso de refactoring, incluyendo un resumen de los componentes cubiertos por este, clases auxiliares y especificaciones para poder ejecutar estos tests.

4.1. Cambios Estructurales

4.1.1. Reemplazo de jerarquía de modelos y eliminación de subclases

Inicialmente se propuso reemplazar la jerarquía en cadena por una basada en el concepto de composición, con el objeto de separar cada tipo de modelo como un ente separado que heredara directamente de la clase *Model*, utilizando instancias de los tipos anteriores como forma de representar el contenido de estos (Ver figura 4.1).

Esta modificación implicaría una separación en la interfaz para permitir acceso al modelo almacenado internamente de forma directa o agregando setters/getters para cada componen-

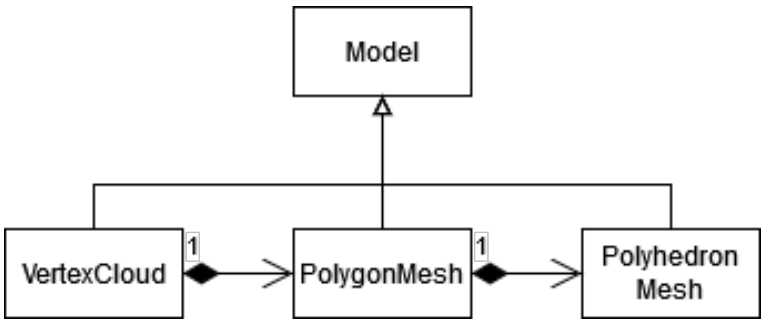


Figura 4.1: Primera propuesta de arquitectura

te, los cuales utilizaran este sub-modelo de forma privada. Esta restricción también implica reemplazar el flujo a través de *upcasting* de punteros, ya que la interfaz de acceso solo permitiría acceder a un componente específico, reduciendo la legibilidad del proyecto en el proceso.

Por otro lado se notó que de por si, cada tipo de modelo internamente contiene un conjunto de elementos geométricos, actuando como una clase contenedora sin una lógica interna adicional. En base a esto se consideró como segundo candidato una representación la cual separara la representación del contenido versus el concepto de modelo en si, utilizando una interfaz única para el acceso de cada contenedor mientras que Model se remitiría a almacenar una serie de contenedores para representar cada tipo (Ver figura 4.2).

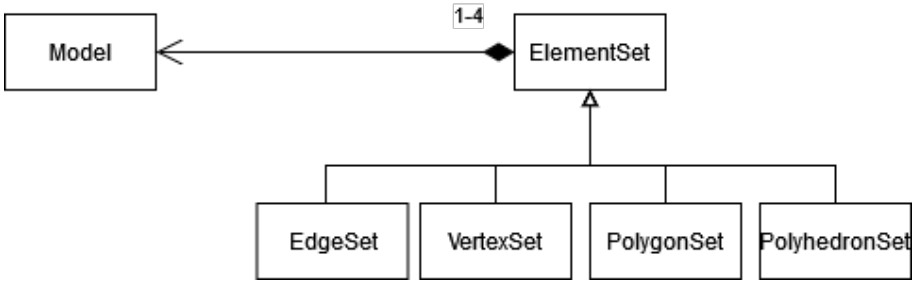


Figura 4.2: Segunda propuesta de arquitectura

En teoría una implementación de esta índole permitiría una representación mas flexible de una malla mixta, posibilitando el construir modelos con la cantidad justa de elementos, los cuales pudieran ser accedidos por una interfaz única. Lamentablemente las restricciones impuestas por C++ sobre la utilización de tipos hacia improbable un diseño de este tipo. En particular el representar un clase *Model* que tuviera una cantidad variable de miembros de diferentes tipos implicaba la creación de un contenedor que pudiera iterar sobre contenedores con diferentes tipos, lo cual resultaría en una implementación excesivamente compleja que seria difícil de mantener a largo plazo. Por otro lado una implementación en base a templates no podría implementar una interfaz única de acceso, ya que C++ no permite la utilización de métodos virtuales con *templates*. Debido a estas limitantes el diseño fue descartado.

En tercera instancia se consideró el remover la jerarquía de herencia en su totalidad, en parte considerando que las subclases implementadas no existen fuera de *Camaron*, reemplazando la clase abstracta *Model* por una clase concreta que almacenara un vector para cada elemento disponible por el programa, entregando acceso a cada vector utilizando la interfaz de los subclases originales en una clase única (Ver figura 4.3).

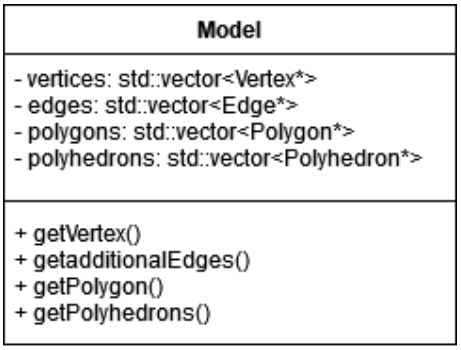


Figura 4.3: Tercera propuesta de arquitectura

Una ventaja de esta implementación consiste en que un objeto de tipo *Model* es funcional inicialmente, permitiendo la creación de modelos vacíos que pueden ser utilizados por cada componente. En caso de que un modelo no tuviera un componente en particular, este retornaría el tamaño del vector correspondiente, por lo que este retornaría cero, sin necesidad de requerir información a través de *double dispatch*. Por otro lado este cambio simplificaría la creación de un nuevo componente dentro del sistema, ya que ahora este no debe crear la infraestructura mencionada en la sección 3.2.1. Debido a que la cantidad de elementos depende del contenido de los vectores, se podrían eliminar los contadores de elementos actualmente almacenados en la clase *Model*.

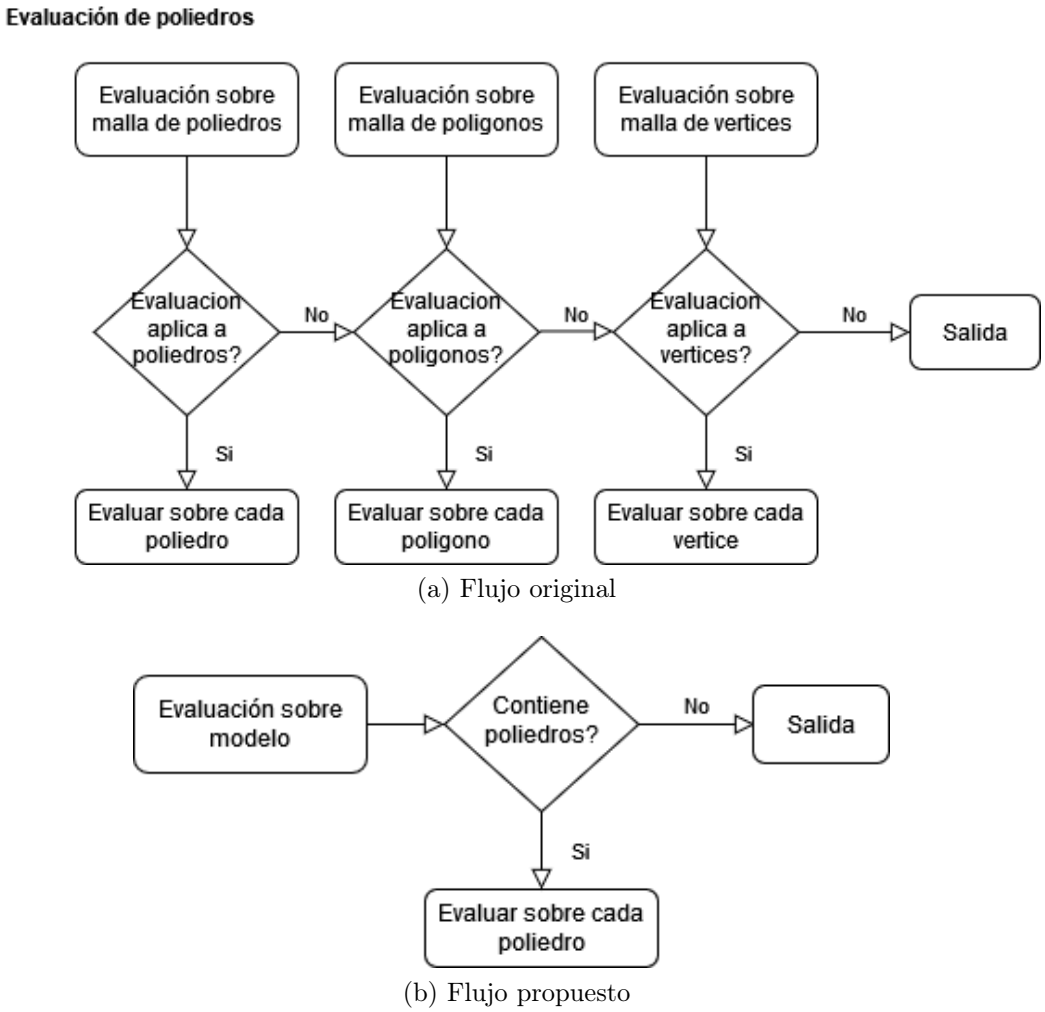
Este cambio implicaría la eliminación de información asociada a cada subclase, así como el reemplazo de los puntos de entrada de cada componente por un punto de entrada único. Por ultimo este tipo de diseño entrega la responsabilidad de escoger el uso apropiado de este modelo al componente en si, resultando en la utilización de condicionales para determinar si un componente existe o no en el modelo recibido.

Para poder determinar la opción a considerar, se decidió estudiar como cada componente actualmente utiliza la clase *Model*, entendiendo las necesidades de cada uno para poder funcionar de forma correcta, en particular se pudo determinar lo siguiente:

- Carga de archivos: Las estrategias conocen los elementos que pueden ser almacenados en cada archivo, requiriendo acceder al contenedor de cada tipo durante la ejecución
- Evaluación: Cada estrategia sabe que tipo de elemento necesita para operar en tiempo de compilación, por lo que necesita acceder a un solo tipo de elemento.
- Selección: Estas estrategias utilizan el tipo almacenado en la clase *Selección*, por lo que este necesita acceder al tipo actualmente activo
- ModelStatistics: Debe poder acceder a todos los elementos almacenados dentro del modelo
- Exporte: Similar a la carga, requiere acceso de forma individual a los contenedores necesarios
- RModel: Necesita saber si un modelo es compuesto solo de vértices o no para operar, requiriendo acceso individual a cada contenedor.

En particular se puede notar que la mayoría de los componentes acceden a cada contenedor individual, requiriendo saber si un modelo posee elementos de un tipo especifico en vez de necesitar acceder a todos los elementos disponibles. Si comparamos esta necesidad con la implementación actual examinada en la sección 3.2.1, se puede notar que los componentes utilizan las llamadas de *double dispatch* para determinar si un tipo de elemento existe en el modelo para luego determinar si la estrategia es válida para ese tipo. Es así como se en realizan estas preguntas para cada tipo de elemento contenido en el sistema hasta encontrar un caso compatible, resultando en un flujo equivalente a utilizar condicionales para preguntar por cada tipo, con un costo extra asociado a la mantenibilidad del código.

En este sentido, la implementación de un modelo único permite que los componentes puedan realizar consultas de forma directa a cada modelo, por lo que en casos como la Evaluación de Modelos se podría simplificar el flujo en base a utilizar la información ya entregada por la estrategia misma. En caso de necesitar acceder a cada contenedor se puede utilizar una cadena de condicionales lo cual sería equivalente a la implementación actual



(a) Flujo original

(b) Flujo propuesto

Figura 4.4: Flujo en el caso de una evaluación de poliedros: (a) representa el flujo actualmente implementado, el cual puede ser reemplazado a través de condicionales en el caso de utilizar un modelo único, (b) representa la simplificación del flujo utilizando la información utilizada por la estrategia, la cual requiere de un modelo único para poder ser implementada.

Notar que en la figura 4.4(a), si la estrategia de evaluación recibe una malla de polígonos. La estrategia debe determinar si puede operar sobre polígonos y vértices, siendo que la evaluación es aplicada a poliedros. Ante lo anterior mas las ventajas expuestas anteriormente, se procedió en la implementación de la tercera propuesta de jerarquía (Modelo único). Esto resulto en las siguientes modificaciones al sistema:

- Reemplazo de puntos de acceso en base a *overloads* por un punto de acceso único, utilizando un puntero de tipo Model.
- Eliminación de las clases VertexCloud, PolygonMesh, PolyhedronMesh
- Eliminación de constantes de tipo (*CONSTANTS::VERTEX_CLOUD, CONSTANTS::POLYGON_MESH, CONSTANTS::POLYHEDRON_MESH*)
- Eliminación de métodos de Double Dispatch en la clase Model
- Eliminación de contadores de elementos

Una ventaja de utilizar un único punto de acceso en cada componente, consiste en que acciones que deben realizarse previamente solo deben realizarse una vez, como ejemplo originalmente la clase RModel liberaba la memoria almacenada en la GPU dos veces en caso de cargar una malla de poliedros debido al flujo anterior. Por otro lado la utilización de condicionales hace mas legible el flujo de operación para cada caso, reduciendo también la presencia de código duplicado en ciertos casos.

Si bien esta implementación logra resolver las problemáticas mencionadas previamente, la simplificación de esta jerarquía puede ser mejorada por una abstracción apropiada, siempre y cuando esta permita a un componente poder tomar decisiones en base a la información de cada tipo.

4.1.2. Localidad espacial en memoria RAM

Tras implementar los cambios a la jerarquía de la clase *Model* se procedió a optimizar la organización de los elementos dentro de la memoria RAM. Para ello se reemplazo la utilización de un vector de referencias, por un vector de instancias de elementos específicos, para asegurar que estos se encontraran almacenados de forma contigua. Para lograr esto, a la hora de cargar el contenido de un modelo, se usa el método *emplace_back*, el cual realiza una construcción de cada modelo insitu, evitando la creación de copias en el proceso. Por otra parte se debió imponer el acceso de estos elementos a través de referencias, para evitar la realización de copias innecesarias.

Una ramificación importante de este cambio corresponde a que cada contenedor debe reservar el número exacto de elementos de cada tipo, quitando la posibilidad de extender un vector de forma dinámica durante la carga. Esto ya que el extender el vector ahora resulta en la eliminación de la sección de memoria que contiene a estos objetos en vez de la creación de nuevas referencias. Debido a esto, el crear referencias a estos elementos durante la carga

resultan en la invalidación de punteros ya existentes, en consecuencia causando instancias de *segfault*.

Las estrategias de carga afectadas correspondían a los formatos Ele/Node y M3D, los cuales al cargar mallas de poliedros, solo obtenían el número de polígonos total tras cargar la totalidad de los poliedros del modelo, debido a que por cada poliedro que se construyera, se agregaban punteros de vecindad los cuales eran invalidados posteriormente. Esto resulto en la deshabilitación temporal de la lectura de estos formatos.

Para sobrellevar esta dificultades, se modificó el algoritmo de lectura para que durante la construcción de poliedros, este almacenara la información relevante a través de la posición esperada de cada polígono único dentro del sistema a través de buffers temporales, los cuales luego son utilizados en una segunda etapa para construir la cantidad exacta de polígonos necesarios para describir el modelo. Al mismo tiempo se modificó el *HashTree* asociado a estos para manipular *integers* con las posiciones en vez de referencias para así no causar invalidación de punteros.

Estos cambios resultaron en una mejora de performance importante en cada etapa del proceso de carga de modelos, ya que la mejora en la localidad espacial permite al procesador optimizar el acceso de elementos adyacentes. Asimismo se logro reducir el overhead de memoria asociado a la alocacion individual de elementos, así como la eliminación del costo asociado a almacenar punteros de cada elemento (8*Número de elementos total en el modelo). En el caso de la carga de modelos Ele/Node y M3D, su performance no se vio afectada de forma significativa, reduciendo el consumo de memoria asociada a la expansión del contenedor durante la alocacion dinámica de su contenido, sin perdida de la funcionalidad original.

4.1.3. Separación entre valores de propiedades y elementos en base a indexación implícita

El siguiente cambio consistió en reducir el consumo de memoria realizados por los cache de evaluación presentes en cada elemento geométrico del modelo. En el caso de las estrategias de evaluación, se decidió el almacenar los valores computados por una estrategia dentro de esta misma, delegando la responsabilidad de actualizar y manipular este cache a la clase misma sin necesidad de acceder a cada elemento por separado. Este cache pudo ser implementado como un vector de *floats*, donde la posición de cada valor estuviera asociado al elemento presente en la misma posición dentro del contenedor de la clase *Model*.

Por otro lado, para almacenar la información cargada de cada campo de propiedad, se modificó la clase *PropertyFieldDef* para que almacenara cada valor leído, realizando la asociación entre elementos de forma similar a lo mencionado en el párrafo anterior. Para poder acceder a esta información, el componente que utilice esta información realiza la consulta al vector de propiedades originalmente presente en *Model* en vez de consultar a cada elemento por separado.

Ambos cambios permitieron la remoción del objeto *LowMemoryHash* dentro de cada elemento, así como la eliminación de los miembros asociados a área y volúmenes existentes en los elementos asociados a estas propiedades. En consecuencia, reemplazando el coste inicial

de memoria asociado cada *LowMemoryHash* presente, por el coste de un vector (24 bytes) por cada propiedad utilizada, permitiendo que un modelo cargado inicialmente solo almacene esta información en caso de necesitarla. Por otro lado, el leer todos los valores computados de forma secuencial se ve beneficiado por esta nueva representación, mejorando la localidad espacial de memoria y en consecuencia obteniendo los beneficios mencionados en la sección anterior.

Por último en el caso de los componentes que deben acceder a estos valores, los cambios realizados permiten a estos componentes el entregar una referencia al cache completo para su uso, reemplazando la implementación anterior la cual requiere consultar los valores de cada valor de forma separada.

4.1.4. Eliminación de variantes *Lightweight*

En base a los cambios realizados a la jerarquía de modelos mas las falencias mencionadas en la sección 3.2.7, se procedió a eliminar las clases asociadas a estas variantes, ya que se determinó que su implementación actual no lograba cumplir con su objetivo y era demasiado costosa su implementación. Esto resulto en la eliminación de las clases *LightWeightVertexCloud, LightWeightPolygonMesh, LightWeightPolyhedronMesh, LWElement, LWVertex, LWPolygon, LWPolyhedron, ModelLoadingOFFLW, ModelLoadingEleNodeLW, ModelLoadingPLYLW y CalculateVertexNormalsLightWeightWorker*, así como las constantes asociadas a cada variante y a la lógica asociada a manipular estos componentes. Estos cambios pueden ser encontrados dentro del commit *a1569fe* en caso de necesitar acceder a la implementación original de estos.

4.1.5. Reemplazo de CharArrayScanner por lector en base a streams

Con el objetivo de corregir la fuga de memoria presente en el componente de carga de modelos y de reducir las instancias de código duplicado, se procedió a refactorizar este componente en base al concepto de *streams* introducido en *C++*, continuando con el trabajo previo realizado en esta materia [4](Sección 4.2.3). En este sentido se recopilo una lista de casos de uso presentes dentro de cada estrategia de carga actualmente implementadas, para así poder guiar el diseño de un nuevo componente.

- Capacidad de leer un archivo sin cargar su contenido completo en memoria RAM
- Entregar una interfaz única para la lectura de diferentes tipos de variables
- Capacidad de escoger una codificación especifica (ASCII, Binario *little endian* o *big endian*)
- Permitir el cambio de la codificación durante la ejecución, para cubrir casos donde un formato posee múltiples codificaciones (PLY)
- Capacidad de mover la posición del lector a la *linea* siguiente (En ASCII corresponde a posicionar el caret tras el fin de linea, mientras que en binario la linea termina tras un numero específico de bytes)

En base a estos requerimientos, se construyó la clase *StreamScanner*, la cual manipula internamente un *stream* de archivos de tipo (std::ifstream). La codificación es representada por un *enum*, almacenado dentro de la clase y que puede ser modificado a través de un setter publico. Para lograr una interfaz unificada, se optó por implementar el operador $\gg$ en base a *templates*, permitiendo la lectura de elementos de diferente tipo al encadenar múltiples variables de diferente tipo.

Tras la implementación de este componente se logró traspasar el manejo de la codificación del archivo a este nuevo componente, permitiendo reducir la complejidad de los algoritmos de lectura al utilizar esta interfaz para definir el método de lectura durante las etapas iniciales del código.

4.2. Modificaciones por componente

4.2.1. Carga de Modelos

A nivel de componente, los cambios realizados a la clase *Model*, posibilitan una utilización mas flexible de este contenedor, permitiendo utilizar un modelo vacío sin necesidad de asumir su tipo previo a la lectura del archivo, permitiendo reservar y llenar cada contenedor interno durante la ejecución, sin necesidad de realizar *casting* ni desambiguaciones en base al tipo especifico escogido. Esto permite implementar un diseño de estrategia centrado en los elementos geométricos que son soportados por los formatos, en contraste con versión anterior la cual se centraba en las subclases de *Model* que podrían ser necesitadas.

Por último, se completó la utilización de excepciones durante las operaciones de lectura de elementos, resultando en la eliminación de mecanismos alternativos para el manejo de errores (*boolean* o *nullptr*), para así lograr un manejo excepciones consistente a lo largo de cada estrategia de carga implementada.

PLY

Para este caso se logró utilizar la integración de *StreamScanner* para extender la estrategia a codificaciones de *little endian*. Al mismo tiempo esta integración permitió unificar las versiones de lectura de elementos binarios y en ASCII. Esta integración también logró corregir la lectura incorrecta de ciertos polígonos al permitir la lectura de los valores numéricos en caso de leer caracteres en formato ASCII a través de la implementación interna de *StreamScanner*

En el caso de la lectura de propiedades, se logró asignar el tamaño correcto a los tipos definidos en los archivos, permitiendo corregir la regresión mencionada durante el análisis de este componente. Sin embargo, la lectura de propiedades de polígonos no pudo ser corregida en el tiempo destinado, por lo que el detalle de esta falla se listará en la sección de recomendaciones.

En el caso de los errores asociados a propiedades de caras o a lectura de archivos con headers dañados, la implementación del mecanismo de excepciones mencionado

anteriormente permite que el componente pueda continuar operativo, corrigiendo los errores de tipo *segfault* asociados a estos casos.

Ele/Node

El error de normalización se debió a que internamente, los polígonos son creados al agrupar los vértices que componen a un tetraedro de forma secuencial, sin considerar la orientación que estos tendrán en el modelo. El manual de TetGen[11] especifica en su sección 5.2.4 la distribución espacial de cada vértice (ver fig 4.5), lo cual permitió definir un orden de lectura correcto para la corrección de este error.

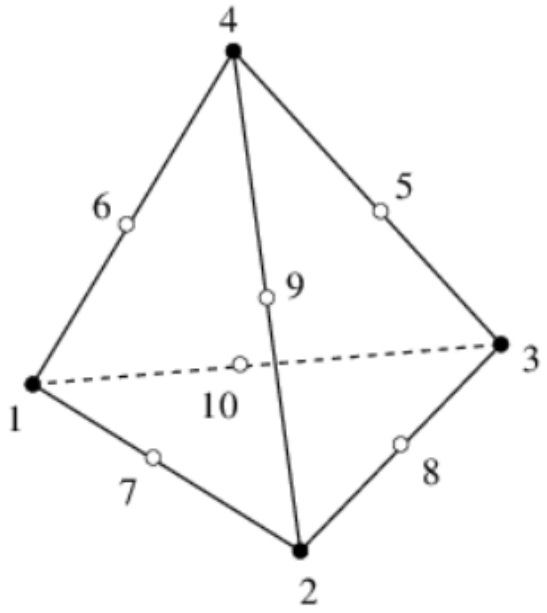


Figura 4.5: Distribución espacial de los vértices de cada tetraedro. Para nuestro caso solo son relevantes los vértices 1 a 4, ya que *Camaron* no considera el resto de los puntos. En el algoritmo original las caras 1-2-3 y 1-3-4 resultarían en el computo de normales invertidas, por lo que estas deben leerse en sentido horario para evitar este tipo de falla.

VisF

Inicialmente se inicio un trabajo de recuperación de los archivos codificados bajo especificaciones deprecadas. Para ello se utilizo un editor de archivos binarios para agregar manualmente un byte de endianness sobre los archivos afectados.

Posteriormente se procedió a extender el formato *VisF* para permitir el soporte de codificación ASCII. Para ello se modificó la especificación existente, definiendo que el primer valor del header del archivo correspondería al tipo de codificación utilizada en este. Este byte siempre utiliza la codificación ASCII, para simplificar la lectura de los archivos.

Por último se realizaron modificaciones similares a las aplicadas sobre el formato PLY para permitir la lectura de archivos para cualquier ordenamiento.

M3D

Por un lado la recuperación de funcionalidad de este formato fue lograda tras la

corrección del algoritmo de tetraedralizacion localizado en el método *PolyhedronUtils::getTetrahedronIndeces*. En teoría este algoritmo debía utilizar el siguiente algoritmo para convertir poliedros convexos a una serie de tetraedros simples:

1. Escoger primer vértice del poliedro (pivote)
2. Por cada polígono incluido en el poliedro que no incluya al pivote;
   (a) Triangularizar cara objetivo
   (b) Crear tetraedro entre pivote y cada triangulo generado
   (c) Representar tetraedro dentro de buffer en GPU

El error fue identificado dentro de la etapa de triangularizacion. En su versión original el método afectado escogía grupos de tres vértices para crear los tetraedros, con la diferencia de que estos grupos siempre eran compuestos de vértices diferentes. Debido a esta diferencia, en el caso que el algoritmo recibiera un poliedro con una cara no triangular, este causaría un *segfault* al intentar utilizar los últimos vértices de la cara, los cuales al ser menos que tres el algoritmo estaría accediendo a información fuera de límite. Este error fue corregido de forma exitosa al aplicar una triangularizacion correcta en la sección afectada, permitiendo la carga de los modelos afectados.

En segunda instancia, el error en la orientación de normales se debió a una falla de lectura similar a la evidenciada en el formato Ele/Node, con la diferencia que este formato carecía de una especificación original que definiera el posicionamiento espacial de cada vértice. En vista de esto, se decidió imponer un ordenamiento especifico sobre las caras de cada prismas actualizando la especificación de este formato en el proceso.

TS

En base a lo expuesto en el análisis de este formato en la sección 3.2.2.1, se resolvió la eliminación del soporte para este tipo de archivo, resultando en la eliminación de las clases *ModelLoadingTS* y *ModelExportTS*. Estas clases pueden ser encontradas dentro la revisión *4277f84* en caso de requerir su recuperación.

4.2.2. Procesamiento de mallas

Debido al acoplamiento entre la carga del archivo y el procesamiento del modelo creado, se procedió a separar ambas etapas. Para ello primero se movió cada rutina asociada al procesamiento de mallas, a la nueva clase *MeshProcessor*, la cual encapsula la inicialización de cada etapa de procesamiento en un método estático, Este método estático puede ser invocado por la clase base *ModelLoadingStrategy*. En segunda instancia se movieron las rutinas de estas fases previamente presentes en cada estrategia a la clase base, siendo estas ejecutadas tras completar el método *load*.

A continuación, en base a lo mencionado en la sección 3.2.2.2, se procedió a quitar la etapa *FixSurfaceNormals*. Al mismo tiempo se reordenaron las etapas de la pipeline con objeto de simplificar el código, resultando en el flujo presentado en el diagrama 4.6.

Notar que en el caso del formato *VisF*, un archivo puede incluir de antemano la información de vecindad asociado al calculo de relaciones polígono-polígono, debido a esto la clase

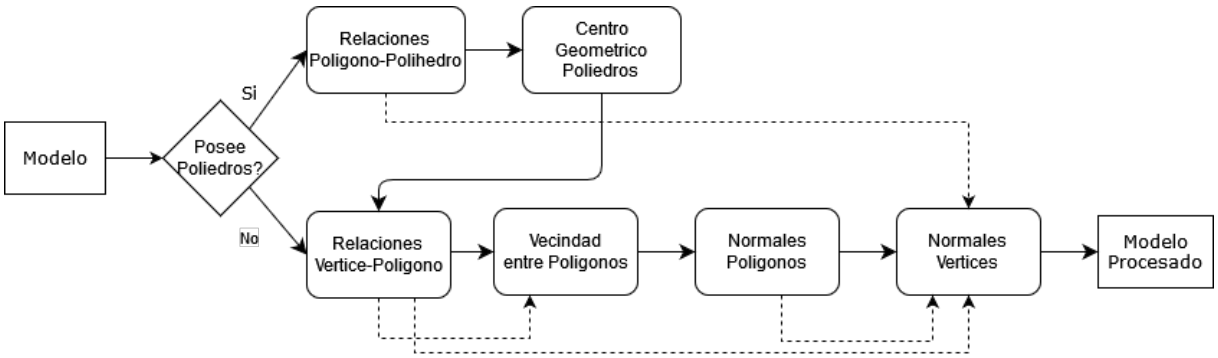


Figura 4.6: Versión actualizada para la fase de procesamiento. Notar que ahora se recibe un puntero de tipo Model, consecuencia de los cambios implementados en la sección 4.1.1

ModelLoadingVisF implementa su propia versión de los métodos asociados a la pipeline, para así evitar la ejecución de este paso mas de una vez, utilizando el mecanismo de *override* sobre estos métodos.

Si bien se consideró en un punto el separar aun mas esta fase al convertirla en un componente de *Camaron* separado, esta dependencia entre el formato *VisF* y la implementación de la pipeline hace dificultoso este prospecto. Debido a la complejidad asociada a separar estos componentes no se implementó esta separación completa durante este proyecto.

Con respecto a la presencia de normales con valores de tipo *NaN*, se determinó que esta falla era causada en modelos donde la suma de las normales de las caras vecinas a un punto sea igual a cero (Ver fig 4.7). La etapa de calculo de normales de vértices obtiene esta normal utilizando la suma anteriormente mencionada, al ser normalizada resulta en una división de cero flotante (Ver sección 3.1.4).

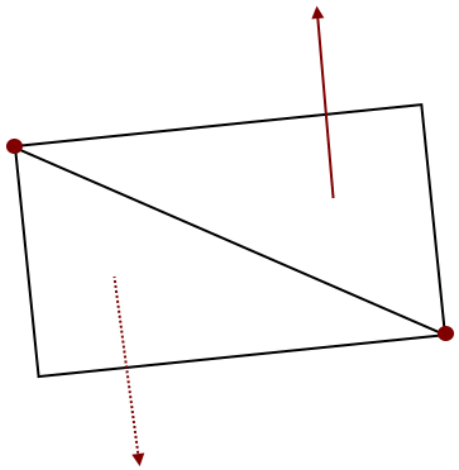


Figura 4.7: Ejemplo de modelo que reproduce el error, en particular los vértices marcados en rojo causan la división por cero, mientras que los demás vértices poseen una normal idéntica a la única cara adyacente a estas

Para corregir esta falencia, se decidió agregar un condicional extra a la operación para

así evitar la normalización de este vector en caso de ocurrir. En el caso del segunda falencia, el error fue corregido simplemente especificando el tipo correcto asociado al parámetro de número de *threads*, resultando en un tiempo de ejecución equivalente a un cuarto del algoritmo equivalente secuencial (Esto realizado con el numero default de threads definido en el código igual a 8)

4.2.3. Estrategias de evaluación

En base al desacoplamiento de propiedades realizado en la sección 4.1.3, se incluyo dentro de cada estrategia de evaluación un cache adicional para el almacenamiento de los datos dentro de cada estrategia. Por otro lado se procedió a dividir el punto de entrada en los métodos *evaluateElementsFrom* y *requestEvaluationResultsAsRModel* permitiendo suplir con los casos de uso específicos de cada caso de uso. Debido a que ahora el cache se encuentra en memoria secuencial, es posible entregar el cache en su totalidad a través de la entrega de una referencia a este.

Por otro lado para reducir la complejidad ciclomática del componente se creo un jerarquía de clases para separa las evaluaciones por tipo geométrico a las cuales son aplicadas (ver Figura 4.8, permitiendo reducir el numero de consultas a un mínimo.

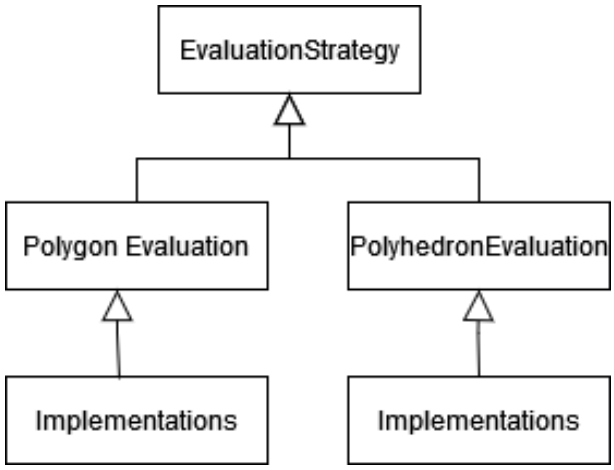


Figura 4.8: Diagrama de nueva jerarquía de evaluación, las evaluaciones sobre elementos agrupan a las implementaciones de cada evaluación originalmente incluida como hijos de la clase EvaluationStrategies

El conjunto de estos cambios, mas los cambios realizados sobre la jerarquía de la clase *Model* permitió el poder implementar el flujo propuesto en la figura, permitiendo un flujo mas simplificado del componente.

4.2.4. Carga de propiedades

En primer lugar se consideró la posibilidad de recuperar la funcionalidad de la carga de modelo de tipo TQS. Inicialmente esto implicaría separar la implementación actual en una estrategia de carga de modelo y una estrategia de carga de propiedades por separado, incluyendo la implementación efectiva de las clases utilizadas en el código fuente original. Si bien esto en teoría seria posible, se notó que *Camaron* no posee implementada una forma de renderizar su contenido, por lo que también debería ser implementada este renderizado. Por otro lado, el formato TQS no es utilizado normalmente por un usuario, siendo un caso de uso de nicho al ser este un formato sin una especificación externa, por lo que el acceso a estos modelos también se encuentra limitada.

En base a lo anterior mas las falencias mencionadas el análisis de este componente y la falta de documentación asociada, se decidió en conjunto con la profesora guía la eliminación del soporte para este formato. Esto resulto en la eliminación de las clases *PropertyFieldLoadingTQS, TQSPropertyFieldDef*. En el caso de necesitar implementar este soporte nuevamente, el código fuente de estas clases puede ser recuperada accediendo al commit 68699f9.

En segundo lugar se logró corregir los dos errores asociados mencionados en la sección de análisis de este componente. En el caso del formato PLY, la interfaz entrega un mensaje de advertencia en caso de leer información binaria, el cual corresponde al comportamiento esperado por la aplicación.

4.3. Cambios generales menores

Un análisis inicial de la ejecución del código dio indicios que la aplicación incurría en *segfault* durante la construcción de la clase Visualizador, la cual a su vez inicializa la mayoría de los componentes del sistema.

Debido a la complejidad del problema, se decidió utilizar los tests unitarios que serían desarrollados en este proyecto, con el objeto de poder aislar los componentes que pudieran causar esta falla. Esta forma de abordar el problema logró rendir frutos durante el análisis del Componente de Estrategias de Evaluación, siendo la falla causada por la utilización del *Widget* de Qt5 *QLineEdit* dentro de la interfaz visual de ciertos componentes.

El componente en cuestión funcionaba de forma correcta durante el desarrollo previo del software, por lo que se consideró una posible incompatibilidad entre la versión de Qt5 utilizada (5.4), con respecto a la versión del compilador dentro de la distribución actual. En base a esto se compilo el programa con la versión LTS de Qt5, eliminando la ocurrencia del error. Debido a que esta migración de versiones ya se encontraba en los planes de desarrollo, se consideró por parchada la falla presentada.

4.4. Test unitarios y de stress

Acompañando el trabajo de reingeniería se implementaron en paralelo una serie de test unitarios y tests de estrés para asegurar la funcionalidad de *Camaron*. En particular debido a que los cambios mas importantes involucraron a la jerarquía de *Model*, la mayoría de estos casos de prueba fueron creados alrededor de los componentes que manipulan esta abstracción de forma directa.

Estos tests utiliza la librería QTest, permitiendo crear tests que interactúen con componentes de la interfaz gráfica que dependan de la funcionalidad de Qt5. Estos tests son compilados de forma separada, a través de los siguientes comandos:

```
qmake "CONFIG += test"
make
```

El ejecutable resultante ejecuta cada test unitario de forma secuencial. Debido a que los tests de estrés son significativamente mas lentos que el resto, estos tests se encuentran inactivos por default, permitiendo su habilitación al incluir el parámetro *-B* en la consola al momento de ejecutar los tests. En el caso de los tests de exporte, es posible acceder a los archivos creados agregando el parámetro *-k* para evitar su eliminación.

Para el apoyo en la creación de tests, se implementaron las siguientes clases auxiliares:

MeshExtensions Permite realizar comparaciones entre modelos, comparando el contenido de cada elemento geométrico almacenado y el número de elementos almacenado en cada caso.

OpenGLContext Clase que permite la creación de un contexto de OpenGL básico. Este contexto es implementado a través de la librería GLFW, la cual permite acceder a esta funcionalidad sin la necesidad de iniciar una aplicación de *Qt* asociada. Este componente es utilizado a la hora de testear funcionalidad que realiza llamadas de OpenGL pero que no utilizan la interfaz de *Camaron*.

UnitCube Clase que contiene diferentes representaciones de un cubo unitario. En particular esta contiene dos representaciones, una en base a un cubo (1 poliedro) y la otra en base a tetraedros (6 poliedros), con cada una incluyendo su representación de acuerdo a las subclases de la clase *Model* existentes.

MockModelLoading Clase similar a *MeshProcessor*, la cual posee implementaciones sin utilización de *thread* para cada etapa presente durante el procesamiento de modelos.

Tras el trabajo realizado, se lograron implementar 98 tests unitarios sobre los componentes asociados a la manipulación de la clase *Model*, por cada componente se describe el tipo de test.

Carga de modelos Para cada formato definido, se implementaron casos de prueba para la validación de archivos, la carga de los diferentes tipos de mallas y de los diferentes tipos de codificación asociadas. Se incluyen casos de borde para alguno de los formatos listados durante la etapa de análisis. Estos tests utilizan modelos de un cubo unitario escritos a mano, los cuales corresponden a las configuraciones entregadas por la clase *UnitCube*

Exporte de modelos Se utiliza la clase *UnitCube* para guardar el contenido de este en archivos temporales, los cuales son cargados a través de estrategias de carga y su contenido es comparado con la clase *UnitCube*

RModel Clase única de test, que corrobora que los procesos de triangularización se realicen de forma de correcta, aplicándose a cada tipo de modelo disponible.

Carga de propiedades Se crean dos clases de testeo para los formatos Ele/Node y PLY, los cuales cargan el contenido de archivos de prueba para corroborar que sean leídos de forma correcta.

Selection Se realizan tests para la clase *Selection*, así como tests asociados a las estrategias: *ChangeSelectionType, ExpandToNeighbors, SelectByID y SelectByProperty*. En el caso de las selecciones de intersección y de mouse, la implementación no pudo ser concretada debido a la complejidad asociada a estos componentes.

Por último se crearon dos tests de estrés, con el objeto de medir el tiempo de ejecución de las etapas de carga y procesamiento de modelos. Estas utilizan la macro QBENCHMARK y la clase CrossTimer previamente implementada en el diseño original[3]. Ambos tipos de tests utilizan los modelos de esferas mencionados anteriormente, por lo que su utilización requiere la presencia de estos archivos. Estas mediciones pueden ser utilizadas posterior a este trabajo, para poder evaluar la performance de estos componentes ante cambios de arquitectura futuros.

Capítulo 5

Resultados

Tras completar el proceso de refactorización, se procedió a rehacer las pruebas de estrés mencionadas en la sección 3.4 de este documento, logrando una mejora significativa en el rendimiento de la aplicación con respecto a su consumo de memoria y velocidad de ejecución.

5.1. Consumo de memoria RAM

Con respecto a este ámbito, se logró una reducción promedio de un 18 % en los casos testeados (Ver tabla 5.1), esta mejora permitió lograr la carga exitosa de los modelos de prueba *lucy* y *Esfera 10*, así como una carga mas expedita de los modelos de menor tamaño al requerir menos recursos.

	Carga		Procesamiento		RModel		Estable	
Modelo	Nuevo	Original	Nuevo	Original	Nuevo	Original	Nuevo	Original
xyz_figurine	2,04	3,81	3,34	4,72	5,72	7,15	5,54	6,96
Esfera 8	2,22	3,48	3,49	4,55	5,52	6,28	5,43	6,3
Esfera 9	4,39	6,62	6,34	8,9	10,59	12,56	10,34	12,53
Esfera 10	8,12	12,06	16.02	–	23.4	–	22.9	–
lucy	5,98	–	9,56	–	15,02	–	14,97	–

Tabla 5.1: Alocación promedio de memoria RAM en GB, comparando los valores obtenidos en la sección 3.9 y la versión actual

En el caso de los modelos de esfera, el modelo de esfera 9 ahora alcanza el máximo de memoria RAM sin *thrashing* ya avanzado la transferencia de información a la GPU, por lo que ahora es factible la carga de este modelo con un bajo riesgo de que la aplicación quede sin responder por un tiempo prolongado.

Por otro lado, se logró mover el límite de carga máximo desde la esfera 9 a la esfera 10 bajo las limitaciones de hardware presentadas durante el desarrollo, permitiendo obtener información de este modelo que puedan ser utilidad en iteraciones posteriores. Si bien el poder

avanzar un escalafón dentro de la lista de esferas pareciese poco, este salto es significativo al este poseer mas de 42 millones de elementos individuales.

5.2. Velocidad de ejecución

Las mejoras de consumo de memoria mas las mejoras de localidad de memoria presentadas durante el proceso, lograron reducir el tiempo de ejecución de forma importante, resultando en un baja promedio de un 75 % del tiempo de ejecución total, siendo la mejora mas grande la presentada durante la carga de la esfera 9, con una reducción de un orden de magnitud comparado con la versión original de *Camaron* (Esfera 9, columna Procesamiento, tabla 5.2).

	Carga		Procesamiento		RModel		Total		Eliminación	
Modelo	Nuevo	Orig	Nuevo	Orig	Nuevo	Orig	Nuevo	Orig	Nuevo	Orig
xyz_fig	4	13	6,8	22	8	74	18	109	2.5	28
Esfera 8	25	43	24	77	17	44	66	164	3	15
Esfera 9	50	117	73	1380	551	1452	674	2949	277	1200
Esfera 10	220	1800	3960	–	11520	–	15660	–	2910	–
lucy	22	–	690	–	2040	–	2760	–	540	–

Tabla 5.2: Tiempo de ejecución de cada etapa en segundos, comparando los valores obtenidos en la sección 3.10 y la versión actual

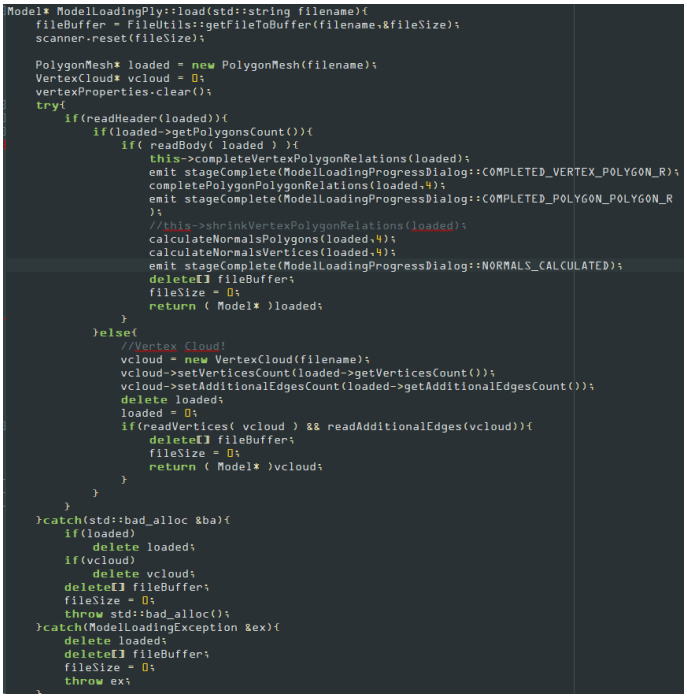
5.3. Mantenibilidad código fuente

5.3.1. Reducción de complejidad algorítmica

El conjunto de modificaciones estructurales listados en el capítulo anterior permitieron la simplificación del control de flujo importante dentro de cada componente revisado, así como de la interacción con la clase *Visualizador* presentada en los diagramas de interacción en el capítulo de Antecedentes.

5.3.2. Carga de modelos

Este componente fue el mas beneficiado dentro de estos cambios. En su versión actual el método *load* ahora solo se encarga de definir el orden en que se deben leer los componentes individuales, permitiendo utilizar un objeto de tipo *Model* durante el proceso completo, construyendo este objeto a medida que el archivo es leído y lanzando una excepción en el caso de ocurrir un imprevisto (Ver Figura 5.1).



```
Model* ModelLoadingPly::load(std::string filename){
    fileBuffer = FileUtils::getFileToBuffer(filename,&fileSize);
    scanner.reset(fileSize);

    PolygonMesh* loaded = new PolygonMesh(filename);
    VertexCloud* vcloud = 0;
    vertexProperties.clear();
    try{
        if(readHeader(loaded)){
            if(loaded->getPolygonsCount()){
                if( readBody( loaded ) ){
                    this->completeVertexPolygonRelations(loaded);
                    emit stageComplete(ModelLoadingProgressDialog::COMPLETED_VERTEX_POLYGON_R);
                    completePolygonPolygonRelations(loaded,4);
                    emit stageComplete(ModelLoadingProgressDialog::COMPLETED_POLYGON_POLYGON_R
                    );
                    //this->shrinkVertexPolygonRelations(loaded);
                    calculateNormalsPolygons(loaded,4);
                    calculateNormalsVertices(loaded,4);
                    emit stageComplete(ModelLoadingProgressDialog::NORMALS_CALCULATED);
                    delete[] fileBuffer;
                    fileSize = 0;
                    return ( Model* )loaded;
                }
            }else{
                //Vertex Cloud!
                vcloud = new VertexCloud(filename);
                vcloud->setVerticesCount(loaded->getVerticesCount());
                vcloud->setAdditionalEdgesCount(loaded->getAdditionalEdgesCount());
                delete loaded;
                loaded = 0;
                if(readVertices( vcloud ) && readAdditionalEdges(vcloud)){
                    delete[] fileBuffer;
                    fileSize = 0;
                    return ( Model* )vcloud;
                }
            }
        }
    }catch(std::bad_alloc &ba){
        if(loaded)
            delete loaded;
        if(vcloud)
            delete vcloud;
        delete[] fileBuffer;
        fileSize = 0;
        throw std::bad_alloc();
    }catch(ModelLoadingException &ex){
        delete loaded;
        delete[] fileBuffer;
        fileSize = 0;
        throw ex;
    }
```

(a) Original

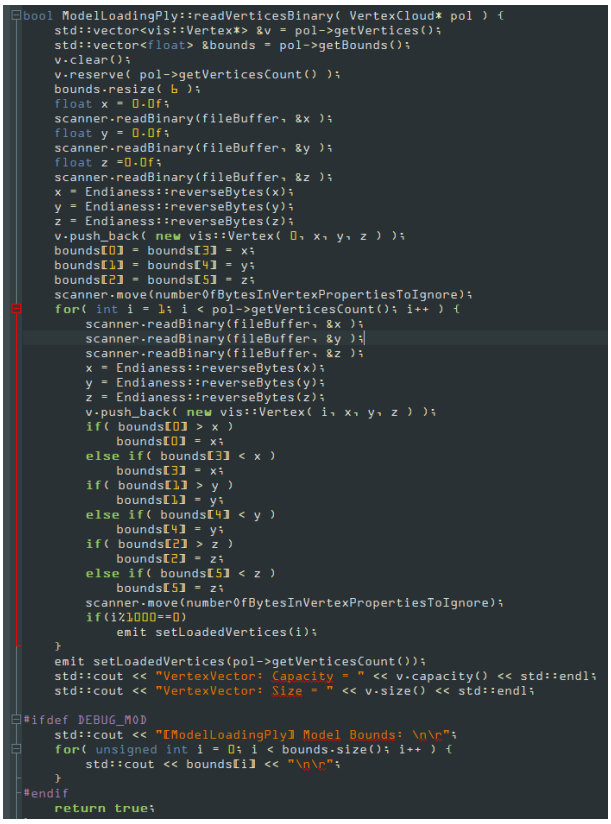
```
Model* ModelLoadingPly::load(std::string filename){
    parser.openFile(filename);

    Model* loaded = new Model(filename);
    vertexProperties.clear();

    try{
        readHeader(loaded);
        readVertices(loaded);
        readPolygons(loaded);
        readAdditionalEdges(loaded);
        parser.closeFile();
        return loaded;
        }

    catch(std::bad_alloc &ba){
        if(loaded)
            delete loaded;
        parser.closeFile();
        throw std::bad_alloc();
        }
    catch(ModelLoadingException &ex){
        delete loaded;
        parser.closeFile();
        throw ex;
        }
    }
```

(b) Refactorizado

Figura 5.1: Comparación de cambios en carga de modelos PLY: (a) el método *readBody* utiliza un condicional extra para verificar que el formato sea binario. (b) Version actualizada tras refactorizacion

La abstracción entregada por *StreamScanner* también logró agrupar implementaciones similares, reduciendo el código duplicado y también simplificando la implementación de cada proceso de lectura asociado. Esto en conjunto con la alocacion de recursos previa permitió crear métodos de carga capaces de trabajar en casos donde un elemento no se encuentra presente (Ver Figura 5.2).

(a) Vértice ASCII



(b) Vértice Binario

(c) Lectura vértices unificada

Figura 5.2: Simplificación de operación en base a *StreamScanner* y delegación de funcionalidades, las dos imágenes superiores corresponden a la implementación original, mientras que la imagen inferior es utilizada actualmente para la carga, definiendo la codificación en una etapa previa

5.3.3. Estrategias de evaluación

La separación de puntos de entrada para cada caso de uso logró simplificar el algoritmo de procesamiento. Al mismo tiempo este permite desarrollar estos casos de uso de forma independiente, reduciendo la dificultad de extender la funcionalidad que interactúa con este

módulo.

Por otro lado la creación de la nueva jerarquía de estrategias de evaluacion mas la eliminación de llamadas de tipo *double dispatch* resultó en un flujo mas directo entre el módulo y los demás sistemas, reduciendo de forma significativa los diagramas de interacción asociados a cada caso de uso, esta reducción se ilustra en los diagramas 5.3, 5.4, 5.5

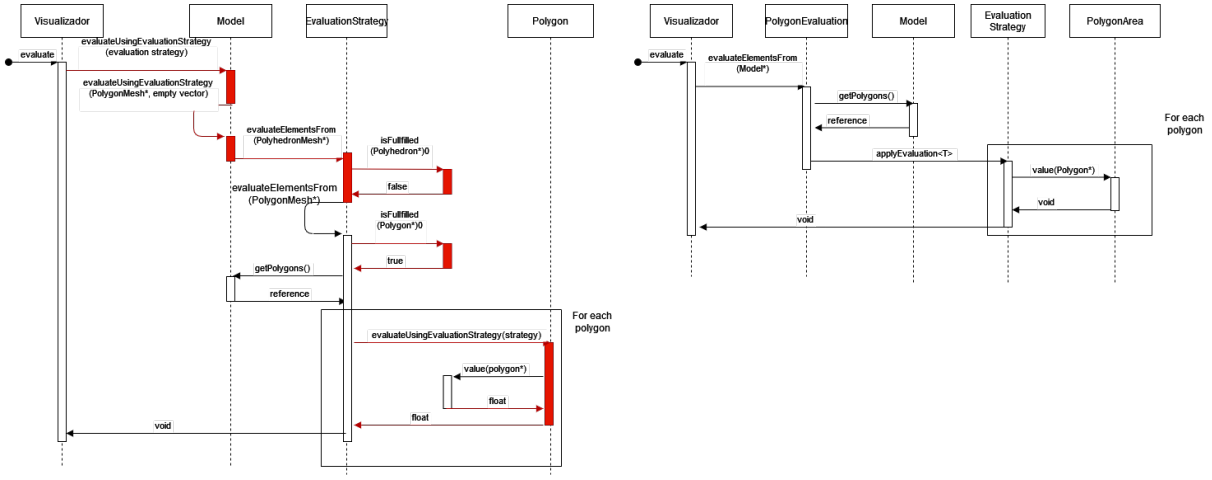


Figura 5.3: Comparación de diagramas de interacción para la evaluación de un modelo. Interacciones eliminadas son marcadas en rojo

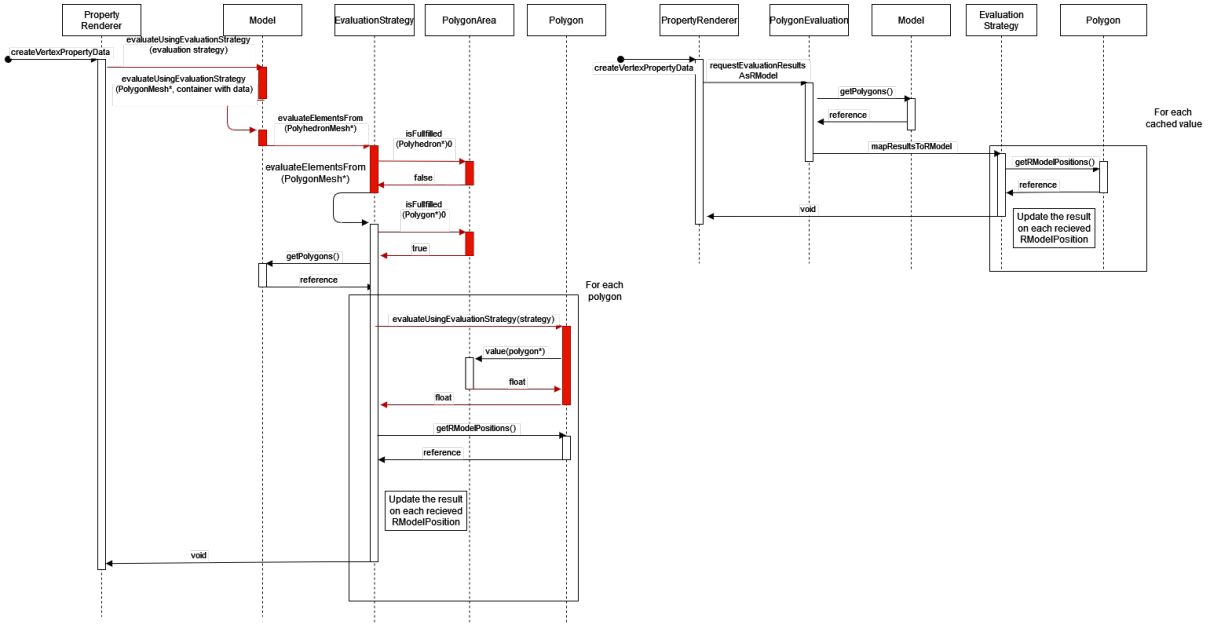


Figura 5.4: Comparación de diagramas de interacción para renderizador de propiedades. Interacciones eliminadas son marcadas en rojo

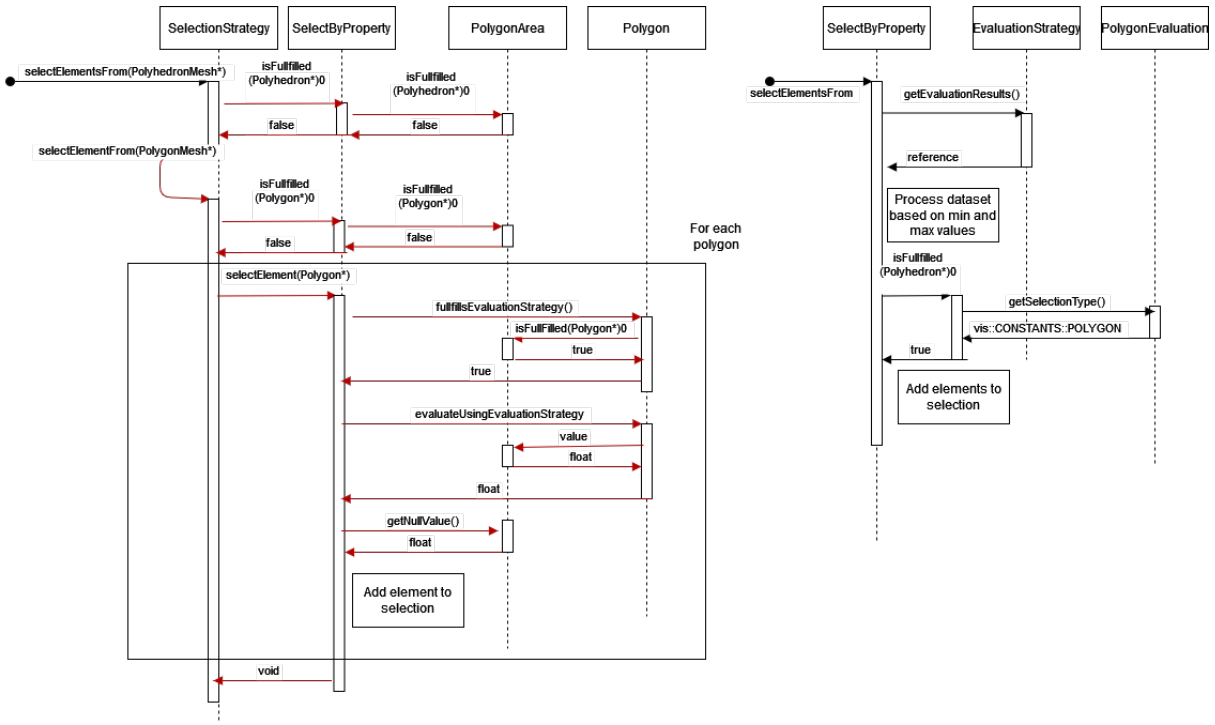


Figura 5.5: Comparación de diagramas de interacción para selección de propiedades. Interacciones eliminadas son marcadas en rojo

5.3.4. Estabilidad del software

El proceso de refactorizacion permitió la reparación de la funcionalidad perdida dentro del módulo de carga de modelos, permitiendo poder cargar nuevamente modelos de tipo PLY binario, M3D, VisF en formato Binario y ASCII.

Este proceso también permitió resolver 30 de las 52 fallas detectadas durante el proceso de análisis. Dentro de las anomalías detectadas, siete correspondían a fallas causantes de *segfaults*, de las cuales seis fueron corregidas, con la séptima siendo parte de una funcionalidad de baja prioridad. Esta falla es documentada en mas detalle en la sección 6.3.1.1. La corrección de estas fallas fatales permite que la versión actual del software sea capaz de ser ejecutado de forma continua sin resultar en la pérdida de progreso ante una caída del sistema, entregando una mejor experiencia al usuario de esta herramienta y entregando mensajes de error apropiados en caso de encontrar falencias.

5.3.5. Portabilidad

Los cambios realizados también permitieron asegurar la portabilidad del sóftware sobre las sistemas operativos Windows y Linux, al lograr corregir todas las anomalías causantes de *undefined behaviour* detectadas a través de *ubsan* y de forma manual. Si bien estos anomalías no son causales de errores fatales en ciertas arquitecturas, su corrección entrega una base importante a la hora de desarrollar el software a futuro, al reducir la posibilidad de encontrar errores difíciles de detectar al ser estos específicos a la arquitectura del desarrollador.

Si bien *Camaron* puede ejecutarse bajo estas restricciones de compilación, es importante recalcar que *ubsan* solo puede detectar falencias de este estilo a medida que el usuario ejecuta el programa, por lo que sigue existiendo la posibilidad de que ciertas decisiones resulten en *undefined behaviour*. Sin embargo, ahora que *Camaron* cuenta con opciones de compilación que permiten habilitar esta restricción de forma sencilla, nuevos desarrolladores pueden detectar estas fallas de forma consistente.

Por último, la migración exitosa del software a la versión *long-term support* de Qt permite asegurar su funcionamiento por un periodo mayor de tiempo. Además permite al usuario el poder descargar la dependencia a través de paquetes ya disponibles en sistemas de gestión de paquetes, en vez de requerir la compilación manual de la versión original, la compilación manual toma un tiempo considerable de tiempo al tener que resolver conflictos entre paquetes asociados a otras dependencias, los que puedan causar errores similares al expuesto en la sección 4.3

Capítulo 6

Recomendaciones

6.1. Cambios a corto plazo

A continuación se listarán una serie de cambios y modificaciones que pueden ser implementadas en el corto o mediano plazo, sin requerir una intervención importante sobre la arquitectura de *Camaron*. Esta sección complementa la información de *Trabajo Futuro*, los cuales corresponden a modificaciones que requieren un trabajo mayor para su implementación.

6.1.1. Implementación de estrategias faltantes

En particular *Camaron* carece de las siguientes estrategias, las cuales podrían ser implementadas con las herramientas presentes tras este trabajo. En particular las estrategias faltantes son:

- Carga de modelos PLY con propiedades asociadas a caras
- Carga de propiedades para modelos PLY binarios
- Carga de propiedades para modelos OFF
- Selección sobre vértices
- Exporte de modelos a formato TRI
- Exporte de modelos a formato PLY
- Exporte de selección para formatos VisF
- Exporte de modelos a formato M3D(*)

En el caso del exporte a formato M3D, debido a la complejidad de su implementación mas su poco uso, la prioridad de esta implementación es significativamente baja, por lo que se considera como una estrategia opcional.

Por ultimo, se recomienda la implementación de estrategias de cargado para formatos de archivos externos populares como Wavefront (.obj), Collada (.dae), Alembic (.abc), entre otros. Esto no solo permitiría extender la adopción de la aplicación, sino que también permitiría la interoperabilidad entre otras herramientas de visualización de modelos 3D.

6.1.2. Excepciones

Si bien *Camaron* posee algunas excepciones asociadas a la carga de modelos y de campos de propiedades, su implementación actual requiere de trabajo para mejorar su utilidad. Al mismo tiempo el resto de los componentes dentro de *Camaron* carecen de excepciones propias, por lo que su implementación permitiría un mejor manejo de errores dentro del sistema. Por ultimo se recomienda implementar estas excepciones como extensiones de las excepciones entregadas por la librería estándar *<exception>*.

6.1.3. Cálculo de centro geométrico para poliedros

Actualmente, el cálculo de este valor es realizado durante la etapa de procesamiento de mallas, resultando en una alocacion extra de 12 bytes por cada poliedro presente. Si bien esta información es utilizada por otras funcionalidades dentro de *Camaron*, se testéo durante el desarrollo el realizar este cálculo de forma dinámica, resultando en una baja de performance menor. Si bien este cambio podría mejorar el consumo de memoria del sistema, su efecto concreto debe ser medido con mayor detalle para poder considerar su remoción, por lo que se recomienda realizar tests sobre los componentes que requieren de esta funcionalidad para determinar la factibilidad de esta modificación.

6.1.4. Exporte de mallas

A nivel de interfaz, la aplicación no entrega suficiente información al usuario para poder determinar que extensiones se encuentran disponibles para exportar. Esto resulta en el usuario recurriendo a ensayo y error para intentar guardar un archivo, recibiendo un mensaje de error en el caso de no ser posible. Si bien esto no es una falla de prioridad alta, resultaría en una mejora en la usabilidad del proyecto si este explicitara cuales formatos se encuentran disponibles de forma transparente.

6.1.5. Exclusividad en el contenido de los objetos de tipo Selección

Si bien la clase *Selection* permite almacenar mas de un tipo de elemento, esta funcionalidad no es un comportamiento útil, debido a los problemas mencionados durante la sección de análisis. En este sentido se recomienda modificar esta clase para que solo permita el almacenamiento de un solo elemento, esto puede ser implementado a través de una jerarquía

de clases estándar o a la utilización de *templates*, por lo que se deja la evaluación de estas opciones a criterio del desarrollador posterior.

6.1.6. Completitud de tests no implementados

Si bien se logro implementar un número importante de pruebas unitarias y de stress durante el desarrollo de este proyecto, existe un grupo de tests adicionales no pudieron ser implementados en el tiempo disponible. Estos se encuentran deshabilitados a través del macro *QSKIP*, listando el caso de uso de acuerdo al caso.

6.1.7. Jerarquía de clases para modelos de prueba

Si bien la clase *UnitCube* cumple con su caso de uso actual, su implementación puede ser mejorada al separar ambas representaciones disponibles en dos clases independientes. Ambas heredando de una clase base la cual podría ser extendida para permitir la creación de otros modelos de prueba bajo una interfaz única.

6.1.8. Creación procedural de los modelos de esfera

Si bien los modelos de cascarones esféricos provistos fueron de aporte significativo en el desarrollo de este proyecto, su tamaño hace prohibitivo el almacenamiento de estos dentro del repositorio de *Camaron*, por otra parte los programas originales utilizados para su creación no fueron incluidos dentro del repositorio asociado, por lo que el poder replicar su generación requiere aplicar ingeniería inversa en base a la información descrita en los trabajos previos. En este sentido, se recomienda el crear un componente auxiliar de tests, que utilice la librería asociada a la herramienta TetGen [12] para crear estos modelos durante la ejecución de los tests.

La implementación de esta funcionalidad permitiría a desarrolladores posteriores el poder crear esferas sin la necesidad de descargar su contenido, permitiendo también la extensión de esta herramienta para la generación de nuevos modelos de prueba, así como la representación de estas esferas bajo otros formatos legibles por Camaron.

6.2. Documentación faltante

Debido a las limitaciones de tiempo y alcance de este proyecto, no se logró documentar de forma exhaustiva a los componentes que se listan a continuación:

Renderizadores En particular hace falta documentar el funcionamiento interno de cada opción presente, la implementación de sus shaders y las clases auxiliares presentes dentro del código asociado a este módulo.

Interfaz Idealmente se debería incluir cual clase corresponde a cual *Widget* dentro de la aplicación, así como su interacción con otros componentes del sistema.

CustomGLViewer Si bien trabajos previos mencionan a esta clase, su implementación interna es bastante compleja, por lo que su documentación es necesaria al ser un componente fundamental dentro de *Camaron*.

Utilidades Como se mencionó en la sección 2.3.8, el directorio *Utils* posee múltiples herramientas que afectan a mas de un componente del sistema. Debido al nivel de dependencia en estos sistemas, es posible que existan errores no documentados en esta área, por lo que se recomienda su análisis.

La documentación efectiva de estos componentes permite la implementación de tests sobre estos, así como su optimización en base al conocimiento obtenido. En el caso de tests, el caso de los renderizadores es especial, ya que el comparar el contenido de la pantalla no es recomendado debido a la facilidad de que estos tests fallen, por lo que se recomienda en este caso el testeo de la lógica asociada a estas clases en la medida de lo posible.

6.3. Falencias no corregidas

Las siguientes secciones buscan incluir información adicional sobre las fallas que no pudieron ser corregidas durante este trabajo. Estas se dividen en errores con o sin causa conocida. En el primer caso se puede entregar detalles sobre los métodos o clases involucradas así como recomendaciones de resolución. Para los casos sin determinar no se tiene información conclusiva suficiente para poder entregar asistencia, por lo que quedan como referencia a trabajos posteriores.

6.3.1. Causa conocida

6.3.1.1. Renderizadores

Visor de Intersección de objetos	Cargar segundo modelo	segfault	Lectura de geometria desconocida retorna puntero nulo

El visualizador afectado se encuentra implementado dentro de la clase *ConvexGeometryIntersectionRenderer*. En particular el método *parseNewGeometryString* utiliza la clase *ConvexGeometryParser* para obtener información de la geometría convexa que sera cargada. Estas geometrías convexas son implementadas bajo una jerarquía de herencia simple, con *GenericConvexGeometry* como la clase padre, implementando subclases para los diferentes tipos de geometría convexa disponibles (Planos, Esferas o Polígonos).

En el caso que el parser no logre determinar el tipo de geometría presente en un archivo, este retorna un puntero nulo. No siendo procesado de forma adecuada por el método *parseNewGeometryString* (Ver figura 6.1), resultando en la mal utilización de este puntero en el código posterior.

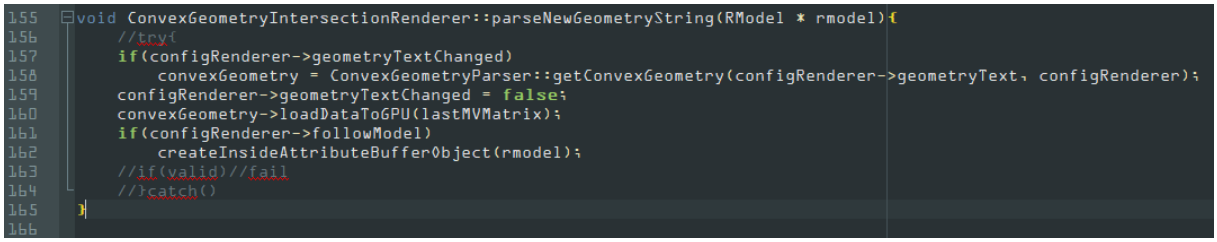


Figura 6.1: Localización de la falla dentro del código

Para la corrección de esta falencia se recomienda incluir una excepción en el caso de no identificar la geometría, para así poder notificar al usuario a través de un mensaje de error y permitir el funcionamiento posterior de la aplicación.

6.3.1.2. Estrategia de Selección

Expansión a vecinos	Aplicar estrategia a selección de poliedros basada en ángulo	Selección ocurre sobre elementos con ángulo mayor al escogido	Comparación implementada de forma incorrecta

Esta falla se encuentra dentro de la estrategia de *ExpandToNeighbors*, presente dentro del archivo *SelectionStrategies/ExpandToNeighbors/expandtoneighbors.cpp* linea 113 (Ver figura 6.2). Si bien este error puede ser corregido invirtiendo la comparación, durante el desarrollo se probó realizar esta corrección sobre el modelo de cubo unitario descrito por tetraedros, obteniendo el comportamiento esperado pero sobre valores negativos. Debido al tiempo acotado no se pudo asegurar la correctitud de estos valores, por lo que se recomienda estudiar este caso en mas detalle antes de realizar la corrección especifica.

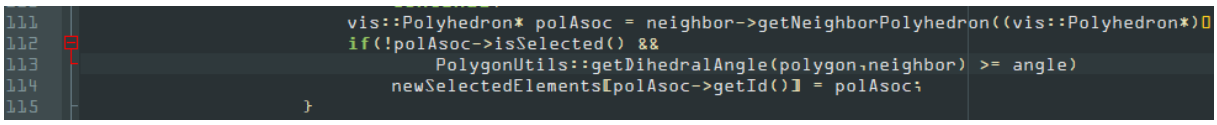


Figura 6.2: Localización de la falla dentro del código

6.3.1.3. Renderizadores

Este error en particular ocurre debido a que para caras no triangulares, el proceso de renderizado efectivamente dibuja cada triangulo por separado pero solo dibujando el contorno del

Visualizar Identificadores	Aplicar sobre malla de poliedro con caras no triangulares	Identificadores repetidos dentro de misma cara	IDs de poliedros son obtenidas en base a relaciones polígono-poliedro, los cuales no incluyen información referente a la triangulación del modelo

poliedro en base a la información disponible. En el caso del componente encargado de mostrar los identificadores sobre los poliedros (*UI/RenderElementsIds/renderelementsids.cpp*), este asigna el identificador de este ultimo sobre cada triángulo incluido dentro de cada cara, causando la repetición de identificadores ilustrada en la figura 6.3). Este error solo ocurre si la opción *Configuration->Main Preferences->Ids->Use GPU Acceleration* se encuentra habilitada (a través de CPU el error no ocurre).

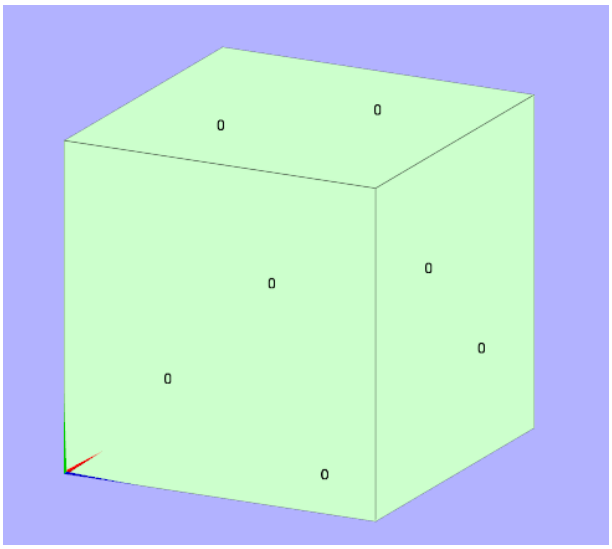


Figura 6.3: Ejemplo sobre modelo de prueba *ModelLoadingTest/data/VisF /polyhedron_mesh.visf*. En este caso solo un poliedro es definido para todo el modelo, por lo que su identificador es mostrado dos veces en cada cara

6.3.1.4. Utilidades

PolygonUtils	Carga de modelo con vértices que tengan solo una cara adyacente	Normales de estos vértices no son renderizadas	Algoritmo asigna flag para ignorar estos vértices de forma incorrecta

La falla pudo ser identificada dentro del archivo *Utils/PolygonUtils.cpp*, dentro de los métodos *getTriangleVertices* y *configRVertexFlagAttribute*. En particular durante la triangu-

lación de la cara, internamente se le asigna a cada vértice los flags *SURFACE_ VERTEX* y *VIRTUAL_ VERTEX*, a medida que se generan los triángulos interiores. En el caso de vértices que tengan un solo vecino, el algoritmo solo asigna el primero de estos atributos, por lo que a la hora de renderizar las normales, este renderizador descarta a estos vértices específicos al carecer esta flag (Ver figura 6.4).

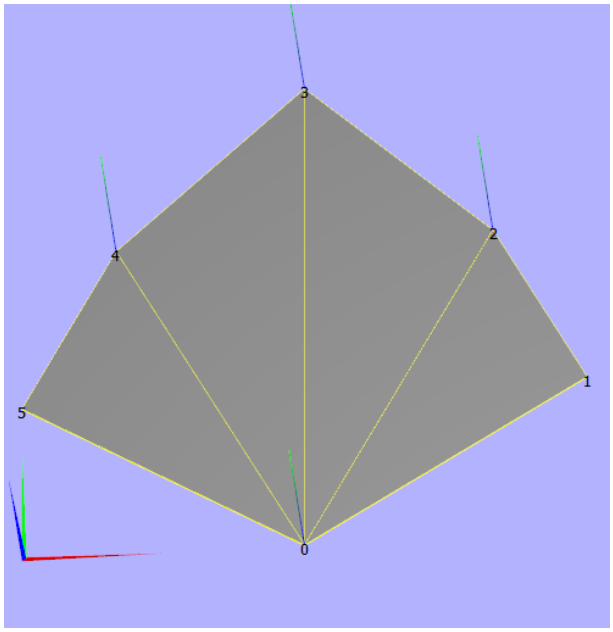


Figura 6.4: Ejemplo de normales no renderizadas en vértices 1 y 5, estos vértices poseen solo un polígono vecino y si incluyen una normal válida, solo que al no estar asignado el atributo especifico el renderizador no las gráfica.

Lamentablemente no se encontró documentación sobre la utilización de estas flags dentro del sistema y tampoco se pudo identificar la utilización de vértices "virtuales"dentro de la literatura asociada al algoritmo de triangulación. Debido a esto se decidió dejar el error pendiente ante la posibilidad de afectar a otros renderizadores, por lo que se recomienda lo siguiente:

1. Documentar el uso de estas flags dentro de *Camaron*
2. Crear test unitario para reproducir esta falencia
3. De acuerdo a lo determinado en el primer paso, reimplementar el algoritmo de triangulación, con objeto de corregir esta falla así como la simplificación de este método.

6.3.2. Causa Indeterminada

6.3.2.1. Interfaz Gráfica

Al habilitar la opción, el objeto desaparece del visor en su totalidad. El usuario debe realizar zoom de forma continua hasta un punto indeterminado, tras el cual el objeto aparece

Opciones de OpenGL	Habilitar *Perspective Projection*	Modelo solo es visible tras realizar zoom por varios segundos, renderizado resulta en artefactos visuales

de forma súbita, notar que en algunos casos se debe modificar el parámetro FOV aledaño para que el modelo aparezca. El modelo aparece con un tamaño fijo que no puede ser modificado con el uso de zoom, con algunas caras desapareciendo dependiendo del ángulo de la Camara. Si el usuario continua aplicando zoom, el renderizador escogido renderizará los elementos geométricos de forma inconsistente, con estos artefactos visuales aumentando a medida que se continua aplicando zoom (ver figura 6.5).

Figura 6.5: Ejemplo de modelo bajo proyección de perspectiva, el modelo fue renderizado utilizando la estrategia *Height Renderer*, la cual genera un degrade de color suave en condiciones normales.

Debido a la forma en el que el modelo aparece o desaparece de la cámara, el error podría ser causado por errores dentro de la definición del *viewport* de la cámara o de la operación de las matrices de transformación bajo esta opción. Sin embargo no se pudo determinar de forma concluyente el causante de esta falla.

Opciones de OpenGL	Habilitar y Desactivar *Face Culling CCW*	*Face Culling* sigue activo

Visualizar Identificadores	Renderizado en GPU	OpenGL Error: Uniform location not found
	Renderizado en GPU con transparencia habilitada	Identificadores reemplazados con rectángulos negros
	Renderizado en GPU con transparencia deshabilitada	Identificadores detrás de la superficie del modelo son renderizados
	Renderizado en CPU	Identificadores renderizados de forma inconsistente dependiendo de posición de la cámara

Los archivos responsables del renderizado de identificadores se encuentran disponibles dentro del directorio *UI/RenderElementsIds*. Estos errores no pudieron ser estudiados en mas detalle debido a la dependencia de estos sobre las opciones de OpenGL, la cual es una funcionalidad que escapa del alcance del proyecto.

El primer error es causado por la definición de un atributo específico dentro de los shaders asociados a esta, el cual no es utilizado durante la ejecución de la pipeline. Al compilar el shader la variable uniforme es eliminada como optimización, causando que al intentar cargar el atributo al shader, este no lo reconozca.

6.3.2.2. Exporte de Modelos

Formato Ele/Node	Exporte de mallas de poliedros	Poliedros escritos poseen orientación incorrecta
Formato OFF	Exporte de modelo cualquiera	Elementos escritos poseen orientación incorrecta

En el caso de Ele/Node, la diferencia en ordenamiento puede ser causada debido a que la estrategia no replica el procesamiento realizado por *HashTree* de forma inversa, por lo que el exportar e importar un mismo modelo resulta en normales inconsistentes. En el caso del formato OFF no se tiene información suficiente para determinar su causa.

6.3.2.3. Renderizadores

Visor de Intersección de objetos	Intersección de plano perpendicular sobre malla solida	Intersección inconsistente al incluir elementos no intersectados	No Identificado

Este error no pudo ser estudiado en mayor detalle al ser descubierto en etapas finales del proyecto. Esta falla ocurre solo si la intersección es definida en base a planos (*Normals*) (Ver figura 6.6), por lo que los componentes dentro del renderizador de intersecciones asociados a este podrían ser los causantes de esta falla.

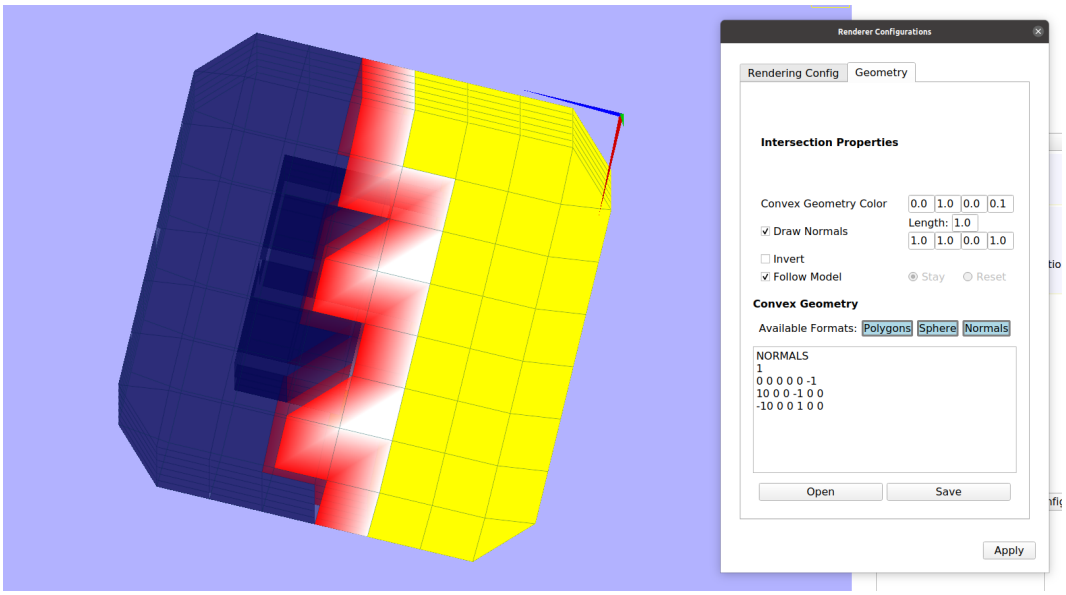


Figura 6.6: Ejemplo intersección afectada. El plano es definido de forma tal que la intersección con el modelo solo deba incluir la mitad derecha del modelo, pero el limite resultante es inconsistente

6.3.2.4. Estrategias de Selección

Selección por Mouse	Realizar selección sobre área con agujeros	Selección inconsistente
	Seleccionar área	OpenGL Error

En el caso de la selección con agujeros, la estrategia solo identifica a geometrías que posean normales orientadas a la cámara, por lo que esto puede resultar en la selección de objetos no deseados. Si bien el componente responsabl de esta funcionalidad se encuentra disponible dentro del directorio *SelectionStrategies/Mouse Selection*, la selección de objetos según la orientación de sus normales es una funcionalidad que se encuentra bajo revisión.

Capítulo 7

Conclusiones

Tras el término de este trabajo, se logró restituir una parte importante de las funcionalidades previamente existentes en *Camaron*. Por otro lado la aplicación redujo su consumo de memoria en un 18 % y mejoró su velocidad de procesamiento en un 75 %.

Al mismo tiempo la versión mas reciente del proyecto se encuentra en un estado mas robusto, logrando la corrección de buena parte de las fallas detectadas, quedando solo un caso de error fatal dentro del sistema, por otro lado se logro la corrección de las instancias de *undefined behavior* detectadas, permitiendo recuperar la portabilidad de la aplicación original. La lista de bugs presentado por este documento busca guiar los esfuerzos de refactoring que provengan de desarrolladores futuros.

Los cambios realizados en la representación de los modelos 3D, así como en los elementos que lo componen, permitieron reducir la complejidad de una parte importante del proyecto, logrando desacoplar funcionalidad entre componentes claves, así como permitir la implementación de las optimizaciones de memoria de este trabajo. Estos cambios tambien permitieron simplificar algunos de los procesos internos, permitiendo mejorar la calidad del código fuente al reducir las instancias de código duplicado necesario para poder implementar ciertas abstracciones.

La documentación técnica incluida en este documento y en el código fuente, más las pruebas unitarias y de stress escritos también entregan un aporte importante para desarrolladores futuros de este proyecto, asegurando el funcionamiento correcto de un parte importante del sistema y reduciendo la posibilidad de introducir regresiones en cambios posteriores.

Sin embargo, la complejidad del sistema evidenciada durante la fase de análisis supero el tiempo presupuestado para poder abarcar la totalidad de este, lo que resulto en la reducción del alcance del proyecto. Si bien se logro progresar en los objetivos del proyecto en las áreas estudiadas, los componentes que quedaron fuera de este alcance no pudieron ser documentados ni modificados apropiadamente.

7.1. Trabajo Futuro

A continuación se listarán una serie de mejoras que podrían ser implementadas a largo plazo en Camaron, con el objetivo de mejorar su performance y usabilidad en trabajos futuros.

7.1.1. Desacoplamiento entre relaciones de vecindad y elementos

Actualmente cada elemento dentro del modelo almacena las relaciones de vecindad asociadas a este, resultando en el almacenamiento de la misma información entre elementos vecinos así como un *overhead* de memoria asociado al almacenamiento inicial de cada vector creado en memoria.

En base a esto se recomienda el trasladar estas relaciones a un componente externo que almacene las relaciones de vecindad del modelo completo. Esto permitiría el poder aplicar optimizaciones de memoria o performance sobre el sistema completo, sin requerir la refactorización de cada clase de tipo *Element*. Por otra parte este cambio mejoraría la mantenibilidad del software, al abstraer estas relaciones en un componente central dentro del código, permitiendo realizar cambios estructurales mayores sobre la representación interna del modelo sin afectar a otros componentes de *Camaron*.

7.1.2. Indexación implícita

Tras los cambios realizados en la sección 4.1.2, se hace factible el reemplazo de algunas referencias por su posición dentro del contenedor respectivo. Esto permitiría una reducción de memoria importante en el caso de las relaciones de vecindad, reduciendo el consumo a la mitad al utilizar índices en vez de puntero.

Este cambio también resultaría en una simplificación dentro de la construcción de elementos geométricos, ya que podría asignarse los índices correspondientes a sus miembros o vecinos sin la necesidad de construir estos elementos para poder obtener una referencia válida.

7.1.3. Implementación de corrección de normales como operación disponible post carga

Si bien el proceso de reingenierá logró corregir la lectura de normales incorrecta para ciertos formatos, existe la posibilidad de que un modelo ingresado por el usuario posea orientaciones erróneas debido a errores externos a *Camaron*. Si bien en este sentido *Camaron* cumple con la capacidad de informar al usuario de esta falencia a través del renderizador de normales, este no permite corregir la orientación de estas dentro de la aplicación.

La corrección de estas normales ya se encuentra parcialmente implementada en *Camaron* (véase sección 3.2.2.2), por lo que existe la posibilidad de reintegrar este componente al sistema como una operación realizable tras la carga del modelo. Este cambio requerirá de

la creación de un nuevo componente que permita realizar cambios tangibles al modelo, así como un *Widget* dentro de la interfaz para su uso.

7.1.4. Paralelización de etapas de procesamiento secuenciales

Si bien la mayoría de las etapas listadas en la sección 2.3.6.1 son realizadas de forma paralela, el almacenamiento de relaciones de vecindad de tipo polígono->vértice y poliedro->polígono son realizadas de forma secuencial. Esto debido a que estas operaciones realizan modificaciones sobre elementos que pueden ser trabajados por otros *threads*, por lo que para lograr su paralelización se debe implementar medidas para corregir estos problemas de concurrencia (*thread-safety*).

Al mismo tiempo, otro bottleneck importante de tiempo a considerar corresponde al proceso de triangulación de modelos, el cual es actualmente ejecutado por la clase *RModel*. Este algoritmo es actualmente implementado por el método *PolygonUtils::getTriangleVertices*, y se aplica sobre cada polígono del modelo de forma secuencial, el poder asegurar la seguridad de concurrencia en la clase *Model* permitiría mejorar la velocidad de ejecución en este punto en particular.

7.1.5. Manipulación de Cámara

Actualmente la posición y orientación de la cámara se encuentra almacenada dentro de la clase *RModel*, permitiendo la manipulación de esta bajo la interfaz actualmente disponible. Si bien esta funcionalidad no presenta problemas, si agrega complejidad a *RModel* que podría ser delegada a un objeto externo. En particular otros objetos que requieren de esta información como *MouseSelection* almacenan copias de esta información, por lo que mantener el estado de la cámara en un solo lugar permitiría reducir el acoplamiento entre estos componentes.

Por otro lado, la implementación actual solo contempla la rotación y zoom sobre un punto central dentro del modelo. Podría extenderse la manipulación de la cámara para permitir traslaciones, así como el enfoque sobre un punto especifico del modelo, lo cual mejoraría la usabilidad del proyecto.

7.1.6. Especificación de tipo de formato

Como se pudo constatar en la sección 3.2.2.1, la carga de archivos ele/node ilustra la necesidad de poder distinguir entre diferentes implementaciones para una misma especificación (*TetGen* vs *TRIANGLE*). Si bien actualmente es posible ignorar esta dsicrepancia con la interfaz existente, se recomienda el abordar esta complejidad extra de forma apropiada, ya que el problema puede resurgir al implementar otros formatos externos en el futuro.

Una implementación correcta de este problema sería un punto de partida para una refactorización del modulo de exporte de modelos, ya que se podría implementar un exporte

separado que pudiera ser escogido por el usuario a través de la interfaz, en vez de asumir una implementación especifica como lo es con el caso de ele/node.

7.1.7. Selección de vértices individuales

Actualmente este módulo posee funcionalidad para vértices que se encuentra deshabilitada en la interfaz. Esto debido a que al seleccionar múltiples puntos, la coloración resultante del grupo no es la adecuada, debido a que el color de cada triangulo se hace en base a interpolar los colores de los vértices que lo componen. Si se seleccionan todos los vértices de un polígono, esto produce la coloración total del este, mientras que en el caso de existir vértices fuera de la selección, el color del polígono resultará de la interpolación entre los valores de cada vértice de este (dentro y fuera de la selección).

Bibliografía

[1] Schäfer, Anja, Hubert Mara, Bernd Breuckmann, Christiane Düffort y Georg Bock: *Large Scale Angkor Style Reliefs; High Definition 3D Acquisition and Improved Visualization using Local Feature Estimation.* En *Proc. of 39th Annual Conference of Computer Applications and Quantitative Methods in Archaeology*, January 2011.

[2] Hubert Mara, Susanne Krömker, Stefan Jakob: *GigaMesh and Gilgamesh - 3D Multiscale Integral Invariant Cuneiform Character Extraction.* En *VAST 2010: The 11th International Symposium on Virtual Reality, Archaeology and Intelligent Cultural Heritage*, January 2010.

[3] Canepa, Aldo: *Camaron: Visualizador y evaluador de mallas geometricas mixtas grandes en 3D, acelerando con shaders en OpenGL. Memoria para optar al titulo de ingeniero civil en computación.* Universidad de Chile, Departamento de Ciencias de la Computación, 2012.

[4] Infante, Gonzalo: *Mejoramiento del software Camaron de Visualizacion de Mallas 3D e inclusion de Visualización CientificaL. Memoria para optar al titulo de ingeniero civil en computación.* Universidad de Chile, Departamento de Ciencias de la Computación, 2016.

[5] Prabhu Ramachandran, Gaël Varoquaux: *Mayavi: a package for 3D visualization of scientific data.* En *Computing in Science & Engineering*, March 2011.

[6] *VisIt, Visualization, animation and analysis tool.* `https://wci.llnl.gov/simulation/computer-codes/visit`.

[7] James Ahrens, Berk Geveci, Charles Law: *ParaView: An End-User Tool for Large Data Visualization.* 2015.

[8] Mara, Hubert, Susanne Krömker, Stefan Jakob y Bernd Breuckmann: *GigaMesh and Gilgamesh 3D Multiscale Integral Invariant Cuneiform Character Extraction.* En Artusi, Alessandro, Morwena Joly, Genevieve Lucet, Denis Pitzalis y Alejandro Ribes (editores): *VAST: International Symposium on Virtual Reality, Archaeology and Intelligent Cultural Heritage.* The Eurographics Association, 2010, ISBN 978-3-905674-29-3.

[9] *IEEE Floating Point Arithmetic Standard.* `https://ieeexplore.ieee.org/document/8766229`.

[10] *C++11 Specification.* `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3337.pdf`.

[11] *TetGen, Delaunay Tetrahedral Mesh Generator.* http://wias-berlin.de/software/tetgen.

[12] *Manual de uso TetGen.* https://wias-berlin.de/software/tetgen/1.5/doc/manual/manual.pdf.

ANEXOS

Anexo A

Detalle soporte de formatos 3D

El siguiente anexo busca entregar una visión general de los diferentes formatos disponibles por Camaron, detallando las secciones que no se encuentran disponibles actualmente, así como los cambios realizados a algunos de los formatos listados en caso de extensiones propias.

A.1. Formatos externos

A.1.1. PLY

La especificacion completa de este formato se puede encontrar en el siguiente link (`http://paulbourke.net/dataformats/ply/`). *Camaron* debe ser capaz de leer archivos de este tipo para los tres tipos de codificación disponibles (ASCII, *little endian* y *big endian*), permitiendo leer vértices y polígonos descritos en la especificación.

Camaron permite la definición de ejes adicionales, estos son declarados en el header a traves del keyword *edge*, el cual es escrito entre las declaraciones de vértices y polígonos. El contenido de estos ejes se encuentra escrito posterior a los datos de los polígonos escritos.

```
ply
format ascii 1.0
element vertex ..
<propiedades de vertices>
element edge ..
element face ..
property list ....
---------
Contenido Vértices

---------
Contenido Polígonos
```

```
---------
<índice vértice 1> <índice vértice 2> <color R> <color G> <color B>
.....
```

Debido a la flexibilidad del formato, algunos formatos dentro de *Camaron* no pueden ser leídos de forma correcta, en particular la lectura de archivos con propiedades de polígonos no son identificadas por el sistema, por lo que la lectura de los datos es causal de errores. Sin embargo, el lector debería ser capaz de recuperarse ante la lectura de estos archivos externos para preservar la usabilidad del software.

A.1.2. Ele/Node

Este formato se compone de dos archivos con diferente extensión, donde el contenido de estos dependerá de la especificación utilizada. Estas especificaciones son definidas por la herramienta que generó los modelos inicialmente. En particular Camaron entrega soporte a dos herramientas de generación de mallas, TRIANGLE (`http://wias-berlin.de/software/tetgen/1.5/doc/manual/manual006.html`) y TetGen [12], enfocadas respectivamente en mallas de triángulos planas y mallas de poliedros respectivamente.

En el caso del archivo *.node*, ambas especificaciones definen los vértices que componen al modelo de la misma forma. Con cada línea almacenando la id de cada punto, sus coordenadas, atributos asociados y un valor booleano final definido como *boundary marker*. *Camaron* actualmente utiliza los dos primeros parámetros durante la carga inicial de modelos, mientras que los atributos pueden ser cargados de forma posterior a través del componente de atributos, con el ultimo elemento actualmente siendo ignorado.

Es importante notar que en el caso de TRIANGLE, el índice del vértice inicial puede iniciar con 0 o 1, por lo que su identificador puede ser diferente al posicionamiento de este elemento dentro del modelo.

TRIANGLE utiliza el archivo *.ele* para almacenar una lista de polígonos, con un formato similar al mencionado anterior, reemplazando los valores de coordenadas por los índices de cada vértice que constituye a cada triangulo y eliminando el valor de *boundary marker*.

TetGen por otro lado utiliza este archivo para almacenar poliedros, utilizando cuatro índices para representar cada tetraedro individual. Debido a que *Camaron* debe crear polígonos para definir poliedros, este tipo de archivo debe pasar por una etapa de procesamiento extra, para determinar la cantidad de polígonos mínimos necesarios para representar al modelo. Al mismo tiempo el formato posee un ordenamiento especifico que debe ser considerado para poder evitar la presencia de normales invertidas del modelo.

Finalmente, estas herramientas también utilizan extensiones complementarias (.poly, .face, .neigh, entre otros), los cuales no son utilizadas por *Camaron* al escapar de su alcance actual.

A.1.3. OFF

El formato es descrito en base la siguiente especificación (`https://people.sc.fsu.edu/~jburkardt/data/off/off.html`). En lo que concierne a *Camaron* solo se leen los datos asociados a los vértices y polígonos de la malla, ignorando información asociada al color de estos y al número de aristas de la malla.

```
OFF
<Numero de vértices> <Numero de poligono> <Numero de aristas>
<coord x> <coord y> <coord z>
...

<Numero de Triangulos>
<id vértice 0> <id vértice 1> <id vértice 2> <color R> <G> <B> <A>
...
```

A.2. Formatos propios

A.2.1. TRI

Corresponde a una variante de el formato *OFF*, la cual contiene exclusivamente triángulos. A diferencia del formato origen, el numero de elementos se encuentra escrito previo a los datos de los elementos, por lo que este carece de un header inicial.

```
<Numero de vértices>
<coord x> <coord y> <coord z>
...

<Numero de Triangulos>
<id vértice 0> <id vértice 1> <id vértice 2>
...
```

A.2.2. VisF

Este formato fue diseñado durante el trabajo realizado por Aldo Canepa ([3], Sección 4.5). Este formato es capaz de describir los tres tipos de modelos anteriormente implementados (Nube de vertices, Malla de Poligonos y Malla de Poliedros), incluyendo información de cada elemento mas las relaciones de vecindad entre polígonos de la malla. En el caso de una malla de poliedros, el formato describe la representación de los polígonos y poliedros, en contraste

con los formatos Ele/Node y M3D, los cuales no incluyen información sobre los polígonos del modelo.

Este formato pasó por múltiples revisiones, resultando en modificaciones presentes durante el estado inicial del proyecto, las cuales codifican la siguiente información:

Endianness del archivo Representado por un byte inicial donde 0 corresponde a codificación *big endian* y 1 a *little endian*

Presencia de relaciones de vecindad polígono-polígono Representado por un byte localizado entre las secciones de datos de polígonos y poliedros, siendo 0 en caso de no presentar estas relaciones de vecindad y 1 en caso contrario.

Durante el transcurso del proyecto se incluyó la capacidad de leer archivos ASCII, lo cual fue implementado a través de la extensión del byte de endianness, siendo 2 el valor para representar contenido en esta codificación. Notar que para poder interpretar correctamente este byte entre diferentes codificaciones, se modificó el formato para que este byte siempre se encuentre escrito como texto ASCII. En base a esta modificación el formato VisF posee la siguiente especificación actualizada:

```
<Endianness (ASCII char)> <Tipo de malla (int)>
<Numero de vértices>
<coord x> <coord y> <coord z>
.....
<Numero de polígonos>
<Numero de vértices de polígono 0> <id vértice 0> <id vértice 1> ...
....
<Presencia de relaciones de vecindad (char)>
<Numero de vecinos polígono 0> <id polígono vecino 0> <id polígono vecino 2> ....
....
<Numero de poliedros>
<Numero de caras> <id cara 1> <id cara 2> ....
....
```

A.2.3. M3D

Este formato se encuentra descrito dentro del trabajo realizado por Aldo Canepa ([3], Sección A.4). Este formato es utilizado exclusivamente para representar mallas de poliedros, compuestas como una serie de hexaedros, tetraedros, pirámides y prismas, y esta descrito de la siguiente forma:

```
[Nodes, ARRAY1<STRING>]
<Numero de vértices>
```

```
<Tipo de vértice (int)> <x> <y> <z> <proy x> <proy y> <proy z>
.....

[Elements, ARRAY1<STRING>]
<Numero de poliedros>

<Tipo de poliedro (char)> <id vértice 0> <id vértice 1> ...
.....
```

Lamentablemente la especificación presentada no es lo suficiente detallada para poder entender el comportamiento esperado asociado a los vértices de proyección, en particular los archivos provistos incluían valores para el tipo de vértices iguales a 1, 2 o 3, sin embargo el lector implementado utiliza el contenido de estos solo si la id presente es 2.

Por otro lado el formato no especifica la forma en que cada componente debe ser leído, resultando en modelos con normales orientadas de forma inconsistente durante la carga. Lamentablemente tampoco se tiene información sobre el ordenamiento específico de cada cara por lo que se escogió un tipo de lectura que permitiera la lectura apropiada de los modelos de prueba disponibles.