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

MEJORAMIENTO DEL SOFTWARE CAMARÓN DE VISUALIZACIÓN DE MALLAS 3D E INCLUSIÓN DE VISUALIZACIÓN CIENTÍFICA

MEMORIA PARA OPTAR AL TÍTULO DE INGENIERO CIVIL EN COMPUTACIÓN

GONZALO FRANCISCO INFANTE LOMBARDO

PROFESOR GUÍA:
NANCY HITSCHFELD KAHLER

MIEMBROS DE LA COMISIÓN:
BENJAMÍN BUSTOS CÁRDENAS.
TOMÁS BARROS ARANCIBIA

SANTIAGO DE CHILE
2016

Resumen

La visualización científica es un área que apoya el análisis de datos generados por simulaciones de fenómenos físicos e ingenieriles, entre otros. Junto a los datos geometrícos del modelo que describen el dominio se requiere visualizar datos escalares y vectoriales asociados a los vértices, arcos y/o caras del modelo. Existen diversas técnicas para visualizar dichos campos tales como coloreado superficial, volume rendering, isolíneas, isosuperficies y otros.

Camarón[8] es una herramienta gráfica de estudio de mallas mixtas que permite conocer propiedades de los elementos que las constituyen. Para hacer más fácil el estudio cuenta con coloreado de vértices, polígonos y poliedros según el valor del criterio de evaluación a elección, además de presentar un histograma con la propiedad en estudio. Camarón además cuenta con múltiples renderers, métodos de selección de elementos, criterios de evaluación y archivos de entrada/salida junto con una arquitectura que lo hace fácilmente extensible. Ocupa la GPU de una máquina en forma activa para permitir una interacción fluida. Las GPU son unidades de procesamiento altamente paralelo que permiten paralelizar la mayoría de los algoritmos de visualización científica. Existen varias herramientas desarrolladas para aprovechar los recursos de una GPU tales como CUDA, OpenCl o OpenGL. La última es usada principalmente para fines gráficos.

El objetivo de esta memoria consitió en introducir la visualización de campos escalares en Camarón principalmente con el uso de isolíneas para modelos superficiales e isosuperficies para volumétricos. La implementación ocupa el lenguaje C++ y aprovecha las capacidades de procesamiento de una GPU moderna con el uso de la biblioteca OpenGL. La arquitectura implementada es extensible y permite el manejo de valores escalares y vectoriales asociados a los vértices de una malla.

Los algoritmos de generación de isosuperficies e isolíneas fueron implementados ocupando las capacidades de paralelismo de una GPU. Más aún el uso de la funcionalidad Transform Feedback en OpenGL permitió que la interacción con la malla fuese fluída. Se mejoró

Dado que existen varios visualizadores en 3D para apoyar el análisis de datos científicos, se comparó el software Camarón con uno de los más usados *ParaView*. Los resultados muestran un desempeño entre 4 y 6 veces mejor para Camarón en términos de rapidez, aunque con sobrecostos en uso de memoria de hasta 6 veces lo usado por ParaView.

A mis familia y amigos

Agradecimientos

Quiero agradecer a mi familia por su apoyo incondicional durante todos estos años en todos los aspectos de mi vida.

A los amigos que he tenido a lo largo de la carrera desde mechón por su compañía durante los primero años universitarios. Un especial agradecimiento al coro de la facultad del que fui parte 4 años y con los que compartí grandes experiencias.

Al Departamento de Computación y TODA su gente. Sin embargo quiero darle un especial y eterno agradecimiento a mis compañeros de computación, con los que he pasado muy buenos y graciosos momentos. Siempre recordaré la buena acogida que hay en la salita como los grandes personajes que se pueden encontrar en ella.

Tabla de Contenido

Índice de Tablas

Índice de Ilustraciones

Capítulo 1

Introducción

La digitalización y análisis de objetos tridimensionales ha sido desde hace más de 30 años un área de investigación en desarrollo con múltiples aplicaciones que van desde la geografía hasta el cine. Hoy en día es un campo de investigación muy activo con cientos de investigadores dedicados a mejorar la forma como computadores resuelven problemas ligados al tratamiento y visualización de mallas geométricas. Una malla en 2D está compuesta generalmente de triángulos o cuadriláteros y, en 3D, de polígonos como triángulos; o poliedros como tetraedros o hexaedros. Aspectos importantes para la industria de videojuegos o el cine es la eficiencia de almacenamiento, su renderizado rápido en pantalla y la calidad de una malla. Es así como aplicaciones usadas para crear, transformar y renderizar modelos en 3D han evolucionado para también satisfacer las necesidades de laboratorios científicos y tecnológicos. A raíz de esto se publica una gran cantidad de artículos al año desarrollando nuevas técnicas o mejorando algunas existentes para tratar problemas asociados a la discretización de objetos.

Una rama de interés se relaciona con medir la calidad de una malla a partir de varios criterios o métricas geométricas, las cuales entregan información acerca de la calidad de la malla. En base a esto es posible determinar qué zonas de un modelo cumplen con los criterios de calidad deseados para su uso posterior. Por ejemplo, simulaciones numéricas usando el método de elementos finitos requiere de mallas sin ángulos muy pequeños.

Un área de interés relacionada es la que se enmarca en la visualización científica, que es la que intenta llevar datos a una representación visual permitiendo así entender fenómenos físicos o de otra índole. Este tipo de visualización es imperiosa en algunas áreas de investigación científica y la ingeniería, pues la interpretación de datos numéricos no es fácil para modelos complejos.

1.1. Motivación

En 2012 se creó en el Departamento de computación de la FCFM el software *Camarón*[6] como el trabajo de título del alumno Aldo Canepa. El objetivo principal de Camarón es permitir a un investigador asistir una simulación científica en una malla geométrica de forma

tal que se eviten errores por defectos en la malla. Para ello Camarón cuenta con un conjunto de herramientas que de manera gráfica le permiten entender a un investigador cuáles son las zonas que podrían causar problemas en una simulación. Estas herramientas analizan una malla con distintos criterios, escogidos por el usuario final y gracias a ello se pueden tomar decisiones sobre qué secciones deben ser corregidas. Entre los criterios disponibles están el área de los polígonos, la relación de aspecto, ángulo diedro y volumen de poliedros entre otros.

Un aspecto importante de Camarón es que su rendimiento es competente frente a otro software del área como TetView o Meshlab, logrando una alta tasa de cuadros por segundo incluso para modelos de alta complejidad. Esto es posible gracias al importante trabajo enfocado en el uso de las capacidades de una unidad de procesamiento gráfico o GPU, lo que permite visualizar a 60 fps[1] modelos volumétricos del orden de los 2 millones de polígonos en un computador de escritorio.

Las tecnologías usadas por Camarón son:

- C++: Lenguaje de desarrollo frecuentemente usado en la industria para desarrollar aplicaciones de alto rendimiento. Es debido a este aspecto que este lenguaje fue escogido para desarrollar Camarón.
- QT: Esta biblioteca permite a un desarrollador crear interfaces de usuario para aplicaciones generalmente desarrolladas en C++. Cuenta con soporte para Linux, Mac OS y Windows.
- OpenGL: Biblioteca de gráficos para que un programador pueda ocupar las capacidades de una tarjeta gráfica usando una interfaz estándar.

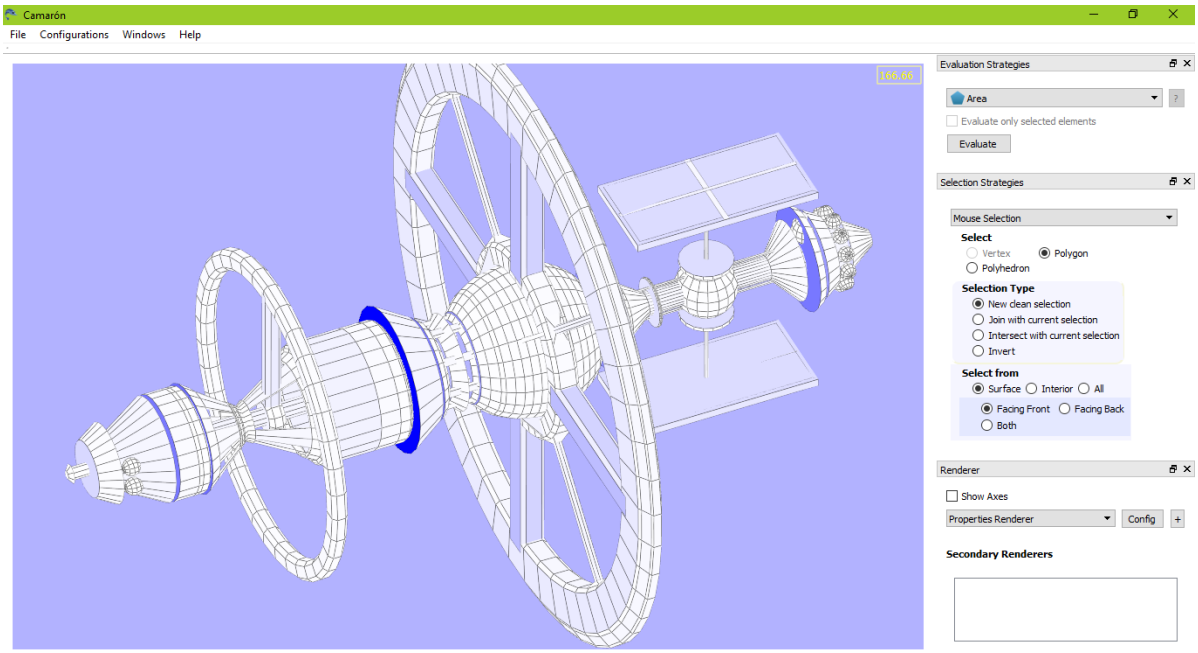

Figura 1.1: Ventana principal de Camarón.

[1]Frames per second

1.2. Objetivos

- Objetivo general:
  - Agregar técnicas de visualización científica al visualizador Camarón.
- Objetivos específicos:
  - Agregar una arquitectura de clases que permita sustentar la adición de campos escalares, vectoriales, etc.
  - Agregar la visualización de isosuperficies e isolíneas usando la GPU.
  - Permitir la carga de campos de propiedades desde memoria secundaria. Ya sea desde el archivo de descripción de geometrías o un archivo adjunto.
  - Comparar eficiencia y funcionalidad con otros visualizadores.
- Objetivos alternativos:
  - Introducir en el software mejores estándares de calidad de código.

1.3. Metodología

Durante el desarrollo de esta memoria se realizaron reuniones semanales con la profesora guía, por lo general, luego de agregar características nuevas a Camarón. De esta forma se hicieron revisiones, intercambios de ideas y las modificaciones pertinentes al software de manera iterativa e incremental.

Las nuevas funcionalidades fueron incorporadas usando patrones de diseño, siguiendo el estilo actual de Camarón, excepto cuando afectan la eficiencia de éste.

1.3.1. Entorno de Trabajo

El desarrollo de Camarón toma lugar en un computador personal con Windows 10 de 64 bits. Para la compilación de C++ se usa el compilador g++ del entorno MinGW. Se uso la versión 4.9.2 de MinGW de 64 bits con sistema de excepciones `seh` y threads estilo `posix`.

Como IDE de desarrollo se usó Qt Creator en su versión 3.4.0 y para ejecución de la aplicación se usó Qt 5.4.1 compilado en MinGW 4.9.2 de 64 bits.

1.3.2. Git

El trabajo actual de Camarón se encuentra en un repositorio Git del sitio SourceForge [2]. Existe sólo un branch principal llamado `master` en el cual se han hecho todos los cambios del software. Para trabajar en el código se clonó el repositorio al computador de trabajo y

[2] http://sourceforge.net/p/camaron/code/ci/master/tree/

se creó una rama de desarrollo llamada `scientific-vis`. El desarrollo se encuentra disponible en un fork del repositorio original (https://sourceforge.net/u/gloix/camaron/ci/scientific-vis/tree/).

1.4. Contenido de la Memoria

En el capítulo de *antecedentes* se dan a conocer los conceptos previos para entender el resto del trabajo. Esto incluye bibliotecas gráficas, lenguajes de programación y visualización científica.

En el capítulo de *diseño y análisis* se introduce Camarón junto a su funcionamiento y se definen los problemas que deben ser resueltos con la especificación de casos de uso. Luego se itera sobre las áreas del software que deben trabajarse y se estudian soluciones a cada una de ellas.

En el capítulo de *implementación* se describen en mayor detalle los algoritmos implementados para resolver problemas geométricos junto con introducir algunos cambios hechos para mejorar la calidad de código.

En el capítulo de *resultados* se muestran mediciones de rendimiento de Camarón y se contrastan con mediciones del software ParaView de visualización científica. Se toman en cuenta resultados de tiempo de procesamiento, como también de uso de memoria principal y de video.

El presente informe finaliza con un capítulo de *conclusiones* que dan a conocer un análisis de los resultados obtenidos junto a reflexiones sobre los mismos.

Capítulo 2

Antecedentes

El trabajo realizado requiere de conocimientos previos relativos a cómo se dibujan objetos usando las capacidades de un chip gráfico convencional y las tecnologías que son usadas en el software Camarón. Ellos son enunciados en esta sección y se recomienda internalizarlos al menos en un nivel básico para adentrarse en el resto del trabajo.

2.1. Visualización científica

Tanto en las áreas de la ciencia como en la ingeniería es necesario visualizar ciertas propiedades de los modelos con los que se trabaja. Es en esta área que la computación gráfica[10] juega un rol importante y se construyen herramientas para permitir entender datos numéricos en forma de una representación visual.

En el área de las simulaciones científicas se encuentran los métodos de elemento finito con los cuales se intenta simular algún fenómeno que ocurre en una malla de polígonos o poliedros (modelo). Este tipo de simulaciones asignan valores a puntos(vértices) de una malla para luego obtener de acuerdo a las reglas de la simulación un modelo con propiedades nuevas. De este modelo se pueden hacer análisis cualitativos en un visualizador de mallas como Camarón.

2.1.1. Campos escalares y vectoriales

Las mediciones de propiedades hechas en una región del espacio pueden ser tanto escalares o vectoriales. Los campos escalares, tal como su nombre lo dice, son un conjunto de puntos en el espacio a los que se asigna un valor escalar y por lo general representan una medición de una propiedad escalar tal como presión, temperatura o altura. Por otro lado, los campos vectoriales asignan vectores a un conjunto de puntos de una región en el espacio, dando así un listado de valores a cada punto. Ejemplos de campos vectoriales son la velocidad de un fluído o el gradiente de dispersión de calor en un objeto.

Estos campos poseen varias formas de representación en una imagen, ya sea usando degradados con colores, símbolos u otros. Las distintas técnicas de representación han sido implementadas como algoritmos para que computadores puedan emularlas usando los recursos que tienen a disposición. Por lo general se intenta llevar estos algoritmos de visualización a una unidad de procesamiento altamente paralelo como lo es una GPU, por ser la mayoría de estos algoritmos inherentemente paralelizables.

2.1.2. Isolíneas e isosuperficies

La visualización de isosuperficies constituye una herramienta útil para representar de manera gráfica las regiones en las que una cierta propiedad se mantiene constante e igual a un valor dado llamado *isonivel* o *equipotencial*. Estas formas de representación sólo están disponibles para campos escalares, y permiten un mejor entendimiento de algunos fenómenos. En el caso de los modelos dados por una superficie las equipotenciales toman la forma de curvas cerradas llamadas *isolíneas*, mientras que para el caso de modelos con un volumen asociado las equipotenciales toman la forma de superficies llamadas *isosuperficies*[16].

Otra forma de entender las isolíneas e isosuperficies es verlos como la región del espacio que actúa de barrera o interfaz entre los valores que son mayores a un valor dado y los que son menores.

La representación con el uso de isolíneas e isosuperficies ayuda a una persona a visualizar de mejor manera el comportamiento de un campo escalar cuando la representación de la propiedad está dada en colores o escala de grises y no ayuda a verlo de manera clara. Las pequeñas fluctuaciones en el campo pueden no ser fácilmente visibles por las leves diferencias en el cambio de color o cambio de intensidad (ver fig. 2.1).

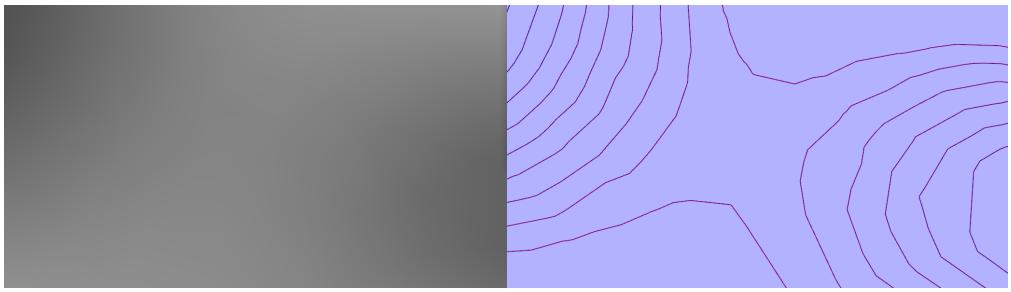

Figura 2.1: Ejemplo de utilidad de la representación con isolíneas. La imagen izquierda es un zoom a la sección de un modelo con un campo escalar donde se pierde la forma que toma el campo. Queda más clara la morfología en la imagen derecha cuando se marcan isolíneas en un intervalo regular.

Uno de los usos más comunes y conocidos de la visualización de isolíneas es en mapas geográficos. En ellos se muestra con curvas los límites de secciones horizontales de terreno en los cuales la elevación del terreno es constante (fig. 2.2).

Por su parte, las isosuperficies son usadas para representar fenómenos en simulaciones de volúmenes. La convección de Raleigh-Bernard que intenta describir el comportamiento de

fluídos en presencia de una fuente de calor puede ser emulada de forma volumétrica para luego visualizar los patrones formados en 3 dimensiones (fig. 2.3).

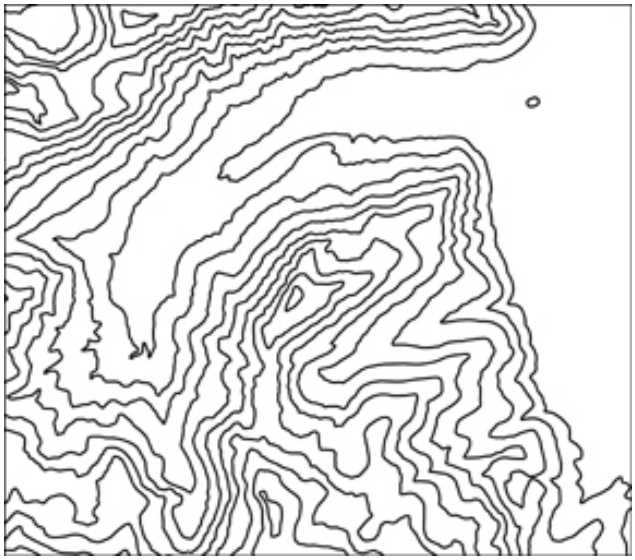

Figura 2.2: Ejemplo de isolíneas: isolíneas de altura en un terreno[1].

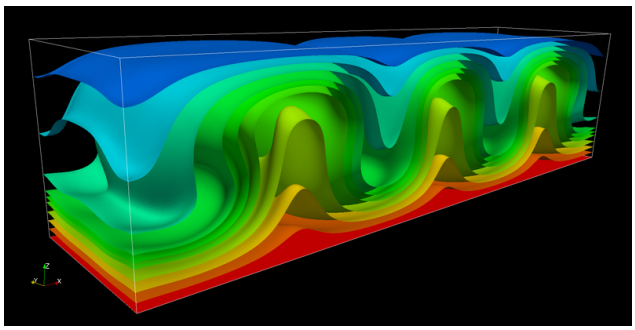

Figura 2.3: Ejemplo de isosuperficies: Simulación del problema de Raleigh-Bernard usando EdgeCFD[2].

2.1.3. Visualizadores de datos científicos

La visualización de mallas y visualización científica viene apoyada de software desarrollado desde hace varios años. Existen diversas alternativas tanto comerciales como también libres, pero en este trabajo sólo se considerará la comparación con aplicaciones consideradas como software libre y en desarrollo. Una biblioteca popular usado para la construcción de aplicaciones de visualización científica es VTK (Visualization Toolkit), la cual que provee un conjunto de herramientas para el desarrollo de software de visualización 3D y procesamiento de imágenes. Entre las aplicaciones más populares del área que usan VTK se encuentran:

[1]Fuente: `https://www.e-education.psu.edu/geog486/node/1873`. Visitado: 2016-05-02
[2]Fuente: `http://www.paraview.org/Wiki/ParaView_In_Action`. Visitado: 2016-05-02

MayaVi
: Software escrito en Python que provee un conjunto de herramientas estándar que incluye visualización de campos vectoriales o escalares.

VisIt
: Permite la visualización de datos a gran escala (del orden de terabytes).

ParaView
: Se especializa en ofrecer la posibilidad de hacer el procesamiento de visualizaciones en un cluster de computadores para luego ver el resultado en un terminal.

La manera de trabajar del software basado en VTK es tal que se cuenta con un pipeline (ver figura 2.4) con el cual se procesa un modelo hasta llegar a ser renderizado en pantalla. Quienes juegan un rol interesante son los filtros(2da etapa), y es en donde se ubican herramientas que pueden ser usadas para convertir un modelo con propiedades escalares asociadas a un conjunto de isosuperficies o isolíneas.

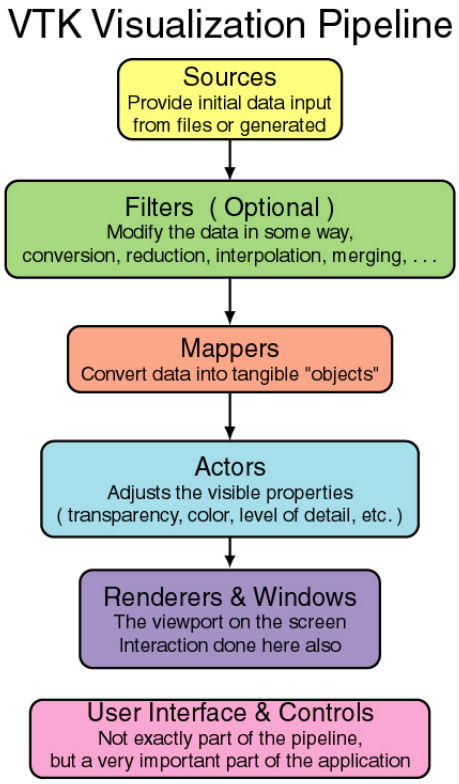

Figura 2.4: Pipeline de VTK[3]

Las alternativas basadas en VTK a pesar de ser bastante usadas en la práctica, no admiten todo tipo de geometrías, y su rendimiento se ve limitado por la velocidad de la CPU dado que los filtros en el pipeline están basados en algoritmos en CPU. Por ello es que en torno a VTK se ha desarrollado software como ParaView o VisIt que permiten llevar el procesamiento de las mallas a un servidor o un cluster, haciendo la visualización de grandes modelos más

[3]Fuente: `https://www.cs.uic.edu/ jbell/CS526/Tutorial/Tutorial.html`. Visitado: 2016-05-02

costosa. Este problema puede ser resuelto, en cambio, ocupando los beneficios que trae el uso de una GPU para procesamiento de datos.

2.2. Herramientas para el desarrollo de software gráfico

El desarrollo de aplicaciones de escritorio que ofrecen visualización científica y en general visualización de modelos complejos en 2D o 3D necesitan de tecnologías gráficas de apoyo. Dentro esas tecnologías se pueden encontrar las bibliotecas para crear interfaces gráficas y las bibliotecas para dibujar gráficos usando un chip de gráficos dedicado.

2.2.1. Interfaz gráfica

Para el desarrollo de aplicaciones con interfaz gráfica se dispone de bibliotecas como *wxWidgets*, *GTK+* o *QT*. Varias son multiplataforma y entregan distintos niveles de integración con cada sistema operativo. Una de las más acogidas por la industria es *QT*.

La versión inicial de QT se remonta a 1995[7] y su continuo desarrollo agrega nuevas funcionalidades para su compatibilidad con los distintos sistemas operativos sobre los que trabaja. En la actualidad soporta los sistemas Windows, Linux, Mac OS, iOS, Android y Windows Phone, además de las arquitecturas X86, X64 y ARM. El principal lenguaje de desarrollo que soporta QT es C++ a pesar de que existen adaptaciones a varios otros lenguajes como Python, Java o Haskell.

Qt provee un conjunto de herramientas para mostrar ventanas con una variedad de componentes. Cada ventana es pareada con una clase que la controla, siendo típicamente esta última la encargada de recibir eventos o cambiar los elementos de la ventana. Esto da como resultado una capa *vista* dentro del marco de desarrollo *MVC*.

Dentro de los programas orientados a manejar eventos en interfaces de usuario o cualquier otro tipo de evento asíncrono existe alguna forma de comunicar a distintos componentes la ocurrencia de dichos eventos. Por lo general el patrón más ocupado es el *Observer*, parte del conjunto de patrones del *"gang of four"*[13].

El patron *Observer* establece una clase que notifica eventos y otras que esperan ser notificadas (observadores). Para ello los observadores se registran previamente con la clase generadora de eventos (observable) y en el momento de que se genere el evento, la clase observable se encarga de notificar a cada uno de los observadores registrados. Por supuesto, para que ello ocurra es necesaria una interfaz común a todos los observadores.

En QT se toma un camino distinto, eliminando la necesidad de que exista una interfaz común a los observadores y sin la necesidad de que la clase observable notifique manualmente a sus observadores. El sistema de *signals* y *slots* establece un mecanismo de comunicación inter objetos en donde el framework se preocupa de la distribución de los eventos. Las clases observables definen los eventos que notificarán usando el concepto de *signals*(señales) especi-

ficando el tipo de los datos que emitirán. Las clases observadoras definen puntos de entrada llamados *slots*, los cuales deben corresponderse con los tipos de datos de las señales a los que sean conectados. Los slots finalmente son un método más de la clase observadora. Una tercera clase(cliente) realiza la conexión entre signals y slots usando la siguiente sintaxis:

```
QObject::connect(&observable, SIGNAL(valueChanged(int)),
                 &observer, SLOT(setValue(int)));
```

2.2.2. Bibliotecas de gráficos

El uso de una GPU en una aplicación permite acelerar procesos que de otra manera se ejecutarían órdenes de magnitud más lentos en una CPU. Esto es debido a que una GPU común posee del orden de cientos de cores o procesadores, mientras que una CPU en general posee entre 2 y 8. Para que una GPU sea utilizada por un sistema operativo, éste necesita un driver apropiado, con el cual se logra una comunicación de bajo nivel. Ahora bien, el uso de un driver no basta para usar una GPU de manera cómoda por un programador, y por ello se crearon las APIs orientadas a utilizar hardware gráfico. Dentro de las APIs más conocidas están Direct3D y OpenGL.

Direct3D es la API gráfica de la empresa Microsoft y forma parte de la familia de APIs *DirectX*; es usada por la mayoría del software comercial. DirectX apunta a proveer una interfaz completa de abstracción de hardware abarcando desde gráficos hasta hardware de input como mouse o controles de juego. Por otra parte está *OpenGL*, que al igual que Direct3D, provee una interfaz para que un programador tenga acceso a una unidad gráfica de forma estándar. OpenGL tiene además la característica de ser un estándar abierto y ha aumentado su uso en los años recientes por su compatiblidad con varios sistemas operativos, mientras que DirectX sólo es compatible en el sistema operativo Windows.

En general las bibliotecas gráficas permiten a un programador llevar un modelo matemático de una figura a una representación en una pantalla compuesta de píxeles, lo que le da una naturaleza discreta a la imagen final. El proceso de discretizar una imagen para llevarla a una matriz de píxeles es llamado *rasterización*, mientras que las imágenes resultantes son llamadas *raster*.

Existen varios modelos de representación de objetos, pero el más usado y cuya rasterización está optimizada por las bibliotecas gráficas usa una descripción discreta. Para describir una figura ya sea 2D o 3D, ésta puede ser descompuesta en polígonos más pequeños. Estos polígonos son típicamente triángulos, debido a que para describir un triángulo inequívocamente basta con especificar tres posiciones en el espacio. Si fuesen polígonos de más lados puede darse el caso que un programador especificara polígonos usando más de tres puntos no coplanares [4].

La forma general como funciona una librería de gráficos es aceptando llamadas a funciones

[4] A veces, se da la posibilidad de especificar polígonos de más de 3 lados, pero finalmente se triangulan en algún punto para resolver problemas de no coplanaridad.

desde un programa corriendo en CPU. Estas llamadas entregan la información necesaria para dibujar una escena como las posiciones de los vértices, colores u otros atributos. Finalmente esto ocasiona que un espacio de la pantalla se redibuje con el resultado final. Al proceso completo se le llama *rendering*. El espacio de pantalla en el que queda la imagen final se corresponde con una matriz en la tarjeta gráfica llamada *framebuffer* y almacena la intensidad de cada color primario (rojo, azul y verde) para un arreglo de píxeles que son llevados mediante hardware a una pantalla.

En la primera versión de OpenGL la forma de uso de la librería es mediante llamadas a funciones de dibujado inmediato. Es decir, con cada llamada se modificaba el framebuffer, haciendo que el uso de CPU fuese intensivo. También existían las *display lists* que permitían guardar una serie de llamadas a OpenGL para finalmente ser guardadas en la GPU como una lista de instrucciones que puede ser llamada a futuro. Lo problemático de esto es que dicho conjunto de instrucciones es estático y, por lo tanto, no se pueden hacer modificaciones a los valores de las instrucciones guardadas.

Con la llegada de OpenGL 2.0 se introdujo el concepto de VBO o Vertex Buffer Object por sus iniciales con el cual se pueden guardar datos en bruto dentro de la VRAM y que pueden ser usados más adelante con una única llamada de dibujado. Para darle sentido a estos datos y trabajar con ellos se crearon los *shaders*, programas escritos en un lenguaje cercano a *C* enfocados a guiar el proceso de dibujo dentro de un *pipeline*.

Pipeline

Las APIs gráficas en general siguen una estructura de procesamiento de datos llamado *pipeline*. Un pipeline de manera genérica es una serie de procesos ejecutados de forma serial en donde la salida de un proceso es la entrada de otro. Dentro de OpenGL existe un pipeline altamente complejo, aunque adaptado a las necesidades de la mayoría de los programadores, y provee puntos determinados donde un programador puede insertar un programa para controlar el flujo o procesamiento de datos.

Los datos de entrada en el pipeline son típicamente información de vértices, tal como su posición o color, aunque no está limitado a ello. Año tras año se desarrollan distintas técnicas de visualización que aprovechan las posibilidades que otorga un pipeline flexible y la cantidad de variables de entrada aumenta para dar más propiedades a los modelos que se quieren dibujar[14]. Un ejemplo de esto son las propiedades de reflexión de un material para aplicar un modelo de iluminación en una escena.

En pipeline de OpenGL puede divirse en dos grandes fases. La primera consiste en el procesamiento de geometrías y es la que toma la definición de los vértices para luego realizar transformaciones sobre ellos. Estas transformaciones incluyen en la mayoría de los casos aquellas que involucran operaciones matriciales para llevar las coordenadas de un vértice en un modelo al espacio de la pantalla. Más detalles de esto se dan en la siguiente sección.

La segunda fase es la rasterización de las geometrías en el espacio de pantalla. Esto significa que los polígonos al llegar a esta etapa son divididos en fragmentos del tamaño de los píxeles

de la pantalla en la que será desplegado el modelo final. Cada fragmento al final de su procesamiento corresponderá con un pixel y llevará un color escogido al final del pipeline.

Espacios y transformaciones

Las coordenadas de los vértices que componen objetos en una escena en el área de computación gráfica pasan por una serie de transformaciones para llegar a ser dibujadas en una pantalla. Estas transformaciones son la forma de cambiar las posiciones en forma de vector pasando por distintos espacios; desde el espacio del modelo hasta el de perpectiva de un observador. El cambio de espacio es realizado con la premultiplicación de los vectores de posición de cada vértice con matrices cuadradas.

La forma más general de representar un proceso de transformación de coordenadas comienza con especificar la posición de un vértice relativo a un punto de origen arbitrario en el espacio del *modelo*. Se pueden tener, entonces, varios modelos separados cuyos vértices están definidos según un origen asociado al modelo. Por ejemplo, los ojos de un personaje pueden ser instancias de un mismo modelo de ojo, y los vértices del modelo de ojo estarían dados respecto al origen escogido para el ojo. Luego, al aplicar una primera transformación a los vértices es posible reubicar todos los vértices del modelo a alguna ubicación en la escena. Se dice entonces que los vértices en esta primera etapa son llevados desde el espacio de modelo al espacio del mundo, y esto se hace premultiplicando por la matriz *model*.

En el espacio *mundo* las coordenadas de los vértices se corresponden todas con un único punto de referencia; el origen de la escena. Para representar una escena desde la posición de un observador, se necesita una transformación que ponga la escena completa desde su punto de vista. Esta transformación se realiza mediante la matriz llamada *view*.

Finalmente, cuando todas las coordenadas han sido transformadas al punto de vista de un observador, se toman en consideración las deformaciones que surgen por una proyección en perspectiva u ortogonal. Esto se realiza usando la matriz de *proyección*. La pantalla dentro de una escena es representada como un rectángulo frente al observador sobre la cual se proyectan los objetos de la escena y sobre la cual los vértices adquieren coordenadas relativas a al centro del rectángulo. Las coordenadas en espacio de pantalla van por lo general en el rango de -1 a 1.

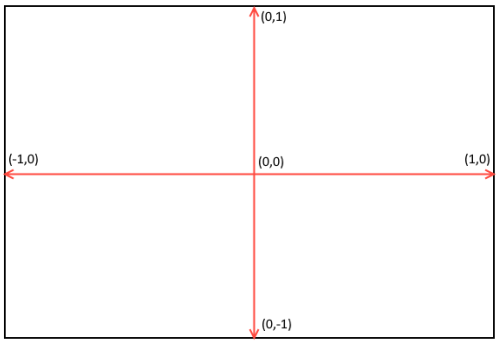

Figura 2.5: Coordenadas en espacio de pantalla en OpenGL

Los tipos de operaciones matriciales típicas aplicadas a vectores son rotaciones, escalados o traslaciones y son aplicadas en cada etapa de transformación. Una operación matricial básica para transformar un vector luce como $M * V$ donde M es una matriz de transformación cuadrada y V un vector de posición. Para concatenar estas operaciones se usa la premultiplicación en cadena de las matrices usadas para transformar el vector. Si se encadenan varias operaciones seguidas se llega a una operación con la siguente forma $M_n * ... M_2 * M_1 * V$ (notar que las operaciones son puestas en orden inverso), y dado que la multiplicación de matrices es asociativa se puede resumir el producto de las n matrices como una matriz única con la que se premultiplica V[5]. La matriz resultante que resume todas las transformaciones entre espacios es comunmente llamda *MVP*, cuyas siglas significan *Model*, *View*, *Projection*.

Shaders

Tanto OpenGL como Direct3D (la API gráfica de DirectX) proveen la capacidad al programador de usar las capacidades de una GPU a través de programas llamados *Shaders*. Éstos se ejecutan en los múltiples cores internos de una GPU, logrando un nivel de paralelismo imposible de alcanzar incluso con las mejores CPUs del mercado. Los *shaders* son compilados en tiempo de ejecución por el driver de la tarjeta gráfica y son enviados a la memoria de la GPU para su posterior ejecución. El lenguaje con el que se escriben los shaders en OpenGL se llama *GLSL* y si bien es estándar para todas las tarjetas gráficas, su versión compilada puede variar entre tarjetas.

GLSL es un lenguaje que fue introducido en OpenGL 2.0 para poder hacer cambios en el pipeline de rendering. Previo a ello se disponía de un pipeline fijo con el cual existían limitaciones que impedían obtener los resultados deseados. Hoy en día el uso de shaders es obligatorio al desarrollar aplicaciones que hagan uso de OpenGL.

Los shaders son insertados en puntos determinados del pipeline para controlar el flujo de datos y lograr resultados distintos en la visualización de un modelo. En cada punto del pipeline está definida la forma como serán entregados los inputs a un shader y también cómo deben ser pasados a la siguiente etapa. Los datos que se entregan al comienzo del pipeline son entregados en forma de paquetes y cada paquete contiene típicamente la información pertiente a un vértice de la escena.

Los tipos de shaders están listados a continuación. Cada tipo está ligado a un punto específico del pipeline:

Vertex shader Es el primer tipo de shader del pipeline y permite tomar un paquete de datos a la vez para hacerle una transformación inicial. Por lo general en esta etapa se calcula la posición que tendrá un vértice en el espacio de la pantalla usando la matriz MVP.

Tessellation Control Shader A partir de su versión 4.0 OpenGL provee la capacidad de teselar un polígono. Es decir, se puede partir un polígono en sub-polígonos para agregar detalle a una sección de un modelo. La etapa de Tessellation Control Shader permite especificar los parámetros de la teselación.

Tessellation Evaluation Shader Permite cambiar la malla resultante del proceso de teselación. De no hacerse este paso la malla teselada es igual a la original en cuanto a forma, sólo que posee una mayor densidad de polígonos.

Geometry shader Fue introducido en la versión 3.2 de OpenGL para dar la posibilidad a un programador de procesar primitivas completas (triángulos, segmentos de línea y otros) y emitir nuevas geometrías a partir de los datos de los vértices que conforman las primitivas de entrada. La forma como se determina el conjunto de puntos de entrada que forman una primitiva es definida por el programador.

Fragment shader Luego de que se determina la posición en pantalla de los vértices, se da paso a una *rasterización* consistente en la discretización de la imagen final en una grilla de fragmentos que finalmente se convierten en píxeles. Con un Fragment Shader se pueden especificar el color de un pixel final dada la información de su respectivo fragmento.

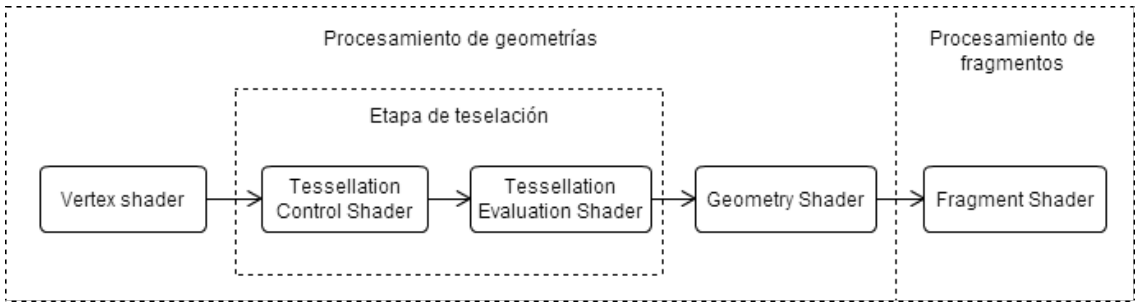

Figura 2.6: Pipeline simplificado de OpenGL.

Se puede ver una mejor organización de estos shaders en la figura 2.6

Si bien, el proceso de renderizado puede ser modificado mediante el uso de shaders, hay áreas que son manejadas con parámetros usando llamadas desde CPU a OpenGL. Una vez que se establece uno de estos parámetros, puede afectar las operaciones posteriores que se realicen hasta que vuelvan a ser modificados nuevamente. En otras palabras, OpenGL funciona como una máquina de estados.

2.3. Programación orientada a objetos

La programación orientada a objetos surge en la época de los 60 con algunos lenguajes que se acercaban a los conceptos actuales de POO. El primer lenguaje que estableció un modelo formalmente aceptado como POO fue *Simula 67* y fue influcencia para varios otros lenguajes como *SmallTalk*, *Pascal* o el mismo *C++*.

Para organizar los datos dentro de una aplicación y la forma de manejarlos se inventó el paradigma POO que introduce el concepto de *objeto* para denotar un conjunto de datos y procedimientos llamados *atributos* y *métodos*, respectivamente. Los atributos mantienen información dentro de un objeto, exigiéndole a éste la responsabilidad de mantenerlos en un

estado consistente. Para modificar los atributos de un objeto o hacer otro tipo de acciones con él, los métodos ofrecen una interfaz disponible para ser usada por otros objetos. Esto da como resultado un ecosistema en el que distintos objetos *colaboran* para resolver un problema mayor.

La construcción de objetos se hace en la mayoría de los lenguajes más usados mediante un sistema de *clases*. Éstas especifican inequívocamente los tipos de objetos que serán usados en una aplicación así como sus atributos y métodos. Sin embargo, se recomienda que la definición de los métodos sea hecha en lo que se llama *interfaz*. Ésta última permite definir en un único lugar la forma que toma un objeto para el resto del software, y de la cual pueden derivarse clases que usen aquella interfaz como máscara para agregar funcionalidad a cada método. De esta manera se logra que las clases de un programa estén más desacopladas, teniendo como intermediario una interfaz. A partir de las clases se pueden *instanciar* nuevos objetos con la estructura especificada para dar paso a objetos de distintos *tipos* que pueden ser usados dentro de un programa.

Un aspecto importante dentro de la POO es la reusabilidad del código. Se ocupa el principio de que si algo ya está hecho puede ocuparse para un nuevo programa, y si es necesario algún cambio en su funcionalidad se ocupa una característica llamada *herencia*. Por medio de ella se pueden redefinir comportamientos de una clase para que se adapten a los requerimientos del nuevo entorno donde debe desempeñarse.

Finalmente, sumado a la ventaja de la reusabilidad de código se puede destacar también el encapsulamiento de roles en que cada clase desempeña un rol específico que no es compartido por otras secciones del software. Esto permite que el mantenimiento de un software sea más fácil y ofrece ventajas a la hora de probar su correcto funcionamiento.

Patrones de diseño

A medida que gran parte del software fue tomando territorio en el mundo de la POO ciertos modos de resolver problemas comenzaron a ser recurrentes. Finalmente algunos autores estudiaron estos patrones y publicaron libros que los detallan, entregando sus ventajas. El libro más famoso de esta área es *Design Patterns*[13] y dentro de los patrones de diseño más comunes y que fueron usados en Camarón podemos destacar:

Factory Method Define una interfaz para crear un objeto, dejando a las subclases decidir cuál clase instanciar. Este patrón permite a una clase delegar la instanciación a sus subclases. Ejemplo: Delegar la instanciación a distintas subclases de un tipo específico de conexión a una base de datos. Esto permite que a la hora de intentar realizar una nueva conexión, el cliente obtenga el objeto correspondiente a una base de datos Postgres, MySql u otro.

Singleton Asegura que existe sólo una instancia de una clase y provee un punto de acceso global a ella. Ejemplo: Una cola de operaciones global para una aplicación debiera ser única. El patrón *Singleton* asegura que se cumpla.

Strategy Define una familia de algoritmos, encapsulándolos y los hace intercambiables. *Stra-*

tegy deja que un algoritmo varíe independiente del cliente que lo use. Ejemplo: Un cliente necesita leer información desde un archivo, pero la existencia de varios formatos da como resultado varios algoritmos para leer dicha información. La solución es entregarle al cliente una estrategia que encapsule la obtención de información para el tipo de archivo correspondiente.

Visitor Representa una operación que debe ser hecha sobre los elementos de la estructura de un objeto. Visitor permite definir una nueva operación sin cambiar las clases de los elementos sobre los que opera. Ejemplo: La serialización de objetos de distinto tipo puede ser hecha por un objeto externo usando el patrón Visitor. Se definiría una forma distinta de serialización para cada tipo de objeto.

Existe también un modelo de arquitectura de aplicaciones llamado MVC[15] (model-view-controller) que establece una separación de roles para las distintas clases de una aplicación. El patrón ubica aquellas clases ligadas a ser una interfaz con el usuario en la capa *view*. Aquellas ligadas al acceso de datos en la capa *modelo* y finalmente las que manejan el tráfico de datos y opera sobre él en la capa *controlador*. Una clase que corresponde a una cierta capa se le nombra por el su rol (i.e. una clase en la capa controlador es nombrada como un *controlador*).

2.3.1. Recolección de basura

Los lenguajes de programación por lo general trabajan sobre dos espacios de memoria principales llamados *stack* y *heap*. En el stack se van apilando las llamadas de funciones en lo que se llaman *frames*. Los frames almacenan las variables temporales y referencias a objetos. Por otro lado, en el heap se guardan objetos que no se quieran asociar a un frame particular en el stack de llamadas de funciones, o cuyo tamaño no está determinado en tiempo de compilación.

Los lenguajes de programación que corren sobre una máquina virtual por lo general son beneficiados con la limpieza automática de objetos del heap. La destrucción de objetos que ya no son referenciados es un mecanismo llamado *Garbage Collection* o recolección de basura y evita que un programador tenga que destruir los objetos que ya no están en uso para liberar memoria. La eliminación de objetos del stack, en cambio, es automática porque al terminar la ejecución de código de un *scope* (bloque de código entre llaves **{}**), el frame correspondiente a dicho scope es liberado de la memoria.

2.3.2. Lenguajes de programación

El lenguaje de preferencia para el desarrollo de aplicaciones de alto rendimiento C++ creado en 1985 surgió como una evolución del lenguaje de bajo nivel C con la adición del paradigma de programación POO (programación orientada a objetos). Al ser uno de los primeros lenguajes abiertos en implementar POO su popularidad creció para transformarse en uno de los lenguajes más usados en la industria.

C++ es un lenguaje imperativo cuya sintáxis se parece mucho a la de C y al igual que su antecesor puede ser ejecutado en varias arquitecturas de procesador, manteniendo un dominio multiplataforma.

Otro aspecto que destaca C++ sobre otros lenguajes es que su código es compilado a instrucciones de procesador directamente, por lo que no es necesaria una máquina virtual de por medio. Esto afecta de manera positiva la rapidez en la ejecución de las aplicaciones escritas en este lenguaje. Un gran competidor de este lenguaje es *Java*, un lenguaje que si bien se parece bastante a C++, es ejecutado en una máquina virtual o intérprete. Esto da ciertas facilidades a los programadores, pero reduce el rendimiento de las aplicaciones.

En cuanto a administración de memoria, C++ al no tener un máquina virtual depende de medios más manuales para la liberación de memoria. En C se usa que el programador solicite un bloque de memoria al sistema operativo con la llamada de función `malloc(size_t size)`, para luego liberar ese bloque con la llamada `free(void *ptr)`. Esto fue dejado de lado en C++ con el keyword `new` que instancia un nuevo objeto y la función `delete(void* ptr)` que lo destruye. Sin embargo, la forma recomendable para instanciar objetos es en el stack, ya que así al terminar un bloque de código se destruye el objeto automáticamente. Esta forma de programar se llama *RAII* (Resource Aquisition Is Initialization) y consiste en que la inicialización de un recurso implica que se tiene control sobre él, para luego perderlo una vez que se destruye el objeto. Una ventaja de RAII en la administración de memoria es que el momento en que ocurre la destrucción de objetos es determinista, es decir, se puede saber en qué momento un objeto será destruido antes de ejecutar el programa. Esto contrasta con los enfoques basados en recolección de basura automático en que no se sabe a priori cuándo se destruirán los objetos del heap, provocando caídas de rendimiento espontáneas cuando se ejecuta el recolector de basura[5].

Es por esto que para aplicaciones cuyo rendimiento es crítico se prefiere usar C++, de otro modo puede sacrificarse rendimiento por comodidades a la hora de programar.

[5]Sin embargo, estas caídas no afectan tanto en la práctica, puesto que los recolectores de basura corren en un thread separado y actualmente están bastante optimizados.

Capítulo 3

Análisis y Diseño

En este capítulo se estudian las posibles soluciones para alcanzar los objetivos planteados. En particular se describe el análisis de requerimientos previo, las nuevas funcionalidades a través de casos de uso y cambios sugeridos en el diseño del software Camarón. Además, se presenta un diseño a llevar a la etapa de implementación que se encuentra dentro de los límites de tiempo disponibles, y las razones por las que se descartaron otros posibles diseños enunciando sus falencias.

3.1. Camarón: Versión original

3.1.1. Características generales

Para facilitar el análisis de propiedades geométricas en mallas tanto de superficie como de volumen se creó Camarón. Este software permite de manera gráfica decidir si una malla satisface ciertos criterios geométricos. Otro software del área no permite realizar esto, por lo que Camarón se convierte en un fuerte competidor para un nicho de científicos e ingenieros dedicados al procesamiento de mallas usando el método de elemento finito.

Aunque ideado inicialmente para el análisis de propiedades geométricas, Camarón evolucionó para ser fácilmente extensible en varios aspectos, permitiendo flexibilidad en su potencial desarrollo y por lo tanto diversificar sus usos. A pesar de que el software cumple con su objetivo original, también incluye funcionalidades adicionales que permiten demostrar la capacidad que tiene su arquitectura de crecer en funcionalidad. Este aspecto es importante a la hora de abarcar diversas necesidades, ya que es posible que se puedan solucionar otros problemas tanto asociados como distintos a los planteados originalmente.

Camarón está fuertemente enfocado en mantener un alto rendimiento, incluso con mallas de gran tamaño. Esto permite a investigadores que trabajan con mallas más finas poder analizar la calidad de los elementos geométricos constituyentes sin mayores problemas. Para ello, tanto el lenguaje de desarrollo como las herramientas disponibles en hardware fueron

considerados para lograr aprovechar los recursos de una máquina de forma eficiente.

Dado que el rendimiento de la aplicación es crítico para soportar mallas de gran tamaño, el lenguaje C++ se acomodó bien junto a las facilidades de la biblioteca QT para hacer una interfaz gráfica amigable. Dado que también el uso de las capacidades gráficas de una máquina es crucial para el software, la alternativa de biblioteca gráfica OpenGL fue escogida por ser una biblioteca potente, usada ampliamente y con soporte multiplataforma.

3.1.2. Arquitectura

El software actualmente es fácilmente extensible en cuanto a:

Formatos de entrada/salida de archivo Actualmente se soportan los formatos *ELE/-NODE*, *OFF*, *PLY*, *M3D*, *TRI*, *TS* y *VISF*. El último de estos fue creado durante el desarrollo de Camarón derivado del formato OFF que incluye listados de caras al final para formar poliedros.

Estrategias de selección de geometrías Permite usando una variedad de formas para seleccionar geometrías ya sea usando un rectángulo de selección, intersección con otra geometría o por valor calculado por algún criterio, entre otras. Esto permite finalmente entre otras cosas poder ocultar las geometrías seleccionadas y realizar un corte en el modelo para visualizar su interior, lo que es bastante útil en modelos volumétricos.

Estrategias de renderizado Gracias a ellas se puede escoger cómo debe dibujarse un modelo en pantalla, destacando así aspectos de interés o simplemente para generar una imagen a usarse en alguna publicación. Entre los tipos de visualización existen sombreado Phong, colores planos con malla de alambre o vectores normales entre otros.

Criterios de evaluación Son funciones que le asignan un valor escalar a cada geometría para luego poder hacer un análisis sobre la distribución y ubicación de dichos valores. Son una parte importante del software puesto que el objetivo principal de Camarón está fuertemente ligado a que un usuario identifique los sectores que no satisfacen algún criterio geométrico.

Los aspectos mencionados anteriormente son extensibles gracias a la manera como está diseñado el software. Existen objetos llamados *registros* o *factories* desde donde se pueden obtener instancias de cada tipo de modo de visualización, forma de selección, estrategia de evaluación e incluso formas de cargar o guardar un modelo desde y hacia un archivo. Luego, para agregar un nuevo componente se deben implementar las interfaces necesarias y agregar una macro que hace el trabajo de agregar una instancia del nuevo objeto al registro correspondiente.

A lo largo de este proyecto se integran nuevas clases en las áreas extensibles del software para lograr funcionalidades nuevas. También se agregan elementos nuevos a la arquitectura original para soportar propiedades nuevas en los modelos ligados a la visualización científica. Además se proponen nuevas formas de desarrollo orientadas a utilizar herramientas nuevas

del lenguaje C++.

El proceso mediante en cual se visualiza un modelo en pantalla inicia con la carga de geometrías desde memoria secundaria (un archivo) escogido por el usuario mediante una interfaz gráfica. Camarón admite varios formatos de entrada de datos y cada uno tiene asociada una clase capaz de leer datos del formato correspondiente. Luego, el controlador principal de la aplicación le pide al registro de objetos para lectura de archivo el objeto adecuado para parsear el archivo seleccionado por el usuario. En seguida se produce la carga de los datos en donde la infomación de vértices, polígonos o poliedros son llevados a una nueva instancia de alguna clase derivada de `Model`. Para la lectura de datos de distintos tipos se ocupan métodos de la clase utilitaria `FileUtils` que permite parsear datos desde un buffer en forma de enteros, strings u otros.

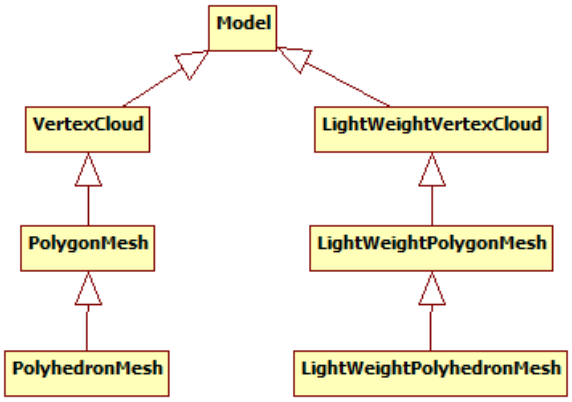

Figura 3.1: Diagrama UML de clases para la representación de modelos

La clase `Model` representa un modelo cargado, aunque es una clase abstracta, con lo que es necesario intanciar una de sus clases heredadas: `VertexCloud` para nubes de vértices, `PolygonMesh` para una malla de polígonos o `PolyhedronMesh` para una malla de poliedros. También existen variantes livianas cuyo propósito es usar menos memoria RAM al cargar modelos más complejos. Éstas son `LightWeightVertexCloud`, `LightWeightPolygonMesh` y `LightWeightPolyhedronMesh`.

Como se ve en la figura 3.1, los tipos de modelo siguen una relación de herencia en cadena. Este tipo de relación no se condice con lo que dicta el concepto de subtipo. El modelo ideal de relaciones de clases debiera ser tal que las clases heredadas que actualmente existen en el software extiendan a la clase `Model` directamente, por ser conceptos distintos. Sin embargo, la estructura de clases se dejó tal como está, dado que no afecta a los cambios que se harán en el software.

La diferencia entre las variantes normales y liviana es que en las primeras se realiza un cálculo de relaciones de vecindad entre geometrías. Es decir, se establecen punteros entre objetos que permiten una navegación rápida por la malla, y gracias a ello se puede realizar el cálculo de propiedades geométricas en un tiempo despreciable. El problema es que esos punteros ocupan memoria adicional no despreciable, pero el uso de RAM por sobre el de CPU es una decisión de diseño que se tomó desde un principio.

El siguiente paso que realiza el controlador de la aplicación una vez que los datos de un modelo están en memoria primaria es calcular las relaciones de vecindad entre geometrías. Luego de ello la información de los vértices es enviada a la tarjeta gráfica. Los datos necesarios para dibujar un modelo deben estar en un lugar de fácil acceso para la GPU y ese es la VRAM o RAM de video, la que se encuentra en la tarjeta de video misma. Esto permite a las aplicaciones que usan las capacidades gráficas de una máquina permitir al procesador gráfico un acceso rápido a los elementos constituyentes de una escena. La clase encargada de realizar este trabajo es `RModel` y es independiente del tipo de modelo cargado, por lo que no posee una jerarquía.

Cada vez que un conjunto de datos es cargado a la tarjeta gráfica, OpenGL retorna un identificador con el que se puede referenciar a dicho conjunto de datos en la VRAM. En Camarón estos identificadores son guardados dentro de `RModel`. Esta clase también mantiene variables relacionadas al estado de la escena a visualizar como la matriz de transformación, contadores de acceso rápido de la cantidad de geometrías, color de fondo, los límites del bounding box que encierra el modelo y otros.

Finalmente, llega el turno de los renderers. El usuario puede seleccionar desde una lista desplegable el nombre de un renderer y como consecuencia cambiar la forma como se muestra el modelo actual. Cada renderer tiene un método `draw(RModel*)` que es llamado cada vez que se requiere redibujar la escena. Al llamarse este método se usan los identificadores disponibles en la instancia actual de `RModel` para finalmente hacer un llamado a la función `glDrawArrays` de OpenGL que ejecuta un dibujado de escena con los parámetros que se hayan configurado hasta ese momento. Cabe destacar que los renderers pueden ser agregados a una lista de renderers para ser combinados, lo que da como resultado que se puedan ocupar varios renderers en una escena.

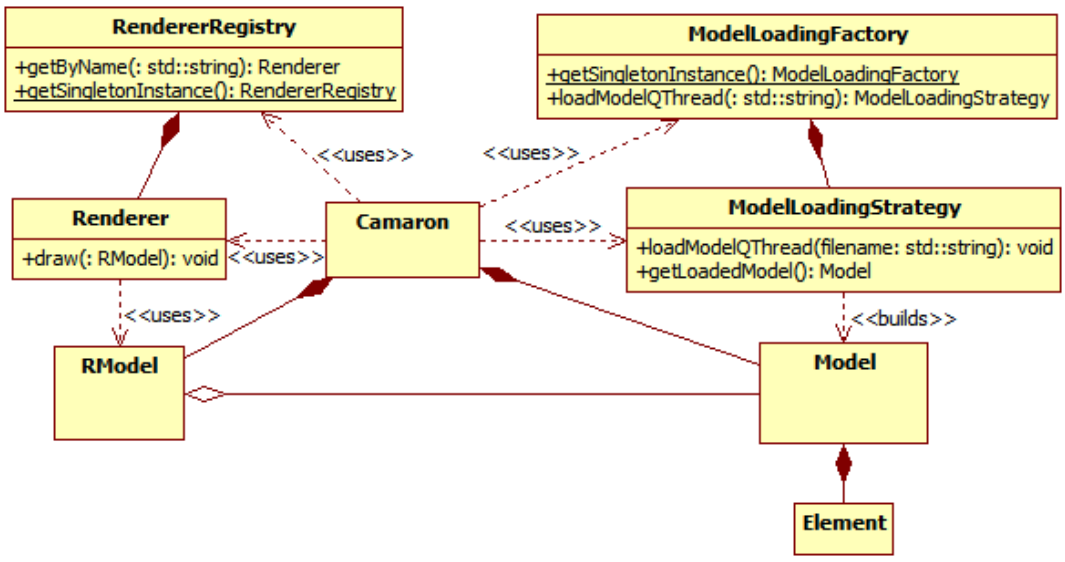

Figura 3.2: Diagrama de las clases involucradas en el proceso de carga de un modelo en Camarón

3.1.3. Entradas

Las geometrías que se cargan en Camarón provienen de archivos en memoria secundaria y se estructuran en general como listas de vértices y polígonos. En el caso de modelos volumétricos, se incluye información de poliedros.

Los vértices contienen información de su posición, descrita por los valores X, Y y Z. Estos valores son típicamente de coma flotante para almacenar valores reales. Adicionalmente pueden tener información de propiedades adicionales como valores escalares. Los polígonos se definen como listas de punteros a los vértices declarados. Cada vértice lleva asociado un identificador que es usado en el resto de la entrada para ser referenciado. Por último, los poliedros se componen de referencias a polígonos. La forma de referenciarlos es la misma que de polígonos a vértices.

Tanto para polígonos y poliedros se exige que sean convexos. Esta limitación no es una restricción muy exigente, ya que la mayoría de los modelos existentes están hechos ya sea de triángulos o tetraedros (inherentemente convexos), o por celdas generadas en un diagrama de Voronoi[1]. Esto permite que el trabajo con la GPU sea más natural y en general se simplifican algunos algoritmos. Al exigir que los poliedros sean convexos se evita que al convertir un volumen a tetraedros se encuente un caso imposible como el poliedro de Schönhardt[2].

Para ilustrar mejor el tipo de entrada podemos describir un cubo como sigue y usando como referencia la figura 3.3:

- Vértices: `0:(0,0,0), 1:(0,0,1), 2:(0,1,0), 3:(0,1,1), 4:(1,0,0), 5:(1,0,1), 6:(1,1,0), 7:(1,1,1)`
- Polígonos: `0:(0, 4, 6, 2), 1:(4, 6, 7, 5), 2:(0, 4, 5, 1), 3:(1, 3, 2, 0), 4:(3, 7, 6, 2), 5:(1, 5, 7, 3)`
- Poliedros: `0:(0, 1, 2, 3, 4, 5)`

3.2. Análisis de requerimientos

Para definir mejor los problemas a resolver, las nuevas funcionalidades se describen a través de casos de uso. Un caso de uso representa la interacción entre un programa y un usuario especificando las acciones presentes en dicha interacción.

Los casos de uso siguientes incluyen lo relacionado a visualización científica, pero también visualización para la depuración de algoritmos de malla. Se debe permitir visualizar aristas adicionales indicadas en un archivo de entrada y asociadas a colores dados en el archivo. La investigación que realiza el alumno tiene como objetivo generar triangulaciones de mallas de poliedros en GPU y para evaluar las transformaciones que se hacen en la malla es necesaria

[1]Este diagrama divide el plano en áreas cuyos puntos interiores son los más cercanos a un conjunto de puntos dado. Este tipo de diagrama se puede extrapolar a más dimensiones.

[2]Este poliedro es el caso más simple de un poliedro que no se puede subdividir en tetraedros ocupando sólo los vértices originales.

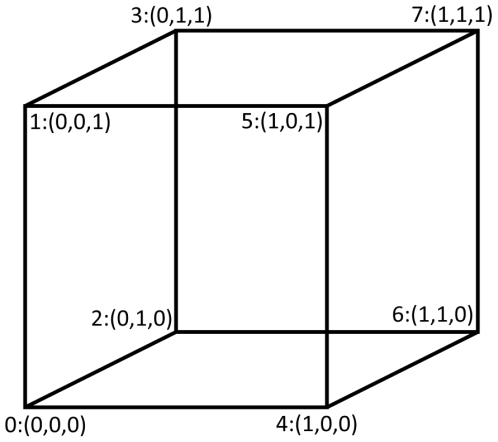

Figura 3.3: Cubo de ejemplo para input.

una herramienta para visualizar un conjunto de aristas seleccionado por un algoritmo desarrollado por el alumno. Para esta aplicación cada arista de este conjunto se correspondería con aristas existentes intrínsecamente en las definiciones de los polígonos del modelo.

Tabla 3.1

CU1: Abrir un modelo con un campo de propiedades asociado	
Precondiciones: Se tiene a disposición un modelo contenido en uno o más archivos soportados por Camarón que contengan uno o más campos escalares.	**Postcondiciones:** Camarón tiene cargado el modelo y se está visualizando con algún renderer. Los valores del campo de propiedades están cargados en la aplicación.
Actores: Usuario de la aplicación	
1. El usuario abre la aplicación 2. Mediante el menú superior de la aplicación el usuario abre el modelo. 3. Si el modelo tiene un campo escalar asociado, el programa ofrece la posibilidad de abrirlo. De otro modo el usuario debe importar el campo escalar una vez más usando el menú superior.	

3.3. Integración de nuevas funcionalidades

En esta sección se describe el diseño de las funcionalidades que se agregan a Camarón para exponer como funcionan y cómo se integran al software.

3.3.1. Campos de propiedades

Para agregar las nuevas funcionalidades a Camarón se agregan clases en los puntos de extensión disponibles. En específico se agregan más renderers y se cambian o agregan formas

Tabla 3.2

CU2: Visualizar isolíneas de un campo escalar asociado a un modelo cargado.	
Requiere: CU1	
Precondiciones: Hay un modelo cargado en Camarón con al menos un campo de propiedades escalar asociado.	**Postcondiciones:** Se visualizan las isolíneas en pantalla especificadas por el usuario. El usuario debe puede girar, hacer zoom o paneo del las isolíneas resultantes para cambiar el punto de vista.
Actores: Usuario de la aplicación	
1. El usuario señala mediante un botón u opción que quiere graficar isolíneas. 2. El usuario escoge el campo escalar con el que desea operar. 3. El usuario escoge el modo de colorear las isolíneas. Ya sea con una función de coloreo pre-programada o con un gradiente de color dados dos colores para el mínimo o máximo. 4. El usuario escoge los valores de las equipotenciales que quiere graficar. 5. El usuario presiona un botón para que se apliquen los cambios y se muestren en pantalla las isolíneas que especificó.	

Tabla 3.3

CU3: Visualizar isosuperficies de un campo escalar asociado a un modelo cargado.	
Requiere: CU1	
Precondiciones: Hay un modelo cargado en Camarón con al menos un campo de propiedades escalar asociado.	**Postcondiciones:** Se visualizan las isosuperficies en pantalla especificadas por el usuario. El usuario debe puede girar, hacer zoom o paneo del las isosuperficies resultantes para cambiar el punto de vista.
Actores: Usuario de la aplicación	
1. El usuario señala mediante un botón u opción que quiere graficar isosuperficies. 2. El usuario escoge el campo escalar con el que desea operar. 3. El usuario escoge el modo de colorear las isolíneas. Ya sea con una función de coloreo pre-programada o con un gradiente de color dados dos colores para el mínimo o máximo. 4. El usuario escoge los valores de las equipotenciales que quiere graficar. 5. El usuario presiona un botón para que se apliquen los cambios y se muestren en pantalla las isosuperficies que especificó.	

de cargar un modelo desde un archivo. Además se agrega una nueva sección del software orientada a los *campos de propiedades* en donde se adiciona un punto de extensión nuevo.

Para integrar visualización científica en un software es necesario asociar a los elementos

Tabla 3.4

CU4: Visualizar aristas adicionales.	
Precondiciones: Se tiene a disposición un modelo en un archivo compatible con Camarón y contiene definiciones de aristas adicionales.	**Postcondiciones:** Se visualizan las aristas adicionales definidas en el archivo de entrada.
Actores: Usuario de la aplicación	
1. El usuario abre la aplicación 2. Mediante el menú superior de la aplicación el usuario abre el modelo. 3. El usuario presiona un botón o escoge una opción para visualizar líneas adicionales en el modelo.	

geométricos propiedades adicionales que representen campos escalares o vectoriales. Agregar este tipo de propiedades impacta el uso en memoria RAM de los elementos geométricos dado que se intenta reservar más espacio para almacenar estas propiedades. Además, el tiempo de carga de archivos se ve afectado al leer una mayor cantidad de información.

En Camarón cada elemento geométrico tal como los puntos, polígonos o poliedros son representados como objetos que extienden de una clase llamada `Element`, la cual contiene métodos y propiedades necesarios en cada uno de los tipos de geometrías usadas en el software. Dentro de las variables de instancia de `Element` se encuentra un mapa (diccionario) de propiedades usado como caché para guardar los valores calculados por los las estrategias de evaluación de geometrías. Este mapa que asigna números identificadores de propiedades a valores numéricos del tipo coma flotante. Las llaves de este mapa son de tipo `byte` sin signo, y por lo tanto permite tener un total de hasta 256 propiedades. En el software existe un total de 15 estrategias de evaluación de elementos geométricos repartidas entre vértices, polígonos y poliedros, lo que implica que el mapa de propiedades de un elemento tiene una ocupación máxima cercana al 5 %. Dado que si bien este mapa de propiedades no reserva espacio para 256 entradas en un principio, es un lugar semánticamente adecuado para almacenar propiedades de campos escalares o vectoriales.

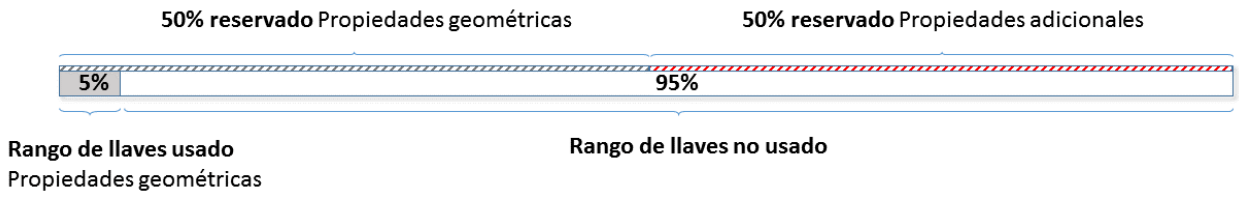

Figura 3.4: Uso del rango de propiedades en la clase Element. Se muestran los porcentajes de uso actual y los rangos que se quieren reservan para almacenar propiedades geométricas y propiedades adicionales, como valores escalares.

Se definió un rango correspondiente a la mitad de los identificadores disponibles en el mapa de propiedades para almacenar exclusivamente valores de campos de propiedades. El problema de identificar a qué corresponde cada valor en el mapa de propiedades recae en un

tipo de objeto nuevo almacenado en la clase `Model` llamado `PropertyFieldDef` que contiene la definición de un campo de propiedades.

Los objetos de tipo `PropertyFieldDef` pueden almacenar el nombre de un campo de propiedades, y deben especializarse según el tipo de datos que almacenan. En el caso de un campo escalar el subtipo `ScalarFieldDef` es usado y permite, por ejemplo, almacenar el máximo y mínimo valor del campo escalar, los cuales son necesarios en otras partes del software.

3.3.2. Cargado de Campos de propiedades

Como requisito el software debe ser capaz de cargar campos de propiedades desde archivos y para ello existe un número de posibilidades. Existen, por ejemplo, los archivos de tipo PLY que pueden almacenar información de geometrías tales como puntos, polígonos o poliedros, pero también puede asociarse propiedades a estos elementos. En el caso de un campo escalar, basta con asociar una propiedad de tipo coma flotante a cada vértice. También está el caso en que el campo de propiedades puede encontrarse fuera del archivo de definición de la geometría.

Para hacer una modificación que incluya ambos escenarios mencionados anteriormente, se establece que un usuario puede cargar un modelo en Camarón y que adicionalmente pueda cargarle campos de propiedades desde un archivo anexo. Esto permite importar incrementalmente campos de propiedades sobre un modelo ya cargado.

Para cargar campos de propiedades desde archivos existe la posibilidad de que las actuales estrategias de carga de modelos (geometrías) sean capaces de cargar campos de propiedades. Esto conlleva a que cada vez que se quiera cargar un campo de propiedades desde un formato, dicho formato también debe permitir almacenar geometrías que deben ser cargadas junto a esas propiedades. Esto se contradice con el requerimiento de que se puedan cargar campos de propiedades desde archivos separados de la definición de geometrías.

Como solución se decide crear una nueva sección del software encargada exclusivamente de cargar campos de propiedades desde archivos. Para aquellos archivos que almacenan tanto geometrías como campos de propiedades (por ejemplo, PLY), se deja la clase original que carga las geometrías y se crea una nueva clase encargada de extraer únicamente los campos de propiedades disponibles.

Para lograr lo anterior se crea un nuevo registro de objetos tal como el de estrategias de selección o cargado de modelos llamado `PropertyFieldLoadingFactory`. Esta clase almacena todas aquellas instancias de objetos que permiten cargar campos de propiedades y ocupa el mismo sistema de registro de objetos que los demás registros.

Junto al nuevo registro, se crean las nuevas clases que realizarán el cargado de los campos de propiedades.

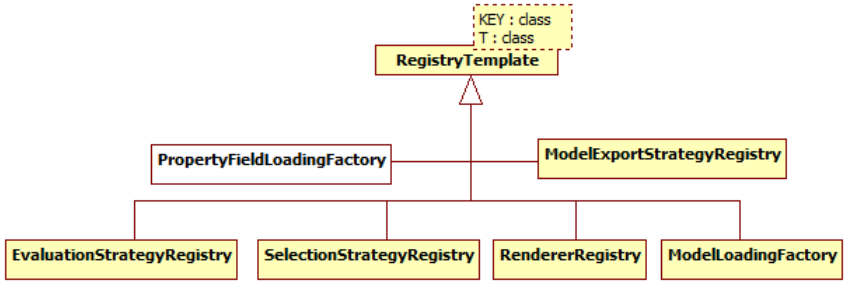

Figura 3.5: Adición de nuevo registro en Camarón. En blanco aparece resaltada la nueva clase añadida.

3.3.3. Campos de propiedades alternativos

Como experimento se realizó el caso de estudio del profesor Alejandro Ortiz del Departamento de Ingeniería Mecánica, quien trabaja analizando mallas de polígonos con valores unidimensionales tal como un campo escalar. La diferencia con los campos escalares vistos hasta ese momento es que los datos están asociados a los polígonos de la malla en ubicaciones internas a los polígonos.

Cada polígono de la malla tiene posiciones asociadas llamadas *puntos de Gauss* en donde hay calculados esfuerzos de una pieza. Dichas posiciones son definidas en forma genérica (independiente de su forma) usando algún sistema de cuadratura determinado. Esto implica que existe una definición previa de dónde se encuentran los puntos, a diferencia de un archivo más común en donde no existe dicha definición, pues los valores están asociados a los vértices y no se requiere más información.

Este es un claro ejemplo de que el software no puede estar limitado a campos de propiedades escalares o vectoriales. Es posible tener casos en que incluso la definición de dónde se encuentran los valores o vectores es más complejo y requiera de especialización de clases para manejar esos casos.

En el anexo A se puede encontrar un ejemplo del formato de archivo sugerido para el almacenamiento de los datos para resolver el problema anterior.

3.3.4. Renderers

Con la adición de campos de propiedades en la estructura de Camarón surge la posibilidad de mostrar isolíneas o isosuperficies asociadas a un modelo con propiedades escalares. Se crearon entonces las clases `IsolineRenderer` y `IsosurfaceRenderer`.

Para parametrizar los renderers de isolíneas e isosuperficies se creó un diálogo de configuración compartido, ya que la forma de parametrización de ambos tiene la misma forma. Este diálogo despliega una lista de campos escalares que pueden ser usados para realizar la visualización. Para ello es necesario que puedan filtrar sólo los objetos de tipo `ScalarFieldDef`

desde la lista completa de objetos `PropertyFieldDef` del modelo cargado. Luego, cada vez que un renderer es notificado de un nuevo Modelo, debe actualizar su lista de propiedades escalares a mostrar. Para ello se usa un patrón *visitor*, permitiendo filtrar un tipo de objeto y agregarlo a una lista. Para agregar campos escalares a una lista se creó una clase llamada `ScalarFieldListAdderVisitor`. Los renderers usan esta clase para visitar cada objeto `PropetyFieldDef` del modelo y agregarlos a su lista de campos de propiedades con la cual se actualiza la interfaz en un paso posterior.

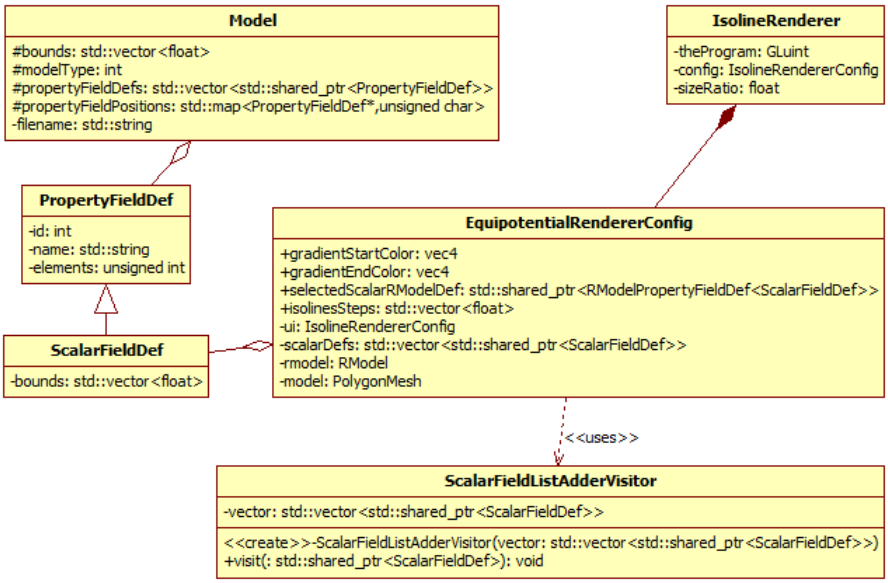

Figura 3.6: Diagrama UML de filtrado de campos de propiedades

Barrido de isolínea o isosuperficie

Una característica interesante que fue implementada es la de hacer un barrido por un modelo en vivo con isolíneas o isosuperficies. La manera como esto funciona es que el usuario ocupa un deslizador para hacer un barrido de un isonivel y ve en vivo cómo se recorre el modelo desde el menor isonivel hasta el mayor.

Para introducir este cambio se requería que un componente en el diálogo de configuración de un renderer refrescara el modelo. En este caso, cada vez que se cambiara el valor de un deslizador. Sin embargo, no se contaba con esta característica.

Para dar contexto, la manera como se dispone en Camarón de varios diálogos de configuración para renderers es con un objeto llamado `RendererConfigPopUp` que es en efecto un diálogo que a su vez incluye un botón *apply* y un contenedor para los controles específicos de un renderer. Estos controles vienen contenidos en un objeto `QWidget` correspondiente a un contenedor genérico de *QT*. Por defecto, el único control que puede provocar un refresco del modelo es el botón *apply*. Para permitir que un control al interior del QWidget provisto por un renderer provoque un refresco del modelo es necesario que ese QWidget también implemente una interfaz común que permita saber cuando algún componente interno requiere

hacer un refresco del modelo visualizado.

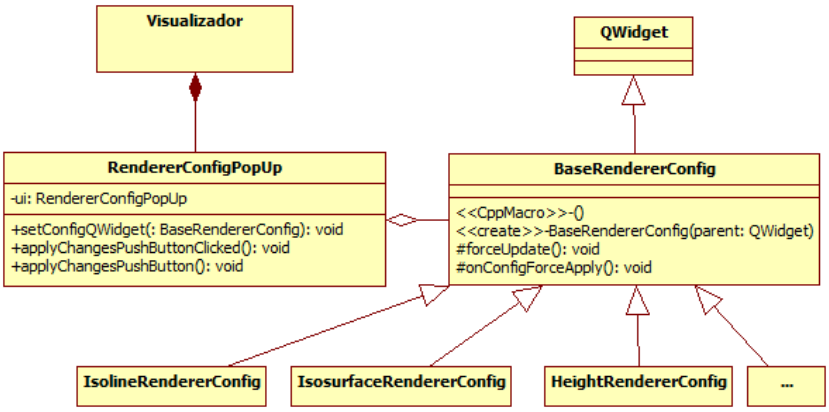

Figura 3.7: Diagrama de clases de diálogos de configuración

Se creó una clase intermedia llamada `BaseRendererConfig` que extiende de QWidget y de la cual ahora penden todos los diálogos provistos por los renderer disponibles. El objeto de tipo `RendererConfigPopUp` luego es capaz de capturar un evento de refresco y enviarlo al controlador de la aplicación. Finalmente el esquema de clases queda como en la figura 3.7.

Manejo en GPU

Dentro del proceso usual de cargado de un modelo se realiza la carga de información a la memoria de la GPU para permitir un renderizado más rápido. La clase encargada de hacerlo es `RModel` y maneja todo lo relacionado a un modelo y su renderizado con GPU.

Con la adición de propiedades agregadas a elementos geométricos se hace necesario que `RModel` también permita cargar dichas propiedades a la memoria de la GPU, las cuales son asociadas a sus respectivos elementos geométricos durante el proceso de rendering con shaders. En principio se carga un sólo campo de propiedades a la vez dado que por lo general la memoria de una GPU es menor que la memoria principal de un computador. La información en la GPU se guarda en buffers identificados por un número y éstos pueden ser usados durante el proceso de dibujado como entradas para el pipeline de OpenGL. Para especificar información de los vértices de una escena en uno o varios buffers es necesario especificar cómo se encuentra guardada dicha información. Para ello se especifican los valores de *stride* y *offset*. *Stride* es la separación entre datos de vértices contíguos en el buffer y *offset* es usado para indicar la primera posición de un dato en el buffer. Junto con el identificador de buffer, estos 3 valores son requeridos por el proceso de dibujado para saber dónde se puede encontrar la información requerida por cada campo de propiedades. Es por ello que se creó el objeto `RModelPropertyFieldDef` cuyo objetivo es el de almacenar el identificador de buffer, *stride*, y *offset* asociados al buffer que almacena los valores en GPU de un campo de propiedades. Este objeto también mantiene una referencia al objeto de tipo `PropertyFieldDef` original. Luego, cada vez que se necesita renderear un nuevo campo de propiedades, es cargado mediante objeto `RModel` y representado mediante un `RModelPropertyFieldDef`.

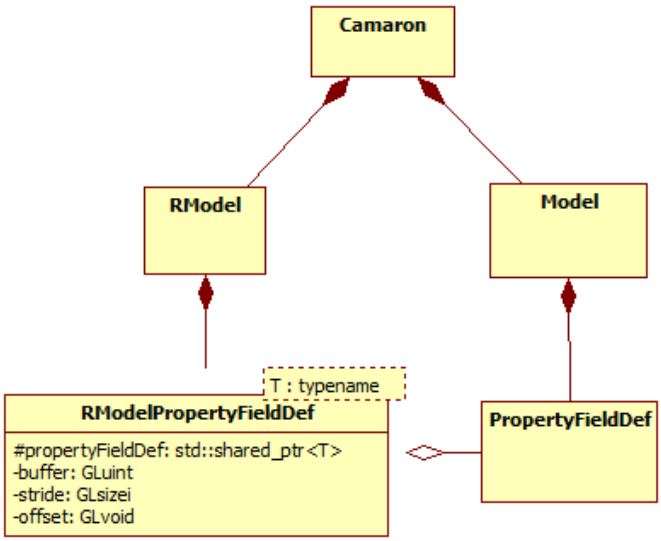

Figura 3.8: Diagrama representativo de la definición de propiedades para la GPU.

3.3.5. Aristas adicionales

Como se describió a grandes rasgos en la sección 3.2, se requiere que Camarón sea capaz de integrar información de aristas especiales en un modelo que tendrán asociado un color. Este último viene dado en el archivo de entrada. Esta funcionalidad tienen por objetivo apoyar la depuración de un algoritmo de optimización de triangulaciones ayudando a visualizar sobre qué arco se aplicará un proceso posterior, aunque no se limita a ello. Con esto por primera

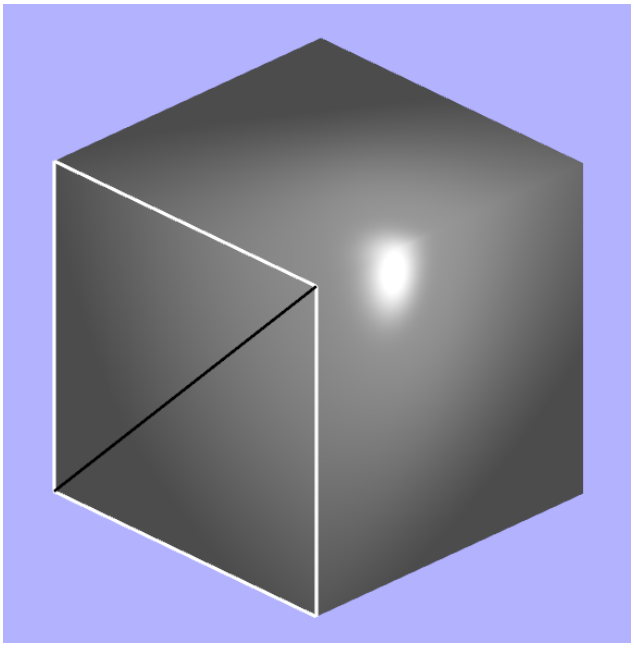

Figura 3.9: Ejemplo de cubo con aristas adicionales. Cuatro aristas son blancas y una cruzada es negra.

vez en Camarón se le asigna un mayor peso al concepto de *arista* dentro de un modelo, permitiendo que ellas tengan información adicional.

Se acordó que el formato de archivo con el que se leerían los datos en Camarón sería PLY, el cual es un formato abierto y posee buena extensibilidad para algunas aplicaciones extra. El formato PLY por lo general es usado para definir objetos de tipo vértice, polígono o poliedro, pero tiene la capacidad de almacenar definiciones de tipos de objetos nuevos dentro del encabezado del archivo. Con ello se dispone de la libertad de crear un nuevo tipo de objeto llamado *edge* con referencias a dos vértices apuntados por sus respectivos índices y con tres valores de tipo *float* que representan las intensidades de los colores *rojo*,*verde* y *azul* del trazo.

Para reflejar este nuevo tipo de constructo en el esquema de clases de Camarón se presentaron 2 opciones: La primera es la de guardar en los objetos de tipo `Polygon` información de trazos destacados que posean y su respectivo color. Si bien esto se adapta bien a las necesidades de la investigación en triangulaciones, presenta dos problemas. El primero es que las aristas son en la mayoría de los casos compartidas por 1 o más polígonos, y eso conlleva a que si se define un color distinto para una arista en los polígonos que la comparten, no queda claro cómo se debe colorear dicha arista. Por otra parte, si se lleva a casos extremos, el archivo de entrada podría definir nuevas aristas que no necesariamente se correspondan con aristas intrínsecas de los polígonos del modelo y luego, la lógica de guardar información de aristas adicionales en polígonos pierde sentido. La segunda opción de implementación es la de guardar las definiciones de nuevas aristas como objetos de tipo `Edge` contenidas en una lista dentro de la clase `VertexCloud` que representa al modelo actualmente cargado (`PolygonMesh` y `PolyhedronMesh` extienden de `VertexCloud`). Esto permite mayor flexibilidad y es agnóstico en cuanto a si los vértices apuntados conviven en un mismo polígono, o incluso si es que hay polígonos en el modelo, ya que el cambio es implementado a nivel de un modelo tipo nube de puntos en vez de una malla de polígonos.

Finalmente, para visualizar las nuevas aristas en el software es necesario que sean tomadas en cuenta por alguno o todos los *renderers*. En Camarón la visualización de aristas implícitas, es decir, parte de los polígonos o poliedros cargados, viene dada en el listado de vértices que conforman los polígonos. Los *renderers* no cuentan con la capacidad de mostrar aristas provistas adicionales y por ello es necesario implementar un nuevo renderer cuyo trabajo sea el dibujado de ellas. Estas aristas son cargadas a la GPU por el objeto `RModel` que hace de interfaz con la GPU y con el uso de un vertex y Fragment Shader simples se logra que cada arista se coloree apropiadamente en el rendering final. Esto da como resultado una vista de las aristas coloreadas, aunque sin el modelo original superpuesto. Para lograr que se muestren ambos se puede usar el apilamiento de renderers disponible consistente en agregar renderers a una lista para que se ejecuten todos para cada operación de rendering.

OpenGL determina qué fragmentos deben dibujarse sobre otros por la profundidad calculada desde el punto de vista del observador. Si un fragmento está más cerca del observador que otro, entonces el primero sustituirá al segundo para luego colorear el pixel correspondiente. Para evitar que partes de una arista dibujada queden detrás de los polígonos vecinos se dibujan las aristas un poco más cerca del observador para asegurar que al menos parte de la línea no sea obstruída. Luego, el orden de dibujado de los renderers no es importante.

Capítulo 4

Implementación

Una vez hechas las decisiones de diseño en el capítulo anterior se procede a describir en este capítulo con mayor detalle la forma como se implementaron las nuevas funcionalidades que se incluyen en Camarón. Se incluye como información adicional algunas modificaciones a nivel de calidad de código.

4.1. Visualización científica

Se crearon 3 nuevos *renderers* para permitir la visualización de un modelo con isolíneas, isosuperficies o degradado de color. Los detalles de las implementaciones se exponen a continuación. Para cada uno de los renderers se hizo una interfaz de usuario adecuada que permite escoger los isoniveles y colores a mostrar, así como seleccionar la propiedad que se quiere visualizar.

4.1.1. Degradado de color

El rendering con degradado de color es uno de los métodos de rendering más simples implementados y sirve para visualizar un campo escalar representado con un degradado de colores partiendo de un color inicial para el valor más bajo hasta otro para el valor más alto.

La lógica de cómo se colorea el modelo viene dada en un *Fragment Shader*. Éste recibe un valor de la propiedad que se quiere mostrar mediante una variable de tipo `smooth`, la cual indica que es una variable que se interpola linealmente en un polígono dados los valores en sus vértices. Los valores son establecidos previamente en los vértices por un *Vertex Shader*.

4.1.2. Isolíneas

La generación de isolíneas fue el primer intento realizado de extensión de funcionalidades en cuanto a renderers. Esto constituyó una primera experiencia en la adición de componentes nuevos al software.

Para la generación de isolíneas se intentaron dos tipos de soluciones. La primera está enfocada en generar geometrías nuevas en el pipeline de OpenGL y la otra funciona pintando las líneas sobre el modelo.

Los métodos que se enunciarán a continuación y fueron implementados en Camarón llevan por nombre *Método Geometry Shader*, *Método Fragment Shader* y *Método Transform Feedback* por la característica principal que usan de OpenGL y son referenciados de esa manera por el resto de este informe.

Método Geometry Shader

En un principio se decidió hacer isolíneas usando un *Geometry Shader* que generara los segmentos de línea de cada isolínea en cada triángulo cada vez que se redibujara el modelo. Esta solución aprovecha las capacidades de paralelismo de la GPU, ya que no hay conflictos entre triángulos sobre los que se trabaja.

Para generar cada segmento de isolínea se verifica en un *Geometry Shader* por cada uno de los isoniveles que el usuario quiere graficar si está entre los valores mínimo y máximo de los triángulos del modelo. En caso positivo, se genera un segmento interpolado usando los

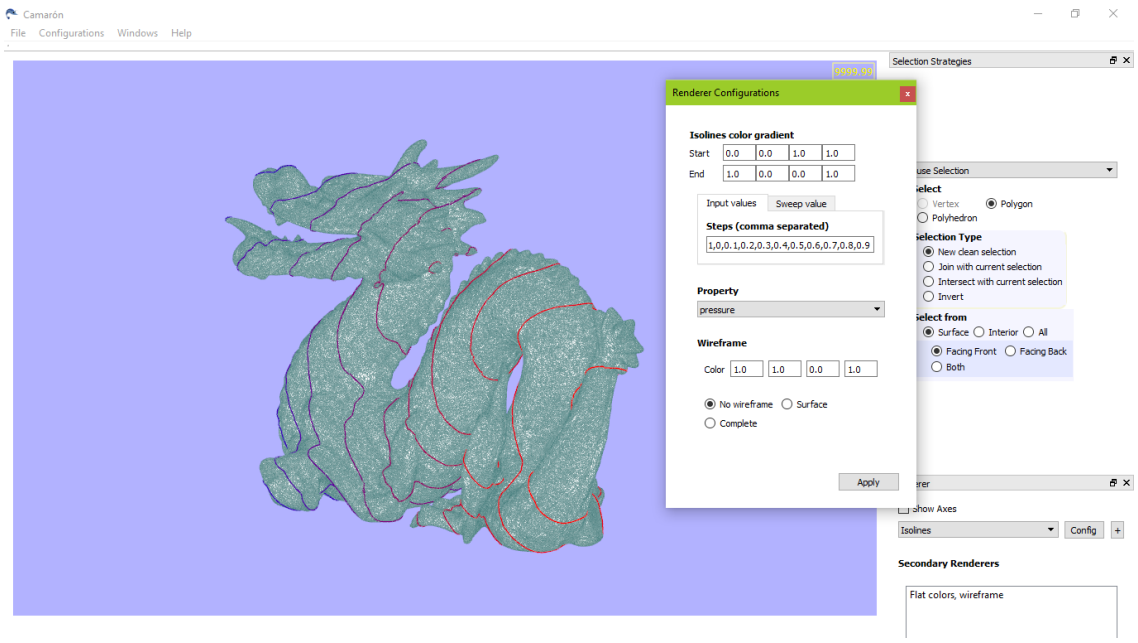

Figura 4.1: Ejemplo de dibujado de isolíneas en Camarón. Se usó una distribución equiespaciada entre el mínimo y máximo valor. Hay dos renderers activos: *Flat-colors* y *Isolines*

valores del triángulo.

Finalmente en un *Fragment Shader* se escoge el color final de cada pixel en cada segmento interpolando linealmente entre un color para el valor mínimo y otro para el valor máximo. El resultado de este proceso se muestra en la figura 4.1. Cabe destacar que las líneas son primitivas de OpenGL y la forma como se dibujan recae en OpenGL mismo.

Como se verá en la sección de resultados, el rendimiento de este método resulta ser bastante bajo.

El modelo usado en la figura 4.1 es *dragon* del repositorio de modelos de la universidad de Stanford con un campo escalar generado con un algoritmo de pseudo-ruido y es usado en el resto de la memoria para varios ejemplos.

Método Fragment Shader

Para mejorar el rendimiento de la generación de isolíneas se intentó realizar una solución sin *Geometry Shader* para evitar generar geometrías en el pipeline.

La solución consiste en marcar con un color distinto en un *Fragment Shader* las áreas de los polígonos que tengan un valor cercano al isonivel que se intenta marcar. El valor en cada punto se determina mediante una interpolación lineal entre los vértices de los triángulos. La interpolación es realizada por OpenGL usando el modificador de variable `smooth`.

La solución goza de un rendimiento muy superior a la solución con la generación de geometrías en *Geometry Shader*, pero como era de esperarse, las líneas generadas no tienen un ancho fijo. El ancho de la línea depende del gradiente de la propiedad que se intenta representar. En efecto, el ancho de la línea generada es inversamente proporcional al gradiente de la propiedad en ese punto.

Para solucionar el problema anterior se ideó una solución consistente incluir el gradiente de la propiedad para determinar el ancho de cada línea. El ancho de la línea se determina exigiendo que la diferencia entre el valor de un punto y el valor del isonivel sea menor a un "valor de ancho". Lo que se hace entonces, es escoger un valor de ancho multiplicado por el valor del gradiente. Finalmente, el Fragment Shader colorea los fragmentos que satisfagan la condición:

$$|V_{ij} - I| < f * s_{ij}$$

donde V_{ij} es el valor de una propiedad en el fragmento (i, j), I es el valor del isonivel, f es un factor de ancho y s_{ij} el gradiente de la propiedad en el fragmento (i, j). El gradiente es calculado para cada triángulo en un *Geometry Shader*.

Luego de la implementación de la corrección de gradiente se vieron tres problemas más:

- Al ver una línea generada desde un ángulo se ve más delgada porque ésta es dibujada sobre la superficie del modelo, mientras que en la solución con Geometry Shader la línea es dibujada como una geometría nueva.

Figura 4.2: Imperfecciones usando el método Fragment Shader. Izquierda: Líneas generadas pueden resultar quebradizas. Derecha: El ángulo de visión afecta el ancho percibido de las líneas.

- El otro problema es que el ancho de las líneas está determinado por un factor que no es fijo para todos los modelos, y el ancho de las líneas generadas puede variar mucho de modelo en modelo.
- Las líneas generadas con este método resultan quebradizas por las diferencias por la estructura irregular de la malla.

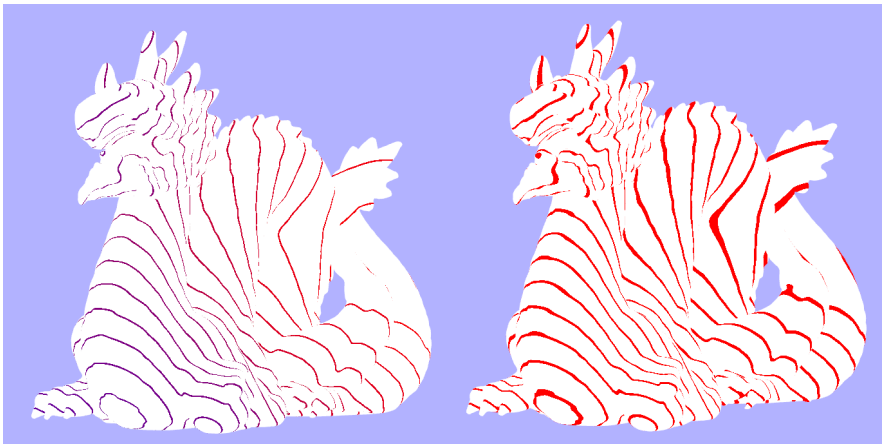

Figura 4.3: Defectos rendering de isolíneas en Fragment Shader con corrección de gradiente. Al hacer un acercamiento se puede notar que la línea generada presenta defectos bastante notorios.

Para resolver los problemas anteriores, se ocupó una característica de OpenGL llamada *Transform Feedback* con la cual se hizo una implementación para el rendering de isolíneas de ahora en adelante llamado *Método Transform Feedback*, a pesar de usar también un Geometry Shader, y su uso es detallado en la sección de generación de isosuperficies. El algoritmo descrito en dicha sección es análogo para el caso de isolíneas.

4.1.3. Isosuperficies

La generación de una isosuperficie a partir de mallas discretas, requiere elaborar una nueva malla, también discreta, de geometrías que conforman una superficie. Lo más simple, pero que a la vez da buenos resultados, es la generación de un conjunto de triángulos unidos por sus aristas de tal forma que emulan una superficie suave. Esto es aceptable para la mayoría de las situaciones y no debería constituir un problema de la misma forma que la malla original es también discreta.

La visualización de isosuperficies se realizó en dos partes: La generación y el rendering de las isosuperficies generadas. La primera etapa hace uso de un Vertex Shader y un Geometry Shader. En conjunto permiten generar todas las geometrías que componen el conjunto de isosuperficies deseadas. La segunda parte consta de dibujado de geometrías generadas en la primera etapa.

La malla de entrada para generar isosuperficies debe estar compuesta por poliedros convexos. Por cada polidero se generan tetraedros de forma simple. Se selecciona un punto de referencia dentro del conjunto de vértices que conforman el poliedro y se genera un nuevo tetraedro uniendo el punto de referencia con cada triángulo que compone el tetraedro y que a la vez no contenga el punto de referencia. Si el tetraedro contiene polígonos que no son triángulos, se realiza una triangulación simple bajo la premisa de que los polígonos son convexos (requerimiento original de todas las mallas de entrada de Camarón).

La generación de tetraedros anterior no resulta en la creación de nuevos objetos en memoria que los representen, sino en una lista de índices de los vértices que conforman cada tetrahedro. La forma como se organizan es tal que cada cuatro índices de la lista se tiene un nuevo tetraedro formado por los vértices apuntados por esos cuatro índices. Es natural que los índices se repitan varias veces a lo largo de la lista, ya que son parte de varios tetraedros.

Esta lista de índices es entregada a la GPU para la etapa inicial de generación de isosuperficies. En ésta los vértices de cada tetraedro pasan por un *Vertex Shader* para luego entrar directamente a un *Geometry Shader* donde se ensabla cada tetraedro. Es aquí donde la GPU tiene acceso y vista a una geometría completa y se emiten los triángulos de la(s) isosuperficie(s) resultante correspondientes a la región limitada por el volumen del tetraedro que se procesa.

Para generar los triángulos correspondiente a la nueva superficie se ocupa un algoritmo basado en interpolación lineal. Éste está basado en la implementación de Cyril Crassin[9]. En la recta infinita formada por cada par de puntos de un tetraedro (6 pares en total) se determina la ubicación del punto por el cual pasaría la isosuperfice interpolando linealmente los valores escalares del par de puntos del tetraedro. Esto da como resultado 6 puntos nuevos que pueden estar tanto dentro como fuera de las aristas del tetraedro. Para producir el extracto de isosuperficie que pasará por el tetraedro actual se tendrán que generar 0, 1 o 2 triángulos y la forma como se seleccionan los puntos para formar dichos triángulos es mediante una tabla precalculada.

La tabla de triángulos permite saber de manera rápida cuáles son los puntos a seleccionar

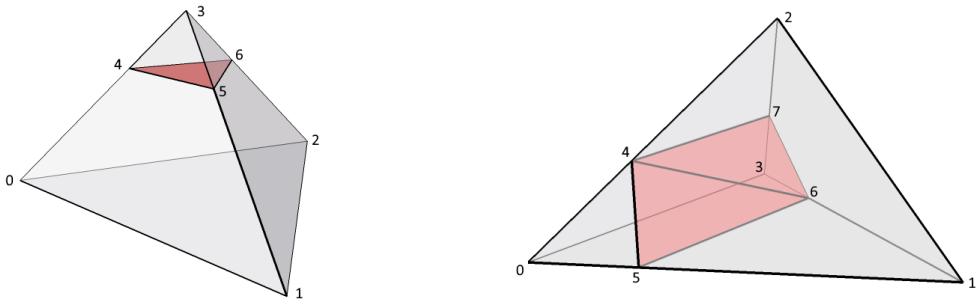

Figura 4.4: Intersección de isosuperficie con un tetraedro. (Izquierda) Isosuperficie interseca dando origen a 1 triángulo (4,5,6). (Derecha) Isosuperficie interseca resultando en 2 triángulos (4,5,6) y (4,6,7).

para generar los triángulos de salida del *Geometry Shader*. Esta tabla contiene información sobre cuáles de los puntos interpolados deben ser usados para construir entre 0 y 2 triángulos de salida. Por cada fila de la tabla se listan de a tres los índices de vértices por triángulo y la selección de estos índices está ligada a la posición que ocupa la fila en la tabla. Cada fila tiene fuertemente asociada la representación en binario de su posición en la tabla. Esta representación binaria se corresponde con cada una de las posibles configuraciones que puede presentar un tetraedro basado en si los valores escalares en los vértices son mayores o menores al isonivel. Un ejemplo de esto es el caso en que todos los vértices tienen valores escalares por debajo del isonivel. Si se codifica un `0` por cada vértices por debajo del isonivel, se tiene entonces la representación binaria `0000` que corresponde al número `0` en decimal. Luego, al obtener la primera fila de la tabla, ésta sólo contiene valores `-1` que simbolizan que no se debe seleccionar ningún punto interpolado, ya que la isosuperficie no pasaría por ese tetraedro. En el caso de otra representación binaria, podrían haberse seleccionado los vértices 0, 1 y 3 para el primer triángulo, por ejemplo.

Sabiendo ya cuáles puntos interpolados serán emitidos agrupados como triángulos, sólo queda que sean guardados en la GPU para su posterior visualización. Notar que por lo general tanto en un *Vertex Shader* como en un *Geometry Shader* las posiciones de las geometrías pasan por operaciones matriciales para llevarlas al espacio de pantalla. Sin embargo, en este caso no se hace, pues no se intentan llevar geometrías a pantalla por lo menos hasta este momento.

Para guardar geometrías en GPU se ocupa una prestación de OpenGL llamada *Transform feedback*. Mediante ésta se pueden guardar atributos de salida seleccionados del último paso de procesamiento de geometrías en un buffer dentro de la memoria de la GPU y así no es necesaria la interacción con la CPU o con la memoria primaria de la máquina. En el caso de esta etapa se guardan las posiciones de los vértices de cada triángulo junto con la normal, la que es incluída para permitir una visualización posterior con un modelo de iluminación básico. Debido a la arquitectura de OpenGL la normal debe repetirse en los 3 vértices. Luego, el tamaño total de cada vértice generado es 24bytes: Posición (3 * 4 bytes), normal(3 * 4 bytes).

El proceso anterior se ejecuta para cada isovalor y los resultados se almacenan en buffers distintos para cada isosuperficie resultante. De otro modo, las geometrías resultantes de todas

las isosuperficies se tendrían que guardar en el mismo buffer y se necesitaría guardar en cada vértice el isovalor que le corresponde dada la arquitectura de OpenGL.

La segunda etapa corresponde a la visualización de las geometrías generadas, y para ello se realiza el proceso tradicional de rendering de geometrías con un Vertex Shader y un Fragment Shader. Se hacen tantos llamados de rendering como isosuperficies se hayan generado en la etapa anterior usando como input los buffers donde quedaron guardadas las geometrías de la primera etapa. En este punto sólo se dibujan triángulos, y el color que les asigna finalmente el Fragment Shader es dado por una variable de tipo uniform[1].

Cuando el usuario rota, hace zoom o paneo del modelo, sus isosuperficies no cambian. Es por esto que si el usuario cambia la vista del modelo, sólo se ejecuta la segunda etapa; de dibujado de isosuperficies. Esto logra un máximo rendimiento al no tener que calcularse nada del modelo luego de haberse ejecutado la primera etapa, y así la inspección de las isosuperficies es fluída.

Un extracto del geometry shader ocupado en este método puede encontrarse en el anexo B.

4.2. Calidad de código y misceláneos

Si bien Camarón tiene un buen funcionamiento de la forma como está implementado, es posible mejorar algunas funcionalidades en cuanto a código para reducir el tamaño de algunas clases y mejorar la legibilidad de las instrucciones en el software.

4.2.1. Smart Pointers

El software Camarón fue desarrollado originalmente en 2012, cuando aún la última versión de C++ llamada *C++11* comenzaba a masificarse. En C++ la liberación de memoria ha sido típicamente tarea del desarrollador; algo acarreado del lenguaje *C*. Sin embargo, en el estándar C++11 se introdujo un número de mejoras que incluye entre otras cosas nuevas herramientas para un mejor manejo de objetos en el heap. Ya que es posible establecer punteros a objetos en el heap desde el stack, es posible entonces crear objetos más complejos que actúen como punteros aunque con características agregadas.

En el estándar C++11 se introducen los *punteros inteligentes* o *smart pointers* que permiten alivianar las tarea de manejo de memoria en el heap. En particular, los *shared pointers* otorgan al programador la facilidad de tener punteros con contador de referencias a un objeto en el heap, lo que permite un manejo de memoria un poco más orientado a los lenguajes que poseen un recolector de basura, ya que no es necesario llamar al destructor de los objetos referenciados.

[1]Las variables uniform son constantes y compartidas en todas las ejecuciones paralelas de un programa de shaders

La manera como funcionan estos punteros es mediante el mecanismo clásico de creación y eliminación de objetos del stack. Al crearse un nuevo objeto en el heap referenciado por un *shared pointer* se crea además un objeto en el heap que mantiene un contador de referencias. Al copiar un shared Pointer, se incrementa el contador de referencias. Al destruirse un objeto del stack por terminarse la ejecución de instrucciones en el scope actual su destructor es llamado. Esto conlleva a que el shared pointer decremente el contador de referencias guardado en el heap. Si este contador de referencias llega a cero, se eliminan el contador y el objeto referenciado.

En las nuevas secciones agregadas se usaron *smart pointers* con el objetivo de desligarse de las tareas de limpieza de un objeto. Por ejemplo, los objetos de tipo `PropertyFieldDef` son referenciados desde objetos de tipo `Model`, `RModelPropertyFieldDef` y algunos diálogos de configuración de renderers. Se logra simplificar así parte del software con el costo de tener que declarar punteros con una notación algo más larga:

```
std::shared_ptr<PropertyFieldDef> mProp(new ScalarFieldDef(...));
```

en vez de

```
PropertyFieldDef* mProp = new ScalarFieldDef(...);
```

4.2.2. *For* basado en rango

En el código se encontraron muchos ciclos *for* que ocupan iteradores o índices para iterar sobre los elementos de un arreglo. La sintaxis correcta para realizar esto en C++ implica saber el tipo de iterador o índice a usar. Para el caso de uso de índices en otros lenguajes el tipo del índice se asume como un valor entero de 32 bits como es el caso de *Java*. Dado que C++ es un lenguaje enfocado a ser ejecutado nativamente en distintas arquitecturas, se exige que el código sea genérico con tal de que no existan problemas de consistencia en ninguna arquitectura.

El tipo de índice de un ciclo *for* en C++ debe corresponderse con el tipo de tamaño que exige la estructura de datos sobre la que se iterará. Es decir, si se itera sobre un vector de `Element*`, el código correspondiente es:

```
for(std::vector<Element*>::size_type i=0; i<myVector.size();i++) {
...
}
```

Como se ve, el código es algo más largo que en otros lenguajes. Se da un caso similar para las iteraciones con iterador.

C++11 provee una forma más sencilla de iterar sobre listas llamada *Range-based for loops*[1] que se encuentra en otros lenguajes. Esta forma de iteración se implementó en todas las partes del código de Camarón donde fuese posible. El ejemplo anterior se vería así:

```
for( Element* element : myVector) {
...
}
```

4.2.3. Expresiones regulares y lectura de datos

La lectura de archivos en Camarón se realiza con la clase `CharArrayScanner` creada por el desarrollador anterior cuya utilidad es la de leer un archivo representado como un arreglo de caracteres para luego extraer valores usando funciones como `readFloat()` o `readString()`.

El problema de esta clase es que la forma de lectura de datos está muy orientado al estilo del lenguaje C en que los strings o cadenas de texto son manejados como arreglos de caracteres en vez de instancias de la clase `std::string` que posee varios beneficios sobre un arreglo de caracteres común y corriente. El problema de usar arreglos de caracteres es que el programador debe preocuparse de la liberación de recursos y esto agrega código innecesario siendo que se está programando en un lenguaje que ofrece herramientas que ofrecen un mejor manejo de memoria.

La lectura de datos de un archivo requiere volcar todo el archivo a memoria primaria para luego procesarlo. Esto tiene problemas evidentes en el uso de RAM ya que hay archivos de modelos que llegan a tener un peso de más de 1GB. Además, para la lectura de datos y su posterior transformación a un tipo deseado se copian bytes a un buffer de tamaño fijo de 256 bytes hasta que se encuentre un caracter de término como un espacio o un salto de línea sin tener en cuenta el límite del buffer. Si bien cada dato leído por el software típicamente no supera los 256 bytes, este límite puede ser superado si se está leyendo un archivo malformado o malicioso que trate de explotar la vulnerabilidad.

Para solucionar los problema anteriores se decidió ocupar *streams* de C++ y *expresiones regulares*. Los stream son usados para procesar un flujo de datos sin la necesidad de tener todos los datos almacenados en un buffer en memoria principal. La lectura de archivos puede ser realizada mediante streams y con ello el uso de memoria al leer un archivo es bastante menor. La expresiones regulares mejoran la manera como se interpretan datos ya que permiten especificar de manera muy precisa cómo se espera que sea el formato del texto leído. Si una línea cumple con el formato, los valores de interés (especificados como grupos a capturar en la expresión regular) son capturados en un objeto de tipo `std::smatch` donde son almacenados como strings. Para la extracción de valores de distintos tipos desde el objeto de tipo `std::smatch` se se ocupa el operador >> de la clase `std::istream` con la que se hace la lectura de valores a variables de tipo float, int, etc.

Capítulo 5

Resultados

En esta sección compara la solución implementada en un ambiente real con el software ParaView, la alternativa libre para realizar visualización científica. Se muestran resultados cuantitativos de velocidad y uso de memoria en varios escenarios.

5.1. Metodología

Los aspectos que se evalúan en el presente trabajo son el rendimiento y uso de memoria de la aplicación al generar y visualizar isolíneas e isosuperficies en modelos de distintos tamaños. Esto permite generar indicadores que ayudan a comparar el software con otros en el mercado. En el caso de este trabajo se evalúan las diferencias con el software ParaView que es parte de la familia de software basado en VTK, la principal herramienta de trabajo en visualización científica de código abierto.

Las pruebas de rendimiento se hacen con modelos cuya cantidad de vértices es parametrizada e incluyen un ruido pseudo-aleatorio. La construcción de estos modelos se explica en la sección 5.2.

5.1.1. Hardware y software

Todas las pruebas realizadas son hechas en un computador Asus con las siguientes especificaciones:

- Modelo: Asus X450ln
- CPU: Intel Core i7-4500U @1.80GHz (boost: 2.4GHz)
- GPU: NVidia GeForce GT 840M 2GB DDR3 (driver 362.0)
- RAM: 8GB DDR3
- OS: Windows 10 (64-bit)

Para realizar comparaciones se usó ParaView 4.4.0 64-bit disponible desde el sitio oficial de la aplicación[3].

Camarón fue compilado usando el compilador *g++ (x86_64-posix-seh-rev1) 4.9.2* disponible en el ambiente *MinGW 4.9.2(64 bits)* para Windows. Las opciones de compilación relevantes son:

-std=c++0x Activa las funcionalidades del estándar C++11.

-O2 Realiza una serie de optimizaciones usuales a los archivos compilados.

-fexceptions Habilita el manejo de excepciones.

-frtti Habilita la generación de información sobre cada clase con funciones virtuales para el uso de las características de identificación de tipo en tiempo de ejecución como *dynamic_cast* y *typeid*.

5.2. Datos de prueba

Para realizar tanto pruebas cualitativas como de rendimiento se generaron campos escalares para modelos existentes. Para ello se usó un algoritmo de generación de ruido pseudo-aleatorio. Esto permitiría decidir si un algoritmo de visualización de isolíneas o isosuperficies funciona de manera adecuada al poder verse de manera fácil si los resultados siguen una tendencia natural o contienen errores en su generación. Con un algoritmo de pseudo-ruido, los campos escalares debiesen ser "suaves", es decir, sin rupturas o quiebres en la continuidad de los valores en el espacio.

5.2.1. Modelos esféricos

Para las pruebas de rendimiento se generaron modelos esféricos con una cantidad de vértices parametrizable. Para ello se usó un script en Python que genera un archivo con formato *node* o *ply* de forma manual con una cantidad de puntos dada en la entrada del script. Para simplificar las cantidades de puntos, se hizo que el script aceptara el exponente de una potencia de dos de la cantidad de vértices a generar.

Para generar los puntos de forma aleatoria se ocupó una función de distribución uniforme en el intervalo [-1,1] para la posición en el eje X, Y y Z. Una vez que se comprueba que el punto pertenezca a la bola unitaria con centro en el origen se agrega al archivo de salida. Este método de generación de puntos usando coordenadas cartesianas es simple y evita problemas con distribuciones no uniformes de puntos en el espacio que surgirían al trabajar con coordenadas esféricas. Además, si se trabaja con coordenadas esféricas se evitan cálculos adicionales de funciones trigonométricas.

Una vez que se genera un archivo de salida, éste sólo contiene vértices. Para unirlos se

ocupa el software abierto *tetgen* que genera una triangulación de Delaunay en 3D a partir del conjunto de puntos. Con esto se generan tetraedros que cumplen las condiciones de Delaunay usando todos los puntos del conjunto de entrada. Esto resulta en la adición de caras y poliedros a la nube de puntos, dando paso a la posterior adición de propiedades escalares.

Finalmente se tiene un método de generación de modelos fácil de aplicar en el que la cantidad de vértices es una variable controlable, mientras que la cantidad de tetraedros no lo es. Sin embargo, la relación entre ambas cantidades está relacionada en forma lineal como se ve en el gráfico 5.1.

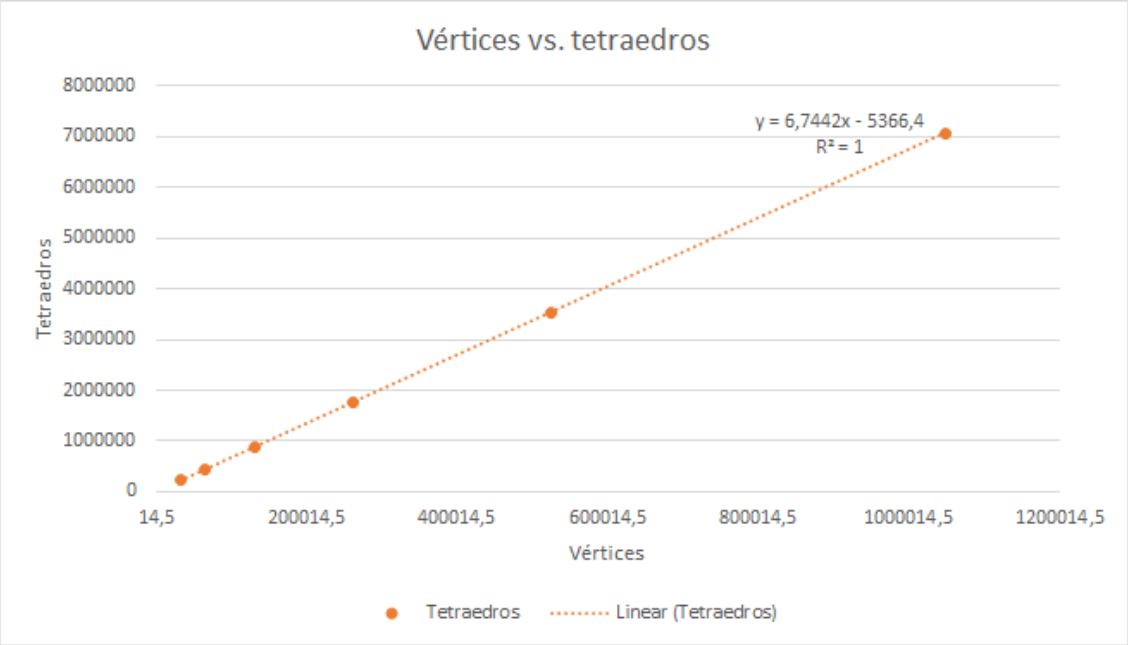

Figura 5.1: Las cantidades de vértices corresponden a 2^x para $x \in \{15, 16, 17, 18, 19, 20\}$.

V	2^{15}	2^{16}	2^{17}	2^{18}	2^{19}	2^{20}
POLÍGONOS	1.642	2.314	3.262	4.562	6.430	9.086
POLIEDROS	217.680	437.402	878.484	1.760.637	3.528.299	7.068.012
TAMAÑO(MB)	9,3	18,8	38,7	81	166	337

Tabla 5.1: Composición de esferas de puntos: V es la cantidad de vértices de cada esfera.

5.2.2. Modelos de cascarón

Con los modelos esféricos detallados en la sección anterior se pretende probar el rendimiento de los algoritmos para generar isosuperficies. Para probar la generación de isolíneas se deben ocupar modelos de superficie. Si bien Camarón puede generar isolíneas en modelos volumétricos, ParaView no tiene la opción de hacerlo, por lo que es necesario entregarle un modelo de superficie a Camarón.

De manera similar a la generación de esferas, esta vez se generan cascarones esféricos usando una variante del script para generar esferas. Cada vez que se genera un punto al

interior de la esfera, su vector de posición se normaliza para llevarlo a la superficie de la esfera (las esferas generadas son unitarias). Esto aprovecha la rapidez en la generación de puntos del script para esferas volumétricas y agrega un pequeño sobrecosto al realizar una normalización. La distribución resultante puede verse en la figura 5.2.

Una vez que se generan los puntos ubicados en el cascarón esférico unitario es necesario unirlos mediante polígonos para obtener una superficie. Esta vez se ocupa el software *QHull*[4], el cual se especializa en la generación de cerraduras convexas para conjuntos de puntos en 2D, 3D o más dimensiones. La generación se realizó con el comando:

```
qconvex s o TI pointcloud TO shellsurface
```

donde `pointcloud` es el archivo de entrada con la nube de puntos y `shellsurface` el nombre de archivo de salida. Los archivos generados por qhull presentan 2 problemas que deben ser corregidos antes de ser leídos por Camarón.

V	2^{15}	2^{16}	2^{17}	2^{18}	2^{19}	2^{20}
POLÍGONOS	65.532	131.060	262.102	523.992	1.046.236	2.080.346
TAMAÑO(MB)	2,5	5,2	10,6	22,0	44,6	89,6

Tabla 5.2: Composición de cascarones de puntos: V es la cantidad de vértices de cada cascarón. El tamaño de la malla corresponde a aquel del formato PLY.

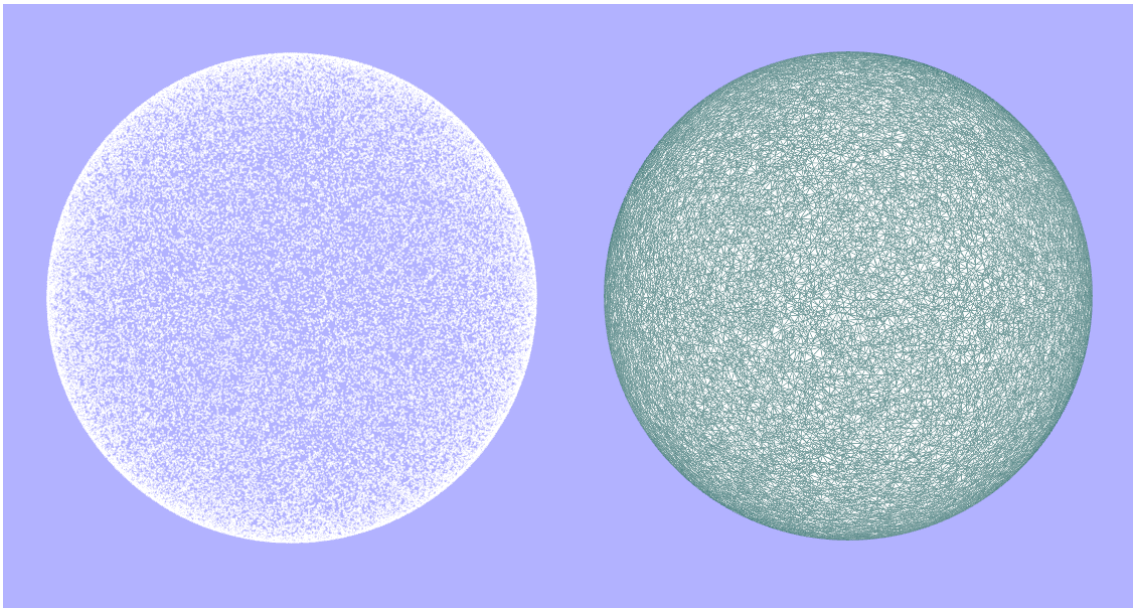

Figura 5.2: (Izquierda) Un cascarón de 2^{16} puntos. (Centro) Cerradura convexa de la nube de puntos de la izquierda.

5.2.3. Generación de ruido

Para generar valores escalares asociados a cada vértice se podría haber usado una función pseudoaleatoria con alguna distribución determinada, pero habría dado como resultado quiebres abruptos a lo largo de la malla. Para evitar esto hay algoritmos que generan un ruido "suave". Dentro de los algoritmos de ruido[12], uno de los más usados históricamente es *Perlin Noise* por su relativa rapidez y buenos resultados. Este algoritmo fue creado en 1983 por Ken Perlin y le valió un premio de la academia por su logro técnico. En 2001 el uso del ruido Perlin fue sustituido en algunas aplicaciones por su versión más rápida llamada *Simplex Noise*, el cual logra resultados comparables con su antecesor.

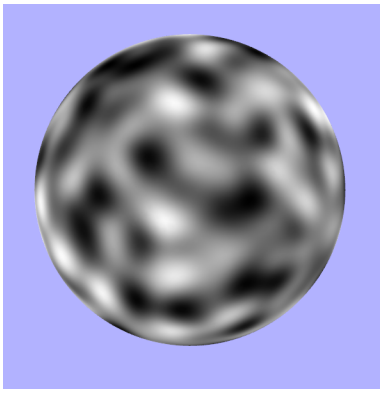

Figura 5.3: Ruido Simplex 3D visto en la superficie de una esfera.

Para los fines de este trabajo, se usó el ruido simplex, dado su mejor rendimiento. Sólo se necesitaba una función de ruido pseudo-aleatorio que no tuviera cambios abruptos como lo sería un algoritmo de generación de valores aleatorios usual.

La generación de ruido se realizó con un script escrito en el lenguaje Python por su simplicidad, usando la biblioteca *noise*[11] disponible de forma abierta. El script creado lee un archivo de entrada correspondiente a un modelo tridimensional y genera otro archivo con el ruido pseudoaleatorio agregado. Para soportar distintos formatos de entrada y salida se crearon clases encargadas de leer los datos y generar los archivos de salida con el mismo formato de entrada. Por simplicidad no se implementó la lectura y escritura por separado, por lo que no es posible hacer la lectura desde un formato y escribir el resultado en otro.

5.2.4. Otros modelos

Para ejemplificar los resultados de las implementaciones de rendering de isolíneas se usa el modelo *dragon* del repositorio de modelos escaneados de la Universidad de Stanford [2]. Cuenta con 566.098 vértices y 1.132.830 triángulos.

En cambio, para los resultados de visualización de isosuperficies se usa el modelo *pmdc* disponible como ejemplo en el software *TetView*. Cuenta con 27.196 vértices, 39.830 caras y 109.189 tetraedros.

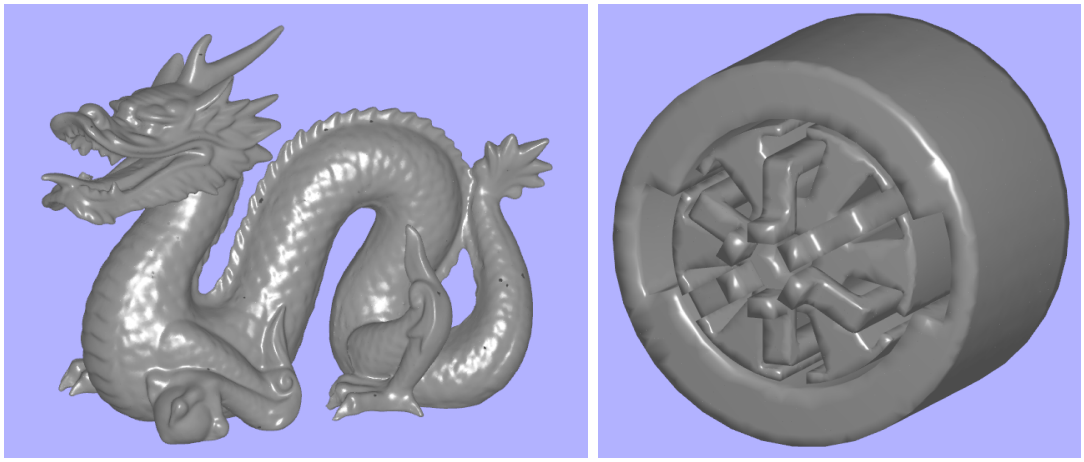

Figura 5.4: Modelos *dragon* y *pmdc*

5.3. Mediciones de tiempo

En el caso de las isolíneas se hacen pruebas de las implementaciones *Geometry Shader*, *Fragment Shader* y *transform feedback*. Hay que notar que en las dos primeras no existe una clara separación entre la generación de las isolíneas y su visualización en pantalla, pues el algoritmo engloba ambos pasos como uno solo. Para las comparaciones que usan dichas implementaciones se suman los tiempos de generación y dibujado de otros métodos en los cuales ambos pasos están claramente separados.

Para el caso de las isosuperficies la única implementación *transform feedback* presenta una clara separación entre generación y visualización, por lo que se toman medidas de ambas fases para luego hacer comparaciones con el resto.

En las mediciones de tiempos de generación de isosuperficies o isolíneas se hacen 3 mediciones por cada isovalor en cada modelo en Camarón y ParaView. Las mediciones fueron hechas de forma manual cambiando los parámetros cada vez. Se hicieron pruebas de automatización del proceso, pero mostraron resultados notablemente diferentes, por lo que se conserva la forma manual.

En el caso de la etapa de visualización se toma un promedio de entre 100 mediciones de los tiempos que demora cada implementación en realizar el dibujado en pantalla. Se exige que se dibuje el modelo completo para evitar que alguna implementación obtenga mejor rendimiento por hacer culling de geometrías[1]. Cada operación de dibujado en pantalla es gatillada por rotaciones al modelo.

ParaView tiene un pipeline que procesa los datos de geometrías de entrada a través de filtros que permiten emitir nuevas geometrías o cambiar sus propiedades antes de ser visualizadas. Los filtros en ParaView son usados en un paso aislado previo a la visualización y por lo tanto, existe una clara separación entre el procesamiento de geometrías y su posterior dibujado en pantalla.

[1]Optimización mediante la cual ciertas geometrías no son procesadas por estar fuera del marco de visión del observador. Esto conlleva a un mejor rendimiento.

Dado que el campo escalar de las esferas de prueba está en el rango $(-1, 1)$ para medir el tiempo de generación de isosuperficies se generó una isosuperfice a la vez en el rango de valores $[-1, 1]$ en pasos de a 0,1. Esto permite tener una serie de mediciones que recorren el rango completo del campo escalar a una resolución suficiente para permitir hacer análisis. Se incluyeron los valores -1 y 1 dentro del rango de mediciones para medir el comportamiento de los algoritmos cuando la equipotencial no pasa por el modelo.

En ParaView

En ParaView las mediciones de tiempo de cada proceso son hechas automáticamente de manera interna en el software y pueden ser vistas en una ventana llamada *Timer Log* accesible desde el menú *Tools*.

En la figura 5.5 se muestran los eventos registrados luego de generar una isosuperficie, mover la vista y actualizar las isosuperficies a nuevos valores. Se distinguen las tres acciones tal como están destacadas en la imagen de referencia. Como se puede observar, los eventos están desplegados como una representación en árbol, siendo los sub-eventos mostrados con indentación por la izquierda.

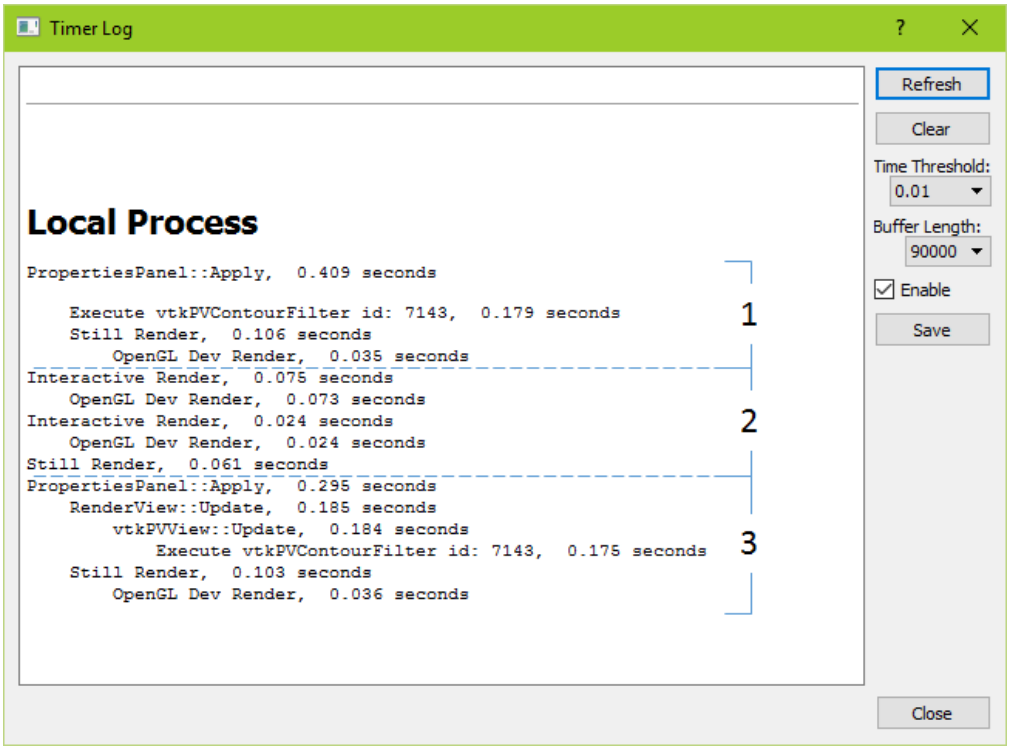

Figura 5.5: Se muestra cada evento registrado con sus eventos descendientes.

La primera sección contempla la aplicación del filtro *vtkPVContourFilter* responsable de generar por primera vez la malla de isosuperficies junto con el dibujado inmediato de la misma. En ParaView la generación y actualización de una malla por acción de filtros son considerados eventos separados, y por lo tanto las mediciones podrían variar en ambos casos. Se considera para este trabajo que la operación medida será la de actualización de una malla,

descartando la construcción de la misma la primera vez.

La segunda sección muestra los eventos correspondientes a los cambios de vista de un mismo modelo en donde hay dos tipos de rendering. El rendering interactivo es un tipo de dibujado que pretende sacrificar calidad en el modelo para obtener una mejor fluidez al hacer movimientos. Esto se logra rendereando una malla de menor tamaño. Una vez que el usuario deja de mover el modelo se gatilla un rendering estático que ocupa todos los elementos del modelo actual para generar una imagen estática de mayor detalle.

El rendering interactivo puede no sacrificar calidad si el tamaño de la malla es pequeño. El tamaño de la malla (medido en `MB`) sobre el cual el rendering interactivo sacrifica calidad es un parámetro ajustable en ParaView. En la configuración de la aplicación, dentro de la pestaña *Render View* el parámetro *LOD Threshold* permite escoger el límite de tamaño de una malla sobre el cual el modelo se mostrará en baja calidad. Dado que Camarón no cuenta con esta opción, y para hacer comparaciones equitativas se estableció un límite arbitrariamente alto en la opción *LOD Threshold*. Esto evita que se ocupe el rendering interactivo de baja calidad.

La tercera y última sección muestra una actualización de la malla de isosuperficies en donde hay dos subsecciones. La primera de ellas es la actualización de la malla, teniendo como paso interno la ejecución del filtro *vtkPVContourFilter* cuyo tiempo registrado es un tanto menor al tiempo completo de actualización. Si bien se podría considerar el tiempo de generación de una malla de isosuperficies como aquel del filtro ejecutado internamente, no se correspondería con lo percibido por un usuario final. Debido a esto, dicho tiempo de generación es considerado como el tiempo total de actualización mostrado en el registro de evento bajo el nombre `RenderView::Update`.

En Camarón

La clase `CrossTimer` es ocupada para medir tiempos de ejecución en la aplicación. Hasta antes de este trabajo, la clase implementaba métodos de medición de tiempo estándares de C++, sin embargo, en la plataforma Windows es posible que algunas implementaciones de compilador ocupen un medidor de tiempo de baja resolución al declarar uno de alta resolución. Las funciones *QPC* de medición de tiempo en plataformas Windows fueron agregadas para compilaciones en dicha plataforma con el objetivo de permitir mediciones de tiempo más precisas. Ellas ocupan un contador de instrucciones de procesador disponible desde hardware, dando como resultados de medición del orden de los microsegundos. Cabe destacar que las funciones *QPC* sólo están disponibles en la plataforma Windows y para el resto de las plataformas se ocupan las funciones de medición usuales, las cuales dependen del compilador correspondiente.

Las funciones de dibujado en OpenGL no son bloqueantes, y por lo tanto, retornan de manera casi inmediata dejando trabajo encolado. Para hacer mediciones de tiempo es necesario determinar cuándo se terminan las operaciones que se están midiendo y para ello se ocupa la función `glFinish()`. Esta función es bloqueante y retorna una vez que todas las operaciones encoladas hayan sido ejecutadas. Se ubicó esta función a antes de empezar y terminar cada medición.

5.3.1. Ajustes previos

Para asegurar que ambos software estén en ventajas similares, se ajustan opciones que permiten equiparar las operaciones ejecutadas en cada aplicación. Se pretende así tener una comparación que ahonda más en las funcionalidades estudiadas y no otras que puedan interferir.

Principalmente los ajustes son hechos en ParaView. Este software contiene funcionalidaes adicionales que pueden exigir un mayor tiempo a la hora de realizar mediciones. En el caso de generación de isosuperficies se desactiva la opción *Compute normals* disponible en la ventana de configuración del filtro *Contour*. Lo que hace esta opción es calcular las normales promediadas en los vértices de la malla resultante. Si bien se calcula la normal de cada cara para aplicar un modelo de iluminación final, el cálculo de normales promediadas es una operación más costosa pero permite un modelo de iluminación final más suave y estético. Para los fines de este proyecto no es necesario el cálculo de normales promediadas y por lo tanto se deja fuera.

5.4. Isosuperficies

En esta sección se exponen las mediciones de tiempo de ParaView y Camarón en generación de superficies. Esto incluye tiempos de generación y fluidez de interacción con el modelo. Las mediciones en esta sección fueron hechas con las esferas volumétricas de pruebas descritas en la sección 5.2.1. Para las mediciones de fluidez en la interacción se usó la esfera de 2^{19} vértices para poner ambas aplicaciones en estrés. El modelo de 2^{20} vértices no fue usado para la comparación por limitaciones de memoria RAM al cargarlo en Camarón.

5.4.1. Tiempo de rendering de isosuperficies

En los resultados de interacción de ParaView los cuadros por segundo partieron en cerca de 30 y fueron decayendo bastante hasta llegar a casi 5 con 10 isosuperficies. El rendimiento se ve reducido para una cantidad mayor de isosuperficies porque es necesario realizar la intersección de más geometrías con la equipotencial. El rendimiento percibido, sin embargo, fue bastante peor y eso se puede ver en la tabla 5.3.

Fps medidos	14,28	2,59	21,73	2,46	14,49	2,49	32,25	2,43	26,31	2,39

Tabla 5.3: Muestra de medición de fps en ParaView con 8 isosuperficies. Se puede ver que la cantidad de fps luego de cada medición varía de forma alternada.

ParaView rendereaba una imagen en un tiempo largo para luego renderear otra vez en un tiempo corto de manera alternada. Finalmente el rendimiento percibido es el de los tiempos largos de rendering, lo que no se ve reflejado en el gráfico 5.6.

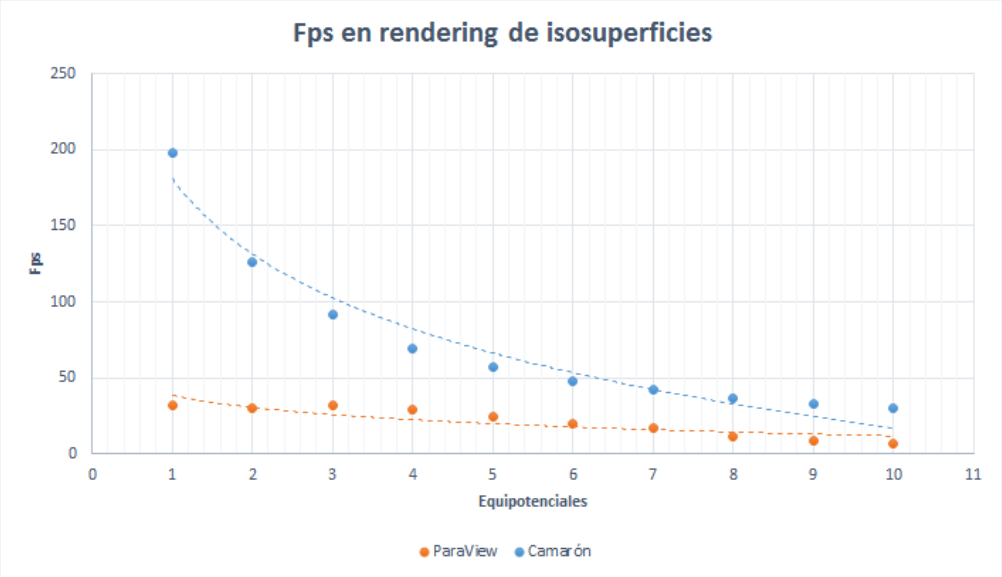

Figura 5.6: Fps en interacción con modelo de isosuperficies en ParaView y Camarón.

En cuanto a Camarón, éste demostró un rendimiento hasta 6 veces mayor en velocidad de interacción. Una vez que se interactuó con 5 isosuperficies en adelante la tasa de cuadros bajo de los 60Hz.

5.4.2. Tiempos de generación de isosuperficies

Se registraron los tiempos de actualización de la malla de isosuperficies y se presentan en la figuras 5.7 y 5.8. Como se ve en ella los mayores tiempo de generación de isosuperficies se experimentan en los valores medios, en donde hay una mayor cantidad de poliedros cortados por la isosuperficie resultante. Los mayores tiempos se encuentran cerca de los 1,8 segundos, lo que es bastante lento y se nota en la práctica.

V	ParaView	Camarón	ParaView/Camarón
2^{15}	64,67	37,07	1,74
2^{16}	118,00	66,92	1,76
2^{17}	216,71	122,67	1,76
2^{18}	408,33	220,137	1,85
2^{19}	787,71	294,60	2,67
2^{20}	1532,71	N/A	

Tabla 5.4: Comparación de tiempos en milisegundos entre ParaView y Camarón en la muestra de mallas esféricas. V es la cantidad de vértices de cada malla. Cada tiempo corresponde al promedio de las mediciones en el rango completo del campo escalar. El valor N/A no pudo ser obtenido por limitaciones de memoria RAM.

En el caso de Camarón se puede destacar que los tiempos de generación de isosuperficies:

- Siguen una tendencia similar a los tiempos de ParaView. Los tiempos de generación

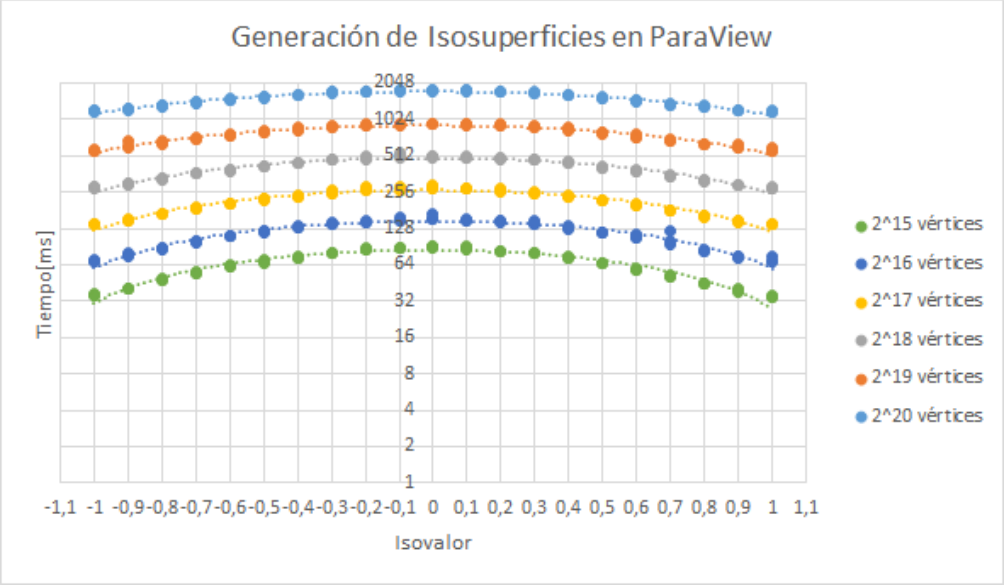

Figura 5.7: Tiempos de generación de isosuperficies en ParaView

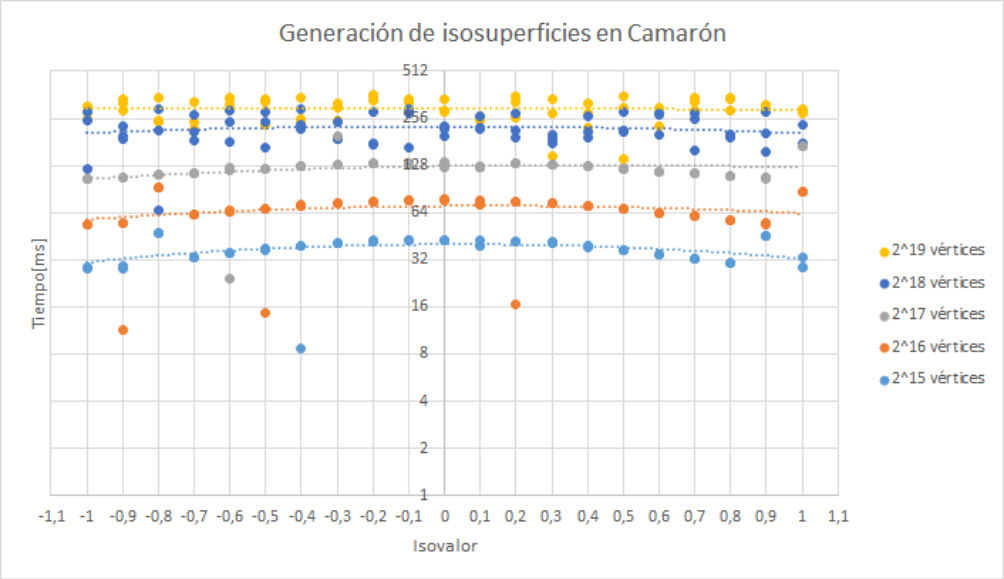

Figura 5.8: Tiempos de generación de isosuperficies en Camarón

se incrementan cuando la equipotencial interseca una mayor cantidad de poliedros, lo cual se presenta en el centro del rango de valores de la propiedad escalar.

- Los tiempos de generación son hasta **2,67** veces menores que en ParaView (ver tabla 5.4).

5.5. Isolíneas

A continuación se exponen los resultados de las mediciones hechas a ParaView, y las implementaciones *Geometry Shader* y *Transform Feedback* de Camarón. Se muestran resultados

de *generación* de isolíneas en cada cascarón esférico de prueba descritos en la sección 5.2.2 y resultados de *interacción* con el modelo generado.

Todos los resultados en que se mide la cantidad de fps en la interacción con el modelo resultante fueron obtenidos desde el modelo de cascarón de esfera de 2^{20} vértices para poner a prueba la eficiencia de las aplicaciones con modelos pesados.

5.5.1. Tiempo de rendering de isolíneas

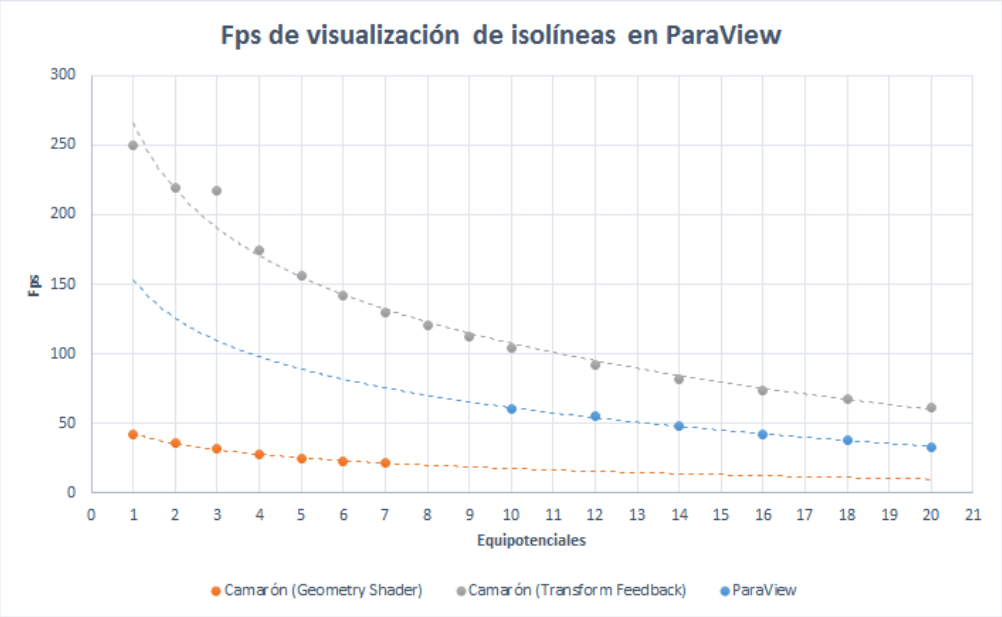

Figura 5.9: Fps en interacción con modelo de isolíneas en ParaView y Camarón en sus implementaciones de métodos Geometry Shader y Transform Feedback. La curva de interpolación de ParaView incluye lo que mediría ParaView quitando la limitación de 60 fps que tiene el programa.

En ParaView la interacción con un modelo de isolíneas comienza a degradarse a partir de aproximadamente 10 isolíneas (notar que esto es válido para el cascarón esférico de 2^{20} vértices) llegando a casi la mitad de la tasa de refresco de una pantalla normal (60Hz). En ParaView la cantidad de fps de interacción está limitada por la tasa de refresco del monitor, por lo que para menos 10 isolíneas la cantidad de fps se mantiene en 60.

La implementación del método Geometry Shader resultó ser no ser muy competente. Como se ve en la figura 5.9, a partir de la primera isolínea desplegada, el rendimiento no alcanza la tasa de refresco habitual de un monitor y comienza a degradarse de forma logarítmica hasta llegar a cerca de 25 fps con 7 isolíneas. Se presume, usando información de diversas fuentes, que el cuello de botella se encuentra en la utilización de un *Geometry Shader* con un número de vértices de salida muy alto, el cual cambia la cantidad de geometrías en el pipeline en forma dinámica y eso afecta el rendimiento de la aplicación.

Los tiempos de interacción en la implementación del método Transform Feedback[2] son bastante buenos, y sólo llegando a las 20 isolíneas desplegadas simultáneamente los cuadros por segundo medidos bajan hasta la tasa de refresco habitual de un monitor.

5.5.2. Tiempos de generación

El método Geometry Shader fue el primer intento de implementación y muestra resultados comparativamente bastante mejores que ParaView. El tiempo máximo de generación bordeó los 160ms, con lo que permite un trabajo fluído con modelos bastante pesados.

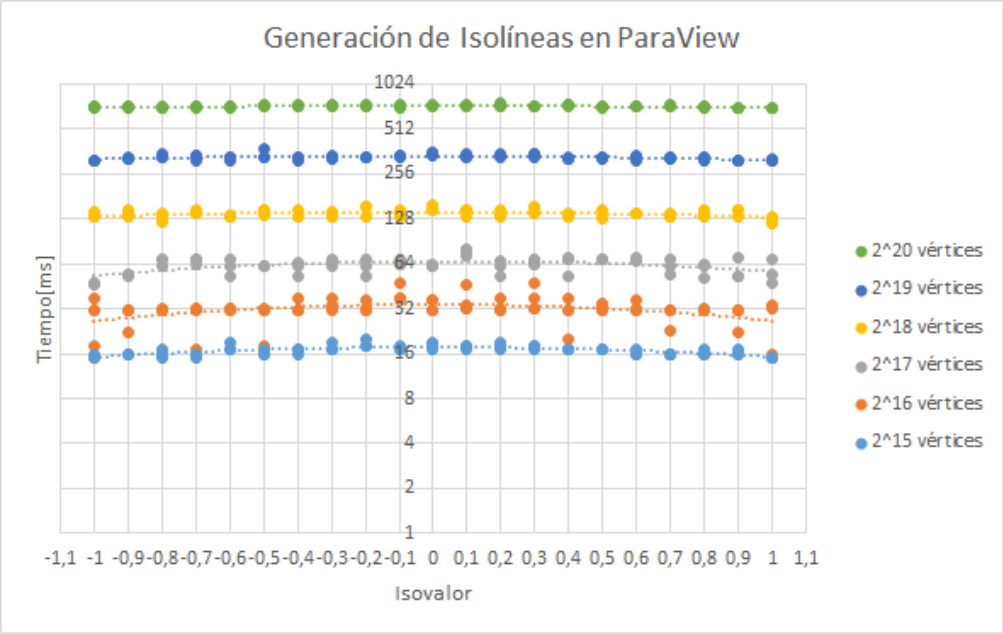

Figura 5.10: Tiempo de generación de isolíneas en ParaView.

Los peores tiempos alcanzados en ParaView bordean los 0,764 segundos, lo que en la práctica es notorio. Por otro lado los tiempos de generación de los cascarones de 2^{17} hacia abajo presentan tiempos de generación aceptables y poco perceptibles ($< 100ms$) para un ambiente de trabajo dinámico.

Por otro lado se aprecia que las curvas de interpolación del gráfico de ParaView (fig. 5.10) no son tan pronunciadas como en la generación de isosuperficies del mismo software, por lo que el tiempo de generación de isolíneas en ParaView es por lo general casi constante e independiente de la cantidad de geometrías cortadas por la equipotencial.

El mayor tiempo de generación del método Transform Feedback fue de 288ms, superando en tiempo al método Geometry Shader. Esto implica que el paso de generación de isolíneas es más costoso, sin embargo, como se ve en la figura 5.9 los tiempos de interacción son mejores.

[2]Recordar de que a pesar de que se le llama en este informe método Transform feedback, éste método usa también un geometry shader.

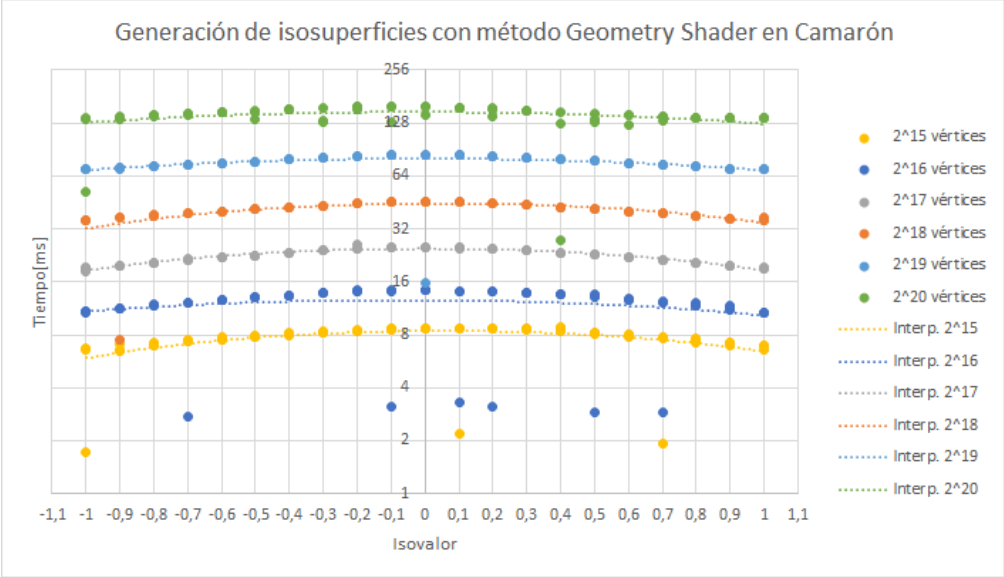

Figura 5.11: Tiempo de generación de isolíneas con método Geometry Shader en Camarón

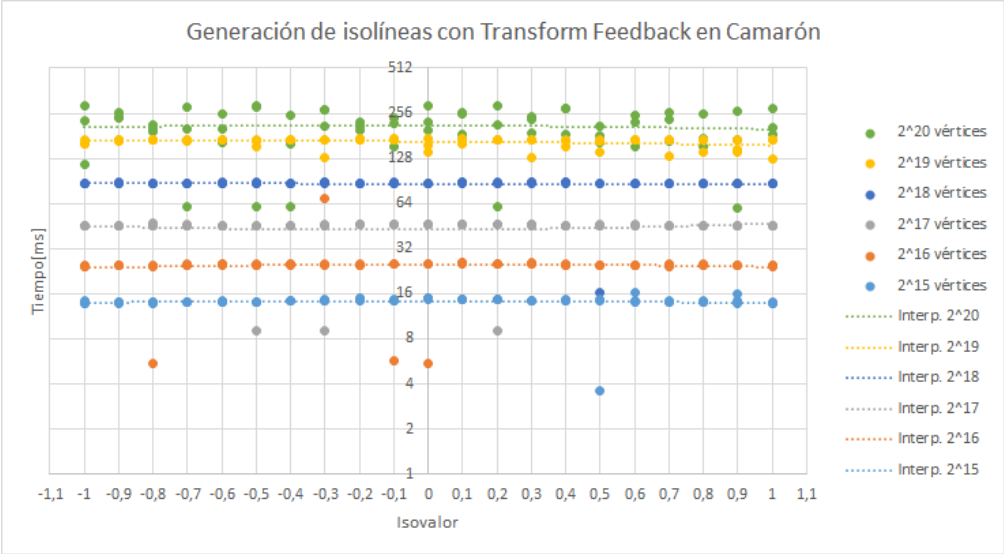

Figura 5.12: Tiempo de generación de isolíneas con método Transform Feedback en Camarón.

El método Transform Feedback de generación de isolíneas demuestra, entonces, un tiempo relativamente aceptable de generación de geometrías y un muy buen tiempo de rendering.

5.6. Uso de memoria

Para todos los gráficos de esta sección: Las mediciones para isosuperficies fueron hechas con la esfera de prueba de 2^{19} vértices y las de isolíneas con el cascarón esférico de 2^{20} vértices. Para el caso de visualización de isolíneas se usó la implementación Transform Feedback.

Para medir el uso de memoria RAM se usó el Monitor de Recursos de Windows tanto

para Camarón como ParaView. Si bien la cantidad de memoria física pareciera ser un buen indicador, no da cuenta de la cantidad de memoria en swap o comprimida[3]. Por lo tanto se usó como indicador la cantidad de memoria listada en la columna *commit* que se corresponde con la memoria virtual reservada por el proceso. Este valor puede tener una cierta diferencia con respecto a lo que realmente está ocupando el proceso en memoria virtual, pero se acerca bastante más que otros indicadores.

La memoria RAM es un aspecto que se pretendió sacrificar desde un principio en Camarón en pos de darle mayor importancia a la velocidad a la aplicación y así lo demuestra la figura 5.13 que compara el uso de memoria RAM entre Camarón y ParaView. Sin embargo, el uso de RAM elevado no va ligado a las equipotenciales mostradas sino al modelo original cargado, lo que se observa en la columna de 0 equipotenciales en el caso de isosuperficies e isolíneas. Cabe destacar que con 0 equipotenciales se muestra el modelo con sombreado simple en ambos programas. Con la introducción de isosuperficies o isolíneas se cambia el modo de rendering para visualizarlas.

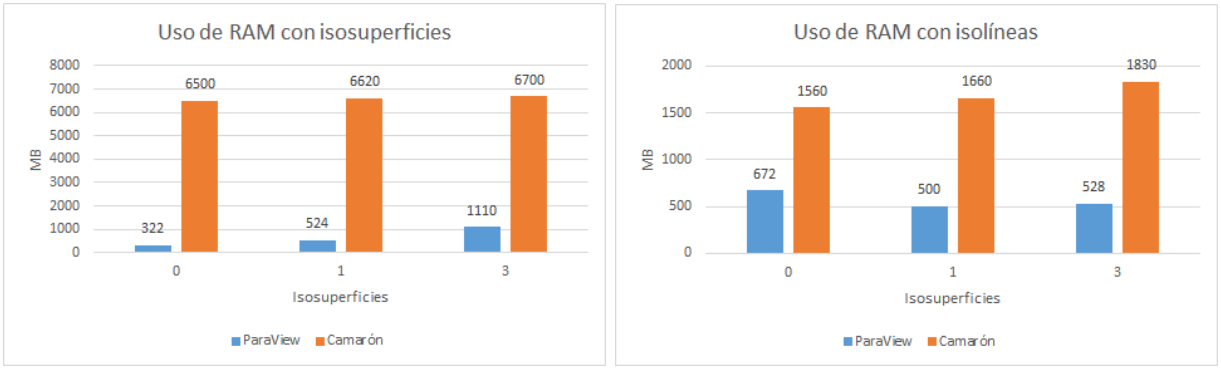

Figura 5.13: Izquierda: Uso de memoria RAM con isosuperficies. Derecha: Uso de memoria RAM con isolíneas.

Las tablas 5.13 muestran las diferencias en términos porcentuales y absolutos en cuanto al uso de memoria RAM. Como se ve en ellas los cambios en uso de memoria en general aumentan al mostrar las nuevas geometrías asociadas con isosuperficies o isolíneas, sin embargo para el caso de isolíneas en ParaView se produce una reducción del uso de RAM. Esto se podría explicar si ParaView tuviera un uso mayor de memoria mientras más geometrías sean dibujadas en pantalla. Es posible que el modelo se cargue parcialmente o de forma comprimida al abrirse en el software y que al momento de visualizarlo, se ocupe más memoria para almacenar propiedades el modelo a mostrar. Cuando se muestran isolíneas se reduce considerablemente la cantidad de geometrías en pantalla, mientras que para el caso de isosuperficies, no se reduce dicha cantidad porque las isosuperficies tienen una mayor cantidad de geometrías (visibles) que el modelo original en el caso de las esferas de prueba.

En Camarón el uso de memoria va en aumento independiente del número de geometrías en pantalla, pues el modelo original siempre es mantenido en RAM con todas sus propiedades. Luego, el uso de RAM con 1 o 3 equipotenciales equivale a lo usado por el modelo original

[3]Windows en su versión 10 integró compresión de memoria primaria. Esto permite que páginas en memoria sean comprimidas para ahorrar espacio y no tener que llevar páginas a memoria secundaria.

y los aumentos medidos son presuntamente producto de errores de medición a pesar de que muestran una cierta relación con la cantidad de equipotenciales en pantalla. No habría en teoría otra forma de explicar dichos aumentos. Cabe recordar que la cantidad de memoria medida es la cantidad total de memoria virtual que asigna el sistema operativo a la aplicación y no refleja exactamente lo que está usando la aplicación, sino más bien un estimado.

Isosuperficies	Camarón	ParaView
1	+1,8 %	+62,7 %
3	+3,1 %	+244,7 %

Isolíneas	Camarón	ParaView
1	+6,4 %	−26,6 %
3	+17,3 %	−21,4 %

Tabla 5.5: (Arriba) Comparación en uso de RAM visualizando isosuperficies. (Abajo) Comparación de uso en RAM visualizando isolíneas. Los resultados hacen la comparación de 1 y 3 equipotenciales con respecto a 0 equipotenciales, en donde 0 corresponde a una vista del modelo con un sombreado simple.

Para las mediciones de memoria VRAM o memoria de video se ocupó el software GPU-Z[4] que entrega dichas lecturas además de otros datos sobre los chips gráficos en el sistema. El programa reporta dos tipos de VRAM: Dedicada y dinámica. La primera es la memoria que efectivamente se encuentra en el dispositivo gráfico y se lista junto a las especificaciones de las tarjetas de video. La segunda es memoria RAM que usa el driver de video.

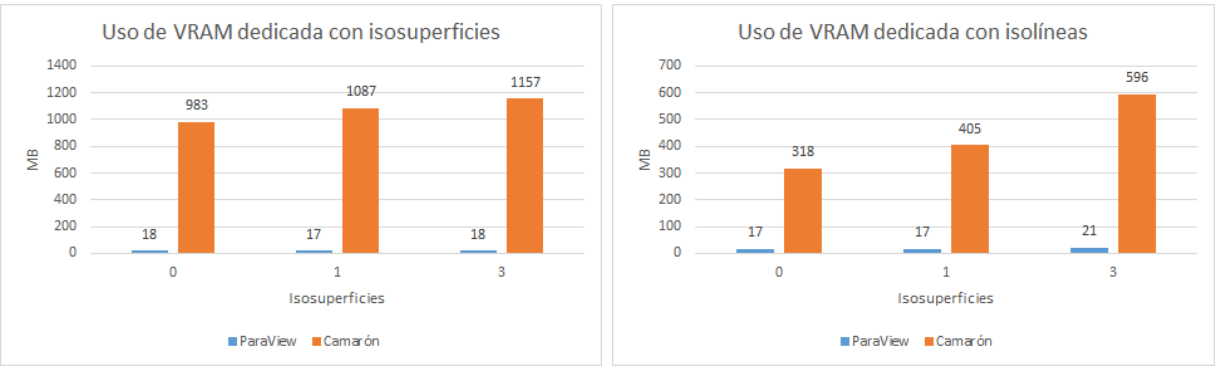

Figura 5.14: Izquierda: Uso de memoria VRAM dedicada con isosuperficies. Derecha: Uso de memoria VRAM dedicada con isolíneas.

Las figuras 5.14 y 5.15 muestran los resultados de las mediciones hechas con GPU-Z. Se puede ver que el uso de VRAM dedicada es alto en Camarón por tener todas las geometrías en el dispositivo gráfico, mientras que ParaView las mantiene en RAM. Un resultado curioso es el uso de VRAM dinámica de ParaView en el caso de las isolíneas. Al parecer lo que está sucediendo es que el modelo cargado ocupa una mayor cantidad de memoria que las isolíneas, y al mostrar las isolíneas se descarga el modelo original, efectivamente reduciendo el uso de memoria.

[4]URL: https://www.techpowerup.com/gpuz/ (visitado 27-03-2016)

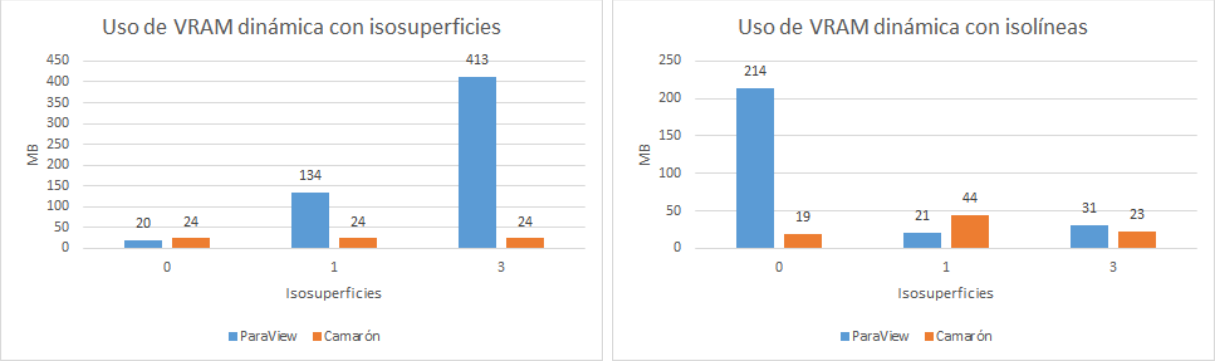

Figura 5.15: Izquierda: Uso de memoria VRAM dinámica con isosuperficies. Derecha: Uso de memoria VRAM dinámica con isolíneas.

5.7. Aspecto visual

Se muestran a continuación ejemplos del modelo *dragon* y *pmdc* con distintos renderers que incluyen los implementados en este trabajo relativos a visualización científica.

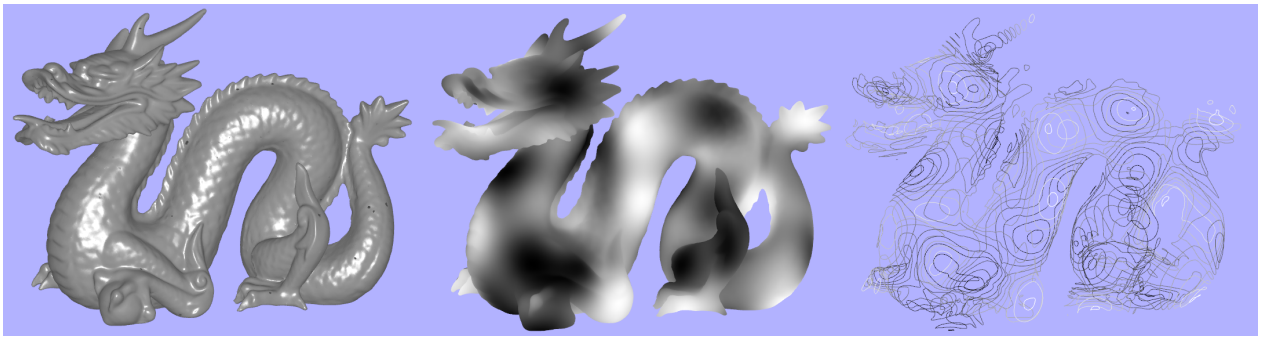

Figura 5.16: Ejemplo de campo escalar en malla de superficie. Izquierda: sombreado phong. Centro: Campo escalar en tono de grises. Derecha: 10 isolíneas con isoniveles equiespaciados

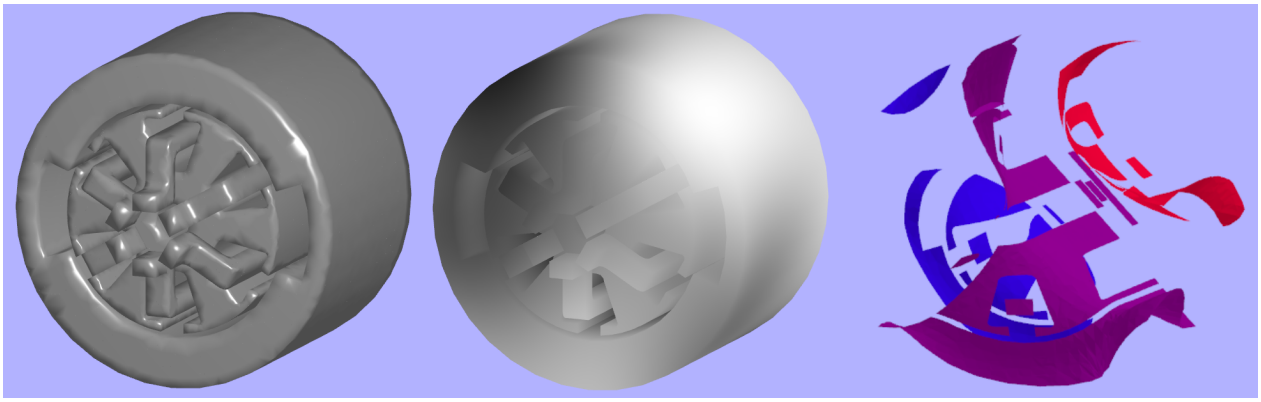

Figura 5.17: Ejemplo de campo escalar en malla de volumen. Izquierda: sombreado phong. Centro: Campo escalar en tono de grises. Derecha: 3 isosuperficies con isoniveles equiespaciados

Conclusión

Camarón fue extendido para albergar nuevas formas de interactuar con datos ligados a la visualización científica, dándole una mayor chance de entrar a competir con otro software del área. Esto permite que potencialmente un sujeto que investigue en el área de simulaciones usando el método de elemento finito encuentre interés en Camarón al proveer más herramientas disponibles.

No sólo se amplió la arquitectura para manejar campos de propiedaes, sino que el procesamiento de los mismos fue llevado a una GPU para permitir un máximo rendimiento. Al ver las mejoras en rendimiento que se logran se ayuda a fomentar el uso de una GPU en programas de este tipo, aprovechando mejor los recursos que ofrece un computador.

Si bien Camarón resultó ser competente en términos de agilidad en el manejo de los datos, queda trabajo que hacer en el apartado de memoria. Un investigador puede verse limitado por la cantidad de RAM o VRAM en su sistema y no podría abrir modelos muy pesados. En base a experimentación, el software presenta serios problemas para abrir archivos del orden de 500MB hacia arriba.

Por otro lado, el competidor ParaView permite abrir mallas de mayor tamaño con coste en rendimiento de algunas funciones. Si bien esto último puede verse como un aspecto negativo, son varios los aspectos positivos que lo hacen una opción tentativa. ParaView es un software mucho más completo a la hora de entregar herramientas, ofrece capacidades de scripting, un pipeline bien documentado y soporte de muchos desarrolladores. Esto no significa que el proyecto Camarón no tenga mayor sentido, sino más bien, el aporte de Camarón radica en introducir nuevas formas de desarrollo que pueden beneficiar otras aplicaciones. Por otro lado Camarón está en vías de desarrollo y todavía falta para que sea un software de uso cotidiano en la industria e investigación, tiene aspectos que lo hacen interesante como el análisis de propiedades de geometrías y el uso de las capacidades de paralelismo de una GPU.

Cualquier tarea intensiva en procesamiento debe intentar ser modelada como un algoritmo en paralelo para permitir escalar el tamaño de la entrada. Con la cantidad de datos generados en la actualidad es fácil generar un modelo altamente detallado que necesita ser visualizado y estudiado. Es por ello que las unidades gráficas disponibles en los computadores actuales están siendo usadas cada vez más para paralelizar tareas que de otra forma se ejecutarían de manera muy lenta.

En las aplicaciones de procesamiento de datos el uso de la GPU cobra mayor protagonismo por las facilidades que se integran en las bibliotecas gráficas y de computación en chip gráficos

como CUDA o OpenCL. Como también se pudo ver en este trabajo, la biblioteca OpenGL también permite el procesamiento de datos; por ejemplo, para generar equipotenciales. Las tareas de procesamiento de geometrías o píxeles van migrando a la GPU con tal de hacer más fluído el trabajo de investigadores, diseñadores y otros.

Si bien, el uso de un dispositivo gráfico se ha estandarizado y ha logrado altos niveles de compatibilidad abarcando hardware de distintas marcas, su uso aún no es masivo para la mayoría de las tareas de procesamiento de datos, y por lo tanto aún falta desarrollo en ese aspecto. Sin embargo, hay áreas donde sí se han aprovechado de sobremanera las capacidades de paralelismo que ofrece una GPU común. En el ámbito de deep learning se entrenan redes neuronales para hacer clasificadores que son entrenados en base a un gran conjunto de entradas. Un ejemplo de ello son las *redes neuronales convolucionales* (CNN), cuyo propósito es clasificar imágenes por su contenido. Este tipo de redes neuronales tiene varias capas de *neuronas* simuladas, y el proceso de entrenamiento para lograr una buena precisión en la clasificación toma mucho tiempo, incluso con el uso de varias GPUs, por lo que se está investigando como hacer los algoritmos de entrenamiento de redes neuronales más eficientes.

La biblioteca de visualización VTK ha enfocado su desarrollo en el último tiempo a usar de mejor manera la GPU con tal de permitir que el procesamiento de geometrías sea más rápido. Esto conllevaría a que el software basado en VTK como ParaView o VisIt goce de dicha mejora.

5.8. Trabajo Futuro

Se han identificado varias posibles mejoras a Camarón que lo podrían hacer un software bastante competente en la industria. Dento de ellas se encuentran:

- Camarón actualmente está enfocado en la utilización de RAM sobre CPU, con lo que se dejan fuera varios computadores que cuentan con poca RAM. En gran medida el uso de RAM viene dado por las relaciones de vecindad que permiten acelerar el proceso de cálculo de propiedades geométricas. Se sugiere dejar las relaciones de vecindad fuera de los objetos correspondientes a elementos geométricos y guardarlas en un punto central pendiendo del objeto `Model`. Esto evita que una relación entre dos geometrías esté duplicada en objetos que se apuntan mutuamente.
- El cargado de geometrías desde un archivo se hace a objetos en RAM para luego ser copiados a la VRAM. Luego de ello, hay información en la VRAM duplicada en RAM. Esto se evitaría si se implementara un sistema de streaming de datos desde la VRAM a los procesos que la necesiten, con lo que se adquiere eficiencia en uso de espacio en RAM.
- El sistema de rendering actual está basado en que cada renderer debe implementar el dibujado estilo malla de alambre o con modelo de iluminación por su cuenta. Además, en el caso específico de un modelo de iluminación, la interfaz gráfica que permite definir las propiedades del foco de luz deben ser implementadas y replicadas en cada renderer que necesite de un modelo de iluminación. Se podría dejar este tipo de funcionalidades

básicas en términos de rendering fuesen fácilmente accesibles para que nuevos renderers que deseen agregarlas a su estilo de visualización. Además, para estas funciones básicas se podría dejar una interfaz común para no replicar código.

- Existen actualmente áreas del software escritas muy a la manera de *C* en donde se usan cadenas de texto como arreglos de caracacteres en vez de usar el tipo `std::string` que permite un mejor manejo de memoria. En lo posible deben usarse objetos amigables con el programador para evitar limpieza innecesaria de código.

Bibliografía

[1] C++ range-based for loop. http://en.cppreference.com/w/cpp/language/range-for. Accessed: 2015-12-31.

[2] Respositorio de modelos escaneados de la stanford university. http://graphics.stanford.edu/data/3Dscanrep/. Accessed: 2016-03-25.

[3] Sitio ofical de paraview. http://www.paraview.org/. Accessed: 2015-12-21.

[4] Sitio ofical de qhull. http://qhull.org/. Accessed: 2016-03-16.

[5] Transformation matrix. https://en.wikipedia.org/wiki/Transformation_matrix. Accessed: 2016-01-04.

[6] Natalia Alarcón. Camarón. a geometric mesh display. http://users.dcc.uchile.cl/~nancy/sites/default/files/Trabajo_Dirigido_Sitio_Web/index.html, 2012. Accessed: 2016-01-03.

[7] Jasmin Blanchette. A brief history of qt. http://my.safaribooksonline.com/0131872494/pref04, 2006. Accessed: 2016-02-27.

[8] Aldo Canepa. *Camarón: Visualizador y evaluador de mallas geométricas mixtas grandes en 3D, acelerado con shaders en OpenGL*. 2012.

[9] Cyril Crassin. Opengl geometry shader marching cubes. http://www.icare3d.org/codes-and-projects/codes/opengl_geometry_shader_marching_cubes.html, 2007. Accessed: 2015-11-19.

[10] Warren Carithers Donald D. Hearn, M. Pauline Baker. *Computer Graphics with OpenGL*. Pearson, 2010.

[11] Casey Duncan. Biblioteca *noise* para la generación de ruido en python. https://pypi.python.org/pypi/noise/. Accessed: 2016-03-04.

[12] David S. Ebert. *Texturing and Modeling: A Procedural Approach*. Academic Press, 2014.

[13] Erich Gamma, Ralph Johnson, Richard Helm, and John Vlissides. *Design Patterns*. Addison-Wesley, 1994.

[14] Hubert Nguyen. *GPU Gems 3*. GPU Gems. Addison-Wesley Professional, 2007.

[15] Trygve Reenskaug. Mvc - xerox parc 1978-79. `http://heim.ifi.uio.no/~trygver/themes/mvc/mvc-index.html`. Accessed: 2016-02-19.

[16] Rephael Wenger. *Isosurfaces*. A K Peters/CRC Press, 2013.

Apéndices

Apéndice A

Format de archivo TQS

Ejemplo de formato de archivo creado para almacenar uno o más valores escalares en puntos interiores a polígonos o poliedros. En caso de ser polígonos, éstos se asumen convexos y se triangulan ocupando un punto en el centro geométrico. En caso de ser poliedros, éstos se asumen convexos y se tetraedrizan usando un punto ubicado en el centro geométrico.

A pesar de ser considerados como valores escalares por la naturaleza del problema que se intentaba resolver, se podrían ver como valores que forman un campo vectorial. Para ello se tendría que dar la posibilidad de combinar campos escalares en Camarón para formar campos vectoriales con los valores de los campos constituyentes.

El formato sugerido fue ideado originalmente por el profesor Alejandro Ortiz del departamento de ingeniería mecánica con adaptaciones por el alumno Gonzalo Infante.

Los comentarios comienzan con %.

```
TQS 1.0
name: Gauss_points
coordinates_system: barycentric
type: surface                    % or volume
elements: 784
% parametrized gauss coordinates
number_of_points: 3
gauss_coordinates:
0.5   0.5   0.0
0.0   0.5   0.5
0.5   0.0   0.5

% Gauss coordinates are given for each triangle formed from an edge to
%  the center of the polygon. In the case of a quadrilateral, there are
% 4 triangles.
dimensions: 3
number_of_points: 5
```

```
gauss_coordinates:
0.5   0.5   0.0   0.5
0.0   0.5   0.5   0.0
0.5   0.0   0.5   0.0
0.5   0.0   0.5   0.5
0.5   0.0   0.0   0.5

% Start listing values per subtriangle grouped by element. Every subtriangle is affect

% For the first elmeent we suppose that it has 5 edges, then we have 5 subtriangles.

variables: sx  sy  sxy
% element 1

100.2   45.3   60.5   %Gauss pt. 1, subtriangle 1
105.6   48.9   58.7   %Gauss pt.  2, subtriangle 1
98.7     44.7   61.4   %Gauss pt.  3, subtriangle 1

104.2   35.3   30.5   %Gauss pt.  1, subtriangle 2
102.6   78.9   58.7   %Gauss pt.  2, subtriangle 2
92.7     54.7   41.4   %Gauss pt.  3, subtriangle 2

106.2   55.3   60.5   %Gauss pt.  1, subtriangle 3
101.6   68.9   38.7   %Gauss pt.  2, subtriangle 3
93.7     64.7   61.4   %Gauss pt.  3, subtriangle 3

112.2   35.3   20.5   %Gauss pt.  1, subtriangle 4
125.6   78.9   28.7   %Gauss pt.  2, subtriangle 4
78.7     24.7   71.4   %Gauss pt.  3, subtriangle 4

104.2   85.3   20.5   %Gauss pt.  1, subtriangle 5
101.6   28.9   68.7   %Gauss pt.  2, subtriangle 5
91.7     94.7   11.4   %Gauss pt.  3, subtriangle 5

% element 2

... % more values

% element 3,

...
```

Apéndice B

Extracto del geometry shader en método Transform Feedback

```
#version 400

// Adaptation of implementation (http://www.icare3d.org/codes-and-projects/codes/
// opengl_geometry_shader_marching_cubes.html)

in VertexData{
    vec3 VertexPosition;
    float VertexScalar;
    uint VertexFlags;
} vertexData[4];

layout(lines_adjacency) in;
layout(points, max_vertices = 6) out;

//Triangles table texture
uniform isampler2D triTableTex;

out vec3 vertexPosition;
out vec3 vertexNormal;

uniform float Isolevel;
uniform int ElementDrawOption;

vec3 vertexInterp(float val, vec3 pos1, float v1, vec3 pos2, float v2) {
    return mix(pos1, pos2, (val-v1)/(v2-v1));
}

int triTableValue(int i, int j){...}
```

```
bool isFlagEnabled(uint f){...}

void main()
{
    int tetindex = 0;
    float tetVal0 = vertexData[0].VertexScalar;
    float tetVal1 = vertexData[1].VertexScalar;
    float tetVal2 = vertexData[2].VertexScalar;
    float tetVal3 = vertexData[3].VertexScalar;

    //Determine the index into the edge table which
    //tells us which vertices are inside of the surface
    tetindex = int(tetVal0 < Isolevel);
    tetindex += int(tetVal1 < Isolevel)*2;
    tetindex += int(tetVal2 < Isolevel)*4;
    tetindex += int(tetVal3 < Isolevel)*8;

    //Tetrahedron is entirely in/out of the surface
    if (tetindex == 0 || tetindex == 15) {
        return;
    }

    //Find the vertices where the surface intersects the tetrahedron
    vec3 vertlist[6];
    vertlist[0] = vertexInterp(Isolevel, vertexData[1].VertexPosition,
                           tetVal1, vertexData[0].VertexPosition, tetVal0);
    vertlist[1] = vertexInterp(Isolevel, vertexData[1].VertexPosition,
                           tetVal1, vertexData[2].VertexPosition, tetVal2);
    ...
    vertlist[5] = vertexInterp(Isolevel, vertexData[2].VertexPosition,
                           tetVal2, vertexData[3].VertexPosition, tetVal3);

    int j=0;
    while(true){
        if(triTableValue(tetindex, j)!=-1 && j != 6){
            vec3[3] vertexPositions;
            vertexPositions[0] = vertlist[triTableValue(tetindex, j)];
            vertexPositions[1] = vertlist[triTableValue(tetindex, j+1)];
            vertexPositions[2] = vertlist[triTableValue(tetindex, j+2)];
            vec3 normal = normalize(cross(vertexPositions[1]-vertexPositions[0],
                                  vertexPositions[2]-vertexPositions[0]));

            //Generate first vertex of triangle//
            //Fill position varying attribute for fragment shader
            vertexPosition = vertexPositions[0];
            vertexNormal = normal;
            EmitVertex();
```

```
            EndPrimitive();

            //Generate second vertex of triangle//
            //Fill position varying attribute for fragment shader
            vertexPosition = vertexPositions[1];
            vertexNormal = normal;
            EmitVertex();
            EndPrimitive();

            //Generate last vertex of triangle//
            ...
        }else{
            break;
        }
    }
}
```