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

CAMARÓN: VISUALIZADOR Y EVALUADOR DE MALLAS GEOMÉTRICAS MIXTAS GRANDES EN 3D, ACELERADO CON SHADERS EN OPENGL

MEMORIA PARA OPTAR AL TÍTULO DE INGENIERO CIVIL EN COMPUTACIÓN

ALDO VINCENZO CANEPA GARAY

PROFESOR GUÍA:
NANCY HITSCHFELD KAHLER

MIEMBROS DE LA COMISIÓN:
BENJAMIN BUSTOS CÁRDENAS
MAURICIO PALMA LIZAMA

Esta memoria fue apoyada parcialmente por Proyecto Fondecyt N° 1120495

SANTIAGO DE CHILE
ENERO 2013

Resumen

El objetivo del presente trabajo de titulación es desarrollar y diseñar una aplicación multiplataforma para visualizar y analizar mallas geométricas mixtas en tres dimensiones. El visualizador debió ser implementado priorizando la eficiencia para poder manejar mallas muy grandes, y además poseer un diseño de calidad que permitiera extenderlo fácilmente.

Una malla geométrica es un conjunto de celdas adyacentes que buscan modelar un objeto complejo de forma discreta; dependiendo del tipo de malla, las celdas serán polígonos y/o poliedros. Los elementos que forman las mallas pueden ser evaluados bajo distintos criterios para medir su calidad.

Algunos modelos tienen millones de elementos, por lo que realizar cualquier tipo de cálculo sobre la malla, incluyendo renderizarla, es muy costoso. Para manejar mallas grandes de forma eficiente, varios algoritmos fueron acelerados utilizando la unidad de procesamiento gráfico de la tarjeta de video (GPU). Las GPU de última generación permiten realizar un gran número de tareas en paralelo, ya que tienen cientos de núcleos de procesamiento.

Para poder aprovechar el potencial de la tarjeta de video se utiliza la API gráfica multiplataforma OpenGL. La interfaz de la API provee un gran número de funciones para controlar la tarjeta de video, y además especifica el lenguaje de programación GLSL, el cual nos permite programar el pipeline de renderizado, todo esto sin preocuparse del tipo de tarjeta de video ni del sistema operativo.

Para abordar el problema, primero se analizaron los requerimientos específicos de la aplicación y se confeccionó un diseño para satisfacerlos. Se propuso un esquema modular, utilizando programación orientada a objetos y patrones de diseño, con lo cual se consiguió una aplicación extensible y capaz de realizar las tareas requeridas.

Después, se implementaron las clases especificadas en el diseño y se generó una interfaz gráfica amigable para el usuario utilizando Qt. Esta interfaz permite al usuario acceder a todas las funcionalidades de la aplicación. Además, se utilizaron múltiples programas de Shaders para producir distintos efectos de iluminación y para acelerar algoritmos que no están relacionados con la visualización.

Como resultado se obtuvo una poderosa herramienta gratuita y multiplataforma para analizar mallas. Con un diseño de calidad se consiguió que la aplicación fuese fácilmente extensible en estrategias de evaluación, modos de visualización, tipos de formatos que lee y exporta, y modos de seleccionar elementos. Además, el visualizador es capaz de procesar mallas que contienen por sobre un millón de elementos en tiempos reducidos, por lo que sigue siendo una aplicación altamente interactiva bajo estas condiciones.

Agradecimientos

A mi familia que me ha dado su pleno apoyo en todos mis años de formación y a los amigos que siempre han estado ahí desde el colegio, los cuales también han sido como una familia, y han sabido aconsejar, comprender y tabanear.

A mis amigos/compañeros de universidad que han hecho de estos 6 años de universidad mucho más alegres y llevaderos.

A Ignacia Pérez Pons por todo su apoyo, ayuda y compañía en este proceso y fuera de él, y a Mitzy Danton por su colaboración en la corrección de la redacción y ortografía del presente trabajo.

Tabla de contenido

Índice de figuras

Índice de tablas

1. Introducción

1.1. Aspectos generales

Actualmente, los computadores son indispensables para el modelamiento y simulación de situaciones reales, las cuales pueden resultar muy complejas. Por ello, es importante que los modelos y simulaciones representen de forma fidedigna la información del mundo real a través de modelos de datos discretos que sean posibles de procesar. [1]

Un ejemplo de modelo de datos son las mallas geométricas. Una malla geométrica consiste en un conjunto de celdas adyacentes que buscan modelar la geometría de un objeto. Cuando se habla de una malla en dos dimensiones (2D) o de superficie, se refiere a que estas celdas serán polígonos planos como triángulos, cuadriláteros, etc. En cambio, en una malla geométrica en tres dimensiones (3D) las celdas serán poliedros tales como tetraedros, pirámides y hexaedros.

La calidad de una malla geométrica juega un rol muy importante en la precisión y estabilidad de la computación numérica que se realice sobre esta. Existen diversas formas de medir y mejorar la calidad de un modelo, observando las distintas propiedades locales de los polígonos o poliedros que componen la malla. [2]

Actualmente existen pocos visualizadores y evaluadores de mallas geométricas que además de gratuitos y multiplataforma, tengan un buen rendimiento aprovechando la potencia del hardware de última generación.

Como respuesta a lo anterior, es que el presente trabajo consiste en el diseño e implementación de un visualizador con estas características. El visualizador cuenta con distintas alternativas para renderizar las mallas, estudiarlas, evaluarlas, cargarlas y exportarlas en distintos formatos.

Además, el visualizador está pensado para trabajar con mallas de gran tamaño (por sobre 1 millón de elementos), por lo que la optimización de las rutinas y el paralelismo es un punto clave. El paralelismo, en este caso, se dará principalmente en un uso intensivo de la unidad de procesamiento gráfico (GPU) para realizar cálculos locales sobre los elementos de la malla.

Para poder alcanzar los objetivos mencionados, la aplicación hace uso de OpenGL, la cual es una especificación de un estándar que define una API gráfica multiplataforma y

multilenguaje. Con OpenGL podremos hacer uso de la GPU para paralelizar los cálculos sobre grandes volúmenes de datos cada vez que sea posible.

1.2. Justificación y motivación

Las mallas son una herramienta fundamental en muchas áreas de las ciencias, ya que permiten modelar procesos y cuerpos complejos, y obtener resultados que de otra forma sería muy caro o imposible de lograr.

Como las mallas geométricas normalmente están descritas por una cantidad de datos muy grande, es muy difícil analizarlas y evaluarlas mirando sólo los números que las definen. Para esto se hacen muy importantes las herramientas de visualización.

Actualmente existen pocos visualizadores gratuitos y los que hay, tienen funciones muy limitadas. Por ejemplo TetView, sólo sirve para visualizar mallas de tetraedros y triángulos, y Geomview permite visualizar mallas de superficie, hacer traslaciones de la malla completa, rotaciones, ver sólo aristas o sólo caras y zoom. Ninguna provee funcionalidades para el análisis numérico de estas, ni tampoco para manipular las coordenadas de los vértices, lo que podría ser útil para mejorar la calidad de la malla.

Geomview y TetView no utilizan las técnicas modernas de renderizado y procesamiento en GPU. Sin embargo en el visualizador, materia de este trabajo, serán implementadas estas tecnologías. En la GPU podemos realizar tareas a bajo costo, que de otra forma serían muy costosas de computar. La tecnología de procesamiento grafico ha cambiado y se ha vuelto mucho más poderosa en los últimos años.

Las principales motivaciones son:

- Conocer más acerca de la computación gráfica, trabajando e investigando una de sus principales herramientas de modelado (mallas geométricas), utilizando e implementado técnicas de renderizado y algoritmos geométricos eficientemente.
- Proveer a la comunidad de un visualizador de calidad y gratuito, para el análisis y visualización de mallas.
- Aprender las técnicas de programación de Shaders para hacer un uso óptimo de la GPU.
- Realizar un diseño de calidad, evitando repeticiones de código, buscando modularidad y bajo acoplamiento para que las partes sean más fáciles de mantener y extender.

1.3. Objetivos

1.3.1. Objetivo general

Diseñar y desarrollar un visualizador y evaluador de mallas geométricas mixtas en 3D que sea fácil de usar. El visualizador debe aprovechar las tecnologías actuales para poder obtener un rendimiento óptimo para que sea posible analizar mallas muy grandes. Además, debe ser capaz de cargar y exportar las mallas en distintos formatos. La aplicación desarrollada debe ser gratis y de código abierto, y debe estar diseñada para que sea fácilmente extensible.

1.3.2. Objetivos específicos

La aplicación debe cubrir los siguientes puntos:

- Ser capaz de visualizar mallas geométricas no estructuradas en 3D, formada por polígonos y poliedros convexos. Las mallas deben poder cargarse y exportarse en distintos formatos.
- Permitir al usuario rotar, trasladar, acercar y alejar la cámara para poder apreciar el modelo desde distintas posiciones y ángulos.
- Dar la posibilidad de escoger entre distintas modalidades de visualización que permita analizar de mejor manera un modelo, como por ejemplo: transparencias, sólo superficie o aristas, solamente vértices, normales, etc.
- Incorporar criterios de evaluación de calidad de los elementos de las mallas y distintas formas de seleccionar dichos elementos.
- Poseer un diseño tal que un desarrollador pueda agregar nuevas estrategias de evaluación, formas de selección y algoritmos de visualización fácilmente.

El visualizador debe ser eficiente trabajando con mallas del orden de un millón de elementos.

1.4. Contenidos

1. Introducción: Se presenta los aspectos generales del tema, la motivación y los objetivos del trabajo realizado.

2. Antecedentes: Se exponen los contenidos necesarios para poder entender el desarrollo y resultado del trabajo. Se revisan conceptos geométricos, herramientas similares a la desarrollada, librerías externas útiles para el proyecto y conceptos relacionados con computación gráfica.

3. Diseño: Se presentan los requerimientos que la aplicación debió cumplir, y además se revisan conceptos básicos relacionados a ingeniería de software que se utilizaron en la confección del diseño. En la sección 3.3 se revisa el diseño completo de la aplicación.

4. Implementación: El capítulo comienza con una revisión del ambiente de desarrollo de la aplicación y detalles sobre el manejo de las librerías externas utilizadas. Luego, se muestran casos de implementación por cada módulo extensible (Estrategias de evaluación, de selección, de renderizado, etc.), y además, la implementación de aquellas secciones que resultaron más interesantes.

5. Mediciones de rendimiento/eficiencia: Se describe un ambiente de pruebas: los modelos, visualizadores y criterios utilizados; y se exponen los resultados obtenidos.

6. Conclusiones: Se muestran las conclusiones obtenidas a partir del trabajo realizado, se revisa si se cumplieron los objetivos y se analiza el posible trabajo futuro que se puede llevar a cabo sobre la aplicación.

7. Referencias: Se presentan las fuentes de información bibliográfica utilizadas en relación al marco teórico y trabajos relacionados sobre el tema en estudio.

8. Anexos: Se presenta información adicional y de interés sobre el desarrollo de la memoria.

2. Antecedentes

En este capítulo se presentan los antecedentes relacionados con el área en que se desarrolla el presente trabajo de memoria. Para poder diseñar e implementar una herramienta que sea capaz de cumplir con los objetivos expuestos es necesario manejar una serie de conocimientos base. En las siguientes secciones del capítulo se expondrán los datos obtenidos de la investigación previa al desarrollo del visualizador.

Primero, se presenta la información necesaria para entender todos los conceptos geométricos que se utilizan en la aplicación, entre estos tenemos los conceptos de malla geométrica, transformaciones y calidad de elementos. Los conceptos se explican desde un punto de vista geométrico y también se relacionan con la computación.

Luego, se exponen las razones de por qué se escoge OpenGL como la API de trabajo y se ahonda en los conceptos básicos para entender cómo se realiza el proceso de renderización de un modelo bajo este estándar. Además, se estudia el uso de las tecnologías relacionadas con shaders, el manejo de transparencias y las transformaciones entre sistemas de coordenadas.

Posteriormente, se expone un pequeño análisis donde se escoge el Framework/librería que se utilizará para construir la GUI (Graphic User Interface) del visualizador de manera que sea multiplataforma. El capítulo termina con la revisión de las características de otros visualizadores gratuitos existentes.

2.1. Conceptos geométricos

2.1.1. Geometría computacional

La Geometría Computacional es una rama de la computación que se dedica al estudio de algoritmos geométricos. Esta área busca esencialmente diseñar y analizar algoritmos para solucionar problemas geométricos de forma eficiente, relacionándose fuertemente con las matemáticas discretas y el análisis combinatorio. [3]

La Geometría Computacional se vio fuertemente impulsada por el avance de la computación gráfica, y el diseño y representación de objetos reales asistido por computador, sin embargo tiene muchas otras aplicaciones dentro de las que se incluyen la robótica (planificación de movimientos y problemas de visualización), diseño de circuitos

integrados, sistemas de localización, estudio de comportamiento de sólidos y en general cualquier fenómeno físico, programación de comportamiento de máquinas, etc.

La simulación de objetos o fenómenos reales puede ser analizada en dos o tres dimensiones, en donde el problema en 2D es una simplificación del problema en 3D. El tema de ésta memoria pretende facilitar el estudio de modelos en 3D.

2.1.2. Mallas geométricas

Una malla geométrica es una colección de vértices, aristas y caras, que define la forma de un objeto complejo en base a polígonos (2D) y poliedros (3D), es decir, en base a formas simples y conocidas. Se distinguen dos tipos de mallas: [4]

- Mallas Estructuradas: se componen de celdas del mismo tipo y tamaño, como por ejemplo, rectángulos o triángulos en (2D) y hexaedros o tetraedros (3D).

- Mallas No Estructuradas: se componen de celdas del mismo o distinto tipo pero de tamaños diferentes. Existen en general, algunas zonas con celdas pequeñas y otras con celdas de mayor tamaño.

La ventaja de las mallas estructuradas con respecto a las no estructuradas, es que las estructuradas son más fáciles de generar y de estudiar sus propiedades, sin embargo no permiten modelar problemas o formas complejas.

Dada una malla cualquiera, existen diversas maneras de analizar la calidad de los elementos que las componen (como se verá posteriormente), para así aplicar métodos que permiten refinarlas o mejorar los atributos de elementos que no cumplen con el estándar de calidad deseado y así mejorar la calidad general de la malla.

Es posible distinguir mallas geométricas, sus características y propiedades, según el espacio en el que se encuentran, ya sea 2D o 3D, tal como se detalla en las siguientes secciones.

2.1.2.1. Mallas Geométricas 2D

En 2D, las mallas más utilizadas son las mallas de triángulos, sobre todo en la modelación de superficies. Las mallas de triángulos se denominan triangulaciones, siendo la más conocida la triangulación de Delaunay, la cual maximiza el ángulo mínimo de los triángulos que la componen.

2.1.2.2. Mallas de Superficie

Las triangulaciones, como se mencionó anteriormente, son comúnmente usadas en la modelación de superficies topográficas o superficies de objetos. La diferencia con las mallas en 2D es que los polígonos están en un espacio 3D.

Ya que este tipo de mallas se componen de elementos en 2D, es decir polígonos y no de poliedros, pero además se les agrega una componente z o de altura, se suele decir que son mallas en 2 ½D.

2.1.2.3. Mallas Geométricas 3D

Una malla geométrica en 3D o de poliedros es una colección de vértices, aristas, caras y poliedros. Un polígono (cara) es compartido a lo más por dos poliedros.

2.1.3. Criterios de evaluación

Existen diversos métodos de evaluar la calidad de los elementos que componen mallas geométricas 3D. Como la calidad de la malla es relativo a cómo se va a procesar esta o que se va a hacer con ella, el visualizador sólo entrega herramientas para evaluar los elementos de la malla, pero la interpretación de estas medidas está a cargo del usuario. Entre estas herramientas están los siguientes criterios de evaluación:

2.1.3.1. Ángulo Sólido

El ángulo sólido, es un ángulo que representa el tamaño aparente de un objeto para un observador ubicado en un punto arbitrario. El tamaño aparente, representado por el ángulo sólido, se obtiene proyectando el objeto sobre una esfera centrada en el observador. El ángulo es independiente del radio de la esfera escogido, sólo depende de la forma del objeto y de la posición relativa del observador al objeto.

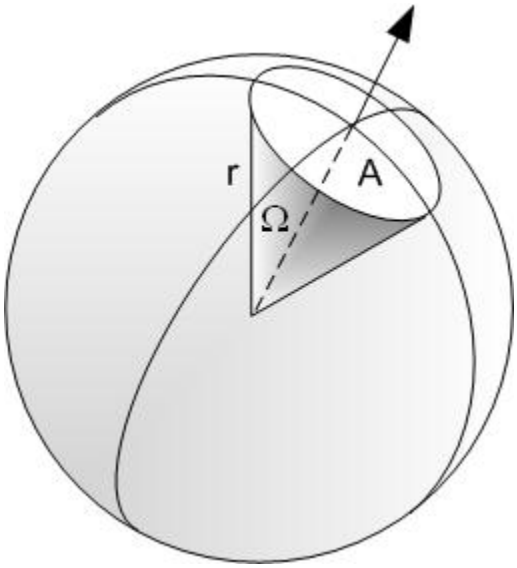


Figura 1 - Ángulo Sólido: La superficie A, es la superficie que cubre la proyección del objeto en la esfera de radio r. El ángulo sólido es representado por la letra griega Omega Ω.

El ángulo sólido es definido por la siguiente ecuación:

$$\Omega = \frac{A}{r^2}$$

Como la esfera es de **r** arbitrario, y el ángulo sólido finalmente no dependerá de éste, el problema principal que hay que resolver aquí es cómo obtener el área A. Una forma conveniente y que sirve para cualquier polígono simple, es dividir las caras posteriores[1] en triángulos, y proyectar cada triángulo sobre la superficie de la esfera. Luego, sumamos las áreas de las superficies de los triángulos proyectados.

Para proyectar un triángulo sobre una esfera centrada en el observador, primero se deben calcular las coordenadas del triángulo relativas a un sistema de coordenadas centrado en dicho punto. Después, se convierten estas coordenadas a coordenadas esféricas, también centradas en el mismo punto, lo que facilita la proyección. La proyección de cada vértice se calcula asignándole el valor **r** a la componente radial, y de esta forma se obtiene el *triángulo esférico* proyectado en la superficie de la esfera de radio **r.**

[1] Podemos determinar esto utilizando el vector normal asociado a cada cara.

Un *triángulo esférico* es un triángulo curvado al estar dibujado en la superficie de una esfera. A diferencia de un triángulo plano común, los ángulos interiores de un triángulo esférico suman más de 180° y menos de 540°.

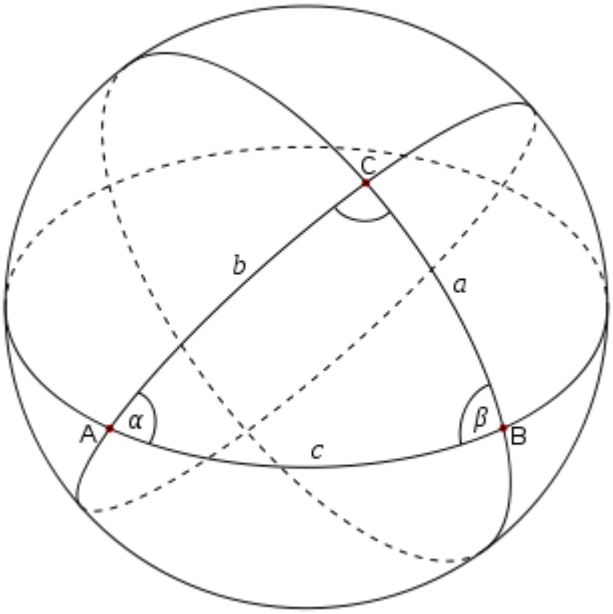


Figura 2 - Triángulo Esférico: El triángulo esférico está definido por los vértices A, B y C. Los arcos del triángulo están continuados para mostrar las circunferencias que los representan. **[5]**

$$180° < \alpha + \beta + \gamma < 540°$$

La cantidad **Ê** por la cual esta suma excede los 180°, se llama exceso esférico del triángulo: [6]

$$\hat{\mathrm{E}} = \alpha + \beta + \gamma - 180°$$

El área de un triángulo esférico puede ser calculada como:

$$A = R^2 \hat{\mathrm{E}}$$

Donde **Ê** es el exceso esférico del triángulo en radianes.

Para obtener los ángulos α, β y γ, podemos hacer uso de las propiedades básicas de triángulos esféricos. Para este caso, nos serán útiles:

Ley de los Senos

sin α / sin a = sin β / sin b = sin γ / sin c

Ley de los cosenos

cos a = cos b cos c + sin b sin c cos α
cos α = -cos β cos γ + sin β sin γ cos a

Como podemos ver, las identidades descritas utilizan funciones de coseno y seno sobre los elementos **a**, **b** y **c.** Esto es posible ya que los lados de un triángulo esférico son arcos, estos pueden ser descritos como ángulos, por lo tanto, tendremos dos tipos de ángulos:

Ángulos α, β y γ: Son los ángulos en los vértices del triángulo, formados por las circunferencias intersectándose en los vértices.

Ángulos a, b y c: Son los lados del triángulo, medidos por los ángulos formados por las líneas conectado los vértices al centro de la esfera. Estos ángulos, que son los faltantes, los obtenemos a partir de las coordenadas esféricas de los vértices.

Finalmente, con lo expuesto anteriormente, se puede realizar el cálculo del ángulo sólido Ω

En este caso, el ángulo sólido se calculará desde los vértices del poliedro mismo. En el visualizador se podrá extraer el mínimo o máximo ángulo sólido de un poliedro.

2.1.3.2. Ángulo Diedro

En geometría, un ángulo diedro se define como el ángulo entre dos planos. El ángulo diedro φ_{AB} entre dos planos A y B, es el ángulo entre los dos vectores normales unitarios de los planos:

$$\cos \varphi_{AB} = n_A \cdot n_B$$

De la ecuación anterior, sacamos que podemos obtener el ángulo calculando el coseno del producto punto entre las normales. El ángulo diedro puede tener signo, ya que por ejemplo, φ_{AB} representa cuanto debe rotarse un plano A (en torno a la línea de intersección) para que se alinee con el plano B. Luego, $\varphi_{AB} = -\varphi_{BA}$.

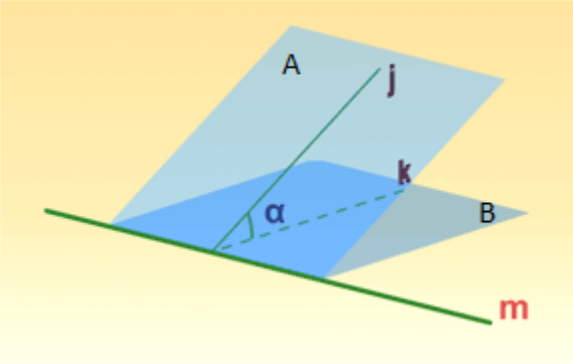


Figura 3 - Ángulo Diedro

En el contexto de la aplicación a desarrollar, los ángulos diedros que interesan son los formados por la intersección de planos definidos por caras adyacentes. Obtener la normal de dichos planos es una operación simple y se realiza en tiempo constante.

2.1.3.3. Razón de aspecto

Los criterios de evaluación en base a razón de aspecto intentan medir la calidad de un poliedro, evaluando qué tan regular es su forma.

Este criterio puede tener más de una variante, por ejemplo:

- Razón entre la arista más larga y la más corta de un poliedro: Es el criterio en su forma más simple. Mientras la razón este más cercana a 1, significa que sus aristas son todas más parecidas entre ellas y el poliedro de mejor calidad.
- Razón entre el volumen del poliedro y la arista más larga al cubo: [7] Este criterio se puede calcular buscando el volumen del poliedro, como se explicó anteriormente, y luego dividiéndolo por su arista más larga al cubo. Esto es equivalente a calcular la razón entre el volumen de un cubo cuyas aristas miden igual a la arista más larga del poliedro y el volumen del poliedro.

Estos criterios son independientes de la escala del poliedro.

Para el visualizador, será implementado el segundo criterio, ya que se considera más útil como medida de evaluación.

2.1.4. Sistemas de coordenadas

Un sistema de coordenadas o espacio de coordenadas, en este contexto, se puede decir que consiste en: [8]

- Una dimensión: En este caso, todos los modelos están en el espacio 3D, y cuando se mencione, estaremos en el espacio de coordenadas homogéneas 4D.
- Una serie de vectores de la dimensión del espacio, los cuales definen los ejes del espacio. No necesariamente son ortogonales[2], pero debe haber uno por dimensión. Cada vector tiene su nombre, por ejemplo: **X**, **Y**, **Z**, etc. Estos son los vectores base del espacio.
- Un punto en el espacio que define el origen. El origen es el punto desde el cual todos los demás puntos son descritos.
- Un área donde los puntos son válidos. Fuera de este rango, las posiciones no son válidas. El rango puede ser infinito.

Una posición o vértice en el espacio está definida como la suma de los vectores base, donde cada vector es multiplicado por un valor escalar llamado coordenada. Geométricamente, esto se ve de la siguiente forma (2D):

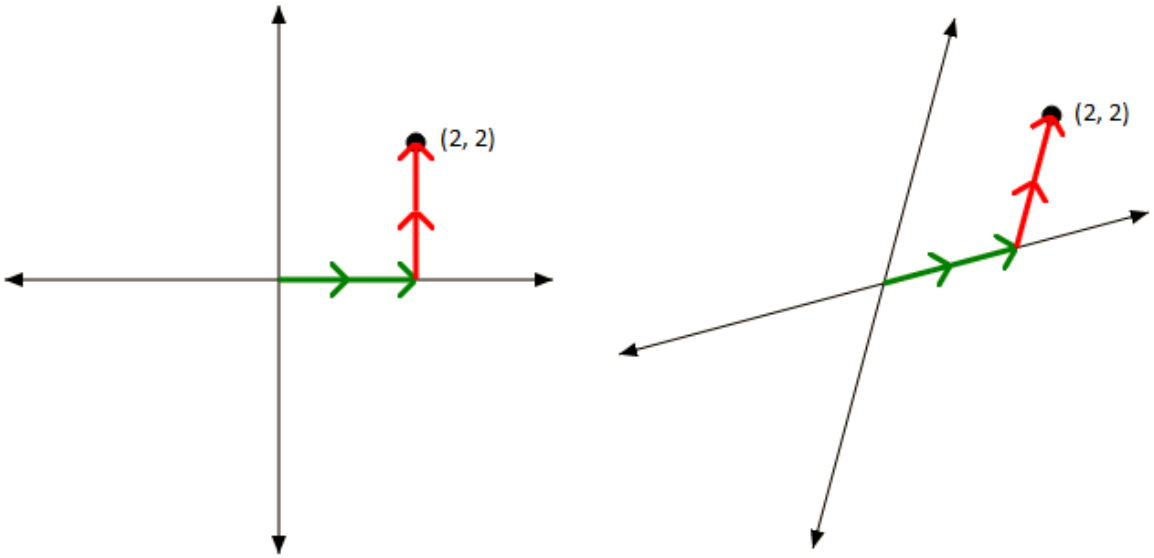


Figura 4 - Sistemas de Coordenadas: Las mismas coordenadas (2, 2) relativas a dos sistemas distintos. Desde un punto de vista neutral como del lector, se puede ver claramente que no corresponde al mismo punto. Esto muestra que una coordenada por sí misma no es suficiente información para saber qué significa, también se debe especificar el sistema de coordenadas. **[8]**

[2] Dos vectores son ortogonales cuando hay un ángulo recto entre ellos.

2.1.5. Transformaciones geométricas

En computación gráfica, para poder generar una imagen de un modelo (renderizar) y además para poder realizar operaciones básicas sobre los elementos del mismo, como rotar, trasladar, escalar, aplicamos transformaciones geométricas.

Las transformaciones geométricas pueden ser aplicadas sobre un vector de coordenadas multiplicándolo por una matriz de transformación apropiada. Esto es análogo a cambiar las coordenadas del vector desde un sistema de coordenadas a otro.

El uso de matrices de transformación permite representar cualquier transformación linear de forma consistente y adecuada para computar. También permite concatenar múltiples transformaciones de forma muy fácil, pre-multiplicando las matrices entre ellas[3].

Por otro lado, existen casos en que se quieren representar transformaciones no lineales por matrices (ejemplo, la traslación). Algunas transformaciones no lineales en un espacio n-dimensional, pueden ser representadas como transformaciones lineales en un espacio n+1-dimensional. Esta propiedad es muy importante, es lo que nos permite en computación grafica describir todas las transformaciones que necesitamos como una matriz cuadrada.

Como las operaciones de traslación y proyecciones son operaciones no lineales, para poder representarlas con una matriz, es necesario llevar las coordenadas del punto al que se quiere aplicar la transformación a una dimensión superior y describir la operación con una matriz del nivel superior. Estas coordenadas de nivel superior son llamadas coordenadas homogéneas.

Asumiendo que estamos trabajando con puntos en 3D, las coordenadas homogéneas son en 4D. Si tenemos un punto **p** en nuestro sistema original de 3D, y el mismo punto **p'** en coordenadas homogéneas, se transforma entre un sistema y otro de la siguiente forma: $\mathbf{p'_x = p_x * w}$, $\mathbf{p'_y = p_y * w}$, $\mathbf{p'_z = p_z * w}$, $\mathbf{p'_w = w}$. Como se puede apreciar en las ecuaciones, un punto **p** en el sistema de coordenadas originales puede tener infinitas representaciones **p'** en coordenadas homogéneas. Lo más simple y frecuente para pasar un punto **p** del sistema de coordenadas original al sistema de coordenadas homogéneas es hacer $\mathbf{w = 1}$, luego $\mathbf{p'_x = p_x}$, $\mathbf{p'_y = p_y}$, $\mathbf{p'_z = p_z}$.

Las ventajas de las coordenadas homogéneas importantes para el proyecto, es que pueden representar puntos en el infinito (en el sistema original) usando coordenadas finitas ($\mathbf{w = 0}$) y que podemos realizar las operaciones no lineales de forma lineal con matrices.

Los detalles de las transformaciones geométricas que se utilizarán en este trabajo pueden ser encontrados en el Anexo G.

[3] La multiplicación de matrices es asociativa y además la multiplicación entre matrices cuadradas de igual dimensión, da como resultado una matriz cuadrada de la misma dimensión.

2.2. API's gráficas

2.2.1. Direct3D

Direct3D es una parte de la API de Microsoft DirectX. Direct3D es una API poderosa, la cual es muy utilizada en la actualidad y tiene una gran cantidad de soporte, pero tiene la limitación de que sólo está presente en el sistema operativo Windows. Existe una permanente discusión al respecto de que API (OpenGL o Direct3D) es mejor, pero no hay consenso, básicamente, cada una supera a la otra en distintos ámbitos.

Para el visualizador, Direct3D queda descartado porque la aplicación debe ser multiplataforma.

2.2.2. OpenGL

OpenGL es la especificación de un estándar que define una API gráfica multiplataforma y multilenguaje. La interfaz provee un gran número de funciones, las cuales son capaces de dibujar escenas en 2D o 3D utilizando primitivas simples (por ejemplo: polígonos, líneas, puntos).

La especificación define lo que vendría a ser una máquina de estados, la cual convierte las primitivas que ingresan en ella a pixeles en la pantalla. La forma de convertir las primitivas está dictada por una serie de operaciones y las configuraciones (o estado) de la máquina.

Con las primeras tarjetas gráficas, y con las primeras especificaciones de OpenGL (antes de 2.0), el proceso de renderizado estaba implementado de forma fija y era muy poco configurable. Este enfoque de programación más rígido se llama "fixed functionality". Este enfoque es el que se enseña actualmente en la universidad y normalmente se escoge porque es más fácil de aprender para un principiante.

El enfoque anterior tiene desventajas: [9]

- Es limitado, el programador sólo puede utilizar las operaciones de renderizado ya implementadas y las configuraciones que puede hacer son mínimas.
- Esconde el real funcionamiento de OpenGL. Normalmente el programador que utiliza este tipo de esquema, no llegara a entender cómo funciona la máquina de estados realmente.

- Programar con las técnicas más modernas se dificulta, ya que es completamente distinto y los conocimientos anteriores tienen que ser omitidos para que no lleven a confusiones. Hay muy poco de programar con "fixed functionality" que pueda servir en programación con Shaders.

Con OpenGL 2.0[4], se incluye en el núcleo de la especificación un lenguaje llamado OpenGL Shading Languaje (GLSL) basado en la sintaxis de C. GLSL provee al desarrollador un control más directo de las operaciones que se realizan en el proceso de renderizado. GLSL es un lenguaje de alto nivel, que nos permite programar la GPU sin tener que usar Assembler o un lenguaje dependiente del hardware.

GLSL es uno de los lenguajes utilizados para programar Shaders, y será el utilizado en este trabajo.

2.3. OpenGL y GLSL

Un Shader es un programa diseñado para ejecutarse en un procesador gráfico (GPU) como parte de las operaciones de renderizado. Los Shader se ejecutan en etapas fijas y bien definidas en el proceso. Estas etapas representan ganchos donde el usuario puede agregar un algoritmo arbitrario para crear un efecto visual específico.

Existen múltiples etapas[5] donde se pueden insertar Shaders para realizar procesamientos útiles para el usuario de forma muy económica. Por ejemplo, en la etapa de procesamiento de vértices individuales, donde se decide si están o no en el campo de visión que se dibujará, aquí se puede agregar un *Vertex Shader* para realizar operaciones independientes y paralelas sobre los vértices de las primitivas que se están procesando.

Los Shaders son ejecutados por la GPU, lo cual permite que se libere mucho tiempo de computación en la CPU, el cual puede ser utilizado para otras tareas. Una desventaja de esto, es que tienen limitaciones que un código que se ejecuta en la CPU no tiene, por lo tanto los alcances de cada tipo de Shaders son restringidos, pero aun así, pueden ayudarnos a realizar muchos cálculos en paralelo que en la CPU serían muy caros.

[4] La especificación OpenGL 2.0 fue publicada en Septiembre del 2004

[5] En OpenGL 4.0, existen 5 etapas para definir Shaders: Vertex, Geometry, Tessellation Control, Tessellation Evaluation y Fragment.

2.3.1. Rendering Pipeline

El Rendering Pipeline es la secuencia de pasos que OpenGL toma para poder renderizar objetos. En este capítulo se revisará de forma resumida, pero con la profundidad suficiente para el entendimiento del presente trabajo. Las etapas que no tienen ninguna relevancia para el trabajo serán omitidas. [10]

Las etapas ordenadas del pipeline son las siguientes:

1. Procesamiento individual de cada vértice por un Vertex Shader. De cada vértice se genera un vértice de salida.
2. Paso opcional de generar primitivas en las etapas de teselado.
3. Paso opcional de procesar las primitivas con un Geometry Shader. Esta etapa tiene como salida una secuencia de primitivas.
4. Clipping de primitivas, división por w de las coordenadas para ir a espacio NDC. Transformar las coordenadas a coordenadas de la ventana.
5. Barrido de conversión (Scan conversión) e interpolación de parámetros a través de la primitiva. Se genera una serie de fragmentos.
6. Cada fragmento es procesado por un Fragment Shader. Un Fragment Shader tiene un número variable de salidas llamadas muestras, aunque normalmente es un color (vector de 4 coordenadas).
7. Procesamiento por muestra:
    a. Scissor Test
    b. Stencil Test
    c. Depth Test
    d. Blending
    e. Logical Operations
    f. Write Mask

Los Vertex Shader, Geometry Shader y Fragment Shader son descritos junto con su funcionamiento en la sección 2.3.2. Así que por el momento serán omitidos.

Clipping Space

Clipping Space es un espacio geométrico finito en 4D definido por OpenGL. Para que las primitivas puedan ser rasterizadas correctamente, es necesario que las coordenadas de los vértices sean transformadas a este sistema de coordenadas (coordenadas clip). Este espacio representa el espacio transformado visible por la cámara. [11]

En el paso 4 del proceso del pipeline, si alguna primitiva queda completamente fuera del Clipping Space, es descartada (esta fuera del rango visual). En el caso de que un triángulo quede con vértices dentro y fuera del Clipping Space, este se divide (Clip) en triángulos más pequeños que queden dentro del espacio y los vértices que estaban afuera son descartados.

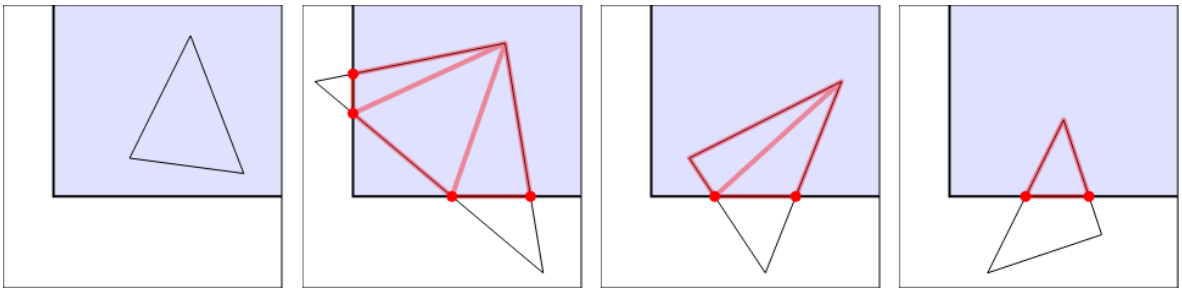


Figura 5 - Triangle Clipping **[12]**

Después de dividir y descartar los triángulos que sea necesario, se transforman las coordenadas al sistema NDC, dividiéndolas por su componente w.

La etapa termina con transformar las coordenadas NDC a coordenadas de ventana. Como lo sugiere el nombre, las coordenadas de ventana son relativas a la ventana en que está funcionando OpenGL. Sin embargo, estas coordenadas tienen tres dimensiones, el vector X va hacia la derecha, el vector Y hacia arriba y la Z representa la profundidad con valores en [0,1], donde 0 es el más cercano. También, es importante notar que con estas coordenadas la precisión no se ha perdido, ya que no son números enteros, son de punto flotante.

El origen del sistema de coordenadas se encuentra en la esquina inferior izquierda de la pantalla.

El programador solamente se debe preocupar de que las coordenadas sean transformadas al espacio Clip, pero las demás transformaciones son responsabilidad de la implementación de OpenGL. Más detalles con respecto a los sistemas de coordenada se pueden encontrar en las secciones 2.3.3 y 4.4.

Scan Conversion, interpolación de parámetros

Después de convertir las coordenadas de un triángulo a coordenadas de la ventana, el triángulo pasa por un proceso llamado *conversión de barrido* (Scan Conversion). Este proceso toma el triángulo y lo rompe basado en los pixeles de la pantalla que cubre.

Figura 6 - Scan Conversion **[11]**

La imagen central muestra la grilla digital de pixeles de salida, el círculo en cada pixel representa su centro. El centro de cada pixel representa una *muestra*. Durante el barrido de conversión, un triángulo generará un fragmento por cada centro de pixel que quede dentro del área en 2D del triángulo

La imagen a la derecha muestra los fragmentos generados por el barrido de conversión del triángulo. El barrido crea una aproximación en pixeles de la forma del triángulo.

Es común que los triángulos procesados compartan aristas. OpenGL nos asegura que, mientras las posiciones de los vértices en una arista compartida sean las mismas, no quedaran hoyos sin fragmentos entre los dos triángulos.

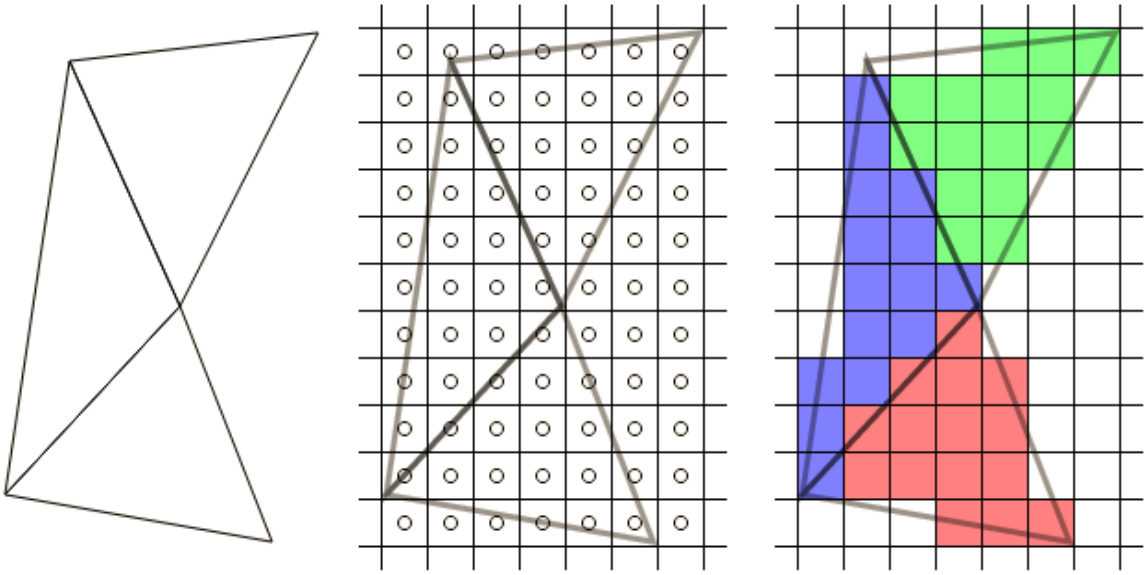


Figura 7 - Scan Conversion en triángulos con aristas compartidas **[11]**

El barrido de conversión es una operación en 2D. El proceso sólo toma en consideración las posición X e Y del triángulo en coordenadas de la ventana para generar los fragmentos. La coordenada Z no se descarta, pero no forma parte directa del proceso de conversión.

El resultado del barrido de un triángulo es una secuencia de fragmentos que cubre la figura del triángulo. Cada fragmento tiene una cierta cantidad de datos asociada. Estos datos contienen:

- La posición en 2D del fragmento en coordenadas de ventana.
- La profundidad Z del fragmento.
- Un número arbitrario de atributos que fueron interpolados a lo largo del triángulo. Estos atributos provienen de la salida de la salida del último Shader (Vertex Shader o Fragment Shader).

Procesamiento de fragmentos

En esta etapa los fragmentos generados de los barridos de los triángulos son procesados. No existe un orden en específico en que se van a procesar los fragmentos generados de un mismo triángulo, esto no tiene importancia ya todos corresponden a pixeles distintos.

Considerando que para un mismo pixel, el orden en que los fragmentos de distintos triángulos son procesados importa, todos los fragmentos de un triángulo son procesados antes de empezar a procesar los fragmentos de otro triángulo.

Cada fragmento es procesado por un Fragment Shader generando una lista de colores, uno para cada buffer en que se está dibujando, un valor de profundidad y un valor de Stencil. El Fragment Shader puede controlar el valor de profundidad y los colores de salida, pero no el de Stencil. Llamaremos *muestra* a la salida del Fragment Shader.

Procesamiento de muestras

Cada muestra debe ser procesada y pasa por las siguientes etapas[6]:

a. Scissor Test: Previamente, el programador debió haber definido un rectángulo en coordenadas de ventana utilizando la función glScissor. Todas las muestras que estén fuera de este rectángulo, serán descartadas.
b. Stencil Test: Compara el valor que ya está guardado en el buffer de stencil con el que viene en la muestra. La función de comparación es configurable.
c. Depth Test: Compara el valor que está en el buffer de profundidad con el que viene en la *muestra*. La función de comparación es configurable.
d. Blending: En este punto, los colores *muestra* se combinan con los colores almacenados en el buffer en el pixel correspondiente. La forma en que los colores son combinados es configurable. Si el Blending está desactivado, el color nuevo reemplazara al que estaba en el buffer.
e. Logical Operations
f. Write Mask: Dependiendo de qué buffers están habilitados para escritura, se actualizan los valores en el pixel correspondiente de cada uno.

Depth Buffer

De forma resumida, el Buffer de profundidad o Z-Buffer (Depth Buffer), es una matriz del mismo tamaño que el buffer de dibujo. Este buffer guarda valores de una dimensión, normalmente punto flotante en [0,1]. Este buffer se utiliza para guardar la profundidad del ultimo fragmento que contribuyó al color del pixel (en caso de que esté habilitada la escritura sobre él).

Este buffer sirve principalmente para que los objetos opacos que estén más cerca de la cámara se dibujen sobre los que están más lejos y no pase que los de más de lejos tapen a los más cercanos.

Esta técnica funciona correctamente cuando los objetos son opacos, pero puede no ser suficiente cuando los objetos son translúcidos. Este problema se verá en la sección 2.3.4.

[6] Los test de Scissor, Stencil y Depth pueden ser desactivados y activados. Cuando una *muestra* falla el test, es descartada.

Stencil Buffer

El Stencil Buffer es un buffer adicional a los buffers de color y profundidad encontrado en tarjetas de video modernas. Este buffer, al igual que los otros, funciona por pixel y debiese tener el mismo tamaño (cantidad de pixeles) que los otros. La principal diferencia es que trabaja con valores entero, usualmente con una profundidad de un byte por pixel. [13]

Este buffer tiene un test (Stencil Test) asociado, el cual es configurable con la función de OpenGL *glStencilFunc*. Cuando los fragmentos fallan el Stencil test, estos se descartan. A su vez, OpenGL tiene la función *glStencilOp* para configurar como el buffer de Stencil actualiza sus valores. El buffer de Stencil puede asociar una función distinta para actualizar su valor en cada uno de los siguientes casos:

- Falla el Stencil Test
- Aprueba el Stencil Test pero falla el Depth Test.
- Aprueba el Stencil Test y falla el Depth Test.

Por ejemplo, podríamos utilizar el Stencil Buffer para contar los fragmentos que pasan el Depth Test. Para esto configuramos el Stencil Test para que los fragmentos siempre pasen. Y hacemos que los valores del Stencil Buffer sólo se actualicen incrementándose en 1 en el caso de que apruebe ambos test.

Las aplicaciones más comunes tienen que ver con limitar el área de dibujado, reflexiones, delineados y sombreados.

2.3.2. GLSL Shaders

OpenGL Shading Language (GLSL) es una parte fundamental e integral de la API de OpenGL. De aquí en adelante, se espera que todo programa escrito utilizando OpenGL, internamente, va a utilizar uno o varios programas escritos en GLSL. Estos "mini-programas" escritos en GLSL son llamados Shaders.

Un Shader es un programa que corre en la GPU, la unidad de procesamiento de la tarjeta de video, y típicamente implementa algoritmos relacionados con efectos de iluminación y sombreado. Sin embargo, un Shader es capaz de hacer mucho más que solamente implementar algoritmos de sombreado. Son capaces de computar animaciones, teselados, e incluso computación genérica. [14]

Los Shaders están diseñados para ser ejecutados directamente en la GPU y por lo general en paralelo. Por ejemplo, un Fragment Shader se ejecuta una vez por cada pixel, con cada ejecución corriendo en paralelo por separado en un hilo de la GPU. El número de procesadores en las tarjetas de video determina cuantos podrán ser ejecutados a la vez. Esto

hace que los Shader sean increíblemente eficientes y proveen al programador de una API simple para poder realizar una gran cantidad de computación en paralelo.

Para mostrar el poder que puede tener paralelizar algún procedimiento en la GPU, podemos ver la siguiente tabla con la cantidad de procesadores de Shaders que tienen las tarjetas de videos actuales: [15]

Modelo	Procesadores de Shaders
GeForce 9800 GTX+	128
GeForce 9600 GSO	96
GeForce GTS 250	128
GeForce GTX 260	192
GeForce GTX 260 Core 216	216
GeForce GTX 280	240
GeForce GTS 450	144
GeForce GTX 460	336
GeForce GTX 480	480
GeForce GTX 590	2x512
GeForce GTX 690	2x1536

Tabla 1 – Cantidad de procesadores de shaders en tarjetas NVidia.

En OpenGL 4.0, existen 5 etapas de Shader: Vertex, Geometry, Tessellation Control, Tessellation Evaluation y Fragment.

Los Shader reemplazan partes del pipeline [7]de OpenGL. Específicamente, hacen que esas partes del pipeline sean programables (introduciendo programas con código arbitrario). El siguiente diagrama muestra una vista simplificada del pipeline de OpenGL con los 5 tipos de Shaders instalados:

[7] Podríamos definir pipeline como una red de procesos de datos conectados en serie. El pipeline de OpenGL procesa las primitivas de acuerdo a la configuración que tenga la máquina de estados en ese momento y genera una imagen final en 2D.

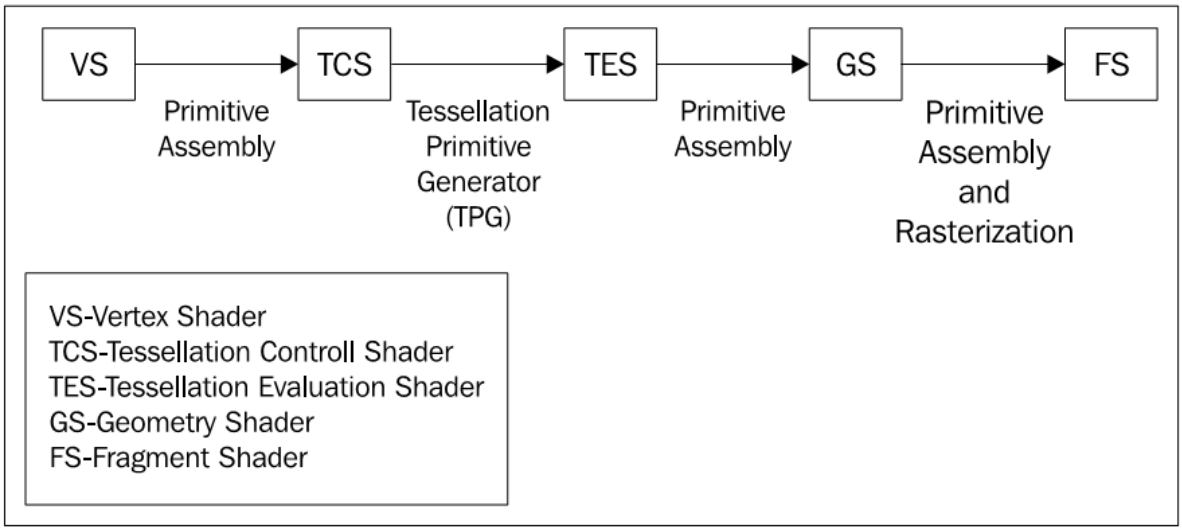


Figura 8 – Orden de las etapas de shaders: Se muestran las distintas etapas de shaders en el orden en que procesan los datos en el pipeline de OpenGL. **[14]**

En general, un Shader recibe sus variables de entrada por variables definidas por el usuario, por variables provenientes de la aplicación principal de OpenGL y/o desde los Shaders de etapas anteriores. Las variables que están disponibles en un Shader y no son las definidas por el programador, están definidas en la especificación de GLSL. Las variables de salida de un Shader también están especificadas por GLSL, y deben ser asignadas antes de que el Shader termine su ejecución.

Tipos de Shader y las operaciones mínimas que deberían hacer:

Vertex Shader (VS):

Se ejecuta una vez por vértice. Los datos correspondientes a las posiciones de los vértices deben ser transformados al espacio de coordenadas *clip*[8]. También, en este Shader se puede computar el color asociado a cada vértices y ser enviado a las siguientes etapas a utilizando las variables de salida.

Fragment Shader (FS):

Se ejecuta una vez por cada fragmento (pixel) de un polígono que está siendo dibujado (típicamente en paralelo). Los datos provenientes del Vertex Shader son, por defecto,

[8] Corresponde a un espacio de coordenadas definido por OpenGL, lo utiliza para determinar qué elementos están dentro del campo de visión.

interpolados manteniendo la perspectiva, y enviados al Fragment Shader. Este Shader determina el color apropiado para cada pixel y lo envía al buffer del cuadro que se está dibujando. La información de la profundidad de los objetos (cual está más cerca de la cámara) se maneja automáticamente, esto es importante para saber, en la imagen final, cuál objeto debe pintase sobre otro.

Geometry Shader (GS):

Se ejecuta una vez por cada primitiva. Tiene acceso a todos los vértices de una primitiva y a todas las variables de entrada asociada a cada vértice. En otras palabras, si una etapa anterior (como un Vertex Shader) provee una variable de salida, el Geometry Shader tiene acceso a ese valor de la variable por todos los vértices de la primitiva. Como resultado, las variables de entrada de un Geometry Shader siempre son arreglos.

Un Geometry Shader puede enviar por sus salida ninguna, una o más primitivas. Esas primitivas no necesariamente tienen que ser del mismo tipo que las que entraron al Shader. Sin embargo, un GS no puede enviar primitivas de más de un tipo distinto por su salida.

Esto le permite a un GS actuar de diferentes maneras, algunos ejemplos:

- Remover primitivas basado en algún criterio.
- Generar geometrías adicionales para refinar una geometría original.
- Computar datos adicionales sobre una primitiva y no modificarla.
- Producir geometrías completamente distintas de las que se le dan por entrada.

Tessellation Shaders (TS):

Cuando los TS están activo, solamente se puede utilizar un tipo de primitiva: *parche* (GL_PATCHES). Intentar renderizar cualquier otro tipo de primitiva resultará en un error. La primitiva *parche* es un trozo arbitrario de geometría (o cualquier tipo de información), el cual está completamente definido por el programador. No tiene ninguna interpretación geométrica aparte de la que se le da en los Tessellation Shaders (son dos).

Los TS son dos Shaders distintos. Uno se llama Tessellation Control Shader (TCS), el cual es responsable de hacer cualquier pre-procesamiento y configurar el Tessellation Primitive Generator (TPG)[9]. Debe definir como las primitivas serán generadas por el TPG (cuántos y qué algoritmos utilizar) y cómo va a producir variables de salida por vértice.

[9] El TPG se encuentra entre el TCS y el TES. Se puede ver como un motor configurable que produce primitivas basado en un set de algoritmos de teselado estándar.

El otro TS es el Tessellation Evaluation Shader (TES). El TES es responsable de determinar la posición (y cualquier otra información) de cada vértice de las primitivas que están siendo generadas por el TPG.

La primitiva *parche* nunca es dibujada realmente. En cambio, el *parche* es utilizado como información adicional para el TCS y el TES. Las primitivas que realmente continúan por el pipeline para ser procesadas son creadas por el TPG.

Los TS son utilizados para obtener un nivel de detalle mayor eficientemente en tiempo real. La cantidad de primitivas que se generan pueden refinar mucho una malla virtualmente. La principal diferencia con las técnicas de manejo de nivel de detalle (LOD) más antiguas, es que en vez de empezar con una malla con una densidad de polígonos grandes y reducir la cantidad de polígonos a medida que se aleja, los TS crean geometrías. Se le da un pequeño número de vértices llamados "puntos de control", para que a partir de esto se generen particiones y geometrías con una precisión que se puede escoger.

Es probable que los Shaders TS no sean de utilidad para esta aplicación, pero si llegase a quedar tiempo sobrante, se podría probar implementar un modo de visualización de mallas con TS para que el usuario pueda ver como se vería su malla más refinada.

Por ahora, sólo está contemplado utilizar VS, GS y FS.

2.3.3. Procesamiento de coordenadas

Utilizando "fixed functionality", el visualizar alguna primitiva, era un procedimiento mucho más simple y el usuario tenía que manejar pocas transformaciones. Bastaba con posicionar la *cámara* y llamar a una serie de instrucciones de la API de OpenGL para crear una primitiva. A la primitiva se le entregaban las coordenadas de los vértices y esta se visualizaba correctamente. [16]

Utilizando Shaders, ahora uno tiene que hacer las transformaciones necesarias para pasar del sistema de coordenadas de la malla hasta el sistema de coordenadas (Clip Space) que necesita OpenGL.

Las transformaciones entre sistemas de coordenadas están representadas por una matriz cuadrada. Para aplicar una transformación de un espacio de coordenadas a otro, el vector de coordenadas de un vértice debe ser multiplicado por la derecha por la matriz que representa dicha transformación.

Para la aplicación actual, se definió que los sistemas de coordenados involucrados serán:

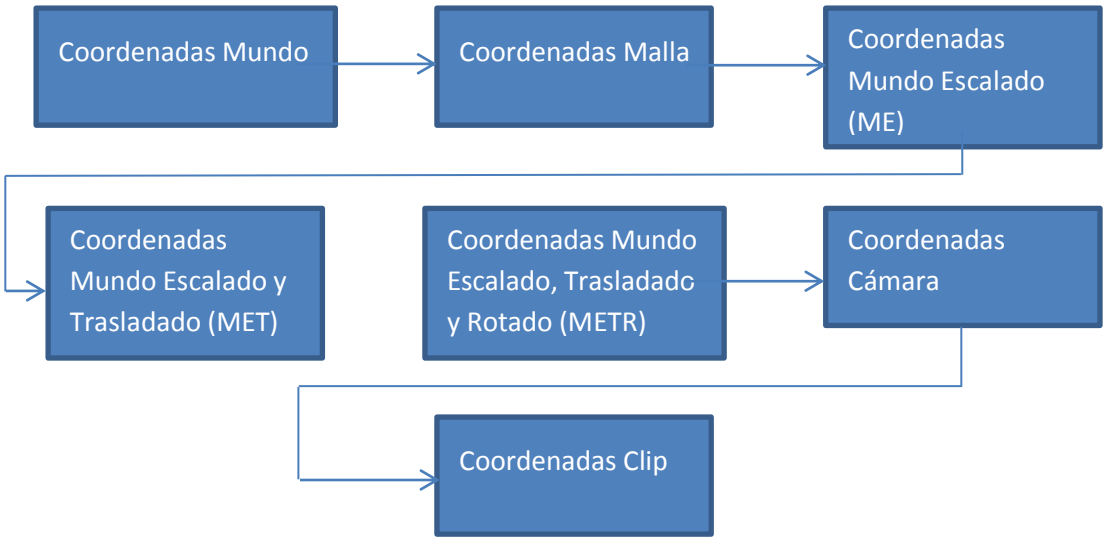


Figura 9 – Transformaciones entre sistemas de coordenadas

Los sistemas ME, MET y METR representan las operaciones de aplicar una traslación, rotación y escalamiento de las coordenadas de los vértices sucesivamente. Estas operaciones son efectivamente cambios de sistema de coordenada y es útil representarlas de esta forma porque podemos usar matrices para aplicarlas.

Para pasar del primer sistema de coordenadas al último, el vector de coordenadas debe ser multiplicado en orden por las matrices de transformación que representan el paso de un sistema al otro. La multiplicación de matrices es una operación no conmutativa, pero si asociativa. Sean M_i las matrices que representan los cambios de sistema de coordenadas, entonces y V el vector de coordenadas:

$$M_n \cdot M_{n-1} \cdot \ldots \cdot M_1 \cdot V = (\ldots((M_n \cdot M_{n-1}) \cdot M_{n-2}) \ldots) \cdot M_1) \cdot V = M_T \cdot V$$

M_T vendría a ser una matriz que representa el paso desde el primer sistema de coordenadas al último. Luego, si podemos multiplicar una vez las matrices entre sí, para obtener esta matriz M_T y luego hacer sólo una multiplicación de matrices por vértice para realizar todo el trabajo de posicionamiento, rotación, escalamiento y perspectivas.

Para renderizar una primitiva, es necesario que los datos correspondientes a los vértices que la definen estén en la memoria RAM de la tarjeta de video. Si hiciéramos la transformación de sistema de coordenadas en la CPU, los resultados quedarían guardados en la RAM principal y después sería muy caro enviar todos los datos a la RAM de la tarjeta de video por cada cuadro. Este es uno de los casos en que el Vertex Shader puede sernos muy útil.

Al VS se le puede entregar como variable de entrada la matriz M_T (previamente calculada en la CPU) y el Shader se encargará de hacer la multiplicación vértice a vértice.

Además, GLSL cuenta con tipos definidos especialmente para el manejo de matrices y vectores, junto con sus operaciones. Los fabricantes de hardware se preocupan de que estas operaciones sean implementadas de forma muy eficiente, por lo tanto, lo ideal es utilizarlas y hacer los cálculos por este medio.

2.3.4. Manejo de Transparencia en OpenGL

OpenGL permite renderizar primitivas translúcidas (deja ver lo que hay detrás), utilizando el canal alfa de los colores y *blending*.

Blending es una etapa del proceso renderizado de OpenGL. En esta etapa, se combina el color de un fragmento junto con el color del pixel correspondiente. OpenGL permite configurar (limitadamente) la función con que se realiza el cálculo del color final. [17]

Lamentablemente, para renderizar correctamente una escena con múltiples objetos traslucidos, no es tan simple como habilitar el *blending*. A continuación se verán los distintos problemas que existen.

Cuando existen objetos translúcidos, el orden de profundidad en que están es importante. Este efecto se puede apreciar en las siguientes imágenes:

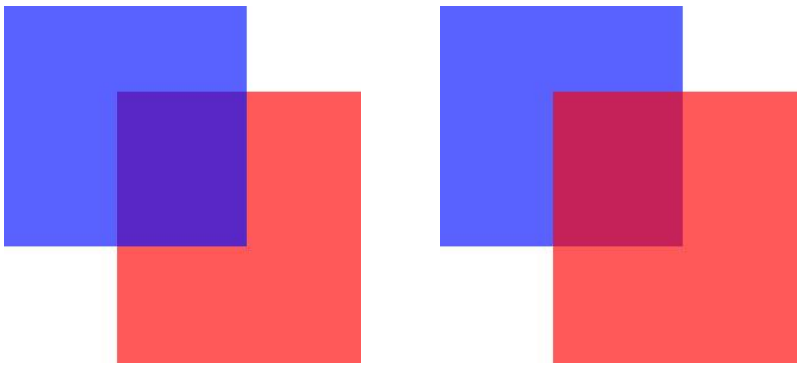


Figura 10 - Orden de transparencia: En la figura izquierda, el cuadro azul está más cerca. En la figura derecha, el cuadro rojo está más cerca.

El color de la zona que se mezcla nos da la percepción de que elemento está más cerca. Por ejemplo, suponiendo que en la escena original el cuadrado azul está en frente del rojo (más cerca de la cámara), si no se toma ninguna precaución y se dibujan los dos cuadrados, es posible que nos veamos en los siguientes escenarios:

- El cuadrado rojo se procesa primero y se dibuja actualizando los valores de profundidad en el Z buffer. Luego, se dibuja el cuadrado azul, como el cuadrado azul está más cerca, todos los fragmentos pasan el test de profundidad y se mezclan con la imagen que ya estaba dibujada. El resultado obtenido es el correcto (imagen izquierda de la figura 10).
- En el otro caso, el cuadrado azul se procesa primero y se dibuja actualizando los valores de profundidad en el Z-buffer. Luego, al dibujarse el cuadrado rojo, los fragmentos que corresponden a los pixeles que modifico el cuadrado azul no van a pasar el test de profundidad, ya que el cuadrado rojo está más lejos. La imagen obtenida está equivocada y obtenemos:

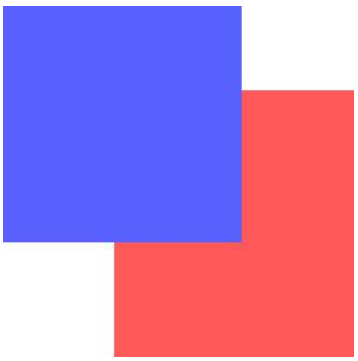


Figura 11 – Fallo de blending por causa del Depth Test: El Depth Test descartó los fragmentos del cuadrado rojo, aunque el azul se translúcido.

Con el ejemplo anterior, podemos ver que si el test de profundidad de OpenGL está activado y dibujamos primero los objetos translúcidos más cercanos, estos impedirán que se dibujen bien los que están atrás, incluso si el objeto de atrás es opaco. Para que la escena, con estas configuraciones, se dibuje bien, deben dibujarse los elementos partiendo desde el más lejano hasta el más cercano.

Una forma de abordar este problema, es ordenar las primitivas antes de enviarlas por el pipeline de OpenGL. Este enfoque, en el contexto de la aplicación desarrollada tiene dos problemas:

- Para poder recorrer las primitivas en orden de profundidad, se necesitaría algún tipo de indexación espacial, la cual sería costosa de implementar en memoria y en complejidad. No basta con ordenarlas al cargar el modelo, ya que el modelo puede ser rotado por el usuario y el orden cambia.
- No resuelve el problema completamente. En el caso, por ejemplo, de que existan 3 triángulos traslucidos A (verde), B (azul) y C (rojo), tal que:
    - A tiene un vértice encima de B y un vértice debajo de C.
    - B tiene un vértice encima de C, pero debajo de A.

- C tiene un vértice encima de A y un debajo de B

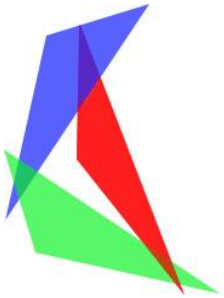


Figura 12 – Blending incorrecto para 3 triángulos entrelazados (caso 1).

Dadas estas condiciones, no existe un orden valido para las primitivas, siempre ocurriría que algún pedazo de triángulo no se dibujaría debido a que no pasa el test de profundidad. Por ejemplo, si se dibujaban en el orden: azul, verde, rojo, la imagen final quedaría de la siguiente forma:

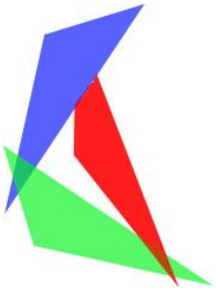


Figura 13 - Blending incorrecto para 3 triángulos entrelazados (caso 2).

En la imagen, los fragmentos del triángulo rojo que quedaron debajo del triángulo azul fueron descartados por el test de profundidad. La solución vendría a ser dividir estos triángulos en triángulos más pequeños que no cumplan estas condiciones, luego, podrían ser ordenados, pero esto ya agrega demasiada complejidad a la solución.

Otra forma de abordar el problema, es dibujar primero los elementos opacos con el test de profundidad activado y con el buffer de profundidad en modo escritura/lectura. Luego, se dibujan los elementos translúcidos, con el test de profundidad activado, pero con el buffer de profundidad como sólo lectura. De esta forma, ningún fragmento traslucido va a bloquear a otro fragmento traslucido. La segunda parte que hay que tomar en cuenta en este caso es la función de *blending*. Con la función que viene por defecto activada en OpenGL, el orden en que los fragmentos se combinan con el color del pixel importa, lo que continúa generando problemas, los fragmentos traslucidos ya no pueden bloquear a otros haciendo que sean descartados, pero el color final de mezclado puede ser incorrecto si las primitivas no se dibujan en orden correcto. Retomando el caso de los 3 triángulos “entrelazados”, el

color de la esquina del triángulo rojo que se perdía, ahora quedaría mal mezclado al dibujarlo en el mismo orden:

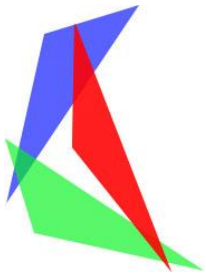


Figura 14 - Blending incorrecto para 3 triángulos entrelazados (caso 3).

Como se puede ver, el triángulo rojo parece estar sobre los otros dos, pero la escena original que se describió no era así, por lo que está mal dibujada. Como opción ingenua final, podemos cambiar la función de blending de OpenGL por una que no sea dependiente del orden en que lleguen los fragmentos, normalmente, esta será una función que sume los colores solamente. El problema generado con esto, es que a medida que se superponen fragmentos, el pixel se satura rápidamente:

Figura 15 - Blending incorrecto para 3 triángulos entrelazados (caso 4).

Los pixeles de la imagen donde el aporte fue de más de un fragmento están más saturados de lo que deberían, y este efecto se nota más a medida que hay más fragmentos aportando a un pixel. Además, se pierde la noción de la profundidad, esto es lógico considerando que el color final es el mismo independiente del orden en que se dibujan los triángulos (debido a la función de blending).

Finalmente, ordenar las primitivas no soluciona completamente el problema, a menos que los triángulos también se subdividan cuando ocurran casos especiales como el expuesto, y además, la aplicación desarrollada está pensada para trabajar con mallas de gran cantidad de elementos y esta solución es muy costosa, por lo que se descartó. Considerando lo anterior, para abordar el problema, se utilizó una serie de técnicas donde el enfoque

principal se basa realizar el proceso de dibujado de tal forma que no importe el orden en que se envíen las primitivas por el pipeline. Este enfoque se llama Order Independent Transparency (OIT). A continuación se expondrán 3 algoritmos que siguen este enfoque para dar solución al problema.

2.3.4.1. A-Buffer

Resumidamente, la técnica del A-Buffer consiste en un arreglo de texturas en 2D del tamaño de la pantalla. Además, existe un buffer adicional que actúa como contador, del mismo tamaño que los otros. El buffer contador, en cada posición (x, y), tiene el número de fragmentos con posición (x, y) que sean generado.

Luego, cada vez que se genera un fragmento que iría en el pixel (x, y), se incrementa el contador correspondiente, y se guarda el color del fragmento en la capa que indique el contador, en la posición (x, y). Esto permite tener una lista de fragmentos por cada pixel, desordenada en cuanto a profundidad y un contador que indica el tamaño de la lista

En una segunda etapa de procesamiento, por cada pixel se ordenan por profundidad los fragmentos que están en el A-Buffer y se combinan adecuadamente los colores. [18]

La ventaja de esta técnica es que solo necesita procesar una única vez la geometría del modelo para llenar los buffers de colores, por lo que computacionalmente no es costosa. Por otro lado, dependiendo de la cantidad de texturas que se utilicen, el costo en memoria de la tarjeta de video puede llegar a ser alto.

2.3.4.2. Linked List per Pixel

Esta técnica es similar a la de A-Buffer, con la excepción de que en vez de tener un número de capas fijo representado por una textura en 2D del tamaño de la pantalla, aquí se tienen listas enlazadas por cada pixel y los fragmentos se van agregando a estas listas. La gran ventaja está en el ahorro de memoria: las listas crecerán sólo mientras sea necesario, mientras que en el A-Buffer hay muchas posiciones que probablemente no se utilizarán.

Esta técnica parece ser la que tiene mejores resultados: Necesita una sola pasada por la geometría del modelo que se esté procesando, utiliza la memoria justa y necesaria, y los resultados son precisos.

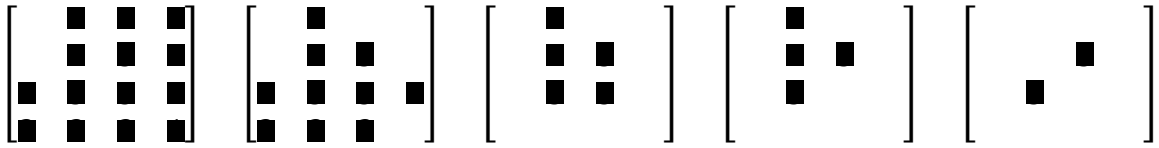
El inconveniente es que las listas enlazadas por pixel están sólo disponibles en DirectX11. En la API de OpenGL (sin extensiones) aún no hay algo que permita acceder a la misma funcionalidad, por lo que queda descartada por el momento para la aplicación. [19]

2.3.4.3. Depth Peeling

Depth Peeling es una técnica que consiste en generar la imagen final a partir de la combinación de distintas capas de profundidad de la escena en orden (puede ser de la más cercana a la más lejana o de la más lejana a la más cercana. Cada capa que se procesa requiere que se procese completamente la geometría del modelo. Por ejemplo, si la imagen final fuera de 4*4 pixeles y en cada pixel vemos la cantidad de fragmentos que contribuyen al color final en la siguiente matriz:

$$\begin{bmatrix} 0 & 4 & 1 & 1 \\ 0 & 4 & 5 & 1 \\ 2 & 5 & 3 & 2 \\ 2 & 2 & 2 & 1 \end{bmatrix}$$

la imagen tiene una complejidad de profundidad de cinco (mayor cantidad de fragmentos que aportan a un pixel), por lo que la imagen final necesita procesar a lo más cinco capas para generarse. Entonces, si las capas que se obtienen en cada pasada, procesando de la más cercana a la más lejana, se obtiene lo siguiente:



Capa 1 Capa 2 Capa 3 Capa 4 Capa 5

finalmente, estas capas son combinadas en orden para obtener una imagen precisa y sin errores. Dividir la imagen en capas de acuerdo a la profundidad sirve para poder ordenar por pixel a los fragmentos que van contribuyendo. Es similar a lo que podemos ver en el A-Buffer, donde también la escena final se separa en capas de profundidad, pero en este caso se procesa una capa a la vez y esta mezcla inmediatamente con la imagen final, por lo que no es necesario guardar un arreglo de texturas para las capas, sino que se tiene una sola textura donde se guarda la capa que se está procesando.

Para el algoritmo, primero, se modifica la función de *blending* de OpenGL, para que los colores correctos se calculen cuando las primitivas se dibujan desde la más cercana a la más lejana de la cámara.

La técnica comienza por renderizar la escena de forma normal con un test de profundidad activado. El test de profundidad se configura para que se guarde el fragmento de profundidad mínima solamente. En las siguientes pasadas, el buffer de profundidad de la pasada anterior, se pasa al Fragment Shader como textura. Para evitar problemas de lectura-escritura (puede haber más de un fragmento procesándose en paralelo para un mismo pixel), el buffer de profundidad de lectura y el buffer de escritura son distintos, y son intercambiados en cada pasada (ping pong).

En cada pasada, si la profundidad del fragmento es menor o igual a la que esta guardada en el buffer de la pasada anterior, el fragmento es descartado (ya fue considerado). Luego, como el test de profundidad esta puesto para que se guarde el de menor profundidad, cada pasada, el único fragmento que no es descartado por pixel, es el que le seguía en profundidad al fragmento que fue considerado en la pasada anterior.

Los fragmentos de menor o igual profundidad se descartan en el Fragment Shader utilizando el buffer de la pasada anterior como textura y los fragmentos más lejanos que el siguiente que debería considerarse, son descartados por el Depth Test configurado. [20]

El algoritmo termina cuando se han procesado el número de capas que se especificó arbitrariamente.

La desventaja principal de esta técnica es que por cada capa que se procesa, la geometría del modelo se procesa completamente, y en modelos grandes puede ser muy lenta. Por otro lado, con un número adecuado de pasadas, dependiendo de la complejidad de la profundidad de la escena, se pueden generar imágenes con resultados muy precisos.

2.3.4.4. Dual Depth Peeling

Es similar al Depth Peeling original, pero avanza desde atrás hacia adelante y al mismo tiempo desde adelante hacia atrás, por lo que el número de pasadas disminuye a la mitad. El problema es que sólo es posible tener un único buffer de profundidad, lo que aumenta la complejidad en la implementación. Los detalles de la implementación de esta técnica no serán cubiertos en este trabajo, si se desea obtener información más detallada se recomienda consultar la fuente. [20] [21]

2.3.4.5. Weighted Sum

Las ecuaciones de *blending* que dan resultados más realísticos son dependientes del orden de los fragmentos que participan en esta.

La técnica Weighted Sum propone simplificar la ecuación de *blending* para hacerla independiente del orden de los fragmentos. La estrategia de la propuesta consiste en expandir la ecuación e ignorar los términos que dependen del orden. Este método produce buenos resultados para valores pequeños de alfa. Sin embargo, produce imágenes demasiado brillantes o demasiado oscuras (dependiendo de los términos ignorados) para alfas mayores a 30%.

Una forma de implementar esta técnica es modificar la función de *blending* para que por pixel. Si *Cd*: Color del pixel (RGB), *Ad*: alfa almacenado en el pixel, *Cs*: color del nuevo fragmento que se va a combinar y *As*: alfa del nuevo fragmento, las funciones de *blending* son las siguientes:

$$Nuevo\ color\ en\ pixel = C_d \cdot 1 + C_s \cdot A_s$$

$$Nuevo\ alfa\ en\ pixel = A_d \cdot 1 + A_s \cdot 1$$

Además, utilizamos como buffer para guardar la imagen que se renderiza una textura con color de fondo (0, 0, 0 ,0) y desactivamos el test de profundidad.

Finalmente, se procesan los valores que quedaron guardados en la textura mediante la siguiente ecuación:

$$Color\ final\ del\ pixel = \sum [A_s \cdot C_s] + C_{bg} \cdot (1 - \sum A_s)$$

Donde $\sum[As\ Cs]$ es la suma ponderada que esta guardada por pixel en la textura creada en el paso anterior y $\sum As$ es la suma de alfas guardada también en la misma textura. C_{bg} es el color del fondo.

Como los términos dependientes del orden se descartan, el color resultante no posee ninguna información con respecto a las capas de profundidad ni el orden de estas. Suponiendo que no estuviera el problema de sobre-oscurecimiento o sobre-iluminación, la imagen obtenida de los cuadrados superpuestos (figura 10), quedaría de la siguiente forma:

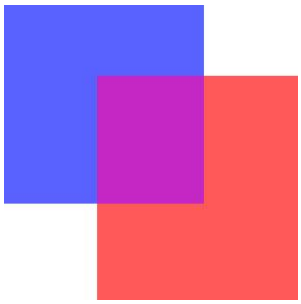


Figura 16 – Blending de Cuadrados superpuestos utilizando weighted sum: Como se puede ver, no es posible distinguir que cuadrado está más cerca de la cámara.

La principal ventaja de esta técnica es la velocidad. Necesita procesar sólo una vez la geometría para obtener todos los valores necesarios. [20]

2.3.4.6. Weighted Average

Es una mejora de la técnica Weighted Sum. La principal diferencia consiste en que en la ecuación de *blending*, para cada pixel, se toma en cuenta la cantidad de fragmentos que contribuyó. En la ecuación de *blending* ya no se descartan términos para hacerla independiente del orden, sino que se trabaja bajo el supuesto de que todos los fragmentos que contribuyen a un mismo pixel son de un mismo color y este color es el promedio ponderado de los colores de los fragmentos que contribuían originalmente. Al considerarse todos los fragmentos de este color promediado, no es importante el orden. La explicación completa puede ser encontrada en la publicación original de la técnica, por ahora se cubrirá lo suficiente para implementarla.

Para implementarla, ahora es también necesario, contar la cantidad de fragmentos que contribuyen a un pixel por cada pixel. Luego, la función para calcular el color de la imagen final está dada por:

$$Color\ final\ del\ pixel = C \cdot (1 - (1 - A)^n) + C_{bg} \cdot (1 - A)^n$$

Donde *n* es la cantidad de fragmentos que contribuyeron al pixel, $A = \frac{\sum A_s}{n}$ y $C = \frac{\sum [A_s \cdot C_s]}{\sum A_s}$.

La ventaja de esta técnica por sobre Weighted Sum, es que funciona bien para alfas pequeños y grandes. Pero aun así, tiene el problema de la perdida de información en cuando al orden de profundidad de los objetos. [20] [21]

2.4 OpenGL Mathematics Library GLM

Las matemáticas son la base de la computación gráfica. En las versiones más antiguas, OpenGL preveía el soporte para manejar las transformaciones de las coordenadas y proyecciones usando pilas de matrices. En las versiones más nuevas (4.0 por ejemplo), todas estas funcionalidades han sido removidas, por lo que el programador debe proveerlas.

Por esta razón, entre otras, la aplicación debe poder realizar diversos cálculos que involucran vectores y/o matrices. Para ahorrar el tiempo que requeriría implementar toda esta lógica, se ha escogido la librería GLM.

GLM es una librería matemática de sólo headers escrita en C++. Toda la implementación está incluida entre los archivos de header. No requiere de una compilación separada ni de tener que enlazarla con el programa que se está desarrollando.

La librería está diseñada para ser utilizada en aplicaciones gráficas, y está basada en la especificación de GLSL.

La librería provee clases y funciones diseñadas e implementadas con las mismas convenciones de nombres y funcionalidades que GLSL. El objetivo de esto es que si un programador conoce GLSL, conocerá GLM también, por lo que le será muy fácil de utilizar.

Por otro lado, tampoco está limitada a las características de GLSL. GLM posee un sistema de extensiones bajo las mismas reglas que las extensiones en GLSL, el cual provee capacidades extendidas. [22]

GLM está escrita en C++98, pero puede utilizar ventajas de C++11 cuando son soportadas por el compilador. Es una librería independiente de la plataforma, y es soportada oficialmente por los siguientes compiladores:

- Clang 2.6 o superior
- CUDA 3.0 o superior
- GCC 3.4 o superior
- Intel C++ Composer XE 2013 o superior
- Visual C++ 2005 o superior
- Cualquier compilador de C++98 o C++11

Las funciones que GLM provee y que serán útiles para el proyecto serán principalmente:

- Módulo, producto punto y producto cruz en vectores.

- Multiplicación entre vectores y matrices.
- Determinante, traspuesta e inversa de matrices.
- Distancia entre vectores.

2.5. Qt (QtCreator) o GTK+ (Glade)

GTK+ y Qt cuentan con paquetes de herramientas para confeccionar interfaces de usuario. Ambos son de código abierto.

Las diferencias más significativas para el proyecto:

La API de Qt está desarrollada desde un comienzo para ser utilizada con C++ y con orientación a objetos. GTK+ está desarrollado en C, y para poder utilizar GTK+ con C++ hay que utilizar una interfaz aparte (GTKmm). Como GTKmm fue agregada sobre GTK+, no presenta un diseño de interfaz tan bien orientado a objetos, ya que GTK+ no está diseñado así.

GTK+ no tiene un IDE oficial. Mucha gente utiliza Glade para construir la GUI visualmente y después codificar la lógica de esta.

Qt tiene como IDE oficial a QtCreator. QtCreator funciona como IDE y además incorpora una herramienta de diseño, con la cual fácilmente se pueden dibujar los elementos de una interfaz y las conexiones entre estos. Qt posee un compilador llamado User Interface Compiler (uic), el cual compila las GUI diseñadas en código eficiente y nativo de C++. El código generado es fácilmente integrado con el resto de la aplicación. [23]

El alumno realizó pruebas confeccionando GUI's utilizando QtCreator y Glade 3. En el ambiente de Ubuntu 11.10 (Oneiric Ocelot), Glade 3 se comportaba de forma muy inestable, arrojando Segmentation Faults seguido y ocasionalmente cerrándose. QtCreator corrió sin ningún problema/error, y nunca se cayó la aplicación.

Por las razones expuestas, se escogió como librería para el desarrollo de la GUI de la aplicación a Qt.

2.6. Otros Visualizadores

2.6.1. Geomview

Es un programa interactivo en 3D para Unix. También puede ejecutarse en Windows usando el Cygwin (simula un entorno similar al de Linux para Windows).

Geomview permite ver y manipular objetos tridimensionales. Utilizando el mouse permite realizar operaciones básicas como: rotar, trasladar, acercarse, etc. También tiene la ventaja de que puede manejar múltiples objetos y cameras en una misma escena. Permite obtener capturas de pantalla del modelo y guardarlas en diversos formatos de imagen. Soporta datos de tipo: polígonos y vértices (.off), cuadriláteros, mallas rectangulares, vectores, superficies de Bézier, entre otros. [24]

Además de poder usarse como un visualizador independiente para ver objetos estáticos, puede ser utilizado como motor gráfico para otros programas que produzcan geometrías que cambien dinámicamente.

Geomview es una herramienta de visualización de mallas, pero no permite evaluarlas.

La última versión de Geomview salió en agosto del 2007. El desarrollo del programa esta descontinuado.

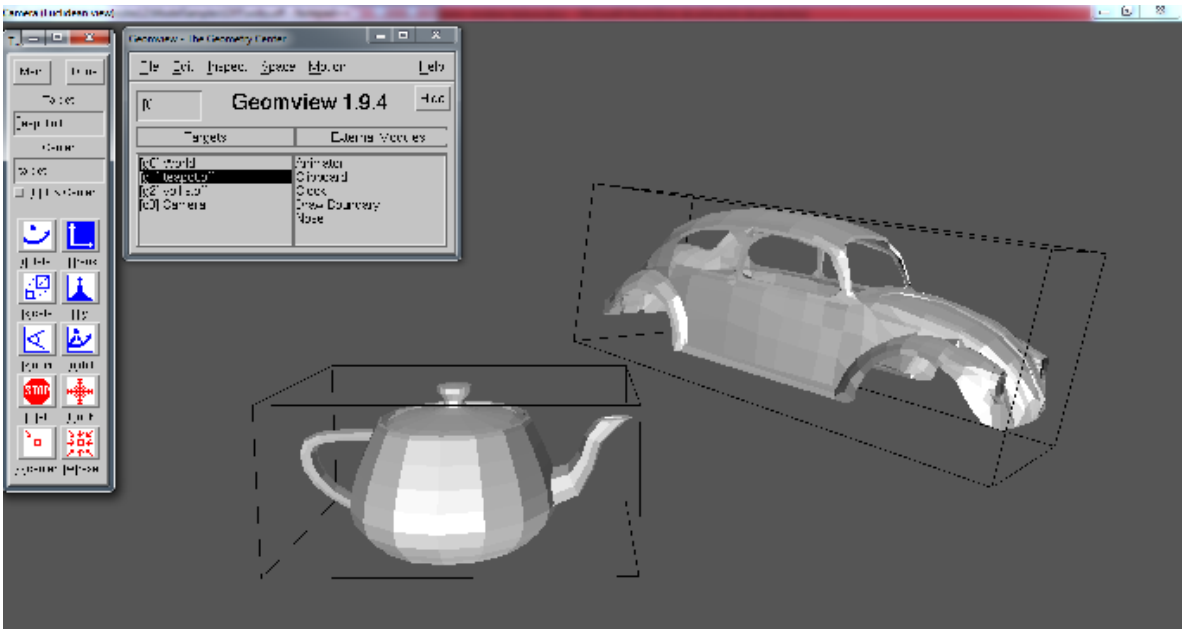


Figura 17 - Geomview.

2.6.2. TetView

Es una pequeña aplicación gráfica para visualizar mallas de tetraedros. Fue creado específicamente para visualizar y analizar la salida y entrada de archivos de TetGen. Sin embargo, tiene las características generales de un visualizador de objetos geométricos, mallas de tetraedros y mallas de triángulos.

TetView es capaz de abrir formatos de mallas de superficie y mallas de tetraedros. Entre estos formatos encontramos: ele, node, smesh, off, ply, etc. Este visualizador no permite realizar ningún tipo de evaluación sobre las mallas, pero puede realizar cortes para mirar su interior utilizando planos. [25]

La versión ejecutable de TetView está disponible para varios sistemas operativos: Windows (9x/NT/2000/XP), Linux, Mac OS X, SGI IRIX64, SGI IRIX N32. Pero esta descontinuado desde el 2004.

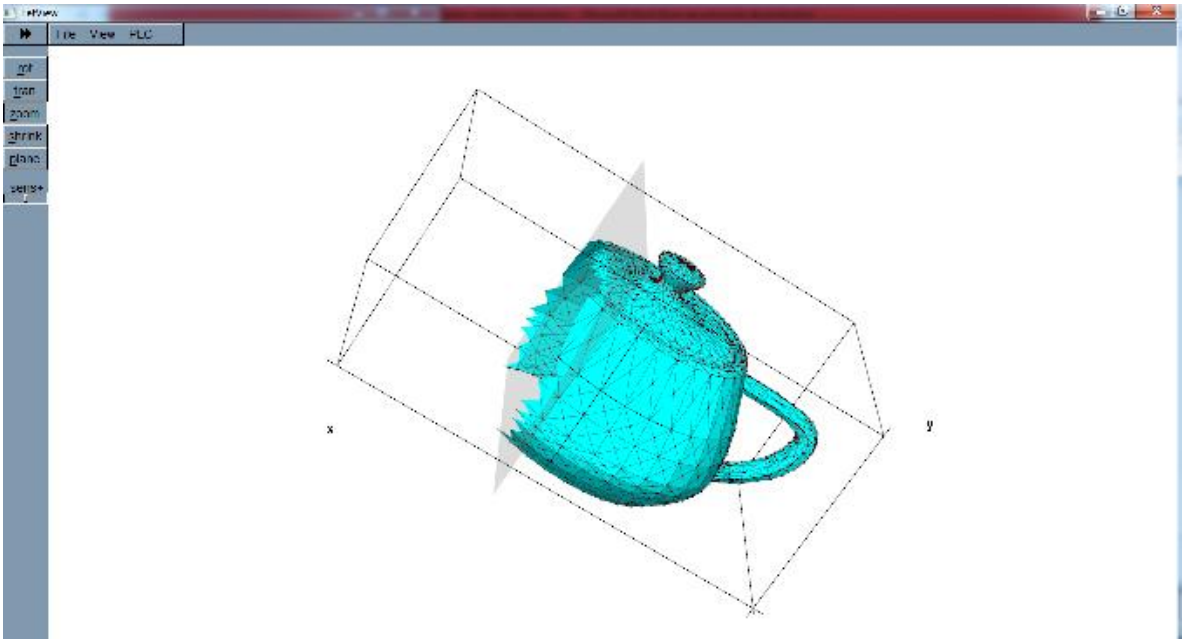


Figura 18 - Tetview.

2.6.3. MeshLab

Es un sistema de procesamiento y edición de mallas no estructuradas de triángulos en 3D, de código abierto, portable y extensible. El sistema está diseñado para asistir el procesamiento de mallas grandes obtenidas desde escaneos en 3D, proveyendo un conjunto de herramientas para editar, limpiar, arreglar, inspeccionar, renderizar y convertir las mallas de un formato a otro. [26]

Las características del programa son:

- Selección interactiva y borrado de porciones de la malla.
- Interfaz de herramientas para seleccionar, suavizar y colorear mallas.
- Permite cargar mallas en diversos formatos: **PLY, STL, OFF, OBJ, 3DS, COLLADA, PTX, V3D, PTS, APTS, XYZ, GTS, TRI, ASC, X3D, X3DV, VRML, ALN**
- Permite exportar las mallas en diversos formatos: **PLY, STL, OFF, OBJ, 3DS, COLLADA, VRML, DXF, GTS, U3D, IDTF, X3D**
- Soporta cargar modelos de nubes de puntos.
- Filtros de para limpiar mallas:
    - Remoción de vértices duplicados, no referenciados, caras nulas.
    - Remoción de pequeños componentes aislados.
    - Unificación coherente de normales e inversión de polígonos.
    - Llenado de hoyos.

- Otros.

Este visualizador parece ser el más completo entre los investigados, pero aún así no cuenta con un módulo de evaluación de mallas.

MeshLab actualmente está en desarrollo y la última versión a la fecha data del 3 de agosto del 2012.

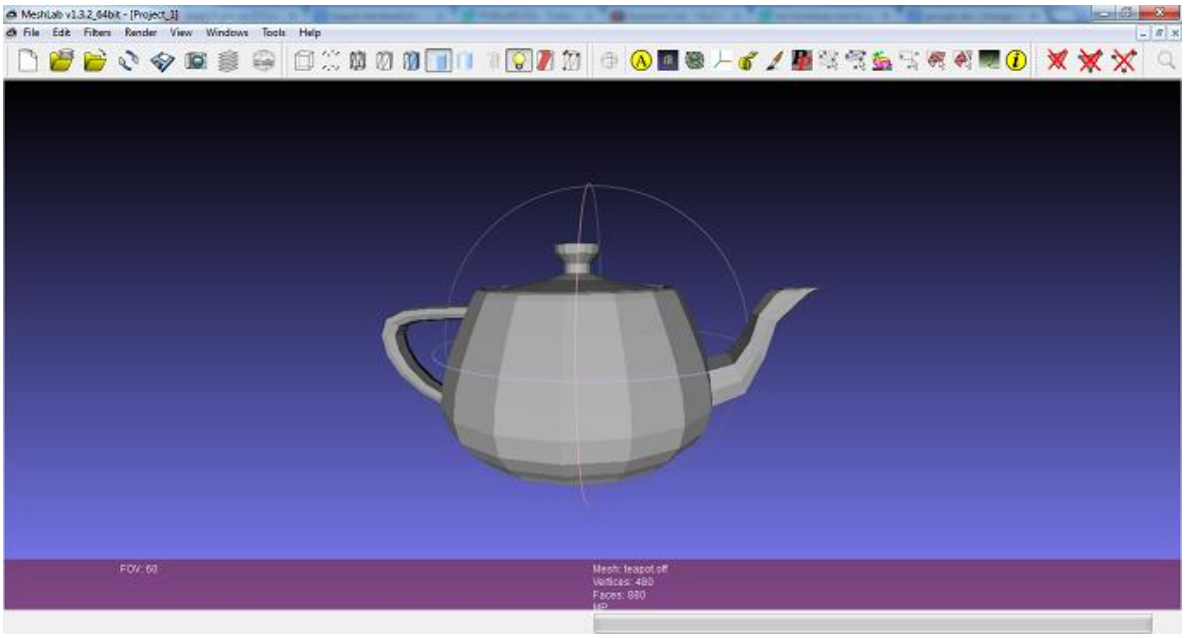


Figura 19 - Meshlab.

3. Diseño

A continuación se presentan los requerimientos que la aplicación debe cumplir y las técnicas de ingeniería de software aplicadas al diseño. Finalmente, se expondrá el diseño de los distintos módulos y las estructuras de datos utilizadas.

3.1. Requerimientos

El software debe proveer las siguientes funcionalidades ordenadas por área:

Características Principales:

Vista:

- Rotación libre de la cámara.
- Traslación libre de la cámara.
- Zoom in, Zoom out.
- Volver al modelo a su posición original.
- Visualizar la malla como una malla de alambres.
- Visualizar la malla que define la superficie del objeto.
- Visualizar la malla como nube de puntos (sólo vértices).
- Visualizar sólo las caras externas, sólo caras internas, ambas.
- Visualizar sólo poliedros externos, poliedros internos, ambos.
- Visualizar la intersección de la malla con distintas geometrías: Planos, Cubos, Esferas.
- Visualizar los identificadores (índices) de los distintos elementos.
- Selección de colores para el background y elementos de la malla.
- Selección de algoritmo de iluminación.

Selección:

- Elegir tipos de elementos a seleccionar: vértices, polígono, poliedro.
- Seleccionar elementos según criterio definido por el usuario, basado en las propiedades de los elementos.

- Seleccionar elementos a partir de la intersección de la malla con distintas geometrías: Planos, Cubos, Esferas.
- Expandir selección a los elementos vecinos de los elementos que ya están seleccionados. Además, el usuario puede especificar un ángulo de tolerancia para limitar como se propaga la selección por una superficie[10].
- Seleccionar elementos con un cuadro arrastrando el mouse.
- Seleccionar los elementos dado su identificador.

Manejo de archivos:

- Lectura de archivos de malla al menos con formatos usados por Tetview (.node y .ele); y formatos .off, .md3 y uno local. La lectura de archivos debe estar optimizada para que tarde lo menos posible en construir la malla.
- Escritura de archivos en los formatos mencionados anteriormente.
- Posibilidad de guardar sólo elementos seleccionados.
- Al cargar las mallas, se chequeará la consistencia de esta.

Estadísticas sobre elementos de la malla:

- Número de elementos de cada tipo de poliedro que contiene la malla.
- Número de caras/vértices/poliedros.
- Mínimos y máximos de cada coordenada
- Rangos de valores encontrados para cada uno de los criterios geométricos aplicados sobre los elementos de la malla.

Criterios de evaluación de los elementos de la malla:

- Razón de aspecto.
- Ángulo diedro.
- Ángulo sólido.

[10] La selección se propagará a los vecinos mientras el ángulo diedro entre las caras superficiales vecinas sea menor que uno escogido por el usuario.

Características Generales:

- El diseño debe ser extensible tal que se puedan agregar fácilmente nuevos criterios de evaluación, nuevas formatos de archivos soportados, nuevas técnicas de selección, entre otros aspectos.

3.2. Pre-diseño de software

Al estudiarse los requerimientos se eligieron los patrones de diseño que serían de utilidad para el proyecto. También, se decidió que se utilizaría un lenguaje orientado a objetos. A continuación se expone lo antes mencionado.

3.2.1. Programación orientada a objetos

"La Programación orientada a objetos (OOP) es un paradigma de programación que se basa en la utilización de "objetos" y su interacción, para diseñar aplicaciones y programas computacionales. Esta técnica de programación incluye características tales como encapsulación, modularidad, polimorfismo y herencia." [27]

Para dar una idea más clara de lo que engloba el paradigma de OOP, se explicaran a continuación las características antes mencionadas:

- Encapsulación: Es el ocultamiento de las características, estado y comportamiento interno de un objeto de manera que todo otro objeto que interactúe con él, solo lo pueda hacer a través de un grupo de funciones bien definidas para este fin. [28]
- Modularidad: Permite que el código se agrupe en módulos haciendo que el programa sea más fácil de extender, o sea, que se puedan agregar nuevos módulos sin tener que rediseñar. Además, se consiguen componentes reutilizables, ya que al dividir la aplicación en módulos, estos por si solos pueden verse como pequeñas soluciones a sub-problemas y estos pueden aparecer en otros contextos o proyectos.
- Polimorfismo: Posibilita que diferentes objetos respondan a un mismo mensaje de diferentes maneras, siendo la respuesta dependiente del tipo de objeto. [28]
- Herencia: Permite construir nuevas clases a partir de otras ya existentes, las cuales heredan características de estas últimas que tienen en común. A su vez, permite que similares objetos compartan comportamientos y propiedades comunes. [28]

Como uno de los objetivos planteados es que la aplicación sea extensible, los beneficios mencionados anteriormente son de gran utilidad, ya que nos permiten hacer que se puedan

agregar nuevos componentes fácilmente y mantener aún un código legible, ordenado y modular, aun cuando el proyecto sea grande. Aprovechando estos beneficios se le pueden agregar componentes a la aplicación sin la necesidad de un rediseño y además hay una gran reutilización de código. [28] [29]

3.2.2. Patrones de diseño

"Un patrón de diseño nombra, abstrae, e identifica los aspectos claves de un diseño estructural común que es útil para la creación de diseños orientados a objetos reusables. El patrón de diseño identifica la participación de clases e instancias, sus roles y colaboraciones, y la distribución de sus responsabilidades. Cada patrón de diseño se enfoca en un problema de diseño orientado a objeto particular. Describe cuando es aplicable y que consecuencias; ventajas y desventajas tiene su uso." [30]

Los patrones son buenas prácticas formalizadas que el programador debe implementar el mismo en la aplicación, no son diseños terminados ni implementaciones que puedan ser utilizadas directamente, sino que más bien son diseños abstractos. Los patrones que principalmente serán parte del diseño son Factory, Strategy, Visitor, Double Dispatch y Singleton [30] [31].

Factory: Se utiliza con el objetivo de crear objetos escondiendo la lógica de inicialización a las clases que lo utilicen y evitando duplicación de código en donde haya que inicializar objetos. También, provee al programador de una interfaz simple y limpia para generar objetos.

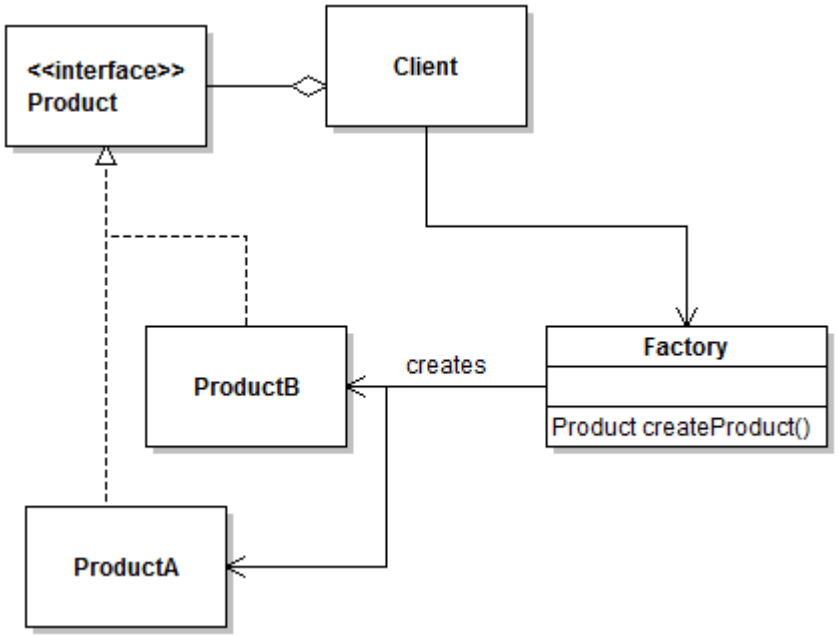


Figura 20 - Patrón de diseño Factory

Strategy: Este patrón sirve para tener múltiples algoritmos distintos con una interfaz de aplicación común, de los cuales, se decidirá cual aplicar en tiempo de ejecución. El cliente que hace uso de las estrategias, se abstrae de las implementaciones de estas y solo las utiliza a través de una interfaz común y simple, esto es útil, por ejemplo, para manejar los distintos algoritmos de renderizado. Este patrón aporta en gran manera a la extensibilidad de la aplicación, ya que por ejemplo, para tener más estrategias de renderizado, basta simplemente con agregar una nueva clase que implemente la interfaz común.

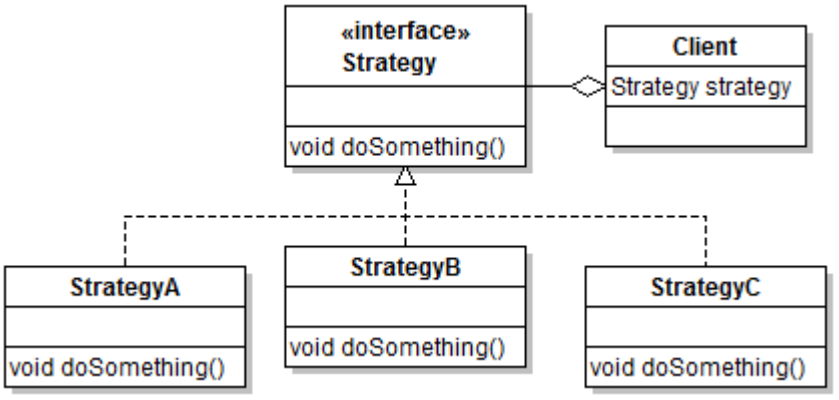


Figura 21 - Patrón de diseño Strategy

Singleton: Se utiliza para garantizar que una clase sólo tenga una instancia y proporcionar un punto de acceso global a ella. Dicha clase se debe preocupar de generar la instancia la

primera vez que se llame a getInstance() y almacenar una referencia a esta, luego, las siguientes llamadas a getInstance() deberían retornar esta única instancia existente. Usualmente este patrón es utilizado para realizar un manejo centralizado de algún tipo de recurso.

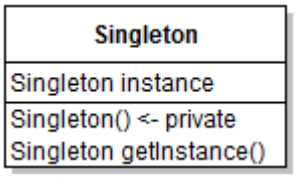


Figura 22 - Patrón de diseño Singleton

Double Dispatch: la idea del patrón es poder escoger qué función ejecutar (cuando hay más de uno con el mismo nombre) en tiempo de ejecución, dependiendo de los dos objetos involucrados en la llamada. Es muy utilizado en conjunto al patrón Visitor.

Visitor: El patrón Visitor puede ser analizado separándolo en dos partes, la primera consiste en la clase abstracta/interfaz Visitor y las clases que la extienden/implementan. Dichas clases en conjunto cumplen con el patrón Strategy, y por lo tanto tiene las ventajas de este patrón: fácil de extender y las estrategias son fácilmente intercambiables en tiempo de ejecución porque la clase Client las maneja a través de su interfaz común.

La segunda parte consiste en la clase abstracta/interfaz Element y las clases que la extienden/implementas. Estas clases tienen la función de recibir a una instancia de clase que extienda/implemente Visitor (sin saber de qué tipo es) y de utilizar la función correcta la interfaz Visitor de acuerdo a su propio tipo pasando el puntero *this* como argumento, esto último es una técnica llamada Double Dispatch.

En conjunto, el patrón Visitor nos da la posibilidad de que la clase Cliente pueda manejar colecciones mixtas de instancias que extiendan/implementen Element desde su interfaz común, y además, poder aplicar sobre ellos las distintas estrategias encapsuladas en las clases que extienden/implementan Visitor manejándolas también solo a través de su interfaz común.

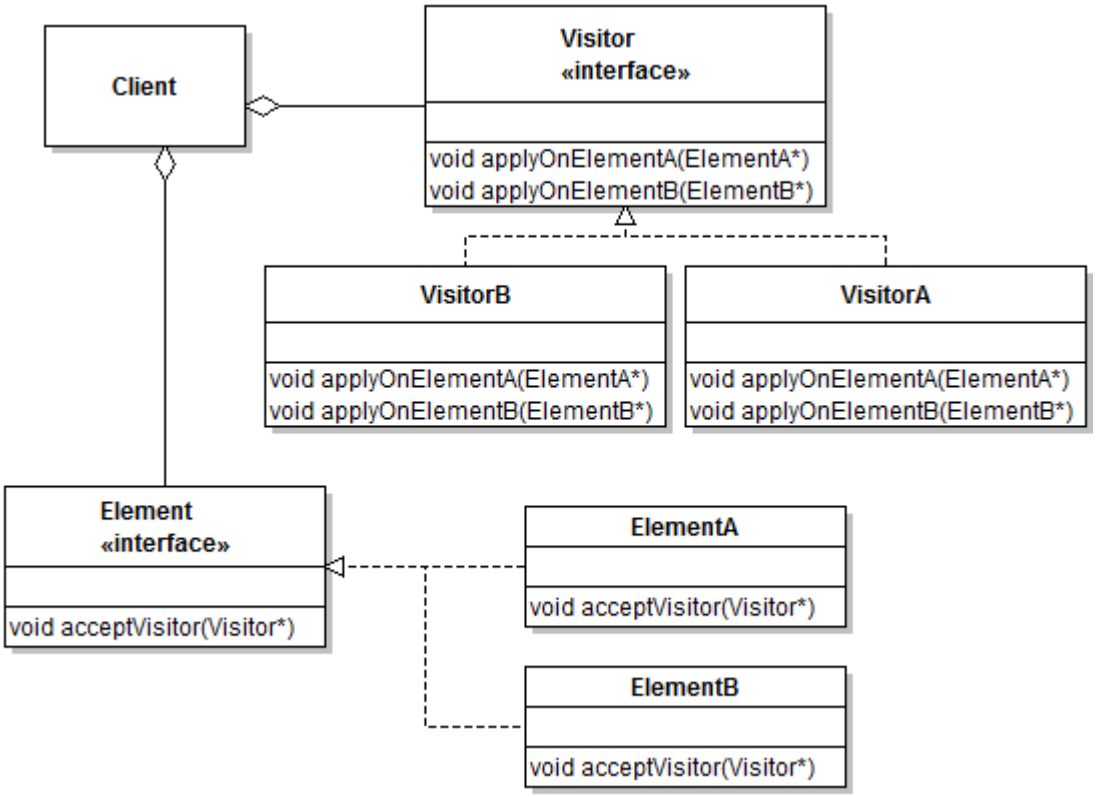


Figura 23 - Patrón de diseño Visitor

3.3. Diseño de la aplicación

El diseño de la aplicación se realizó priorizando el tiempo de procesamiento por sobre el ahorro en memoria RAM. Esto significa que si existe una relación necesaria para realizar alguna tarea, esta relación será agregada al modelo de datos para que esté disponible cada vez que se necesite y no deberá ser calculada cada vez.

Para que la aplicación pudiera manejar mallas mixtas, con poliedros y polígonos de cualquier tipo, se utilizó herencia de clases y polimorfismo. De esta forma, existe una interfaz común para acceder a las propiedades de distintos elementos, pero una implementación distinta (optimizada por tipo de elemento).

Además, la aplicación fue diseñada a modo de framework. Un usuario/programador, puede agregar nuevos elementos de carga de archivo, o renderizado, o evaluación de malla, etc., con facilidad, sin tener que modificar las clases ya existentes de la aplicación. Para extender las funcionalidades de la aplicación, sólo se debe crear una clase nueva, hacer que se registre en el registro apropiado, compilarla y enlazarla correctamente.

Finalmente, el modelo está dividido en varios módulos que interactúan unos con otros. El diseño separado por módulos está descrito en las siguientes secciones.

Para programar la aplicación se escogió el lenguaje de programación C++11. El lenguaje se escogió principalmente por su eficiencia y por permitir programar orientado a objetos. Se escogió la última versión porque esta posee tablas de hashing incluidas en la librería estándar (std::unordered_map) y la mayoría de los compiladores más utilizados la implementan.

Para ahorrar tiempo y no reinventar la rueda, se utilizaron algunas estructuras de datos que vienen en la librería estándar de C++. Se prefirieron las estructuras de la librería estándar porque estas deberían estar en todos los compiladores de C++, con lo que se consigue una aplicación más portable.

Las principales estructuras utilizadas fueron:

- **std::vector**: Es un template de clase, que modela un contenedor para un tipo abstracto (template). Consiste en un arreglo extensible, el cual tiene las funciones de push y pop (de un stack) y cada vez que hace overflow, duplica su tamaño. Además, el tiempo de acceso es constante, al igual que un arreglo común de C, los datos se guardan en memoria contigua y tiene métodos para obtener el tamaño y la capacidad del arreglo en tiempo constante también. La inserción mediante push también es de tiempo amortizado constante.
  En la implementación del visualizador, siempre se prefirió utilizar este tipo de vectores, en vez de arreglos planos de más bajo nivel, a excepción de que se quisiera ahorrar memoria. Los std::vector tienen como miembros dos variables adicionales, dos números para guardar la capacidad del arreglo y la cantidad de elementos que contiene actualmente, pero esto puede ser innecesario si en tiempo de compilación ya conocemos el tamaño que va a tener y más aún si es una estructura que se va a crear millones de veces.
- **std::map**: Es un template de clase que modela un contendor de pares ordenados por una llave. Se utiliza para tener acceso rápido a elementos (log n) dado una llave y poder insertar rápido también (log n). Otra ventaja, es que si se extraen los elementos con un iterador, estos vendrán ordenados por la llave, esto es útil en algunos casos. Normalmente está implementado con un árbol rojo-negro.
- **std::unordered_map**: Es un témplate de clase que consiste en una tabla de hashing. Este template está presente desde el estándar C++11. Los tiempos de acceso e inserción en promedio son más rápidos que en un std::map, pero tiene la desventaja de que los elementos no están guardados en orden. Si uno itera a través de la tabla de hashing, los elementos se obtendrán desordenados.

Una característica común de **std::vector**, **std::map** y **std::unordered_map** es que cuando se insertan elementos en ellos se insertan copias, haciendo imposible insertar referencias. Esta es la razón de por qué, en los diseños que se verán en la siguiente sección, se utilizan punteros al estilo C. Luego, se crean los vértices, polígonos y otros objetos en el stack (no en el heap) para evitar generar múltiples copias de estos elementos y así se podrá mantener múltiples punteros guardados en estas estructuras de datos a instancias únicas.

3.3.1. Modelamiento de los elementos[11] básicos

El diseño de las clases que modelan los elementos base de las mallas es una parte sustancial del programa. Existe un importante trade-off entre utilizar memoria RAM para guardar relaciones de vecindad entre elementos y priorizar la velocidad de cálculos de propiedades, o priorizar el ahorro de memoria RAM y calcular dichas relaciones cada vez que sean necesarias.

Como en la aplicación se decidió darle prioridad a minimizar el uso de la CPU por encima del de memoria RAM, cada vez que una relación de vecindad era útil para algún cálculo, esta se agregaba al modelo de los elementos base y se pre calculaba al cargar las mallas.

Por esta razón, los elementos del modelo (polígonos, vértices, poliedros) almacenan las siguientes relaciones de vecindad:

- Los vértices tienen punteros hacia los polígonos que lo comparten.
- Los polígonos tienes punteros hacia los vértices que lo forman, hacia los polígonos vecinos y hacia los poliedros de los que forma parte.
- Los poliedros tienen punteros hacia los polígonos que describen sus caras.

Además, todos los elementos de la malla tienen la posibilidad de almacenar valores en una tabla. Esta es por si se desea almacenar el cálculo de alguna propiedad y no tener que recalcularla cada vez que la propiedad se requiera.

Finalmente, el diseño incremental de las clases que representan los elementos fue el siguiente:

[11] Para este trabajo, cuando se habla de elementos, se hace referencia a cualquier componente de la malla, tales como vértice, polígono o poliedro.

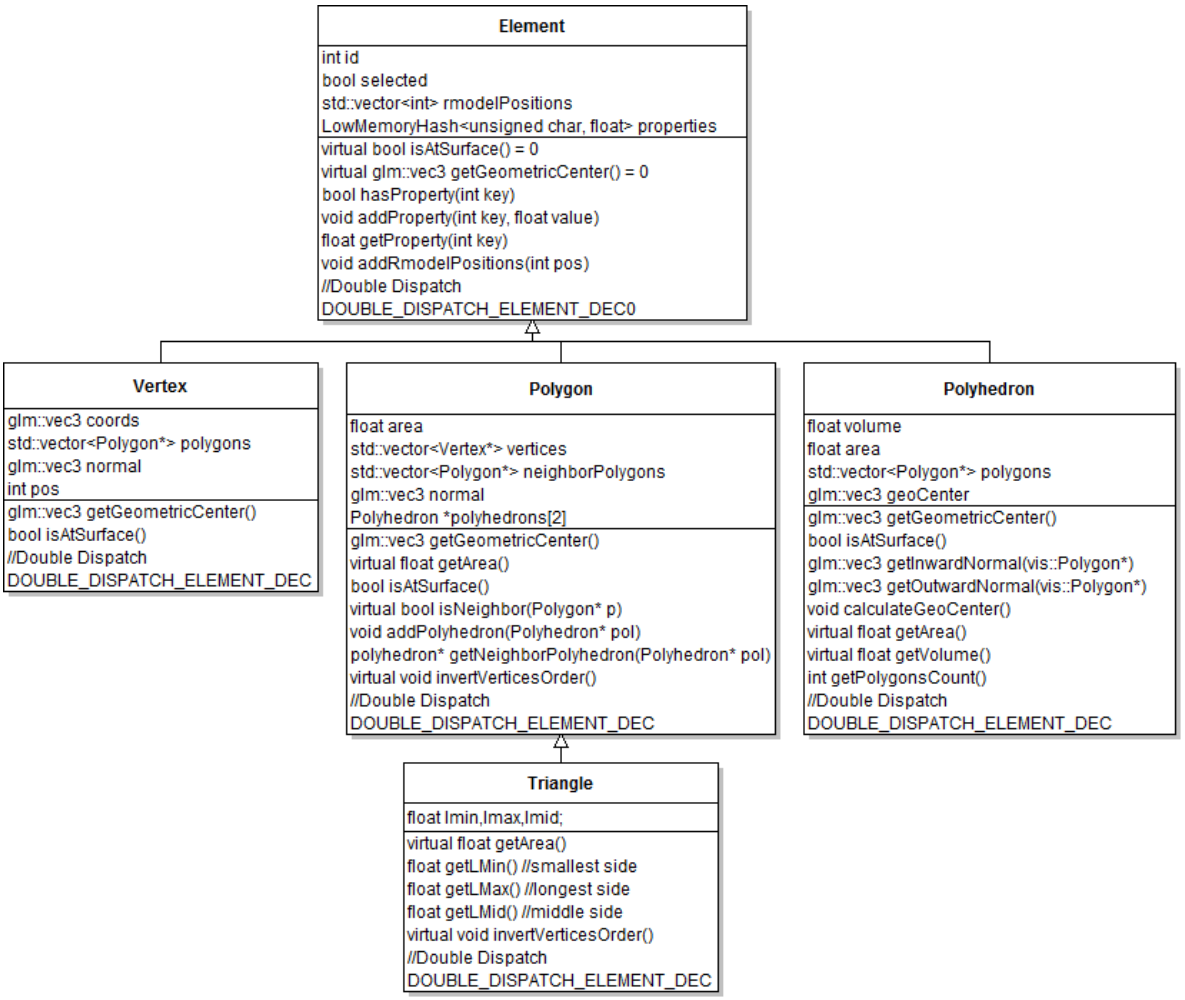


Figura 24 – Diagrama de clases que modelan los elementos de las mallas.

Como se puede ver, existe una clase común a todos los elementos llamada Element. Esta clase tiene las propiedades y funciones básicas que todos los tipos de elementos comparten. Uno de los objetivos principales de esta jerarquía es que en muchas partes, hay segmentos de código que operan sobre un Element y no se necesita copiar código para cada uno de los tipos de elementos.

Las Macros DOUBLE_DISPATCH_ELEMENT_DEC0 y DOUBLE_DISPATCH_ELEMENT_DEC, contienen las declaraciones de todas las funciones que hacen double dispatch y que cada clase que extiende de Element debe tener. También existe la macro DOUBLE_DISPATCH_ELEMENT_DEF, la cual va en la implementación de cada clase, esta declara e implementa las funciones de double dispatch.

```
#define DOUBLE_DISPATCH_ELEMENT_DEC \
virtual bool fullFillsEvaluationStrategy(EvaluationStrategy*);\
virtual float applyEvaluationStrategy(EvaluationStrategy*);\
virtual bool applySelectionStrategyDD(SelectionStrategy*, Selection*);
```

```
#define DOUBLE_DISPATCH_ELEMENT_DEF(x) \
bool x::fullFillsEvaluationStrategy(EvaluationStrategy* e){\
        return e->isFullFilled(this);\
}\
float x::evaluateUsingEvaluationStrategy(EvaluationStrategy* e){\
        return e->value(this);\
}\
bool x::applySelectionStrategyDD(SelectionStrategy* stra, \
                                 Selection* sel){\
        return (stra->isFullFilled(this))? \
                stra->selectElement(this,sel):false;\
}
```

Con el diseño anterior las estructuras de dato tienen un peso base de (en bytes):

Tamaño de Element:	64
Tamaño de Vertex:	128
Tamaño de Polygon:	152
Tamaño de Triangle:	168
Tamaño de Polyhedron:	112

La cantidad de espacio que se utilizan en la práctica es mayor, ya que la estructura de datos std::vector y LowMemoryHash crecen dinámicamente.

Considerando que el visualizador está pensado para abrir mallas de gran tamaño, por sobre 1.000.000 de polígonos, hay que intentar minimizar el uso de memoria de estas clases al máximo.

Para minimizar el uso de memoria de estas clases, se tomaron las siguientes medidas:

- La cantidad de atributos y relaciones de vecindad que tiene cada una se redujo al mínimo necesario para calcular las propiedades de los elementos rápidamente.
- Para la tabla de hash de propiedades no se usó la estructura std::unordered_map que viene en la librería estándar de C++11, ya que esta estructura tiene un peso base de 64 bytes y se desconoce cómo crece su tamaño en memoria a medida que se agregan elementos. Se creó una nueva estructura llamada LowMemoryHash, la cual implementa una tabla de hash minimizando el uso de memoria con respecto a std::unordered_map. La estructura LowMemoryHash tiene un peso base de 24 bytes.

3.3.2. Diseño de Modelos

El diseño del modelo debía ser capaz de soportar mallas (estructuradas, no estructuradas, mixtas) de polígonos, mallas de poliedros y nubes de vértices.

Se propuso dos diseños, el primero consistía en tener una jerarquía de clases de la siguiente forma:

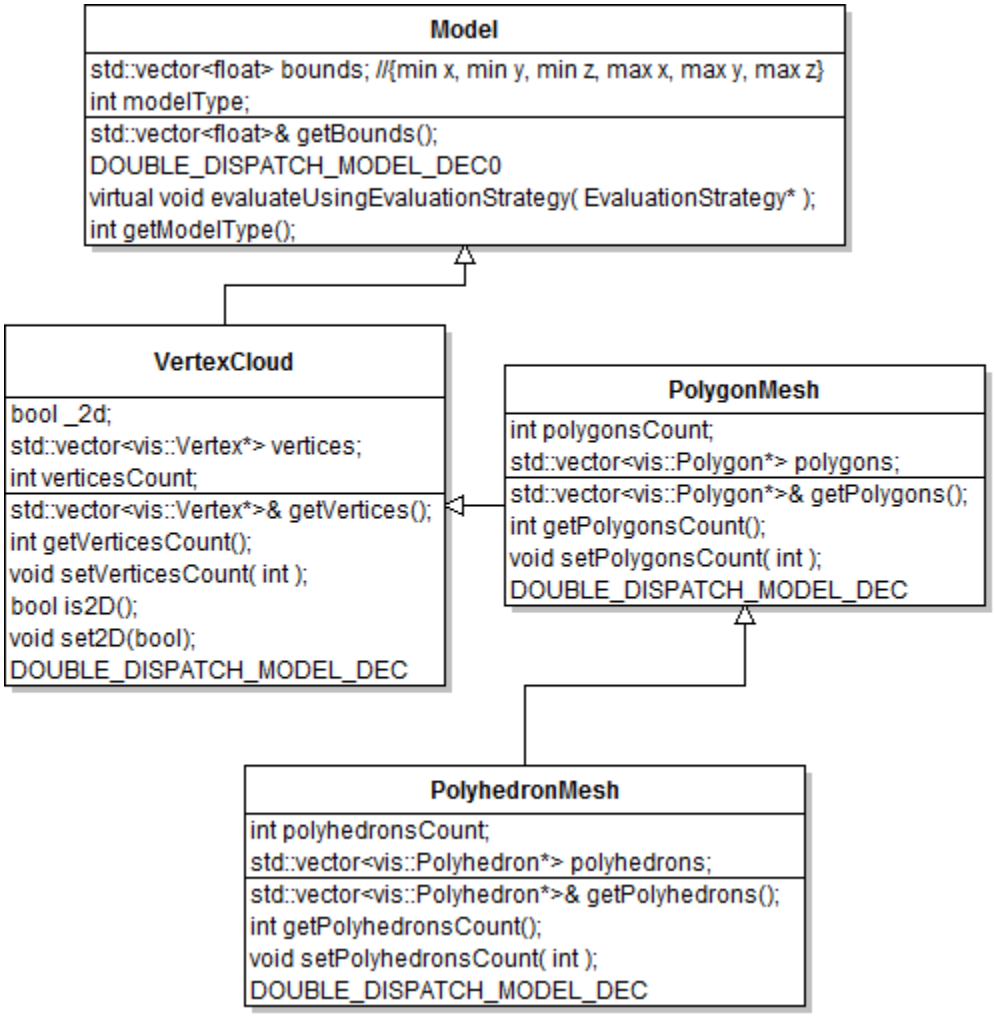


Figura 25 – Diseño del modelo.

La segunda forma consistía en tener una única clase, que almacene todas las propiedades de las clases que se ven en el diseño.

Con el primer diseño, el controlador principal de la aplicación manejaría el modelo a través de la clase base Model. Luego, cada vez que el modelo se utilizara se haría a través de un double dispatch, a menos que los datos en Model sean suficiente. Esto agrega un poco de complejidad al manejo del modelo, pero facilita el trabajo en la extensión de módulos a través de las clases base (Módulo de estrategias de evaluación, selección, cargado de

mallas, etc…), ya que no hay que replicar ninguna validación para identificar el tipo de modelo con que se está manejando.

En cambio, con el segundo diseño, no existiría el nivel de información que provee el double dispatch, y el tipo de malla tendría que chequearse preguntando por algunas condiciones.

Las Macros DOUBLE_DISPATCH_MODEL_DEC0 y DOUBLE_DISPATCH_MODEL_DEC, contienen las declaraciones de todas las funciones que hacen double dispatch.

3.3.3. Módulo de carga de Modelos

El módulo de carga consiste en una clase base llamada *ModelLoadingStrategy* y clases que la extienden. Cada una de las clases que extienden a *ModelLoadingStrategy* representa la carga de algún formato específico, y deben implementar el código necesario para esto.

La clase *ModelLoadingStrategy* posee dos funciones que deben ser implementadas (*load* y *validate*) por la clase que la extiende. La función *validate(..)* debería devolver true o false dependiendo de si el archivo es válido para la estrategia de cargado seleccionada. Si *validate(..)* retorna true, se procede a llamar a la función *load* para que construya el modelo. Si la función *load* falla, esta debe retornar un puntero de tipo *Model** con valor 0.

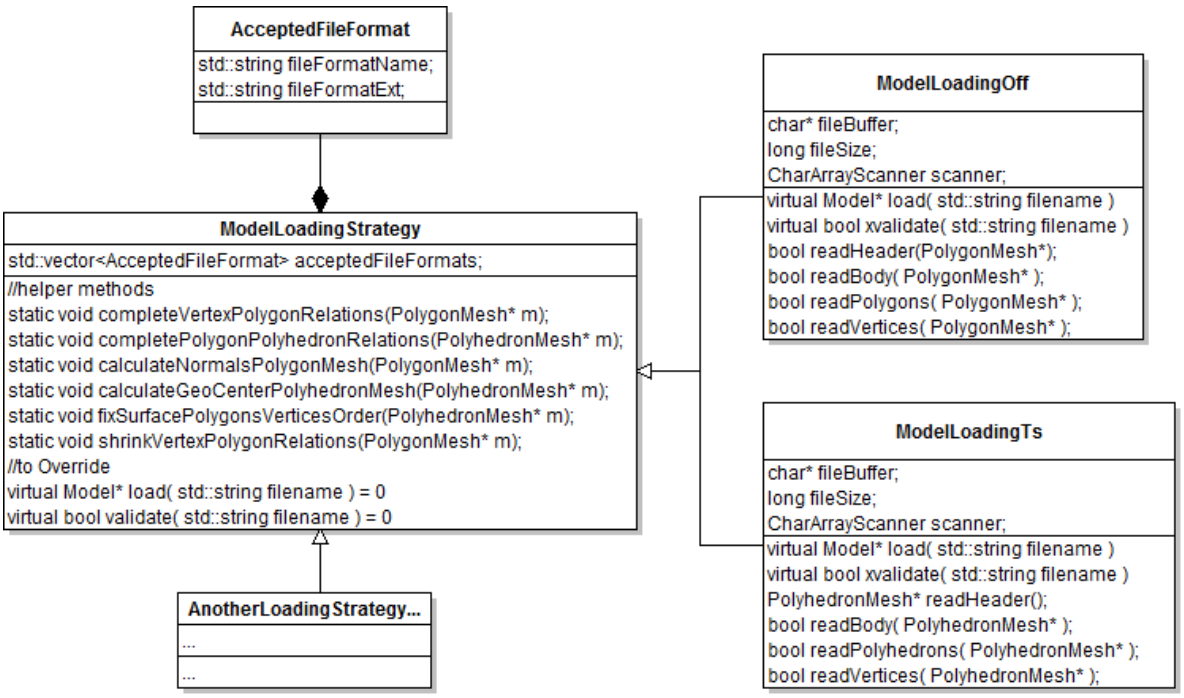


Figura 26 – Diagrama de clases del módulo de carga de mallas.

3.3.4. Módulo de renderizado

Para poder renderizar un modelo, utilizando Shader de forma eficiente, primero se debe guardar en la memoria RAM de la tarjeta de video todos los datos que representan los atributos asociados a cada uno de los vértices que se van a utilizar en el renderizado.

Como en este caso, muchos de los atributos asociados a cada vértice son constantes en el tiempo, o varían muy poco, es conveniente enviarlos a la memoria de la tarjeta de video y mantenerlos ahí para ser utilizados.

Existen dos formas principales de hacer el dibujado eficiente, la primera es que aunque los vértices sean compartidos por distintas primitivas (triángulos, líneas, etc), estos se envían repetidos a la tarjeta de video, uno por cada primitiva que lo necesite. La segunda forma, consiste en enviar la cantidad de vértices que existen en el modelo original, y las primitivas obtienen sus vértices desde este conjunto usando un arreglo de índices que se le provee.

Con la segunda forma existe un ahorro en la cantidad de vértices que debe procesar la GPU, pero tiene un problema que la hace imposible de utilizar para esta aplicación. En OpenGL, los atributos están asociados a los vértices, por lo que un vértice compartido tendrá los mismos atributos en todos los polígonos que lo comparten. Esto imposibilita, por ejemplo, que existan dos polígonos que compartan un vértice y que cada uno de estos polígonos tenga un color plano distinto. Para que un vértice tenga un color para un polígono y otro color para otro que lo comparte, es necesario repetirlo. Es por esta última razón, que esta alternativa no es viable para el proyecto.

Debido a esto, para enviar los datos correctamente a la tarjeta de video, se creó una clase llamada RModel, la cual está encargada de procesar los distintos tipos de modelo, extraer los atributos correspondientes a los vértices y enviarlos a la memoria de la tarjeta de video. Para las mallas de polígonos o poliedros, toma todos los polígonos y los triangula, luego, en arreglos separados para cada atributo, envía cada atributo de cada vértice que forma estos triángulos, de tal forma que los primeros 3 atributos corresponden a los vértices del primer triángulo, los siguientes 3 corresponden a los del segundo triángulo y así sucesivamente.

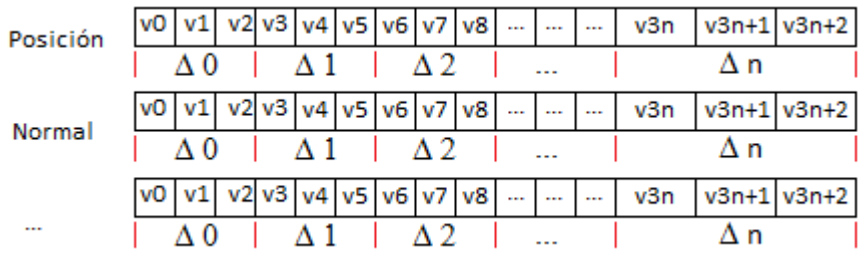


Figura 27 – Arreglos de atributos de vértices: Aquí se muestra un ejemplo de cómo se ordenan los atributos en distintos arreglos.

Los atributos que se suben por vértice son:

- Id del vértice
- Posición del vértice
- Normal del vértice
- Id Polígono/Poliedro
- Unsigned int con 32 Flags
- Posición que le corresponde al vértice en los arreglos de atributos.

El atributo de Flags, corresponde a un número entero sin signo, en el cual cada bit significa algo. Este unsigned int de flags esta encapsulado en la clase RVertexFlagAttribute.

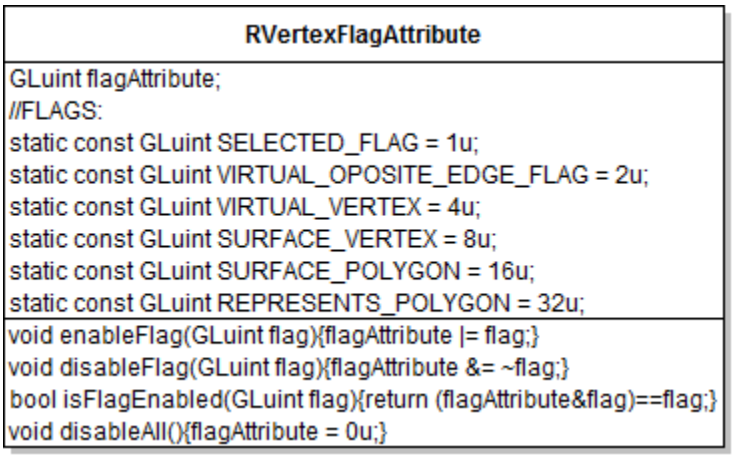


Figura 28 – Diagrama de clase de RVertexFlagAttribute.

Luego, estos flags pueden ser utilizados en los shaders siempre que el atributo se habilite. Este atributo representa un caso especial, es el único que queda guardado en la RAM principal del computador, ya que requiere actualizarse bajo algunas circunstancias y volver a enviarse a la tarjeta de video. Los demás atributos apenas son enviados a la tarjeta de video, se liberan de la memoria RAM principal

Finalmente, teniendo los atributos correspondientes a los vértices guardados en la tarjeta de video, se diseñó una clase base llamada Renderer, la cual tiene una serie de funciones base que deben ser re implementadas.

Una clase que extiende de Renderer es una clase que tiene que contener la lógica necesaria para poder renderizar el modelo utilizando los arreglos de atributos que están en la tarjeta de video. La clase Renderer tiene métodos reimplementables que le permiten a la clase que extiende:

- Tener un QWidget para poder definir una configuración.

- Renderizar un modelo a partir de los datos procesador por RModel o a partir del model base.

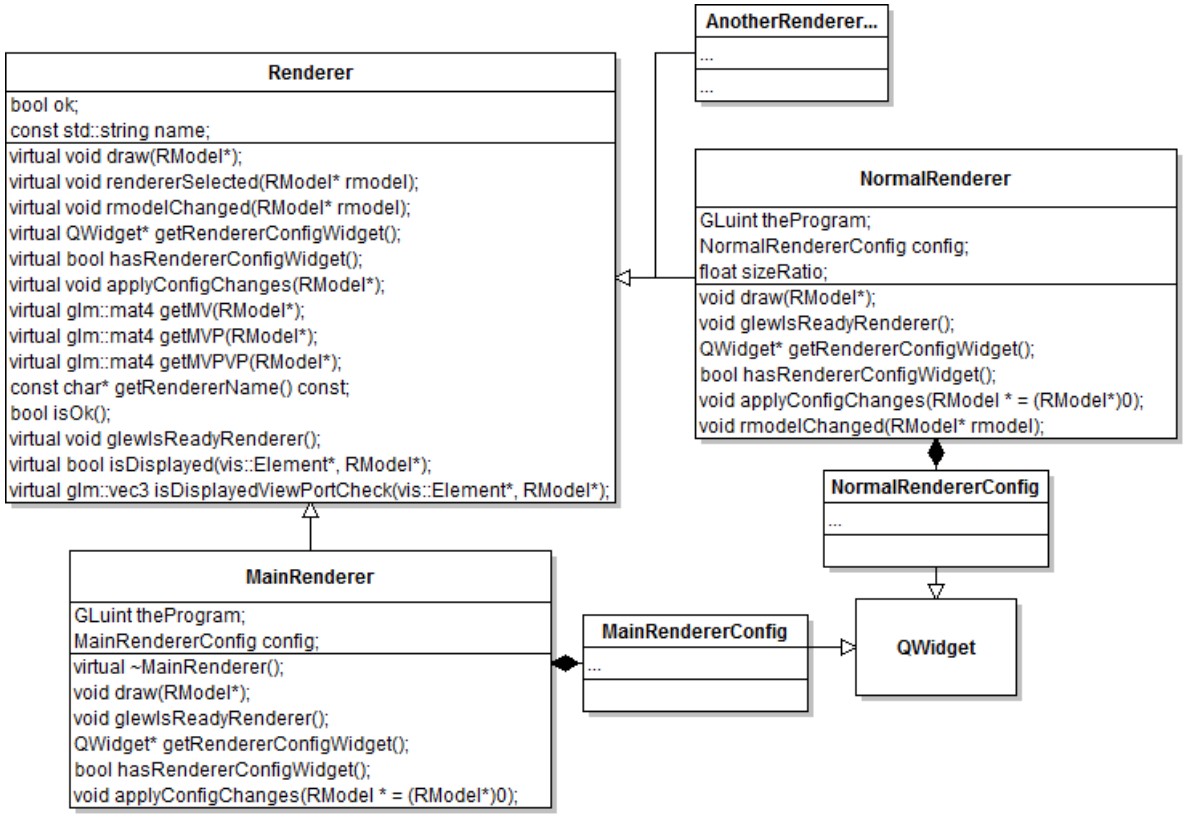


Figura 29 - Diagrama de clases del módulo de renderers.

3.3.5. Módulo de evaluación de elementos

La clase principal de este módulo es EvaluationStrategy. Las estrategias de evaluación sólo pueden ser aplicadas sobre los elementos individuales que conforman los modelos.

Las estrategias de evaluación que extienden EvaluationStrategy pueden ser escogidas por el usuario utilizando la GUI para evaluar un modelo, para obtener estadísticas, para pintar el modelo según los valores obtenidos y para seleccionar elementos del modelo.

Cuando se aplica una estrategia primero se chequea si es aplicable sobre el elemento o no, utilizando la función isFullFilled. Si es aplicable, se utiliza la función value sobre el elemento. También, se pueden obtener la cantidad de elementos cuya evaluación resultó entre un rango de valores mediante la función getValuesCountInRange.

La clase que extiende EvaluationStrategy, tiene la opción de definir si esta estrategia genera estadísticas o no. También, debe sobrescribir una de las funciones isFullFilled y una de las

value. El diseño no contempla que una misma estrategia pueda evaluar dos elementos de distinto tipo. Por ejemplo, si la métrica de evaluación es válida para polígonos y poliedros, se debe hacer una clase distinta para cada una.

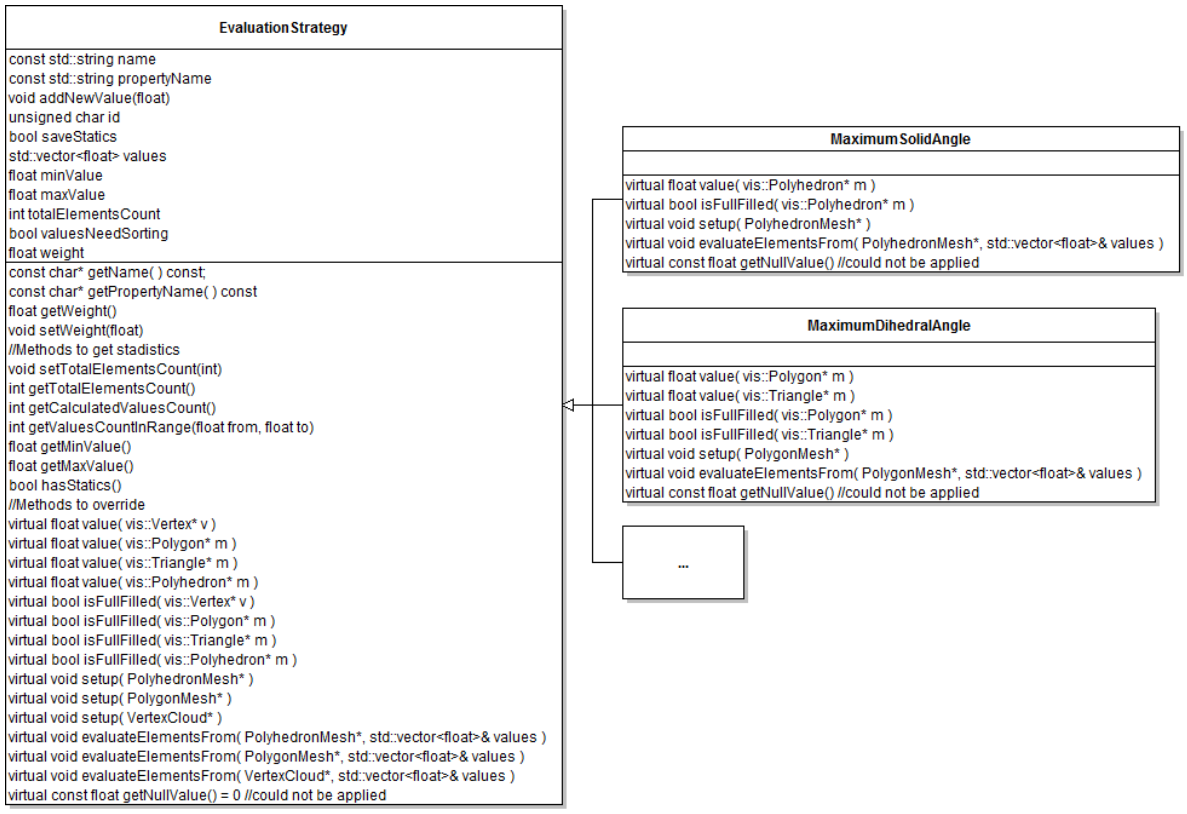


Figura 30 - Diagrama de clases del módulo de estrategias de evaluación.

3.3.6. Módulo de selección de elementos

La clase principal de este módulo es SelectionStrategy. Las estrategias de selección pueden ser aplicadas sobre los elementos, modelos completos o sobre el conjunto de elementos que ya está seleccionado. Estas poseen la posibilidad de tener un Widget de configuración.

Las estrategias de selección que extienden SelectionStrategy pueden ser escogidas por el usuario utilizando la GUI. El controlador hace uso de ellas a través de su clase base.

Cuando se aplica una estrategia, se diseñó el siguiente flujo:

a. Se llama a la función sin argumentos llamada Setup. Esta debe realizar cualquier trabajo de configuración que sea necesario para los siguientes pasos. Por ejemplo, leer la configuración que ha escogido el usuario en el Widget de configuración de la estrategia.
b. Se revisa si la estrategia se debe aplicar sobre el conjunto de elementos seleccionados con la función isAppliedOnSelection. Esta función no está

hecha para re-definirla, sino que en la estrategia se debe cambiar el valor del boolean que retorna esta función.

c. Si debe ser aplicada sobre la selección:
    a. Se aplica la función setupPreApplying, la cual toma como argumentos el RModel y los elementos seleccionados. Esta función está hecha para hacer cualquier pre procesamiento antes de aplicar la estrategia. Por ejemplo, limpiar el conjunto de selección. Si la función retorna false, el flujo termina sin aplicar la estrategia.
    b. Se aplica la función principal selectElement de la estrategia para seleccionar elemtos y el flujo termina.
d. Si debe ser aplicada sobre el modelo:
    a. Se llama a la función *selectElementsFrom* utilizando un double dispatch para escoger la correcta según el modelo.
    b. *selectElementsFrom* en su implementación base, se encarga de aplicar la estrategia sobre los elementos (vértices, polígonos o poliedros) correctos.

Cuando para aplicar la estrategia sobre los elementos básicos de una malla, primero se utiliza la función isFullFilled para saber si se puede aplicar, luego se utiliza la función selectElement correspondiente.

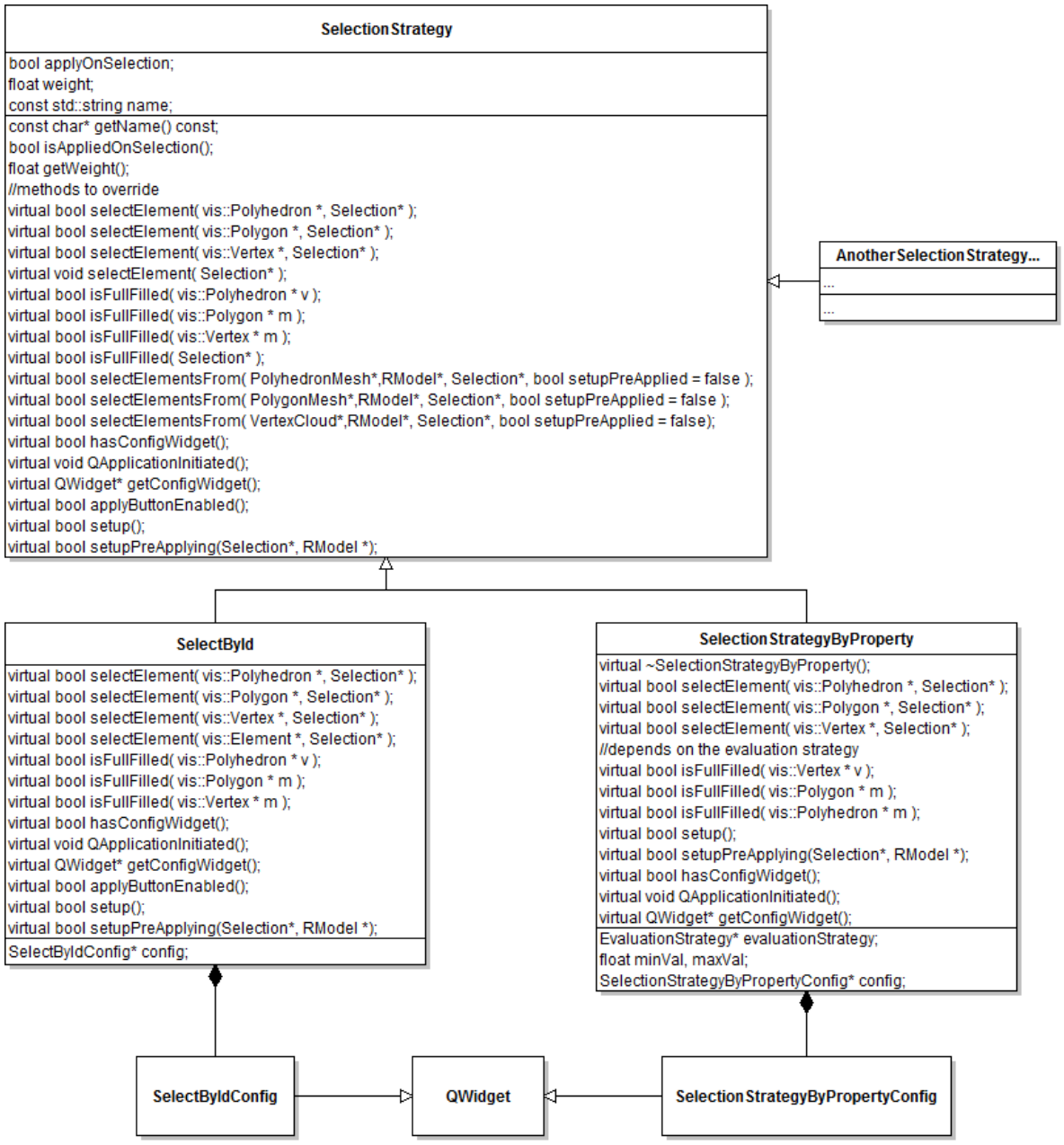


Figura 31 - Diagrama de clases del módulo de estrategias de selección.

3.3.7. Registros

Para que la aplicación pueda hacer uso de las extensiones que se van agregando, sin tener que modificar la GUI u otra sección, existen distintos registros dinámicos. Estos registros están implementados para que exista una sola instancia de cada uno, utilizando el patrón de diseño *Singleton.*

Cada módulo extensible debe tener su registro asociado. Luego, existe un registro de Renderers, otro de ModelLoadingStrategy, otro de EvaluationStrategy y así sucesivamente. Todos estos registros extienden de clase template base llamada RegistryTemplate.

RegistryTemplate es un template de registros. Guarda punteros a distintas instancias de clases indexados por una llave a elección. También tiene una tabla de prioridades versus llaves, la cual sirve para ordenar los distintos elementos de la forma que el programador quiera que aparezcan en los combobox de la GUI. Por ejemplo, se agrega una estrategia de evaluación y se quiere que aparezca primero en la lista en la GUI.

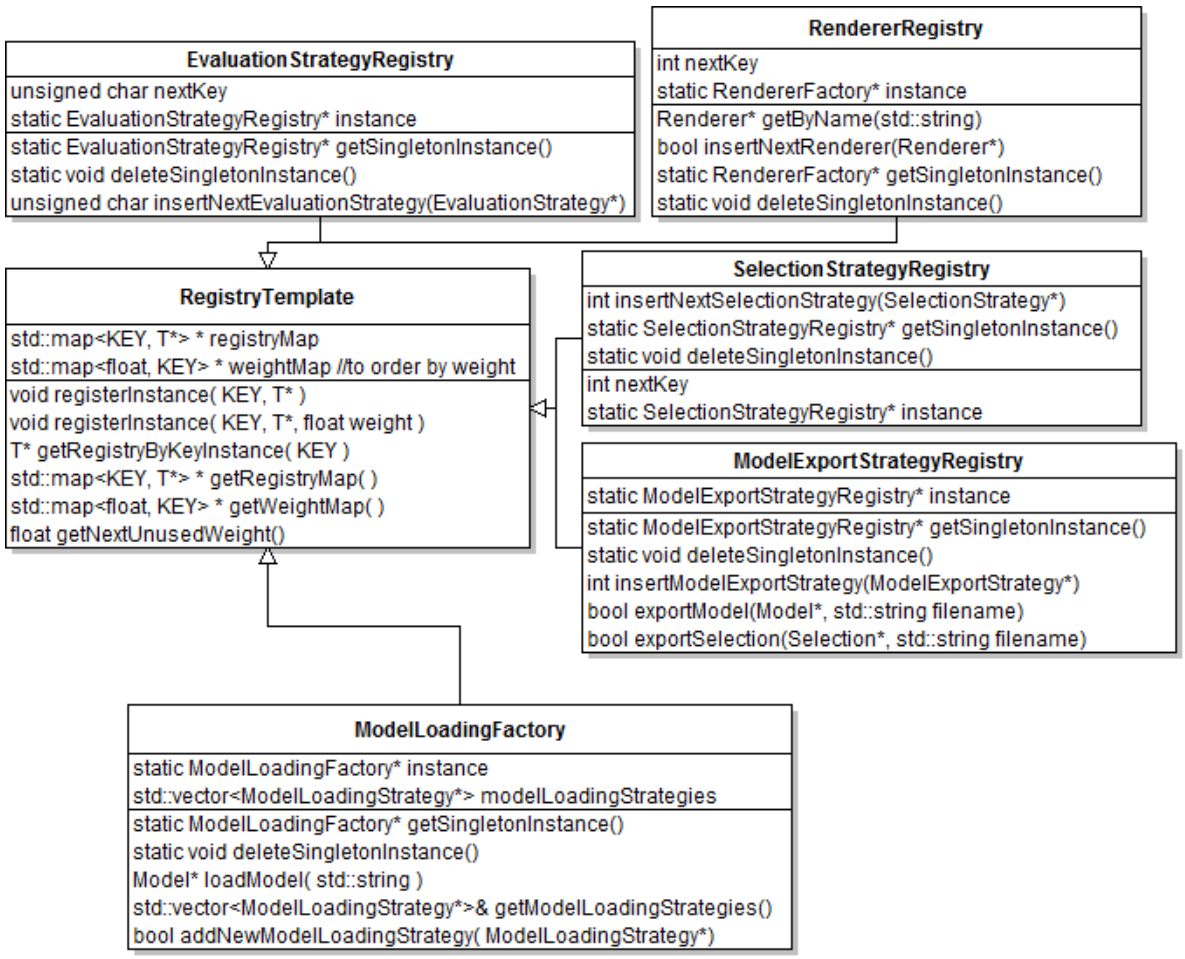


Figura 32 - Diagrama de clases del módulo de registros.

3.3.8. Controlador

Se diseñó una clase que actúa como controlador entre los distintos componentes de la aplicación. Esta clase, llamada *Camarón*, extiende la clase QMainWindow. La clase posee punteros hacia todos los registros y es la encargada mediante los eventos enviados por el usuario a través de la GUI, realizar todas las operaciones necesarias entre componentes.

Camarón es la encargada de inicializar GLEW cuando el contexto de OpenGL se haya creado y es la encargada también de notificar a todas las clases que lo necesiten, que GLEW está listo.

El contexto de OpenGL es creado cuando el componente CustomGLViewer es instanciada. CustomGLViewer es una clase que extiende a QGLWidget. Este widget se ubica en el centro de la ventana del visualizador, y es donde se despliegan las imágenes producto del renderizado del modelo cargado.

QGLWidget es un widget para renderizar gráficos con OpenGL. Provee la funcionalidad necesaria para desplegar grafico OpenGL integrados con una aplicación Qt. Es muy simple de usar. Simplemente se puede extender y utilizar como cualquier QWidget, con la excepción de que se puede escoger entre utilizar el tradicional QPainter para dibujar en Qt o utilizar comandos estándar de OpenGL.

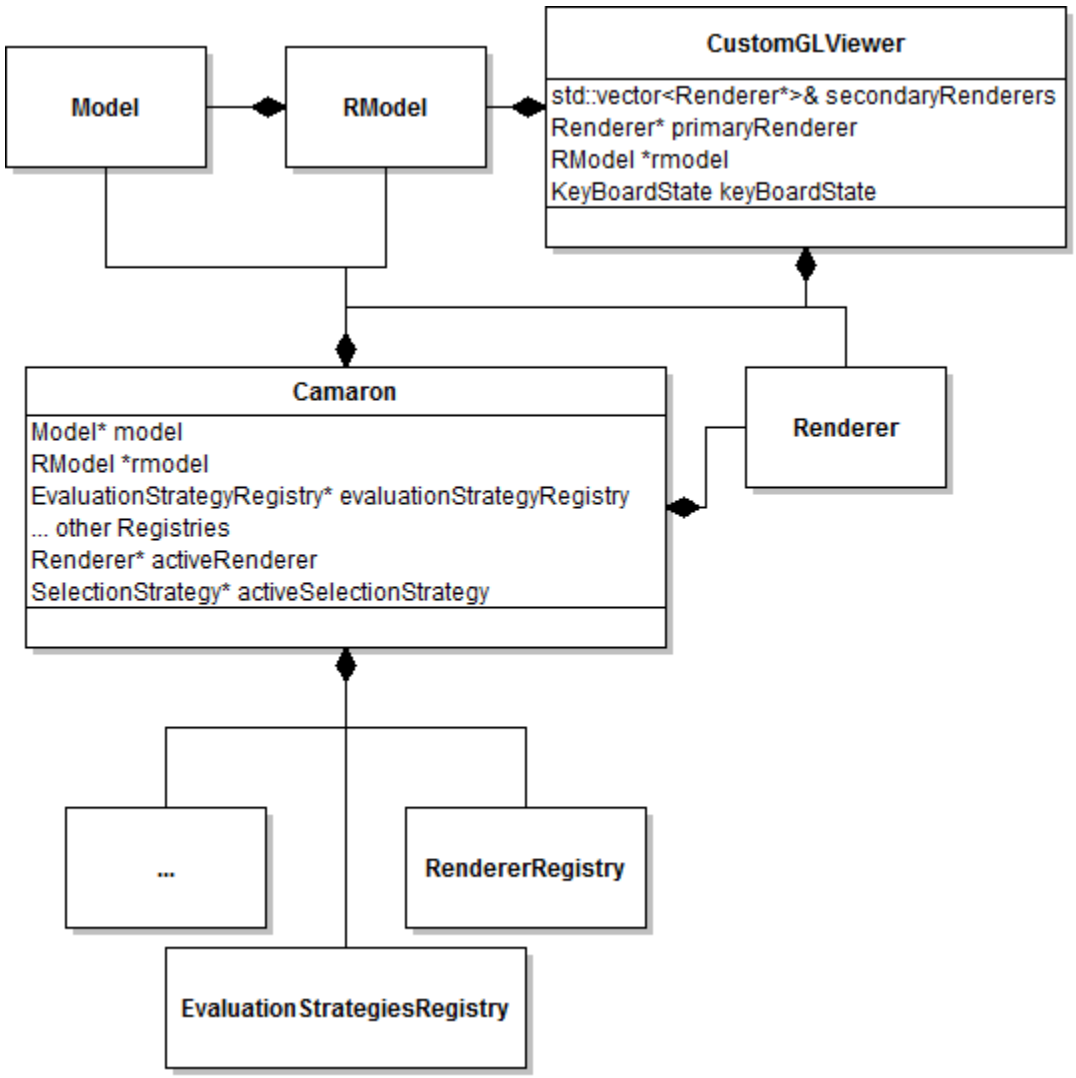


Figura 33 - Diagrama de clases del controlador principal.

4. Implementación

En la próxima sección se describen los puntos de la implementación que se consideraron más importantes e interesantes de exponer. Vale mencionar que el desarrollo de la aplicación comenzó con el trabajo de título, no es la extensión de una aplicación anterior.

En algunos algoritmos se utilizan clases como *ShaderUtils* u *OpenGLUtils* y estructuras de datos intermediarias como *ShaderLoadingData*, las descripciones de estas estructuras y clases se pueden encontrar en el Anexo E.

Para simplificar la implementación del visualizador, se decidió que sólo se aseguraría un buen funcionamiento para mallas con elementos convexos[12]. El visualizador es capaz de cargar polígonos y poliedros cóncavos, pero es muy probable que existan errores en el renderizado y en algunas operaciones que se realizan sobre los elementos.

Trabajar sólo con elementos convexos es una limitación pequeña, ya que un elemento cóncavo puede ser representado mediante elementos convexos, sólo habría que pre procesar la malla. Además, son pocas las mallas que utilizan elementos cóncavos porque estos no otorgan ninguna ventaja (en este contexto) y son más complicados de procesar. Más aún, es muy común que las mallas sean solamente de triángulos y/o tetraedros.

4.1. Ambiente de desarrollo

Toda la aplicación se desarrolló utilizando Windows 7, pero ésta no es dependiente de Windows. Se tuvo especial cuidado en que todos los componentes externos que se utilizaran fueran multiplataforma.

Como IDE se utilizó QtCreator, ya que es una aplicación liviana, gratuita, ofrece opciones de auto completado del código, tiene la posibilidad de agregar un debuguer y compilador externo. Además, QtCreator genera automáticamente el makefile para compilar la aplicación dependiendo del sistema operativo en que se encuentre. QtCreator es multiplataforma.

Se utilizó el SDK de Qt que viene con mingw versión 4.4 como compilador, además de un debugger, QtCreator 2.4.1 y Qt 4.7.4. El único incoveniente era que la versión de Qt es de

[12] La malla como un todo, sí puede modelar un objeto cóncavo.

32 bits. Para que la aplicación pudiera utilizar una cantidad mayor de memoria, se debía compilar en 64 bits.

Para poder compilar la aplicación en 64bits, fue necesario recompilar la librería Qt en 64 bits, utilizando el compilador mingw de 64 bits.

4.1.1. OpenGL, Windows, Glew

Microsoft hace muchos años que suspendió el soporte a OpenGL en Windows, por lo que la versión que Windows dispone para el programador es la 1.1 (versión muy antigua). Para poder tener acceso a las funcionalidades de versiones más nuevas de OpenGL, debemos tener una tarjeta de video adecuada, con los drivers correspondientes y obtener estas funciones mediante punteros dinámicos.

Si escribimos el código para obtener estas funciones en Windows, probablemente vamos a ensuciar mucho el código con líneas que son solamente válidas si se está trabajando en Windows.

Para solucionar el problema anterior existe la librería GLEW (The OpenGL Extension Wrangler Library), la cual es multiplataforma y provee mecanismos eficientes para obtener las funcionalidades de OpenGL que estén disponibles a través del driver de la tarjeta de video. Esta librería esconde todo el código extra que habría que utilizar en Windows para poder acceder a estas funcionalidades. Sólo hay que importar el header de GLEW antes que el de OpenGL, y podremos utilizar las funciones de OpenGL más nuevas que estén disponibles de forma transparente. La única condición es que hay que llamar a la función *glewInit*() apenas se crea el contexto de OpenGL de la aplicación. [32]

4.2. Registro de componentes agregados

Los registros de estrategias se implementaron de forma que al agregar un nuevo componente, este no tuviera que registrarse agregando líneas de código en la clase del registro (EvaluationStrategyRegistry, RendererRegistry, etc.) correspondiente. Para conseguir lo anterior se definió una macro en el header de cada registro, y utilizando dicha macro se puede registrar un componente en su mismo archivo de implementación, sin tener que modificar la clase del registro respectivo. El objetivo principal, es que para agregar un nuevo componente, se deba modificar lo menos posible las demás clases de la aplicación, quedando la responsabilidad de llenar los registros en los componentes y no en los registros.

Además, fue necesario implementar todos los registros utilizando el patrón de diseño Singleton. Con este patrón, podemos acceder desde las distintas clases que se van a registrar, a la misma instancia de registro. El registro se realiza mediante una llamada una macro, por ejemplo, para el caso del registro de Renderers, existe la siguiente macro en el header del RendererRegistry:

```
#define REGISTER_RENDERER(x) \
        bool x##_dummy_var_render  = \
        RendererRegistry::getSingletonInstance()->insertRenderer(new x())
```

Con esta macro se define una variable global, la cual en su nombre tiene concatenado el nombre del renderer que se está registrando, lo que impide que haya colisiones de nombres de variables de registro, además, la variable es inicializada, pero el valor no se utiliza. Para inicializarla, se obtiene la única instancia[13] que existe de RendererRegistry y luego se inserta el Renderer[14]. Para el caso del MainRenderer, la macro se utiliza con la siguiente línea:

```
REGISTER_RENDERER(MainRenderer);
```

Registrar los componentes de esta forma, trae dos problemas que son descritos en las siguientes dos secciones.

4.2.1. Inicialización de componentes gráficos (GLEW, QApplication, OpenGL Context)

Las funciones de registrarse, al ser llamadas para inicializar variables globales, se ejecutan antes de ejecutar el código de la función *main()*, por lo que se están instanciando los componentes antes de que GLEW o la aplicación de Qt se inicialice.

Si un componente, ya sea un Renderer o SelectionStrategy, tienen un QWidget de configuración, este widget no puede ser inicializado antes de que se inicialice la aplicación de Qt. Además, para inicializar los Shaders, se utilizan funciones de OpenGL, las cuales son obtenidas a través de GLEW, y por lo tanto, no pueden utilizarse antes de llamar a la función *glewInit()*.

Cuando se está utilizando la librería GLEW, es necesario antes de cualquier llamada a una función de OpenGL, llamar a la función *glewInit()*. Se podría pensar que la solución más simple es agregar un fragmento de código que inicialice GLEW con una variable global

[13] Utilizando el patrón Singleton.
[14] Se utiliza una variable global que no se utiliza más tarde por que el estándar de C++ no permite llamar a una función en este contexto sin que este inicializando una variable.

(arreglando de alguna forma para que esto ocurra antes de la creación de los Renderer). El problema es que GLEW necesita que el contexto de OpenGL haya sido creado.

El contexto de OpenGL recién es creado cuando se instancia la clase CustomGLViewer. CustomGLViewer se instancia en el constructor de la aplicación principal.

Para solucionar el problema del instanciado de los QWidget y la inicialización de Shaders prematura, la inicialización de estos se quitó del constructor de los componentes y se agregó una función a cada interfaz, la cual es llamada después de instanciar la aplicación de Qt, crear el contexto de OpenGL y llamar a glewInit(). Los componentes que utilizan un QWidget de configuración y/o programas de Shaders, deben sobrescribir esta nueva función de la interfaz e inicializar dichos elementos en ella.

4.2.2. Orden de inicialización de variables estáticas y globales

Ni el estándar de C++, ni sus siguientes versiones, garantizan algún orden de inicialización entre variables que se encuentren en distintas unidades de compilación. Por otro lado, los registros: EvaluationStrategyRegistry, RendererRegistry, SelectionStrategyRegistry, guardan los componentes en su tabla en el orden en que fueron agregados.

Dadas estas condiciones, el orden de los registros quedaba fuera del manejo del programador y aparecían en un orden aleatorio en la GUI del visualizador, mientras que probablemente se habría preferido un orden por tipo o por nombre.

Para solucionar este problema, al RegistryTemplate se le agregó una tabla de prioridades versus llave. Esta tabla adicional, sirve como índice para ordenar los registros a conveniencia del programador. Las prioridades son números de punto flotante que el programador puede definir sobre la clase que va a registrar, luego, los elementos son extraídos de la tabla utilizando el orden que le dan dichas prioridades.

4.3. Manejo de transparencia en OpenGL

El visualizador desarrollado tiene opciones para escoger los colores con que se dibujan los elementos. En la mayoría de los casos, se pueden escoger las 4 componentes RGBA del color, por lo que se puede hacer que los elementos se dibujen translúcidos.

Como se revisó en la sección de Antecedentes, en la sección **2.3.4. Manejo de Transparencia en OpenGL**, poder renderizar una escena con elementos translúcidos no es trivial.

En el visualizador desarrollado, se envían los atributos de cada vértice (incluida la posición) en un orden fijo, luego, cada vez que se genera una imagen, las primitivas son procesadas en un mismo orden. Es por esto, que sin un tratamiento especial de la transparencia, obtenemos la siguiente falla en la visualización:

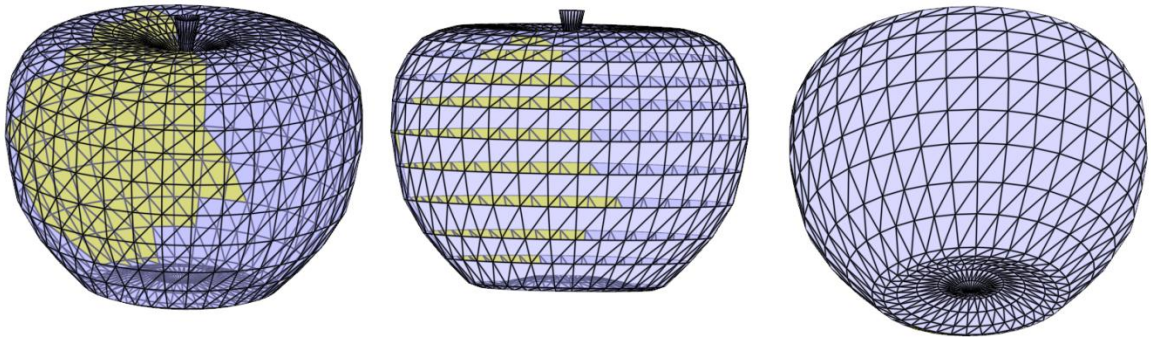


Figura 34 - Fallo en blending sin tratamiento especial.

En ciertos ángulos, las primitivas se dibujan en el orden correcto, pero variando el ángulo, se llega a un punto en las primitivas no se dibujan en el orden que deberían y las de más adelante bloquean a las de atrás, aun siendo translucidas (alfa menor a 1).

Para abordar el problema, tomando en cuenta que la aplicación desarrollada dispone de distintos *Renderer*, cada uno con su programa de Shaders distinto, podemos separar las técnicas de manejo de transparencia en dos conjuntos:

- Técnicas que deben ser implementadas en cada Renderer, ya que necesitan que el shader asociado a este realice alguna tarea adicional, agregándole complejidad. En este conjunto está Depth Peeling y A-Buffer, por ejemplo en el primero, el Fragment Shader debe descartar fragmentos de acuerdo a la profundidad de estos. Los *Renderer* que implementen alguna de estas técnicas, tienen un flag para poder activarla/desactivarla.
- Técnicas que no requieren que los Shaders realicen alguna tarea adicional. Weighted Sum, Weighted Average y doble pasada con test de alfa, pueden ser implementadas sin modificar los Shaders. Son transversales a los *Renderer*. Estas técnicas puede activarse en la configuración global de OpenGL de la aplicación.

Para la aplicación, se implementaron 4 técnicas:

- Depth Peeling
- Doble pasada con Test de Alfa
- Weighted Average
- Weighted Sum

A-Buffer se descartó por la gran cantidad de memoria que utiliza y la aplicación sin esta técnica ya está ocupando mucha memoria.

En las siguientes sub-secciones, se explica cómo se implementaron las distintas técnicas, sus pros y contras.

4.3.1. Implementación Depth Peeling

Esta técnica es implementada en el código de los *Renderer*. A los *Renderer* que cuentan con esta técnica, se les agrego en la configuración un checkbox para habilitarla.

El orden de procesamiento de las capas se escogió de la más cercana a la más lejana porque si se escogen menos capas de las necesarias, el efecto de que falten capas lejanas es menos notorio a que si faltaran de las más cercanas.

El usuario debe escoger el número de pasadas. Este puede ser inferior al número que realmente se requería para la complejidad de la escena. Escoger un número menor puede ser suficiente para que la escena se vea bien y puede ahorra tiempo de computo. Escoger un número mayor que el necesario, no sirve. Los resultados según el número de capas escogido dependen del modelo.

En la implementación se utilizó un Frame Buffer Object (FBO). Los FBO se utilizan para generar imágenes en un cuadro alternativo o en una textura, no en el cuadro principal de la aplicación. Para realizar lo anterior, los FBO tienen espacios para que se les asocie un buffer de profundidad, uno de stencil y una serie de buffers para guardar imágenes de colores (texturas o Render Buffers). Luego, los FBO nos dan la posibilidad de dibujar distintas imágenes al mismo tiempo sobre sus puntos de anclajes en una misma pasada, sin modificar los buffers principales.

Para el presente informe, cuando una textura es asociada como buffer de profundidad, se da por entendido que esta se enlaza al punto de anclaje para buffer de profundidad y las texturas con formatos para almacenar colores se enlazan al punto de anclaje de buffers de colores número cero (GL_COLOR_ATTACHMENT0).

Para el FBO se crearon cuatro texturas, dos para ser utilizadas como buffers de profundidad (DEPTH_TEX1 y DEPTH_TEX2), otra como buffer de colores para guardar la capa que se obtiene en cada pasada (LAYER_TEX) y una última textura de colores para ir guardando el resultado final (FINAL_TEX).

Antes de procesar las distintas capas, se enlaza el FBO y la textura FINAL_TEX se limpia con el color (0, 0, 0, 1).

Al comienzo de cada iteración, al FBO se le enlazan las texturas DEPTH_TEX1 y LAYER_TEX, y se limpian[15]. La función de testing de profundidad se escoge como GL_LESS para que admita los fragmentos si estos tienen menor profundidad que los valores en DEPTH_TEX1. Además, es necesario que el Fragment Shader descarte los fragmentos si estos pertenecen a una capa que fue procesada en una iteración anterior. Esto último se consigue comparando la profundidad de cada fragmento que está siendo procesado con la profundidad almacenada en la textura DEPTH_TEX2, la cual guarda la información de profundidad de los fragmentos que formaron parte de la capa anterior.

Con las condiciones anteriores, por cada pixel, si los fragmentos llegan en orden desde el más lejano al más cercano y hay más de uno que está en capas que aún no se procesan, los colores de los fragmentos se van a combinar, pero se necesita que se guarde solo el color del fragmento más cercano que pase los test, no la combinación. Luego, se escoge la función de blending con glBlendFunc(GL_ONE, GL_ZERO) para que borre completamente el contenido de los pixeles y los reemplace con la información de los fragmentos que van siendo aceptados.

Al final de cada pasada, después de llenar la textura intermedia, al FBO se le enlaza la textura FINAL_TEX, se desactiva el test de profundidad y se dibuja un cuadrado del mismo tamaño que el cuadro de dibujo con la textura LAYER_TEX sobre este. Como las capas están siendo procesadas desde la más cercana a la más lejana, se utiliza la función de blending glBlendFuncSeparate(GL_DST_ALPHA, GL_ONE,GL_ZERO,GL_ONE_MINUS_SRC_ALPHA) y el Fragment Shader, en este paso, se encarga simplemente de pre-multiplicar las componentes RGB de la textura por su componente alpha para cada fragmento [20]. Después mezclar los colores de las texturas, se intercambian la textura DEPTH_TEX1 con DEPTH_TEX2.

Las componentes de la textura final se mezclan con las de la textura LAYER_TEX utilizando las siguientes ecuaciones:

$$RGB_{FINAL\_TEX} = A_{FINAL\_TEX} \cdot \left(A_{LAYER\_TEX} \cdot RGB_{LAYER\_TEX}\right) + RGB_{FINAL\_TEX}$$

$$A_{FINAL\_TEX} = (1 - A_{LAYER\_TEX}) \cdot A_{FINAL\_TEX}$$

Para las componentes RGB_{FINAL_TEX}, la operación $A_{LAYER\_TEX} \cdot RGB_{LAYER\_TEX}$ es realizada en el Fragment Shader.

Finalmente, después de que se ha procesado el número de capas escogido y la textura final contiene el agregado de todas estas, FINAL_TEX mezclada con el color de background utilizando la siguiente formula:

$$Color\ Final = A_{FINAL\_TEX} \cdot RGB_{bg} + RGB_{FINAL\_TEX}$$

[15] LAYER_TEX se limpia con el color (0, 0, 0, 0).

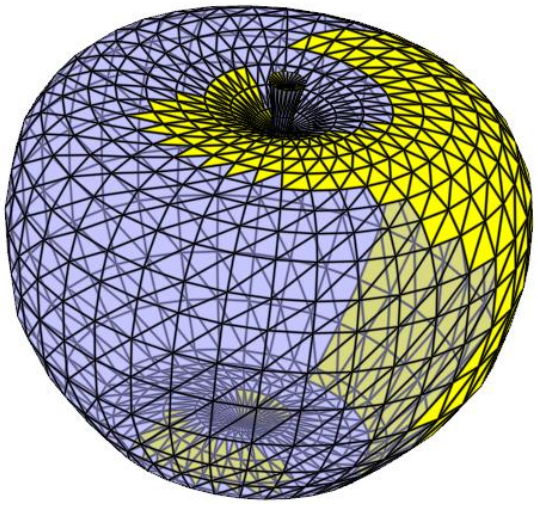


Figura 35 - Escena con Depth Peeling

La escena renderizada utilizando Depth Peeling da el mejor resultado que podría esperar, ya que es análogo a haber ordenado los fragmentos para que el *blending* funcione correctamente. Los inconvenientes de esta técnica es que hay que implementarla en cada Renderer, y además, es costosa computacionalmente, la geometría del modelo se procesará por completo tantas veces como capas se utilicen.

4.3.2. Implementación Doble pasada con Test de Alfa

Es la técnica más básica de todas. Está basado en el uso del Alpha Test de OpenGL.

El Alpha test de OpenGL consiste en un test que se puede habilitar y configurar con ciertas limitaciones. Este actúa sobre la componente alfa de un fragmento que está listo para combinarse, si pasa el test, el fragmento se combina con el color del pixel, de otra forma, se descarta.

La técnica consiste en que el usuario puede escoger un valor de alfa **A**, luego el modelo se dibuja en dos pasadas: en la primera pasada se descartan todos los fragmentos que tengan un alfa menor a **A** (se dibuja la parte más opaca del modelo), luego, en un segunda pasada se descartan los fragmentos con un alfa mayo o igual a **A**. La segunda pasada se realiza con el buffer de profundidad en modo solo lectura.

Con esta técnica, el modelo se procesa completamente dos veces. Además, al habilitar el Alpha test, se consumen más recursos. Los FPS (cuadros por segundo) disminuyen a un 30% aproximadamente.

La técnica consigue buenos resultados si todo el modelo (todas las primitivas) es de un mismo color, o si hay dos colores con distinto grado de opacidad y las primitivas de distintos colores se dibujan en pasadas distintas.

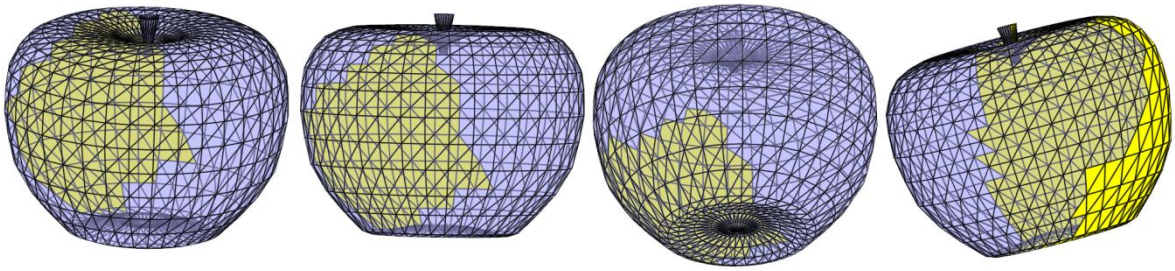


Figura 36 - Doble pasada con Alpha Test: Resultados de doble pasada con Alpha Test y A = 0,9. Color azul tiene una componente alfa de 0.5, amarillo con alfa de 1 y negro de las aristas con alfa de 1.

Las líneas del Wireframe y el color amarillo son dibujados en la primera pasada, modificando el Z-Buffer. En la segunda pasada se añaden los fragmentos de color azulado, los que tienen un alfa de 0,5, y estos no pueden bloquear a otro fragmento, ya que el Z-Buffer está habilitado para solo lectura.

Dejar habilitado el test de profundidad y dejar el Z-Buffer como solo lectura tiene por objetivo impedir que un fragmento que se está dibujando en la segunda fase pueda bloquear a otro fragmento de la segunda fase (estos son translúcidos, no deberían bloquear), pero estos sí pueden ser bloqueados por los elementos más opacos que se dibujaron en la primera fase.

Esta técnica falla en el caso de que existan distintos colores que queden dentro de la misma fase. Por ejemplo, cambiamos la componente alfa del color amarillo a 0,5:

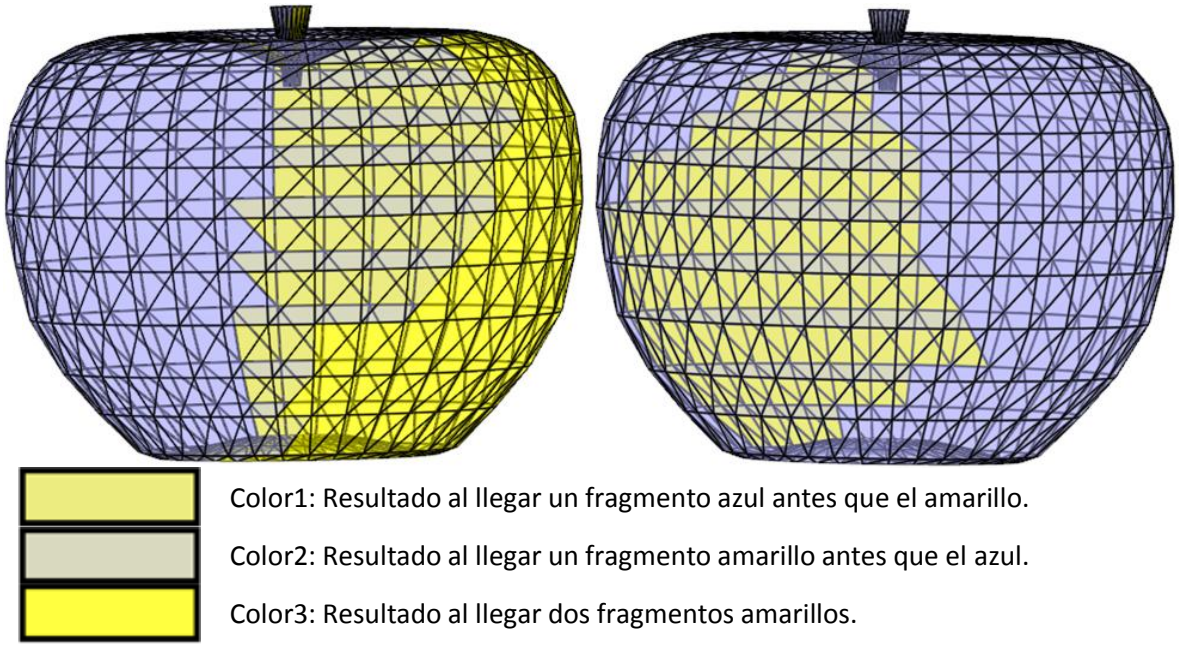


Figura 37 - Fallo de blending con doble pasada Alpha Test: Resultados de doble pasada con Alpha Test y A = 0,9. Color azul con alfa de 0.5, amarillo con alfa de 0,5 y negro de las aristas con alfa de 1.

La combinación del amarillo y el azul, al estar en una misma fase de renderizado, no siempre se hace en el orden correcto. En la foto derecha, los elementos amarillos están en la cara trasera de la manzana, luego, todos los pixeles donde están mezclados azules con amarillos deberían ser de Color 2, pero hay algunos Color 1, lo que quiere decir que los fragmentos aportaron al pixel en el orden inverso. Esta técnica no funciona para este caso.

4.3.3. Implementación Weighted Sum

En el primer paso, para la implementación, se utilizó un FBO con un depth buffer asociado y una textura para guardar la imagen generada. La textura de colores se escogió del tipo RGBA32F para poder almacenar valores mayores a 1.0 por componente y se definió del tamaño de la ventana del dibujo original. Se modificó la función de blending para que en la textura se fueran guardando las sumas ponderadas de colores y la suma de alfas, y se procedió a renderizar utilizando el *Renderer* escogido. De este paso obtenemos una textura con todos los datos por pixel que necesitamos para la etapa final.

Luego, se dibuja cuadrado en 2D que cubre todo el espacio de la ventana de dibujo original utilizando la textura generada en el paso anterior y un Shader especial. En el Fragment Shader, el color final se calcula en base a la fórmula de Weighted Sum:

```
vec4 outputColor = vec4(texColor.xyz + backGroundColor.xyz*(1- texColor.w),1.0f);
```

donde texColor vendría a ser el vector de colores obtenido de la textura generada en el paso 1.

Como resultado obtenemos las siguientes imágenes:

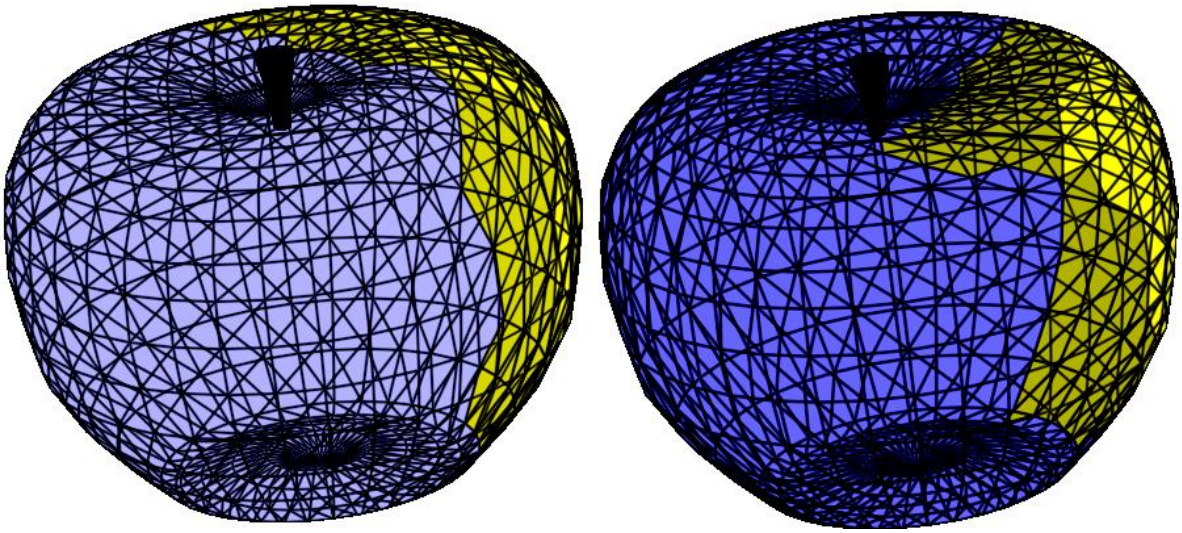


Figura 38 - Weigthed Sum: En la imagen izquierda el color azul tiene un alfa de 0.5, y el amarillo un alfa de 1. En la imagen derecha, ambos colores tienen alfa de 1.

Con el Weighted Sum, ya no existe el problema del orden de los fragmentos, pero aparecen otros problemas:

- Aunque el color tenga un alfa de 1, igual se ve lo que hay detrás. Esto es por causa de que el color final es un promedio ponderado de todos los fragmentos, por lo que no importa si hay un objeto opaco enfrente. Siempre va a dar la apariencia de que todo es traslúcido.
- Los colores con alfa grande se oscurecieron más de lo que deberían.
- No se percibe bien en qué sentido está el modelo. Es imposible saber de qué lado están los elementos seleccionados. La información del orden de los fragmentos se perdió completamente.

Con los resultados anteriores, podemos ver que esta técnica no sirve en absoluto si queremos que en la escena haya objetos opacos también.

Si se utiliza la técnica de Weighted Sum en combinación con una doble pasada con alfa test, podemos dibujar los elementos opacos primero y después los elementos translúcidos usando Wighted Sum, obteniendo un mejor resultado:

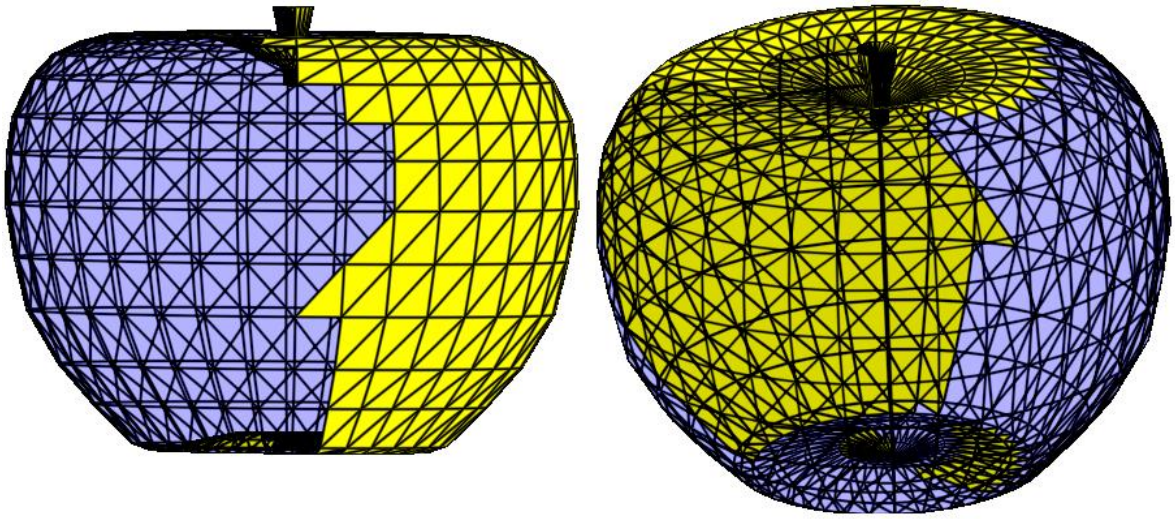


Figura 39 - Weighted Sum junto con doble pasada con Alpha test

Al utilizar ambas técnicas juntas, se procesa dos veces la geometría del modelo, por lo que es considerablemente más costoso computacionalmente que utilizar Weghted Sum solamente, sin embargo, los objetos opacos se muestran correctamente. Además, existe una percepción de orden entre objetos opacos y no opacos. Entre objetos traslucidos sigue no existiendo un orden de profundidad perceptible y la imagen se sigue viendo más oscura de lo que debería ser.

4.3.4. Implementación Weighted Average

Weighted Average se implementó de forma similar a Weighted Sum. La diferencia está en que hay que contar la cantidad de fragmentos que contribuyen a cada pixel, modificar la función en el Fragment Shader que calcula el color final y pasarle una textura adicional con los contadores fragmentos por pixel.

Se utilizó el mismo Fragment Shader que para Weighted Sum, y simplemente se colocó un condicional basado en un variable Uniform junto con un nuevo sampler. Luego, se le da el valor a esta variable y se enlaza el sampler con la nueva textura dependiendo de la configuración que escogió el usuario.

El problema principal es poder generar la textura con los contadores de fragmento que contribuyeron por pixel.

Para resolver eso, al FBO se le enlazó una textura en la posición del Stencil Buffer. Se habilitó el Stencil Test y se configuró para que siempre pasaran los fragmentos. Para poder contar los fragmentos, se escogió que el valor del Stencil Buffer en la posición

correspondiente al fragmento se incrementaría en 1 cuando el fragmento pasara ambos test (Depth Test y Stencil Test), de otra forma, el valor se mantiene.

Con esta configuración, en la textura asociada al Stencil Buffer quedan guardados los contadores de fragmento por cada pixel. Esta textura posteriormente es enlazada a un sampler del Fragment Shader para el procesamiento final.

Esta técnica, al igual que Weighted Sum, hace que aunque una primitiva sea opaca se vea traslucida. Por lo tanto, cuando hay objetos opacos presentes que deberían bloquear a otros, se recomienda habilitar la pasada doble con Alpha Test para que la parte opaca se dibuje primero.

4.4 Transformaciones geométricas

El paso completo de las coordenadas en la malla original hacia las coordenadas en 2D de la pantalla, en el contexto de OpenGL, se puede ver en el siguiente diagrama:

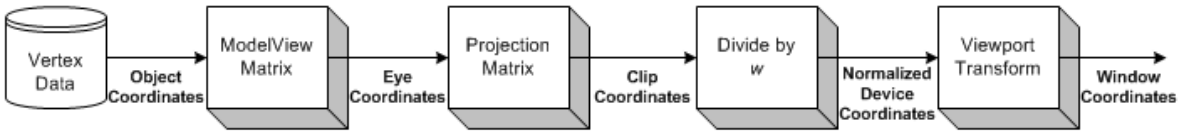


Figura 40 - Transformadas OpenGL **[33]**

En este diagrama, los cubos representan transformaciones sobre las coordenadas. Las flechas, con sus etiquetas, representan el paso por un sistema de coordenadas. Aquí se describe el proceso de transformación base de OpenGL y los sistemas de coordenadas que utiliza.

Object Coordinates vendrían a ser las coordenadas de la malla en el sistema de coordenadas original.

La matriz de ModelView, vendría a estar formada por la multiplicación de las matrices:

- Matriz que centra el model y lo escala para que todos los puntos estén máximo a una distancia de 0,5 del centro.
- Matriz de escalado. El escalado es definido por el usuario utilizando la rueda del mouse.
- Matriz de rotación. El objeto rota sobre su centro (al haber sido centrado antes). La rotación también es definida por el usuario, utilizando el click derecho o izquierdo del mouse mientras lo arrastra por la pantalla.

- Matriz de traslación. La traslación es definida por el usuario utilizando las flechas del teclado.

Al multiplicar las coordenadas por la matriz ModelView, las coordenadas quedan en el espacio *Eye Coordinates* (coordenadas de la cámara).

Luego, la matriz de proyección puede ser escogida por el usuario. Puede escoger una proyección ortogonal o una proyección en perspectiva, escogiendo el ángulo.

De los siguientes pasos, se encarga OpenGL. Después de que las coordenadas fueron pasadas al espacio de coordenadas Clip, OpenGL las divide por su coordenada w (estaban en coordenadas homogéneas) y finalmente las multiplica por la matriz de transformación ViewPort, la cual las pasa al espacio en 2D de la pantalla.

Para ahorrar cantidad de cálculos, se calcula la matriz ModelView y se multiplica por la de proyección, todo en la CPU. Luego, a los Shaders se envía una única matriz que transforma las coordenadas del modelo original al espacio Clip, con una sola multiplicación.

Existen casos donde es útil poder llevar las coordenadas hasta el espacio de coordenadas de la pantalla. Por ejemplo, para dibujar las líneas de la malla de alambres encima de un modelo sólido o para saber si algún elemento está quedando dentro del cuando de selección, se calculan las coordenadas en el espacio de la ventana.

Para calcular las coordenadas en el espacio de la ventana, primero multiplicamos por la matriz MVP (ModelViewProjection), luego dividimos el vector de coordenadas por la componente w, dejándolas en el espacio NDC. Finalmente multiplicamos las coordenadas NDC por la matriz ViewPort.

4.5. Definición formato local de mallas

Para el presente trabajo, uno de los requerimientos era diseñar un formato de almacenamiento de mallas para el visualizador desarrollado.

Con la experiencia de cargar diversos modelos en distintos formatos, se encontró que la mayor cantidad del tiempo se invertía en calcular las relaciones de vecindad entre vecinos. Estos cálculos pesaban aún más en las mallas de poliedros, donde los polígonos tienen muchos más vecinos que en una malla de superficie.

Dadas estas observaciones, se decidió que el formato debería guardar estas relaciones de vecindad entre polígonos en el archivo. Esto último aceleraría el tiempo de carga, pero también haría que aumente el tamaño de los archivos.

También, se decidió que el formato guardaría los archivos en binario. De esta forma no hay que parsear los números, simplemente se leen bytes. Esto aumenta la velocidad de carga del modelo y disminuye el espacio que requiere el archivo, pero viene con la complejidad de tratar con binarios.

El formato diseñado tiene la siguiente estructura:

```
Tipo de malla <int>
Número de vértices <int>
Por cada vértice v:
	Coordenadas  x del vértice v <float>
	Coordenadas  y del vértice v <float>
	Coordenadas  z del vértice v <float>
Número de polígonos <int>
Por cada polígono p:
	Número de vértices que forman el polígono <int>
	Por cada vértice v que forma el polígono:
		Id del vértice v <int>
Por cada polígono p:
	Número de polígonos vecinos al polígonos p <int>
	Por cada polígono vecino n:
		Id del polígono n <int>
Número de poliedros <int> (si fuera el caso)
Por cada poliedro p:
	Número de polígonos que forman el poliedro <int>
	Por cada polígono n que forma el poliedro:
		Id del polígono v <int>
```

Al visualizador se le agregó un componente para exportar en este formato y uno para leer desde este formato.

4.6. Dibujar con Qt y OpenGL

La forma tradicional de dibujar utilizando OpenGL en un entorno Qt, es crear una clase que extienda *QGLWidget* y re-implementar las funciones de alto nivel *resizeGL*, *paintGL* e *initializeGL*. En este caso, la clase que extiende es *CustomGLViewer*.

Esta técnica basta para la mayoría de los casos, pero no era suficiente para el visualizador en desarrollo. Tal como se puede suponer del nombre, la función *paintGL* está pensada para que realice llamadas a la API de OpenGL, pero no funciona correctamente si se intenta dibujar utilizando un *QPainter* de Qt.

Qt tiene una completa librería para dibujar y para la aplicación desarrollada era especialmente útil QPainter, con el cual se pueden dibujar imágenes (QImage), textos y

rectángulos en dos dimensiones, entre otras cosas. Por ejemplo, se utilizó para dibujar la caja con los FPS, o para dibujar los Ids de los elementos sobre el modelo renderizado (en el caso que se haga utilizando la CPU).

Para poder mezclar OpenGL junto con las herramientas de visualización de Qt, hay que re implementar la función *paintEvent*. Esta función es la encargada de procesar el evento de re pintado de cualquier Widget y también de *QGLWidget*.

Dentro de *paintEvent* es posible crear un *QPainter*. Cada vez que se hagan llamadas directas a la API de OpenGL, debemos guardar antes el estado de OpenGL y recuperarlo después. Solamente hay que tener los siguientes puntos en consideración:

- El constructor de *QPainter* automáticamente llama a *glClear()* a menos que se haya llamado antes a *setAutoFillBackground(false)*.
- El destructor de *QPainter* automáticamente llama a *QGLWidget::swapBuffers()* para intercambiar el buffer visible con el que no se estaba mostrando. Eso no sucede si se ha llamado a *setAutoBufferSwap(false)* antes.
- Mientras que el *QPainter* este activo, podemos intercalar funciones a la API de OpenGL mientras nos preocupemos guardar y recuperar el estado antes y después respectivamente.

4.7. Main Renderer

MainRenderer es una de las clases que extiende *Renderer*. Es uno de los renderer más simples, pero sirve para explicar el funcionamiento de estos y como se debe extender la clase *Renderer*. Tiene un *QWidget* de configuración, utiliza shaders para procesar el modelo y generar la imagen final.

En el panel de configuración, el usuario puede escoger colores para los elementos seleccionados y elementos sin seleccionar, también puede elegir si se dibujan o no las aristas y si se dibujan, puede escoger el grosor de la línea y el color. Además, el usuario puede elegir entre que elementos quiere que se dibujen.

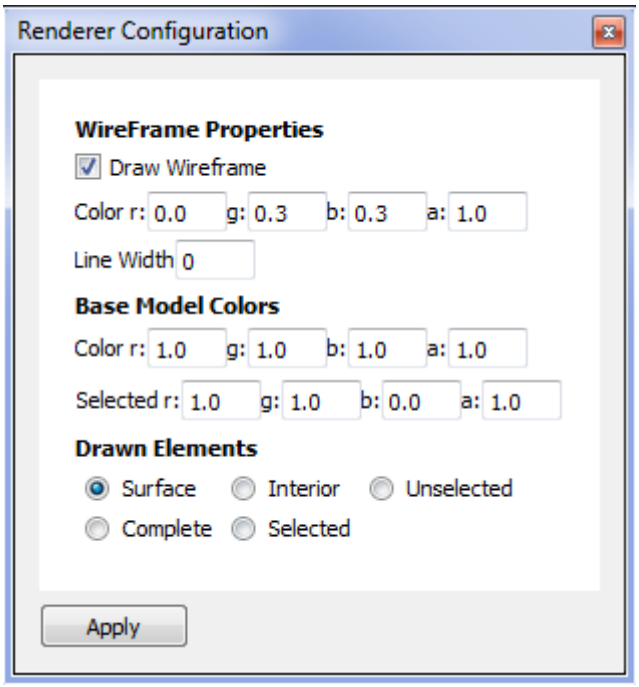


Figura 41 – Panel de configuración de MainRenderer.

La clase *MainRenderer* esta descrita por el siguiente encabezado:

```
class MainRenderer: public Renderer
{
        public:
                MainRenderer();
                virtual ~MainRenderer();
                virtual void glewIsReadyRenderer();
                void draw(RModel*);
                QWidget* getRendererConfigWidget();
                bool hasRendererConfigWidget();
                void applyConfigChanges(RModel * = (RModel*)0);
        private:
                GLuint theProgram;
                MainRendererConfig* config;
};
```

MainRendererConfig es una clase que extiende *QWidget* y contiene la estructura del panel de configuración. También tiene una función *readConfig*(), la cual lee el estado de los componentes en el panel de configuración y asigna los valores de sus variables públicas de acuerdo a dichos estados. *MainRenderer* para configurar como los shaders van a procesar el modelo utiliza los valores de las variables públicas de *MainRendererConfig.*

```
namespace Ui {
class MainRendererConfig;
}
class MainRendererConfig : public QWidget
{
                Q_OBJECT
```

```
        public:
                explicit MainRendererConfig(QWidget *parent = 0);
                virtual ~MainRendererConfig();
                void readConfig();
                glm::vec4 wireFrameColors;
                glm::vec4 baseModelColors;
                glm::vec4 selectedElementColors;
                int wireFrameLineWidthM;
                bool drawWireFrame;
                int elementDrawnOption;
                static const int DRAW_ALL = 0;
                static const int DRAW_ONLY_SURFACE = 1;
                static const int DRAW_ONLY_INTERIOR = 2;
                static const int DRAW_ONLY_SELECTED = 3;
                static const int DRAW_ONLY_UNSELECTED = 4;

        private:
                Ui::MainRendererConfig *ui;
};
```

Cuando el usuario aplica los cambios que realizó en la configuración (con el botón *Apply* de la GUI), se llama a la función *applyConfigChanges()*, la cual llama a la función *readConfig()* del *MainRendererConfig* para actualizarla.

Como *MainRenderer* utiliza shaders y además una clase que extiende de *QWidget* para la configuración, estos deben ser inicializados después de que se ha inicializado GLEW. Por esta razón, la inicialización de estos componentes está en la función *glewIsReadyRenderer()*, la cual es llamada una sola vez después de que GLEW se ha inicializado.

```
void MainRenderer::glewIsReadyRenderer(){
        config = new MainRendererConfig();
        ShaderLoadingData vertexShaderData(GL_VERTEX_SHADER);
        vertexShaderData.addFile("Rendering/Renderers/ModelMainRenderer/mmr.vert"
);
        ShaderLoadingData geometryShaderData(GL_GEOMETRY_SHADER);
        geometryShaderData.addFile("Rendering/Renderers/ModelMainRenderer/mmr.geo
m");
        ShaderLoadingData fragmentShaderData(GL_FRAGMENT_SHADER);
        fragmentShaderData.addFile("Rendering/Renderers/ModelMainRenderer/mmr.fra
g");
        std::vector<ShaderLoadingData> shaderList;
        shaderList.push_back(vertexShaderData);
        shaderList.push_back(geometryShaderData);
        shaderList.push_back(fragmentShaderData);
        VertexAttributeBindingData selectAttr = {VERTEX_FLAGS_ATTRIBUTE,
"flags"};
        VertexAttributeBindingData positionAttr = {POSITION_ATTRIBUTE,
"VertexPosition"};
        std::vector<VertexAttributeBindingData> attributeList;
        attributeList.push_back(positionAttr);
        attributeList.push_back(selectAttr);
```

```
        theProgram = ShaderUtils::CreateProgram(shaderList,attributeList);
        if(theProgram == ShaderUtils::FAIL_CREATING_PROGRAM)
                this->ok = false;
        this->applyConfigChanges();
}
```

En el código podemos ver que esta clase utiliza un Vertex Shader, un Geometry Shader y un Fragment Shader para construir el programa de shaders completo. Gracias a las funciones de la clase *ShaderUtils*, la compilación del programa la podemos hacer en una sola línea utilizando la función *CreateProgram(…)*. Si la creación falla, el boolean *ok* tomará valor falso, por lo que *MainRenderer* no sería agregado a la lista de renderers para ocupar.

El Vertex Shader asociado al programa es el siguiente:

```
#version 400
in vec4 VertexPosition;
in uint flags;
uniform vec4 ModelBaseColor;
uniform vec4 SelectedElementColor;
uniform mat4 MVP;
out vec4 Color;
out uint VFlags;
void main()
{
    Color = ModelBaseColor;
    if((flags&1u)==1u)
        Color = SelectedElementColor;
    gl_Position = MVP*VertexPosition;
    VFlags = flags;
}
```

En las dos primeras líneas del código se pueden ver los dos atributos asociados a los vértices que se utilizan. El primero es un vector de 4 dimensiones en punto flotante, el segundo atributo *flags* corresponde a un unsigned int correspondiente al *RVertexFlagAttribute* asociado al vértice. Este shader se encarga de escoger el color correcto del vértice dependiendo de si esta seleccionado o no. Además, transforma la posición original del vértice al Clipping Space utilizando la matriz MVP. De este shader salen dos atributos, el color escogido y el unsigned int flags sin modificación.

Luego, para el Geometry Shader, el código está resumido y la versión completa se encuentra en el Anexo F.

```
#version 400
layout( triangles ) in;
layout( triangle_strip, max_vertices = 3 ) out;
in vec4 Color[];
```

```
in uint VFlags[];
in vec4 VClipPosition[];
noperspective out vec3 GEdgeDistance;
out vec3 IgnoreEdges;
out vec4 ColorToFrag;
uniform mat4 ViewportMatrix;
uniform int ElementDrawOption;
struct LineInfo {
    int Width;
    vec4 Color;
    int isDrawn; //1 = wire, 0 = no wire
};
uniform LineInfo Line;
bool primitiveIsDrawn();
bool isFlagEnabled(uint f);
vec3 getVerticesDistanceFromOppositeEdge(…);
vec3 getWireFrameIgnoredEdges(…);
void main()
{
   if(primitiveIsNotDrawn()){
        vec3 heights = vec3(0.0,0.0,0.0);
        if(Line.isDrawn==1){
                heights =
                getVerticesDistanceFromOppositeEdge(…);
        }
        GEdgeDistance = vec3( heights.x, 0, 0 );
        gl_Position = VClipPosition[0];
        ColorToFrag = Color[0];
        EmitVertex();
        GEdgeDistance = vec3( 0, heights.y, 0 );
        gl_Position = VClipPosition[1];
        ColorToFrag = Color[1];
        EmitVertex();
        GEdgeDistance = vec3( 0, 0, heights.z );
        gl_Position = VClipPosition[2];
        ColorToFrag = Color[2];
        EmitVertex();
        EndPrimitive();
    }
}
```

El Geometry Shader recibe como variables de entrada arreglos de variables del mismo tipo de las variables de salida del Vertex Shader. Estos arreglos tienen tres elementos cada uno, ya que los elementos son siempre triángulos. Este shader se encarga de generar los datos necesarios para poder dibujar las aristas de un color distinto sobre el modelo.

La técnica extraída del libro *OpenGL 4.0 Shading Languaje Cookbook*[16] calcula la altura asociada a cada vértice, y crea un vector de distancias para cada uno. El vector de distancias asociado a un vértice, es la distancia que tiene hacia cada arista. [14]

[16] En el libro se puede apreciar la técnica con más detalle

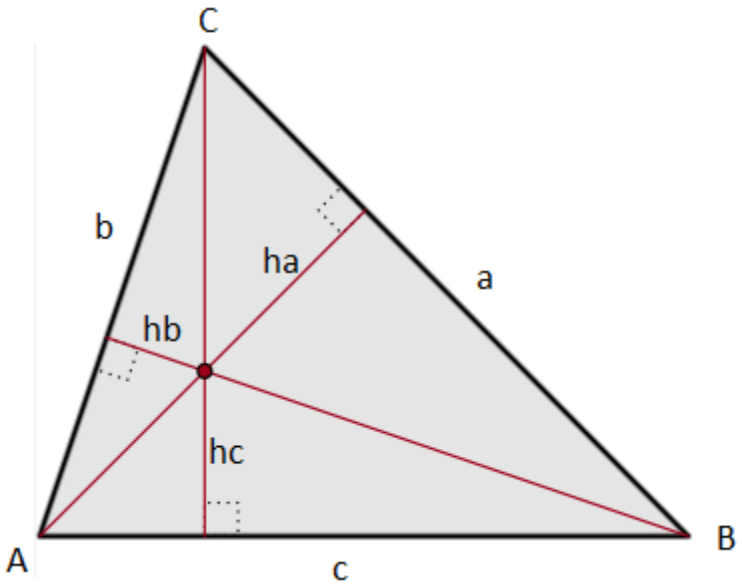


Figura 42 – Triángulo y sus alturas

Los vectores de distancias asociados a los vértices A, B y C serían, (ha, 0, 0), (0, hb, 0) y (0, 0, hc) respectivamente. Estos vectores son variables de salida del shader por lo que son interpolados para cada fragmento que se obtenga del triángulo.

El Fragment Shader se encarga de darle el color final al fragmento dependiendo de si está ubicado sobre una arista o si esta al interior del triángulo, para verificar esto toma en consideración el ancho de la línea especificado por el usuario. El código es el siguiente:

```
#version 400
struct LineInfo {
    int Width;
    vec4 Color;
    int isDrawn; //1 = wire, 0 = no wire
};
uniform LineInfo Line;
noperspective in vec3 GEdgeDistance;
in vec3 IgnoreEdges;
in vec4 ColorToFrag;
out vec4 FragColor;
void main() {
    if(Line.isDrawn==1){
        float d = 10.0f;//more than line width
        if(IgnoreEdges[0]==0.0f)
            d = min(d,GEdgeDistance.x);
        if(IgnoreEdges[1]==0.0f)
            d = min(d,GEdgeDistance.y);
        if(IgnoreEdges[2]==0.0f)
            d = min(d,GEdgeDistance.z);
        float mixVal = smoothstep(Line.Width -1, Line.Width +1, d);
```

```
        FragColor = mix( Line.Color,ColorToFrag , mixVal );
    }
    else
        FragColor = ColorToFrag;
}
```

En la función *draw(RModel*)*, se le indica a OpenGL que se utilizará el programa de shaders que hemos creado. Luego, se le proporcionan las variables que declaramos como uniform y enlazamos los atributos de los vértices con los índices de los vectores correspondientes guardados en RModel. Cuando el programa de shaders está listo con los datos necesarios, se procede a dibujar los triángulos utilizando la función *glDrawArrays(...)*. Después de dibujar los triángulos, se deben desactivar los vectores de atributos y desenlazar el programa de shaders junto con el buffer.

```
void MainRenderer::draw(RModel* rmodel){
        if(rmodel->positionDataBufferObject == RModel::NULL_BUFFER)
                return;// Create and set-up the vertex array object
        //Preparación
        glUseProgram(theProgram);
        ShaderUtils::setUniform( theProgram,
                                  "MVP",rmodel->getMVP());
        ShaderUtils::setUniform( theProgram,
                                 "ViewportMatrix", rmodel->getViewPortMatrix());
        ShaderUtils::setUniform( theProgram,
                                 "Line.Width", config->wireFrameLineWidthM);
        ShaderUtils::setUniform( theProgram,"Line.Color",
                                 config->wireFrameColors);
        ShaderUtils::setUniform( theProgram,"Line.isDrawn",
                                 config->drawWireFrame);
        ShaderUtils::setUniform( theProgram, "ModelBaseColor",
                                 config->baseModelColors);
        ShaderUtils::setUniform( theProgram, "ElementDrawOption",
                                 config->elementDrawnOption);
        ShaderUtils::setUniform( theProgram, "SelectedElementColor",
                                 config->selectedElementColors);¿
        glEnableVertexAttribArray(POSITION_ATTRIBUTE);
        glEnableVertexAttribArray(VERTEX_FLAGS_ATTRIBUTE);
        glBindBuffer(GL_ARRAY_BUFFER, rmodel->positionDataBufferObject);
        glVertexAttribPointer( POSITION_ATTRIBUTE, 3, GL_FLOAT, GL_FALSE, 0,
                                                    (GLubyte *)NULL );
        glBindBuffer(GL_ARRAY_BUFFER, rmodel->vertexFlagsDataBufferObject);
        glVertexAttribIPointer( VERTEX_FLAGS_ATTRIBUTE, 1, GL_UNSIGNED_INT, 0,
                                                        (GLubyte *)NULL );
        //Renderizado
        glDrawArrays(GL_TRIANGLES, 0, rmodel->vertexFlagsAttribute.size() );
        //Desenlazar
        glDisableVertexAttribArray(POSITION_ATTRIBUTE);
        glDisableVertexAttribArray(VERTEX_FLAGS_ATTRIBUTE);
        glBindBuffer(GL_ARRAY_BUFFER,0);
        glUseProgram(0);
}
```

4.8. Estrategias de evaluación, estadísticas y almacenamiento de propiedades

Para implementar nuevas estrategias de evaluación, se debe extender la clase EvaluationStrategy. Ahora se revisará la clase TriangleRaidusRatio a modo de ejemplo:

```
class TriangleRadiusRatio:public EvaluationStrategy
{
        public:
                TriangleRadiusRatio();
                virtual ~TriangleRadiusRatio();
                float value( vis::Triangle* t);
                bool isFullFilled( vis::Triangle* t);
                const float getNullValue();
};
```

El constructor de una estrategia de evaluación, debe llamar al constructor de EvaluationStrategy, dándole el nombre de la estrategia y el nombre de la propiedad evaluada. El nombre de la estrategia se despliega en los combo box. El nombre de la propiedad es visible en la tabla que despliega las propiedades de los elementos seleccionados.

```
TriangleRadiusRatio::TriangleRadiusRatio():
        EvaluationStrategy("Triangle Radius Ratio","Radius Ratio")
{
        weight = TRIANGLE_CRITERIA_WEIGHT_BASE + 0.5f;
}
TriangleRadiusRatio::~TriangleRadiusRatio(){}
```

Como se puede ver, en el constructor también se le da el peso a la estrategia. El peso es para que el programador pueda escoger en qué posición del combobox va a salir esta estrategia.
El destructor, en este caso, es un destructor genérico sin ninguna funcionalidad específica.
Las tres funciones que vienen a continuación, definen sobre qué elementos y cómo se calculará la estrategia.

```
float TriangleRadiusRatio::value( vis::Triangle* t) {
        if(t->hasProperty(this->id))
                return t->getProperty(this->id);
        float radiusRatio = 16*t->getArea()*t->getArea()/
                        (t->getLMax()*t->getLMid()*t->getLMin()*
                        (t->getLMax()+t->getLMid()+t->getLMin()));
        t->addProperty(this->id, radiusRatio);
```

```
        this->addNewValue(radiusRatio);
        return radiusRatio;
}
```

La función *value* se debe encargar de calcular el valor de la propiedad sobre el elemento. En la primera parte, se verifica si la propiedad ya está almacenada en el elemento (triángulo en este caso), si está, no hay nada que hacer y retornamos el valor almacenado. Si la propiedad no estaba, se calcula, se almacena en el elemento y la guardamos con *addNewValue* al vector de estadísticas de la estrategia de evaluación, finalmente retornamos la propiedad. Guardar la propiedad en el elemento también sirve para saber que esta ya se calculó antes sobre el elemento y no debe ser guardada de nuevo en el vector de estadísticas, sino estas se distorsionarán.

```
bool TriangleRadiusRatio::isFullFilled( vis::Triangle* t) {
        return true;
}
```

Como en este caso, esta es una estrategia de evaluación para triángulos, la única función *isFullFilled* que se sobrescribe es la que tiene por argumento un triángulo. La función retorna true siempre, ya que siempre es aplicable sobre triángulos.

```
const float TriangleRadiusRatio::getNullValue(){
        return -1.0f;
}
```

El valor nulo es utilizado para comprobar en algunas ocasiones si la estrategia ha sido aplicada sobre un elemento o no. Debiese ser un valor inválido para la propiedad. En este caso, el Radius Ratio no puede tener un valor negativo, por lo que se escoge como valor nulo el -1.0f.

```
#include <Factories/EvaluationStrategyRegistry.h>
REGISTER_EVALUATION_STRATEGY(TriangleRadiusRatio);
```

Esta última línea, registra la estrategia de evaluación automáticamente al iniciar el programa en el registro de estrategias de evaluación.

Las estadísticas se manejan con un vector ordenado de valores dentro de cada estrategia de evaluación. También, se almacenan los valores mínimos y máximos. La clase EvaluationStrategy tiene la función "int getValuesCountInRange(float from, float to)" que permite extraer la cantidad de elementos que existen entre un rango de valores para una propiedad.

Si se implementa una estrategia de evaluación, de la cual no interesa generar estadísticas para graficar, en el constructor habrá que agregar la siguiente línea:

```
saveStatics = false;
```

Ademas, ya no será necesario utilizar la función addNewValue.

El usuario puede ver los valores de las propiedades por 3 medios: utilizando el visualizador de estadísticas de elementos, escogiendo el *PropertyValuesRenderer* y configurarlo correctamente, y en la tabla de elementos seleccionados.

4.9. Renderer de propiedades.

Es similar al *MainRenderer*, solo que utiliza un atributo adicional para los vértices. El atributo adicional contiene el valor de la propiedad que se quiere visualizar. El usuario puede escoger cual es la propiedad que se quiere visualizar y los colores que se utilizaran para esto.

Primero, se crea un nuevo arreglo de atributos para los vértices. En este arreglo se guardan todos los valores que fueron calculados con la *EvaluationStrategy* escogida. Si la *EvaluationStrategy* no pudo ser aplicada a algún elemento o no fue aplicada algunos elementos, en el arreglo se guarda el valor V_{null} que retorna la función *getNullValue*(). También, se obtienen los valores mínimos y máximos que se obtuvieron de la evaluación.

El usuario define 3 colores distintos para la visualización:

- Color C_{null} para los elementos cuyo valor de la propiedad es V_{null}.
- Color C_{max} asociado al valor máximo V_{max} que tomó la propiedad.
- Color C_{min} asociado al valor mínimo V_{min} que tomó la propiedad.

El VertexShader, encargado de colorear las primitivas, recibe los colores especificados por el usuario a través de variables Uniform y los valores de las propiedades de los vértices como atributos de estos. Luego, cuando la propiedad tiene un valor en $[V_{min}, V_{max}]$, el color se obtiene de una interpolación lineal entre los colores C_{min} y C_{max}, si la propiedad tiene un valor V_{null} se le asigna el color C_{null} al vértice. El código del shader es el siguiente:

```
#version 400
in vec4 VertexPosition;
in float VertexValue;
in uint VertexFlags;
uniform float minValue;
uniform float maxValue;
uniform float noValue;
uniform vec4 NoValueColor;
uniform vec4 MinValueColor;
uniform vec4 MaxValueColor;
```

```
uniform mat4 MVP;
out vec4 Color;
out uint VertexFlagsGeom;
void main()
{
    VertexFlagsGeom = VertexFlags;
    if(noValue != VertexValue){
        float rat = (VertexValue-minValue)/(maxValue-minValue);
        Color = mix(MinValueColor,MaxValueColor, rat);
    }
    else
        Color = NoValueColor;
    gl_Position = MVP*VertexPosition;
}
```

Como se puede ver, primero se calcula un ratio que indica que tan lejano del mínimo está. Si el ratio es 1, significa que el valor es el máximo, si el valor es 0, la propiedad para este elemento vale igual al mínimo. Después, se utiliza la función *mix(…)* de GLSL, la cual permite mezclar dos valores (vec4 en este caso) en proporciones definidas por un número en [0, 1].

El Geometry Shader y el Fragment Shader son iguales a los de *MainRenderer*, excepto en que en el Fragment Shader de este renderer, el color y grosor de las aristas depende de si están delineando un elemento seleccionado o no.

4.10. Visualización de los identificadores de los elementos.

La visualización de identificadores (Id) de elementos se implementó en dos variantes, una que realiza la mayoría del trabajo en la CPU y la otra utilizando la CPU junto con las herramientas de Qt para dibujar texto.

El usuario puede escoger entre cual variante quiere en las preferencias generales del visualizador. Ambas variantes comparten un grupo de configuraciones y además tienen configuraciones específicas de cada una. Tiene configuraciones específicas porque en la versión que utiliza la GPU, como se tiene una visión local de cada primitiva independientemente, existen limitaciones y diferencias en los algoritmos.

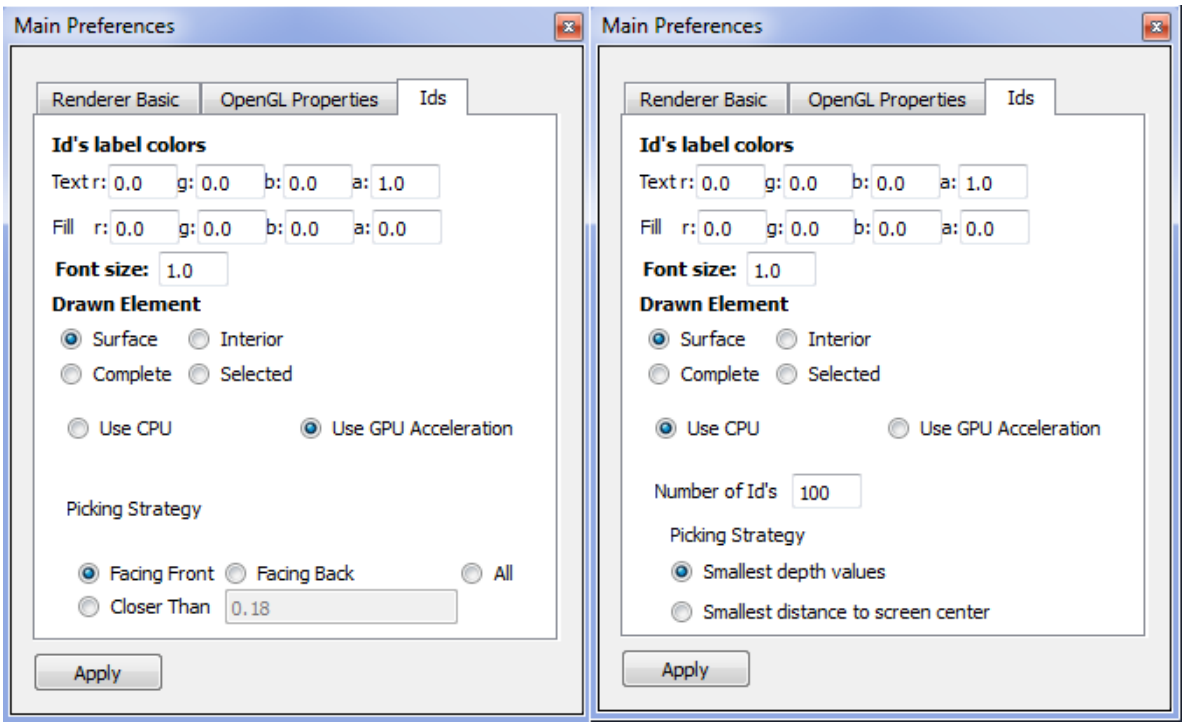


Figura 43 – Panel de configuraciones de visualización de identificadores.

Las configuraciones de colores para las etiquetas, el tamaño de la letra y elegir de cuales elementos se mostrará el Id, son comunes para las dos estrategias.

Estrategia de dibujado de Id's con uso intensivo de la CPU

Cuando se van a dibujar las etiquetas, primero, es necesario calcular las posiciones en la ventana de cada elemento realizando las transformaciones necesarias. Se verifica que este dentro de la ventana y se agrega al conjunto de elementos a los que se les va a dibujar el Id.

El conjunto de elementos fue modelado con la *clase DynamicOrderedContainerWithMaxSize.* Tal como lo dice el nombre, esta clase modela un contenedor que crece dinámicamente hasta cierto límite, además, los elementos en su interior están ordenados por una llave. El usuario puede escoger el tamaño límite de este contenedor y la llave entre dos valores:

- Profundidad del elemento: los elementos del conjunto serán aquellos cuya coordenada Z sea lo más pequeña posible (están más cercanos a la cámara).
- Cercanía al centro de la ventana: los elementos del conjunto serán aquellos cuya distancia euclidiana sea lo más pequeña hacia el centro de la ventana.

El proceso completo consiste, en cada cuadro, vaciar el conjunto fijo de la iteración anterior, recorrer los elementos del tipo deseado, y por cada elemento:

a. Obtener el centro geométrico y transformar sus coordenadas a coordenadas de la ventana.
b. Si las coordenadas del elemento transformadas quedan fuera del campo visual, este se descarta y se pasa al siguiente elemento.
c. Agregarlo al conjunto. El conjunto se preocupa de ordenarse y ajustar el tamaño a medida que se insertan.

Finalmente, se dibujan los Id's de todos los elementos que quedaron guardados en el conjunto.

Estrategia de dibujado de Id's acelerado por GPU

El proceso es similar al de la estrategia que utiliza solamente la CPU, o sea, por elemento se transforman las coordenadas del centro geométrico a coordenadas de la ventana, y si quedan fuera, se descartan.

Con un Geometry Shader, los identificadores son desplegados sobre el modelo generando rectángulos con la siguiente textura aplicada sobre ellos:

La diferencia principal con la técnica en CPU, es que con Shaders no podemos realizar un ordenamiento global, ya que no es posible comparar valores entre primitivas. Por ejemplo, en la GPU no podemos encontrar "los 100 más cercanos a la cámara".

Es por esta razón, que las configuraciones son distintas. Para la técnica en la GPU, para escoger los elementos a los que se les dibuja el Id, sólo fue posible basarse en la normal de los elementos y en una profundidad máxima dada por el usuario.

Comparaciones entre las dos técnicas

El algoritmo para escoger los elementos (ordenándolos) que tiene la técnica que apoya en la CPU, da mejores resultados en cuanto a discriminación de elementos para dibujar los Id's. Cuesta más configurar la técnica acelerada por GPU si se quiere una discriminación más precisa y que aparezco sólo un número pequeño de Id's.

Por otro lado, el rendimiento de la versión acelerada por GPU es mucho mayor. Esta diferencia es posible apreciarla en la siguiente tabla:

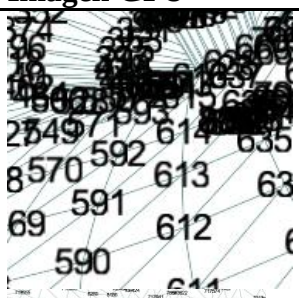
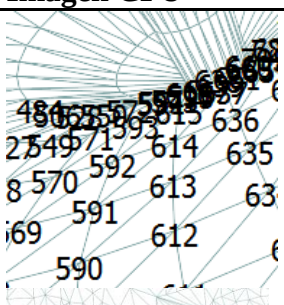
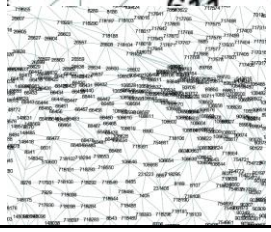
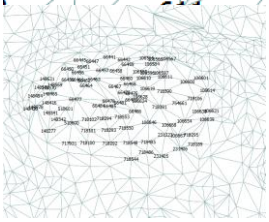
Modelo*	Imagen GPU	Imagen CPU	FPS CPU	FPS GPU
Manzana: **V: 891** **P: 1.704**			125	>1000
Ramsés: **T: 1.652.528** **V: 826.266**			2,4	12

Tabla 2 – Comparación entre identificadores desplegados utilizando shader y utilizando librería grafica de Qt.

*T: Triángulos, V: Vértices.

Cuando el tamaño de letra se incrementa mucho, como en la versión acelerada por GPU, se utiliza una única textura, por lo que las letras pierden mucha calidad.

4.11. Selección utilizando mouse

Captura de eventos, tipos de selección y flujo del controlador

Se definieron dos formas de seleccionar utilizando el mouse:

- Seleccionar elementos con solamente un click, utilizando la coordenada del mouse.
- Seleccionar elementos arrastrando el mouse con el botón izquierdo presionado, formando un rectángulo entre los puntos de inicio y final.

Como arrastrar el mouse con cualquiera de los dos botones presionados se escogió para rotar el modelo, para la selección con mouse hay que mantener Shift presionado.

Primero, para capturar eventos del mouse y del teclado en el cuadro donde se despliega la imagen del modelo, en *CustomGLViewer* se re-implementaron las funciones *mousePressEvent, mouseReleaseEvent, mouseMoveEvent, wheelEvent, keyPressEvent y keyReleaseEvent*. Todas estas funciones tienen por argumento un tipo de evento, del cual se puede extraer información como que botón/tecla se está presionando o soltando, o para el caso de mouse, en que coordenadas del widget está, etc.

Para manejar combinaciones de teclas y/o botones del mouse presionados, se construyó la clase *KeyboardState*.

La clase *KeyboardState*, tal como lo supone su nombre, se encarga de mantener una tabla con las teclas que están presionadas. Cada vez que una tecla se presiona o se suelta, se actualiza en la única instancia de la clase.

```
class KeyboardState
{
        public:
                KeyboardState();
                void keyPressed(int key);
                void keyRelesed(int key);
                bool isKeyPressed(int key);
        private:
                std::unordered_map<int,bool> keymap;
};
```

En el caso de la selección utilizando un rectángulo, era necesario dibujar el rectángulo sobre el modelo. La forma simple, era simplemente renderizar el modelo y luego dibujar el rectángulo sobre él. Pero se encontró que era mucho más eficiente extraer la última imagen del modelo que había sido generada y luego mientras se estira el rectángulo, desplegar la imagen y no re-procesar el modelo.

Luego, como el diseño no se hizo pensando para que ningún componente extensible capturara eventos, ni que una clase que no extendiera a *Renderer* pudiera dibujar, *MouseSelection* tuvo que tener un manejo especial. *MouseSelection* es la única estrategia de selección que no se registra ni se inicializa sola, y la clase controladora tiene un puntero específico a la instancia de esta.

Como *CustomGLViewer* captura el evento de mouse, este revisa si la tecla SHIFT esta apretada, luego, envía una señal de Qt de tipo void selectionBox(int x1,int y1,int x2,int y2), la cual es capturada por el controlador principal. Esta señal lleva información sobre los límites del rectángulo de selección, en caso de ser una selección puntual los puntos (x1, y1) y (x2, y2) son iguales.

El controlador cuando recibe la señal de *selectionBox*, reemplaza la *SelectionStrategy* que estaba seleccionada temporalmente por la instancia que tiene de *MouseSelection*. Luego, configura *MouseSelection* y la procesa como si fuera una *SelectionStrategy común*. Finalmente, vuelve a dejar la *SelectionStrategy* que estaba antes del cambio como la seleccionada.

MouseSelection

La selección por mouse es una clase, que como cualquier otro tipo de selección, extiende a *SelectionStrategy*.

El algoritmo de selección comprueba si a un elemento se le hizo click o si un elemento está dentro del rectángulo de selección, obteniendo las coordenadas de la ventana del elemento y comparándolas con las coordenadas del mouse. Por lo tanto, se hace una comprobación en 2D. Las coordenadas de la ventana de un elemento, se obtienen multiplicándolo por la matriz ModelViewProjection, dividiendo por *w*, y finalmente multiplicando por la matriz de ViewPort.

También, el usuario puede configurar la selección eligiendo:

- Tipo de elemento a seleccionar: vértice, polígono o poliedro.
- Seleccionar elementos de la superficie, interior o todos. En caso de ser elementos superficiales, se puede escoger si se quiere considerar sólo los elementos que miren a la cámara, o en sentido contrario, o todos.
- La selección es completamente nueva (limpia el conjunto de selección antiguo), la selección nueva se une al conjunto antiguo, la selección nueva se intersecta con el conjunto antiguo, se invierte la propiedad de selección que tienen los elementos.

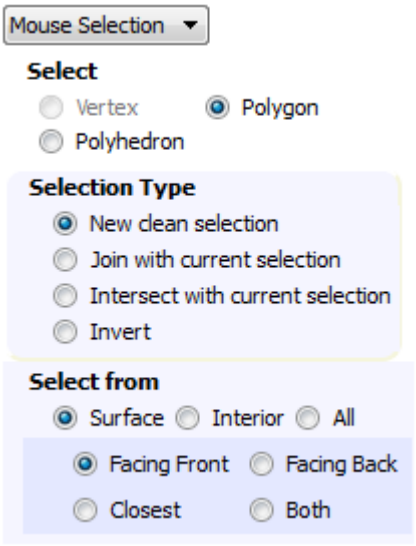


Figura 44 – Panel de configuraciones de MouseSelection.

Para el algoritmo, se implementó una versión que utiliza sólo la CPU y otra que se apoya en la GPU.

La versión de la CPU, debe realizar en la CPU todos los cálculos necesarios para revisar si un elemento reúne las condiciones de selección. Entre estos cálculos, el más pesado es el del paso de coordenadas del modelo a coordenadas 2D de la ventana.

La única posibilidad de extraer cálculos de la GPU utilizando Shaders en OpenGL 4.0 es generando imágenes. Es por esto que no se puede realizar completamente todo el procedimiento en la GPU, es necesario actualizar el conjunto de selección y el estado de cada elemento en la memoria principal.

Para la versión que se apoya en la GPU, se utiliza un FBO para capturar la imagen que se genera con los resultados. Se procesa el modelo, revisando si cada elemento cumple las condiciones o no. Si un elemento cumple las condiciones, en el Geometry Shader, aprovechando que puede generar primitivas nuevas, genera un punto blanco que se dibujará exactamente en la posición de la textura que coincide con el Id del elemento.

El código del Geometry Shader es el siguiente:

```
#version 400
struct VertexData{
    int Id;
    vec2 Position;
    uint Flag;
    vec3 Normal;
};
struct ConfigurationData{
    int onlySurface, onlyInterior, onlyFront, onlyBack,closestToCamera;
    int elementsType;//polygon, vertex or polyhedron
    int selectionType;
    int rectSelection;
    float pixelTolerance;
    vec2 start,end;
    vec3 CameraPosition;
};
uniform float CellWidth;
uniform int BufferWidthHeight;
uniform ConfigurationData config;
layout(triangles) in;
layout(points, max_vertices = 3) out;
in VertexData vdata[3];
flat out uint fcolor;
void getPosition(int id,out vec4 position);
bool selectVertex(VertexData vda);
bool selectPolygon();
void main()
{
    if(config.elementsType != 0){
        if(selectPolygon()){
            getPosition(vdata[0].Id,gl_Position);
            fcolor = 255u;
            EmitVertex();
            EndPrimitive();
        }
    }
}
```

La función *selectPolygon()* verifica si la primitiva cumple con los criterios de selección, si los cumple, se procede a generar un punto blanco en la posición asociada al id del primer vértice del triángulo. La posición asociada al vértice se obtiene con la función *getPosition(...)*.

Después de tener la imagen generada, por cada elemento, se revisa en la textura si el punto correspondiente al Id es negro o blanco. Este proceso es inevitable y no es posible realizarlo en la GPU, por lo que en mallas grandes igual tiene una demora considerable.

Comparaciones

Para comparar las dos versiones del algoritmo, se escogió la malla Neptune.off, por ser de gran tamaño. Con mallas grandes son más visibles las optimizaciones. La malla Neptune tiene 4.007.872 triángulos y 2.003.932 vértices. Los tiempos que se presentan son los promedios de 20 muestras cada uno.

El tiempo que tarda la selección acelerada por GPU está dividido en dos partes:

- TGPU: Tiempo que le toma a la GPU procesar el modelo completo y generar la imagen con la información de los elementos que fueron seleccionados.
- TCPU: Tiempo que le toma a la CPU cambiar el estado de selección de los elementos según los valores de la textura generada en el paso anterior.

En la versión acelerada por GPU, TT = TGPU + TCPU.

Condición inicial	Condición final	Solo CPU	Acelerado por GPU		
			TGPU	TCPU	TT
Ningún elemento seleccionado	Mitad de los elementos seleccionados	3,28	0,24	0,66	0,90
	Todo seleccionado	5,61	0,32	1,89	2,21
Todos los elementos seleccionados	Mitad de los elementos seleccionados	3,30	0,15	1,10	1,25
	Todo seleccionado	4,41	0,27	1,19	1,46

Tabla 3 – Comparación de tiempo (segundos) entre algoritmos de selección (rectángulo) por mouse acelerado por GPU y no acelerado.

Condición inicial	Sólo CPU	Acelerado por GPU		
		TGPU	TCPU	TT
Ningún elemento seleccionado	4,7	0.14	0.17	0,31
Mitad de los elementos seleccionados	4,9	0.13	0.42	0,55
Todos los elementos seleccionados	5,1	0.29	0.51	0,80

Tabla 4 – Comparación de tiempo (segundos) entre algoritmos de selección (puntual) por mouse acelerado por GPU y no acelerado.

Como se puede observar, cuando el algoritmo es acelerado por la GPU, los tiempos mejoran considerablemente. Esto es debido a que toda la lógica de determinar si un elemento está dentro o fuera de la selección se paralelizó en la GPU.

En la versión acelerada por GPU, el tiempo de realizar las trasformaciones de coordenadas y comprobar si el elemento es seleccionado disminuyó a un promedio aproximado de 0.2 segundos. La mayoría del tiempo total (TT) es por en agregar/quitar los elementos del conjunto de selección y actualizar su estado, por esta razón, no se pudieron disminuir más los tiempos.

4.12. Selección y renderizado de modelo intersectado con geometrías convexas

Entre los requerimientos del modelo, se debía poder renderizar la intersección del modelo con algunas geometrías convexas en específico. También, la intersección con geometrías debiese poder ser utilizada para seleccionar elementos.

Para alcanzar este objetivo, se diseñó un componente que heredaba de *Renderer* y *SelectionStrategy* a la vez. De esta forma, era capaz de generar una visualización de la intersección y además, se podía utilizar para seleccionar. La idea de que sea un solo componente que extienda las dos clases base, es para que el usuario pueda fácilmente escoger primero la geometría deseada, visualizándola. Después, simplemente escoge el tipo de selección (unión, intersección, etc.) y elementos (polígonos o poliedros) a seleccionar de la intersección.

A su vez, fue necesario diseñar una forma para que el usuario pudiera especificar geometrías convexas. Para eso, se definieron 3 formas de describir las geometrías:

- Especificando el radio y centro de una esfera. Opcionalmente se puede especificar el número de puntos que se utilizará para dibujarla (no tiene relación con la intersección, solamente con la visualización de la esfera):

```
SPHERE
<Radio>
<Center X> <Center Y> <Center Z>
<N° Vértices>
```

- Especificando una serie de polígonos. A estos polígonos se les extraerá los planos en los que están y la intersección se calcula en cuanto a los planos:

```
POLYGONS
```

```
<N° Vértices>
<Vetice0 X> <Vetice0 Y> <Vetice0 Z>
…
<Vetice(N-1)  X> <Vetice(N-1)  Y> <Vetice(N-1) Z>
<M° Polígonos>
<Polygono0 K Vertices>  < IdVertice(0,0)> … < IdVertice(0,K-1)>
…
<Polygono(M-1) J Vertices>  < IdVertice(M-1,0)> … < IdVertice(M-1,J-
1)>
```

- Especificando directamente los planos a través de la normal y un punto:

```
NORMALS
<N°Planos>
<P1 Origen X> <P1 Origen Y> <P1 Origen Z>
<P1 Normal X> <P1 Normal Y> <P1 Normal Z>
…
<P(N-1) Origen X> <P(N-1)  Origen Y> <P(N-1)  Origen Z>
<P(N-1)  Normal X> <P(N-1)  Normal Y> <P(N-1)  Normal Z>
```

El componente tiene dos configuraciones distintas, una para *Renderer* donde está todo lo que tiene que ver con visualización (2 pestañas) y otra para *SelectionStrategy* que tiene las propiedades de selección:

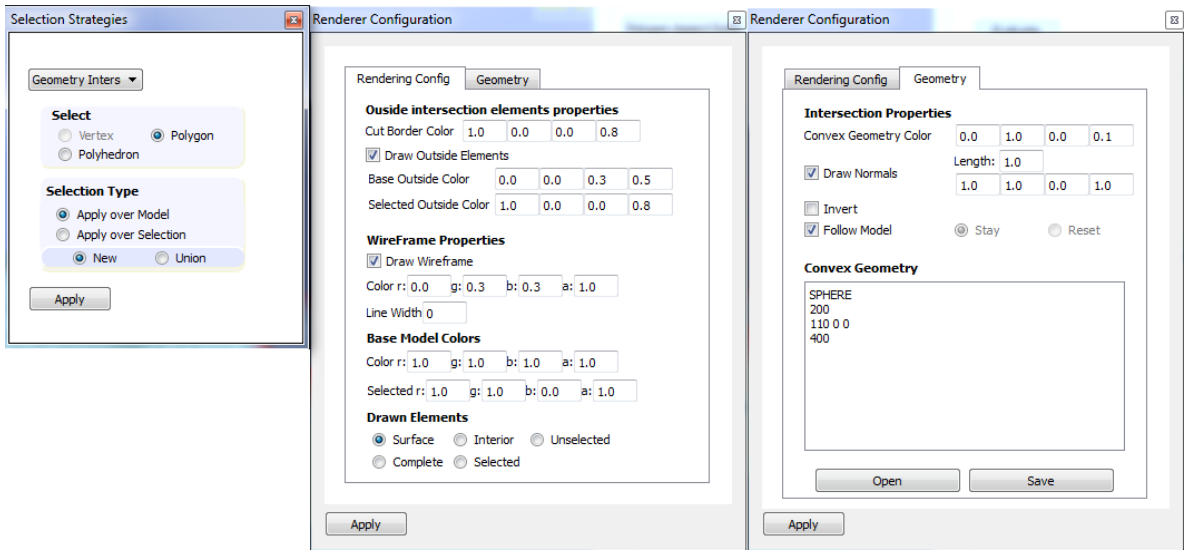


Figura 45 – Paneles de configuraciones de intersección con geometrías convexas.

En la configuración de visualización se especifica la geometría en un cuadro de texto. El cuadro de texto tiene botones para guardar/cargar geometrías. Además, se puede elegir si la geometría se mueve junto con el modelo o se queda en la posición donde está, o vuelve a su posición original.

Cuando la geometría se mueve junto con el modelo, la intersección se calcula una vez antes, y el Shader se preocupa sólo de dibujar los elementos según las demás configuraciones de visualización. Esta opción sirve también para reacomodar la geometría y después dejarla estática. Para optimizar el cálculo previo de la intersección, este se calcula en la GPU.

Cuando la geometría está en una posición fija, se envía al Shader la información que describe los planos de intersección o la información de la esfera (según sea el caso). En este caso, el Shader calcula en tiempo real la intersección de la geometría convexa con el modelo.

Para seleccionar elementos, simplemente toma la configuración de la geometría que se definió en la configuración del *Renderer* para obtener la intersección, y se aplican los criterios de selección. Para obtener la intersección, si no está calculada de antes (la geometría no está siguiendo el modelo), se obtiene con el mismo Shader que se utiliza para pre calcular la intersección cuando la geometría es móvil.

Al igual que en la selección por mouse acelerada con Shaders, para obtener los resultados de la intersección desde el Shader, se dibuja en una textura enlazada a un FBO. Se generan puntos blancos en las posiciones de la textura correspondientes a los identificadores de los elementos que están en la intersección.

Como resultado final, tenemos un módulo que nos permite visualizar y seleccionar elementos con la siguiente vista:

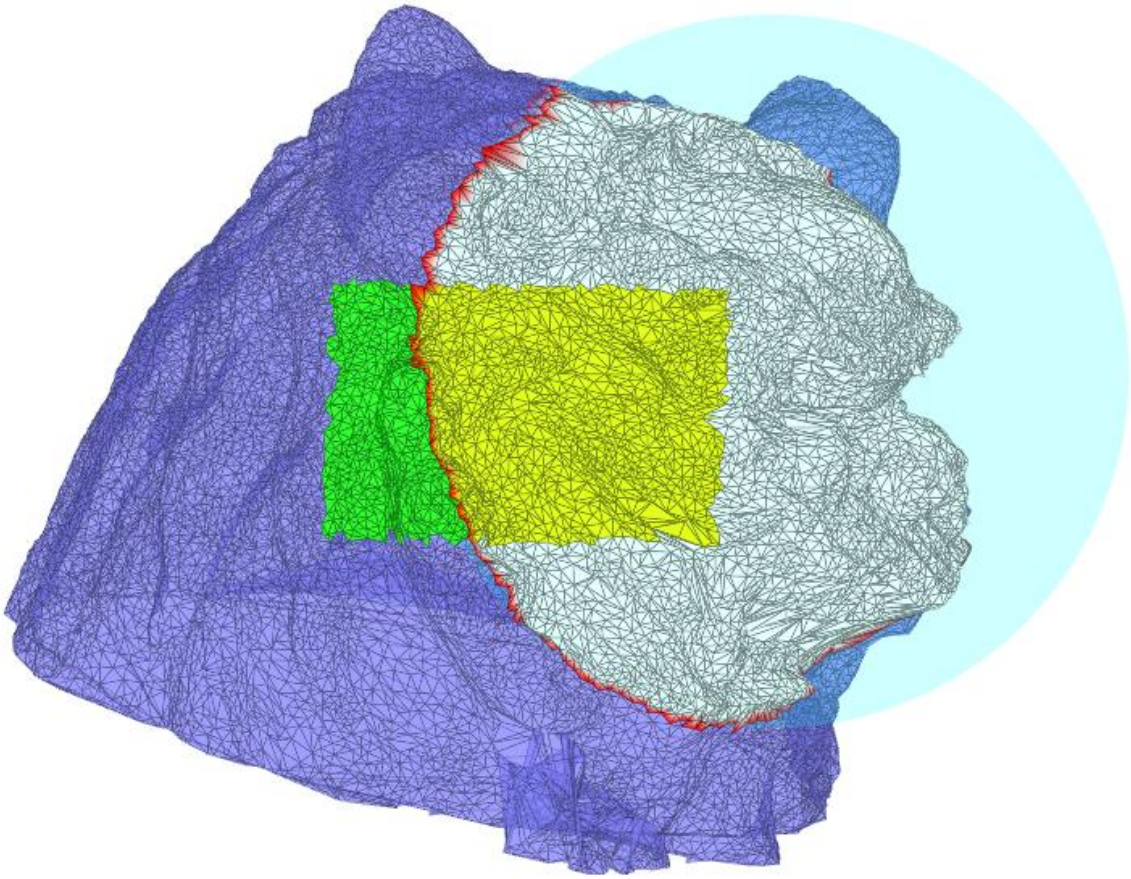


Figura 46 – Resultado de intersección del modelo con esfera.

En la imagen se puede ver la esfera de color celeste claro. Previamente se seleccionaron algunos elementos para la demostración. Los colores (configurables), en este caso, representan lo siguiente:

- Amarillo: Elementos seleccionados que están dentro de la intersección.
- Verde: Elementos seleccionados que están fuera de la intersección.
- Blanco: Elementos no seleccionados que están dentro de la intersección.
- Azul: Elementos no seleccionados que están fuera de la intersección.
- Rojo: Elementos que están en el límite de la intersección.

4.13. Interfaz de usuario

La aplicación tiene un menú superior. Cada ítem del menú puede tener una lista de sub-items (árbol).

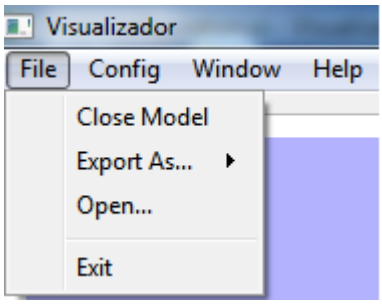


Figura 47 – Submenú File de la GUI.

En el menú *File* encontramos las opciones para cargar un modelo, cerrarlo, exportar los elementos seleccionados o exportar el modelo completo, y la opción de salir del visualizador.

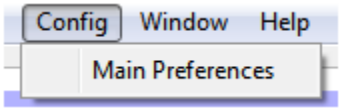


Figura 48 - Submenú Config de la GUI.

En *Config*, por el momento, encontramos una sola entrada. En *Main Preferences* el usuario puede escoger las preferencias generales del visualizador como: forma en que se despliegan los Id's de los elementos, configuraciones generales de OpenGL y propiedades de visualización.

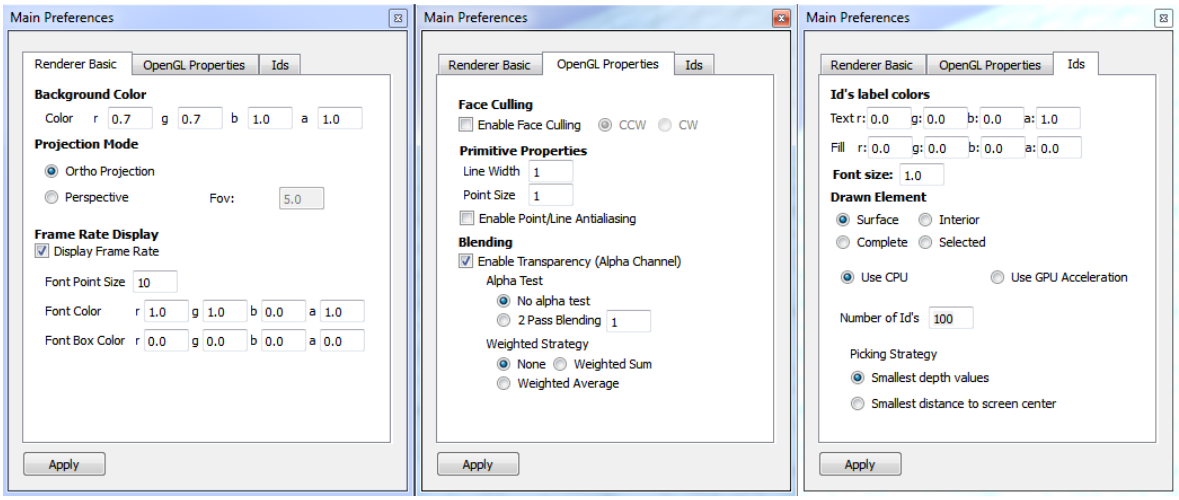


Figura 49 – Panel Main Preferences.

En el menú *Window*, el usuario puede re-abrir widgets que ha cerrado o widgets que no aparecen abiertos al correr la aplicación.

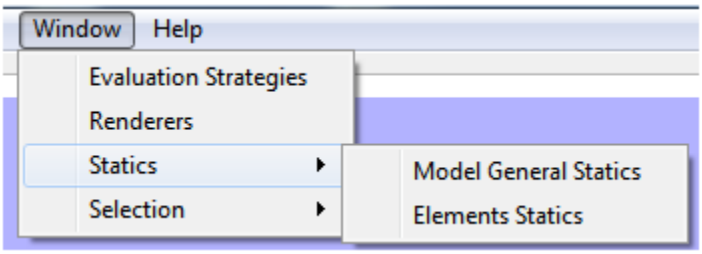


Figura 50 - Submenú Window->Statics de la GUI.

Model General Static consiste en una ventana con un texto que despliega las características generales del modelo. Por ejemplo, para una malla de polígonos con polígonos de 4 y 3 lados:

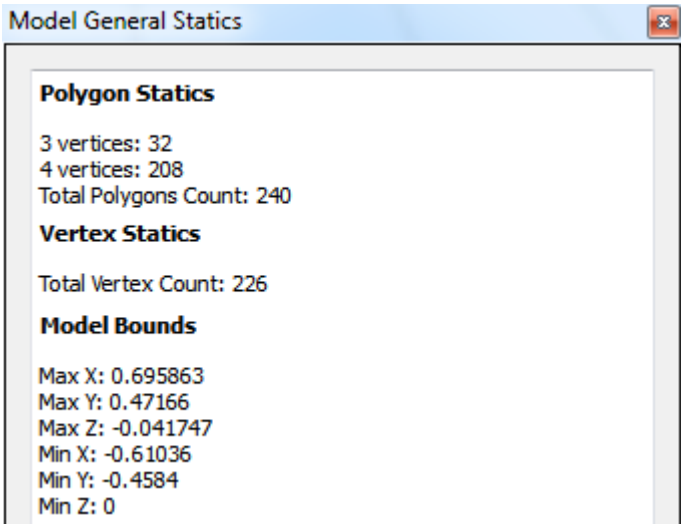


Figura 51 – Panel Model General Statics.

Elements Statics es un widget que permite visualizar las propiedades de los elementos a través de la malla utilizando un gráfico de barras. Al gráfico se le puede configurar la cantidad de barras y el intervalo de los elementos que se van a graficar. También, se puede configurar el color haciendo click sobre las barras. Al pasar el mouse sobre una barra, se despliega la cantidad de elementos que está representando, el porcentaje del total y el intervalo.

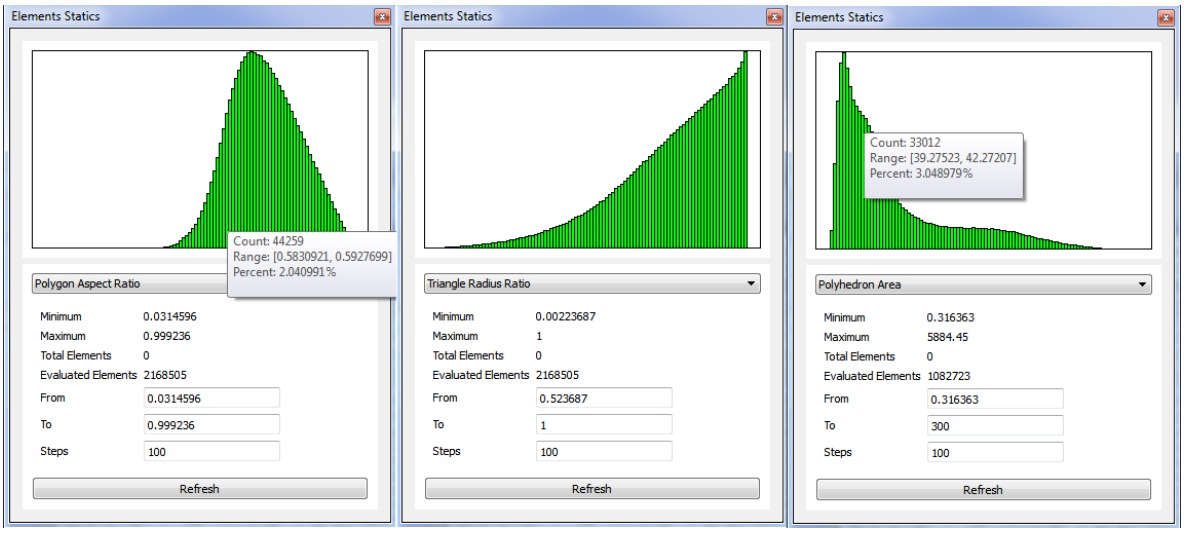


Figura 52 – Panel Elements Statics: Muestra gráficos configurables para visualizar los valores de una de las propiedades que han sido calculadas en los elementos. El usuario puede escoger que propiedad quiere visualizar utilizando el combobox.

Como submenú de *Selection*, tenemos a *Selection Table* y *Selection Strategies*.

Figura 53 - Submenú Window->Selection de la GUI.

Selection Table es una widget donde aparecen los elementos seleccionados. Se muestran las propiedades que se le han calculado y su Id.

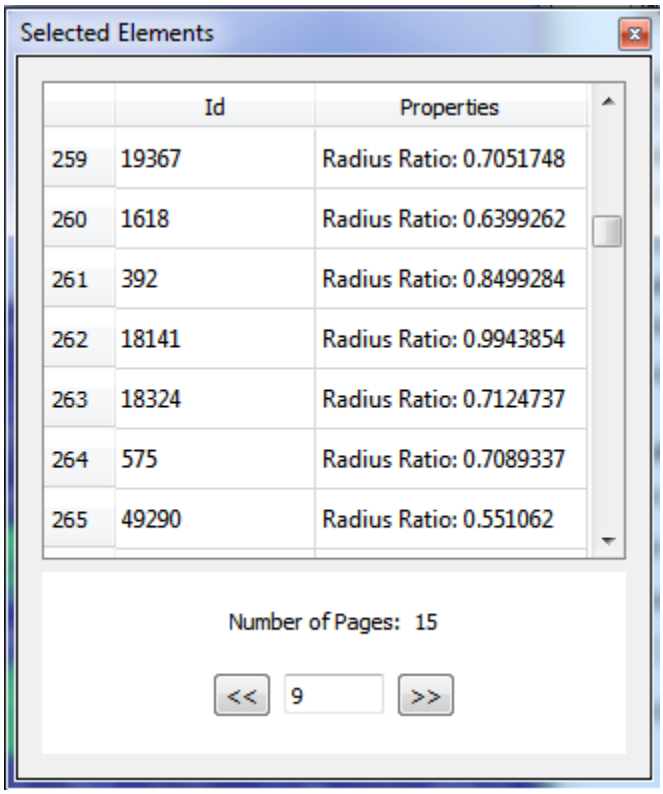


Figura 54 – Panel Selected Elements: Tabla que despliega información de los elementos seleccionados paginada.

Renderers abre el widget anclable para poder seleccionar el *Renderer* a utilizar, configurarlo y poder seleccionar *Renderers* secundarios. *Selection Strategies* abre el widget anclable donde el usuario puede seleccionar elementos utilizando las distintas estrategias de selección. Aquí se configura también la selección por mouse.

En el widget anclable para seleccionar *Renderers*, cuando el *Renderer* es configurable, el botón *Config* estará habilitado. Al hacer click en el botón, se abrirá un nuevo widget de configurarcion del *Renderer*. El botón + se utiliza para agregar un *Renderer* a la lista de *Renderers* secundarios. Para quitar elementos de la lista se utiliza la tecla suprimir.

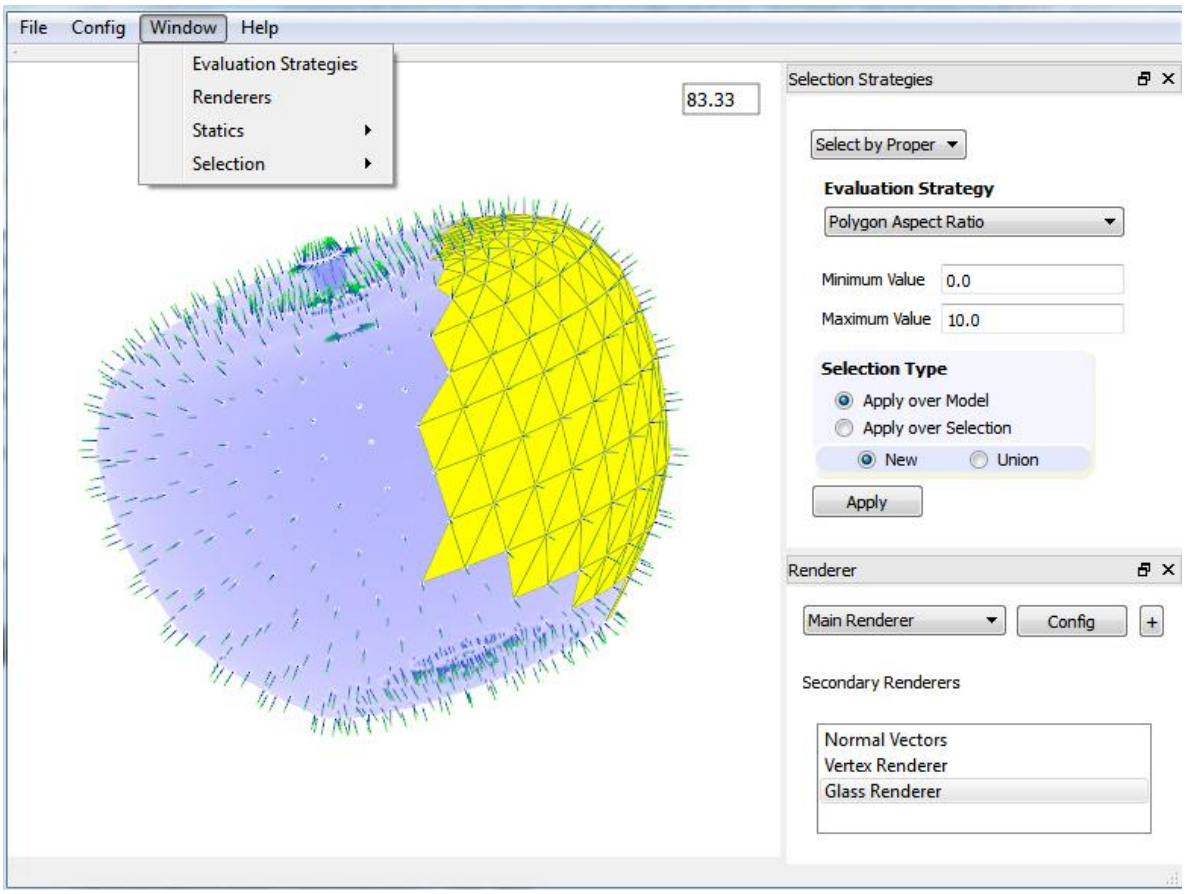


Figura 55 – Vista general del visualizador.

5. Mediciones de rendimiento/eficiencia

El rendimiento del visualizador desarrollado se comparó con los otros visualizadores mencionados en la sección 2.6. Además, se probó el desempeño del formato diseñado para el visualizador.

5.1. Ambiente de pruebas

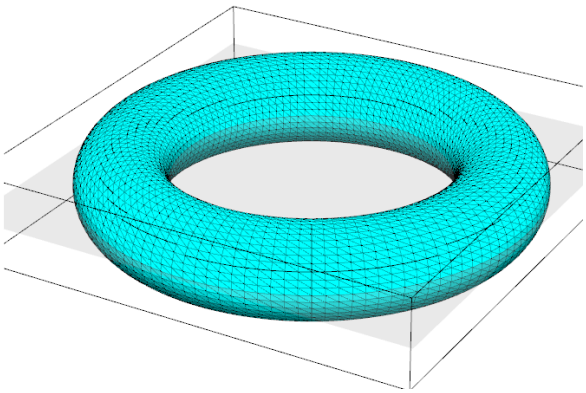
El computador donde se realizaron las pruebas tenía las siguientes características:

- S.O.: Windows 7.
- Procesador: Intel Core i5-3210M CPU, corriendo a 2.5 GHz.
- Tarjeta de Video: NVIDIA GeForce GT 650M.
- Memoria RAM: 8 GB, DDR3, 800 MHz.

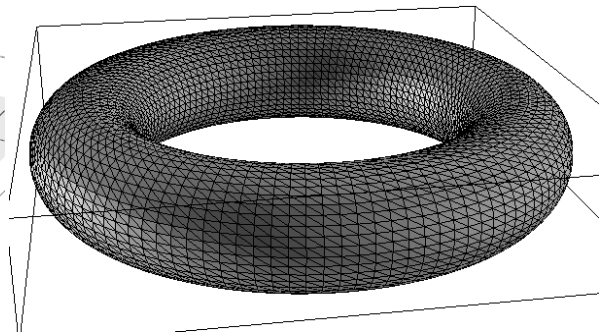
Los visualizadores comparados fueron los siguientes:

- Camarón x64 bits
- TetView
- Geomview 1.9.4 (Solo para mallas de superficie)
- MeshLab 1.3.2 x64 bits

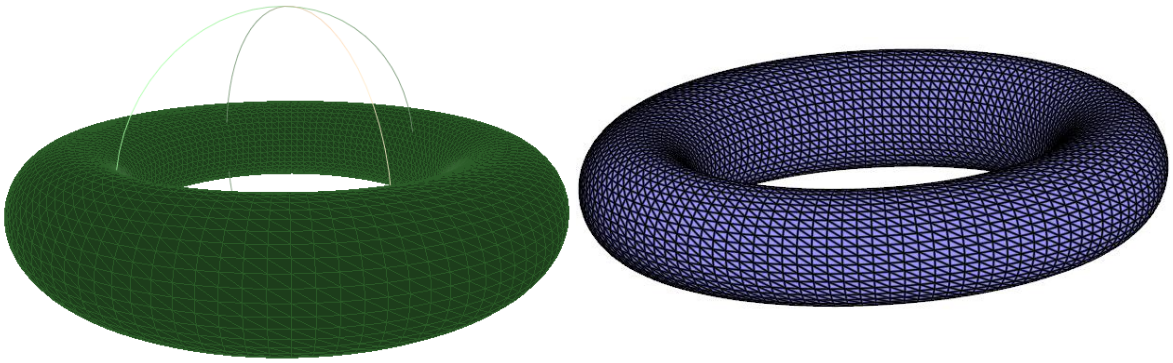
Para todas las pruebas el tipo de visualización escogida fue un modelo de caras solidas (sin transparencias), con sombreado por vértice y con las aristas dibujadas sobre el modelo:



TetView



Geomview



MeshLab **Camarón**

Figura 56 - Comparación visual entre visualizadores.

Los modelos escogidos para las pruebas se dividieron en dos grupos: mallas de poliedros y mallas de superficie, ambos en 3D. Para cada grupo se seleccionaron 11 mallas donde las cantidades de vértices fueran cercanas a 2^{12}, 2^{13}, 2^{14}, 2^{15}, 2^{16}, 2^{17}, 2^{18}, 2^{19}, 2^{20}, 2^{21} y 2^{22}. El número mínimo que se escogió fue el 2^{12} porque todos los visualizadores se desempeñan perfectamente con mallas más pequeñas y no es interesante compararlos en esos rangos, por otro lado, el número mayor escogido fue 2^{22} porque en este rango el desempeño es muy malo para los visualizadores que utilizan tecnologías más actuales (Camarón y MeshLab) y para los más antiguos (TetView y Geomview) son incapaces de manejar mallas de estos tamaños.

Para el caso de las mallas de superficie se escogieron modelos extraídos de distintos repositorios gratuitos en internet, los cuales tenían cantidades de vértices cercanas a los números mencionados, además, todas las mallas fueron transformadas a formato OFF (formato de lectura compartido por los cuatro visualizadores en cuestión) utilizando a Camarón. Para las mallas de poliedros los modelos fueron generados con un programa hecho en Java que generaba puntos utilizando la función nextDouble() de la clase Math.Random y luego se generaba la tetrahedralización utilizando TetGen. El programa Java generó para cada caso la cantidad de puntos exacta, distribuyéndolas radialmente en coordenadas esféricas de la siguiente forma:

- double radio = random.nextDouble()*random.nextDouble()*100.0;
- double phi = random.nextDouble()*Math.PI*2.0;
- double theta = random.nextDouble()*Math.PI;

Como se ve en las fórmulas (y en las figuras de Tabla 6), los modelos tienen una densidad de punto mayor más cerca del centro.

A continuación se describen los modelos que forman parte de ambos grupos utilizando la siguiente nomenclatura:

- N: Nombre del modelo.
- V: Numero de vértices de la malla.
- Pg: Numero de polígonos de la malla.
- Pd: Numero de poliedros de la malla.
- F: Formato del modelo.

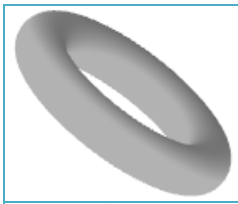
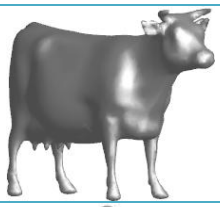
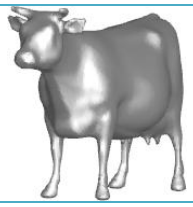
 **Torus** [34] V: 4.800 Pg: 9.600 F: TRI	 **Vaca** [35] V: 8.245 Pg: 16.510 F: OFF	 **Vaca2** [35] V: 16.612 Pg: 33.244 F: OFF
 **Conejo** [36] V: 35.947 Pg: 69.451 F: PLY	 **Mano** [37] V: 50.085 Pg: 99.999 F: OFF	 **Bola de golf** [34] V: 122.882 Pg: 245.760 F: TRI
 **Lucy** [34] V: 262.909 Pg: 525.814 F: TRI	 **Dragón Chino** [36] V: 437.645 Pg: 871.414 F: PLY	 **Ramsés** [38] V: 826.266 Pg: 1.652.528 F:OFF
 **Neptuno** [39] V: 2.003.932 Pg: 4.007.872 F: OFF	 **Estatua Thai** [36] V: 4.999.996 Pg: 9.999.764 F: PLY Binary	

Tabla 5 - Mallas de superficie utilizadas en las comparaciones de visualizadores.

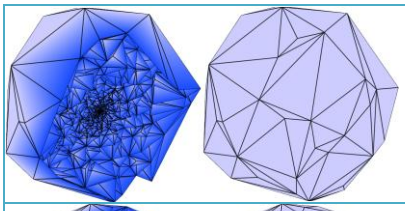
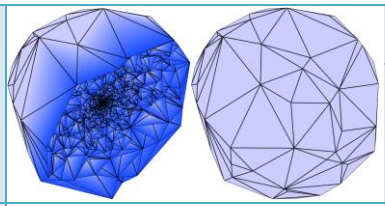
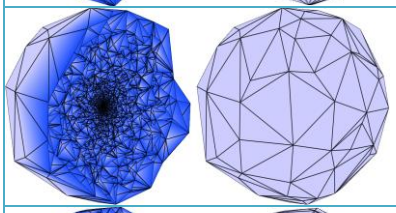
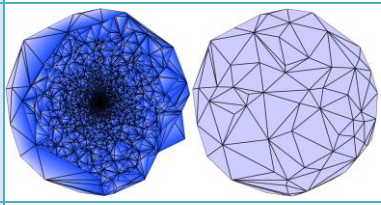
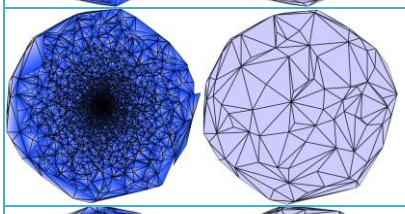
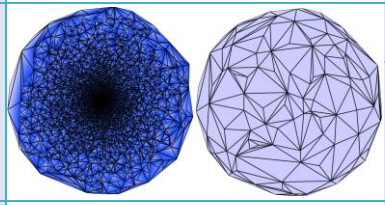
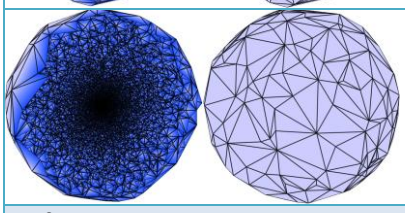
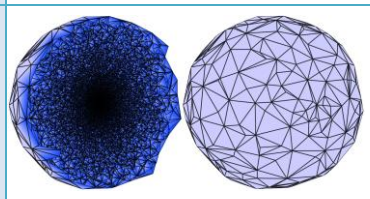
	**Esfera1** V: 4.096 Pg: 54.247 Pd: 27.098 F: ELE		**Esfera2** V: 8.192 Pg: 109.300 Pd: 54.616 F: ELE
	**Esfera3** V: 16.384 Pg: 219.642 Pd: 109.776 F: ELE		**Esfera4** V: 32.768 Pg: 439.898 Pd: 219.890 F: ELE
	**Esfera5** V: 65.536 Pg: 882.974 Pd: 441.409 F: ELE		**Esfera6** V: 131.072 Pg: 1.768.357 Pd: 884.080 F: ELE
	**Esfera7** V: 262.144 Pg: 3.540.332 Pd: 1.770.053 F: ELE		**Esfera8** V: 524.288 Pg: 7.085.413 Pd: 3.542.559 F: ELE
Esfera9 V: 1.048.576 Pg: 14.176.211 Pd: 7.087.920 F: ELE	**Esfera10** V: 2.097.152 Pg: 28.365.692 Pd: 14.182.610 F: ELE	**Esfera11** V: 4.194.304 Pg: 56.738.594 Pd: 28.369.007 F: ELE	

Tabla 6 - Mallas de tetraedros utilizadas en las comparaciones de visualizadores.

Como todos los visualizadores son capaces de manejar mallas de superficie, todos fueron probados con dicho conjunto. En estas pruebas se midieron los siguientes parámetros:

- FPS: Cuadros que dibujan por segundo. Los FPS de los demás visualizadores se midieron utilizando la aplicación Fraps. Los visualizadores externos, tenían una cota superior de alrededor de 60 fps, por lo que se elige este punto como una cantidad de FPS satisfactoria.
- Uso de memoria: Se midió el uso de memoria en el caso de las mallas grandes. En las mallas pequeñas es despreciable.
- Tiempo de carga: Tiempo que se demora un visualizador en abrir un modelo.

Para las pruebas con mallas de poliedros solo participaron los visualizadores Camarón y TetView. En estas pruebas se midieron los mismos parámetros anteriores, junto con 1 adicional:

- FPS Corte: Cuadros que dibujan por segundo cuando el modelo ha sido “cortado” por el plano XY. Los FPS de TetView se midieron utilizando la aplicación Fraps, el

cual tiene una cota superior de 60 fps, por lo que se elige este punto como una cantidad de FPS Corte satisfactoria.

5.2. Resultados de las comparaciones entre visualizadores

	Visualizador				
Malla de superficie		TetView	Geomview	MeshLab	Camarón
	Torus	60	40	60	60
	Vaca	60	24	60	60
	Vaca2	60	13	60	60
	Conejo	57	6	47	60
	Mano	35	4	36	60
	Bola de golf	18	2	15	60
	Lucy	10	1	7	60
	Dragón chino	7	<1	4	60
	Ramsés	5	<1	2	60
	Neptuno	S.R.[17]	<1	1	21
	Estatua Thai	S.R.	<<1	<1	6,5

Tabla 7 - Comparación de cuadros por segundos (FPS) entre visualizadores utilizando mallas de superficie.

	Visualizador		
Malla de tetraedros		TetView	Camarón
	Esfera1	60: 60	60: 60
	Esfera2	60: 60	60: 60
	Esfera3	60: 60	60: 60
	Esfera4	60: 58	60: 60
	Esfera5	60: 47	60: 60
	Esfera6	60: 33	60: 60
	Esfera7	60: 24	60: 21,6
	Esfera8	60: 14	21: 8,8
	Esfera9	60: 9	S.R.
	Esfera10	60: S.R.	S.R.
	Esfera11	S.R.	S.R.

Tabla 8 - Comparación de FPS y FPS Corte (FPS: FPS Corte) entre visualizadores utilizando mallas de tetraedros. Para la Esfera10 TetView no era capaz de realizar el corte utilizando el plano.

*Tetview aparentemente se desempeña muy bien dibujando el exterior de los modelos de tetraedros, por muy grandes que sean, porque para dibujar la capa externa solamente

[17] S.R.: Sin resultados, esto se debe a que el visualizador no fue capaz de cargar el modelo.

procesa los elementos superficiales, mientras que Camarón procesa todos los vértices y descarta los que no están en la superficie.

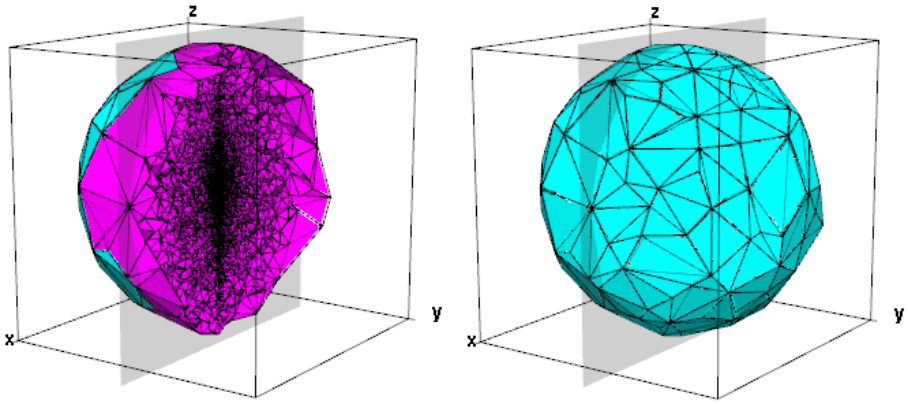


Figura 57 - Corte del modelo utilizando un plano en TetView: La imagen de la izquierda corresponde al renderizado del modelo Esfera5 sin cortar (sólo la superficie). La imagen de la derecha corresponde al renderizado del modelo Torso dividido por el plano XZ.

	Visualizador				
Malla de superficie		TetView	Geomview	MeshLab	Camarón
	Torus	52	5	79	73,8
	Vaca	53	5,5	81	74,3
	Vaca2	57	6,7	85	82
	Conejo	68	9,9	96	102
	Mano	77	12,3	102	116
	Bola de golf	120	24	140	193
	Lucy	190	46	165	444
	Dragón chino	303	74	205	602
	Ramsés	437	137	281	913
	Neptuno	S.R.	326	465	2.086
	Estatua Thai	S.R.	806	970	4.645

Tabla 9 - Comparación del uso de memoria RAM (MB) entre visualizadores utilizando mallas de superficie.

	Visualizador		
Malla de tetraedros		TetView	Camarón
	Esfera1	50	104
	Esfera2	53	148
	Esfera3	56	221
	Esfera4	64	382
	Esfera5	79	698
	Esfera6	110	1.346
	Esfera7	170	2.478
	Esfera8	296	5.010
	Esfera9	531	S.R.
	Esfera10	1.012	S.R.
	Esfera11	S.R.	S.R.

Tabla 10 - Comparación del uso de memoria RAM (MB) entre visualizadores utilizando mallas de tetraedros.

En las Tablas 9 y 10 se puede apreciar una de las principales desventajas de Camarón: la cantidad de memoria que utiliza el visualizador desarrollado es considerablemente mayor a la que utilizan los demás visualizadores, por lo que si se pretende utilizar con mallas de gran tamaño, el computador debe tener mucha memoria RAM disponible y utilizar la versión de x64 bits.

	Visualizador				
Malla de superficie		TetView	Geomview	MeshLab	Camarón
	Torus	<0,5	<0,5	<0,5	0,141
	Vaca	0,5	<0,5	<0,5	0,31
	Vaca2	0,9	<0,5	<0,5	0,521
	Conejo	1,1	0,7	0,5	1,05
	Mano	1,5	0,9	0,9	1,581
	Bola de golf	2,9	1,4	1,3	3,338
	Lucy	5,9	2,4	2,6	7,276
	Dragón chino	10,3	3,8	4,1	12,032
	Ramsés	18,9	7,3	7,8	22,56
	Neptuno	S.R.	17,1	18,4	55,725
	Estatua Thai	S.R.	31	46	144,867

Tabla 11 - Comparación del tiempo (segundos) que le toma cargar a cada visualizador cada malla de superficie.

	Visualizador		
Malla de tetraedros		TetView	Camarón
	Esfera1	<0,5	4,134
	Esfera2	<0,5	8,028
	Esfera3	0,8	16,123
	Esfera4	1,1	32,588
	Esfera5	2,1	64,466
	Esfera6	3,2	122,904
	Esfera7	6,6	245,503
	Esfera8	13,9	541,956
	Esfera9	28,8	S.R.
	Esfera10	59	S.R.
	Esfera11	S.R.	S.R.

Tabla 12 - Comparación del tiempo (segundos) que le toma cargar a cada visualizador cada malla de tetraedros.

Como se puede ver en las Tablas 11 y 12, el visualizador desarrollado es varias veces más lento que los otros visualizadores en cargar los archivos. Esto se debe principalmente a que se demora en calcular las relaciones de vecindad entre los elementos que componen los modelos. Cabe destacar que las mediciones para el visualizador Camarón se tomaron precisamente sacando la diferencia de tiempo entre el comienzo y fin de la carga, mientras que en los demás, se utilizó un timer externo. Con las mallas de tetraedros, el efecto del costo de cargar las vecindades de polígonos se ve potenciado debido a que en este tipo de modelo, sus triángulos tienen muchos más triángulos vecinos que en una malla de superficie, donde tienen a lo más 3.

5.3 Pruebas sobre el nuevo formato

Las pruebas se realizaron utilizando sólo el visualizador desarrollado en el presente trabajo, en la versión de 64 bits.

Para realizar pruebas del formato, se calculó cuánto demoraba en cargar modelos en su formato original[18] y en formato nuevo. También, se comparó el tamaño de los archivos en distinto formato.

El tiempo de carga está dividido en dos partes. La primera consiste en la carga del modelo, la construcción de las relaciones entre elementos y el cálculo de las normales. La segunda

[18] El "Formato Original" de las mallas corresponde a los formatos en que se hicieron todas las pruebas, es decir, para las mallas de superficie corresponde al formato OFF y para las de tetraedros el formato ELE.

parte consiste en la construcción del modelo de datos que se envía a la tarjeta de video a partir del modelo original. Con el nuevo formato, el tiempo que se verá afectado será sólo el de la primera parte, por lo que las comparaciones se realizaran en cuanto a este.

		Formato Original	Formato Nuevo (VISF)	Mejora[19]
Malla de superficie	Torus	0,141	0,066	113,64%
	Vaca	0,31	0,119	160,50%
	Vaca2	0,521	0,247	110,93%
	Conejo	1,05	0,577	81,98%
	Mano	1,581	0,768	105,86%
	Bola de golf	3,338	1,655	101,69%
	Lucy	7,276	3,566	104,04%
	Dragón chino	12,032	6,033	99,44%
	Ramsés	22,56	11,773	91,62%
	Neptuno	55,725	29,192	90,89%
	Estatua Thai	144,867	75,825	91,05%
Malla de tetraedros	Esfera1	4,134	0,643	542,92%
	Esfera2	8,028	1,33	503,61%
	Esfera3	16,123	2,69	499,37%
	Esfera4	32,588	5,499	492,62%
	Esfera5	64,466	11,253	472,88%
	Esfera6	122,904	22,463	447,14%
	Esfera7	245,503	45,787	436,18%
	Esfera8	541,956	94,146	475,65%
	Esfera9	S.R.	S.R.	S.R.
	Esfera10	S.R.	S.R.	S.R.
	Esfera11	S.R.	S.R.	S.R.

Tabla 13 - Comparación de tiempo de carga de malla entre formato original y formato nuevo.

Como se puede observar, los tiempos de carga disminuyeron en más de la mitad. Y en el caso de los modelos de tetraedros, la disminución fue mucho más dramática debido a la gran cantidad de relaciones de vecindad entre polígonos que ya no se tuvieron que calcular.

		Formato Original	Formato Nuevo (VISF)	Mejora
Malla de superficie	Torus	290	357	-18,77%
	Vaca	497	613	-18,92%
	Vaca2	1.095	1.234	-11,26%
	Conejo	2.441	2.591	-5,79%
	Mano	4.068	3.712	9,59%
	Bola de golf	8.481	9.121	-7,02%
	Lucy	18.308	19.513	-6,18%

[19] Mejora = (100 * Resultado en Formato Original / Resultado en Formato Nuevo) - 100, con dos decimales. Una mejora con porcentaje negativa, es en realidad un empeoramiento.

	Dragón chino	32.614	32.344	0,83%
	Ramsés	63.687	61.325	3,85%
	Neptuno	165.001	148.730	10,94%
	Estatua Thai	381.728	371.086	2,87%
Malla de tetraedros	Esfera1	1.154	4.612	-74,98%
	Esfera2	2.321	9.303	-75,05%
	Esfera3	4.673	18.712	-75,03%
	Esfera4	9.464	37.460	-74,74%
	Esfera5	19.088	75.245	-74,63%
	Esfera6	39.172	150.727	-74,01%
	Esfera7	82.004	301.798	-72,83%
	Esfera8	167.915	604.069	-72,20%
	Esfera9	341.093	S.R.	S.R.
	Esfera10	717.740	S.R.	S.R.
	Esfera11	1.476.596	S.R.	S.R.

Tabla 14 - Comparación del tamaño de los archivos de las mallas entre formato original y formato nuevo.

Los resultados se explican en que por un lado, el tamaño del archivo disminuyó al haber escogido formato binario. Pero, por otro lado, aumenta al guardar las relaciones de vecindad entre polígonos, sobre todo en el caso de las mallas de tetraedros.

En las mallas de superficie grande, el ahorro en espacio producto del formato binario es mayor que el espacio que utiliza guardar la vecindad de polígonos. Esto no ocurre con las mallas de poliedros.

5.4 Comparaciones de funcionalidades de visualizadores

Cada uno de los visualizadores con que se comparó el visualizador desarrollado, tienen características propias y algunas que comparten. Para tener una mejor idea de las capacidades de cada uno, se ha elaborado la siguiente tabla:

	Geomview	Tetview	MeshLab	Camarón
En Desarrollo	No	No	Si	Si
Nube de puntos	No	Si	Si	Si
Mallas de polígonos	Si	Si	Si	Si
Mallas de poliedros	No	Si	No	Si
Mallas no estructuradas	Si	Si	Si	Si
Mallas con elementos mixtos	Si	Si	Si	Si

Evaluación de elementos	No	No	No	Si
Selección de elementos	No	No	Si	Sólo polígonos y poliedros
Desplegar Ids de elementos	No	Sólo vértices	Si	Si
Mostrar Elementos Independientemente	Sólo aristas y polígonos	Si	Si	Si
Mostrar elementos en conjunto	Si	Si	Si	Si
Mostrar Normales Independientemente	Si	No	No	Si
Exportar Mallas	No	No	Si	Si
Texturas	No	Si	Si	No
Colores desde archivo	Si	Si	Si	No
Operaciones para modificar malla	No	No	Si	No
Múltiples mallas en una misma escena	Si	No	No	No
Código Abierto	Si	Si	Si	Si
Multi-plataforma	Si[20]	Si	Si	Si
Trasladar, rotar, zoom	Si	Si	Si, tiene un sistema más elaborado que permite hacer los movimientos de forma más precisa.	Si
Realizar cortes en la geometría, para seleccionar o sólo visualizar interior	No	Utilizando un plano.	Utilizando un plano.	Utilizando figuras convexas, planos y esferas.
Animación	Los modelos pueden tener inercia.	No	No	No
Modelos de iluminación.	Pocos modelos, configurables.	Modelo único, poco configurable	Muchos modelos, configurables. Extensibles.	Muchos modelos, configurables. Extensibles.
Formatos de lectura	OOGL (QUAD, OFF, MESH, VECT,	SMESH, OFF, MESH, STL, PLY, ELE,	PLY, STL, OFF, OBJ, 3DS,	ELE, OFF, TS, TRI, M3D, PLY,

[20] Para poder correrlo en un ambiente que no sea Linux, se necesita un ambiente que emule el de Linux.

	etc.)	FACE, EDGE	COLLADA, PTX, V3D, PTS, APTS, XYZ, GTS, TRI, ASC, X3D, X3DV, VRML, ALN	VISF(local)
Formatos de escritura	Ninguno	Ninguno	PLY, STL, OFF, OBJ, 3DS, COLLADA, VRML, DXF, GTS, U3D, IDTF, X3D	ELE, OFF, TS, VISF(local)

Tabla 15 – Comparación de características y funcionalidades entre visualizadores.

6. Conclusiones

6.1 Resultados obtenidos

En el presente trabajo de titulación se diseñó e implementó un visualizador de mallas geométricas capaz manejar mallas estructuradas, no estructuradas y mixtas en 3D. La aplicación nos da la posibilidad de cargar estas mallas desde los formatos: ELE, OFF, TS, TRI, PLY, M3D y VISF, además puede evaluar los elementos que la componen, seleccionarlos, renderizarlos con distintas técnicas y exportar por partes o completamente la malla en los formatos: ELE, OFF, TS y VISF.

Mediante el uso de patrones de diseño y la programación orientada a objetos, especialmente con la herencia y el polimorfismo, se pudo diseñar una aplicación fácilmente extensible agregando nuevas componentes que extiendan una serie de clases base (Renderer, SelectionStrategy, etc.). Además, estas extensiones son capaces de registrarse con una línea de código en el mismo archivo en que se implementa la nueva clase, sin tener que modificar las clases de los registros.

Se obtuvo un diseño dividido en los siguientes módulos:

- Renderers: Está compuesto de varias componentes que se encargan de generar distintas formas de visualizar el modelo.
- Estrategias de selección: Conjunto de componentes encargadas de implementar las distintas estrategias para seleccionar elementos de la malla cargada, entre ellas están la selección por mouse, selección por propiedades, expandir la selección a elementos vecinos y selección utilizando la intersección con una geometría convexa definida por el usuario.
- Estrategias de evaluación: Conjunto de componentes utilizadas para evaluar los elementos del modelo. Las evaluaciones que fueron implementadas son: mínimo y máximo ángulo diedro, mínimo y máximo ángulo sólido, razón de aspecto (arista más corta/arista más larga) y razón de radio para triángulos.
- Lectores de malla: Son todas las clases que implementan el código para cargar mallas de los distintos formatos.
- Exportadores de malla: Clases que realizan la exportación del modelo o una selección de este a un formato particular.
- Registros de componentes: Contiene todos los registros que la aplicación utiliza, como el registro de renderers, de estrategias de selección, etc. Existe un registro por cada clase base que está hecha para generar extensiones a partir de ella.

La GUI del visualizador le da al usuario la posibilidad de escoger entre los distintos componentes para evaluar, seleccionar y visualizar los elementos, y configurar dichos componentes. También le otorga controles para poder rotar, trasladar, contraer y expandir la malla para poder apreciar el modelo desde distintas posiciones y ángulos.

El diseño de la malla y sus elementos que guardan varias relaciones de vecindad, además de la utilización de tecnologías de shaders para procesarla, permite que la aplicación sea capaz de trabajar con mallas de millones de elementos y mantener aun así una respuesta rápida. Cabe destacar que al guardar las relaciones de vecindad, que aceleran algunos algoritmos, tiene un alto costo en memoria RAM y estas no son útiles para todos los casos de uso, es por esto, que sería bueno darle la posibilidad al usuario de elegir si quiere que estén almacenadas o se calculen cada vez que sean necesarias haciendo más lentos los algoritmos de procesamiento.

En la aplicación se utilizan una gran cantidad de programas de shaders en OpenGL para generar imágenes con distintos efectos de iluminación o para procesar grandes cantidades de datos que no tienen un objetivo final visual. Además, se abordó el problema de la transparencia dependiente del orden de procesamiento de las primitivas utilizando las técnicas Depth Peeling, Weighted Average, Weighted Sum y doble pasada con Alpha Test. Se implementaron varias técnicas y se dejó a criterio del usuario escoger cual utilizar, porque estas varían mucho en los resultados de visualización obtenidos y cómo perjudican el rendimiento.

Finalmente, la aplicación obtenida es una herramienta altamente configurable y extensible que cumple con los objetivos planteados para la memoria, y que permite al usuario obtener un gran provecho de ella para visualizar y evaluar mallas. Además, para un programador, la aplicación al ser de código abierto sirve como ejemplo de implementación de distintas técnicas y esta, a su vez, es fácilmente extensible. También,

6.2 Trabajo futuro

Como propuesta, el trabajo en relación a la aplicación puede ser extendido en los siguientes ámbitos:

- Modificar el visualizador para que soporte la carga de varios modelos en la misma escena y que estos tengan la posibilidad de simular inercia (ejemplo, como en Geomview)
- Soporte de lectura y visualización de mallas con texturas asociadas.

- Soporte de lectura y visualización de mallas con atributos extra asociados a los elementos que la componen.
- En la exportación de mallas, darle la posibilidad al usuario de escoger alguna propiedad o color para que sea exportado junto con los elementos.
- Agregar estrategias de evaluación, selección, visualización y formatos para cargar/exportar mallas.
- Modificar el diseño en caso de ser necesario para que sea posible reproducir animaciones de distintos formatos. Una posible forma de abordar este problema con el diseño actual es agregar una clase que extienda a *Renderer* para mostrar una animación. Esta nueva clase avanzaría un cuadro de la animación cada vez que se llame a la función *draw*. Luego, para avanzar la animación, el usuario tendría que forzar a que se redibuje con las teclas de control de cámara y la animación se detiene si el usuario no interactúa. Otra forma podría ser que a la clase base *Renderer* se le agregue una función para saber si hay una animación para reproducir, y si la tuviera, iterativamente dibujarla cuando el usuario apriete alguna tecla.
- Agregar modos de carga de mallas para el visualizador. Considerando que las relaciones de vecindad son principalmente útiles para las estrategias de evaluación y para el criterio de selección por vecindad, es que el usuario debería poder cargar las mallas escogiendo si es prioridad el ahorro de memoria o el tiempo de cálculo de algunas propiedades. Si éste desea ahorrar memoria, los criterios de vecindad se calculan cada vez que son necesarios y no estarían almacenados desde un comienzo.
- Agregar un módulo para poder editar la malla. Mover vértices y borrar los repetidos, borrar elementos y asignar colores a estos mismos, refinar las mallas, etc.

7. Referencias

1. **Watt, Alan y Watt, Mark.** *Advanced Animation and Rendering Techniques: Theory and Practice.* Wokingham, England : Addison-Wesley, 1992.

2. Sharcnet. *Artículo online sobre la calidad de mallas geométricas.* [En línea] [Citado el: 17 de Enero de 2013.] https://www.sharcnet.ca/Software/Fluent13/help/flu_ug/flu_ug_mesh_quality.html.

3. **Mascaró, Javiera.** *Visualizador de Mallas Geométricas Mixtas 3D.* Facultad de Ciencias Físicas y Matemáticas, Universidad de Chile. Santiago, Chile : 2010.

4. **Foley, Van Dam, Feiner y Hughes.** *Computer Graphics: Principles and Practice.* Second. s.l. : Addison-Wesley, 1990.

5. **Cook, John.** johndcook. *Artículo online sobre trigonometría esférica.* [En línea] [Citado el: 17 de Enero de 2013.] http://www.johndcook.com/spherical_trigonometry.html.

6. Wolfram. *Artículo en Wolfram sobre la trigonometría esférica, el cual trata el tema de triángulos esféricos.* [En línea] [Citado el: 17 de Enero de 2013.] http://mathworld.wolfram.com/SphericalTrigonometry.html.

7. **Shewchuk, Jonathan Richard.** *What Is a Good Linear Finite Element? Interpolation, Conditioning, Anisotropy , and Quality Measures.* University of California at Berkeley : Department of Electrical Engineering and Computer Sciences, December 31, 2002.

8. **McKesson, Jason L.** Arcsynthesis. *Online book: Learning Modern 3D Graphics Programming, Objects in Motion.* [En línea] 2012. [Citado el: 17 de Enero de 2013.] http://www.arcsynthesis.org/gltut/Positioning/Tutorial%2006.html.

9. **McKesson, Jason L.** Arcsynthesis. *Online book: Learning Modern 3D Graphics Programming, About this Book.* [En línea] 2012. [Citado el: 17 de Enero de 2013.] http://www.arcsynthesis.org/gltut/About%20this%20Book.html.

10. Opengl. *Artículo online de OpenGL: Rendering Pipeline Overview.* [En línea] [Citado el: 17 de Enero de 2013.] http://www.opengl.org/wiki/Rendering_Pipeline_Overview.

11. **McKesson, Jason L.** Arcsynthesis. *Online book: Learning Modern 3D Graphics Programming, Graphics and Rendering.* [En línea] 2012. [Citado el: 17 de Enero de 2013.] http://www.arcsynthesis.org/gltut/Basics/Intro%20Graphics%20and%20Rendering.html.

12. **McKesson, Jason L.** Arcsynthesis. *Online book: Learning Modern 3D Graphics Programming, Boundaries and Clipping.* [En línea] 2012. [Citado el: 17 de Enero de 2013.] http://www.arcsynthesis.org/gltut/Positioning/Tut05%20Boundaries%20and%20Clipping.html.

13. OpenGL. *Manual de la página oficial de OpenGL, que describe el funcionamiento dle Stencil Buffer.* [En línea] [Citado el: 17 de Enero de 2013.] http://www.opengl.org/sdk/docs/man/xhtml/glStencilFunc.xml.

14. **Wolff, David.** *OpenGL 4.0 Shading Language Cookbook.* Birminghan, UK : Packt Publishing Ltd., 2011.

15. Nvidia. *Sitio web oficial de Nvidia con las especificaciones de los distintos modelos de tarjetas de video.* [En línea] [Citado el: 17 de Enero de 2013.] http://la.nvidia.com/object/geforce_family_la.html.

16. **McKesson, Jason L.** Arcsynthesis. *Online book: Learning Modern 3D Graphics Programming, World Space.* [En línea] 2012. [Citado el: 17 de Enero de 2013.] http://www.arcsynthesis.org/gltut/Positioning/Tutorial%2007.html.

17. OpenGL. *Artículo online de OpenGl, Transparency Sorting.* [En línea] [Citado el: 17 de Enero de 2013.] http://www.opengl.org/wiki/Transparency_Sorting.

18. **Crassin, Cyril.** Icare3D Blog. *Artículo Online: Fast and Accurate Single-Pass A-Buffer using OpenGL 4.0+.* [En línea] [Citado el: 17 de Enero de 2013.] http://blog.icare3d.org/2010/06/fast-and-accurate-single-pass-buffer.html.

19. **Barta, Pál y Kovács, Balázs.** *Order Independent Transparency with Per-Pixel Linked Lists.* Budapest, Hungary. : Budapest University of Technology and Economics, 2011.

20. **Bavoil, Louis y Myers, Kevin.** *Order Independent Transparency with Dual Depth Peeling.* Santa Clara, CA : NVidia, 2008.

21. **Rigazzi, Alessandro.** *Order Independent Transparency.* [En línea] [Citado el: 17 de Enero de 2013.] http://www.slideshare.net/acbess/order-independent-transparency-presentation.

22. *Sitio web oficial de la librería matemática GLM.* [En línea] [Citado el: 17 de Enero de 2013.] http://glm.g-truc.net/.

23. GLM. *Sitio web oficial de la librería matemática GLM.* [En línea] [Citado el: 17 de Enero de 2013.] http://www.wikivs.com/wiki/GTK_vs_Qt.

24. Geomview. *Sitio web oficial del visualizador Geomview.* [En línea] [Citado el: 17 de Enero de 2013.] http://www.geomview.org/.

25. Tetview. *Sitio web oficial del visualizador Tetview.* [En línea] [Citado el: 17 de Enero de 2013.] http://tetgen.berlios.de/tetview.html.

26. MeshLab. *Sitio web oficial del visualizador MeshLab.* [En línea] [Citado el: 17 de Enero de 2013.] http://meshlab.sourceforge.net/.

27. **Singh, Jaspreet y Kaur, Mrs. Pinkiparampreet.** *Object Oriented Programming Using C++.* Pune, India : Technical Publications Pune, 2008.

28. **Parsons, David.** *Object Oriented Programming with C++.* Segunda. New York : Continuum, 1997.

29. **Balagurusamy, E.** *Object Oriented Programming with C++.* Cuarta. New Delhi : Tata McGraw-Hill, 2008.

30. **Gamma, Erich, y otros.** *Design Patterns: Elements of Reusable Object-Oriented Software.* s.l. : Addison-Wesley, 1994.

31. Oodesign. *Sitio web sobre patrones de diseño en programación orientada a objetos.* [En línea] [Citado el: 17 de Enero de 2013.] http://www.oodesign.com/.

32. Glew. *Sitio web oficial de la librería GLEW.* [En línea] [Citado el: 17 de Enero de 2013.] http://glew.sourceforge.net/.

33. *Artículo online sobre las transformaciones utilizadas en el proceso de renderizado en OpenGL - OpenGL Transformation.* [En línea] http://www.songho.ca/opengl/gl_transform.html.

34. University of Maryland - Department of Computer Science - CMSC 741: Geometric and Solid Modeling (Fall 2010) - TRI Datasets. [En línea] 2012. [Citado el: 18 de 1 de 2013.] torus.tri http://www.cs.umd.edu/class/fall2010/cmsc741/datasets/tri/.

35. AIM@SHAPE Shape Repository v4.0. *Group: Cow_MC, Group ID: 196, Model name: Cow_MC-lr, ID: 223.* [En línea] [Citado el: 18 de 1 de 2013.] http://shapes.aimatshape.net/view.php?id=223.

36. Stanford Computer Graphics Laboratory. *The Stanford 3D Scanning Repository.* [En línea] [Citado el: 18 de 1 de 2013.] http://www-graphics.stanford.edu/data/3Dscanrep/.

37. AIM@SHAPE Shape Repository v4.0. *Group: Laurent hand, Group ID: 785, Model name: Laurent's hand - Raw mesh after merging, ID: 788.* [En línea] [Citado el: 18 de 1 de 2013.] http://shapes.aimatshape.net/view.php?id=788.

38. AIM@SHAPE Shape Repository v4.0. *Group: Ramesses, Group ID: 814 ,Model name: Ramesses - clean, watertight, ID: 814.* [En línea] [Citado el: 18 de 1 de 2013.] http://shapes.aimatshape.net/view.php?id=814.

39. AIM@SHAPE Shape Repository v4.0. *Group: Neptune, Group ID: 803, Model name: Neptune - clean, watertight (4M triangles), ID: 803.* [En línea] [Citado el: 18 de 1 de 2013.] http://shapes.aimatshape.net/view.php?id=803.

40. *Sitio web con la descripción del formato OFF de Geomview.* [En línea] [Citado el: 17 de Enero de 2013.] http://people.sc.fsu.edu/~jburkardt/data/off/off.html.

41. Neon Storm. *Artículo online, The area of a simple polygon.* [En línea] [Citado el: 17 de Enero de 2013.] http://neonstorm242.blogspot.com/2010/12/area-of-simple-polygon.html.

42. **Franklin, Wm. Randolph.** Short Notes - Volume of a Polyhedron. [En línea] http://www.ecse.rpi.edu/~wrf/Research/Short_Notes/volume.html.

43. Wolfram. *Artículo de Wolfram sobre pirámides. Contiene información de cómo calcular el volumen de este tipo de poliedros.* [En línea] [Citado el: 17 de 01 de 2013.] http://mathworld.wolfram.com/Pyramid.html.

44. Wolfram. *Artículo en Wolfram sobre tetraedros. Contiene información de cómo obtener el volumen de un tetraedro.* [En línea] [Citado el: 17 de 1 de 2013.] http://mathworld.wolfram.com/Tetrahedron.html.

45. **McKesson, Jason L.** Arcsynthesis. *Online book: Learning Modern 3D Graphics Programming, Rotation.* [En línea] 2012. [Citado el: 17 de 1 de 2013.] http://www.arcsynthesis.org/gltut/Positioning/Tut06%20Rotation.html.

46. Michigan Technological University – Department of Computer Science - CS3621 Notes. [En línea] [Citado el: 17 de 1 de 2013.] http://www.cs.mtu.edu/~shene/COURSES/cs3621/NOTES/geometry/geo-tran.html.

47. **McKesson, Jason L.** Arcsynthesis. *Online book: Learning Modern 3D Graphics Programming, Translation.* [En línea] 2012. [Citado el: 17 de 1 de 2013.] http://www.arcsynthesis.org/gltut/Positioning/Tut06%20Translation.html.

48. **McKesson, Jason L.** Arcsynthesis. *Online Book: Learning Modern 3D Graphics Programming, Scale.* [En línea] 2012. [Citado el: 17 de 1 de 2013.] http://www.arcsynthesis.org/gltut/Positioning/Tut06%20Scale.html.

49. **Ahn, Song Ho.** Songho. *Artículo online sobre la proyección en perspectiva y ortogonal en OpenGL, OpenGL Projection Matrix.* [En línea] http://www.songho.ca/opengl/gl_projectionmatrix.html.

Anexo A. Formatos de almacenamiento de mallas geométricas

Para almacenar mallas geométricas, existen diversos formatos. Los distintos formatos varían en:

- El tipo de malla geométrica y de elementos que son capaces de almacenar.
- Los tipos de información asociada a cada elemento que son capaz de almacenar. Por ejemplo, hay formatos que soportan sólo posiciones para los vértices, mientras que otros pueden contener más atributos asociados a los vértices.
- La forma en que la información esta ordenada dentro del archivo.
- Si los archivos se guardan en formato binario o en formato de texto.

En las siguientes secciones se describirán los tipos de archivos que el visualizador desarrollado debe ser capaz de cargar.

A.1. OFF

El formato OFF (Object file format) representa una colección de polígonos planares, posiblemente con vértices compartidos, la cual es una manera conveniente de describir poliedros. Los polígonos pueden ser cóncavos, pero no permite describir polígonos con hoyos. [40]

Además, como información adicional, permite especificar colores para los vértices o caras, en punto flotante ASCII, en formato RGBA. Para el visualizador, los colores que vengan en los archivos u otros atributos extras, no serán cargados.

Los archivos OFF, están en formato de texto, son un formato de mallas popular.

A.2. Ele node

Este formato está compuesto por dos archivos, uno con extensión .ele y el otro .node.

El archivo de extensión .node, es el encargado de almacenar la información de los vértices de la malla. Contiene una lista de puntos en 3D. Cada punto tiene 3 coordenadas (x,y,z), uno o más atributos y una región.

Para el visualizador, no está contemplado que cargue información adicional a parte de las posiciones, por lo tanto, el resto de los atributos y la región serán ignorados.

Por otro lado, el archivo de extensión .ele, es el encargado de almacenar la información relacionada con los polígonos o poliedros, dependiendo del tipo de malla.

A.3. TRI

TRI es un formato para almacenar mallas de superficie que están compuestas solamente de triángulos. Es similar al formato OFF, sólo que los polígonos son siempre de 3 vértices.

A.4. M3D

Este formato sirve para guardar mallas de poliedros mixtas, descrita mediante una lista de vértices de poliedros. Los archivos en que se guarda este formato son de tipo ASCII.

El archivo comienza con el número que indica la cantidad de vértices de la malla. A continuación, por cada uno existe una línea con su descripción. Cada una de estas líneas comienza con un número que indica si la dirección de proyección del vértice está dada o no. Después, le siguen las coordenadas y, dependiendo del caso, termina con o sin las coordenadas de la dirección de la proyección. Los vértices se enumeran implícitamente desde 0.

Posteriormente, viene un número que indica la cantidad de poliedros de las malla. Cada línea con un poliedro tiene una letra al comienzo que representa su tipo y una lista de vértices que lo conforman.

Los archivos tienen la siguiente estructura:

```
<# de vértices>
<tipo de vértice> <x> <y> <z> <proy x> <proy y> <proy z>
...
<# de poliedros>
<tipo poliedro> <vértice 1>...<vértice #>
...
```

Donde <tipo poliedro> puede tener cualquiera de los siguientes valores:

- H = hexaedro
- T = tetraedro
- P = pirámide
- R = prisma

Anexo B. Cálculo del área de un polígono

Para calcular el área de un polígono simple, podemos generar un triángulo por cada arista del polígono. Cada triángulo estará formado por los dos vértices de la arista y un tercer vértice, el tercer vértice será arbitrario y común para todos los triángulos. Es importante que las aristas estén todas en el mismo sentido (por ejemplo, CCW), luego, las áreas saldrán negativas o positivas dependiendo de esto. Para obtener el área total del polígono, se suman todas las áreas de los triángulos (negativas y positivas)[21]. [41]

Por ejemplo, sea el polígono con aristas f1, f2, f3, f5, f6, f7, f8. Asumiendo que los vértices del polígono están en CCW. Por cada arista, formamos un triángulo contra un vértice en común arbitrario.

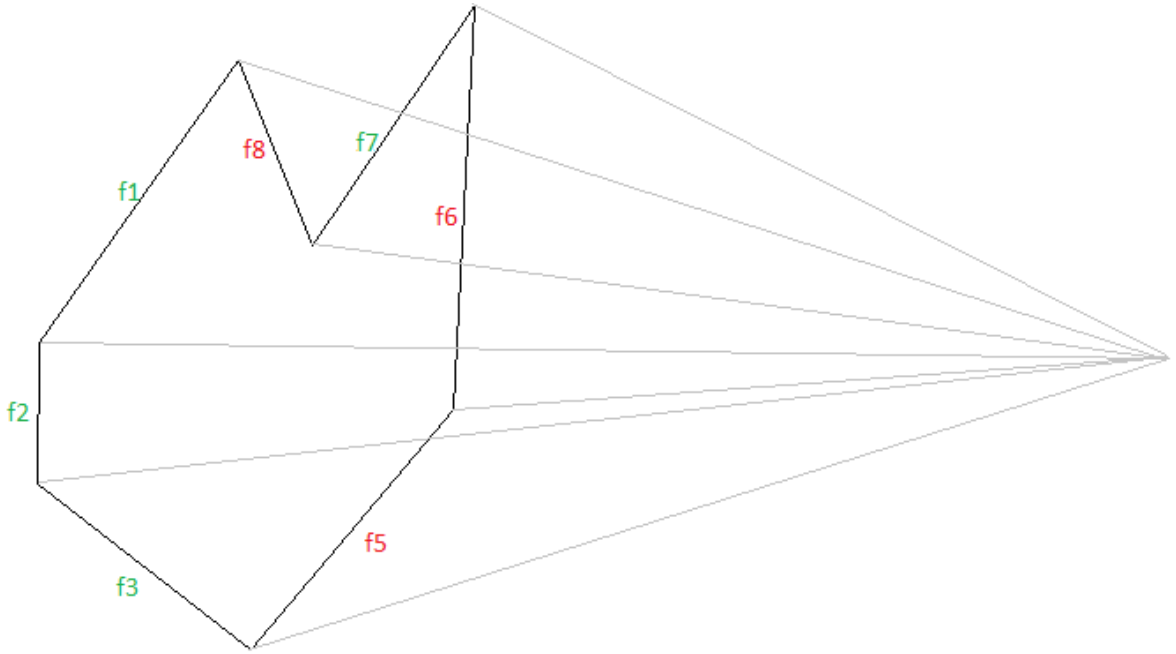
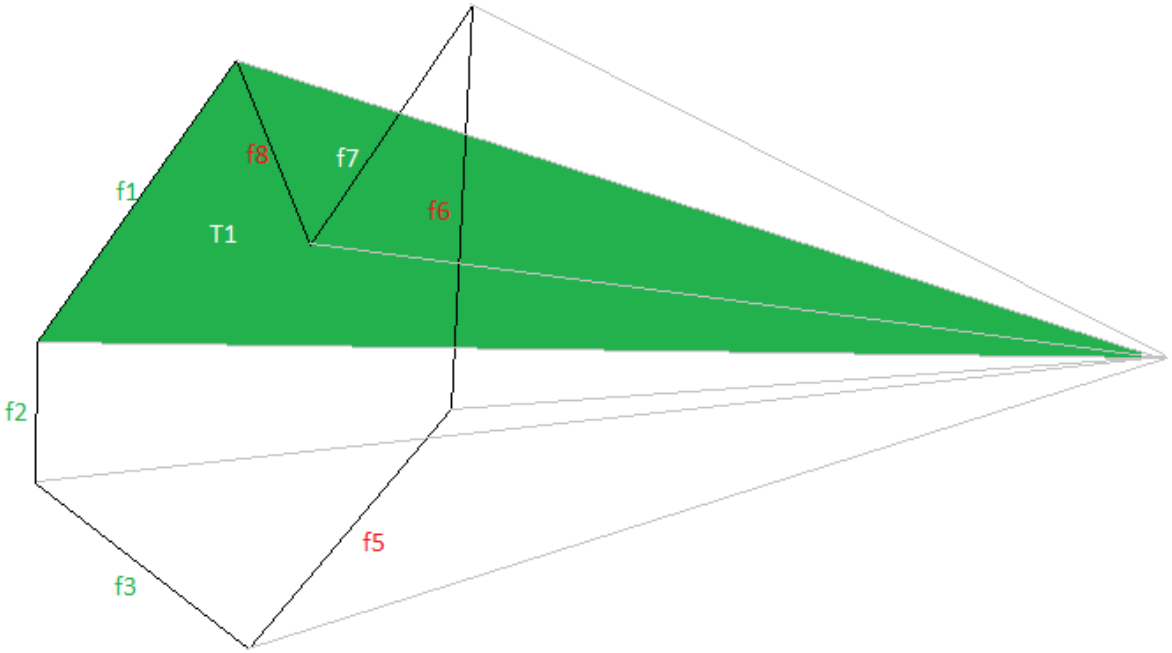
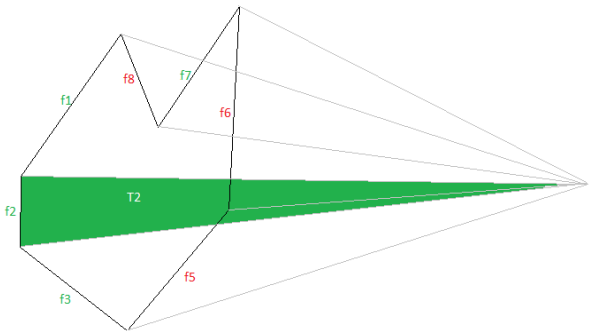


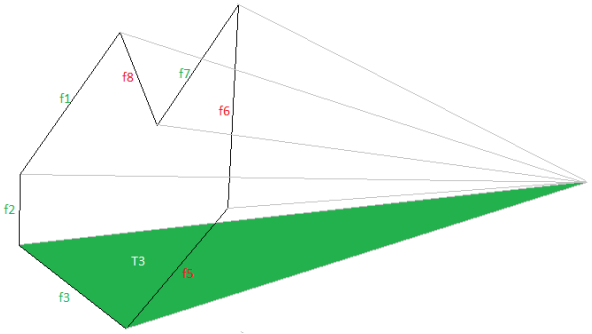
Figura 58 – Área polígono, triángulos obtenidos por arista.

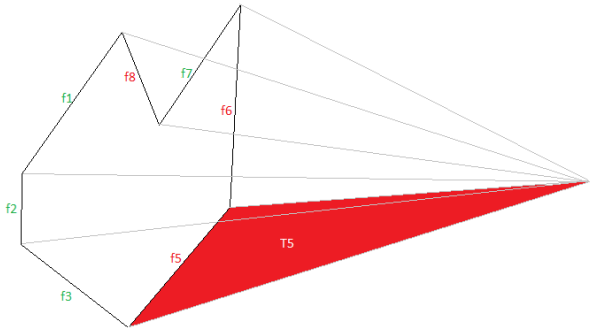
El triángulo T1, corresponde a la arista f1. Las áreas de un triángulo las calculamos utilizando producto cruz. Dado que las áreas son calculadas utilizando producto cruz, estas darán positivas o negativas dependiendo del orden de los vértices. Con verde marcamos los triángulos de área positiva y con rojo los de área negativa.

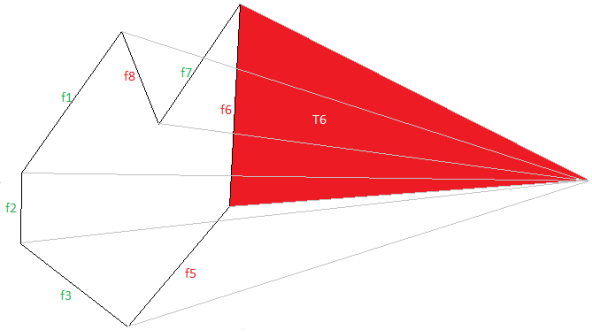
[21] En el anexo hay un ejemplo visual.


f1
f8
f7
f6
T1
f2
f5
f3


f1
f8
f7
f6
T2
f2
f5
f3


f1
f8
f7
f6
T3
f2
f5
f3


f1
f8
f7
f6
T5
f2
f5
f3


f1
f8
f7
f6
T6
f2
f5
f3

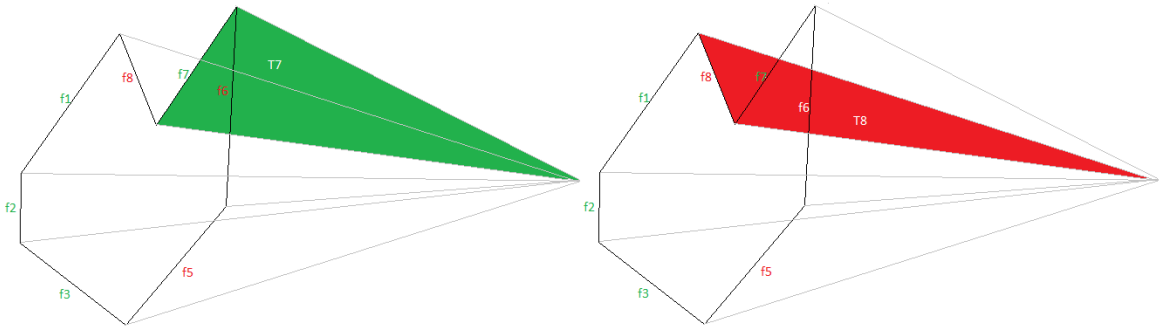


Figura 59 - Área polígono, triángulos obtenidos por arista.

A continuación, si marcamos las distintas áreas que se forman:

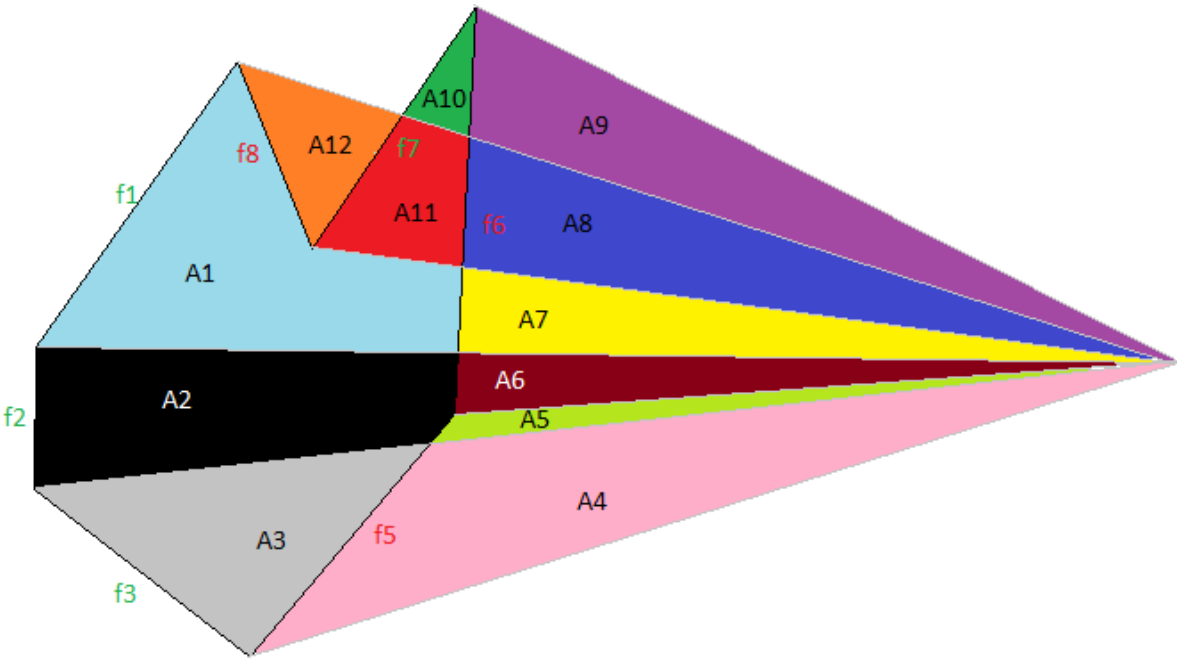


Figura 60 - Área polígono, sub-áreas encontradas.

Podemos expresar el área de los triángulos de la siguiente forma:

T1 = A1 + A7+ A8 + A11 + A12

T2 = A2 + A5 + A6

T3 = A3 + A4

T5 = -(A4 + A5)

T6 = -(A6 + A7 + A8 + A9)

T7 = A8 + A9 + A10 + A11

T8 = -(A8 + A11 + A12)

Entonces,

T1 + T2 + T3 + T5 + T6 + T7 + T8 = A1 + A7+ A8 + A11 + A12 + A2 + A5 + A6 + A3 + A4 – A4 – A5 – A6 – A7 – A8 – A9 + A8 + A9 + A10 + A11 – A8 – A11 - A12

= A1 + A2 + A3 + A4 – A4 +A5 –A5 + A6 – A6 + A7 – A7 + A8 – A8 + A8 – A8 – A9 + A9 + A10 + A11 + A11 – A11 + A12 – A12

= A1 + A2 + A3 + A10 + A11 = Área polígono completo.

Con esto, vemos que efectivamente la suma de las áreas de los triángulos da como resultado el área del polígono inicial.

Luego, el problema se reduce a calcular el área de triángulos. El área de un triángulo se puede calcular en tiempo constante de diversas formas, para el trabajo se considerará implementar alguna de estas:

Formula de Herón

Considerando un triángulo cuyos lados tienen largos **a**, **b** y **c**. El semi-perímetro **S** de dicho triángulo está dado por la siguiente formula:

$$S = \frac{a + b + c}{2}$$

La fórmula de Herón nos permite calcular el área del triángulo de la siguiente forma:

$$Area = \sqrt{S \cdot (S - a) \cdot (S - b) \cdot (S - c)}$$

Formula vectorial

Sea un triángulo en un espacio 3D, donde las coordenadas de sus 3 están dadas por los vectores A, B y C. El área del triángulo se puede calcular con la siguiente ecuación:

$$Area = \frac{\|AB \times AC\|}{2}$$

En el visualizador, para los triángulos, se utilizará la fórmula de Herón, ya que es más eficiente.

Anexo C. Cálculo del área de un poliedro

Para el cálculo del volumen de un poliedro genérico, el problema es análogo a calcular el área en un polígono simple, pero en tres dimensiones. Podemos generar pirámides virtuales y calcular los volúmenes de estas, luego, sumamos estos volúmenes. Cada pirámide virtual generada, tiene como su base una cara del poliedro, y todas tienen como cúspide un vértice común y arbitrario. [42]

El volumen de una pirámide cuya base es un polígono simple, está dado por la siguiente integral:

$$\frac{b}{h^2}\int_0^h (h-y)^2 dy = \frac{-b}{3h^2}(h-y)^3\Big|_0^h = \frac{1}{3}bh$$

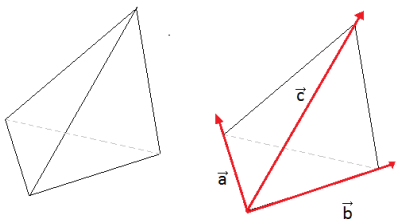
La integral se reduce a una fórmula simple, la cual depende de la base y la altura de la pirámide. La variable **h** es la altura de la pirámide, y será negativa o positiva dependiendo de si el vértice cúspide está a la izquierda o derecha del plano que contiene la base. La altura está definida por la distancia perpendicular entre el vértice de la cúspide y el plano en el que se encuentra la base. **b** es el área de la base de la pirámide, la cual se puede calcular con el método mencionado en el ítem anterior. [43]

En el caso de que las caras del poliedro estén trianguladas, las pirámides serán tetraedros. Un tetraedro es una pirámide de 4 caras y 4 vértices. Para el cálculo del volumen de un tetraedro, podemos hacerlo eficientemente con las siguientes fórmulas:

$$V = \frac{1}{3!}\begin{vmatrix} x_1 & y_1 & z_1 & 1 \\ x_2 & y_2 & z_2 & 1 \\ x_3 & y_3 & z_3 & 1 \\ x_4 & y_4 & z_4 & 1 \end{vmatrix}$$

Donde (x_i, y_i, z_i) corresponde a las coordenadas del i-esimo vértice del tetraedro. [44]

Otra forma de calcular el volumen de un tetraedro es utilizando los vectores que lo definen. Entonces, si conocemos los vectores **a**, **b** y **c** que definen el tetraedro con respecto a un vértice común:



podemos utilizar la siguiente formula:

$$V = \frac{|a \cdot (b \times c)|}{3!}$$

Anexo D. Rendering pipeline resumido de OpenGL

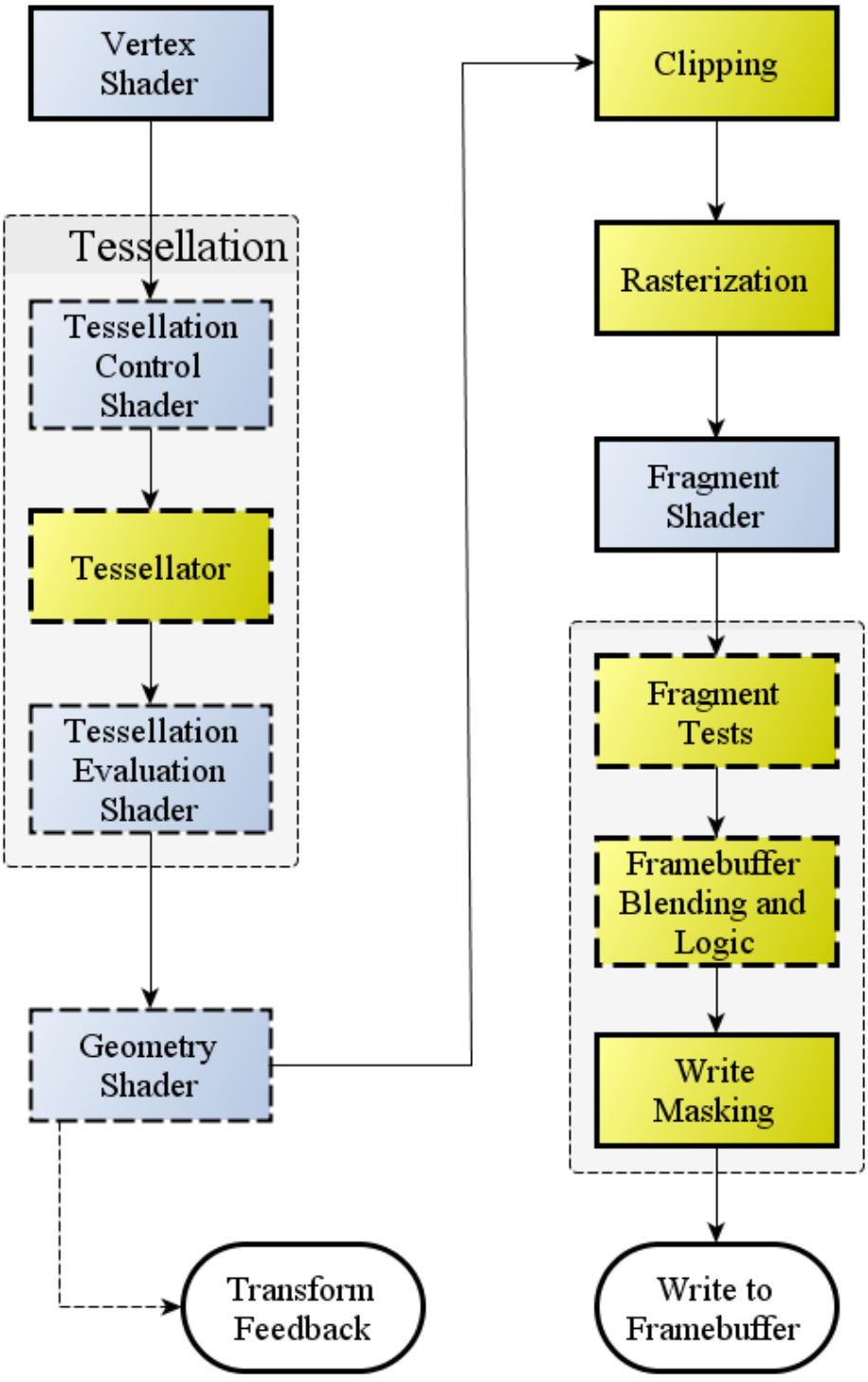


Figura 61 – Rendering Pipeline de OpenGL resumido. **[10]**

Anexo E. Clases auxiliares para manejo de shaders

Como la aplicación requiere compilar y utilizar programas de shaders en numerosas partes, se creó la clase auxiliar *ShaderUtils*, la cual tiene una serie de funciones que encapsulan las llamadas a OpenGL y proveen al programador de una interfaz en C++ fácil de utilizar.

Antes de revisar la clase, se necesitan revisar dos estructuras que son utilizadas como argumentos de varias funciones:

```
struct ShaderLoadingData{
        public:
                ShaderLoadingData(GLenum t){eShaderType = t;}
                void addFile(std::string t){strShaderFiles.push_back(t);}
                GLenum eShaderType;
                std::vector<std::string> strShaderFiles;
};
```

ShaderLoadingData es una estructura que contiene los datos necesarios para compilar una etapa de un programa de shaders. Está diseñada para guardar un vector de *std::string* con los nombres de los archivos que componen el shader y un *GLenum* que guarda el tipo de shader al que corresponde. Guarda lista de archivos porque para ahorrar código, existen funciones comunes a varios shaders que están guardados en archivos apartes y estos deben ser concatenados para que compilen correctamente.

```
struct VertexAttributeBindingData{
        public:
                GLuint index;
                const char* strShaderFile;
};
```

VertexAttributeBindingData guarda los datos necesarios para unir los atributos de los shader por su nombre a un índice de tipo GLuint.

Descritas estas estructuras, ahora se revisará el código de la clase *ShaderUtils:*

```
class ShaderUtils
{
public:
        static const GLuint FAIL_CREATING_SHADER = 99999;
        static const GLuint FAIL_CREATING_PROGRAM = 99998;
        static GLuint CreateProgram(const std::vector<GLuint> &shaderList,
                const std::vector<VertexAttributeBindingData> &attributes,
                const std::vector<VertexAttributeBindingData> &fragData);
        static GLuint CreateProgram(
                const std::vector<ShaderLoadingData> &shaderList);
        static GLuint CreateProgram(
                const std::vector<ShaderLoadingData> &shaderList,
                const std::vector<VertexAttributeBindingData> &attributes);
        static GLuint CreateProgram(
                const std::vector<ShaderLoadingData> &shaderList,
                const std::vector<VertexAttributeBindingData> &attributes,
                const std::vector<VertexAttributeBindingData> &fragAt);
        static GLuint CreateShader(ShaderLoadingData data);
```

```
        //set uniforms
        static bool setUniform(   GLuint program,const char* uniformName,
                                  glm::vec4 val);
        static bool setUniform(   GLuint program,const char* uniformName,
                                  glm::mat4 val);
        static bool setUniform(   GLuint program,const char* uniformName,
                                  glm::mat3 val);
        static bool setUniform(   GLuint program,const char* uniformName,
                                  glm::vec3 val);
        static bool setUniform(   GLuint program,const char* uniformName,
                                  glm::vec2 val);
        static bool setUniform(GLuint program,const char* uniformName, int val);
        static bool setUniform(GLuint program,const char* uniformName,float val);
        static bool setUniform(GLuint program,const char* uniformName, bool val);

        //buffers
        static void deleteBuffer(GLuint*);
        template <class myType>
        static GLuint createDataBuffer(std::vector<myType>&);
        template <class myType>
        static GLuint createDataBuffer(myType*,int n);
        template <class myType>
        static bool setDataBuffer(GLuint bufferHandle,std::vector<myType>&);
private:
        ShaderUtils();
};
```

Como se puede ver, esta clase tiene el constructor declarado como privado, esto quiere decir que la clase no se puede instanciar y es lógico considerando que todas sus funciones son estáticas. Todas las funciones son estáticas por qué no necesitan de ningún estado almacenado, solo encapsulan una serie de llamados a funciones de OpenGL en una única función.

La segunda ventaja importante de utilizar esta clase es que sus funciones están sobrecargadas[22], esto es posible gracias a C++, haciendo que el programador tenga que aprenderse menos nombres de funciones. Por ejemplo, si se utilizaran directamente las funciones de OpenGL para asignar los valores de las variables uniform en los shaders, el programador para una variable de tipo mat4 tendría que usar la función glUniformMatrix4fv o para una de tipo entero debería utilizar la función glUniform1i. Este problema de OpenGL es producto de que la API utilizada es para C, y este lenguaje no permite sobrecargar funciones.

La función *setUniform(…)*, está altamente sobre cargada y provee al programador con un nombre único de función para asignar valores a variables uniform de cualquier tipo a través del nombre de la variable. Además, este mecanismo requiere por lo menos de 3 líneas de código para implementarse si se usa la API de OpenGL directamente.

Luego, tenemos la función sobrecargada *CreateProgram(…)*. Esta función permite de distintas maneras compilar un programa de shaders y obtener el índice que lo representa. La

[22] Se dice que una función esta sobrecargada cuando existe más de una función con el mismo nombre, pero con distintos argumentos. C++ permite sobrecargar funciones, pero C no.

función se encarga, a partir de sus argumentos, de compilar los shaders, enlazarlos y unir los atributos de los vértices a índices de tipo *GLuint*. Si el proceso de compilación o enlazado del programa de shaders falla, el error será impreso en la salida de errores y la función retornará el valor *ShaderUtils::FAIL_CREATING_PROGRAM.* Esta función simplifica mucho el proceso, ya que se encarga de leer los archivos (y concatenarlos si fuera el caso), compilar los shaders individualmente, enlazarlos, unir los atributos de los vértices a un identificador numérico, y en caso de haber errores, extraer e imprimir el mensaje.

El grupo de funciones *deleteBuffer(...)*, *createDataBuffer(...)*, *setDataBuffer(...)* son utilizadas con menos frecuencia en la aplicaciones, pero de igual forma que las anteriores, simplifican el código que debe utilizar el programador para implementar ciertas lógicas que involucran OpenGL. Estas funciones se utilizan para manipular los buffers de datos que están guardados en la memoria de la tarjeta de video, borrándolos, creándolos y enviando datos o solo enviando datos a uno existente.

Anexo F. *MainRenderer* Geometry Shader

```
#version 400
layout( triangles ) in;
layout( triangle_strip, max_vertices = 3 ) out;
in vec4 Color[];
in uint VFlags[];
in vec4 VClipPosition[];
noperspective out vec3 GEdgeDistance;
out vec3 IgnoreEdges;
out vec4 ColorToFrag;
uniform mat4 ViewportMatrix;
uniform int ElementDrawOption;
struct LineInfo {
    int Width;
    vec4 Color;
    int isDrawn; //1 = wire, 0 = no wire
};
uniform LineInfo Line;
bool primitiveIsDrawn();
bool isFlagEnabled(uint f);
vec3 getVerticesDistanceFromOppositeEdge(vec4 Vertex1ClipPositions,
                                         vec4 Vertex2ClipPositions,
                                         vec4 Vertex3ClipPositions,
                                         uint Vertex1Flag,
                                         uint Vertex2Flag,
                                         uint Vertex3Flag,
                                         mat4 viewPortMatrix);
vec3 getWireFrameIgnoredEdges(uint Vertex1Flag,
                              uint Vertex2Flag,
                              uint Vertex3Flag);
```

```
void main()
{
    // Transform each vertex into viewport space
    if(primitiveIsDrawn()){
        vec3 heights = vec3(0.0,0.0,0.0);
        if(Line.isDrawn==1){
            heights = getVerticesDistanceFromOppositeEdge(VClipPosition[0],
                                                          VClipPosition[1],
                                                          VClipPosition[2],
                                                          VFlags[0],
                                                          VFlags[1],
                                                          VFlags[2],
                                                          ViewportMatrix);
            IgnoreEdges = getWireFrameIgnoredEdges(VFlags[0],
                                                   VFlags[1],
                                                   VFlags[2]);
        }
        GEdgeDistance = vec3( heights.x, 0, 0 );
        gl_Position = VClipPosition[0];
        ColorToFrag = Color[0];
        EmitVertex();
        GEdgeDistance = vec3( 0, heights.y, 0 );
        gl_Position = VClipPosition[1];
        ColorToFrag = Color[1];
        EmitVertex();
        GEdgeDistance = vec3( 0, 0, heights.z );
        gl_Position = VClipPosition[2];
        ColorToFrag = Color[2];
        EmitVertex();
        EndPrimitive();
    }
}
bool primitiveIsDrawn(){
    if(ElementDrawOption == 1 && !isFlagEnabled(8u))
        return false;
    else if(ElementDrawOption == 2 && isFlagEnabled(8u))
        return false;
    else if(ElementDrawOption == 3 && !isFlagEnabled(1u))
        return false;
    else if(ElementDrawOption == 4 && isFlagEnabled(1u))
        return false;
    else
        return true;
}
bool isFlagEnabled(uint f){
    return (VFlags[0]&f)==f && (VFlags[1]&f)==f && (VFlags[2]&f)==f;
}
vec3 getVerticesDistanceFromOppositeEdge(vec4 Vertex1ClipPositions,
                                         vec4 Vertex2ClipPositions,
                                         vec4 Vertex3ClipPositions,
                                         uint Vertex1Flag,
                                         uint Vertex2Flag,
                                         uint Vertex3Flag,
                                         mat4 viewPortMatrix){
    vec3 p0 = vec3(viewPortMatrix * (Vertex1ClipPositions /
```

```
                                        Vertex1ClipPositions.w));
    vec3 p1 = vec3(viewPortMatrix * (Vertex2ClipPositions /
                                        Vertex2ClipPositions.w));
    vec3 p2 = vec3(viewPortMatrix * (Vertex3ClipPositions /
                                        Vertex3ClipPositions.w));
    // Find the altitudes (ha, hb and hc)
    p0.z = 0.0f;
    p1.z = 0.0f;
    p2.z = 0.0f;
    float a = length(p1 - p2);
    float b = length(p2 - p0);
    float c = length(p0 - p1);
    float area = length(cross(p0-p1, p0-p2))/2.0f;
    return vec3(area*2/a,area*2/b,area*2/c);
}
vec3 getWireFrameIgnoredEdges(uint Vertex1Flag,
                              uint Vertex2Flag,
                              uint Vertex3Flag){
    vec3 IgnoreEdges = vec3(0.0f,0.0f,0.0f);
    if((VFlags[0]&2u)==2u)
        IgnoreEdges.x = 1.0f;
    if((VFlags[1]&2u)==2u)
        IgnoreEdges.y = 1.0f;
    if((VFlags[2]&2u)==2u)
        IgnoreEdges.z = 1.0f;
    return IgnoreEdges;
}
```

Anexo G. Transformaciones geométricas en detalle

Asumiendo que las coordenadas se trabajarán en coordenadas homogéneas, las transformaciones geométricas básicas que se utilizarán en la aplicación serán las siguientes:

G.1. Rotación

La rotación es una transformación isométrica[23]. Es una operación donde un punto rota en torno a un eje, con un ángulo fijo. En nuestro caso, las rotaciones que nos interesan son las que se hacen en torno a los ejes **X**, **Y**, **Z**. [45]

Las matrices que representan esta transformación en el espacio homogéneo son:

[23] Una transformación isométrica mantiene la forma del objeto. Las distancias relativas entre puntos se mantienen.

Dado un ángulo de rotación θ,

Rotación en torno a X

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Rotación en torno a Y

$$\begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Rotación en torno a Z

$$\begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

También, se puede ver como un cambio de sistema de coordenadas, donde el nuevo sistema tiene los vectores base rotados, pero mantienen sus posiciones relativas entre ellos.

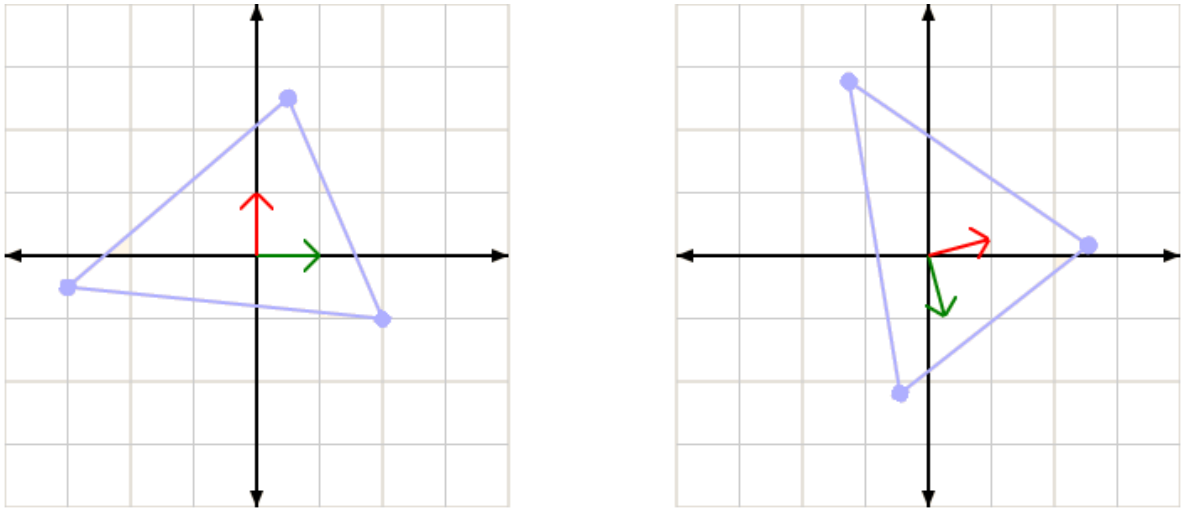


Figura 62 - Rotación 2D. **[45]**

G.2. Traslación

Descrita en R^n, una traslación es una isometría en el espacio euclidiano caracterizada por un vector $\vec{\mu}$, tal que, a cada punto P de un objeto o figura se le hace corresponder otro punto P' tal que:

$$\begin{cases} T_{\vec{\mu}}: R^n \to R^n, & \overrightarrow{PP'} = \vec{\mu} \\ P \to P' = T(P) = P + \vec{\mu} \end{cases}$$

La traslación no cambia la orientación de una figura, mantiene la forma y el tamaño del elemento trasladado.]

La matriz que representa la traslación en coordenadas homogéneas de 4 dimensiones, es la siguiente:

$$T_v = \begin{bmatrix} 1 & 0 & 0 & v_x \\ 0 & 1 & 0 & v_y \\ 0 & 0 & 1 & v_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Donde la traslación esta descrita por un vector **v = (v_x, v_y, v_z, 0)**.

Luego, la operación completa sobre un punto **p = (px, py, pz, 1)** en coordenadas homogéneas será la siguiente:

$$T_v \cdot p = \begin{bmatrix} 1 & 0 & 0 & v_x \\ 0 & 1 & 0 & v_y \\ 0 & 0 & 1 & v_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} p_x \\ p_y \\ p_z \\ 1 \end{bmatrix} = \begin{bmatrix} p_x + v_x \\ p_y + v_y \\ p_z + v_z \\ 1 \end{bmatrix} = p + v$$

Aquí se puede ver que la multiplicación por la matriz de transformación de el mismo resultado que simplemente sumar los vectores. [46]

Visto como un cambio de sistema de coordenadas, el nuevo sistema tiene los vectores base sin cambios, pero el origen se traslada.

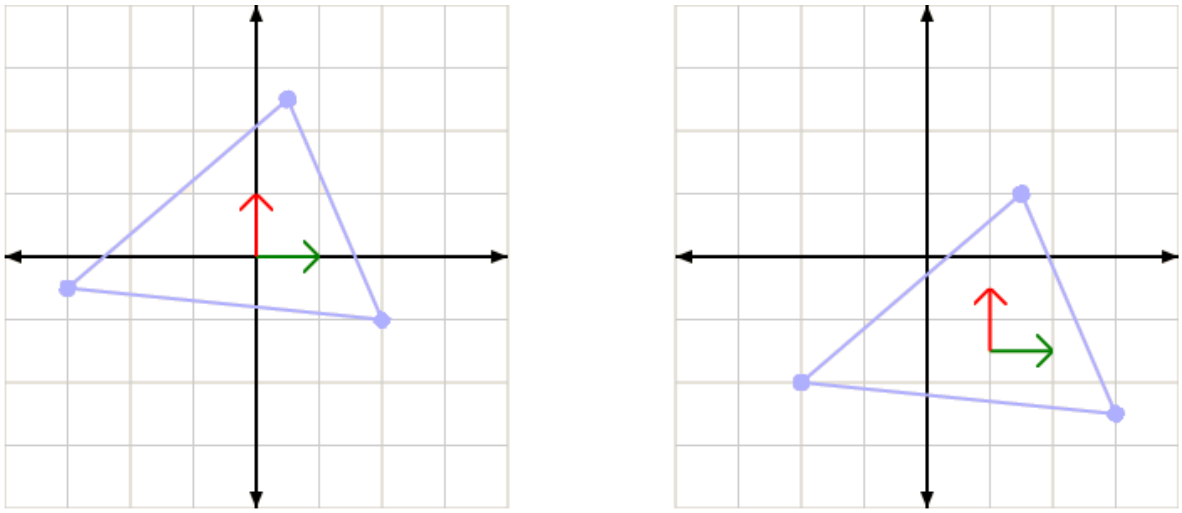


Figura 63 - Traslación 2D. **[47]**

G.3. Escalado

Escalar no es una transformación isométrica. Es una operación donde los puntos se acercan hacia el origen en una proporción dada, manteniendo la misma dirección. En nuestro caso, sólo se utilizará escalamiento con números mayores a 0, ya que con números negativos, los sentidos de los ejes se invierten y no interesa este efecto para la aplicación, ni la complejidad extra de manejarlo.

Luego, las matrices que representan esta transformación en el espacio homogéneo son:

$$S(x,y,z) = \begin{bmatrix} x & 0 & 0 & 0 \\ 0 & y & 0 & 0 \\ 0 & 0 & z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Donde las variables (x, y, z) representan el factor de escalamiento en cada dimensión. [48]

Esta transformación, vista como cambio de sistema de coordenadas, consiste en escalar los vectores base del sistema original, sin rotarlos ni desplazar el origen:

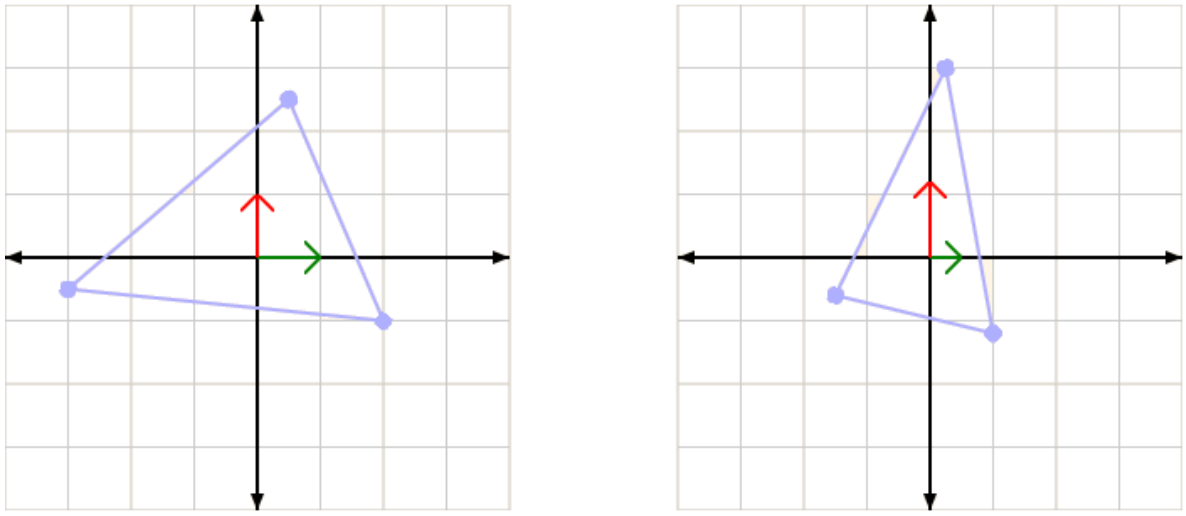


Figura 64 - Escalado 2D. **[48]**

G.4 Proyecciones

Una proyección, en el contexto del proceso de renderizado, es una forma de transformar los elementos en alguna dimensión a otra dimensión distinta. Para esta aplicación, se trabaja con coordenadas 3D, luego, la proyección de los elementos en 3D seria llevarlos a 2D.

En el visualizador, se trabajó con dos tipos de proyecciones para generar una imagen en 2D a partir de los elementos en 3D, las cuales se presentarán en las secciones siguientes.

G.4.1. Proyección Ortogonal

En una proyección ortogonal, todas las líneas de proyección son ortogonales al plano de proyección (3D a 2D). En OpenGL, esta proyección es bastante simple de construir. Todas las componentes X, Y, Z en el espacio de la cámara son escaladas de forma lineal a un espacio cúbico llamado NDC y trasladadas al origen.

NDC (Normalized Device Coordinates) es un espacio definido por OpenGL, cuyos límites son -1 y 1 para los 3 ejes.

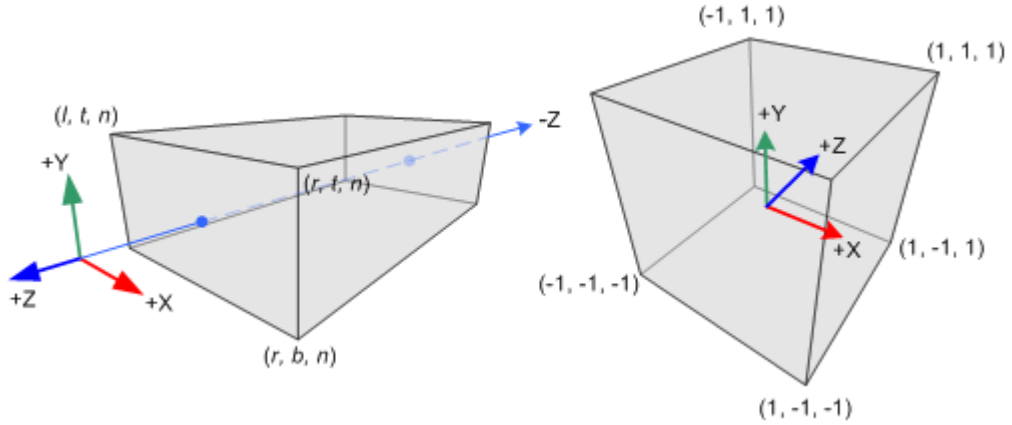


Figura 65 - Proyección Ortogonal. **[49]**

La componente w no es necesaria en las proyecciones ortográficas, es por esto que la última fila es (0, 0, 0, 1), y al multiplicarla por un vector de coordenadas homogéneas, la coordenada homogénea se mantiene.

La matriz que representa esta transformación queda definida de la siguiente forma:

$$Matriz\ de\ proyección\ ortogonal = \begin{bmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\ 0 & 0 & \frac{-2}{f-n} & -\frac{f+n}{f-n} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Donde los puntos extremos (r, b, n) y (l, t, f) definen el paralelepípedo en 3D que representa el espacio de coordenados que es capaz de ver la cámara. Los valores r, l, b, t, n, y f se refieren a derecha, izquierda, abajo, arriba, cerca y lejos respectivamente. [49]

G.4.2. Proyección en Perspectiva

En las proyecciones en perspectiva, el espacio de la cámara está definido por una pirámide truncada llamada Frustum. La pirámide esta truncada por el plano donde se va a realizar la proyección y la cámara, ubicada tras el plano, quedaría en lo que sería la cúspide de la pirámide.

Las coordenadas de los puntos que se encuentra en el espacio del Frustum son transformadas al espacio NDC.

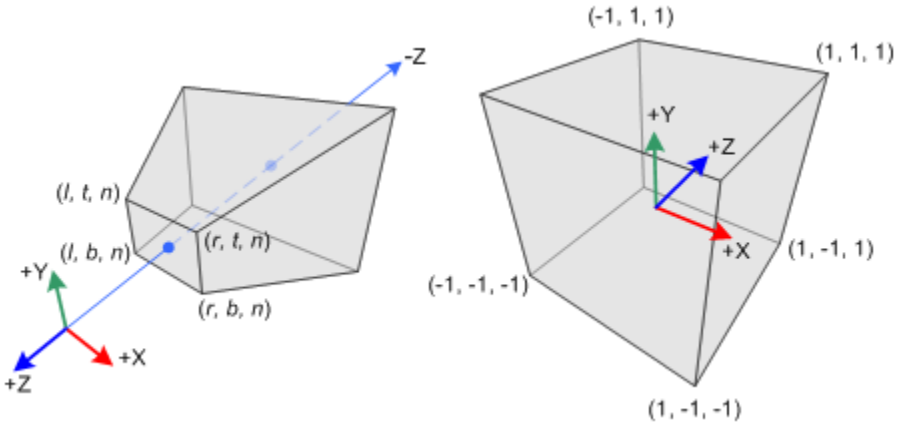


Figura 66 – Frustum (Pirámide truncada) de perspectiva. **[49]**

En OpenGL, las coordenadas de los puntos en el espacio de la cámara son proyectadas sobre el plano *cercano* (definido por n). El siguiente diagrama muestra como un punto (x_e, y_e, z_e) en el espacio de la cámara, es proyectado al punto (x_p, y_p, z_p) en el plano cercano.

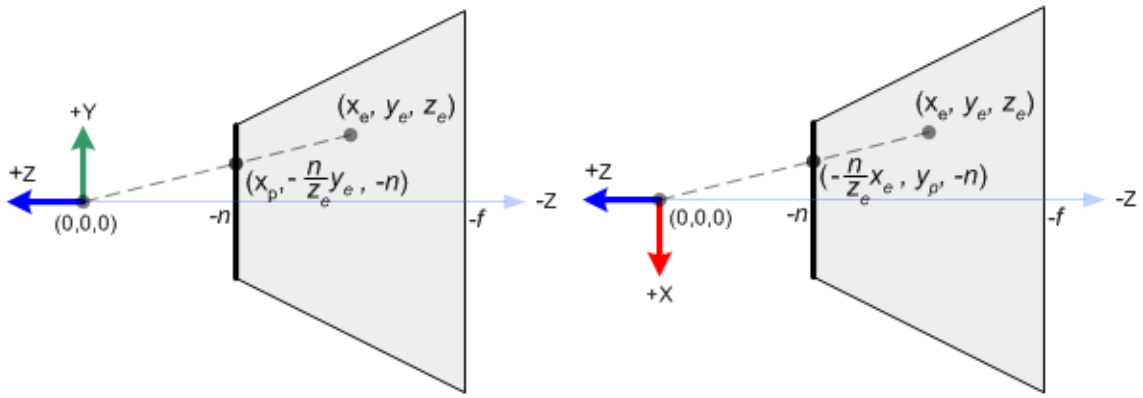


Figura 67 - Frustum de perspectiva: La figura izquierda nos muestra un corte lateral de la pirámide truncada de proyección. La figura derecha nos muestra un corte superior de la misma pirámide. [49]

Para el contexto del visualizador, la *cámara* apunta a lo largo del eje z. Luego, para la matriz de perspectiva, basta con tener distancia al plano ortogonal más cercano y al más lejano (n y f respectivamente) y dos ángulos que representan la apertura del Frustum. La matriz queda definida por:

$$Matriz\ de\ proyección\ en\ perspectiva$$

$$= \begin{bmatrix} Frustum\ Angle_{Horizontal} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & Frustum\ Angle_{Vertical} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & -\frac{f+n}{f-n} & -\frac{2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

Dado el ángulo horizontal, calculamos el ángulo vertical, manteniendo la proporción con las dos dimensiones del widget para desplegar la imagen renderizada del modelo. Luego, la escala con respecto al ángulo se calcula como 1/tan(ángulo/2.0), donde el ángulo está en radianes. [49]