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

VISUALIZADOR Y EVALUADOR DE MALLAS GEOMÉTRICAS SOBRE UNA PLATAFORMA WEB

MEMORIA PARA OPTAR AL TÍTULO DE INGENIERO CIVIL EN COMPUTACIÓN

DAVID MIGUEL MUÑOZ LEÓN

PROFESOR GUÍA:
NANCY HITSCHFELD KAHLER

MIEMBROS DE LA COMISIÓN:
BENJAMIN BUSTOS CÁRDENAS
JUAN MANUEL BARRIOS

Este trabajo ha sido parcialmente financiado por Proyecto Fondecyt N°1181506

SANTIAGO DE CHILE
2018

RESUMEN DE LA MEMORIA PARA OPTAR AL
TITULO DE: Ingeniero Civil en Computación
POR: David Miguel Muñoz León
PROFESOR GUÍA: Nancy Hitschfeld

Visualizador y Evaluador de Mallas Geométricas sobre una Plataforma Web

El objetivo del presente trabajo de título consiste en el diseño y desarrollo de una aplicación web para la visualización y evaluación de la calidad de mallas de polígonos. Las mallas de polígonos se refieren a un conjunto de polígonos, aristas y vertices que definen alguna forma u objeto.

Actualmente existen variados visualizadores de mallas de polígonos, pero la existencia de esta aplicación se justifica en que ninguno de los visualizadores existentes provee capacidades de evaluación según las propiedades geométricas de las mallas (y si las tiene, no es de manera explícita). Con esta característica, la aplicación busca ser de utilidad para investigadores que trabajen con mallas de polígonos, ayudándolos a encontrar defectos o problemas en estas de manera rápida y sencilla.

Durante el desarrollo de la aplicación se usan conceptos de “interacción humano computador” para lograr una aplicación que sea agradable estéticamente, y que sea fácil de aprender utilizar.

El enfoque principal del trabajo es el de aprender y aplicar conceptos de Computación Gráfica. Esto incluye entre otras la manipulación de las mallas utilizando matrices, modelos de iluminación y hacer uso de la GPU.

En esta línea, se procura también que el código de la aplicación sea lo suficientemente legible para que sirva también como una herramienta educativa, en que los estudiantes o personas interesadas en Computación Gráfica puedan ver en el código y documentación, los conceptos matemáticos y de programación que hay detrás de esta.

Como resultado final se obtiene una herramienta que permite visualizar mallas geométricas, las cuales pueden ser inspeccionadas con acciones como mover, rotar y escalar. Para evaluarlas permite seleccionar elementos de las mallas según propiedades geométricas, y permite generar gráficos respecto a estas mismas propiedades. La aplicación posee una interfaz usable, que es útil tanto para investigadores como para estudiantes. También tiene un buen rendimiento, que hace que la aplicación pueda soportar sin problemas mallas de hasta 1.500.000 de polígonos; y actualmente está a la par con otros visualizadores web existentes.

Todo lo anterior, sumado a que la aplicación es gratis y de código abierto, permite que cualquier persona pueda utilizarla sin restricción, ya sea para evaluar mallas o para aprender Computación Gráfica. También, cualquier usuario con conocimientos de programación tiene la libertad de adaptarla para sus necesidades específicas. En el futuro cercano, se planea mejorar el rendimiento de la aplicación, y aumentar el número de formas de visualizar y evaluar las mallas geométricas.

Agradecimientos

A mi papá, quien paso de ser el menos interesado en que estudiara computación a ser el más feliz y motivado en poco tiempo. Finalmente fue el que me ayudo a corregir la ortografía y redacción de este trabajo.

A mi mamá, por alimentarme todos estos años con la mejor comida. Si no fuese por ella habría muerto de hambre hace mucho tiempo… literalmente.

A mi hermana Lisy, quien fue la persona con la que más hable y me desahogue en mis peores momentos durante la carrera, y aún con su estrecho horario se hacía el tiempo para hablar conmigo.

A mi hermana Tole, quien siempre me cuido desde que era pequeño, y durante mis años universitarios fue mi compañera de ocio, especialmente para ver series.

A mi hermana Linda, por entender todos mis chistes y ser mi compañera de videojuegos. Realmente espero poder seguir jugando con ella por mucho tiempo.

A la gente del grupo Anime no Seishin Doukoukai, sin ellos mi paso por la universidad habría sido demasiado aburrido y monótono. Me gustaría mencionar especialmente a Alonso, Jorge, Sebastián y Orti. Aun cuando ahora no tengo mucho contacto con ellos, en realidad los estimo mucho.

A los amigos que hice y la gente que conocí durante el camino. En particular a John y Fabian, con quienes pase mis primeros años de universidad hasta que las circunstancias nos separaron; y a Juan Pablo e Ignacia, nuestro robot de LEGO todavía lo considero como una de las mejores cosas que hice en la carrera.

Al profesor Jeremy, quien fue el primero que me convenció que era capaz de aspirar a más, y me apoyo con muchos consejos durante el transcurso de la carrera.

Finalmente, agradecer a la profesora Nancy Hitschfeld, por darme la oportunidad de realizar este trabajo con ella. Su apoyo y comentarios realmente ayudaron a mantenerme motivado durante todo este proceso.

Tabla de contenido

1 Introducción

1.1 Antecedentes Generales

Una malla de polígonos se define como una colección de polígonos que representan algún objeto. Estas mallas tienen diversos usos; desde videojuegos, hasta simulaciones físicas. Son ampliamente utilizadas y estudiadas dentro del campo de la computación gráfica. En adelante también se referirá a las mallas de polígonos como “modelos”.

Aparte de una colección de polígonos, los modelos suelen contener más información, como vértices y aristas. Además, casi cualquier información faltante es sencilla de deducir; por ejemplo, se puede determinar que 2 polígonos de un modelo son vecinos si comparten una arista. Toda la información, explícita o implícita, puede ser relevante para determinar cuál es la calidad de un modelo.

La calidad de un modelo es un factor importante para la estabilidad de los cálculos y operaciones que se quieran hacer sobre él [1]. Sin embargo, el concepto de calidad suele ser subjetivo y depende del uso que se le quiera dar al modelo. Por lo mismo una de las mejores formas de determinar si un modelo es de suficiente calidad es visualizando, tanto el modelo en sí como sus propiedades. De esta forma el usuario podrá determinar si lo que observa es suficientemente bueno para el uso particular que él le quiere dar.

Hoy en día existen muchas herramientas de visualización de modelos; sin embargo, ninguna de ellas está especializada en evaluar la calidad de estos (más allá de visualizarlos), y aunque ciertamente en algunos es posible observar algunas propiedades, la curva de aprendizaje para utilizar estas herramientas suele ser demasiado alta como para que valga dedicar tiempo y esfuerzo.

El año 2013, también como resultado de un trabajo de título, se desarrolló un software de visualización de modelos llamado “Camarón”. Este tomó como base la siguiente premisa:

“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” [2]

Camarón logró cumplir con lo que se estaba proponiendo. Pero, aun cumpliendo con ser un evaluador, gratuito, multiplataforma y veloz, terminó sufriendo la misma falla que los otros visualizadores: una curva de aprendizaje muy alta. Además, cometió errores que los otros no presentaban: una interfaz poco intuitiva y la carencia de un instalador (que lo hacía extremadamente inaccesible).

El presente trabajo tiene como objetivo ser una versión web de Camarón, siguiendo los mismos objetivos de este. También debido a las críticas realizadas, se agregó a los objetivos originales el preocuparse de la usabilidad y la interfaz. Por otro lado, debido a las limitaciones de JavaScript, que por ejemplo no permite hacer manejo de memoria explícitamente, el rendimiento de la aplicación paso un poco a segundo plano y sólo se consideró en la medida de lo posible.

1.2 Motivación

Como ya se mencionó, existen numerosos visualizadores de modelos. Es necesario explicar porque se considera que crear un nuevo visualizador es necesario, y en que se diferenciará de los demás.

El software Camarón tenía sus propios motivos, pero principalmente apuntaba a la antigüedad de la mayoría de los visualizadores, causando que no aprovecharan el hardware y las técnicas de última generación. Pero en la práctica, a pesar de tomar en cuenta el hardware, se demoraba más en cargar las mallas que los visualizadores contra los que se comparó. La mayoría de las ventajas se observaban poscarga de la malla, pero estás ventajas no eran fácilmente apreciadas, y eran opacadas por una pobre usabilidad.

La mayoría de los otros visualizadores son muy antiguos y no han recibido soporte en años. En particular en esta categoría están TetView y GeomView, descontinuados en 2004 y 2007 respectivamente [3][4]. Estos softwares no tienen una curva de aprendizaje muy alta, pero sus limitadas funciones y su antigüedad hacen que no sean muy buenas opciones hoy en día.

Quizá el visualizador más utilizado en el ámbito científico es MeshLab. Sigue siendo mantenido hoy en día e incluso ha recibido premios por su contribución al “progreso científico”. Es, en realidad, más que un visualizador, según su propia descripción puede “editar, limpiar, sanar e inspeccionar mallas” [5]. Sin embargo, debido a su cantidad de funcionalidades, también tiene una curva de aprendizaje muy alta. Su interfaz es poco intuitiva y hace casi imposible comenzar a utilizarlo sin un tutorial.

El presente trabajo está desarrollado fuertemente bajo una motivación personal: entrar al mundo de la computación gráfica. Crear una versión web de Camarón se presentó como una buena forma de hacer esto. La motivación detrás de esta aplicación sería la misma que tuvo Camarón: crear un visualizador y evaluador de mallas geométricas, gratuito y de código abierto, que aproveche la potencia del hardware actual.

Además, ya no era simplemente crear un visualizador más, si no que al ver los problemas que presentaban los otros, principalmente los de usabilidad, se percibió también como una necesidad la existencia de un software que fuese simple de utilizar.

Es así como nace “Camarón Web”, que en adelante será referido como “la aplicación”. Su principal diferenciación con respecto a otros visualizadores es que está escrito en JavaScript, no depende de un servidor ni de una conexión internet y utiliza conceptos de diseño con el fin de otorgar una interfaz simple e intuitiva.

Esta facilidad de uso apunta también a que la aplicación no solo sea una herramienta de apoyo a la investigación, sino que también sea educativa. Esto último permite que los estudiantes interesados en computación gráfica tengan acceso a una herramienta fácil de utilizar, entender y modificar (teniendo en cuenta que esto último igual requiere cierto nivel de conocimientos de programación).

1.3 Objetivos

1.3.1 Objetivo General

El presente trabajo de título tiene como objetivo diseñar y desarrollar un visualizador de mallas de polígonos. El visualizador debe ser fácil de instalar, utilizar y modificar. Tiene que ser desarrollado como una plataforma web, tomando las ventajas de las tecnologías actuales que permiten el uso de la GPU desde el mismo browser.

Además, la aplicación será gratis y de código abierto. Esto permitirá que, de ser necesario, sea fácil extenderla y modificarla para los usos particulares que le quiera dar cada usuario (esto último asumiendo que el usuario es parte del público objetivo y tiene ciertos conocimientos de programación, o que cuenta con la ayuda de un programador).

1.3.2 Objetivos Específicos

Al ser una versión web de Camarón, la aplicación desarrollada debe proveer las mismas funcionalidades que presentaba este. Específicamente:

1. Ser capaz de visualizar mallas de polígonos.
2. Permitir al usuario control libre sobre la cámara para poder inspeccionar las mallas desde distintas posiciones y ángulos.
3. Proveer de diversas modalidades de visualización para facilitar el análisis del modelo.
4. Otorgar al usuario variadas formas en las que se puedan seleccionar los distintos elementos de una malla.
5. Incorporar criterios de evaluación de las mallas, según distintas propiedades de esta que permitan al usuario determinar su calidad.
6. Que la estructura del código sea suficientemente mantenible para que otros desarrolladores sean capaces de agregar nuevas visualizaciones, formas de seleccionar y criterios de evaluación sin mayor problema.

En el proyecto original, el rendimiento y optimización de memoria fue uno de los puntos centrales. Como se mencionó en la sección 1.2. la optimización de memoria en JavaScript, aunque no imposible, es complicada. Aun cuando este no es uno de los enfoques principales, el rendimiento de la aplicación igual es comparado contra el del visualizador MeshLabJS, el cual es una versión web de MeshLab.

Además, a diferencia del proyecto original, se utilizó elementos y conceptos de diseño para desarrollar la interfaz de usuario. Esto, junto con el hecho de ser una plataforma web hace necesario especificar dos objetivos más:

7. Aplicar conceptos de Interacción Humano-Computador (HCI) durante el desarrollo de la aplicación para el desarrollo y diseño de la interfaz.
8. Que la aplicación sea ejecutada exclusivamente desde el lado del usuario y no dependa de ningún servidor; permitiendo así que se pueda ejecutar sin acceso a internet sin mayor problema.

1.4 Metodología de Trabajo

Para el desarrollo de la aplicación se optó por el uso un proceso de desarrollo “incremental e iterativo”; es decir siguiendo las etapas de análisis, diseño, implementación y evaluación. Esta metodología incluye tanto la planeación e implementación del código, como el diseño de la interfaz.

En la etapa de análisis se determinan los requerimientos del proyecto, además de definir detalles como qué tipo de usuario es el público objetivo, y cuáles son las herramientas por utilizar durante el desarrollo.

En la etapa de diseño se realiza por un lado la planeación del modelo de la aplicación (esto es, la estructura y como será implementada), y por el otro lado el diseño de la interfaz de usuario aplicando conceptos de usabilidad.

La etapa de implementación corresponde a la programación en sí, y a como se conectan todas las funcionalidades programadas con el diseño desarrollado. Aquí se deben seguir todos los lineamientos determinados en la etapa de diseño.

Finalmente, la evaluación corresponde a hacer medidas y comparaciones de rendimiento en el caso del código; y análisis de calidad/usabilidad en el caso de la interfaz de usuario.

Esta metodología en particular explicita que las funcionalidades planeadas deben dividirse en fragmentos de tiempos llamados incrementos. Cada incremento debe entonces pasar por las 4 etapas mencionadas, de manera iterativa, agregando en cada una de estas una nueva funcionalidad a la aplicación.

Todas estas etapas serán mostradas en orden en las secciones 3, 4, 5 y 6. Para una mejor compresión de la aplicación, no se explicitarán los incrementos específicos realizados definidos en la metodología, sin embargo, por completitud esta información podrá ser encontrada en el anexo C.

1.5 Alcance del Proyecto

Hasta el momento se ha mencionado la existencia de varios visualizadores. Sin una diferenciación clara, el presente proyecto puede dar la sensación de ser innecesario, o de simplemente ser uno más.

Es necesario entonces especificar que, a pesar de haber muchos visualizadores, los enfoques y objetivos cambian entre ellos. Se puede decir que la mayoría tienen solo una cosa en común: la capacidad de mostrar un modelo en la pantalla. Luego de eso, que se puede hacer con ese modelo varía entre visualizadores.

En particular, el enfoque de la aplicación es único, y es comparable en funcionalidad solo con el proyecto Camarón original. Este enfoque es el de la evaluación de calidad, que se apoya en dos acciones: seleccionar y evaluar. Seleccionar permite elegir alguna propiedad del modelo, y seleccionar elementos en el modelo respecto a esta propiedad. Evaluar permite elegir alguna propiedad del modelo, y mostrar información general respecto a esta propiedad. Un ejemplo concreto sería el de los ángulos internos de los polígonos de uno modelo. Seleccionar permitiría decir algo como “marcar todos los polígonos con ángulos entre 0 y 60”, mientras que evaluar genera un histograma mostrando que ángulos están presentes en el modelo y en qué proporción.

Esta funcionalidad es única de la aplicación (y del proyecto Camarón original), y apunta a apoyar a los investigadores que trabajan con modelos a encontrar fallas, o identificar ciertas propiedades de un modelo más allá de lo que se puede ver a simple vista.

También es necesario destacar que, aunque se ha mencionado el uso de HCI, y que se considera de relativa importancia, el principal tema de este trabajo y el enfoque principal es el uso de conceptos y técnicas de Computación Gráfica.

1.6 Contenidos

1. **Introducción:** En este capítulo se introduce el tema, se expresa la motivación y se presentan los objetivos del proyecto. También se explicita cual es el alcance esperado del trabajo, y cuál es la metodología utilizada.

2. **Marco Teórico:** En este capítulo se presentan los conceptos teóricos necesarios para entender el desarrollo del trabajo. Se presentan conceptos de Computación Gráfica, que es el enfoque del trabajo; y de Interacción Humano Computador, del cual se utilizan técnicas de desarrollo.

3. **Análisis:** En este capítulo se presenta el análisis hecho para determinar cuál es el tipo de usuario al que está dirigida la aplicación, determinar los requerimientos de esta, y que herramientas se seleccionaron para el desarrollo.

4. **Diseño:** Este capítulo se presenta en 2 partes. En la sección 4.1 se presenta el diseño del modelo de la aplicación, el cual es usado como guía para el código. En la sección 4.2 se presenta como se realizó el diseño de la interfaz.

5. **Implementación:** En la sección 5.1 de este capítulo se presentan las explicaciones de cómo funcionan cada una de las acciones que se pueden realizar, y como se ven estas en la interfaz. La sección 5.2 entra en detalle respecto a las partes específicas y más de bajo nivel de Computación Gráfica, y muestra cómo se ponen en la práctica varios de los conceptos expuestos en el marco teórico.

6. **Evaluación:** En este capítulo se muestran que acciones se tomaron para asegurar la usabilidad de la aplicación. También se expresan las mediciones de rendimiento (como uso de RAM y GPU), y se comparan contra otro visualizador.

7. **Conclusiones:** En este capítulo se presentan las conclusiones del trabajo. Se toman en consideración la motivación, los objetivos, los requerimientos y el resultado final para determinar el éxito del trabajo. También se discute sobre posible trabajo futuro que puede ser llevado a cabo.

8. **Referencias:** Aquí se presentan las fuentes de información utilizadas para el desarrollo del proyecto, principalmente referidas a los conceptos del marco teórico, trabajos relacionados, y técnicas específicas de Computación Gráfica.

9. **Anexos:** Se presentan 4 secciones adicionales, que presentan información adicional sobre conceptos utilizados durante el trabajo.

2 Marco Teórico

2.1 Computación Gráfica

La computación gráfica es la ciencia de comunicarse visualmente a través de una pantalla de computador y sus dispositivos de interacción. Es un campo que usa conocimientos de varias disciplinas. Por ejemplo, se puede usar física para modelar la luz o matemática para definir formas.

Dentro del campo de la computación grafica existe el concepto de modelo, que puede referirse a dos cosas. Primero, están los modelos geométricos, que se refieren a representaciones de las cosas que se quiere mostrar en la pantalla (y es en esta categoría donde caen los definidos en la sección 1.1). Segundo están los modelos matemáticos que se refieren a representaciones de algún proceso físico [6].

Una de las formas más populares de representar modelos geométricos son las mallas.

2.1.1 Mallas Geométricas

Hasta el momento se han mencionado las mallas de polígonos. Dentro de la computación gráfica también existen los que se llaman mallas de poliedros, que son una colección de poliedros (comúnmente tetraedros o hexaedros). Por su lado cada poliedro de estas mallas está definido por polígonos, y cada polígono por vértices. Al igual que las mallas de polígonos, se usan para representar objetos en 3 dimensiones.

La aplicación desarrollada solo funciona con mallas de polígonos; por lo tanto, cuando se mencione la palabra “malla” o “modelo” siempre corresponderá a dicho concepto a menos que se indique lo contrario. Planteado esto, está planeado extender la aplicación para soportar todo tipo de mallas en un futuro cercano.

2.1.2 Matrices

Se sabe que las mallas de polígonos son un conjunto de polígonos. Además, dichos polígonos están presentados por un conjunto de vertices, y los vertices son puntos en el espacio representados por un vector. Para poder mover una malla, es necesario mover todos los vertices que la conforman de igual forma. Lo mismo ocurre para rotar o para escalar.

Inicialmente se podría pensar que esto es simple. Si quiero mover una malla 3 posiciones hacia la izquierda en el eje x, basta con restar 3 a todos los “x” de los vertices de la malla, lo cual sería correcto. Sin embargo, rotar y escalar no es tan sencillo. Por lo anterior, en computación gráfica se ocupan las llamadas “matrices de transformación” [7].

Una matriz, en términos simples, es un arreglo de números con una cantidad predefinida de filas y columnas. Las matrices de transformación son de tamaño 4x4 y existen 3 tipos: de traslación, de escala, y de rotación. En ellas se contiene información sobre que alteraciones se quiere realizar a los vertices de una malla.

Una matriz de traslación se define de la siguiente forma:

$$\begin{matrix} 1 & 0 & 0 & X \\ 0 & 1 & 0 & Y \\ 0 & 0 & 1 & Z \\ 0 & 0 & 0 & 1 \end{matrix}$$

Donde X, Y y Z representan cuanto se quieren trasladar los vértices en las direcciones correspondientes.

Una matriz de escala se define de la siguiente forma:

$$\begin{matrix} X & 0 & 0 & 0 \\ 0 & Y & 0 & 0 \\ 0 & 0 & Z & 0 \\ 0 & 0 & 0 & 1 \end{matrix}$$

Donde nuevamente X, Y y Z indican por cuanto se quiere escalar en cada eje. Por ejemplo, si el modelo se quiere duplicar en tamaño, las tres variables deberán ser marcadas con el valor 2.

Las matrices de rotación son más complicadas, pues estas dependen alrededor de que eje se está girando. Dicho eje no necesariamente corresponde a “x”, “y” o “z”, si no que puede ser definido por un vector arbitrario. La estructura de una matriz de rotación puede ser consultada en el Anexo A.

Lo importante de entender es que al multiplicar un vértice por cualquiera de estas matrices producirá el cambio deseado. Quizá la mejor propiedad que tienen estas matrices es que se pueden multiplicar entre sí, sin importar el tipo, y la matriz resultante todavía contiene todas las transformaciones. Esta matriz en computación gráfica se conoce como “Model Matrix”. Al crearla hay que tener en consideración 2 detalles:

1. El orden en que se multiplican las matrices sí afecta el resultado. Realizar una traslación y luego una rotación es distinto a realizar una rotación y luego una traslación.
2. Al multiplicar matrices de transformación, la última matriz en multiplicarse es la primera operación que se hará efectiva sobre el vértice.

En la práctica, las matrices de transformación no es lo único que importa. El usuario quiere ser capaz de proyectar el modelo 3D en la pantalla, y solo las posiciones de los vertices no son suficientes. Con estas, se puede ubicar el modelo en el mundo virtual, pero hay que asegurarse de que sea visible. La mejor forma de hacer esto es imaginarse una cámara. Dicha cámara parte en la posición (0, 0, 0), y también se le pueden aplicar matrices de transformación.

La mayoría de las librerías gráficas incluyen un método que se llama “lookAt”, que recibe una posición y un destino. Sin importar nuestra posición, si nuestro destino es el centro del modelo, tendremos una matriz que representa una cámara que esta “mirando” al modelo. A esta matriz se le conoce como “View Matrix”.

Finalmente, es necesario definir lo que se llama la matriz de proyección. Con la “View Matrix” todas las coordenadas quedan definidas relativas a la cámara. Al aplicar la matriz de proyección, las coordenadas quedan definidas dentro de un cubo, y todo lo que quede dentro de ese cubo, es lo que finalmente se logrará ver en la pantalla. Esta matriz es llamada “Projection Matrix”.

Existen 2 tipos de proyecciones: perspectiva y ortogonal. Perspectiva simula la visión humana, y se asegura que los elementos que están más lejos se vean más pequeños, mientras que los más cercanos se vean más grandes. La proyección ortogonal dibuja todo del tamaño que esté definido, sin importar si está más lejos o más cerca. Esta diferencia se logra apreciar en la figura 1. Al igual que lookAt, las librerías gráficas suelen incluir métodos para generar las matrices de proyección.

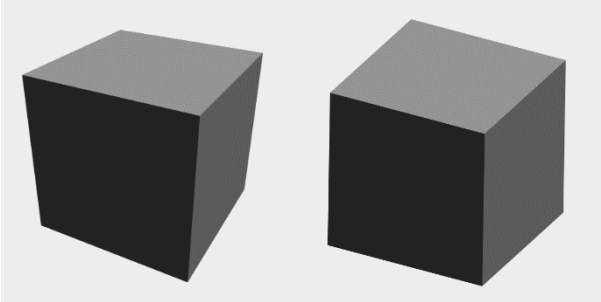

Figura 1 – Proyección en perspectiva (izquierda), y proyección ortogonal (derecha).

Cabe destacar que multiplicar cada una de estas matrices corresponde a cambiar el sistema de coordenadas que se está utilizando. El modelo esta inicialmente en coordenadas del modelo, al multiplicar la “Model Matrix” se pasa a coordenadas del mundo, luego con “View Matrix” se pasa a coordenadas de cámara, y finalmente con “Projection Matrix” se pasa a lo que se conoce como coordenadas homogéneas. Es posible multiplicar estas 3 matrices entre sí antes de aplicarlas al modelo, obteniendo así una única matriz que contiene las 3 transformaciones, la que se denomina como “Model View Projection Matrix”.

2.1.3 OpenGL y WebGL

OpenGL es una API gráfica para el manejo de modelos 2D y 3D. Fue introducido en 1992 y actualmente es la API gráfica más usada y mantenida. Existe un subconjunto de OpenGL llamado OpenGL ES que es especializado para funcionar en sistemas embebidos (teléfonos, consolas, vehículos, etc.) [8].

WebGL es una API gráfica para JavaScript, basada en OpenGL ES, y que fue diseñada para funcionar con el nuevo tag “canvas” de HTML5. Está integrado en la mayoría de los browsers web modernos, y por lo mismo no requiere de ningún plugin o programa externo para funcionar. [9]

2.1.4 Graphics Rendering Pipeline, Shaders y GLSL

La “Graphics Rendering Pipeline”, es como se conoce a la cadena de acciones que se deben realizar para “renderizar” una imagen en 2D, dada una cámara virtual, objetos tridimensionales, fuentes de luz, texturas, y más [10].

Aunque los pasos del rendering pipeline pueden variar de una API otra, OpenGL define en particular los pasos mostrados en la figura 2. En esta imagen los cuadros amarillos corresponden a etapas automáticas del pipeline, mientras que los azules son etapas programables. Los cuadros con línea punteada indican que programar dicha etapa es opcional [11].

WebGL, al estar basado en OpenGL comparte el mismo pipeline, pero, no incluye los pasos programables de “Tessellation” y “Geometry Shader”. En el contexto de este trabajo, solo se referirá a WebGL.

En WebGL, se puede definir el “Vertex Shader” y “Fragment Shader”. Un shader es por definición un programa definido por el usuario para correr en alguna etapa del rendering pipeline [12]. Para el caso de WebGL, los shaders están escritos en un lenguaje llamado GLSL, que literalmente significa “OpenGL Shading Language”.

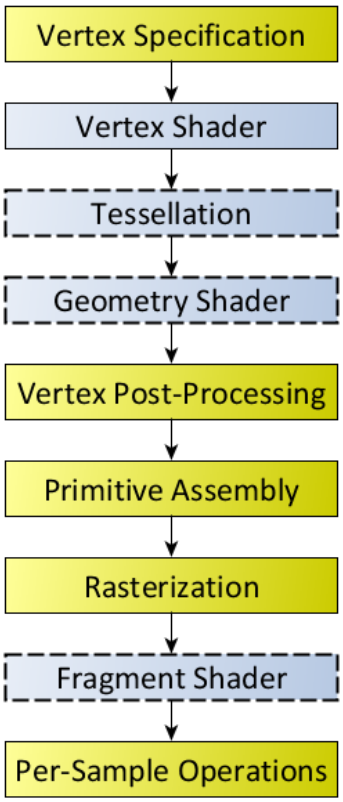

Figura 2 – OpenGL Rendering Pipeline.

El “Vertex Shader” es el encargado de calcular la posición de los vertices; esto corresponde a pasar desde las coordenadas del modelo, hasta coordenadas homogéneas. Opcionalmente es esta etapa se puede calcular cualquier información necesaria asociada a los vértices; como normales, color, textura, etc.

En la etapa de “Vertex Post-Processing”, se transforman las coordenadas del “Vertex Shader” a coordenadas de pantalla. Esto es, determinar a qué pixel corresponde cada vértice. Posteriormente en la etapa de “Rasterization”, se determina por cada primitiva (comúnmente triángulos), según las posiciones de sus vertices en coordenadas de pantalla, cuáles son los pixeles que caen dentro de esta. Cada uno de estos pixeles es lo que se conoce como fragmentos.

El “Fragment Shader” es el encargado de indicar como se dibujan los fragmentos. Esto puede incluir desde mostrar colores sencillos, hasta aplicar una textura o técnicas de sombreado. Este Shader puede recibir información generada desde el “Vertex Shader” como normales, colores o texturas, la cual, en caso de existir, fue apropiadamente interpolada en la etapa de “Rasterization”.

Es necesario destacar que todo este pipeline se ejecuta extremadamente rápido, pues en la GPU todos los vertices son procesados al mismo tiempo.

2.1.5 Modelos de Iluminación

Para poder dibujar un modelo 3D, es necesario que sea posible distinguir su profundidad una vez que esté dibujado. Si se indica el mismo color para todos los vertices, la imagen generada se vería plana.

Es para eso que existen los modelos de iluminación, que corresponderían a lo que se llamó “modelo matemático” al principio de la sección 2.1. Existen muchos modelos de iluminación, que toman en cuenta distintas propiedades físicas y tratan de simular cómo se comporta la luz de una forma lo más real posible.

Para el presente trabajo, solo un modelo de iluminación es relevante. Es el más simple de todos y no sirve para visualizaciones realistas, pero es suficiente para inspeccionar y evaluar un elemento. El modelo se llama “Iluminación Direccional”.

Este modelo asume que toda la luz viene uniformemente desde una única dirección. En el mundo real, la luz del sol puede ser considerada como direccional, pues el sol esta tan lejos que se pueden percibir los rayos como si llegaran de forma paralela [13].

Calcular una luz direccional es bastante simple (siempre que no se consideren cosas como sombra, o material del modelo). Solo con saber la dirección de la luz, y la dirección de cada cara, es posible determinar cuál es la intensidad de la luz en cada cara.

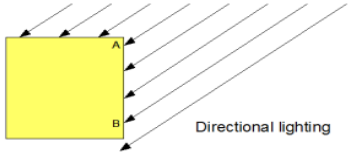

Figura 3 – Luz Direccional [14]

La dirección de una cara corresponde a su vector normal. Asumiendo que ambos vectores están normalizados, el producto punto entre el vector de iluminación y el vector normal puede dar un número entre 1 y -1, que corresponde a la intensidad de la luz en la cara. El color de una cara también está representado por un vector, y si multiplicamos dicho vector por el numero resultante, obtenemos el color por el que hay que pintar la cara. Si el número es 1, obtenemos el mismo color, y si es menor, la intensidad del color comienza a bajar. Si el número es 0 o menor (es decir negativo), se obtiene negro.

El modelo de iluminación se implementa en el Fragment Shader. Pero como se indicó en la sección anterior, estos reciben la información de los Vertex Shaders, los cuales trabajan con vertices. Para aplicar iluminación, se necesitan entonces las normales de los vertices. El Vertex Shader entrega esta información, la cual es interpolada en la etapa de “Rasterization”, y luego el Fragment Shader puede aplicar el color con la iluminación adecuada por pixel. Dependiendo de la dirección de la normal se pueden obtener distintos efectos. La figura 4 muestra como ejemplo, la diferencia entre usar las normales reales de los vertices y usar la normal de cada cara en sus vertices.

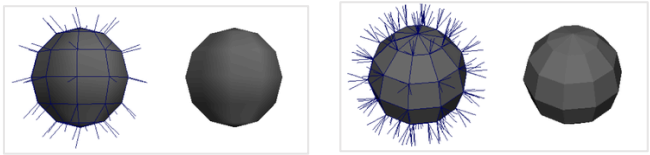

Figura 4 – Calculo de luz con las normales de los vertices (izquierda) y con las normales de las caras (derecha) [15]

2.1.6 Visualizadores Existentes

En la sección 1.2 ya se nombraron tres visualizadores existentes: GeomView, TetView y MeshLab. Estos fueron los que se utilizaron oficialmente para compararse contra el proyecto Camarón original.

Sin embargo, para la aplicación desarrollada, el contexto de trabajo es muy diferente al de estas aplicaciones, pues se trata una aplicación web. Para poder tener un buen punto de referencia es necesario buscar aplicaciones que se manejen en el mismo contexto.

Existen varios visualizadores online, pero la mayoría se enfoca en mostrar trabajos artísticos y no permiten subir modelos en tiempo real. Entre estos existe SketchFab, que es un repositorio de modelos 3D y que permite realizar todas las acciones básicas de movimientos de modelos (mover y rotar) [16].

Para el contexto específico de “herramienta de apoyo a la investigación” se encontró otro visualizador web, el cual es una versión web de MeshLab llamada MeshLabJS [17]. En la figura 5 se puede ver cómo es la interfaz de esta aplicación.

MeshLabJS es de código abierto y está escrita en C++ y JavaScript. Todo el código escrito en C++ fue compilado a JavaScript usando un compilador llamado “emscripten”. Hace utilización de Three.js, que es una librería gráfica que facilita el uso de WebGL [18].

MeshLabJS no ofrece las mismas funcionalidades que Camarón Web, pero para la acción de visualizar un modelo es un buen punto de referencia para comparar rendimientos y tiempos de espera.

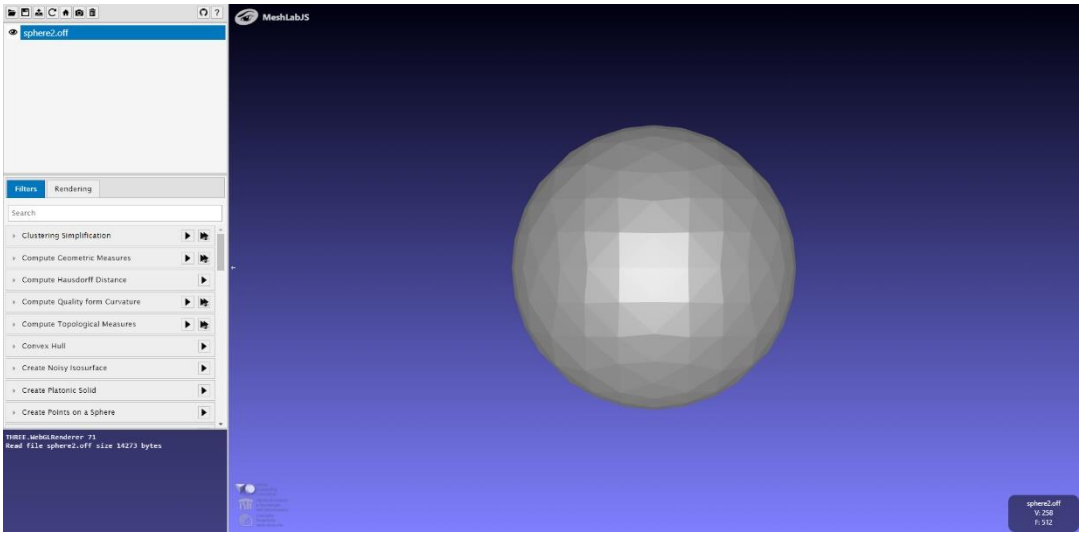

Figura 5 – Interfaz de MeshLabJS, con modelo de una esfera abierto.

2.2 Interacción Humano Computador

La interacción humano computador (o HCI por sus siglas en inglés) es una rama de estudio enfocado en el diseño y uso de tecnologías computacionales. En particular se enfoca en la interacción entre los usuarios con los computadores.

HCI tiene un enfoque académico, y se realiza mucha investigación basada en entender de manera empírica a los usuarios [19]. El presente trabajo no es en sí una investigación en el área de HCI, si no que toma elementos de HCI para poder diseñar mejor la interfaz de la aplicación a desarrollar.

2.2.1 Contextos

En HCI contexto se refiere a las condiciones reales bajos las cuales una aplicación será utilizada. Los contextos son necesario para poder enfocar el diseño, y realizar evaluaciones de usabilidad bajo las condiciones correctas [20].

Existen varios tipos de contexto. Para el trabajo actual se consideraron 2:

1. **User Context:** Representa la información del tipo de usuario que utilizará la aplicación, esto puede incluir información como edad, preferencias, ubicación, situación social, etc.
2. **Computing Context:** Es todo lo relacionado con recursos computacionales disponibles, esto puede incluir información como redes disponibles, ancho de banda, costos de comunicación, etc.

2.2.3 Wireframes

Al momento de diseñar una interfaz, una de las técnicas más utilizadas es la creación de wireframes. Un wireframe es una guía visual para representar la distribución o esqueleto de una aplicación [21]. En palabras más simples, un wireframe es una representación de la interfaz, antes de que exista la aplicación.

Existen varias formas y herramientas para crear wireframes. Desde dibujos, hasta software diseñados especialmente para diseñar interfaces. Durante el proceso de diseño se suelen usar dibujos y diagramas en las etapas iniciales de un proyecto, y herramientas más especializadas en etapas más avanzadas.

La idea principal de los wireframes es que sea posible testear la usabilidad, y definir los lineamientos de diseño en cada iteración de este, aun cuando la aplicación no está implementada. Cuando uno trabaja con un cliente, los wireframes suelen evolucionar hasta que son aceptados por el cliente, dándole confianza al desarrollador para el resto del trabajo [22].

Notar que en el presente trabajo también se usa la palabra wireframe en el contexto de “wireframe de un modelo”. Este se refiere a dibujar en la pantalla las aristas de un modelo, haciendo que parezca una malla de alambres. Siempre será claro según el contexto que tipo de wireframe es al que se está refiriendo.

2.2.4 Principios de Diseño Universal

Los principios de diseño universal son una serie de lineamientos que sirven para guiar un proceso de diseño, de forma que el resultado final sea más accesible y usable. No es un concepto exclusivo de HCI, pues este existe de mucho antes y se puede aplicar a cualquier tipo de diseño. Varios de estos principios han sido estudiados desde el punto de vista de HCI y siempre son una buena guía para tener en cuenta. Estos son [23]:

1. **Uso Igualitario:** Se refiere a accesibilidad, y se insta al diseñador a pensar en usuarios con diferentes habilidades.

2. **Flexibilidad de Uso:** Se refiere a dar a los usuarios varios caminos o formas de realizar distintas acciones, haciéndolos sentir con más libertad.

3. **Uso Simple e Intuitivo:** Se refiere a reducir complejidad del diseño, y reducir cargas mentales.

4. **Información Perceptible:** Se refiere a hacer que la información mostrada sea fácil de entender y acceder. Redundancia de información es favorable en este sentido.

5. **Tolerancia al Error:** Se refiere a minimizar riesgos y peligros sobre acciones accidentales o no intencionadas.

6. **Bajo Esfuerzo Físico:** Se refiere a que los diseños deben poder ser usados de manera cómoda y provocando un mínimo de fatiga.

7. **Tamaño y Espacio para Facilidad de Uso:** Se refiere a considerar el tamaño y posición de todos los elementos del diseño, de forma que sean fáciles de alcanzar y manipular.

Según el contexto, no es en realidad necesario aplicar todos estos principios. Sin embargo, es bueno tenerlos todos en mente al crear o modificar un diseño. Algunos de estos lineamientos son difíciles (o poco intuitivos) de considerar en desarrollo de aplicaciones, pero todos pueden ser utilizados en mayor o menor medida.

Para el presente proyecto se consideró explícitamente los principios 4, 5, 6 y 7. El principio 3 también se siguió, pero fue considerado indirectamente. Esto pues, aplicar conceptos de HCI al diseño, en particular el uso de wireframes y reuniones con usuarios, tiene como objetivo aumentar la usabilidad y reducir la complejidad. Esto va en línea con dicho principio.

2.2.5 Evaluación de Interfaces

Para evaluar la usabilidad de una interfaz, existen diversos métodos de evaluación. Estos métodos se pueden clasificar de distintas maneras según en qué etapa del proyecto se pueden aplicar, o el tipo de datos que produce [24]. Las clasificaciones se presentan de a pares, donde se indica que una técnica puede ser o una, u otra.

Entre estas clasificaciones tenemos:

1. **Evaluación Rigurosa vs Rápida:** Una evaluación es rigurosa cuando es precisa, confiable y generalizable. Permite indicar si una interfaz es mejor que otra. Una evaluación es rápida cuando no se espera precisión y suele utilizarse de manera interna en los equipos de desarrollo.

2. **Evaluación Analítica vs Empírica:** Una evaluación es analítica cuando solo se necesita razonamiento para aplicarla. Se considera empírica cuando depende de hacer observaciones y medidas.

3. **Evaluación Formativa vs Sumativa:** Una evaluación es formativa cuando puede ser aplicada en cualquier etapa del diseño, y sirve para refinar ideas. Es sumativa cuando se aplica en las etapas finales de un proyecto y sirve para evaluar sistemas enteros.

4. **Evaluación Cualitativa vs Cuantitativa:** Una evaluación es cualitativa cuando sus resultados involucran palabras, y dan información sobre los usuarios y su contexto. Es cuantitativa cuando sus resultados involucran números que pueden ser usados para comparar datos.

Todos los métodos de evaluación pueden ser clasificados usando estas categorías. Las evaluaciones rigurosas suelen tener un alto costo y demandan tiempo. Por un tema de tiempo, y considerando que el enfoque final no es la formalidad en uso de HCI, este tipo de evaluaciones no se realizaron en el presente proyecto.

Expuesto esto, si se aplicaron en particular, dos tipos de evaluaciones rápidas [25]:

1. **Recorrido de diseño:** Es un tipo de evaluación analítico, formativo y cualitativo. Se puede utilizar con usuarios representativos. Consiste en que los diseñadores explican oralmente lo que van haciendo a los usuarios, y los usuarios identifican posibles problemas y los discuten.

2. **Evaluación basada en escenarios:** Es un tipo de evaluación analítico, formativo y cualitativo. Se basa en definir un conjunto de escenarios que describen como los usuarios interactúan con la aplicación. Se guía a los usuarios a través de un prototipo. Los usuarios contrastan el prototipo con los escenarios e identifican potenciales problemas.

2.3 Misceláneos

2.3.1 Programación Orientada a Objetos

La programación orientada a objetos es un paradigma de programación que se basa en la utilización de lo que se llama “objetos” para diseñar aplicaciones [26]. Un objeto es una forma de modelar elementos o ideas del mundo real que se quieran representar dentro de un programa. Los objetos pueden contener datos y códigos, que representan la información y el comportamiento de lo que se esté modelando [27]. Los objetos se pueden diseñar usando diagrama de clases, que son diagramas que muestran la información, comportamiento y relaciones con otros objetos.

2.3.2 Patrones de Diseño

Los patrones de diseño son soluciones generalizables para problemas recurrentes que aparecen al momento de programar. Se usan para ahorrar tiempo de desarrollo y existen bajo la premisa de que “alguien ya resolvió tu problema” [28].

En orientación a objetos existen patrones de diseño que se suelen expresar mostrando las interacciones entre objetos, y explicando que problema solucionan. Para el presente trabajo, solo se utilizó de manera explícita uno: Strategy Pattern. Al ser soluciones generalizables, existe la posibilidad de haber usado otro inconscientemente.

Strategy Pattern es un patrón de diseño que se utiliza para poder definir una familia de algoritmos y hacerlos intercambiables entre ellos. En la figura 6 se muestra el diagrama de clases general del Strategy Pattern.

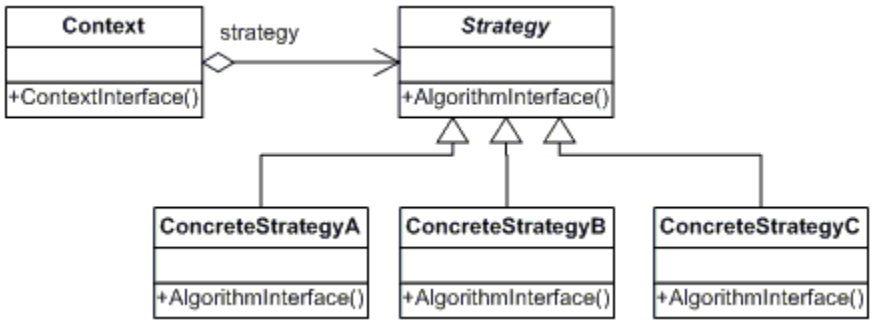

Figura 6 – Diagrama de clases de Strategy Pattern [29].

Se define una interfaz Strategy que luego debe ser implementada por las ConcreteStrategy. Cada “ConcreteStrategy” implementa el algoritmo requerido usando los datos de la interfaz. De esta forma los algoritmos se pueden seleccionar e intercambiar durante la ejecución.

3 Análisis

El objetivo último de esta etapa es determinar los requerimientos. En el presente trabajo no fue necesario hacer un análisis riguroso, pues la mayor parte de la funcionalidad ya estaba determinada por el software Camarón original.

El principal enfoque fue extraer los requerimientos originales y decidir qué cosas considerar, que cosas no considerar, y que cosas nuevas agregar.

Como un aspecto nuevo, se determinó el perfil que iban a tener los usuarios de la aplicación. Esto no afecta directamente a los requerimientos, sino que tiene la intención de dirigir el diseño.

También se agrega a esta sección el análisis que se realizó para la selección de herramientas que se utilizó durante el desarrollo de la aplicación.

3.1 Usuarios

A nivel de la aplicación misma, no se encontró motivo para separar por tipo de usuario. Todos pueden realizar las mismas acciones con el mismo nivel de libertad. Por lo anterior, al momento de escribir los requerimientos, solo se considera un tipo de usuario.

A nivel conceptual, se considera que la aplicación tiene como público objetivo dos tipos de usuarios: Investigadores y Estudiantes.

Los investigadores son personas que dentro de su área específica trabajan con mallas de polígonos. La mayoría seguramente tiene experiencia usando visualizadores y por lo mismo tendrán ciertas expectativas sobre cómo deben funcionar. Si un visualizador se vuelve muy complejo prefieren hacer sus propias herramientas para extraer información, pues es más simple y tiene más valor para ellos que aprender a usar un software.

Los estudiantes son personas que aún están en formación, pero que presentan cierto interés por la computación gráfica. No se espera que tengan experiencia con visualizadores, pero muchas veces su curiosidad los ha llevado a instalar visualizadores y ver modelos. No necesariamente estarán involucrados en investigación, por lo que muchas veces tendrán más curiosidad acerca del código que del visualizador mismo.

En sentido práctico, la definición de estos usuarios no tiene un impacto claro (por ejemplo, no podemos decir que un botón está en cierto lugar porque a un investigador le gustaría). Su importancia radica más en el efecto que tienen cuando uno diseña o implementa, siempre teniendo al público objetivo en mente. Además, estos son los perfiles de las personas que se debería buscar al momento de hacer la evaluación del software.

3.2 Requerimientos

Los requerimientos fueron escritos como una lista sencilla de acciones y funcionalidades posibles. Los requerimientos, separados por categoría son:

Vista del modelo:

- Rotación libre del modelo.
- Traslación libre del modelo.
- Zoom in, Zoom out.
- Volver modelo a la posición original.
- Visualizar modelos como malla de alambres.
- Visualizar modelo como nube de puntos.
- Seleccionar método de iluminación del modelo.

Selección de polígonos:

- Seleccionar polígonos del modelo según criterio definido por el usuario, basado en las propiedades de estos.
- Seleccionar polígonos según su identificador.

Manejo de Archivos:

- Permitir lectura de archivos en formato off.
- Chequear consistencia del modelo al cargarlo.

Estadísticas modelo:

- Mostrar número de caras y vértices.
- Mostrar medidas del modelo (largo, ancho, profundidad).

Evaluación:

- Evaluar modelos según criterio definido por el usuario, basado en las propiedades de los polígonos que lo componen.
- Como resultado, mostrar rangos de valores encontrados para la propiedad seleccionada en la evaluación.

Ciertos requerimientos del proyecto original se dejaron fuera porque no eran considerados dentro del alcance del proyecto (por ejemplo, todo lo que tiene que ver con mallas de poliedros) o porque no aportaban al enfoque del proyecto (por ejemplo, la aplicación no genera cambios sobre los modelos, por lo que guardar el modelo no se consideró). Estos ciertamente son interesantes, y se evaluará agregarlos en el futuro.

Otros requerimientos se redujeron a su esencia. Por ejemplo, solo se consideró abrir archivos en formato “off”. Pero la implementación permite agregar soporte de nuevos formatos de manera sencilla (para alguien familiar con dicho formato).

3.3 Selección de herramientas

Al momento de seleccionar las herramientas por utilizar, hay que considerar las restricciones que se infieren de los objetivos planteados. Estas son:

1. Que sea fácil de “instalar”.
2. Que sea mantenible.
3. Que sea fácil de entender y modificar.

Con las decisiones tomadas, el primer objetivo se cumple por completo. La aplicación desarrollada no necesita instalación y basta con descargar el repositorio y abrir el archivo HTML. Los otros dos objetivos, aunque ayudan a tomar decisiones con respecto a las herramientas, terminan dependiendo mucho más de la calidad del código que de la herramienta seleccionada.

En las siguientes secciones se indican las herramientas seleccionadas, junto con la justificación de su elección. En caso de ser pertinente también se indican herramientas alternativas junto con la explicación de porque se descartaron.

3.3.1 JavaScript

Al ser una aplicación web, es claro que el lenguaje de programación debe ser JavaScript. Es cierto que existen herramientas para compilar otros lenguajes de programación a JavaScript, pero el uso de estas agrega dificultad de instalación y de comprensión, lo cual es contrario a las restricciones 2 y 3.

Para aplicaciones web, la mayoría de los desarrolladores opta por el uso de la librería JQuery. En este proyecto se optó por evitarlo lo más posible. Esto pensando en que usuarios interesados en ver cómo funciona el código pueden no dominar JavaScript, y enfrentarlos con JQuery antes de aprender bien JavaScript podría llevar a confusiones no deseadas.

Sin embargo, JQuery no se evitó por completo, debido a que el archivo entregado por el diseñador, que controla todas las animaciones de la interfaz está escrito exclusivamente con JQuery. A dicho archivo se le agregó al principio una advertencia indicando que es el único que utiliza JQuery y que no es necesario entenderlo para comprender las funcionalidades gráficas desarrolladas en el resto de la aplicación.

3.3.2 WebGL

Para la parte gráfica, se optó por utilizar WebGL directamente. Esto dado que es la API estándar para la manipulación y renderización de gráficos 3D en JavaScript. El uso de WebGL, junto con que el browser este bien configurado, es lo que asegura que se están ocupando las ventajas de la tarjeta gráfica en caso de que esté disponible. Además, en caso de que no esté disponible, igual es capaz de trabajar con los gráficos integrados, haciendo que la aplicación pueda funcionar en casi cualquier computador.

Existen varias librerías gráficas para JavaScript que utilizan WebGL y sirven para simplificar la sintaxis. En particular la más utilizada es three.js. Sin embargo, se optó por no usar ninguna. La razón principal es que uno de los objetivos del proyecto, era introducirse en el mundo de la computación gráfica. WebGL trabaja a muy bajo nivel para poder comunicarse con la GPU y su sintaxis es igual a la de OpenGL. Por lo mismo, era una de las mejores instancias para aprender cómo funciona todo, y tiene la ventaja de que el conocimiento adquirido es generalizable a otras herramientas más utilizadas.

Se podría pensar que no utilizar una librería gráfica va en contra de la tercera restricción. Y aunque es cierto que programar usando WebGL directamente puede hacer que el código se vuelva más confuso, se trabajó con la intención de compensar eso a través de planear un buen diseño de clases y una clara organización del código.

La idea principal es generar un código que sirva de guía y modelo para alguien que desee aprender a desarrollar en WebGL, donde se puede ver todas las operaciones matemáticas de manera más explícita que si se utiliza una librería.

3.3.3 glMatrix

Al trabajar con modelos, es necesaria mucha manipulación de matrices y vectores. Ni JavaScript ni WebGL proveen este tipo de operaciones por defecto. Por lo mismo, se optó por utilizar la librería glMatrix, la cual está diseñada exclusivamente para este tipo de trabajo, e incluso considera operaciones que se usan exclusivamente en aplicaciones gráficas [30].

3.3.4 plotly.js

Plotly.js es una librería de código abierto utilizada para la generación de gráficos. Esta se seleccionó exclusivamente para la generación de los histogramas resultado de la evaluación de un modelo.

Existen muchas librerías para generar gráficos en JavaScript, pero esta seleccionó particularmente por ser de código abierto, su facilidad de uso y que tiene soporte para histogramas explícitamente. Además, existen versiones de plotly para otros lenguajes de programación.

4 Diseño

La etapa de diseño se divide en 2 partes: El diseño del modelo, y el diseño de la interfaz. En ambas se toma como base la lista de requerimientos obtenidas en la etapa de análisis. Notar que el uso de la palabra modelo acá se refiere a modelo de los objetos de la aplicación, y no mallas de polígonos.

4.1 Modelo

Es necesario clarificar que JavaScript es un lenguaje basado en prototipos y no maneja el concepto de clases. En un sentido práctico, un prototipo y una clase pueden considerarse equivalentes, y por lo mismo serán utilizados diagramas de clases para el modelamiento de estos.

Otra consideración para tener en cuenta es que en los diagramas que se mostrarán se indican el tipo de las variables y de los retornos de los métodos. Esto puede causar confusión pues JavaScript es un lenguaje de tipado débil. La razón por la que se decidió indicar el tipo de las variables es para facilitar la comprensión a nivel conceptual. En varios casos es necesario tener el tipo de ciertas variables en mente, pues WebGL necesita un tipado fuerte al momento de compilar los shaders.

En las siguientes secciones se mostrarán los diagramas de clase de los prototipos utilizados, la explicación de a que corresponden dentro del contexto de la aplicación y una breve descripción de cómo funcionan. No se discutirán todos los métodos, solo aquellos que son relevantes.

4.1.1 Element

Element (figura 7) es el objeto que se utiliza para representar los elementos que conforman un modelo. Solo contiene 2 variables: un número para indicar el id del elemento y un booleano para indicar si dicho elemento esta seleccionado o no.

Existen 2 objetos que heredan de Element, y que representan elementos más concretos de una malla: Vertex y Polygon.

Vertex representa los vértices, y agrega 3 nuevas variables: un vector de coordenadas, una lista de polígonos y un vector normal. El vector de coordenadas es el vértice en sí. La lista corresponde a aquellos polígonos que contienen el vértice, y en la práctica sirven para calcular la normal del mismo. Esto último es exactamente lo que hace el método “calculateNormal”.

Polygon representa los polígonos de un modelo. Dentro de sus variables se encuentra el área, una lista de vértices, una lista de vecinos y un vector normal. La lista de vértices es en realidad lo que define el polígono y el orden en el que están guardados es importante para calcular la normal. La lista de vecinos corresponde a los polígonos adyacentes. Se define el método “calculateNormal”, el cual solo requiere de los vértices para calcular la normal, y un método “getAngles”, que calcula y retorna una lista con los ángulos internos del polígono.

Se puede notar que tanto Vertex como Polygon definen la variable “normal” y el método “getNormal”. Esto podría llevar a pensar que sería correcto ponerlos en Element.

La razón por la que no se hizo esto fue pensando que en una iteración posterior la aplicación podría soportar las mallas de poliedros. Esto implicaría agregar un nuevo tipo de Element: los poliedros. Pero estos no definen una normal y estarían heredando una propiedad que no les corresponde.

También puede llamar la atención que Polygon no tiene la variable “angles”, pero si el método “getAngles”. Este es un ejemplo que muestra que ciertas propiedades de los elementos se pueden calcular fácilmente y no es necesario que sean indicadas o guardadas de antemano. Expresado esto, sí está considerado agregar “angles” y otras propiedades a la lista de variables, pues pre calcular las propiedades puede ayudar a aumentar la eficiencia al momento de seleccionar o evaluar un modelo.

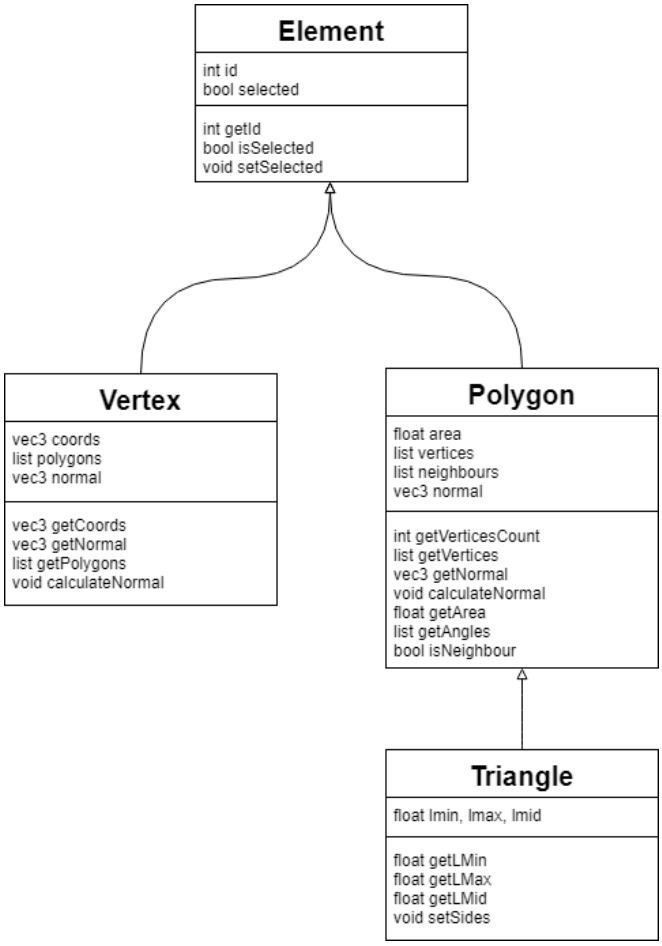

Figura 7 – Diagrama de clases de Element.

Finalmente, se tiene el objeto Triangle, que hereda de Polygon y define 3 nuevas variables que corresponden a las longitudes de sus lados.

Este objeto se define porque las mallas de triángulos son las más comunes al momento de crear modelos, y es necesario poder identificar cuando una malla es solo de triángulos. Además, debido a su amplio uso pueden existir propiedades que son exclusivas de este tipo de mallas.

La mayoría de las propiedades guardadas en todos estos objetos no son necesarias para dibujar un modelo, y existen exclusivamente para el cálculo de otras propiedades (por ejemplo, los vecinos de un polígono sirven para calcular el ángulo diedro), o directamente para poder ejecutar una selección o evaluación (por ejemplo, los ángulos internos de los polígonos). Lo único que en realidad se necesita al momento de dibujar el modelo, es la lista de vértices y las normales.

4.1.2 Model y RModel

El objeto Model (figura 8) corresponde a lo que hasta aquí se ha referido como modelo. También se ha mencionado modelo como sinónimo de una malla de polígonos, pero la forma en la que se presenta en el diagrama es que una malla de polígonos hereda de Model. Esto es porque un modelo puede estar definido de otras formas, en particular existen las nubes de puntos y las mallas de poliedros. Aunque en la aplicación solo existe una única forma de representación de modelos, el objeto Model fue pensado de tal manera que se pudiesen agregar nuevas formas en el futuro.

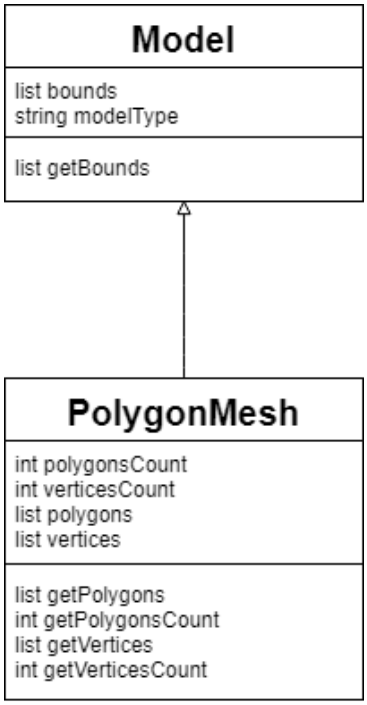

Figura 8 – Diagrama de clases de Model.

Model solo define dos variables que son sus límites (máxima y mínima posición en cada eje), y un string indicando que tipo de modelo es.

PolygonMesh es la representación particular del modelo como una malla de polígonos, y contiene una lista de polígonos y una lista de vértices. Los elementos de la lista corresponden a los objetos de tipo Element definidos en la sección anterior. Además, incluye dos valores que indican cual es la cantidad de polígonos y vértices de la malla.

El objeto RModel (figura 9) es quizá uno de los más grandes y complicados de entender. Su principal objetivo es calcular y entregar los datos necesarios a WebGL, en particular a los shaders, para poder dibujar el modelo en el canvas. Contiene 23 variables y 25 métodos. RModel siempre contiene una referencia al Model que se quiere dibujar. La mayoría de las otras variables corresponden a las matrices y vectores que necesitan los renderers para calcular la posición de un modelo.

Para inicializar un RModel, solo es necesario entregarle un Model, y llamar al método “loadData”. Este método llamara a otro método según el “modelType” del Model. Para la versión actual esto solo corresponde a “loadDataFromPolygonMesh”. Este método inicializa el resto de las variables según los datos contenidos en Model.

Se podría pensar que Model podría contener todos estos datos y funcionalidades perfectamente; sin embargo, se optó por dejar que Model sea una representación simple del modelo, que se mantenga fiel al archivo original que fue abierto. Otra razón es que WebGL solo se maneja con triángulos, líneas y puntos, así que, si el modelo está formado por, por ejemplo, cuadrados, la información de Model no es suficiente para dibujarlo. Finalmente, WebGL no funciona con los objetos creados en la aplicación, si no que necesita listas de números. Para dibujar triángulos, por ejemplo, necesita una lista en la que cada vértice es representado por 3 números seguidos dentro de la lista, y un triángulo es representado por 3 vertices seguidos.

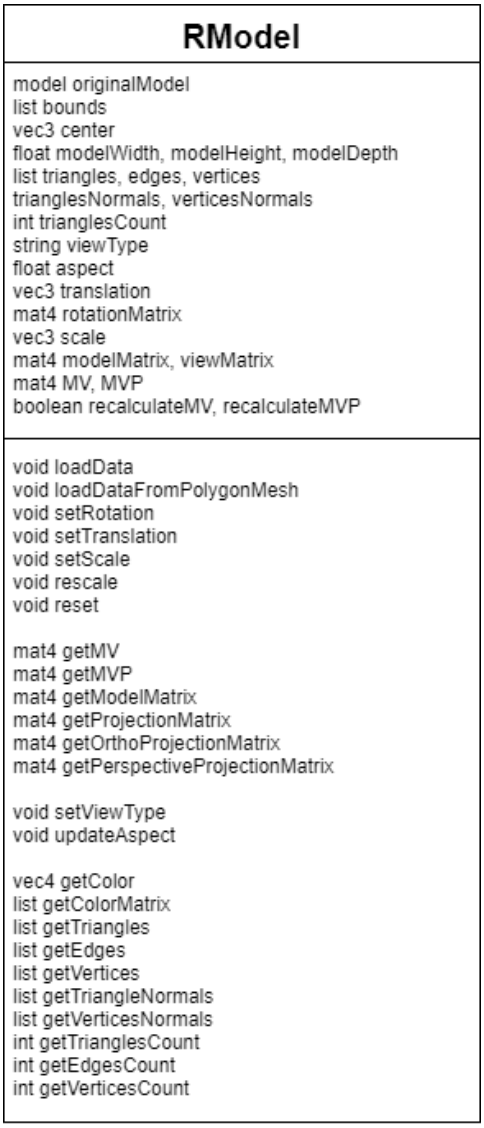

Figura 9 – Diagrama de clases de RModel.

Es por esto último que existe la variable “triangles”, que es una lista en RModel. Para calcularla lo que se hace es tomar los polígonos del Model, triangularlos, obtener los vértices de los triángulos generados, obtener las coordenadas de los vértices, y agregar las coordenadas a la lista.

La variable “edges”, que también es una lista, sigue un proceso similar. Model no maneja las aristas, por lo que es necesario tomar los polígonos, y sacar los pares de vértices que correspondan para representar una arista. Por cada par de vértices se obtienen las coordenadas y se agregan a la lista. El orden de los vértices en Model es fundamental para triangular bien los polígonos y determinar bien las aristas.

La variable “vertices”, que también es una lista, es la más simple de determinar, pues solo corresponde a sacar las coordenadas de los vértices y agregarlos a la lista.

Las variables “trianglesNormals” y “verticesNormals” son importantes para determinar cómo se verá el modelo. En modelos de iluminación las normales sirven para determinar en qué ángulo llega la luz y como eso afecta al color de una cara en el modelo. Determinarlas no es complicado, el objeto Model ya las tiene incluidas, pero si hay que tener en cuenta un par de detalles importantes.

Primero, es que WebGL solo se entiende con normales de vértices. Segundo, hay que ser consistentes al entregar las normales, hay que pasar exactamente una normal por vértice.

Por lo anterior, una forma de revisar la consistencia de “trianglesNormals” y “verticesNormals” es asegurarse que estas listas sean del mismo tamaño que “triangles”.

Para calcular “trianglesNormals” lo que se hace es que durante el proceso de triangular los polígonos de Model, al agregar un triángulo a la lista de “triangles”, se pide la normal del polígono que estaba siendo triangulado, se extraen sus coordenadas, y se agrega tres veces a la lista. Esto es, una vez por cada vértice del triángulo que se está agregando.

Para “verticesNormals” se ocupa el mismo proceso, y se aprovecha del hecho que la técnica de triangulación utilizada para los polígonos no genera nuevos vértices. Esto permite que en vez de agregar tres veces la normal del polígono a la lista, se agrega una sola vez la normal de cada vértice que este formando el triángulo que se está agregando.

Un detalle importante respecto a “verticesNormals”, es que, a pesar de que se podría entender por su nombre, no se utiliza con la variable “vertices”, sino que se usa con “triangles”. Esto es porque “vertices” son literalmente puntos en el espacio y no necesitan normales para ser dibujados, mientras que “verticesNormals” es para una estrategia de iluminación de caras.

RModel también guarda la matriz de rotación, el vector de traslación y el vector de escala. Sin embargo, no es quien los manipula y estos en realidad deben ser actualizados por los métodos “setRotation”, “setTranslation” y “setScale”. Cabe destacar eso sí, que a pesar de que todas estas variables se inicializan de forma que el modelo no sea afectado (es decir, aparezca en su posición por defecto), internamente siempre se le aplica una traslación extra para centrarlo en la posición (0, 0, 0). Esto es porque en la aplicación, la cámara (que corresponde a la variable “viewMatrix”) siempre está en la misma posición (0, 0, 2*w), donde w corresponde a la variable “modelDepth”. Es aquí donde entra en juego la variable “center”, que corresponde al centro real del modelo: si el centro corresponde a la posición (x, y, z), siempre se aplicará una traslación de (-x, -y, -z) antes de aplicar el resto de los movimientos.

Se podría argumentar que dejar el modelo en su posición original y mover la cámara es una mejor solución. El motivo por lo que no se aplica de esta forma es que, en particular, las rotaciones siempre ocurren respecto al origen, y no respecto al centro del modelo. La forma de rotar el modelo en este caso entonces, igual implicaría moverlo al origen, rotarlo, y devolverlo a su posición original.

Los shaders de WebGL necesitan solo una matriz para dibujar el modelo: “Model View Projection” o “MVP”. El resto de las matrices son las necesarias para calcular esta matriz. La variable “modelMatrix” es una matriz que representa todos los cambios que sufre el modelo y corresponde a la multiplicación de las matrices de escala, traslación y rotación. La variable “viewMatrix” como se indicó anteriormente corresponde a la cámara, se calcula entregándole la posición de la cámara, un punto al que debe mirar y un vector (que indica hacia donde esta “arriba” en el espacio) al método “lookAt” de la librería glMatrix. La variable “MV” corresponde a la multiplicación de “modelMatrix” con “viewMatrix”. Finalmente “MVP” se calcula multiplicando “MV” con la matriz de proyección. Esta no está definida dentro de las variables y se obtiene a través del método “getProjectionMatrix”. La matriz de proyección es la que asegura que se pueda ver la profundidad y puede ser de dos tipos: ortogonal o perspectiva. El tipo de proyección está definido en la variable “viewType”.

Cada vez que hay un cambio en traslación, rotación o escala, es necesario calcular las matrices “MV” y “MVP” de nuevo. Para esto se fijan como indicadores las variables “recalculateMV” y “recalculateMVP”. Los métodos “getMV” y “getMVP” recalculan estas matrices si es que los indicadores correspondientes están marcados como true. Si no simplemente entregan el valor que existía anteriormente.

4.1.3 ModelLoadStrategy

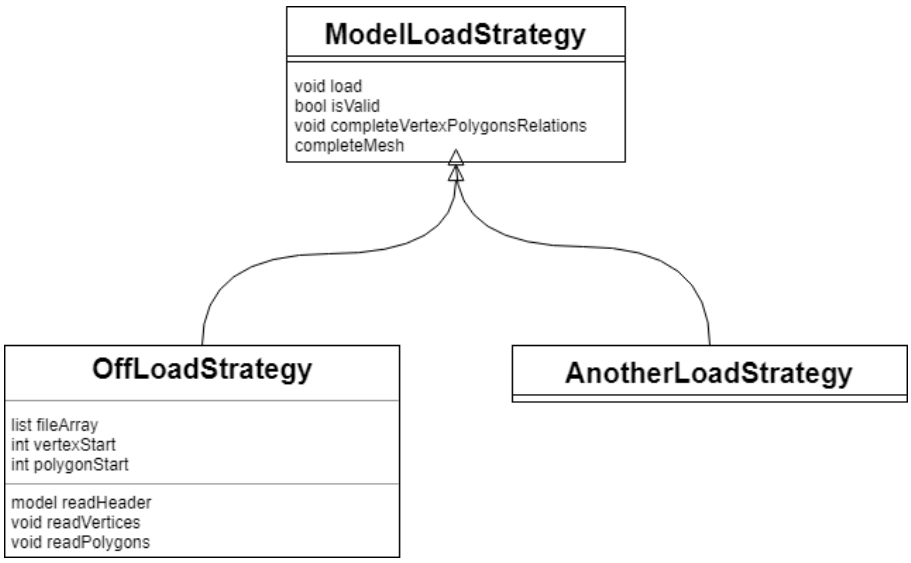

Figura 10 – Diagrama de clases de ModelLoadStrategy.

El objeto ModelLoadStrategy (figura 10) es el encargado de leer los archivos de los modelos, y crear un objeto Model a partir de ellos. Para este objeto se usa el patrón de diseño “Strategy Pattern”. En este contexto ModelLoadStrategy es la interfaz que debe ser implementada por las estrategias de carga concretas. Es necesario notar que JavaScript no maneja el concepto de interfaces, así que en realidad esto es una convención dentro de la aplicación. ModelLoadStrategy no tiene variables, y los métodos “load” y “isValid” no hacen nada, pues deben ser implementados por las estrategias específicas.

La única estrategia de carga implementada fue OffLoadStrategy, que se encarga de abrir archivos en formato “off”. JavaScript no tiene métodos de lectura de archivos que permitan leer línea por línea, sino que solo permite recuperar el texto entero. Para simular una lectura por línea, es que existe la variable “fileArray”, que corresponde a una lista de Strings, donde posición corresponde a una línea del archivo original.

El método “readHeader” crea un objeto de tipo Model. Luego los métodos “readVertices” y “readPolygons” leen el resto del archivo y llenan las listas del objeto de tipo Model creado con objetos de tipo Vertex y Polygon.

Las variables “vertexStart” y “polygonStart” indican a partir de que línea del archivo están especificados los vértices y los polígonos. Estos no son posibles de determinar desde un principio. “vertexStart” es determinado al final del método “readHeader” y “polygonStart” es determinado al final del método “readVertices”.

La implementación del método “load” que hace particularmente OffLoadStrategy, son llamadas a los métodos “readHeader”, “readVertices y “readPolygons” en ese orden.

Antes de ejecutar el método “load”, se ejecuta “isValid”, y después de ejecutar “load” se ejecuta “completeMesh”. El método “isValid” se utiliza para revisar la consistencia del archivo y evitar que tenga algún problema durante la carga. El método “completeMesh” es para terminar de calcular ciertas propiedades del modelo, y se define en ModelLoadStrategy porque es independiente del tipo de archivo abierto. El método “completeVertexPolygonsRelations” es llamado dentro de “completeMesh”.

AnotherLoadStrategy está en el diagrama de clases solo para representar que se pueden agregar nuevas estrategias de carga de archivos extendiendo a ModelLoadStrategy e implementando los métodos necesarios. La forma de decidir qué estrategia debe ejecutarse es revisando la extensión del archivo subido.

4.1.4 Renderer

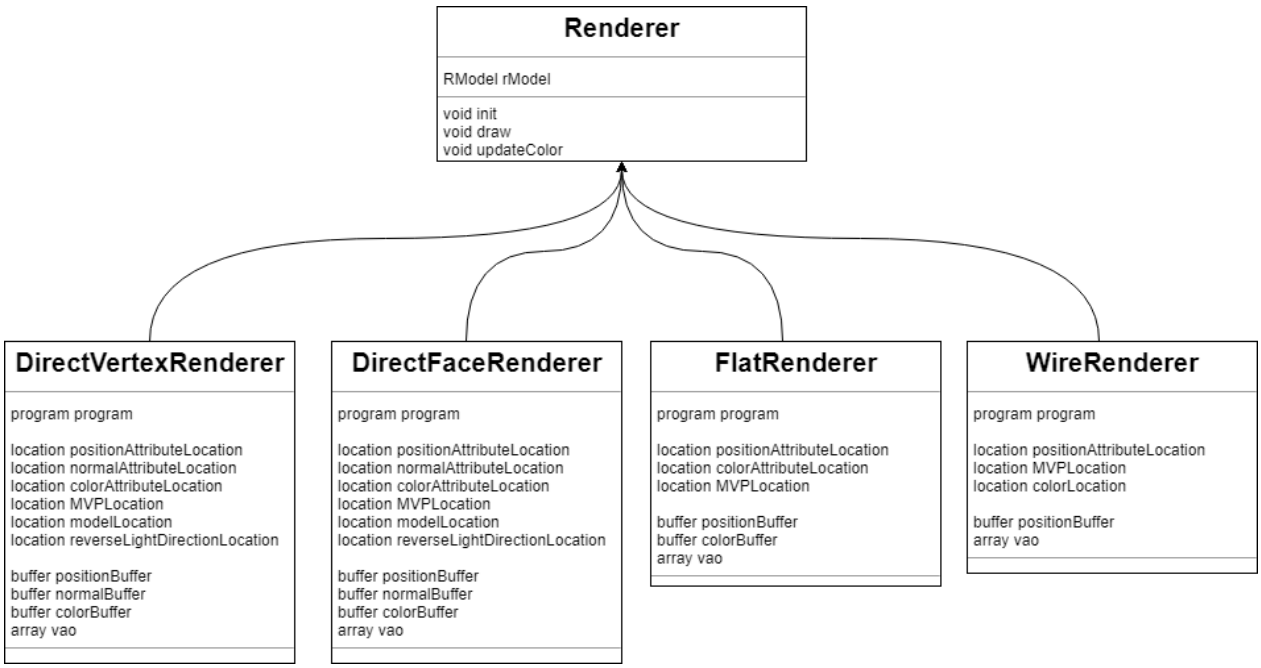

Figura 11 – Diagrama de clases de Renderer.

El objeto Renderer (figura 11) es el encargado de dibujar el modelo en el canvas, y es donde esta toda la lógica de manipulación de WebGL. Recibe como variable un RModel, que como se explicó anteriormente es el objeto que puede proveer a los shaders de la información que necesita.

Renderer por sí solo no hace nada, solo indica los métodos “init”, “draw” y “updateColor”, pero dichos métodos no están definidos y deben ser implementados por sus objetos hijos. Un renderer, lo que hace es compilar y enviar información a los shaders, los cuales determinan que pixeles pintar dentro del canvas y de qué color. La forma de elegir el color depende de la estrategia de iluminación seleccionada, y es allí donde entran en juegos los distintos tipos de renderers.

Un renderer no se utiliza necesariamente para dibujar el modelo. Se pueden dibujar otras cosas que son propiedades del modelo. Por esa razón es que se hizo la separación entre lo que se llamó “main renderers” y “secondary renderers”. Los main renderers son aquellos que dibujan el modelo, y los secondary renderers son aquellos que dibujan propiedades del modelo. Esta separación se realizó a nivel conceptual, pero a nivel de implementación todos los renderers heredan de la clase Renderer.

Dibujar un modelo de 2 o más formas distintas al mismo tiempo no tiene sentido (pues el último dibujo taparía a los anteriores). Así que se definió que en la aplicación solo puede existir solo un main renderer a la vez. Por otro lado, las propiedades suelen ser cosas distintas, y no hay problema en dibujar varias al mismo tiempo, ni que estén encima del dibujo del modelo. Por esto se definió que en la aplicación puede existir más de un secondary renderer al mismo tiempo, siempre y cuando sean distintos.

En el diagrama de clases de la figura 11, DirectVertexRenderer, DirectFaceRenderer y FlatRenderer corresponden a main renderers. Mientras que WireRenderer corresponde a un secondary renderer. También existen los secondary renderers VCloudRenderer, FNormalsRenderer y VNormalsRenderer, encargados de mostrar la nube de puntos, las normales de las caras, y las normales de los vertices respectivamente. Como todos funcionan de la misma manera, se explicará los renderers de forma general.

Lo primero que llama la atención es que todos tienen 3 tipos de variables, que en el diagrama se expresaron como “program”, “location” y “buffer/array”. La variable “program” es el compilado de un programa llamado shader. Como ya se mencionó, este programa es el encargado final de pintar los pixeles del canvas de los colores que correspondan. Además, es la parte de la aplicación que se ejecuta en la GPU.

Los shaders define ciertas variables, y el renderer después se tiene que encargar de pasar información a estos. Como el programa solo existe en la GPU, JavaScript no tiene una forma directa de pasar información a las variables. Para esto es que se definen las variables de tipo “location”. Estas variables apuntan a la dirección de memoria de la GPU que contiene las variables del shader.

También están las variables de tipo “buffer”, que son las que se llenan con la información que necesita el shader para dibujar el modelo. Esta información es la que se obtiene del RModel y normalmente son las listas de triángulos, normales o colores

Existen dos tipos de variables location: “attribute” y “uniform”. Los “attribute” se usan para sacar información de algún buffer. Es necesario que estén asociadas a un buffer específico, y explicitar como se debe sacar la información de dicho buffer. Por otro lado, los “uniform” son esencialmente variables globales del shader y se les puede asignar información directamente. La variable que recibe la matriz “MVP” es un “uniform”.

Finalmente, está la variable “vao”, que corresponde a un array. Esta variable en estricto rigor no es necesaria, pero sirve para guardar referencias a todos los buffers y ahorrar procesamiento futuro. Esta variable está definida por WebGL y evita el proceso de tener que crear una estructura propia para guardar la información de los buffers.

4.1.5 SelectionStrategy y EvaluationStrategy

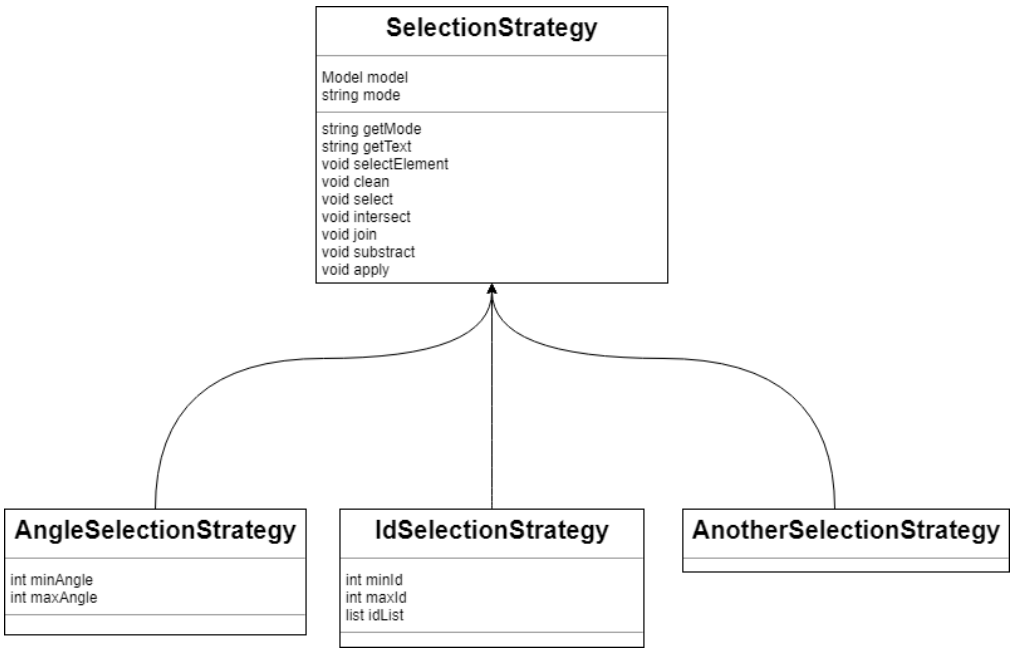

Figura 12 – Diagrama de clases de SelectionStrategy.

SelectionStrategy (figura 12) y EvaluationStrategy (figura 13) son los objetos que se utilizan para analizar un modelo. Ambos siguen el patrón de diseño “Strategy Pattern”.

Para la selección, SelectionStrategy actúa como una interfaz. Sus hijos tienen la responsabilidad de implementar los métodos “selectElement” y “getText”. El resto de los métodos son comunes entre todos, pero dependen de “selectElement”, haciendo que SelectionStrategy no pueda funcionar por sí solo. La aplicación está diseñada para aplicar varias selecciones seguidas, sin restricción al tipo de selección.

Al realizar una selección, la estrategia seleccionada recibe una referencia al Modelo que esté abierto en ese momento. Luego ejecuta el método “apply”, que según el modo de selección (correspondiente a la variable “mode”) ejecuta alguno de los métodos “select”, “intersect”, “join” o “substract”. Todos estos métodos ocupan “selectElement” de alguna forma.

El método “selectElement” lo que hace es recibir un objeto del tipo Element (en el caso de la aplicación este es particularmente un Polygon), y marcarlo como seleccionado si pasa cierto criterio, o desmarcarlo en caso de que no pase el criterio.

El modo de selección es especificado por el usuario, pero los métodos “intersect”, “join” y “substract” dependen de que exista una selección limpia realizada con el método “select” previamente. Lo que hace “select” es recorrer los polígonos del modelo y ejecutar “selectElement” sobre cada uno de ellos.

El método “intersect” también ejecuta “selectElement”, pero solo sobre los polígonos ya seleccionados, esto provoca que efectivamente el elemento se desmarca si no pasa el criterio definido. Esto produce un efecto de intersección con respecto a la selección anterior. Los métodos “join” y “substract” aplican estrategias similares a “intersect”.

La aplicación implementa tres estrategias de selección: AngleSelectionStrategy, AreaSelectionStrategy y IdSelectionStrategy. El primero recibe las variables “minAngle” y “maxAngle” y su criterio de selección es cualquier polígono que tenga algun ángulo interno entre esos dos valores. El segundo funciona de manera similar, pero con las variables “minArea” y “maxArea”. El tercero puede recibir “minId y maxId” y aplicarse de la misma forma que los anteriores, o puede recibir una lista de identificadores, y seleccionar los polígonos cuyo identificador este contenido en la lista.

La selección afecta directamente al modelo, pues guarda sus resultados en los mismos polígonos. Esto afecta a RModel, porque al generar la matriz de colores que se le entrega al renderer, fija un color distinto para los polígonos que están seleccionados.

AnotherSelectionStrategy se muestra en el diagrama para representar que se pueden definir nuevas estrategias de selección extendiendo a SelectionStrategy.

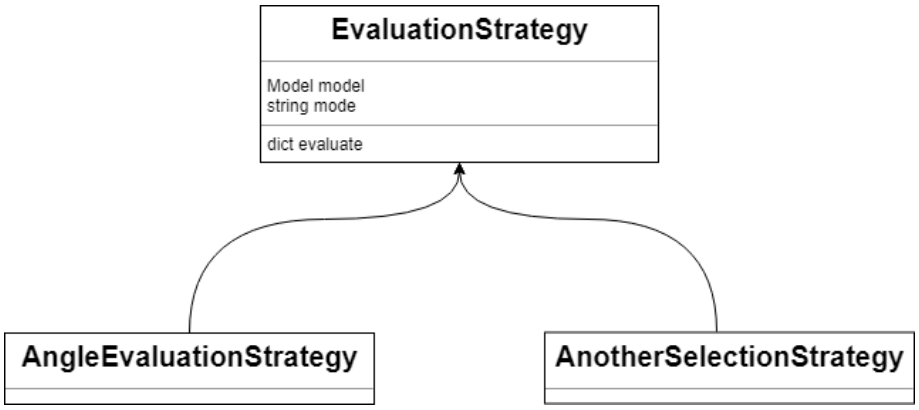

Figura 13 – Diagrama de clases de EvaluationStrategy.

Para la evaluación, EvaluationStrategy es quien actúa como la interfaz. Sus hijos solo deben implementar el método “evaluate”. Aquí la variable “model” corresponde al modelo que está abierto en el momento de la evaluación y la variable “mode” corresponde a un String para indicar si la evaluación se realiza sobre todo el modelo, o solo sobre los polígonos seleccionados (en caso de que exista una selección).

El método “evaluate” al ejecutarse genera una estructura de tipo diccionario con información del Model relacionada a la estrategia seleccionada. Por ejemplo, en el caso de AngleEvaluationStrategy, se determinan cuáles son los ángulos mínimos y máximos presentes en el modelo, y una lista de todos los ángulos encontrados (que sirven para generar un histograma en la vista). A diferencia de la selección no puede haber dos evaluaciones simultáneas. Si se ejecuta una evaluación cuando ya existía otra, la anterior se elimina.

El modelo no es afectado de ninguna manera por las evaluaciones. El resultado de las evaluaciones en la interfaz es mostrado en un canvas independiente al canvas del modelo.

Igual que antes, AnotherSelectionStrategy es mostrado en el diagrama para representar que es simple definir nuevas estrategias de evaluación.

4.1.6 Scalator, Translator y Rotator

Anteriormente se indicó que RModel no es el encargado de manipular los vectores de escala, traslación y rotación. Esta funcionalidad se agregó en tres objetos simples, llamados Scalator, Translator y Rotator (figura 14).

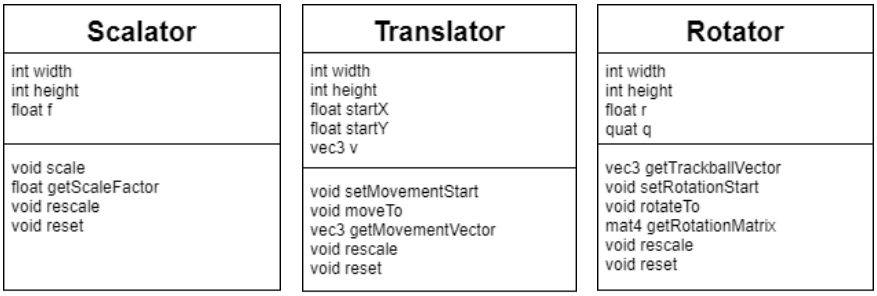

Figura 14 – Diagramas de clases de Scalator, Translator y Rotator.

Estos se inicializan cuando se crea un modelo, y guardan el estado de posición del modelo. Cuando el mouse interactúa con el canvas, estos son los objetos que realizan los cambios sobre estas propiedades, y después le pasa los valores correspondientes a RModel.

Se puede observar que todos tienen la variable “width” y “height”. Esto corresponde al tamaño del canvas, y es necesario para que el movimiento en el canvas sea preciso. Si el tamaño del canvas cambia se llama al método “reScale”, el cual actualiza estos valores. También todos presentan el método “reset”, este reinicia los valores a aquellos antes de que el modelo sufriera modificaciones.

Scalator maneja una variable “f”, que se refiere al factor de escala. Este se controla con la rueda del mouse. Si se hace scroll hacia arriba, “f” sube de valor, y si se hace scroll hacia abajo, “f” baja de valor.

Translator maneja la variable v, que es el vector indicando cuanto se quiere trasladar en cada eje. Si se toma el modelo con el click derecho y se mueve el modelo en la dirección horizontal (o vertical), se puede detectar la posición de inicio y fin. Con esto se calcula cuánto fue el desplazamiento y se aplica al vector “v”.

Rotator es un poco más complejo, pues trabaja con ángulos, y transformar movimientos del mouse sobre la pantalla en rotaciones no es simple. Solo se mencionará que guarda la información de rotación en un vector. Una mejor explicación de cómo esta implementado este objeto será dará en la sección de funcionamiento.

4.2 Interfaz

Para el diseño de interfaz de la aplicación se contó con la ayuda de un diseñador externo. Una vez terminada la etapa de análisis, se le entregó al diseñador la lista de requerimientos para iniciar este proceso. En este contexto a veces se actuó como cliente haciendo observaciones y críticas a los diseños presentados, y otras se actuó del lado del diseñador, presentando el proyecto a potenciales usuarios, y recibiendo observaciones de ellos.

4.2.1 Distribución Espacial

Para determinar la distribución espacial de los elementos en la pantalla, se presentaron propuestas a través de iteraciones de wireframes y reuniones de validación.

El diseñador fue muy detallado y preciso en las consultas que hacía, logrando así que la segunda versión del wireframe enviada se transformara en la versión definitiva. Las diferencias entre la primera y la segunda versión fueron más que nada de vocabulario, y se debieron más que nada a la falta de experiencia del diseñador en computación gráfica al no entender bien algún concepto.

El diseñador también indico que se inspiró en softwares de manejo de fotografías, y de sistemas CAD. Esto pues varias de las acciones que ofrece la aplicación son similares, y estas aplicaciones tienen lineamientos de diseño y usabilidad bastante específicos que se llevan reafirmando hace años. En la figura 15 presenta la versión final del wireframe.

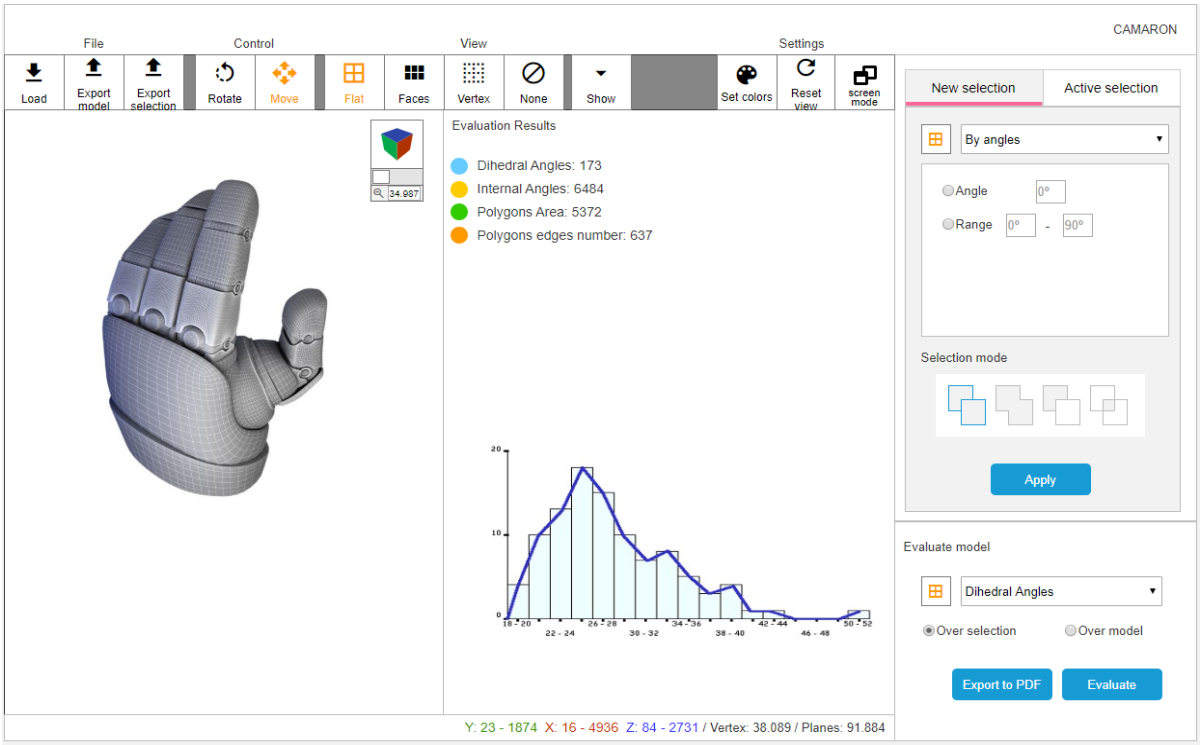

Figura 15 – Versión Final Wireframe.

4.2.2 Aspecto (Look and Feel)

Una vez aprobada la distribución espacial corresponde la tarea de darle identidad a la aplicación. Esta es la parte más importante de la etapa de diseño, pues como se indicó en la sección anterior, la distribución espacial estaba en particular bastante estandarizada. Dado esto, el diseño estético es el que define como los usuarios perciben la aplicación, y que tanta confianza les dará utilizarla.

4.2.2.1 Íconos

Para la mayoría de las acciones que uno puede realizar en casi cualquier software, existe una estandarización de íconos. Por ejemplo, “play” es un ícono bien conocido y utilizado en todos los reproductores de videos y música.

Sin embargo, todas las acciones que se pueden realizar en un visualizador de mallas no tienen íconos estandarizados, y cada visualizador tiene su propia forma de expresarlos. Es importante, asegurarse que no hay ambigüedad en lo que los íconos deben expresar.

En este tema, jugo a favor la poca experiencia del diseñador en computación gráfica. Porque hubo que explicar las cosas a nivel fundamental, permitiendo captar la esencia de lo que se quería hacer con mayor precisión.

Se expondrá aquí el caso de los renderers. En la aplicación se definieron dos tipos de renderers: main y secondary. Los main renderers son aquellos que indican con qué modelo de iluminación se dibujara el modelo y solo puede haber uno seleccionado a la vez. Los secondary renderers son aquellos que dibujan algo sobre el modelo (como wireframe de un modelo y vértices), y pueden seleccionarse varios a la vez.

En la figura 16 se puede observar el mismo cubo dos veces, pero aplicando un main renderer distinto cada vez. Uno hace que la iluminación se calcule con las normales de las caras, y otro hace que la iluminación se calcule con las normales de los vértices.

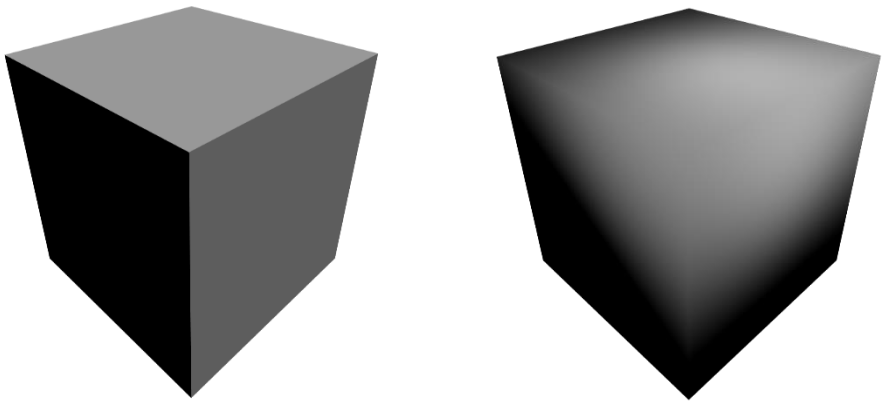

Figura 16 – Cubo con iluminación por caras (izquierda) y Cubo con iluminación por vértices (derecha).

En la aplicación están presentes los íconos para cambiar entre main renderers en cualquier momento. Esta funcionalidad en particular también está presente en el visualizador MeshLab y los íconos para esto son los dos de la derecha que se pueden observar en la Figura 17.

Como se observa, MeshLab ocupa cilindros para representar sus íconos de renderers, pero dado el tamaño que tienen, resulta difícil apreciar la diferencia. Además, tiene los botones para mostrar el wireframe y los vértices de un modelo, y están ubicados directamente a la izquierda de los otros botones.

Figura 17 – Íconos de Renderers en MeshLab

MeshLab no hace diferenciación entre tipos de renderers. Sin embargo, en la práctica trata los últimos dos botones como mutuamente excluyentes (al seleccionar uno se deselecciona el otro), pero los primeros dos como botones independientes. Visualmente no transmiten ese comportamiento, pues los cuatro botones pertenecen a la misma sección. Este es un ejemplo de mal diseño de interfaz.

Para el diseño de la aplicación, se usó el cubo como la figura base, pues es más simple a la vista, y el efecto que causa calcular la iluminación sobre los vertices en un cubo, como se aprecia en la figura 16, es extraño y bien conocido.

Además, se separó los main renderers de los secondary renderers en secciones aparte, para indicar que su comportamiento, aunque similar, es independiente. Estos íconos se pueden observar en la figura 18.

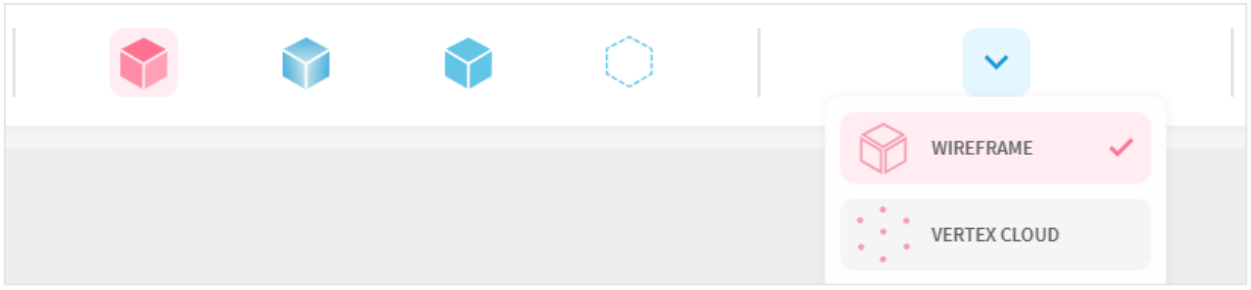

Figura 18 – Íconos de Renderers en la aplicación.

Aquí se logra apreciar hay 2 main renderers más aparte de los 2 mostrados en la figura 16, que corresponden a ausencia de iluminación (el modelo se ve plano), y que el modelo no se dibuja (útil para cuando solo se quiere ver un secondary renderer).

Por su parte, los secondary renderers ahora están en una lista, quedando relegados a segundo plano. Al presionar uno, queda en evidencia que se puede seleccionar más de uno a la vez a través del acto de dejar una marca a la derecha del nombre.

También se procuró que los íconos fuesen de un tamaño tal que resultase cómodo presionarlos, acorde con el principio “tamaño y espacio” expuesto en la sección 2.2.4.

4.2.2.2 Paleta de Colores

La elección de la paleta de colores es algo que se suele ignorar. Elegir una paleta de colores antes de empezar a diseñar ayuda a dirigir el resto del diseño, y a mantener la consistencia al agregar nuevos elementos.

Se utilizaron colores magenta, cian y morado en tonalidades pasteles, junto con blanco y gris para los fondos. Todos estos colores se pueden apreciar en la figura 12.

Acá es donde se cometió un pequeño error: aunque la paleta de colores contrasta bien, algunos tipos de daltonismo confunden, y les cuesta distinguir algunas tonalidades de cian contrastado con blanco. Este es un contraste que se usó mucho en la aplicación. Este problema no se alcanzó a corregir, pero la solución ya está determinada y se aplicará en un futuro cercano.

Antes de indicar dicha solución es necesario exponer otro detalle: gran parte del público objetivo trabaja dentro del campo de la computación, y es bien sabido que la gente que trabaja en esta área, especialmente desarrollando, prefiere temas oscuros. Por lo mismo, está considerado para una iteración posterior del proyecto, implementar un tema oscuro a la aplicación, que pueda ser cambiado a voluntad. Dicho tema se determinará con más cuidado, sirviendo también como solución para los usuarios daltónicos.

4.2.2.3 Diseño Final

El diseño final de la aplicación es minimalista. Considerando la cantidad de botones que hay disponibles en la pantalla, se decidió evitar sobrecargar al usuario con mucho texto. Por lo anterior es que se dispuso tanto cuidado a los íconos, pues tienen que ser lo suficientemente claros para funcionar sin texto.

El sistema se basa en la premisa de invitar al usuario a investigar. Para un usuario experimentado en computación gráfica, los íconos son lo suficientemente claros para que se imagine con precisión que funcionalidad ofrecen. Para un usuario no familiarizado, los íconos pueden ser extraños, pero sus efectos se pueden ver inmediatamente al presionarlos; esto además ayuda a la memorización.

En el uso de la aplicación no existe penalización por equivocarse en alguna acción. Todas las acciones están pensadas para poder deshacerse. Esto es así, a pesar de que no se cuenta con botones de undo/redo. Esto facilita al usuario a aprender sin miedo y se adecua el principio “tolerancia al error” expuesto en la sección 2.2.4.

Se puede decir que no hay mucha diferencia entre un usuario avanzado y uno novato. A los dos se les presenta la misma interfaz, y el usuario novato puede alcanzar rápidamente un entendimiento sobre todas las funcionalidades de la aplicación.

Es necesario aclarar que todas las decisiones tomadas en esta sección son teóricas. El diseño se creó para ser así, pero en la práctica debería ser evaluado con usuarios reales para finalmente determinar si los resultados esperados en virtud del diseño se lograron.

Dicho esto, el uso de las técnicas de HCI y principios de diseño igual aseguran un nivel de calidad y usabilidad, independiente de si se realizan evaluaciones o no.

En la figura 19 se muestra el diseño final de la aplicación. Esta imagen no corresponde a la implementación en HTML del diseño, sino que fue generada por el diseñador en un software especializado para diseño de interfaces.

Se puede ver cierta discrepancia con respecto a la figura 18, y es que la flecha de los secondary renderers está en la misma sección que los main renderers. Este es un problema que se solucionó en la implementación final. Esta situación ejemplifica que incluso después de la etapa de diseño pueden existir correcciones para mejorar la usabilidad, y que incluso un experto puede pasar detalles por alto.

También se pueden ver colores no mencionados en la paleta de colores (amarillo y verde), esto es porque corresponden a colores alternativos en caso de que se necesiten más, pero no son utilizados para los elementos de la interfaz.

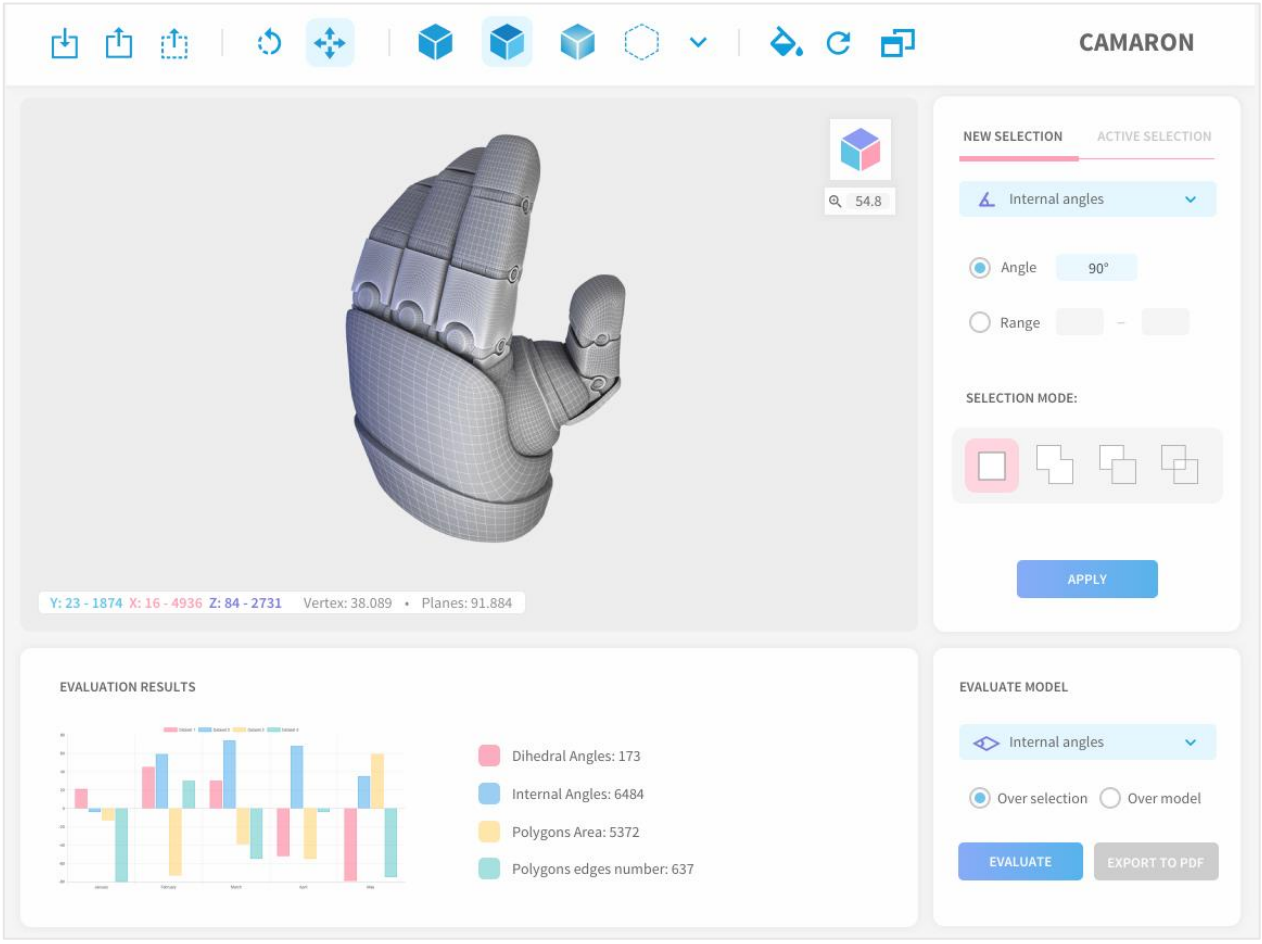

Figura 19 – Diseño Final de la Aplicación

4.2.3 HTML del Diseño

El trabajo final del diseñador consistió en entregar el HTML del proyecto, asegurándose que fuese fiel al diseño final presentado. En la figura 20 se logra observar la aplicación en su estado final. Se puede notar que varios de los elementos que se ven en el diseño, ahora están ausentes.

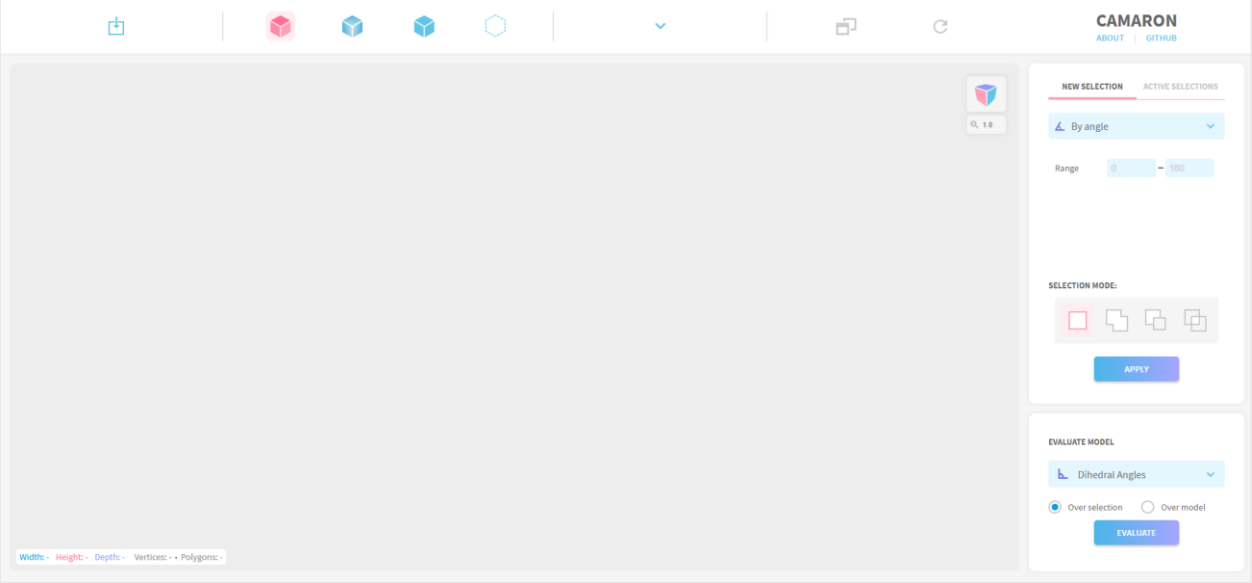

Figura 20 – HTML final de la aplicación

Estos elementos pueden estar ausentes por 2 razones:

1. El elemento está oculto y no aparece hasta que ocurre alguna acción específica.
2. Se determinó que la experiencia no era clara y se reemplazó, esto basado en opiniones de potenciales usuarios.

Dentro de la primera razón se tiene, por ejemplo, toda la sección que en la figura 5 se ve como un cuadro con un gráfico. Esto corresponde a la sección donde se despliegan los resultados de una evaluación. Pero si no existe una evaluación, el cuadro solo ocupa espacio.

En la segunda razón se tienen específicamente los botones de movimiento, que se pueden observar en la figura 21. Estos eran para indicar que el modelo estaba en modo “rotación” o “traslación”, y esencialmente cambiaban cual era la acción del click sobre el canvas donde se dibuja el modelo.

Figura 21 – Botones de movimiento

En una reunión con potenciales usuarios se determinó que al mover un modelo se suele cambiar entre estas 2 acciones muy rápido, y tener que ir a buscar el botón cada vez era molesto, pues requería mucho esfuerzo físico (lo que va en contra del principio "bajo esfuerzo físico" de los mostrados en la sección 2.2.4). Por lo mismo los botones fueron eliminados, y en la aplicación las acciones de rotación y traslación se conectaron con los botones del mouse izquierdo y derecho respectivamente.

Considerando también que el zoom está ligado a la rueda del mouse, el cambio permitió agilizar la manipulación del modelo, porque permite ejecutar todas las acciones de movimiento sin salir del canvas.

Otro detalle importante por destacar es que cuando se cargan, seleccionan o evalúan modelos muy grandes (del orden de los 800.000 polígonos), la aplicación se demora en su ejecución. Este tiempo depende de las características del computador donde se ejecute la aplicación. Es necesario indicarle al usuario que la aplicación está trabajando, por lo que a esta versión fue necesario agregar un indicador de carga. Este se puede observar en la figura 22.

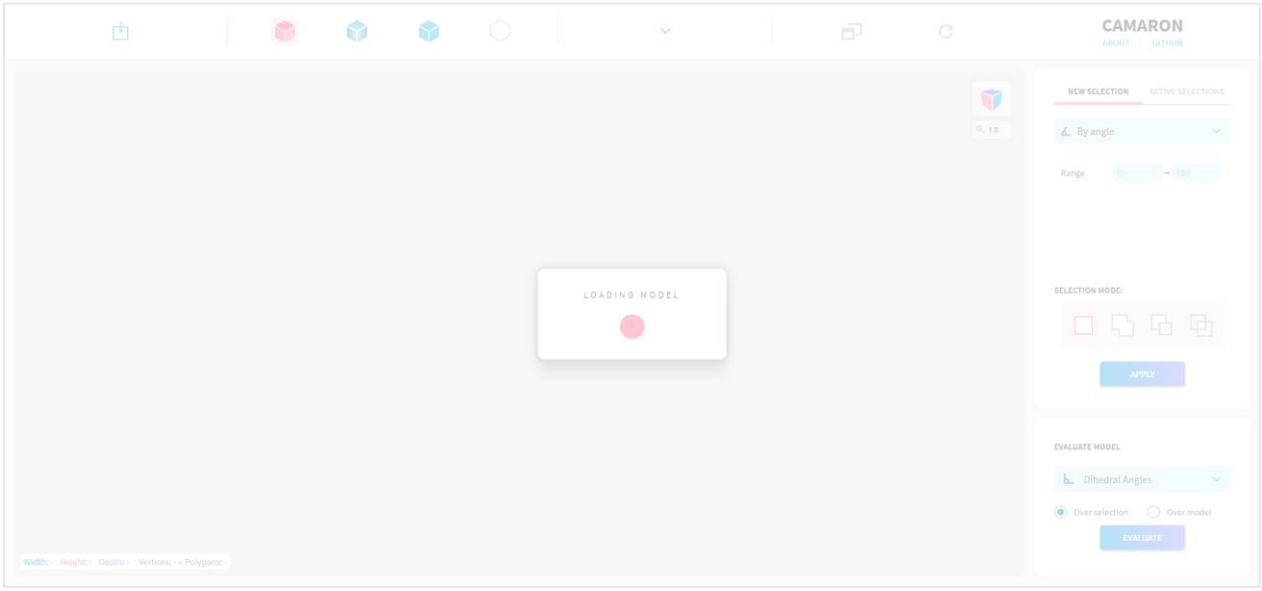

Figura 22 – Indicador de que la aplicación está trabajando.
En este caso particular en la carga de un modelo.

Además, se realizaron varios arreglos menores, como poner enlaces al repositorio debajo del título, o agregar estilo visual específico para botones deshabilitados. Los elementos e interacciones discutidos en esta sección corresponden en realidad a los que eran necesarios para explicar todos los aspectos del diseño.

El resto de los elementos en la vista se discutirán en detalle en la siguiente sección (Implementación). Esto pues desde el punto de vista de diseño, todos siguen los mismos lineamientos ya discutidos. Lo que falta indicar, tanto para los elementos ya mencionados como los que no, es su funcionalidad y como funciona todo a nivel de código.

5 Implementación

La etapa de implementación se refiere a la programación de la aplicación en sí, tomando en consideración los resultados de la etapa de diseño.

5.1 Funcionamiento General

En esta sección, se explica de manera general como están implementadas las distintas funcionalidades de la aplicación, y como se presentan estas en la interfaz. De ser necesario se muestran ciertas porciones de código.

5.1.1 Inicialización de la aplicación

Para inicializar la aplicación, solo basta con abrir el archivo “camaron.html” presente en la raíz del proyecto. Al abrir el archivo, se cargan todos los archivos JavaScript necesarios, en particular los archivos “camaron.js” y “sitio.js”, que pueden ser encontrados en el directorio “/js”.

“sitio.js” es el que maneja todas las animaciones e interacciones de la interfaz, y fue entregado por el diseñador al final de la etapa de diseño. Es el único en toda la aplicación que está escrito usando JQuery.

“camaron.js” es el que inicializa y maneja toda la lógica de la aplicación. Define las funciones para abrir los modelos, inicializar los renderers, aplicar selecciones, aplicar evaluaciones y dibujar el modelo, entre otras. Además de esto hace las uniones entre estas funciones y los elementos de la interfaz.

El punto más importante de la inicialización es revisar que WebGL está disponible, pues sin este la aplicación no funciona. Esto se realiza al inicio del archivo “camaron.js”. Para revisar que esté disponible, se debe solicitar el canvas en el que se dibujará el modelo. Luego a este canvas se le asigna el contexto de dibujo “webgl2”. El código para esto se ve de la siguiente forma:

```
var canvas = document.getElementById("glCanvas");

var gl = canvas.getContext("webgl2");
if (!gl) {alert("No WebGL");}
```

Como se observa, si WebGL no está disponible, la variable “gl” queda vacía y la aplicación no funcionará. Si WebGL está disponible, la variable “gl” queda asignada con un objeto del tipo “WebGL2RenderingContext”. Este objeto contiene todos los métodos que necesitan los renderers para comunicarse con la GPU y dibujar en el canvas.

5.1.2 Carga de Modelos desde archivos

La carga de modelos en la interfaz se realiza haciendo click sobre el botón de “Importar” (figura 23). Dicho botón está asociado en el HTML a un elemento input de tipo file. Por lo mismo, al hacer click sobre él, se abre la selección de archivos por defecto del sistema operativo que se esté utilizando.

Figura 23 – Botón de importación de modelos.

Seleccionar el archivo gatilla un evento “change”, provocando que en “camaron.js” se ejecute la función que carga el modelo. Dicha función lee el archivo, crea una lista en que cada elemento de esta corresponde a una línea del archivo, y con esa lista crea un objeto ModelLoadStrategy cuyo tipo depende de la extensión del archivo. Luego procede a verificar que el archivo está bien estructurado con el método “isValid”, para finalmente generar un objeto Model con el método “load”.

Una vez que el modelo existe se ejecuta la siguiente porción de código:

```
rModel = new RModel(model);
rModel.loadData();
changeViewType();
setMainRenderer();
setSecondaryRenderers();
rotator = new Rotator();
translator = new Translator();
scalator = new Scalator();
updateInfo();
draw();
enable_model_dependant();
updateEventHandlers();
```

Esto es: inicializar el RModel, crear los main y secondary renderers que correspondan, crear los objetos encargados de la rotación, traslación y escala del modelo, actualizar la información del modelo en la vista y finalmente dibujar el modelo.

La función “changeViewType” se considera como parte de la inicialización del RModel y corresponde a revisar en la interfaz que tipo de proyección esta seleccionada y guardar el dato en RModel. La función “enable_model_dependant” lo que hace es desbloquear en la interfaz las interacciones que dependían de la existencia de un modelo. Finalmente, “updateEventHandlers” lo que hace es asignar las acciones de las variables “rotator”, “translator” y “scalator” a acciones del mouse.

Las variables “model”, “rModel”, “rotator”, “translator” y “scalator” son globales dentro de la aplicación. Esto implica que, si se carga un modelo distinto, estas serán reemplazadas. Abrir el mismo modelo por segunda vez no tiene efecto, pues importar un modelo con el mismo nombre de archivo no provoca el evento “change”.

5.1.3 Dibujar el Modelo en el canvas

Dibujar el modelo en el canvas es tarea de la función “draw”. Como se vio en la sección anterior, esta se llama al final de la carga de un modelo. En la interfaz no está presente de forma explícita, sino que se llama cada vez que el modelo sufre algún cambio. Este cambio no es necesariamente de posición, si no que puede ser producto de una evaluación o un cambio del tipo de proyección.

La implementación de “draw” es relativamente simple, pues son en realidad los renderers los que se llevan todo el trabajo. La implementación es como sigue:

```
function draw(){
  resizeCanvas(gl.canvas);
  gl.viewport(0, 0, gl.canvas.width, gl.canvas.height);
  gl.clearColor(0, 0, 0, 0);
  gl.clear(gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT);
  gl.enable(gl.DEPTH_TEST);
  gl.enable(gl.CULL_FACE);
  if(mainRenderer != null){
    mainRenderer.draw();
  }

  for(var i = 0; i < secondaryRenderers.length; i++){
    secondaryRenderers[i].draw();
  }
}
```

La función “resizeCanvas” lo que hace es detectar si el canvas cambio de tamaño e indica a WebGL para que el modelo se dibuje bien. El canvas podría cambiar de tamaño por ejemplo si la ventana se achica, o se abre la consola del browser. Si no se indican estos cambios, el modelo podría perder calidad o verse deforme al dibujarse.

Luego, lo que se hace es “limpiar” el canvas, es decir borrar cualquier cosa que esté previamente dibujada. Seguidamente se activan las funcionalidades de “depth test” y “face culling”. Depth test se usa para evitar que se dibujen elementos que están al fondo, sobre elementos que están al frente, y face culling es para que no se dibuje la parte posterior de los polígonos. Finalmente, los renderers llaman a su propio método draw, el cual termina dibujando el modelo (en el caso del main renderer) y sus propiedades (en el caso de los secondary renderers) en el canvas.

Para seleccionar cual es el main renderer a utilizar, el usuario debe seleccionar en la interfaz alguno de los botones mostrados en la figura 24.

Figura 24 – Botones de selección de main renderer.

La diferencia entre main renderers corresponde al modelo de iluminación, los botones de la imagen, de izquierda a derecha representan:

1. La iluminación se calcula según las normales de las caras.
2. La iluminación se calcula según las normales de los vertices.
3. La iluminación no se calcula, se dibujan todas las caras del mismo color.
4. No hay renderer, el modelo no se dibuja.

Los efectos que produce cada main renderer se puede observar en la figura 25, estos se muestran en el mismo orden que en los botones. El resultado del cuarto botón no se muestra, pues correspondería al canvas vacío.

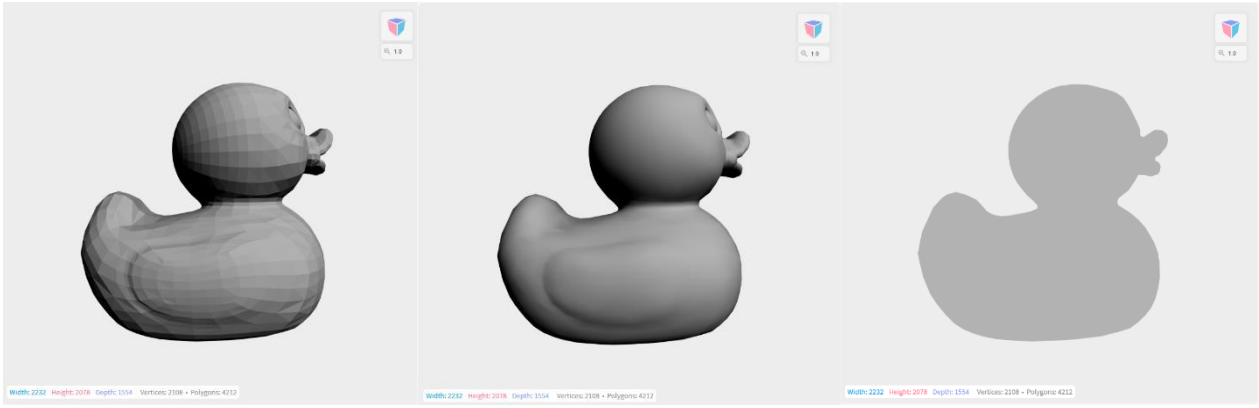

Figura 25 – Modelo dibujado con distintos main renderers.

Otro detalle importante para destacar es que activar face culling significa que el interior del modelo no puede ser observado, porque si se acerca lo suficiente el modelo a la cámara como para ver su interior, uno estaría viendo efectivamente la parte posterior de los polígonos. Sin embargo, la aplicación sí soporta inspeccionar el modelo por dentro.

La razón por la que face culling esta activado es porque al dibujar un polígono sin face culling, la parte posterior de cada polígono también calcula como “le llega la luz”, y como la parte de atrás no está enfrentando a la fuente de luz, el modelo se ve muy oscuro por dentro. Esto se aprecia en la figura 26.

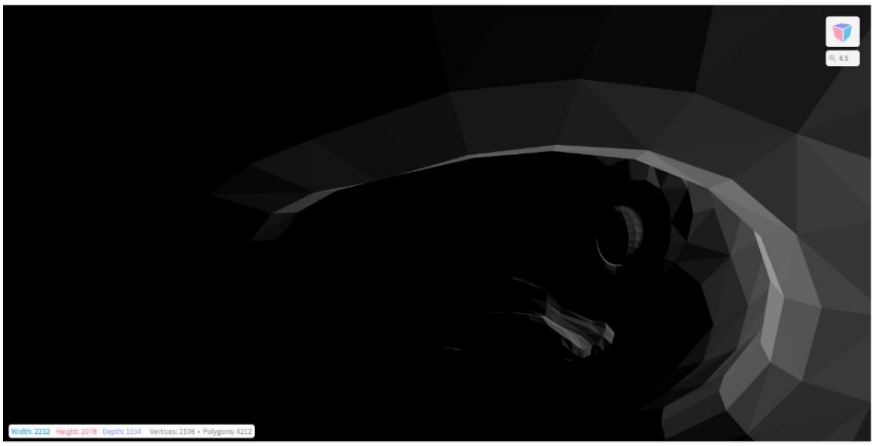

Figura 26 – Interior de un modelo con face culling desactivado.

La solución que se utilizó fue activar face culling, y dibujar el exterior y el interior de manera independiente. Al dibujar el interior, se invierte la dirección de la luz, haciendo que tanto el interior como el exterior queden iluminados. Esto se aprecia en la siguiente porción de código.

```
gl.cullFace(gl.BACK);
gl.uniform3fv(this.reverseLightDirectionLocation, lightDirection);
gl.drawArrays(gl.TRIANGLES, 0, this.rModel.getTrianglesCount()*3);

gl.cullFace(gl.FRONT);
gl.uniform3fv(this.reverseLightDirectionLocation, vec3.negate(lightDirection, lightDirection));
gl.drawArrays(gl.TRIANGLES, 0, this.rModel.getTrianglesCount()*3);
```

El método “cullFace” indica a qué lado del polígono se está aplicando culling. Al entregar el valor “BACK”, se está indicando que la parte de atrás no se dibuja, y al entregar “FRONT” se está indicando que la parte frontal no se dibuja. Luego se fija la variable del renderer que indica la dirección de la luz. Se puede observar que la primera vez se entrega “lightDirection”, pero la segunda vez se aplica el método “negate”. Finalmente se dibuja todo sobre el canvas con el método “drawArrays”. El resultado de como se ve un modelo por el interior usando esta técnica se puede observar en la figura 27.

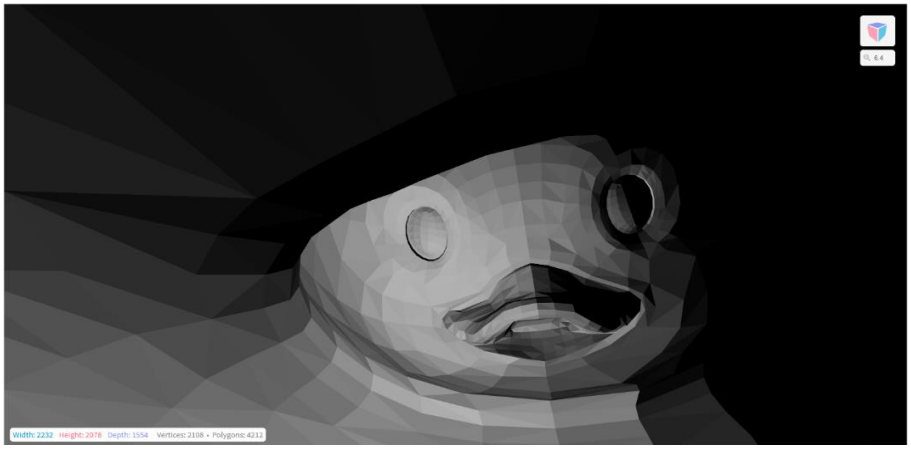

Figura 27 – Interior de un modelo con face culling activado, y dibujando el interior del modelo de manera independiente.

Un detalle interesante es que, en el código mostrado, se aprecia que se está dibujando el interior después de dibujar el exterior. La razón por la que esto funciona es porque depth test esta activado.

Existe un último elemento de la interfaz que afecta el cómo se dibuja el modelo, y eso es la proyección. Como bien se sabe, la proyección puede ser de dos formas: perspectiva y ortogonal. Esto se selecciona en la interfaz con los botones mostrados en la figura 28.

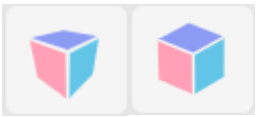

Figura 28 – Botón para cambio de proyección.

En realidad, en la interfaz corresponde a un solo botón cuyo ícono cambia cuando se pasa de una forma de proyección a la otra. A nivel de código, lo único que hace este botón es cambiar la variable “viewType” del RModel, de forma que la siguiente vez que se dibuje el modelo, ocupe una matriz de proyección distinta.

La diferencia visual entre proyección en perspectiva y ortogonal puede observarse en la figura 29. La imagen presenta el mismo modelo 2 veces en la misma posición.

Más detalles sobre cómo funciona RModel, y sobre cómo se relaciona con los Renderers y los Shaders puede ser consultada en la sección 5.2.

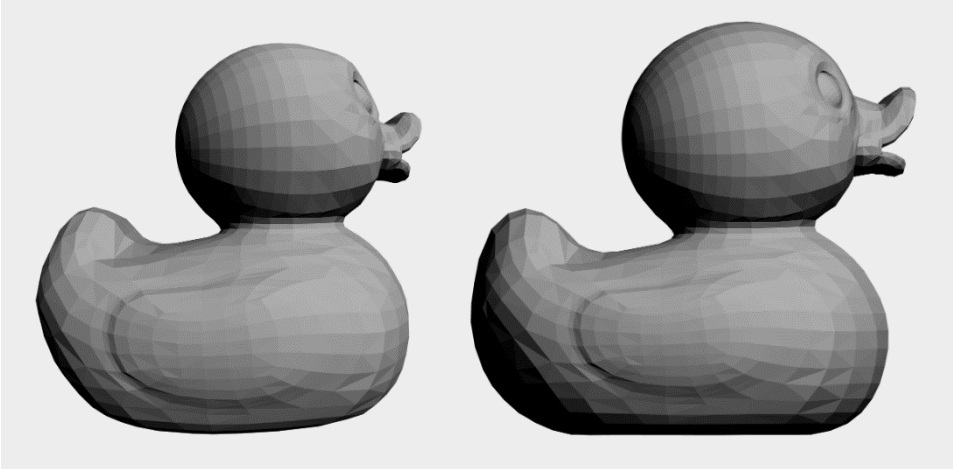

Figura 29 – Modelo con proyección en perspectiva y proyección ortogonal.

5.1.4 Dibujar Propiedades del Modelo

El dibujo de las propiedades del modelo corresponde a los secondary renderers. Estos aparecen en la interfaz como una lista de opciones (figura 30). Su principal caracteristica es que se pueden seleccionar varias propiedades al mismo tiempo.

Esto funciona, porque la aplicación maneja una lista de secondary renderers, y cuando se llama a la función “draw”, se ejecuta también el método “draw” de todos los secondary renderers. La porción de código que maneja esto es la misma mostrada al inicio de la sección 5.1.3.

Aparte de la capacidad de seleccionar varios al mismo tiempo, funcionan esencialmente igual que un main renderer, y también son afectados por cosas como las matrices de transformación.

La figura 30 muestra un total de 6 secondary renderers, pero en la versión actual de la aplicación están implementados los primeros 4. En la figura 31 es posible observar como se ve el wireframe de un modelo sin ningún main renderer aplicado.

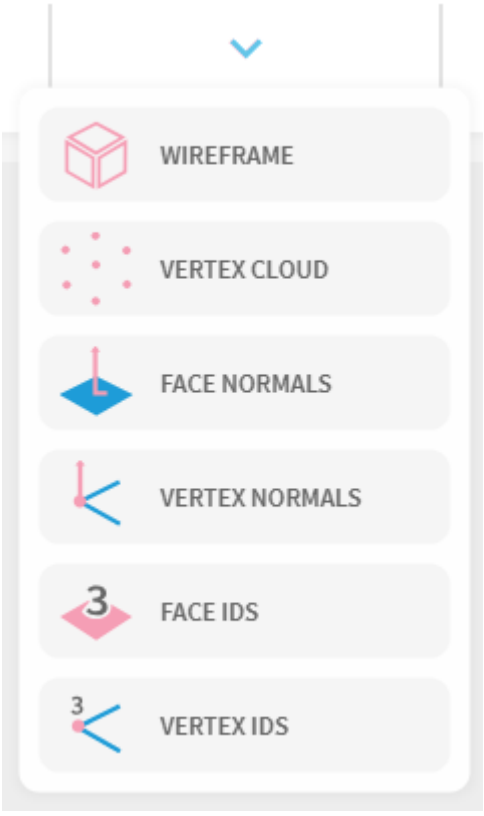

Figura 30 – Selección de secondary renderers en la interfaz.

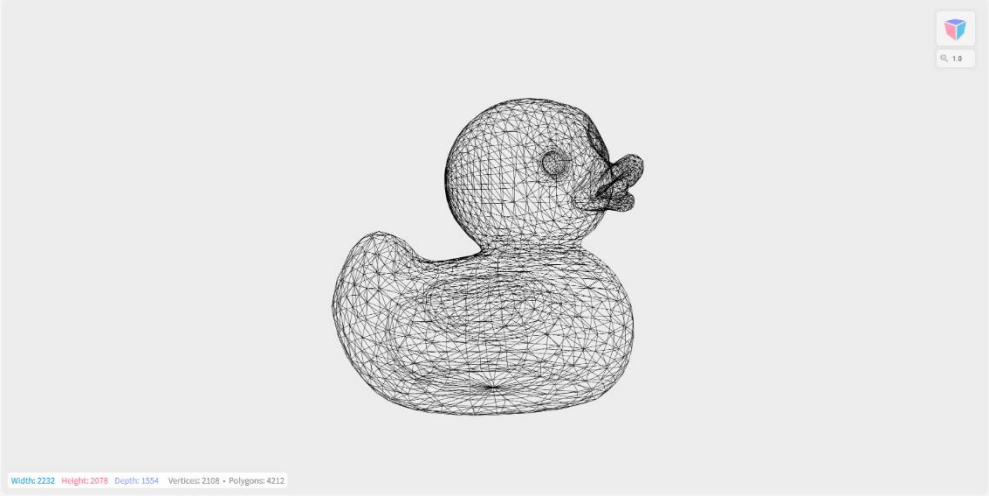

Figura 31 – Wireframe de un modelo sin ningún main renderer activado.

5.1.5 Manipulación del modelo

La manipulación del modelo se refiere a los conceptos de rotación, traslación y escala. Estos en la interfaz no están asociados a ningún botón, y se manejan exclusivamente con el mouse. También existe el botón “reset view”, mostrado en la figura 32, que devuelve el modelo a su posición original. Esto lo hace reiniciando los objetos “rotator”, “translator” y “scalator”.

Figura 32 – Botón para resetear la vista.

Manejar la escala es relativamente sencillo, la escala no está diseñada para ser independiente por eje, así que se puede representar con un único número, el cual inicialmente es 1. Si con el mouse se hace scroll hacia arriba, el número sube, y si se hace scroll hacia abajo, el número baja.

En cuanto a la traslación también es simple. Esta está ligada al click derecho del mouse. Internamente se maneja como un vector. El vector de traslación se inicializa con el valor (0, 0, 0). Esto quiere decir que no hay traslación. Presionar el click derecho del mouse “agarra” el modelo. Si el mouse se mueve hacia la derecha en el eje x, se le suma a la variable x, y si se mueve hacia la izquierda se le resta. Lo mismo sucede para el eje y. La traslación no considera desplazamiento en el eje z (hacia adentro de la pantalla), y la escala es una buena forma de simular que el objeto se está acercando/alejando.

La rotación es la más complicada de comprender. Se podría pensar en hacer algo como la traslación en la que desplazar el mouse en una dirección aumenta/disminuye el ángulo en alguno de los 2 ejes disponibles. Pero eso produce efectos extraños en la interacción. El defecto más común es mover el cursor hacia arriba, provocando que el modelo rote y quede en una posición “acostada”, luego desde esta posición se mueve el mouse hacia la derecha. La interacción esperada es que el modelo rote alrededor del eje x, pero lo que sucede es que rota alrededor del eje z. Esto sucede porque no se está aplicando rotaciones sobre las rotaciones, sino que solo se está actualizando el ángulo, haciendo que actualizar el eje “x” e “y” siempre consideren la rotación a partir de la posición original del modelo.

La solución implementada para la rotación es una técnica conocida como “virtual track ball” [31]. Esta estrategia consiste en “proyectar” una esfera alrededor del modelo. Al mover el mouse sobre esa esfera, se puede calcular en que parte de la esfera se inició el movimiento y en que parte se terminó. Con eso se pueden obtener 2 vectores, uno que va desde el centro de la esfera hasta el punto de inicio, y el otro que va desde el centro de la esfera hasta el punto de termino. Con estos vectores se pueden obtener dos datos, el ángulo entre ellos, y el eje alrededor de cual rotan (equivalente al producto cruz entre los vectores). Con estos dos datos es posible calcular la matriz de rotación que se le debe aplicar al modelo. Luego cuando exista un nuevo movimiento del mouse, se calcula la nueva matriz de rotación usando la misma técnica y se multiplica con la antigua.

Con esta técnica se puede mover el modelo de forma natural y sin preocuparse por las limitaciones de los ejes en la pantalla.

5.1.6 Selección de Polígonos

La selección está representada en la interfaz como un pequeño formulario al costado derecho. Este se puede observar en la figura 33. Este formulario está formado por 4 elementos:

1. Un select para indicar la propiedad por la que se quiere seleccionar.
2. Una forma de ingresar que valores de la propiedad se van a considerar para seleccionar.
3. Una lista para indicar el modo de selección.
4. Un botón de aplicar.

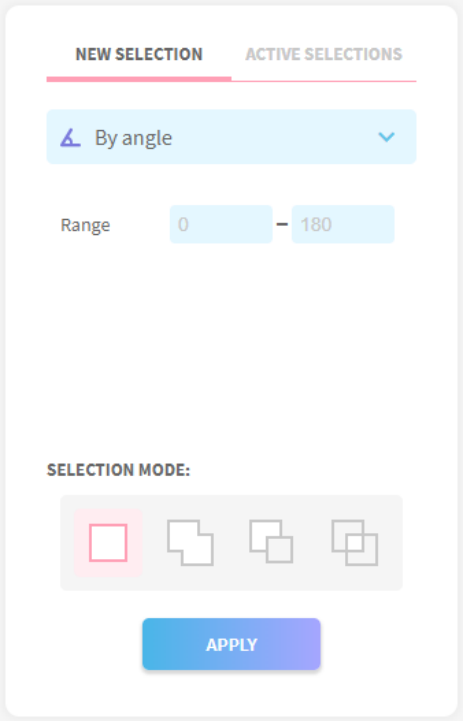

Figura 33 – Selección de polígonos en un modelo. Pestaña de nueva selección.

En el caso de la imagen, la propiedad son los ángulos internos de los polígonos, y se indican los valores con un rango. Al presionar el botón aplicar se llama a una función que toma los datos del formulario, crea un objeto del tipo SelectionStrategy y lo guarda en una lista. Luego se llama a la función “apply_selection”, la cual recorre la lista de selecciones y las aplica sobre el modelo.

La lógica de tener una lista de selecciones tiene varias utilidades. Una es poder mostrarle al usuario cuales son las selecciones que están activas actualmente de la manera más precisa posible. Esto se puede ver en la pestaña de “active selections” del mismo elemento de la interfaz. La figura 34 muestra como se ve la pestaña de “active selections” con 3 selecciones activas.

En esta imagen se logra apreciar que cada selección activa está representada por 4 elementos. Un ícono que indica el tipo de selección, un texto que indica la propiedad seleccionada, un recordatorio de los valores de selección, y un botón de eliminar.

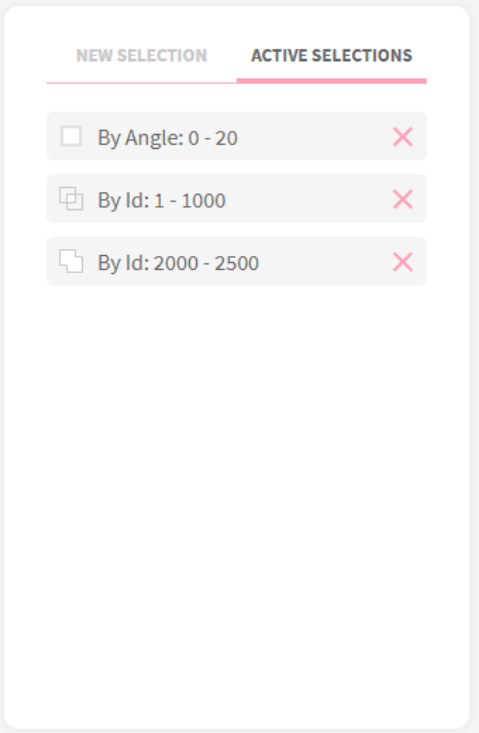

Figura 34 – Selección de polígonos en un modelo. Pestaña de selecciones aplicadas.

Las selecciones siempre se aplican en forma secuencial, por lo que el ejemplo de la imagen se interpreta como: Se seleccionaron los polígonos con ángulos entre 0 y 20, luego eso se interseco con los polígonos con id entre 1 y 1.000, y el resultado de eso se sumó con los polígonos con id entre 2.000 y 2.500.

Este orden es el mismo orden en el que están guardados en la lista, revelando otras de las utilidades de poseer una lista: eliminar una selección es fácil. Al presionar el botón de eliminar, la selección simplemente se elimina de la lista, y las que quedan se vuelven a aplicar de forma secuencial.

Un detalle importante es que eliminar la primera selección genera una reacción en cascada eliminando todas las selecciones. Esto pues no tiene sentido intersecar, sumar y restar si no hay una selección base para realizar estas operaciones.

Aplicar una selección afecta el modelo, y hace que, en la vista, los polígonos seleccionados se marquen en rojo. En la figura 35 se puede observar como se ve el resultado de una selección.

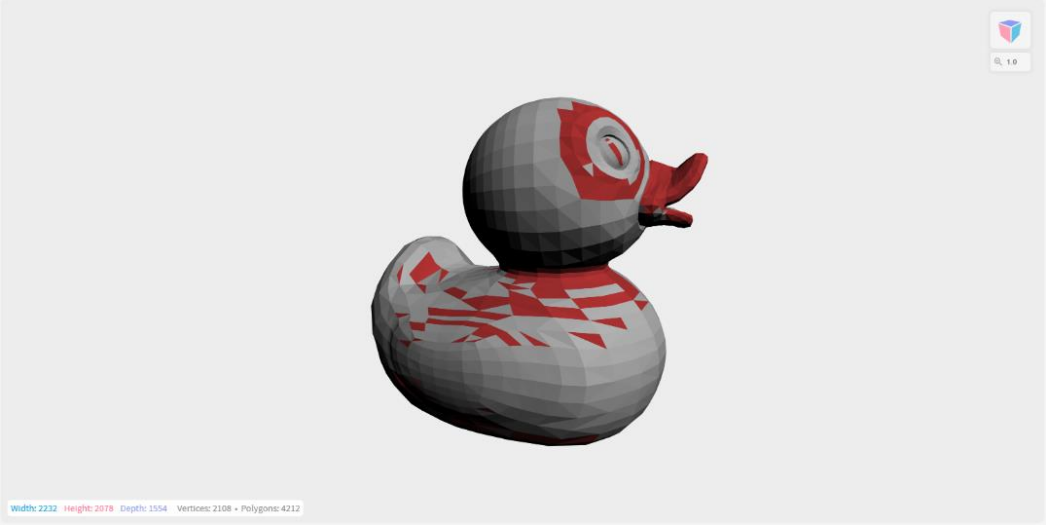

Figura 35 – Resultado de una selección sobre el modelo.

5.1.7 Evaluación del Modelo

En la interfaz la evaluación del modelo está ubicada justo debajo de la selección. Como se muestra en la figura 36, esta contiene:

1. Un select en donde se indica que propiedad se quiere evaluar.
2. Una opción indicando si la evaluación debe ser realizada sobre la selección aplicada actualmente o sobre todo el modelo.
3. Un botón de evaluar.

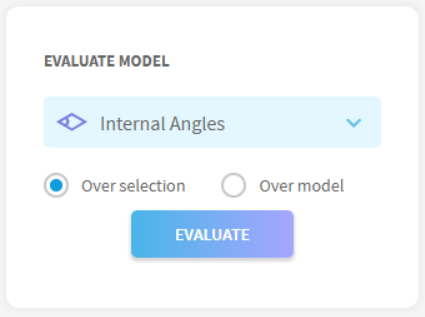

Figura 36 – Evaluación del modelo

El resultado de la evaluación internamente corresponde a un objeto de tipo diccionario que contiene información sobre la evaluación realizada.

Figura 37 – Botón para cambiar el modo de vista.

Esta información es mostrada en la interfaz en un nuevo canvas que aparece en la interfaz después de que se ejecuta una evaluación. Actualmente en este canvas se dibuja un histograma relativo a la propiedad siendo evaluada, permitiendo así al usuario ver si existen valores extremos y en qué proporción. También puede servir para tomar la decisión de realizar una selección, pues todas las propiedades consideradas como estrategias de evaluación también fueron consideradas como estrategias de selección.

Junto con el despliegue de este canvas también se activa el botón mostrado en la figura 37. Este botón se utiliza para cambiar el modo de vista. Este corresponde en realidad a ocultar o mostrar el nuevo canvas, de manera que si el usuario quiere volver a una vista en donde solo se ve el modelo, lo pueda hacer sin problemas (y sin la necesidad de eliminar la evaluación). En la figura 38 se puede apreciar la interfaz con el resultado de una evaluación visible.

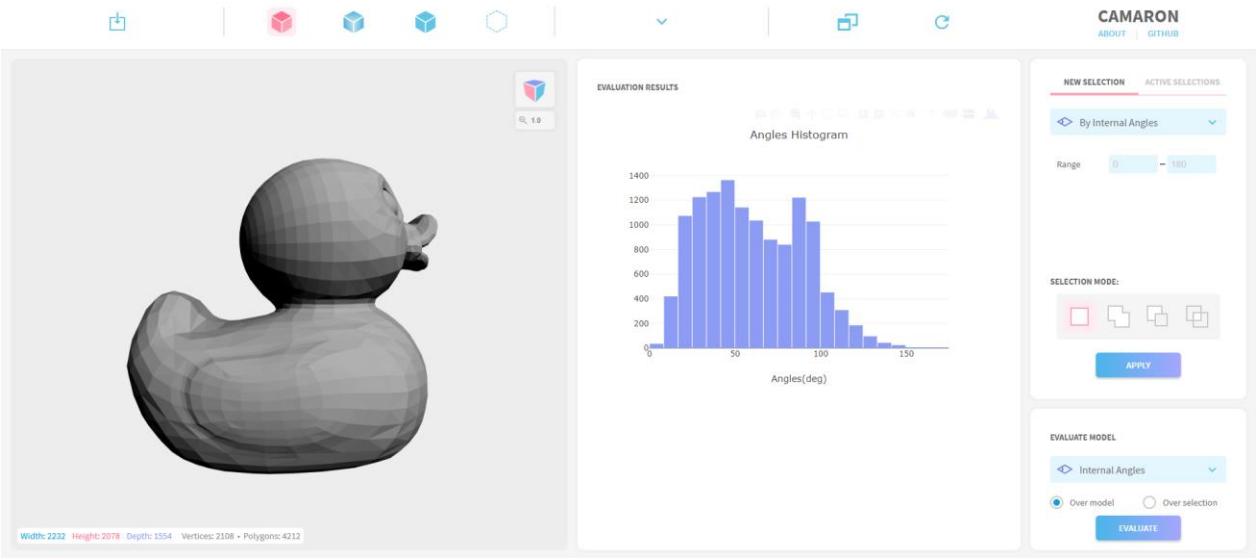

Figura 38 – Vista de la aplicación después de que se realiza una evaluación.

5.2 Código Computación Gráfica

En esta sección se expone y explica las porciones de código que están directamente relacionadas con Computación Gráfica. Estas representan el “corazón” de la aplicación, y son las que hacen posible el visualizar la información expuesta en la sección anterior.

5.2.2 RModel

Como ya se expuso en la sección 4.1.2, al iniciar un RModel, solo es necesario entregar un objeto de tipo model, y llamar al método “loadDataFromPolygonMesh”. Este es el método que inicializa todos los datos que después serán requeridos por los Renderers. La implementación es la siguiente:

```
RModel.prototype.loadDataFromPolygonMesh = function(){

  // Load Model Data
  this.loadTriangles();
  this.loadTrianglesNormals();
  this.loadVertexNormals();
  this.loadEdges();
  this.loadVertices();
  this.loadVertexNormalsLines();
  this.loadFaceNormalsLines();

  // Set Model Matrix
  var translation = vec3.fromValues(-this.center[0], -this.center[1], -this.center[2]);
  this.modelMatrix = mat4.create();
  mat4.translate(this.modelMatrix, this.modelMatrix, translation);

  // Set View Matrix
  var camera = vec3.fromValues(0, 0, this.modelDepth*2)
  var target = vec3.fromValues(0, 0, 0)
  var up = vec3.fromValues(0, 1, 0);
  this.viewMatrix = mat4.create()
  mat4.lookAt(this.viewMatrix, camera, target, up);
}
```

Se logra apreciar que existe una gran cantidad de llamadas a otros métodos internos de RModel. Cada uno de ellos calcula y guarda alguna propiedad que puede ser necesaria en el futuro. Todos estos métodos funcionan de la misma manera: recorren los polígonos del objeto de tipo Model original, y generan un arreglo con los datos que correspondan.

Posteriormente se puede apreciar que se inicializan la matriz del modelo “Model Matrix” y la matriz de la cámara “View Matrix”. “Model Matrix” se inicializa con una traslación inicial equivalente al centro del modelo original, pero con sus coordenadas en negativo. Esto es para que el modelo quede centrado en la posición (0, 0, 0) cuando se pase a coordenadas de mundo. Es esta matriz la cual será sometida a todos los cambios de rotación, traslación y escala que sean recibidas desde la interfaz de usuario.

La cámara se inicializa al centro de la pantalla, pero desplazada en el eje z dos veces la profundidad del modelo, indicando así que está al frente del modelo. “View Matrix” se calcula usando la función “lookAt”, que apunta la cámara hacia la posición “target”, la cual corresponden a la posición (0, 0, 0), que anteriormente se fijó como el centro del modelo. Esta matriz se mantiene fija durante toda la ejecución del programa, esto quiere decir, la cámara nunca cambia de posición.

Volviendo a los métodos internos que cargan información, se expone aquí la implementación de “loadTriangles” para ejemplificar como se calcula la información que será enviada a los Renderers:

```
RModel.prototype.loadTriangles = function(){
  var polygons = model.getPolygons();

  var polygon; var polygonVerticesCount; var polygonVertices;
  var vertex1; var vertex2; var vertex3;

  for(var i = 0; i < this.polygonsCount; i++){
    polygonVerticesCount = polygons[i].getVerticesCount();
    for(var j = 1; j < polygonVerticesCount-1; j++){
      this.trianglesCount++;
    }
  }

  var triangles = new Float32Array(this.trianglesCount*9);
  var tcount = 0;

  for(var i = 0; i < this.polygonsCount; i++){
    polygon = polygons[i];
    polygonVerticesCount = polygon.getVerticesCount();
    polygonVertices = polygon.getVertices();
    for(var j = 1; j < polygonVerticesCount-1; j++){
      vertex1 = polygonVertices[0].getCoords();
      vertex2 = polygonVertices[j].getCoords();
      vertex3 = polygonVertices[j+1].getCoords();

      triangles[tcount] = vertex1[0]; triangles[tcount+1] = vertex1[1]; triangles[tcount+2] = vertex1[2];
      triangles[tcount+3] = vertex2[0]; triangles[tcount+4] = vertex2[1]; triangles[tcount+5] = vertex2[2];
      triangles[tcount+6] = vertex3[0]; triangles[tcount+7] = vertex3[1]; triangles[tcount+8] = vertex3[2];

      tcount += 9;
    }
  }
  gl.bindBuffer(gl.ARRAY_BUFFER, this.trianglesBuffer);
  gl.bufferData(gl.ARRAY_BUFFER, triangles, gl.STATIC_DRAW);
  this.loaded += 1;
}
```

Primero se hace un conteo de los triángulos que contiene el modelo. Este no es equivalente a la cantidad de polígonos del modelo original, pues dichos polígonos no son necesariamente triángulos. Entonces, la forma en la que se está recorriendo es equivalente a hacer una triangulación sobre cada polígono del modelo.

Es necesario saber la cantidad de triángulos de antemano para poder pedir el espacio en memoria para el arreglo a generar. Esto corresponde a la llamada “new Float32Array(this.trianglesCount*9)”. La multiplicación por 9 se realiza porque cada triangulo tiene 3 vertices, y cada vértice tiene 3 coordenadas, y el arreglo de triángulos corresponde a una lista de valores, en donde cada 3 posiciones se consideran un vértice, y cada 3 vertices se considera un triángulo. Esta forma de estructurar no es arbitraria, sino que es la requerida por WebGL (y OpenGL).

Una vez creado el arreglo, se procede a iterar de nuevo, pero esta vez no para contar los triángulos, si no que para llenar el arreglo. Por la forma de recorrer las variables i y j, se hace relativamente complicado saber qué posición del arreglo se debe llenar en qué momento, por lo que estas se relegaron a que solo indiquen el polígono actual, y el vértice con el que se esté trabajando. Con el objetivo de llevar el conteo de la posición del arreglo se crea la variable “tcount”.

Una vez que los triángulos fueron calculados, se guarda el resultado en un buffer, en este caso particular “trianglesBuffer”. Esta parte de código es particular de WebGL: “trianglesBuffer” es en realidad un puntero a un buffer creado en la GPU, “gl.bindBuffer” asigna “trianglesBuffer” como el buffer en el que se está trabajando, y finalmente “gl.bufferData” solicita el espacio y llena el buffer con la información que se le entregue. Entonces, al guardar el resultado en un buffer, queda efectivamente almacenado en la GPU, y RModel queda con un puntero a dicho buffer.

La última línea, la cual incremente una variable interna, es simplemente una variable para chequear la consistencia interna de RModel una vez que termine de cargar todo.

5.2.3 Renderers

A continuación, se muestra la implementación del “DirectVertexRenderer”, que es el que dibuja el modelo calculando la iluminación considerando las normales de los vertices. En la sección 4.1.4 se indicó que un renderer debía implementar 3 métodos: “init”, “draw” y “updateColor”. El código de “init” para “DirectVertexRenderer” corresponde a:

```
DirectVertexRenderer.prototype.init = function(){
    gl.useProgram(this.program);
    this.positionAttributeLocation = gl.getAttribLocation(this.program, "a_position");
    this.normalAttributeLocation = gl.getAttribLocation(this.program, "a_normal");
    this.colorAttributeLocation = gl.getAttribLocation(this.program, "a_color");
    this.MVPLocation = gl.getUniformLocation(this.program, "u_worldViewProjection");
    this.modelLocation = gl.getUniformLocation(this.program, "u_world");
    this.reverseLightDirectionLocation = gl.getUniformLocation(this.program, "u_reverseLightDirection");

    this.positionBuffer = this.rModel.getTrianglesBuffer();
    this.normalBuffer = this.rModel.getVerticesNormalsBuffer();
    this.colorBuffer = gl.createBuffer();
    this.vao = gl.createVertexArray();

    gl.bindVertexArray(this.vao);

    gl.enableVertexAttribArray(this.positionAttributeLocation);
    gl.bindBuffer(gl.ARRAY_BUFFER, this.positionBuffer);
    gl.vertexAttribPointer(this.positionAttributeLocation, 3, gl.FLOAT, false, 0, 0);

    gl.enableVertexAttribArray(this.normalAttributeLocation);
    gl.bindBuffer(gl.ARRAY_BUFFER, this.normalBuffer);
    gl.vertexAttribPointer(this.normalAttributeLocation, 3, gl.FLOAT, false, 0, 0);

    gl.enableVertexAttribArray(this.colorAttributeLocation);
    gl.bindBuffer(gl.ARRAY_BUFFER, this.colorBuffer);
    gl.bufferData(gl.ARRAY_BUFFER, this.rModel.getColorMatrix(), gl.STATIC_DRAW);
    gl.vertexAttribPointer(this.colorAttributeLocation, 3, gl.FLOAT, false, 0, 0);
}
```

Es necesario notar que todos los métodos que empiezan con “gl”, son definidos por WebGL. La línea que contiene “gl.useProgram” es usada para especificar el shader que se va a utilizar, el cual fue anteriormente definido en el constructor del Renderer.

Luego viene una serie de llamadas a “gl.getAttributeLocation” y “gl.getUniformLocation”. Estas se usan para obtener punteros a las variables definidas en el shader. El primer argumento es entonces el shader, y el segundo un string con el nombre de la variable.

Luego viene la inicialización de los buffers. En este ejemplo se logra observar que los primeros 2 buffers se piden directamente al RModel, pues como se indicó en la sección anterior, este crea y guarda los punteros necesarios a los buffers que se necesitan. En particular “colorBuffer” no está contenido en RModel, pues los colores pueden cambiar producto de una selección. RModel guarda los colores como un arreglo directamente.

También se aprecia la creación de una variable llamada “vao”. Esta corresponde a un “Vertex Array Object”, y es la encargada de guardar punteros a los buffers, y como se saca la información de estos. Al llamar a “gl.bindVertexArray” se marca “vao” como activo.

Luego, por cada buffer se llama al método “gl.enableVertexAttribArray”, que recibe el puntero a un buffer y lo guarda en “vao”. Luego se llama al método “gl.bindBuffer”, que como se indicó en la sección anterior se usa para indicar cual es el buffer sobre el que se está trabajando y, en este caso particularmente, asocia el buffer a la posición recibida por “gl.enableVertexAttribArray”. Finalmente “gl.vertexAttribPointer” indica cómo se debe sacar la información del buffer que acaba de ser asociado, información que posteriormente será recordada por “vao”.

Se puede observar en el código, que la tercera vez que se hace esto, se llama a “gl.bufferData”. Esto es lo mismo que se explicó en la sección anterior, y corresponde a llenar el buffer de colores, con los resultados de la llamada “rModel.getColorMatrix()”.

Todo este proceso, deja inicializado el renderer, pero todavía no se dibuja nada. Para esto es que existe el método draw:

```
DirectVertexRenderer.prototype.draw = function(){
    gl.useProgram(this.program);
    gl.bindVertexArray(this.vao);
    var lightDirection = vec3.fromValues(0.5, 0.7, 1);
    var lightDirection = vec3.normalize(lightDirection, lightDirection);

    gl.uniformMatrix4fv(this.MVPLocation, false, this.rModel.getMVP());
    gl.uniformMatrix4fv(this.modelLocation, false, this.rModel.getModelMatrix());

    gl.cullFace(gl.BACK);
    gl.uniform3fv(this.reverseLightDirectionLocation, lightDirection);
    gl.drawArrays(gl.TRIANGLES, 0, this.rModel.getTrianglesCount()*3);

    gl.cullFace(gl.FRONT);
    gl.uniform3fv(this.reverseLightDirectionLocation, vec3.negate(lightDirection, lightDirection));
    gl.drawArrays(gl.TRIANGLES, 0, this.rModel.getTrianglesCount()*3);
}
```

Aquí nuevamente se llama a “gl.useProgram”, para indicar que shader se utilizara; y a “gl.bindVertexArray”, para asignar el “vao” (quien contiene toda la información inicializada por el método anterior) como activo. Esto último es necesario, pues todos los renderers definen su propio “vao”, y solo puede haber uno activo. No volver a activarlo podría implicar utilizar un “vao” equivocado, generando resultados inesperados.

Luego se crea un vector indicando la dirección de la luz, y posteriormente se asignan valores a las variables de tipo “uniform” que fueron inicializadas en “init”. Esta asignación de valores se hace en “draw” y no en “init”, pues corresponden a las matrices que controlan la posición del modelo, y estas cambian cada vez que el modelo se desplaza, rota, o se escala.

El resto del código es el mismo expuesto en la sección 5.1.3. Un detalle no explicado allí, es que “gl.drawArrays” recibe el tipo de primitiva que se está dibujando. En particular acá es “gl.TRIANGLES”. Esto le indica al shader que se va a dibujar un triángulo cada 3 vertices procesados. Esta información también es útil para cuando se debe interpolar los colores, o las normales durante el proceso de “Rasterization”.

WebGL y OpenGL manejan 3 tipos de primitivas: “gl.TRIANGLES”, “gl.LINES” y “gl.POINTS”. “gl.TRIANGLES” se utiliza para todos los main renderers. “gl.LINES” es utilizado en para los secondary renderers que dibujan el wireframe, las normales de las caras y las normales de los vertices. “gl.POINTS” es utilizado para el secondary renderer que dibuja la nube de vertices.

Finalmente se tiene el método “updateColor”, el cual es bastante simple y es expuesta acá solo por completitud:

```
DirectVertexRenderer.prototype.updateColor = function(){
    gl.enableVertexAttribArray(this.colorAttributeLocation);
    gl.bindBuffer(gl.ARRAY_BUFFER, this.colorBuffer);
    gl.bufferData(gl.ARRAY_BUFFER, this.rModel.getColorMatrix(), gl.STATIC_DRAW);
    gl.vertexAttribPointer(this.colorAttributeLocation, 3, gl.FLOAT, false, 0, 0);
}
```

Esto es equivalente a cargar el buffer de colores de nuevo, y las 4 líneas de código, son las misma últimas 4 líneas de código del método “init”. La diferencia en realidad está en que “rModel.getColorMatrix” entrega un resultado distinto en caso de que en el model original exista algún polígono seleccionado.

5.2.4 Shaders

Por simplicidad, hasta ahora se ha referido al Shader de manera singular. Sin embargo, el referirse a “Shader”, en realidad se quiere decir “programa de Shaders” (por esto mismo es que la variable se llama “program”). Y un programa de Shaders está compuesto por un “Vertex Shader” y un “Fragment Shader” (ver sección 2.1.4).

Para explicar cómo funcionan los programas de shaders en la aplicación, mostraremos como es el constructor de un Renderer:

```
var DirectVertexRenderer = function(rModel){
    Renderer.call(this, rModel);
    this.program = createProgramFromSources(gl, [normalVertexShader, normalFragmentShader]);
}
```

Aquí se logra apreciar que se llama a una función llamada “createProgramFromSources”, que recibe una lista con las variables “normalVertexShader” y “normalFragmentShader”. Estas son variables globales que están definidas en el archivo “shaders.js”, y que corresponden a Strings con el código de un programa escrito en “GLSL”.

Lo que hace entonces “createProgramFromSources” es compilar un programa de shaders, entregándole el código de un Vertex Shader y el código de un Fragment Shader. Es este el programa que posteriormente se entrega como parámetro a “gl.useProgram” mostrado en la sección anterior.

La aplicación cuenta con cuatro Vertex Shaders y cuatro Fragment Shaders. Para motivos de la explicación, se expondrá uno de cada uno. “normalVertexShader” se define como:

```
var normalVertexShader = `#version 300 es

in vec4 a_position;
in vec3 a_normal;
in vec4 a_color;

uniform mat4 u_worldViewProjection;
uniform mat4 u_world;

out vec3 v_normal;
out vec4 v_color;

void main() {
  gl_Position = u_worldViewProjection * a_position;
  v_normal = mat3(u_world) * a_normal;
  v_color = a_color;
}
`;
```

La primera línea “#versión 300” indica la versión de GLSL que se está utilizando. Si no se indica, WebGL ocupa por default la versión 100. Por otro lado, la versión 300 es la máxima soportada, y la necesaria al utilizar la última versión de WebGL.

Se definen 3 tipos de variables: “in”, “uniform” y “out”. Las variables “in” corresponden a información entregada desde los buffers. No se reciben los buffers enteros, si no las porciones correspondientes a un único vértice. Entonces en este caso “a_position” recibe la posición del vértice, “a_normal” recibe la normal del vértice y “a_color” recibe el color del vértice.

Las variables “uniform” corresponden a variables globales, compartidas entre todas las instancias del Vertex Shader. En particular aquí se están entregando la matriz “MVP” en la variable “u_worldViewProjection”, y la matriz del modelo en la variable “u_world”.

Las variables “out” corresponden a variables que serán procesadas durante la etapa de “Rasterization” y posteriormente recibidas por el Fragment Shader. En este caso se define una normal, y un color.

Luego se tiene una función “main”, esta es la que se ejecuta por cada vértice. Cada vértice se procesa al mismo tiempo, por lo que existen varias instancias del Vertex Shader (y por eso se indicó que “uniform” son variables compartidas entre todas las instancias). Aquí se asignan valores a las variables de tipo out, y en particular es necesario especificar “gl_Position”, la cual es una variable definida por defecto en WebGL.

“gl_Position” es la posición que posteriormente será transformada en coordenadas de ventana en el proceso de “Rasterization”, y corresponde a la posición del vértice, multiplicada por la matriz “MVP”, que es quien deja el vértice en coordenadas homogéneas.

“v_normal” corresponde a la normal del vértice, pero multiplicada por la matriz del modelo. Esto se hace para aplicar a la normal el mismo movimiento que se le aplique al modelo. Esto es importante, pues en caso de no mover las normales junto con el modelo, da la impresión de que la fuente de luz se mueve junto con el modelo, dando resultados de iluminación no deseados.

“v_color” simplemente pasa el mismo color que fue recibido directamente. En la etapa de “Rasterization”, “v_normal” y “v_color” son interpoladas y pasadas al Fragment Shader. Se muestra a continuación, el código del Fragment Shader:

```
var normalFragmentShader = `#version 300 es

precision mediump float;

in vec3 v_normal;
in vec4 v_color;

uniform vec3 u_reverseLightDirection;

out vec4 outColor;

void main() {
  vec3 normal = normalize(v_normal);
  float light = dot(normal, u_reverseLightDirection);

  outColor = v_color;
  outColor.rgb *= light;
}
`;
```

La línea “precisión mediump float” indica el nivel de precisión con el que la GPU debe manejar las variables. Este es necesario especificarlo, o de lo contrario el shader no va a compilar. Esto es de acuerdo con las especificaciones de WebGL, pues para los Fragment Shaders no tiene un valor por defecto.

Los Fragment Shaders trabajan con fragmentos. Cada fragmento, en términos simples, corresponde a uno de los pixeles que están dentro de la primitiva con la que se esté trabajando en él momento. En la teoría no se pude considerar que un fragmento es equivalente a un pixel, pues múltiples fragmentos pueden representar a un único pixel. En la práctica, en la aplicación no se trató con ese tipo de casos.

Las variables de tipo “in” en este caso corresponden a los valores de las normales y los colores interpolados, recibidos de la etapa de “Rasterization”. Se recibe en una variable de tipo “uniform” la dirección de la luz, y se define una variable de tipo “out”, que dará como resultado el color a pintar en el fragmento recibido.

La función “main” se ejecuta una vez por cada fragmento (o pixel) dentro de una primitiva. Como se puede observar, lo único que se hace es normalizar la normal (esto es, hacer que su largo sea 1), calcular el producto punto entre la dirección de la luz y la normal, y multiplicar este valor por el color recibido. Esto corresponde al proceso descrito en la sección 2.1.5.

Al terminar el Fragment Shader, los pixeles correspondientes son pintados de los colores determinados, haciendo así que el modelo sea visible en el canvas de la aplicación. Cada vez que el modelo sufre algún cambio, se actualiza las variables de tipo “uniform” y todo el proceso es ejecutado de nuevo.

Los Fragment Shader no siempre entregan solo un color para pintar, es acá donde se pueden aplicar texturas, distintos algoritmos de iluminación o efectos como transparencias o sombras. En el caso de la aplicación los shaders son relativamente sencillos, pues iluminación directa y colores planos, son suficientes para visualizar de manera satisfactoria las propiedades de un modelo.

La aplicación también define shaders específicos para dibujar líneas, y para dibujar puntos. A pesar de trabajar con primitivas distintas, le implementación y uso de estos siguen la misma lógica que los shaders expuestos en esta sección.

6 Evaluación

La evaluación de la aplicación se dividió en 2 partes. Primero, la evaluación de usabilidad, que corresponde a la aplicación de los conceptos de interacción humano-computador. Segundo, la evaluación de rendimiento, donde se mide cuáles son los tiempos de ejecución y como se compara la aplicación contra MeshLabJS.

6.1 Usabilidad

Para medir la usabilidad de la aplicación, se hizo utilización de los métodos rápidos conocidos como “recorrido de diseño” y “evaluación basada en escenarios”.

El método “recorrido de diseño” se realiza cuando se están desarrollando los wireframes y corresponde a realizar reuniones con potenciales usuarios, mostrándoles las acciones y funcionalidades que pueden realizar en la interfaz, y recibiendo sus comentarios. Cada versión del wireframe fue revisada por lo menos por un potencial usuario y en la práctica sirvió para corregir problemas de vocabulario (expresiones que no se entendían), y de posicionamiento de los elementos en la interfaz.

El método de “evaluación basada en escenarios” se realiza sobre la aplicación terminada. Aquí se presentó la aplicación a 3 potenciales usuarios, y se les indico ciertos escenarios (cosas como “identificar ángulos menores a 10 grados”, o “observar el interior de un modelo”), y por cada uno de estos se les iba mostrando como hacer dicha acción. Indicar un escenario antes de realizar la acción ayuda a que los usuarios se enfoquen en lo que esperan, y lo comparen respecto a lo que realmente ven que se está haciendo. Es aquí donde se hicieron notar los problemas con los botones de movimiento (indicados en la sección 4.2.3).

Al ser métodos rápidos, los resultados de las evaluaciones son totalmente cualitativos. En particular los usuarios señalaron, durante el proceso de diseño, cosas como que la aplicación se ve bien estéticamente, y que la distribución se siente natural. Esto, junto con las técnicas de HCI aplicadas; los lineamientos de diseño que se determinaron y siguieron; y las evaluaciones rápidas realizadas; permite afirmar que la aplicación desarrollada tiene una buena usabilidad.

Es posible aplicar métodos rigurosos para terminar de concluir que la usabilidad obtenida es la deseada. Los métodos rigurosos son más costosos de realizar en cuanto a tiempo y recursos; y se suelen aplicar al final de los proyectos. Estos son normalmente para confirmar la usabilidad conseguida al seguir los lineamientos de diseño. Por tiempo, aplicar este tipo de métodos no se consideró como parte de los objetivos del trabajo, aunque sería bueno aplicarlos en el futuro.

6.2 Rendimiento

A pesar de que el rendimiento no fue uno de los enfoques del trabajo, igual es importante de considerar, aunque sea para tener una referencia para comparar como se puede mejorar en el futuro, y como se compara contra el visualizador MeshLabJS.

Las medidas de tiempo fueron tomadas con un cronometro externo. Esto pues MeshLabJS no indica cuanto se demora, y aunque en la aplicación desarrollada se puede obtener el tiempo preciso, es mejor usar la misma técnica de medición para que los resultados sean comparables.

6.2.1 Ambiente de pruebas

Las pruebas fueron ejecutadas en 2 computadores. El computador 1 cuenta con las siguientes características:

- Sistema Operativo Windows 10
- Procesador Intel Core i5-6500, 3.2 GHz.
- Tarjeta de Videos NVidia GeForce GTX 1060 de 6GB
- RAM: 8GB, DDR4, 2133MHz
- Browser Google Chrome

Mientras que el computador 2 cuenta con las siguientes características:

- Sistema Operativo Windows 10
- Procesador Intel Core i7-7700HQ, 2.8 GHz.
- Tarjeta de Video NVidia GeForce GTX 1050 de 4GB
- RAM: 16GB, DDR4, 2400MHz
- Browser Google Chrome

Se escogieron 4 modelos para hacer la comparación: uno de 4.000 caras, otro de 50.000 caras, otro de 800.000 caras y finalmente uno de 2.000.000 de caras. Estos corresponden a modelos de un pato, un torso, un dragón, y un modelo 3D de la geometría del centro de Chile respectivamente y se pueden observar en la figura 39.

La mayoría de estos modelos son libres de descargar en internet. El modelo del pato en particular requiere que se indique su licencia, la cual se puede encontrar en las referencias [32]. El torso es un archivo de ejemplo incluido con el programa MeshLab. El dragón fue obtenido en una página de modelos llamada SketchFab [33]. El modelo de Chile fue recibido de parte de uno de los potenciales usuarios para ver cómo se veía en la aplicación.

Figura 39 – Modelos utilizados para la evaluación de rendimiento.

6.2.2 Resultados

Uso de Memoria RAM en MB: Se determinó usando el administrador de tareas integrado de Chrome, que entrega información de uso de RAM por pestaña.

Computador 1	Caras	Camarón Web	MeshLabJS
Pato	4.000	55	1150
Torso	50.000	110	1160
Dragon	800.000	900	1190
Chile	2.000.000	2110	1209

Computador 2	Caras	Camarón Web	MeshLabJS
Pato	4.000	55	1125
Torso	50.000	110	1140
Dragon	800.000	900	1160
Chile	2.000.000	2110	1185

Uso de Memoria GPU en MB: También se determinó utilizando el administrador de tareas integrado de Chrome.

Computador 1	Caras	Camarón Web	MeshLabJS
Pato	4.000	96	50
Torso	50.000	108	51
Dragon	800.000	300	73
Chile	2.000.000	850	183

Computador 2	Caras	Camarón Web	MeshLabJS
Pato	4.000	101	50
Torso	50.000	114	51
Dragon	800.000	318	73
Chile	2.000.000	597	102

Tiempo de carga en segundos: Se determino usando un cronometro externo. El tiempo se tomó desde que se apretó el botón “abrir” hasta que el modelo era visible en la pantalla.

Computador 1	Caras	Camarón Web	MeshLabJS
Pato	4.000	<1	< 1
Torso	50.000	1.7	1
Dragon	800.000	6	6
Chile	2.000.000	15	14

Computador 2	Caras	Camarón Web	MeshLabJS
Pato	4.000	<1	< 1
Torso	50.000	1.5	1
Dragon	800.000	5.4	5.5
Chile	2.000.000	11	12

6.2.3 Discusión

Primero hay que notar que los computadores que se utilizaron para medir el rendimiento, aunque distintos, no es fácil afirmar cual es mejor que el otro. Se puede observar que los resultados obtenidos son bastante similares en ambos. Por lo mismo, se prioriza comparar el rendimiento entre aplicaciones más que entre computadores.

En cuanto a uso de memoria RAM, el comportamiento es consistente en ambos computadores. Se observa que la cantidad que utiliza Camarón Web crece a medida que aumenta el tamaño del modelo. Esto seguramente se debe a la cantidad de objetos que se crean: en un modelo de 4.000 caras se crean 4.000 objetos de tipo “Polygon”, mientras que en uno de 50.000 se crean 50.000 objetos de tipo “Polygon”. Por su lado, MeshLabJS también crece a medida que aumenta el tamaño del modelo, pero este crecimiento no es lineal, e incluso el modelo más pequeño ocupa una gran cantidad de memoria.

Esto se puede deber a varios factores. Lo más probable es que tenga que ver con la manera de almacenar modelos de three.js, que es la librería gráfica que utiliza MeshLabJS. Seguramente para el modelo se crea una estructura pesada, en la que después se guarda el modelo de manera eficiente, haciendo que el uso de memoria crezca lentamente.

Hasta el modelo de 800.000 caras, Camarón Web utiliza menos memoria que MeshLabJS, pero para él modelo de 2.000.000 de caras Camarón Web utiliza casi el doble de lo que requiere MeshLabJS. En este sentido MeshLabJS es más escalable que Camarón Web, pero Camarón Web es más eficiente al trabajar con modelos pequeños/medianos. Para los usos particulares de camarón, este es el mejor escenario, pues visualizar propiedades se vuelve más difícil a medida que las mallas van creciendo.

En cuanto a uso de memoria en GPU, el comportamiento en ambos computadores también es relativamente consistente, pero hay una anomalía específica: el computador 2, tanto en Camarón Web como en MeshLabJS, ocupo mucha menos memoria para el modelo de 2.000.000 de caras. Al momento de realizar este análisis, no se ha encontrado ninguna explicación de porque sucede esto.

Independiente de esta anomalía, lo que sí es consistente es que MeshLabJS utiliza efectivamente, menos memoria en GPU que Camarón Web. La explicación de esto es que, en Camarón Web, se está explícitamente guardando toda la información posible en la GPU, incluso antes de utilizar los renderers y dibujar el modelo. Esta situación fue expuesta en la sección 5.2.2.

En cuanto a los tiempos de carga de los modelos, ambas aplicaciones fueron más rápidas en el computador 2. Dicho esto, MeshLabJS se comportó mejor que Camarón Web en el computador 1, mientras que Camarón Web fue más rápido que MeshLabJS en el computador 2 (excepto para el modelo de 50.000 caras).

En esta línea vale la pena destacar que Camarón Web tiene un delay intencional de medio segundo, utilizado para abrir el modal de carga. Sin ese delay, el modal simplemente no se abre debido a la naturaleza asíncrona de JavaScript. Este modal permite darle al usuario un feedback de que la aplicación está trabajando en la carga del modelo. Considerando que la carga de los modelos más grandes puede demorarse más de 5 segundos, este feedback es necesario desde un punto de vista de usabilidad. MeshLabJS por su parte no provee ningún tipo de feedback durante la carga de un modelo, a pesar de los posibles tiempos de espera.

La ganancia de tiempo en el computador 2, en ambas aplicaciones, dan a entender que uno de los factores más importantes es la RAM, pues es la única diferencia considerable. El computador 1 tiene un procesador más rápido (aunque de una generación anterior), y una mejor tarjeta gráfica, y aun así es más lento en cargar modelos.

Todos estos resultados y comparaciones dan bastante información en cómo mejorar la aplicación en un futuro para mejorar su rendimiento. En particular el uso excesivo de objetos hace que Camarón Web no escale muy bien con el tamaño del modelo, y se podría tratar de reducir de varias formas. Por ejemplo, hay varias propiedades o elementos que podrían representarse como listas en vez de objetos.

Otro detalle interesante, es que no hay paralelismo involucrado en la aplicación (aparte del que da por defecto hacer uso de la tarjeta gráfica), y dado los tiempos de carga similares, da la impresión de que MeshLabJS tampoco aplica paralelismo. Aplicar paralelismo en CPU en JavaScript es complicado, pues la única forma de hacerlo es con los llamados “web workers” que son muy simples y no proveen de herramientas básicas de sincronización. Este es un problema interesante para abordar, y que podría aumentar el rendimiento de la aplicación.

7 Conclusiones

7.1 Sobre la aplicación desarrollada

El objetivo general del trabajo corresponde a el desarrollo de una aplicación de visualización y evaluación de mallas geométricas, el cual se logró satisfactoriamente.

Para comprobar los requerimientos, basta con recorrer la lista y verificar que todos están presentes. En cuanto a los objetivos específicos, casi todos van en línea con los mismos requerimientos. También hay un objetivo que indica que la aplicación debe ser ejecutada totalmente en el lado del cliente, y este también se cumplió. Por otro lado, es necesario referirse explícitamente al objetivo de lograr mantenibilidad.

El código procuró escribirse con mucho cuidado, poniendo nombres compresibles a las variables, funciones y métodos; ordenando los archivos de una manera comprensible; y documentando el código apropiadamente. Todo esto con el fin de aumentar la legibilidad, y que este sea fácil de seguir incluso para los programadores menos experimentados, en particular teniendo a los estudiantes en mente.

A pesar del cuidado que se tuvo en legibilidad, hay un detalle que impide afirmar una mantenibilidad total: la falta de Testing. No es que falte hacer Testing a la aplicación, si no que este no está automatizado. Por cada funcionalidad, se definían ciertas acciones para comprobar que la funcionalidad era correcta: esto corresponde a los test. Además, cada vez que se añadía una nueva funcionalidad, se repetían las acciones de las funcionalidades anteriores para asegurarse que todo siguiese funcionando correctamente. A un nuevo programador se le podría entregar esta lista de acciones, pero sin estar familiarizado con la aplicación, no podrá determinar si el resultado que ve es el esperado o no, además que mientras la aplicación crece, testear manualmente se hace extremadamente lento. Por lo mismo, es necesario agregar Testing automatizado a la aplicación, y al no estar presente en la versión actual se dirá que el objetivo de mantenibilidad se cumple parcialmente.

7.1 Sobre el uso de HCI

Otro de los objetivos del proyecto correspondía a aplicar conceptos de interacción humano-computador. Este objetivo realmente puede considerarse como exitoso pues todo el desarrollo se llevó a cabo usando técnicas y conceptos de dicha área.

El proyecto se desarrolló con el usuario y el contexto en mente. Y aunque en ingeniería de software tradicional eso también se considera una buena práctica, suele ser considerado lo que requiere el usuario, pero no su percepción. La interacción humano-computador le da un nivel mucho más profundo al concepto de usuario. Poder decir que la aplicación no solo cumple la funcionalidad requerida, sino que también es intuitiva y atractiva visualmente, aumenta mucho el valor de un desarrollo.

7.3 Comentarios

Con respecto a la motivación personal “introducirse al mundo de la Computación Gráfica”, el proyecto se considera muy satisfactorio. Se alcanzó una gran comprensión sobre conceptos claves de Computación Gráfica, que van a permitir adentrarse a cosas más complejas en un futuro cercano. Incluso, tomando todo el aprendizaje obtenido durante el proyecto, y mirando los módulos más tempranos de la aplicación, que fueron desarrollados cuando mucho conocimiento no existía, nacen muchas ideas sobre cómo mejorar la aplicación, en particular respecto a rendimiento.

La motivación principal del proyecto, “crear un visualizador y evaluador de mallas geométricas, gratuito y de código abierto, que aproveche la potencia del hardware actual”, se basa principalmente en la escasez de herramientas para evaluar mallas. Este es un proceso que suele quitar tiempo a los investigadores.

Esta motivación va más allá de si se logró específicamente lo enunciado, y aquí es donde hay que contrastarlo con el alcance del proyecto. Como se mencionó en la misma sección 1.5, el enfoque de evaluación que tiene esta aplicación es único, y solo comparable con el software Camarón original. La aplicación cumple con ser un evaluador, pero para lograr realmente que ayude a los investigadores, es necesario que la aplicación empiece a ser utilizada.

Existen muchas medidas para lograr esto, en particular presentar el proyecto directamente en alguna conferencia (lo cual está en proceso de concretarse). Una vez presentado, se puede destacar que el enfoque en técnicas de HCI y la facilidad de instalación serán factores decisivos en lograr que la aplicación sea utilizada. Esto pues no tener que instalar invita a la gente a probar la aplicación sin mayores problemas, y una buena usabilidad ayuda a los usuarios a que le encuentren utilidad a la aplicación.

Todo lo expuesto le da un gran alcance potencial a este trabajo, y a partir de ahora se trabajará en que esto se logre. La motivación principal, la creación de la aplicación, se cumplió; y con esto empieza el desafío de captar usuarios, y seguir mejorando la aplicación.

Por otro lado, la aplicación también se enfocó en la legibilidad del código no solo para mantenibilidad, sino que también para ser de ayuda a los estudiantes interesados en él tema. Se espera que todo el conocimiento obtenido durante el desarrollo sea fácil de percibir en el código y la documentación, y sirvan de base para la gente que está interesada en el mundo de la Computación Gráfica.

A modo general se puede decir que la terminación del proyecto fue exitosa. Todos los objetivos fueron cumplidos (y el objetivo que se cumplió parcialmente será abordado como corresponde), y que el potencial del proyecto para seguir creciendo es muy grande.

7.4 Trabajo Futuro

No esta demás repetir que dentro de las posibilidades futuras para la aplicación esta agregar más de las características que ya tiene: nuevos formatos de archivos soportados, nuevos diseños de iluminación, más secondary renderers, más estrategias de evaluación y más estrategias de selección.

Sin embargo, el reino de las posibilidades es mucho más amplio que las funcionalidades que ya existen. Dentro de las funcionalidades que podrían agregar valor a la aplicación están:

- Que se soporte mallas de poliedros (por ejemplo, mallas formadas por conjuntos de tetraedros).
- Que se pueda dibujar más de un modelo a la vez en la misma escena.
- Permitir que se suban modelos más complejos (por ejemplo, con texturas).
- Que soporte ediciones (por ejemplo, eliminar/mover vértices).
- Agregar algoritmos de triangulación para mallas que no sean de triángulos.
- Poder exportar las mallas (solo tiene sentido si se pueden editar los modelos).

Es importante indicar que si se decide implementar alguna de estas funcionalidades (u otras), se debería seguir la misma estrategia de diseño incremental. Es necesario planear como estas nuevas funcionalidades se deben ver en la interfaz, y como afecta esto a las antiguas funcionalidades.

Otras cosas que se pueden mejorar es lo relativo a rendimiento. La sección 6.2, por ejemplo, muestra que el uso excesivo de objetos hace que la aplicación use mucho espacio. También se puede tratar de aplicar paralelismo, y buscar técnicas de Computación Gráfica más rápidas para las acciones que ya se realizan.

8 Bibliografía

1. Sharcnet. Mesh Quality. [en línea] <https://www.sharcnet.ca/Software/Ansys/17.0/en-us/help/flu_ug/flu_ug_mesh_quality.html> [consulta: 7 septiembre 2018]

2. Cánepa, Aldo. 2013. Camarón: visualizador y evaluador de mallas geométricas mixtas grandes en 3D, acelerado con Shaders en OpenGL. Memoria de Ingeniero Civil en Computación. Santiago, Universidad de Chile, Facultad de Ciencias Físicas y Matemáticas.

3. TetView. Sitio web oficial del visualizador TetView. [en línea] <http://wias-berlin.de/software/tetgen/tetview.html> [consulta: 7 septiembre 2018]

4. GeomView. Sitio web oficial del visualizador GeomView. [en línea] <http://www.geomview.org/> [consulta: 7 septiembre 2018]

5. MeshLab. Sitio web oficial del visualizador MeshLab. [en línea] <http://www.meshlab.net/> [consulta: 7 septiembre 2018]

6. Hughes “et al”. 2013. Computer Graphics: Principles and Practice. 3ra edición. Addison-Wesley.

7. opengl-tutorial. Tutorial 3: Matrices. [en línea] <http://www.opengl-tutorial.org/beginners-tutorials/tutorial-3-matrices/> [consulta: 7 septiembre 2018]

8. OpenGL. OpenGL Overview. [en línea] <https://www.opengl.org/about/> [consulta: 7 septiembre 2018]

9. Khronos. OpenGL ES for the Web. [en línea] <https://www.khronos.org/webgl/> [consulta: 7 septiembre 2018]

10. Akenine-Möller “et al”. 2008. Real-Time Rendering. 3ra edición. A K Peters

11. Khronos. Rendering Pipeline Overview. [en línea] <https://www.khronos.org/opengl/wiki/Rendering_Pipeline_Overview> [consulta: 7 septiembre 2018]

12. Khronos. Shader. [en línea] <https://www.khronos.org/opengl/wiki/Shader> [consulta: 7 septiembre 2018]

13. WebGL2Fundamentals. WebGL 3D – Directional Lighting. [en línea] <https://webgl2fundamentals.org/webgl/lessons/webgl-3d-lighting-directional.html> [consulta: 7 septiembre 2018]

14. LearningWebGL. WebGL Lesson 7 – basic directional and ambient lighting. [en línea] <http://learningwebgl.com/blog/?p=684> [consulta: 7 septiembre 2018]

15. AutoDesk. Polygon normals. [en línea] <http://help.autodesk.com/view/MAYAUL/2017/ENU/?guid=GUID-9C257D44-924D-4B3F-ADEF-C71FAA98EAB1> [consulta: 7 septiembre 2018]

16. SketchFab. Sitio web oficial de la plataforma SketchFab. [en línea] <https://sketchfab.com> [consulta: 7 septiembre 2018]

17. MeshLabJS. Sitio web de la aplicación MeshLabJS. [en línea] <http://www.meshlabjs.net/> [consulta: 7 septiembre 2018]

18. MeshLabJS. MeshLabJS Code Documentation. [en línea] <http://www.meshlabjs.net/doc/html/> [consulta: 7 septiembre 2018]

19. Interaction Design Foundation. Human-Computer Interaction (HCI) [en línea] <https://www.interaction-design.org/literature/topics/human-computer-interaction> [consulta: 7 septiembre 2018]

20. McMaster University. Contexts for HCI. [en línea] <http://wiki.cas.mcmaster.ca/index.php/Contexts_for_HCI> [consulta: 7 septiembre 2018]

21. Karr, Ashley. 2014. ACM Interactions. Wireframes Defined. [en línea] <http://interactions.acm.org/blog/view/wireframes-defined> [consulta: 7 septiembre 2018]

22. ExperienceUX. What is wireframing? [en línea] <https://www.experienceux.co.uk/faqs/what-is-wireframing/> [consulta: 7 septiembre 2018]

23. Ruby Zheng. Learn to Create Accessible Website with the Principles of Universal Design [en línea] <https://www.interaction-design.org/literature/article/learn-to-create-accessible-websites-with-the-principles-of-universal-design> [consulta: 15 octubre 2018]

24. University of Cambridge. 2013. Human-Computer Interaction Lecture 8: Usability evaluation methods. [diapositivas] [en línea] <https://www.cl.cam.ac.uk/teaching/1314/HCI/HCI2013-lecture8.pdf> [consulta: 7 septiembre 2018]

25. Practical UX Methods. Design Walkthrough. [en línea] <http://practicaluxmethods.com/product/design-walkthrough/> [consulta: 7 septiembre 2018]

26. Singh, Jaspreet y Kaur, Pinkiparampreet. 2008. Object Oriented Programming Using C++. Technical Publications Pune.

27. MDN web docs. Object-oriented JavaScript for beginners. [en línea] <https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/Object-oriented_JS> [consulta: 7 septiembre 2018]

28. Freeman, Eric y Freeman, Elisabeth.2004. Head First Design Patterns. O'Reilly.

29. Dofactory. Strategy. [en línea] <https://www.dofactory.com/net/strategy-design-pattern> [consulta: 7 septiembre 2018]

30. glMatrix. Sitio web oficial librería glMatrix. [en línea] <http://glmatrix.net/> [consulta: 7 septiembre 2018]

31. Marcus Lab. Sin título. Ejemplo de implementación de virtual track ball. [en línea] <http://ensharable.appspot.com/lab/virtual_track_ball.html > [consulta: 7 septiembre 2018]

32. SPDX. SCEA Shared Source License. [en línea] <https://spdx.org/licenses/SCEA.html> [consulta: 7 septiembre 2018]

33. SketchFab. Drogon Game of Thrones (free download). [en línea] <https://sketchfab.com/models/97eb2b4fc00c47aab30566adc777faa1> [consulta: 7 septiembre 2018]

34. Gruber, Diana. Fastgraph. The Mathematics of the 3D Rotation Matrix. [en línea] <http://www.fastgraph.com/makegames/3drotation/> [consulta: 7 septiembre 2018]

35. Shewchuk, Jonathan. 200). What Is a Good Linear Finite Element? -Interpolation, Conditioning, Anisotropy, and Quality Measures. Proceedings of the 11th International Meshing Roundtable. 73.

36. Philippe P. Pébay, & Baker, T. 2003. Analysis of Triangle Quality Measures. Mathematics of Computation, 72(244), 1817-1839.

37. Materialize. Página principal del framework materialize. [en línea] <https://materializecss.com/> [consulta: 7 septiembre 2018]

38. Florida State University. OFF Files, Geomview Object File Format. [en línea] <https://people.sc.fsu.edu/~jburkardt/data/off/off.html> [consulta: 7 septiembre 2018]

Anexo A: Matriz de Rotación

A diferencia de las matrices de escala y traslación, las matrices de rotación son complejas y existen muchas formas de crearlas. En el presente anexo se expondrán las dos formas más comunes. Notar que no se incluye la información matemática para explicar porque funciona, sino que solo se muestran las técnicas.

La primera forma, que es la que se suele mostrar cuando se enseña sobre las matrices de transformación, es ocupar las matrices específicas para los ejes X, Y y Z. Si se quiere aplicar una rotación θ alrededor de cualquiera de los tres ejes, entonces tenemos lo siguiente [34]:

Matriz de rotación respecto al eje X:

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

Matriz de rotación respecto al eje Y:

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

Matriz de rotación respecto al eje Z:

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

La segunda forma consiste en aplicar una rotación sobre cualquier eje arbitrario. Este matriz se define de la siguiente forma:

$$\begin{matrix} tx^2 + c & txy - sz & txz + sy & 0 \\ txy + sz & ty^2 + c & tyz - sx & 0 \\ txz - sy & tyz + sx & tz + c & 0 \\ 0 & 0 & 0 & 1 \end{matrix}$$

Donde c es $\cos\theta$, s es $\sin\theta$, t es $1 - \cos\theta$ y (x, y, z) corresponde a un vector representando el eje alrededor del cual ocurre la rotación. Es interesante destacar que, si el eje corresponde a cualquiera de los ejes X, Y o Z tradicionales, esta matriz da como resultado alguna de las tres matrices anteriores.

Anexo B: Criterios de Calidad de un Modelo

Aunque se mencionó que los criterios de calidad suelen ser subjetivos dependiendo del uso que se le quiera dar a un modelo, los detalles en los que hay que fijarse casi siempre son propiedades geométricas.

También es cierto que existen casos en que la calidad de un modelo es objetivamente mala, por ejemplo, cuando uno de sus polígonos tiene área 0.

Acá presentamos medidas de calidad comunes que suelen ser aplicados a un modelo [35] [36]. En particular estos son aplicables a triángulos, que son los polígonos más comunes al momento de crear una malla:

1. **Ángulos Extremos**: Es una de las medidas más importantes e intuitivas. Un triángulo se considera de mala calidad cuando alguno de sus ángulos se acerca mucho a 0 o a 180. Curiosamente un ángulo cercano a 180 causa más error que ángulos cercanos a 0.

2. **Razón de Radios (ρ):** Esta medida se define como la razón entre el radio de la circunferencia circunscrita y el radio de la circunferencia inscrita en el triángulo. Un triángulo se considera de buena calidad si ρ/2 es cercano a 1.

3. **Razón de Aristas (τ)**: Esta medida se define como la razón entre la arista más larga y la arista más corta. Puede ser también definida en función de los ángulos extremos. Se dice que un triángulo es de buena calidad si τ es cercano a 1.

4. **Arista a Circunradio (ω)**: Se define como la razón entre el radio de la circunferencia circunscrita, y la arista más larga. Un triángulo es de buena calidad si 2ω es cercano a 1. Hay que tener cuidado con esta métrica, pues en un triángulo con un ángulo muy cercano a 0 esta medida también tiende a 1.

5. **Triángulos degenerados**: Más que una medida de calidad, un triángulo degenerado indica directamente que el modelo tiene una falla. Un triángulo se llama degenerado cuando su área es 0.

A pesar de que estos criterios fueron definidos para triángulos, los referidos a ángulos se pueden aplicar a cualquier polígono. Es por esta razón que en la aplicación se utilizó ángulos para la primera implementación de los criterios de evaluación y selección.

Anexo C: Incrementos de la Aplicación

Para el proyecto desarrollado se aplicaron tres incrementos durante la etapa de implementación. Estos son:

Incremento 1 (inicio del trabajo): El primer incremento se basó principalmente en poder abrir y visualizar el modelo junto con sus estadísticas más comunes. No se consideró ningún tipo de interacción. Esto incluye los siguientes requerimientos:

- Visualizar el modelo.
- Seleccionar método de iluminación del modelo.
- Permitir lectura de archivos en formato off.
- Chequear consistencia del modelo al cargarlo.
- Mostrar número de caras y vertices.
- Mostrar medidas del modelo.

Incremento 2: El segundo incremento consideró la capacidad de manipular la posición del modelo. Esto incluye toda la implementación de interacciones con el mouse. En la practica la rotación se perfeccionó en el incremento 3, pero su primera implementación fue acá. También es acá donde se implementaron los secondary renderers. Los requerimientos correspondientes fueron:

- Rotación libre del modelo.
- Traslación libre del modelo.
- Zoom in, Zoom out.
- Volver modelo a la posición original.
- Visualizar modelo como malla de alambres.

Incremento 3: En el último incremento se implementó todo lo que tiene ver con selección de polígonos y evaluación del modelo. Además, en este incremento es cuando se recibió la versión final del diseño, así que también se consideró la unión de todas las funcionalidades anteriores con el diseño. Los requerimientos correspondientes son:

- Seleccionar polígonos del modelo según criterio definido por el usuario, basado en las propiedades de estos.
- Seleccionar polígonos según su identificador.
- Evaluar modelos según criterio definido por el usuario, basado en las propiedades de los polígonos que lo componen.
- Como resultado, mostrar rangos de valores encontrados para la propiedad seleccionada en la evaluación.

Antes de contar con el diseño, se usó un HTML simple creado con la ayuda del framework css “materialize” [37]. En la figura 40 se puede observar cómo se veía la interfaz de la aplicación al final del incremento 2.

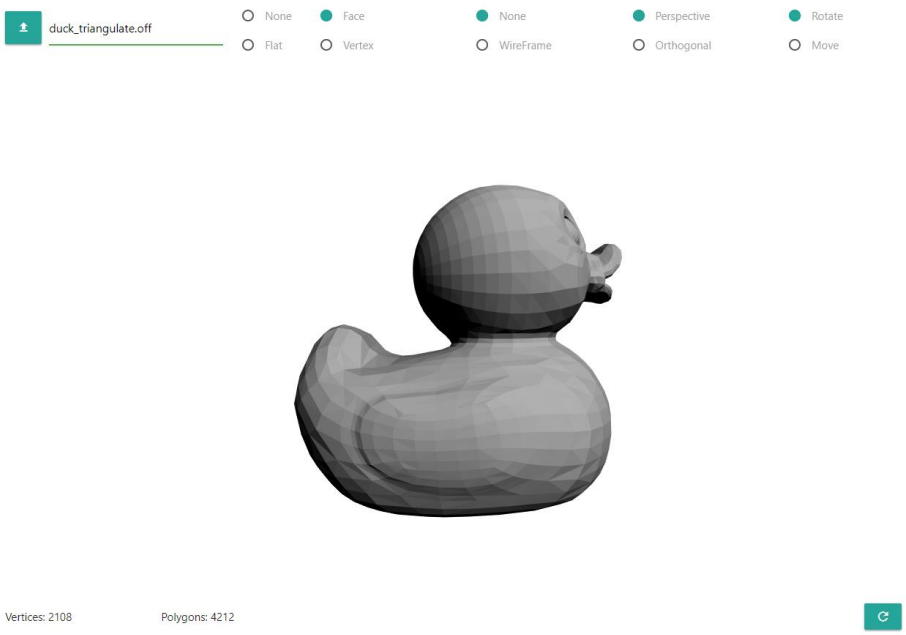

Figura 40 – Interfaz de la aplicación antes de contar con diseño.

Esta interfaz baso su distribución espacial en los primeros wireframes desarrollados por el diseñador. Se logra apreciar que todos los elementos son en efecto del tipo input o select, y no íconos. Además, todavía se cuenta con las opciones de seleccionar el modo de movimiento (rotar y mover).

Anexo D: El formato OFF

El formato de archivo OFF fue creado para el visualizador GeomView y es un tipo de archivo útil para guardar la descripción de objetos 2D y 3D que están formados por polígonos [38]. Dentro de las características de estos archivos se tiene que:

- Está escrito en ASCII
- Puede incluir información de colores
- No está comprimido.

Estructura

La primera línea siempre debe ser la palabra “OFF”.

La segunda línea contiene la información sobre cantidad de vértices, cantidad de caras y cantidad de aristas, en ese orden. Ejemplo:

8 6 12

Esta línea indica que hay 8 vertices, 6 caras y 12 aristas.

Cada línea siguiente corresponde a un vértice. Por cada vértice se debe dar información de las coordenadas X, Y y Z en ese orden. Ejemplo:

1.632993 0.000000 1.154701

Una vez definidos todos los vertices, cada línea siguiente corresponde a un polígono. En estas líneas se indican el número de vertices del polígono y los vertices que lo componen. Los vertices corresponden a índices, donde el índice es el orden de aparición en la sección de vértices del archivo. Ejemplo:

4 7 4 0 3

Esta línea indica que el polígono tiene 4 vértices formados por los vértices número 7, 4, 0 y 3. El orden de los vértices es relevante, pues definen la dirección de las aristas.

Si se quiere dar información de color, esta se puede agregar a la derecha de los vértices, o de los polígonos, indicando los valores r, g, b y a.