Universidad Central de Venezuela
Facultad de Ciencias
Escuela de Computación
Centro de Computación Gráfica

TÉCNICAS DE REDUCCIÓN DE MALLAS EN TIEMPO REAL EMPLEANDO WEBGL

Trabajo Especial de Grado en la Lic. De Computación

Autor: Andrés Agustín Gomes Restrepo
Tutor: Esmitt Ramírez

Caracas, Febrero de 2018

Universidad Central de Venezuela
Facultad de Ciencias
Escuela de Computación
Centro de Computación Gráfica

ACTA DEL VEREDICTO

Quienes suscriben, Miembros del Jurado designado por el Consejo de la Escuela de Computación para examinar el Trabajo Especial de Grado, presentado por el Bachiller Andres Gomes C.I.:18.188.298, con el título *Técnicas de Reducción de Mallas en Tiempo Real Empleando WebGL*, a los fines de cumplir con el requisito legal para optar al título de Licenciado en Computación, dejan constancia de lo siguiente:

Leído el trabajo por cada uno de los Miembros del Jurado, se fijó el día 20 de Febrero de 2018, a las 2:30 pm, para que su autor lo defendiera en forma pública, en el Centro de Computación Gráfica, lo cual se realizó mediante una exposición oral de su contenido, y luego respondió satisfactoriamente a las preguntas que les fueron formuladas por el Jurado, todo ello conforme a lo dispuesto en la Ley de Universidades y demás normativas vigentes de la Universidad Central de Venezuela. Finalizada la defensa pública del Trabajo Especial de Grado, el jurado decidió Aprobarla.

En fe de lo cual se levanta la presente acta, en Caracas a los 20 días del mes de Febrero del año dos mil dieciocho, dejándose también constancia de que actuó como Coordinador del Jurado el Profesor Esmitt Ramírez. El Prof. Ramírez actuó como Jurado por Videoconferencia. Por ello, el Prof. Robinson Rivas como Director de la Escuela de Computación, firma la presente acta en original avalando al Prof. Ramírez.

Prof. Esmitt Ramírez
Tutor

Prof. Francisco Sans
Jurado

Prof. Héctor Navarro
Jurado

Agradecimientos y dedicatorias

Dedico este trabajo principalmente a mi familia y amigos por haberme dado su apoyo incondicional. En especial a mis padres Maritza Restrepo y Agustín Gomes y mi tío José Alex Restrepo que gracias a ellos tuve la oportunidad de cursar mis estudios en esta institución. Agradecimientos a mi tutor Esmitt Ramírez por ser mi guía durante todo este proceso.

Resumen

En Computación Gráfica, lograr un balance entre el nivel de detalle y la portabilidad de un algoritmo es esencial. Al aumentar la complejidad del modelo geométrico se suavizan los bordes y se mejora la percepción de sus elementos, pero se reduce su portabilidad. Con el objetivo de mantener el balance es necesario reducir dicha complejidad en dispositivos con menor capacidad. Para lograr esto es necesario un algoritmo que simplifique el modelo, sea en tiempo real, adecuado para cualquier dispositivo y conserve la topología del modelo, manteniendo la apariencia estética. En este Trabajo Especial de Grado se presenta un algoritmo capaz de realizar simplificaciones en tiempo real utilizando las capacidades de procesamiento paralelo de la *GPU* (*Graphics Processing Unit*) sobre *WebGL*. Esto permite que empleando un navegador con capacidades de acceso al *API* (*Application Programming Interface*) de *WebGL* pueda realizarse el proceso de simplificación y exportarse de forma eficiente para su uso en otras aplicaciones. El algoritmo propuesto reduce la cantidad de polígonos de una malla geométrica sin afectar su topología. Las pruebas realizadas demuestran la efectividad de esta propuesta.

Keywords: Reducción de mallas, nivel de detalle, simplificación, WebGL, API, Threejs

Índice

Introducción

En Computación Gráfica, uno de los principales objetivos es lograr un balance entre el nivel de detalle y la portabilidad de un algoritmo. Nivel de detalle se entiende como la complejidad que posee un modelo geométrico en computadora. Al aumentar la complejidad se suavizan los bordes y se mejora la percepción de los diferentes elementos que componen el modelo. Por otro lado, la portabilidad se entiende como la capacidad que tiene un algoritmo de ejecutarse en diferentes dispositivos. Al aumentar la portabilidad, una mayor cantidad de usuarios son capaces de hacer uso de la aplicación en sus dispositivos.

La evolución en la complejidad de los modelos superó la capacidad de procesamiento de los dispositivos, debido a la creación de herramientas digitales capaces de hacer una correspondencia entre objetos del mundo real y modelos de computadora (proceso conocido como *mapping*), como los escáneres CT (*Computed Tomography*), MRI (*Magnetic Resonance Imaging*) o los láseres para escanear grandes superficies que generan modelos en computadoras con niveles de detalle inmanejables por los dispositivos de uso doméstico (e.g. generalmente, se requieren equipos con altas capacidades gráficas).

Para un dispositivo de distribución comercial, no es posible desplegar en una aplicación gráfica múltiples de estos modelos. Una de las soluciones más implementadas para solventar inconveniente es el uso de la técnica LOD (*Level Of Detail*), que reduce la complejidad de los modelos según su distancia con respecto a la cámara. Una forma de reducir dicha complejidad para un modelo es utilizando algoritmos de simplificación.

Un algoritmo de simplificación busca reducir los elementos que componen un modelo sin afectar su topología, manteniendo la apariencia estética. Estos algoritmos suelen ser lentos debido a que se deben hacer cálculos complejos por cada elemento del modelo, con el fin de determinar la mejor manera de eliminar un elemento del objeto 3D.

En este Trabajo Especial de Grado se describe un algoritmo que es capaz de realizar simplificaciones en tiempo real utilizando las capacidades de procesamiento paralelo de la GPU (*Graphics Processing Unit*). Esto implementado en la *API* (*Application Programming Interface*) gráfico WebGL [1], sobre el lenguaje Javascript. En el capítulo 1 se plantea el inconveniente que se desea resolver con el algoritmo de reducción de mallas geométricas. En el capítulo 2 se explica en detalle los algoritmos de simplificación y el área de la Computación enfocada en la simplificación de mallas. En el capítulo 3 se describen los detalles referentes a la solución desarrollada en este trabajo. En el capítulo 4 se describe la implementación del algoritmo de simplificación de mallas en tiempo real. En el capítulo 5 se exponen los resultados obtenidos al realizar pruebas de rendimiento sobre el algoritmo de simplificación en tiempo real. En el capítulo 6 se presentan las conclusiones y trabajos futuros.

Capítulo 1 Planteamiento del problema

Antes de poder describir el inconveniente existente al trabajar con modelos con alto nivel de detalle se deben definir algunos conceptos básicos.

1.1 Conceptos Básicos

La Computación Gráfica es el campo de las ciencias de la computación encargada del estudio, diseño y despliegue de imágenes por medio del computador. Existe la representación digital de imágenes en dos dimensiones, como objetos geométricos 2D, texto e imágenes digitales en general. Esta representación se hace modificando imágenes con diferentes transformaciones geométricas. A continuación estudiaremos las transformaciones geométricas de traslación, escalamiento y rotación.

Traslación, en la translación se desplaza cada punto de la imagen, una distancia constante en una dirección específica, con lo que permanecerá inalterada la imagen pero trasladada del punto original, a esta transformación geométrica se la denomina rígida ya que la imagen permanece inalterada (rígida) durante la transformación.

$$T_v\boldsymbol{P} = \begin{pmatrix} 1 & 0 & 0 & v_x \\ 0 & 1 & 0 & v_y \\ 0 & 0 & 1 & v_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} p_x \\ p_y \\ p_z \\ 1 \end{pmatrix} = \begin{pmatrix} p_x + v_x \\ p_y + v_y \\ p_z + v_z \\ 1 \end{pmatrix} = \boldsymbol{p} + v$$

Escalamiento, en el escalamiento se multiplican los valores del modelo geométrico por un vector $v = (\, v_x, v_y, v_z,)$, en el que cada elemento de ese vector alterará el grosor del modelo original proporcionalmente a su valor en su respectivo eje (puede incluso ser negativo y hacer un efecto espejo).

$$S_v = \begin{pmatrix} v_x & 0 & 0 \\ 0 & v_y & 0 \\ 0 & 0 & v_z \end{pmatrix} \begin{pmatrix} p_x \\ p_y \\ p_z \end{pmatrix} = \begin{pmatrix} v_x p_x \\ v_y p_y \\ v_z p_z \end{pmatrix}$$

Rotación, en la rotación se hace rotar al modelo geométrico según un vector de rotación junto a un ángulo de rotación, esta transformación también es rígida, ya que no altera la forma del modelo, solo su dirección.

$$R(\theta) = \begin{pmatrix} cos\theta & -sin\theta \\ sin\theta & cos\theta \end{pmatrix}$$

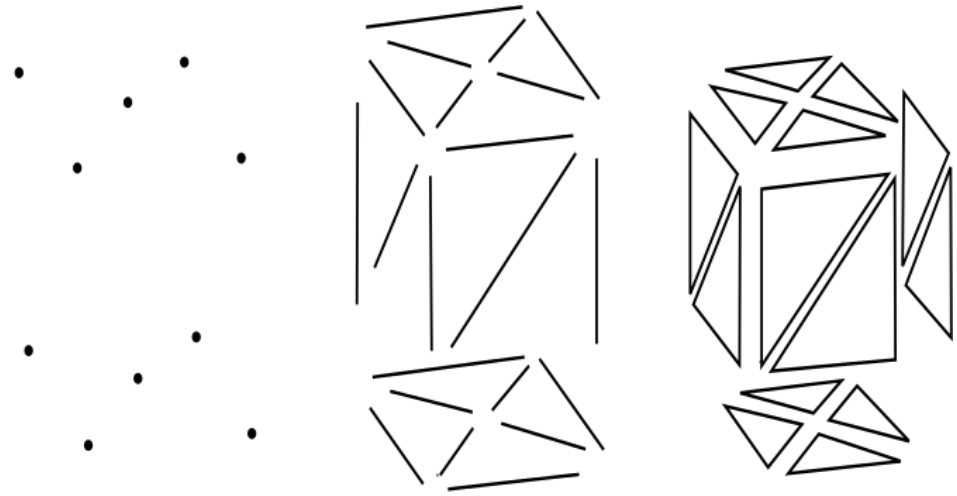
$$R(\theta)v = \begin{pmatrix} cos\theta & -sin\theta \\ sin\theta & cos\theta \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{matrix} x' = xcos\theta - ysin\theta \\ y' = xsin\theta + ycos\theta \end{matrix}$$

La rotación será en dirección de las agujas del reloj si el ángulo es negativo, y contra las agujas del reloj si es positivo. La principal limitante de la representación en dos dimensiones es la dificultad para hacer una representación fiel a la realidad, para poder lograr esto se necesitará usar estructuras geométricas más complejas.

En Computación Gráfica se hace uso de algoritmos particulares para entornos 2D, 3D, y en muchos trabajos se mezclan libremente por lo que a veces es difícil determinar con certeza las diferencias entre estos campos. En la computación en 3D el elemento principal para desplegar objetos y superficies son las mallas, a continuación se explicará en qué consisten las mallas.

1.2 Computación en tres dimensiones y mallas

Una malla es una superficie compuesta por figuras geométricas generadas por intercepciones entre puntos en el espacio, formando figuras geométricas como triángulos, cuadriláteros, tetraedros, polígonos convexos o polígonos con agujeros. Una malla posee, vértices, aristas, caras, polígonos y superficies. Estos elementos son los que forman la base de la estructura de una malla. Se puede ver un ejemplo de estos elementos en la **Figura 1**.



(a) Vértices (b) Aristas (c) Caras

Figura 1: *Elementos básicos de una malla. (a) Vértices, o puntos en el espacio. (b) Aristas, conexiones entre los vértices. (c) Caras, figura geométrica formada a partir de la conexión de 3 o más vértices.*

Los vértices son la base atómica de la malla que describen una posición en el espacio y otros atributos como color, vector normal y coordenadas de textura, los vértices por si mismos no describen una superficie por ser elementos inconexos. Las aristas son las conexiones entre dos vértices relacionados, estas no poseen atributos ni valores por sí mismas, ya que son estructuras derivadas de las caras. Las caras son superficies planas formadas por la conexión entre tres o más vértices. En las caras es donde se hará uso de los atributos pertenecientes a los vértices aplicando en ella los colores, texturas y fenómenos físicos que requieran de los vectores normales.

En el trabajo de Hoppe [2] una malla se define como un par $(\boldsymbol{V}, \boldsymbol{K})$ donde $\boldsymbol{V} = \{ \boldsymbol{v_i} \in \boldsymbol{R^3} \mid \boldsymbol{i} \in \{ \boldsymbol{1}, \ldots, \boldsymbol{m} \} \}$ es un conjunto de vértices en el espacio y K un complejo simplicial. Un complejo simplicial es un espacio topológico construido mediante puntos unidos por una conexión lineal, formando los polígonos que darán forma al modelo.

El conjunto simplicial está conformado por subconjuntos de $\boldsymbol{V}$ que representan los elementos de la malla. Los simplejos que contengan solo un elemento, son los vértices de la malla, los simplejos que contengan dos elementos, son las aristas de la malla, y los simplejos que contengan tres elementos, son las caras de la malla.

$$\{v_0\} = 0 - simplejo$$

$$\{v_0, v_1\} = 1 - simplejo$$

$$\{v_0, v_1, v_2\} = 2 - simplejo$$

En algunas definiciones se hará uso de operadores para englobar conjuntos de simplejos. Estas operaciones solo se utilizan como herramienta para resumir algunos conceptos, por lo que es posible que no se consigan otras referencias además de Lindstrom [3].

Los operadores ⌊ ⌋ ⌈ ⌉ que se denominarán en este trabajo como operadores de conjuntos. Funcionan de la siguiente manera. Siendo $\boldsymbol{s}$ un n-simplejo, el operador de conjunto inferior $\lfloor \boldsymbol{s} \rfloor$ retorna todos los elementos (n-1)-simplejo pertenecientes a $\boldsymbol{s}$ y el operador de conjunto superior $\lceil \boldsymbol{s} \rceil$ retorna todos los (n+1)-simplejos que contengan a $\boldsymbol{s}$ (ver **Figura 2)**.

Los 0-simplejos son todos los vértices de la malla. Los 1-simplejos son las aristas de la malla $\mathbf{e} = \{ \lfloor \boldsymbol{e} \rfloor_{\boldsymbol{0}}, \lfloor \boldsymbol{e} \rfloor_{\boldsymbol{1}} \}$. Si la arista es orientada se la expresa como $\vec{\boldsymbol{e}} = (\lfloor \boldsymbol{e} \rfloor_{\boldsymbol{0}}, \lfloor \boldsymbol{e} \rfloor_{\boldsymbol{1}})$. Todas las aristas se consideran orientadas y solo se diferenciarán entre $\bar{s}\ y\ \vec{\boldsymbol{s}}$ para resolver ambigüedades. Las aristas se pueden clasificar como borde, simple o múltiple, las aristas bordes solo coinciden con una cara $||\lceil \boldsymbol{s} \rceil|| = \mathbf{1}$, las aristas simples coinciden con exactamente dos caras $||\lceil \boldsymbol{s} \rceil|| = \mathbf{2}$, y las aristas múltiples coinciden con tres o más caras $||\lceil \boldsymbol{s} \rceil|| \geq \mathbf{3}$.

Los 2-simplejos son las caras de la malla $\boldsymbol{t} = \{ \vec{\boldsymbol{e}}^{\boldsymbol{t}}_{\boldsymbol{0}}, \vec{\boldsymbol{e}}^{\boldsymbol{t}}_{\boldsymbol{1}}, .., \vec{\boldsymbol{e}}^{\boldsymbol{t}}_{\boldsymbol{n}} \} = \{ (\boldsymbol{v}^{\boldsymbol{t}}_{\boldsymbol{0}}, \boldsymbol{v}^{\boldsymbol{t}}_{\boldsymbol{1}}), (\boldsymbol{v}^{\boldsymbol{t}}_{\boldsymbol{1}}, \boldsymbol{v}^{\boldsymbol{t}}_{\boldsymbol{2}}), .., (\boldsymbol{v}^{\boldsymbol{t}}_{\boldsymbol{n}}, \boldsymbol{v}^{\boldsymbol{t}}_{\boldsymbol{0}}) \}$. Por facilidad se expresará como $\boldsymbol{t} = (\boldsymbol{v}^{\boldsymbol{t}}_{\boldsymbol{0}}, \boldsymbol{v}^{\boldsymbol{t}}_{\boldsymbol{1}}, .., \boldsymbol{v}^{\boldsymbol{t}}_{\boldsymbol{n}})$ que significa $\{ (\boldsymbol{v}^{\boldsymbol{t}}_{\boldsymbol{0}}, \boldsymbol{v}^{\boldsymbol{t}}_{\boldsymbol{1}}), (\boldsymbol{v}^{\boldsymbol{t}}_{\boldsymbol{1}}, \boldsymbol{v}^{\boldsymbol{t}}_{\boldsymbol{2}}), \dots, (\boldsymbol{v}^{\boldsymbol{t}}_{\boldsymbol{n}}, \boldsymbol{v}^{\boldsymbol{t}}_{\boldsymbol{0}}) \}$.

En la **Figura 2** se exponen unos ejemplos del uso de los operadores de conjunto. (a) El operador de conjunto superior aplicado a un vértice retorna las aristas que contengan ese vértice. (b) Al aplicarle el operador de conjunto superior al resultado de (a) retorna las caras que contienen las aristas que contienen al vértice v. (c) Al aplicarle el operador de conjunto inferior al resultado de (a) retorna los vértices contenidos en las aristas que contienen al vértice v. (d) El conjunto inferior de una arista retorna los vértices que la componen. (e) Al aplicar el operador de conjunto superior al resultado de (d) retorna todas las aristas que contengan al menos uno de los vértices que contiene la arista *e*. (f) Al aplicar el operador de conjunto superior al resultado de (e) retorna las caras que contengan al menos una arista que contenga al menos uno de los vértices que contiene la arista *e*.

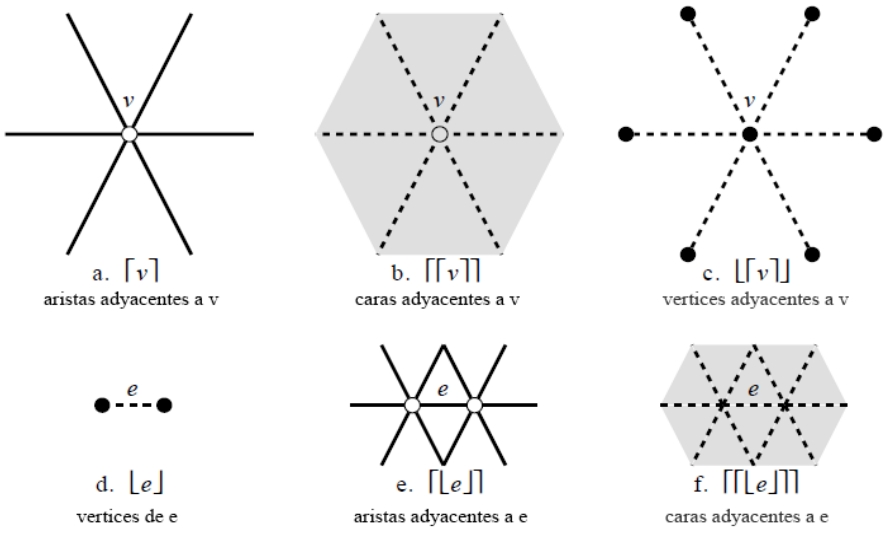


Figura 2: E*jemplo del uso de los operadores de conjunto* $\lceil s \rceil$ *y* $\lfloor s \rfloor$.

Existen otros elementos que posee la malla que se diferencian de estos primeros en que no contienen información referente a la topología de la malla. Estos atributos poseen información como el color, las coordenadas de la textura, vectores normales, que son utilizados en la fase de renderizado del modelo para añadir más detalles a la malla, como más realismo o una mejor apariencia.

Las fuentes más comunes para generar las mallas son cuando un artista o ingeniero crea el modelo utilizando alguna aplicación de modelado o cuando se escanea directamente de un objeto del mundo real. Esto genera un archivo que contiene todos los vértices de la malla y sus conexiones para formar la topología, en la mayoría de los casos, el elemento geométrico más utilizado para las mallas es el triángulo, por lo que es común que la mayoría de los formatos estén planteados originalmente para trabajar con triángulos, algunos ejemplos de estos formatos son:

1. .raw: Este es un formato básico. En este cada línea contiene tres vértices descritos de la forma: X1 Y1 Z1 X2 Y2 Z2 X3 Y3 Z3, lo que además de proporcionar los vértices, también describe las caras para formar la topología de la malla. Hay que tomar en cuenta que este formato no contiene información relacionada a los atributos, como las texturas o normales, solo la topología del modelo.

2. .obj: Este es un formato más completo. En este se especifican los vértices uno por uno, luego los vértices de textura, los vectores normales y las caras. También dispone de soporte para otros atributos como los materiales o vectores paramétricos, para un mayor nivel de detalle. A continuación se presenta un ejemplo de un archivo obj.

   #De esta forma se declara un comentario

   #Los vértices se declaran con una v seguida de las coordenadas (x, y, z) más una coordenada w opcional que será un valor de contrapeso utilizado para curvas racionales y superficies, si no se especifica su valor por defecto es 1
   …
   v -5.000000 5.000000 0.000000
   v 5.000000 5.000000 0.000000 1.000000
   …
   #Los vértices tendrán un índice según su orden comenzando por 1 que se utiliza para declarar otros elementos.
   #Las coordenadas de textura comienzan con vt y se especifican las coordenadas (x, y) y un valor opcional w que definirá la profundidad, por defecto es 0
   …
   vt 0.500000 1.000000 0.000000
   vt 1.000000 0.000000
   …
   #Los vectores normales comienzan con vn y la dirección del vector (x, y, z), estos vectores no necesariamente son vectores unitarios.
   …
   vn 0.707000 0.000000 0.707000
   ...
   #Las caras se definen comenzando con una f y tres o más grupos de índices que representan los vértices mencionados anteriormente, si solo se desea definir caras sin atributos solo se deben agregar los índices

de los vértices, si se desea agregar atributos, se concatena el índice del atributo al índice del vértice utilizando el símbolo "/" como separador.

...
#Vértices sin atributos
f 1 2 3
...
#Vértices con texturas
f 3/1 4/2 5/3
...
#Vértices con texturas y vectores normales
f 6/4/1 3/5/3 7/6/5
...
#Vértices con vectores normales sin texturas
f 6//1 3//3 7//5
...

3. .dae: Este formado comúnmente conocido como COLLADA, es usado con el fin de servir de formato de intercambio para aplicaciones interactivas en 3D. En este se define un esquema XML estándar con el que se puede intercambiar información digital entre diferentes aplicaciones gráficas que de otra manera usarían protocolos privados e incompatibles entre ellas.

   En este documento la información se guarda en un formato de etiquetas XML quedando de la siguiente forma:

```
<library_visual_scene>
        <visual_scene>
                <node>

<library_physics_scene>
        <physics_scene>
                <instance_physics_model>
                        <instance_rigid_body>
                                Target=""
                        <instance_ridig_constraint>
                        <instance_force_field>

<library_physics_models>
        <physics_model>
                <rigid_body>
                        <instance_physics_material>
                <rigid_constraint>

<library_physics_materials>
        <physics_material>

<library_force_fields>
```

```
        <force_fields>

    <scene>
        <instance_physics_scene>
        <instance_visual_scene>
```

Las mallas son utilizadas en una gran variedad de programas de computadora donde se busca representar de una manera más gráfica al usuario el resultado del programa. Algunos programas que hacen uso de las mallas son los editores de video con los que se pueden generar animaciones con altos niveles de detalle o agregar efectos especiales a un video. Otros serían los editores CAD (*Computer Aided Design/Drafting*), que son aplicaciones cuyo objetivo principal es el apoyo para la creación, modificación y análisis de un modelo geométrico con los que se pueden diseñar elementos a ser fabricados o impresos en 3D con una precisión de milímetros, los motores gráficos para videojuegos, los videojuegos buscan tener los gráficos más realistas y con el mayor nivel de detalle que el hardware disponible en su época pueda soportar. Un ejemplo de algunos programas que hacen uso de las mallas son:

1. AutoCAD: Es un conjunto de herramientas para el modelado en 3D dirigido para los aficionados a esta área de trabajo. Trae un conjunto de herramientas básicas de modelado junto a algunas opciones para exportar al formato STL (STereoLithography) que describe la superficie del modelo con un formato triangulado especificando únicamente los vértices ordenados y las normales de las caras. Este formato es ampliamente utilizado para las impresiones 3D y fabricación asistida por computadora [4].

2. Cinema4D: Es un programa profesional para realizar animaciones y modelos 3D. Es utilizado comúnmente para generar efectos especiales en televisión. Este tipo de programa se enfatiza en generar modelos realísticos, y la interacción entre estos queda en manos del usuario [5].

3. Video Juegos: Los video juegos son una de las áreas de la computación que más uso hacen de las mallas para sus representaciones gráficas. Un ejemplo de estas aplicaciones es la saga de juegos Mass Effect, implementado sobre el motor gráfico Unreal Engine. Este motor gráfico en su última versión incluyó una nueva técnica de iluminación en tiempo real con voxel desarrollada por la empresa Nvidia, y hace uso del motor de efectos de Nvidia physx. En este tipo de programas se hace énfasis en hacer lo más realista las físicas e interacciones entre los modelos y dar un nivel de detalle aceptable [6].

4. X-Plane y Google Earth: En estos programas su principal objetivo recae en otras áreas de la computación, pero hacen uso de la Computación Gráfica para hacer más inmersiva la experiencia. En el caso de X-Plane la Computación Gráfica tiene un mayor peso ya que su propósito es simular la experiencia de pilotar un avión con el mayor realismo (ver [7][8]).

1.3 Descripción del problema

Un aspecto que tienen en común la gran mayoría de programas que hacen uso de las mallas, es que al momento de desplegar la escena requieren de un variado número de mallas para representar el escenario y los elementos que interactúan en ella. Estas mallas que se despliegan pueden llegar a poseer magnitudes enormes de información. En los últimos años el nivel de detalle de las mallas ha llegado al nivel de utilizar cientos de millones de polígonos.

En el proyecto “The Digital Michelangelo” [9] de la universidad de Stanford en California, Estados Unidos, que comenzó oficialmente en 1997, se escaneó la famosa estatua de David de 1504 del escultor Michelangelo. La versión digital más grande generada por este proyecto contiene aproximadamente dos billones de polígonos y siete mil imágenes a color. Existen otros proyectos como “The Forma Urbis Romae Fragment” [10] también de la universidad de Stanford donde se escaneó un mapa de la antigua Roma. La versión digital completa de este mapa posee cerca de ocho billones de polígonos.

En el juego “Infamous Second Son” de Sony Computer Entertainment [11], los modelos de los personajes poseen cada uno alrededor de ciento veinte mil polígonos y hasta veintiocho megabytes en texturas, lo cual puede ser considerado una gran cantidad cuando coincidan muchos personajes en un mismo instante de tiempo.

Cuando una aplicación gráfica requiere de hacer uso de un modelo en malla, primero necesita cargarlo en memoria para poder acceder a su información interna. Esto no siempre es posible, debido a la alta exigencia de recursos. Esto genera un inconveniente, ya que esto reduce la cantidad de usuarios que serán capaces de ejecutar la aplicación en sus dispositivos de uso personal.

Una primera solución podría ser cargar en la memoria solo el segmento que se necesita de la malla en un momento dado y no tener que cargarla por completo. El inconveniente con esto es que el pasar información del disco duro a la memoria RAM es un proceso lento que puede causar un efecto cuello de botella con el resto de los componentes de la aplicación.

El objetivo es reducir la cantidad de polígonos que se necesitan desplegar sin tener que reducir el nivel de detalle de los modelos originales, esto se logra con algoritmos de simplificación, que alteran de manera temporal el modelo según los requerimientos del usuario, facilitando su despliegue sin alterar el modelo original.

1.4 Objetivo General

Desarrollar un algoritmo de simplificación de mallas en tiempo real aprovechando las capacidades de procesamiento paralelo de la GPU.

1.5 Objetivos Específicos

- Construir una solución computacional que permita, dada una malla como entrada, obtener una malla de salida con un menor nivel de detalle manteniendo las cualidades visuales originales.
- Crear un algoritmo de reducción de mallas para la GPU.
- Soportar la variación de los niveles de simplificación, incrementando el factor de reducción de elementos.
- Permitir visualizar el modelo original y el simplificado con el fin de realizar comparaciones entre ambos modelos.
- Proveer la extracción del modelo simplificado de la aplicación para su uso en otras aplicaciones.

Capítulo 2 Reducción de mallas

En este capítulo se describe los algoritmos clásicos para realizar simplificaciones de mallas, y algunos de los más recientes, donde se definen las funciones matemáticas utilizadas en el algoritmo de simplificación implementado en este trabajo.

2.1 Primeros algoritmos de simplificación de mallas

El objetivo principal de la simplificación de mallas es reducir el número de polígonos a desplegar de una forma dinámica y posible de regular. En la **Figura 3**, se puede observar el efecto de reducir el nivel de elementos en un modelo 3D.

Figura 3: *De izquierda a derecha: Ejemplo de la simplificación de malla con el modelo original y dos niveles diferentes de simplificación.*

La estructura general de los algoritmos de simplificación consiste en dos o tres etapas. En la primera se hace uso de una heurística para discernir el orden como se eliminarán los polígonos, aristas o vértices. Esto se logra categorizándolos según su disposición, o agregándoles un costo según su impacto en la topología o su distancia con respecto a la solución.

Otro trabajo que hizo avances importantes en esta fase de la simplificación fue Garland [13]. En este trabajo se plantea cada vértice $\boldsymbol{v}$ como la solución de un conjunto de planos. Estos planos se generan a partir de expandir las caras que poseen al vértice $\lceil\lceil \boldsymbol{v} \rceil\rceil$ como se expone en la **Figura 4**.

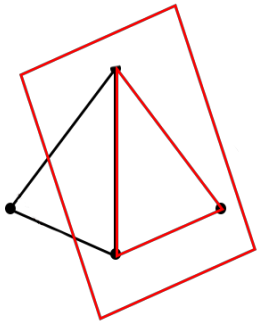


Figura 4: *Ejemplo del plano generado a partir de una cara.*

A cada vértice se le asigna un error que está dado por la distancia al cuadrado entre el vértice y estos planos, que originalmente será cero, y a medida que se colapsen aristas los nuevos vértices acumularán error según su distancia con respecto a los planos de los vértices colapsados. En la segunda fase, se eliminan o colapsan los elementos de la malla. El orden estará dictado por el resultado de la primera fase, dando prioridad a los elementos que poseen menos impacto en la topología general de la malla.

Se dice que un elemento es colapsado cuando se reduce su magnitud de elementos. La forma de colapso más común es el colapso de aristas. Esta operación consiste en contraer una arista a un nuevo vértice $\boldsymbol{x}$, colapsando en el proceso los polígonos que hagan uso de esta arista como se puede ver en la **Figura 5**.

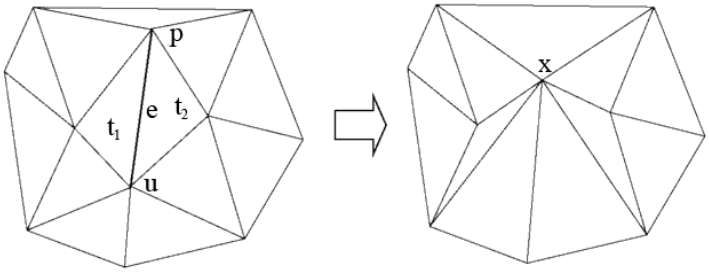


Figura 5: *Ejemplo de la operación de colapso de arista, la arista e es colapsada uniendo sus dos vértices p y u en un nuevo vértice x, eliminando en el proceso las caras* t_1 *y* t_2

Cuando se elimina un elemento de la malla se generan agujeros que afectan negativamente la apariencia de la malla, por lo que es deseable cubrir esa zona nuevamente. Se plantea un ejemplo de este proceso en la **Figura 6.**

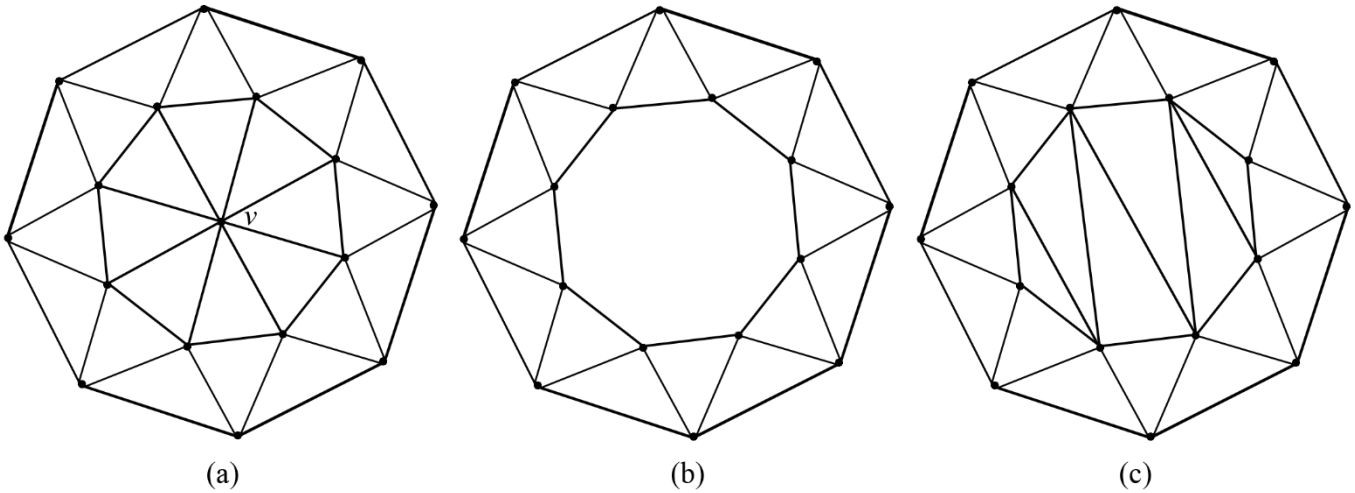


Figura 6: *Ejemplo del efecto de eliminar un vértice. (a) Malla original. (b) malla resultante al eliminar el vértice v. (c) Ejemplo de triangulación para cubrir la zona afectada. Nótese la diferencia en caras entre el ejemplo a (8 caras) y el ejemplo c (6 caras)*

Ya con algunos conceptos básicos aclarados, se explicarán los algoritmos de simplificación de algunas de las bibliotecas gráficas más relevantes.

2.2 Bibliotecas para la reducción de mallas

Con el pasar de los años, se ha trabajado y profundizado en el área de la simplificación de mallas, lo que ha llevado a los algoritmos de simplificación a estar en constante cambio y evolución. Existen muchos tipos de bibliotecas que proveen algoritmos visuales que ayudan a añadir interfaces a la aplicación, o desplegar fácilmente modelos y escenas. Otra clase de bibliotecas son las orientadas en algoritmos geométricos, cuya función es proveer algoritmos capaces de crear, manipular o desplegar elementos geométricos complejos como las mallas. A continuación se describe en detalle dos de las bibliotecas más utilizadas orientadas en algoritmos geométricos, que emplean algunos de los más resaltantes algoritmos de simplificación desarrollados.

2.2.1 Biblioteca CGAL

La primera biblioteca que describiremos será la biblioteca CGAL [14] (*The Computational Geometry Algorithms Library*) fundada como el proyecto CGAL, creada originalmente en 1996 por la ESPRIT (*European Strategic Programme for Research in Information Technology*) con la alianza de importantes institutos de Europa e Israel, (ver [15] [16] [17] [18] [19] [20] [21] [22]).

La biblioteca CGAL originalmente estaba distribuida bajo una licencia libre para uso académico y una licencia comercial para otros usos. Actualmente se encuentra bajo licencia GPL (*General Public License*) versión 3. CGAL ofrece algoritmos orientados en geometría computacional, escrita originalmente para el lenguaje C++, luego se agregó soporte a los lenguajes Python y Java.

La técnica de reducción estándar presentada en la biblioteca CGAL está fuertemente basada en los artículos [3] y [23], con algunas contribuciones de [2], [13] y [24]. El algoritmo usado en esta biblioteca desglosa en 2 etapas importantes.

La primera etapa, denominada etapa de recolección, consiste principalmente en asignarle un costo a cada arista de la malla. La segunda etapa, denominada etapa de colapso, consiste en ir evaluando las aristas según su costo de manera creciente, si la arista evaluada cumple ciertos requisitos geométricos y topológicos, esta se contrae y es reemplazada por un nuevo vértice. Se prosigue por recalcular el costo de todas las otras caras involucradas y se reorganizan según los nuevos costos las aristas restantes. En caso de no cumplir los requisitos, la arista es descartada del proceso de selección de aristas, esto es para prever que el contraer la arista lleve a un agujero en la malla u otros escenarios desfavorables.

Existen algoritmos que pueden contraer aristas inexistentes como lo es el caso de [13], en el cual se puede considerar a un par de vértices parte de una pseudo-arista, y contraer tanto aristas como pseudo-aristas, esto implica que se pueden generar aristas múltiples, pero el algoritmo utilizado por la biblioteca CGAL solo puede manejar aristas simples por lo que se omite esta solución.

Durante la etapa de recolección se itera por todas las aristas de la malla, y a cada arista se le realiza el cálculo de su costo según el impacto que tendría en la apariencia general de la malla de ser contraída, en la biblioteca CGAL se dispone de dos estrategias para calcular este costo, la primera es el algoritmo de Lindstrom, planteado en los artículos [3] y [23]. Este es el algoritmo que es utilizado por defecto en la biblioteca CGAL. En este algoritmo se toma en cuenta el impacto en la forma y volumen de la malla para calcular el costo.

La segunda estrategia para calcular el costo en la biblioteca CGAL, consiste en calcular el tamaño de la arista y asignarlo como su costo. Al contraer la arista (ver **Figura 7**) el nuevo vértice es colocado en el centro anterior de la arista. Esta estrategia es mucho más rápida pero tiene un gran impacto en la apariencia general de la malla, ya que en las zonas cóncavas se expande y en las zonas convexas se contrae.

Algoritmo de Lindstrom

En el algoritmo de Lindstrom se simplifica el modelo utilizando la operación de colapso de aristas. Para elegir la posición del nuevo vértice $\boldsymbol{x}$ se intenta minimizar ciertos cambios en la geometría de la malla como el volumen y el área, para conseguir esto se requiere resolver una ecuación lineal de la forma $\boldsymbol{a_i{}^t x = b_i}$ siendo $\boldsymbol{x}$ la intercepción entre tres planos no paralelos. Se requieren tres restricciones de este tipo, pero se utilizan más de tres restricciones $(\boldsymbol{a_n, b_n})$, esto en el caso de que dos o más de ellas sean linealmente dependientes. Si dos de estos planos $\boldsymbol{a}_n$ son cercanamente paralelos, pequeñas perturbaciones en los coeficientes del plano generan grandes variaciones en el resultado final. Para evitar esto, los planos deben cumplir las siguientes condiciones antes de poder ser aceptados:

$$(i) n = 1: a_1 \neq 0$$
$$(ii) n = 2: ({a_1}^T a_2)^2 < (\|a_1\|\|a_2\| \cos(\alpha))^2$$
$$(iii) n = 3: ((a_1 \times a_2)^T a_3)^2 > (\|a_1 \times a_2\|\|a_3\| \sin(\alpha))^2$$

La primera condición para poder aceptar el plano candidato (i) es que el ángulo del plano no sea igual a 0, la segunda y tercera condición (ii), (iii) descarta todo plano con una diferencia angular inferior a $\boldsymbol{\alpha}$ entre el plano candidato y los planos previamente aceptados.

Si estas condiciones se cumplen se puede decir que es compatible con los otros planos. Con los tres planos compatibles, conseguir $\boldsymbol{x}$ solo será resolver la siguiente ecuación.

$$x = A^{-1} b$$

Siendo $\boldsymbol{a_n}^{\boldsymbol{T}}$ las filas de la matriz $\boldsymbol{A}$.

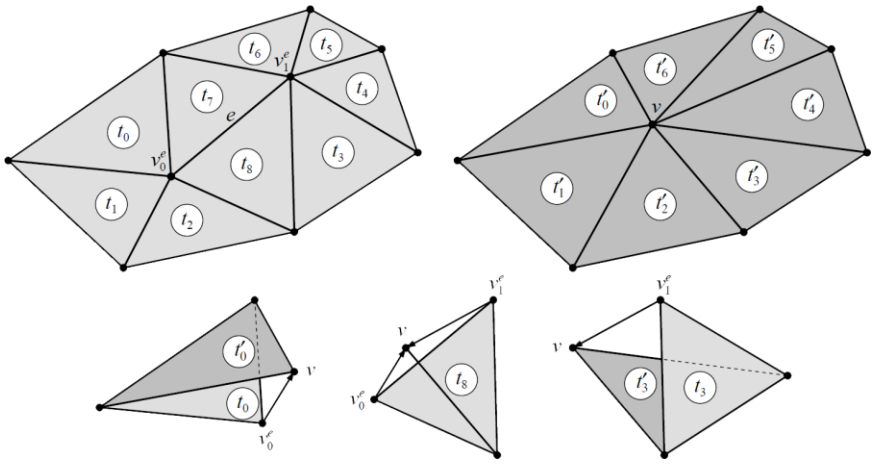


Figura 7*: Ejemplo de los tetraedros generados por el nuevo vértice v y las caras* t_0, t_3 *y* t_8

Varias de las ecuaciones se consiguen minimizando una función ligada a las ecuaciones ya conseguidas, esta función se expresa de la siguiente forma:

$$\begin{aligned} f(x) &= \tfrac{1}{2}x^T Ax - b^T x + \tfrac{1}{2}c \\ &= \tfrac{1}{2}(x^T\ 1)\begin{pmatrix} A & -b \\ -b^T & c \end{pmatrix}\begin{pmatrix} x \\ 1 \end{pmatrix} \\ &= \tfrac{1}{2}\bar{x}^T\bar{A}\bar{x} \end{aligned}$$

Donde $\overline{\boldsymbol{A}}$ la Hessiana de $\boldsymbol{f}$. La matriz Hessiana, es la matriz compuesta por las segundas derivadas de la función $\boldsymbol{f}$. Teniendo una forma como esta:

$$\bar{A} = \begin{pmatrix} \dfrac{\partial^2 f}{\partial x_1^2} & \dfrac{\partial^2 f}{\partial x_1 \partial x_2} & \cdots & \dfrac{\partial^2 f}{\partial x_1 \partial x_n} \\ \dfrac{\partial^2 f}{\partial x_2 \partial x_1} & \dfrac{\partial^2 f}{\partial x_2^2} & \cdots & \dfrac{\partial^2 f}{\partial x_2 \partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \dfrac{\partial^2 f}{\partial x_n \partial x_1} & \dfrac{\partial^2 f}{\partial x_n \partial x_2} & \cdots & \dfrac{\partial^2 f}{\partial x_n^2} \end{pmatrix}$$

Quedando $\overline{\boldsymbol{A}}$ como una matriz de tamaño $\boldsymbol{4 \times 4}$ semidefinida simétrica. Para minimizar $\boldsymbol{f}$ teniendo $\boldsymbol{n}$ ecuaciones $(\boldsymbol{a_i}, \boldsymbol{b_i})$ ya encontradas, definimos $\boldsymbol{Q}$ como una matriz de tamaño $\boldsymbol{3 - n \times 3}$ cuyas filas son ortogonales entre ellas y los vectores $\boldsymbol{a_i}$. Quedando las otras $\boldsymbol{3 - n}$ ecuaciones como.

$$Q(Ax - b) = 0$$

Donde $\boldsymbol{Ax - b}$ es el gradiente de $\boldsymbol{f}$. Por lo que el mínimo de $\boldsymbol{f}$ se obtiene cuando la proyección de su gradiente en el espacio de $\boldsymbol{Q}$ desaparece.

Preservación de volumen

A continuación se detalla como calcular la nueva posición del vector $\boldsymbol{x}$. El objetivo es conseguir un balance entre tres factores topológicos importante. Estos son, reducir el impacto en el volumen, en la geometría y en el área, para minimizar el cambio en el volumen de la malla, se debe calcular el volumen de los tetraedros creados entre el nuevo vértice $\boldsymbol{x}$ y los otros tres vértices de una cara $\boldsymbol{t_i}$. En la **Figura 7** se plantea el proceso de colapso de una arista y como este altera los demás elementos de la malla.

$$\begin{aligned} V(x, x^{t_i}{}_0, x^{t_i}{}_1, x^{t_i}{}_2) &= \frac{1}{6}\det(\bar{x} \;\; \bar{x}^{t_i}{}_0 \;\; \bar{x}^{t_i}{}_1 \;\; \bar{x}^{t_i}{}_2) \\ &= \frac{1}{6}((x^{t_i}{}_0 \times x^{t_i}{}_1 + \; x^{t_i}{}_1 \times x^{t_i}{}_2 + \; x^{t_i}{}_2 \times x^{t_i}{}_0)^T x \; - [x^{t_i}{}_0, x^{t_i}{}_1, x^{t_i}{}_2]) \\ &= \frac{1}{6}((x^{t_i}{}_0 \times x^{t_i}{}_1 + \; x^{t_i}{}_1 \times x^{t_i}{}_2 + \; x^{t_i}{}_2 \times x^{t_i}{}_0)^T \; - [x^{t_i}{}_0, x^{t_i}{}_1, x^{t_i}{}_2])\bar{x} \\ &= \frac{1}{6}\bar{G}_{V_i}\bar{x} \end{aligned}$$

El volumen queda representado en la forma de una matriz de tamaño $\boldsymbol{1 \times 4}$ asociada a la cara $\boldsymbol{t_i}$. Al contraer la arista $\boldsymbol{a}$ se añade o substraer volumen de la malla, por lo que se desea que el volumen generado entre el nuevo vértice y los triángulos tales que $\{\boldsymbol{t_i}\} = \lceil\lfloor \boldsymbol{a} \rfloor\rceil$ sea lo más cercano a 0.

$$\frac{1}{6}\sum_i \bar{G}_{V_i}\bar{x} = 0$$

Esto limita la solución a un plano y se puede añadir esto como la primera restricción lineal para **x.** $\boldsymbol{f_{V_p}}$ se puede expresar como un problema cuadrático indeterminado, quedando expresado de la siguiente forma:

$$f_{V_p}(x) = \frac{1}{2}\bar{x}^T \bar{A}_{V_p}\bar{x} = \frac{1}{2}\bar{x}^T \left(\frac{1}{18}\sum_i \bar{G}_{V_i}{}^T \sum_i \bar{G}_{V_i}\right)\bar{x}$$

De esta forma se tiene que $\boldsymbol{f_{V_p}}$ es la distancia al cuadrado entre $\boldsymbol{x}$ y la solución óptima (el plano que no altera el volumen de la malla) por lo que nada diferente de $\boldsymbol{f_{V_p} = 0}$ será un mínimo garantizado, como son varias las restricciones que se deben cumplir, como preservar el volumen, área, geometría, no existe un valor $\boldsymbol{x}$ que minimice todas las restricciones por lo que se debe designar un peso a cada función con el cual calcular el costo de la arista.

Optimización de volumen

Calcular el volumen de los tetraedros asociados al vértice, determina el volumen que se agrega o substrae al volumen general de la malla, pero también se debe tomar en cuenta el volumen general sin signo que se genera, con esto se puede calcular la desviación general del volumen después de contraer la arista, como es común se utiliza el cuadrado de los valores en vez de su valor absoluto para resolver los cálculos.

$$f_{V_0}(x) = \frac{1}{2}\bar{x}^T \bar{A}_{V_0}\bar{x} = \frac{1}{2}\bar{x}^T \left(\frac{1}{18}\sum_i \bar{G}_{V_i}{}^T \bar{G}_{V_i}\right)\bar{x}$$

Se calcula similar a la ecuación de preservación de volumen.

Optimización de la topología

En las zonas planas donde $\boldsymbol{f_{V_0}}$ es igual a cero, cualquier valor que se elija de $\boldsymbol{x}$ dejará intacto el volumen general de la malla, existen casos donde el mejor resultado se obtiene al elegir valores de $\boldsymbol{x}$ que eviten incluir triángulos delgados y alargados, buscando una mayor uniformidad en la forma general de los triángulos involucrados, esto se logra minimizando la suma de las longitudes al cuadrado de las aristas $\lceil \boldsymbol{x} \rceil$, y maximizar el área por perímetro de los triángulos $\left\lceil \lceil \boldsymbol{x} \rceil \right\rceil$ quedando:

$$x - x_i = (I - x_i)\bar{x} = \bar{G}_{S_i}\bar{x}$$

Donde $\boldsymbol{\bar{G}_{S_i}}$ está asociada al vértice formado por $\boldsymbol{x}$ y un vértice adyacente $\lfloor \lceil \boldsymbol{x} \rceil \rfloor - \{\boldsymbol{x}\} = \{\boldsymbol{x_i}\}$, quedando así la función objetiva con la siguiente forma:

$$f_S(x) = \frac{1}{2}\bar{x}^T \bar{A}_S \bar{x} = \frac{1}{2}\bar{x}^T \left(2 \sum_i \bar{G}_{S_i}{}^T \bar{G}_{S_i} \right) \bar{x}$$

De esta forma se obtiene una solución única para $\boldsymbol{x}$.

2.2.2 Biblioteca VTK

La biblioteca VTK (*Visual ToolKit*) [25] fue originalmente creada con el fin de servir como apoyo práctico para el libro "The Visualization Toolkit: An Object-Oriented Approach to 3D Graphics" en el año 1993, escrita por Schroeder, Ken Martin y Bill lorensen. Escrita para el lenguaje C++, recientemente se introdujo soporte para Python y Java.

En esta biblioteca se presentan cuatro métodos de simplificación: vtkDecimate, vtkDecimatePro, vtkQuadricDecimation y vtkQuadricClustering, a continuación se explicará brevemente cada uno.

Función vtkDecimate

Esta función está basada en el trabajo de Schroeder [12]. Donde se hacen varias iteraciones sobre todos los vértices, si el vértice cumple con ciertos criterios, se elimina junto con todos los triángulos que utilicen este vértice, luego se cubrirá el agujero creado al eliminar estos triángulos con una nueva triangulación. El algoritmo se divide en 3 pasos:

1. Caracterizar cada vértice según su geometría y topología
2. Evaluar el criterio de simplificación y eliminar el vértice según sea el caso
3. Re-triangular el agujero creado en el Paso 2

Caracterización del vértice según su geometría y topología

El primer paso es caracterizar los vértices según la geometría y topología local por cada uno. Con el resultado de este paso se podrá determinar si un vértice es un potencial candidato a ser eliminado, y que criterio utilizar.

En la **Figura 8** se listan una serie de etiquetas con las cuales se catalogan los vértices según una de las siguientes categorías: simple, complejo, borde, arista interior o vértice esquina.

Figura 8: *Diferentes categorías de vértices*

Un vértice simple está rodeado por un círculo completo de caras, y cada arista que usa al vértice es usada por exactamente dos caras. En caso de ser utilizado por una cara que no se encuentra dentro del círculo que lo rodea se le denominará complejo.

Un vértice que se encuentre en el borde de la malla, o que esté dentro de un semicírculo de caras, será un vértice borde. Un vértice simple puede a su vez clasificarse como una arista interior, esto depende de la geometría local, ya que si el ángulo diedro (el ángulo formado entre dos planos que parten de una arista común) entre dos caras adyacentes es superior a cierto valor predefinido, existirá una arista característica. Si un vértice es utilizado entre dos aristas características, el vértice es categorizado como arista interior. Cuando un vértice es utilizado por tres o más aristas características este se clasificara como vértice esquina. Los vértices complejos no serán eliminados, todos los demás son candidatos para eliminar.

Criterio de simplificación

En este paso se determina cuando los triángulos que componen el círculo que encierra el vértice seleccionado pueden ser eliminados o no. Para los vértices simples se utiliza la distancia del vértice al plano como criterio, en la **Figura 9** se muestra un ejemplo de esta distancia. Si la distancia está dentro del rango predefinido, puede ser eliminado, de lo contrario se conserva el vértice.

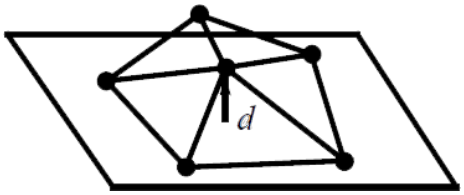


Figura 9: *Distancia del vértice con respecto al plano promediado entre los vértices que lo rodean.*

Los vértices borde y arista interior utilizan como criterio la distancia al borde. En la **Figura 10** se puede ver un ejemplo de este escenario. En este caso el algoritmo calcula la distancia entre la línea definida por los dos vértices que crean el borde o arista característica. Si la distancia es menor que un valor *d* predefinido, el vértice puede ser eliminado.

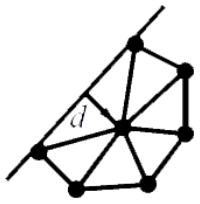


Figura 10*: Distancia entre el vértice y el borde o arista característica*

Si un vértice puede ser eliminado, el área que se removió debe ser triangulada nuevamente. Se denominará bucle al área creada por eliminar el vértice. Como se muestra en la **Figura 11**, se tratará de dividir el bucle trazando una línea entre dos vértices que lo compongan, con el fin de tener tres o más vértices en ambos lados. Si el bucle se puede dividir de esta manera, se triangula cada mitad por separado.

Triangulación

Al eliminar un vértice y sus triángulos asociados se debe generar un nuevo conjunto de vértices que los reemplacen. Para cada bucle se debe realizar una triangulación cuyos triángulos no se intercepten ni degeneren. En adición, es importante que al momento de generar nuevos triángulos se tome en cuenta la calidad visual del modelo.

La triangulación se realiza con una división de bucle recursiva. Cada bucle se divide en dos y cada mitad se trabaja individualmente, la división está dada por la línea generada por dos vértices no adyacentes en el bucle. La recursión se detendrá cuando queden solo tres vértices, los cuales serán el triángulo que se guardará en la malla y se finalizará el proceso.

Como se aprecia en la **Figura 11**, cuando el bucle no es plano y tiene forma de estrella, el bucle debe dividirse según un plano de división, que será ortogonal al plano promedio generado por los vértices del bucle. Esto para evitar que se sobrepongan los nuevos triángulos. Si se cumple la condición de que existan 3 únicos vértices de un lado del plano se acepta el plano de división, en caso de no poder generarse un plano de división que cumpla este requisito se toma como un fallo y se revierte el colapso del vértice.

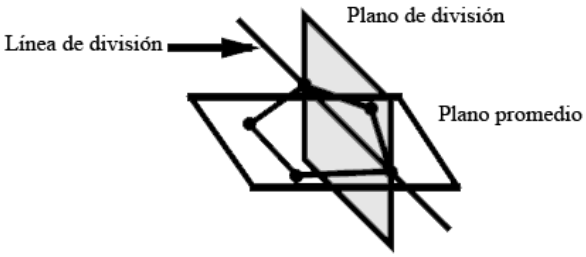


Figura 11: *Ejemplo de un plano de división que divide en dos el bucle generado al eliminar el vértice.*

Como existen muchas formas de realizar esta división, el objetivo es tomar la solución que reduzca la diferencia con la malla original, por lo que se debe utilizar una relación de aspecto, con la cual se medirá la distancia entre el bucle y el plano de división dividido por el tamaño de la línea de división. El mejor plano es el que maximice la relación de aspecto, si se define un umbral mínimo de distancia para aprobar un plano por ejemplo 0.1, se generan resultados aceptables.

Función vtkDecimatePro

El cambio más importante entre esta función y vtkDecimate, es la generación progresiva de mallas, que está basado en el algoritmo presentado en el trabajo de Hoppe de mallas progresivas [26]. Este algoritmo igual que vtkDecimate está basado en el trabajo de Schroeder, con la diferencia de que los vértices son clasificados según la distancia que hay entre el vértice seleccionado y el plano que se genera promediando las normales de todos los vértices que componen las caras pertenecientes el bucle de triángulos que rodea el vértice (bucle mencionado anteriormente que es formado al eliminar un vértice). Después de ser clasificados, se procesan y la malla es simplificada. Entonces la malla es dividida según las aristas características y complejas para repetir el proceso recursivamente en cada uno de estos fragmentos de malla según una profundidad específica.

Mallas progresivas

En Hoppe [2] se describe un método de simplificación de malla que con el uso de tres transformaciones simples: colapso de arista, división de arista e intercambio de arista, simplifica la topología de las mallas. En este se plantea utilizar solo la transformación simple de colapso de arista, que llaman $ecol(\{v_s, v_t\})$, y se plantea una función inversa llamada separación de vértice. Esta función, denominada *vsplit(s,l,r,t,A)* toma un vértice $\boldsymbol{v_s}$ y agrega uno nuevo $\boldsymbol{v_t}$ y dos nuevas caras $\{\boldsymbol{v_s}, \boldsymbol{v_t}, \boldsymbol{v_l}\}$ y $\{\boldsymbol{v_s}, \boldsymbol{v_t}, \boldsymbol{v_r}\}$ (si la arista $\{\boldsymbol{v_s}, \boldsymbol{v_t}\}$ es una arista borde, se tiene $\boldsymbol{v_r} = \mathbf{0}$ y solo se agrega una cara).

Una malla inicial $\boldsymbol{M^n}$ se puede simplificar a una malla más simple $\boldsymbol{M^0}$ utilizando una secuencia de **n** transformaciones $ecol(\{v_s, v_t\})$:

$$M^n \xrightarrow{ecol_{n-1}} \dots \xrightarrow{ecol_1} M^1 \xrightarrow{ecol_0} M^0$$

Y como la transformación de colapso de arista es reversible podemos representar una malla $\boldsymbol{M^n}$ como una serie de transformaciones aplicadas a una malla simple $\boldsymbol{M^0}$:

$$M^0 \xrightarrow{vsplit_0} M^1 \xrightarrow{vsplit_1} \dots \xrightarrow{vsplit_{n-1}} M^n$$

Geomorphs

Una propiedad que se puede resaltar en este planteamiento de mallas progresivas es la transición visualmente fluida entre dos mallas $\boldsymbol{M^i}$ y $\boldsymbol{M^{i+1}}$ (una geomorph) que se puede lograr. Asumiendo que la malla no contiene más atributos además de la posición de los vértices, se puede plantear las divisiones de arista como:

$$vsplit_i(s_i, l_i, r_i, A_i = (v_{s_i}^{i+1}, v_{m_0+i+1}^{i+1}))$$

La idea es construir un geomorph $\boldsymbol{M^G(\alpha)}$ cuyo parámetro varíe entre $\mathbf{0} \leq \boldsymbol{\alpha} \leq \mathbf{1}$ de tal forma que $\boldsymbol{M^G(0)}$ sea $\boldsymbol{M^i}$ y $\boldsymbol{M^G(1)}$ sea $\boldsymbol{M^{i+1}}$, y así la aplicación puede pasar de una malla a otra de una forma fluida. Ya que la transformación $ecol$ es transitiva, se puede generar una transformación compuesta de una secuencia de transformaciones $ecol$, con esto se puede construir un geomorph entre cualquier malla $\boldsymbol{M^n}$ y otra $\boldsymbol{M^i}$.

Dada una malla $\boldsymbol{M^c}$ y otra versión más simplificada de esta $\boldsymbol{M^f}$, existe una correspondencia indirecta entre cada vértice de $\boldsymbol{M^f}$ con un único vértice de $\boldsymbol{M^c}$, esta relación estará dada por la transformación compuesta $\boldsymbol{A^c}$ que será la

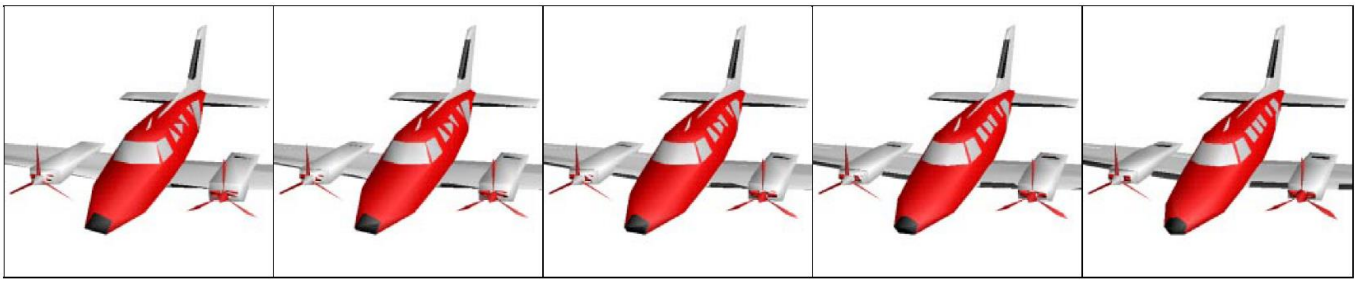
unión de todas las transformaciones $ecol$. Más precisamente, cada vértice $\boldsymbol{v_j}$ de $\boldsymbol{M^f}$ corresponde a un vértice $\boldsymbol{v_{A^c(j)}}$ en $\boldsymbol{M^c}$

$$A^c(j) = \begin{cases} j & , \quad j \leq m_0 + c \\ A^c\left(S_{j\,-\,m_0-1}\right) & , \quad j > m_0 + c \end{cases}$$

Con esta correspondencia se pueden generar geomorphs $\boldsymbol{M^G(\alpha)}$ tal que $\boldsymbol{M^G(0)}$ sea $\boldsymbol{M^c}$ y $\boldsymbol{M^G(1)}$ sea $\boldsymbol{M^f}$, simplemente definiendo el geomorph tal que $\boldsymbol{M^G(\alpha) = (K^f, V^G(\alpha))}$ para tener la conectividad de $\boldsymbol{M^f}$ y la posición de los vértices

$$v_j^G(\alpha) = (\alpha)v_j^f + (1-\alpha)v_{A^c(j)}^c$$

En la **Figura 12** podemos ver un ejemplo de un geomorph entre dos mallas, una $\boldsymbol{M^{175}}$ y otra $\boldsymbol{M^{425}}$



(a) α = 0.00 (b) α = 0.25 (c) α = 0.50 (d) α = 0.75 (e) α = 1.00

Figura 12*: Ejemplo de un geomorph $M^G(\alpha)$ definido entre $M^G(0) = M^{175}$(con 500 caras) y $M^G(1) = M^{425}$ (con 1000 caras)*

Función vtkQuadricDecimation

En esta se utiliza una nueva métrica de error cuadrática descrita en el trabajo de Hoppe [27]. El algoritmo de simplificación utilizado es el mismo que en vtkDecimate y vtkDecimatePro.

Nueva métrica cuadrática para simplificación de mallas

En la mayoría de los algoritmos de simplificación, se busca reducir el nivel de detalle de la malla con una secuencia de colapsos de aristas, estas transformaciones tienen la ventaja de poder almacenarse su inversa y con esta generar una presentación progresiva de la malla.

En un esquema basado en colapsos de arista se deben resolver 2 cuestiones. La primera es asignar la posición y atributos unificados al nuevo vértice $\boldsymbol{v}$ y la segunda es ordenar como se realizan estos colapsos. Una manera muy común de resolver esto es definir una métrica $\boldsymbol{C}$ para determinar ambos. A $\boldsymbol{v}$ se le asigna un valor minimizando $\boldsymbol{C(v)}$ que se utiliza también para ordenar los candidatos para ser colapsados.

En este trabajo se define la función $\boldsymbol{C(v)}$ como una función geométrica cuadrática basada en el trabajo de Garland [13] con la diferencia, de que en vez de incluir los atributos en los cálculos ampliando las dimensiones de los vectores ej.: $\boldsymbol{v \in R^{3+m}}$ se calculan de manera separada.

En el caso de no poseer atributos, en otras palabras que **m=0,** se calcula para cada cara la distancia de un punto $\boldsymbol{v} = (\boldsymbol{p}) \in \boldsymbol{R^3}$ al plano que contiene la cara. Y a cada vértice se le asigna la sumatoria de las distancias de las caras que contienen al vértice, ponderada por el área de la cara:

$$Q^v(v) = \sum_{f \ni v} area(f) \cdot Q^f(v).$$

Derivando $\boldsymbol{Q^f(v)}$ para una cara $\boldsymbol{f} = (\boldsymbol{v_1}, \boldsymbol{v_2}, \boldsymbol{v_3})$ recordando que se está planteando el caso **m=0**. La distancia de un punto **p** a un plano $\boldsymbol{P} \subset \boldsymbol{R^3}$ que contiene a *f* es $\boldsymbol{n^t p} + d$, teniendo que la normal **n** se puede expresar como $\boldsymbol{n} = (\boldsymbol{p_2} - \boldsymbol{p_1}) \times (\boldsymbol{p_3} - \boldsymbol{p_1}) / \|(\boldsymbol{p_2} - \boldsymbol{p_1}) \times (\boldsymbol{p_3} - \boldsymbol{p_1})\|$ y el escalar $\boldsymbol{d} = -\boldsymbol{n^t p_1}$. Una manera de conseguir estos parámetros es resolviendo el siguiente sistema de ecuaciones.

$$\begin{pmatrix} p_1^T & 1 \\ p_2^T & 1 \\ p_3^T & 1 \end{pmatrix} \begin{pmatrix} n \\ d \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}$$

Con la condición de que $\|\boldsymbol{n}\| = \boldsymbol{1}$. La distancia cuadrada de entre un punto **p** y el plano **P** es.

$$Q^f\big(\boldsymbol{v} = (\boldsymbol{p})\big) = (\boldsymbol{n}^T v + d)^2 = \boldsymbol{v}^T(\boldsymbol{n}\boldsymbol{n}^T)v + 2d\boldsymbol{n}^T\boldsymbol{v} + d^2$$

Que puede ser representado como una función cuadrática $\boldsymbol{v^T A v + 2 b^T v + c}$ donde **A** es una matriz simétrica de tamaño $\boldsymbol{3 \times 3}$, **b** es un vector de tamaño 3, y **c** un escalar, quedando como.

$$Q^f = (\boldsymbol{A}, \boldsymbol{b}, \boldsymbol{c}) = ((\boldsymbol{n}\boldsymbol{n}^T), (d\boldsymbol{n}), d^2)$$

Como se mencionó antes, en el trabajo de Garland [13], se añaden los atributos de los vértices y caras de la malla (normales, texturas, etc.), incrementando la dimensión donde se trabaja, y calculando todo al mismo tiempo, pasando de calcular la distancia de un punto al plano que contiene a la cara, se calcula la distancia de un punto al híper-plano que contenga no solo a la cara, sino también a los atributos, lo que conlleva a que no necesariamente se busque el punto geométricamente más cercano, sino un punto geométrico probablemente más lejano, con una mejor aproximación a los atributos. Para compensar esto se hace uso de un atributo $\boldsymbol{\lambda_j}$ de precisión elegido por el usuario que va a denotar la importancia de cada atributo $\boldsymbol{j} \in \{\boldsymbol{1} \ldots \boldsymbol{m}\}$.

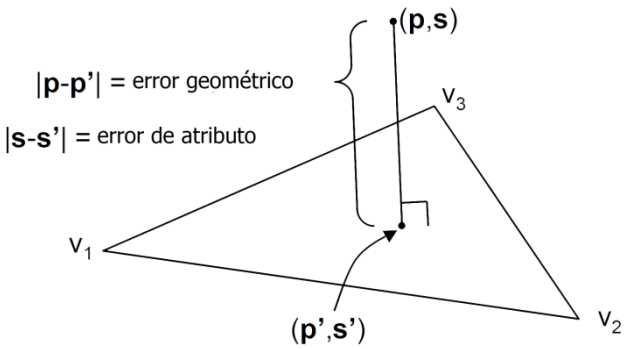


Figura 13: *Correspondencia entre el punto* $\boldsymbol{p}$ *con sus atributos* $\boldsymbol{s}$ *y su proyección en el plano que contiene a la cara* (v_1, v_2, v_3).

En [2], como se muestra en la **Figura 13**, se calcula de manera diferente, en vez de calcular todo en una misma función de dimensión $\boldsymbol{R}^{3+m}$, aquí se proyecta en $\boldsymbol{R}^3$ y se calculan por separado.

De esta forma la métrica de error para una cara $\boldsymbol{f}$ queda como la suma.

$$Q^f\left(\boldsymbol{v} = \binom{\boldsymbol{p}}{s}\right) = Q_p^f(\boldsymbol{v}) + \sum_{j=1}^{m} Q_{s_j}^f(\boldsymbol{v})$$

Donde el error geométrico $\boldsymbol{Q_p^f(v)}$ es la distancia cuadrada desde p a una proyección $\boldsymbol{p'}$ en $\boldsymbol{R^3}$ que contiene $\boldsymbol{f}$, y el error de los atributos $\boldsymbol{Q_{s_j}^f(v)}$ es la desviación cuadrática entre **s** y la interpolación $\boldsymbol{s'}$ de los atributos de la cara $\boldsymbol{f}$.

El error geométrico es solo una versión extendida con ceros del trabajo de Garland [13]:

$$Q_p^f = (\boldsymbol{A}, \boldsymbol{b}, \boldsymbol{c}) = \left(\begin{pmatrix} \boldsymbol{nn}^T & \ddots\ 0\ \ddots \\ \ddots\ 0\ \ddots & \ddots\ 0\ \ddots \end{pmatrix}, \begin{pmatrix} d\boldsymbol{n} \\ 0 \end{pmatrix}, d^2\right)$$

Para formar el término de error de atributos $\boldsymbol{Q_{s_j}^f}$ se define primero una función lineal

$$\hat{s}_j(\boldsymbol{p}) = \boldsymbol{g}_j^T\boldsymbol{p} + d_j$$

Que representa los valores esperados para todos los atributos de todos los puntos $\boldsymbol{p} \in \boldsymbol{R^3}$. El gradiente $\boldsymbol{g_j}$ y el escalar $\boldsymbol{d_j}$ se definen de la siguiente manera. Como $\hat{\boldsymbol{s}}_{\boldsymbol{j}}(\boldsymbol{p})$ debe interpolar los vértices de la cara $\boldsymbol{f} = (\binom{p_1}{s_1}, \binom{p_2}{s_2}, \binom{p_3}{s_3})$ y coincidir la interpolación lineal sobre el plano $\boldsymbol{P}$. Adicionalmente $\hat{\boldsymbol{s}}_{\boldsymbol{j}}(\boldsymbol{p})$ para un punto arbitrario $\boldsymbol{p} \in \boldsymbol{R^3}$ debe ser idéntico al valor $\hat{\boldsymbol{s}}_{\boldsymbol{j}}(\boldsymbol{p'})$ por ser su proyección en el plano $\boldsymbol{P}$; esto es equivalente a predefinir $\boldsymbol{n^T g_j = 0}$. Se pueden conseguir los valores de $\boldsymbol{g_j}$ y $\boldsymbol{d_j}$ resolviendo este sistema de ecuaciones.

$$\begin{pmatrix} \boldsymbol{p}_1^T & 1 \\ \boldsymbol{p}_2^T & 1 \\ \boldsymbol{p}_3^T & 1 \\ \boldsymbol{p}_4^T & 0 \end{pmatrix} \begin{pmatrix} g_j \\ d_j \end{pmatrix} = \begin{pmatrix} s_1, j \\ s_2, j \\ s_3, j \\ 0 \end{pmatrix}$$

Y $\boldsymbol{Q}_{s_j}^{\boldsymbol{f}}(\boldsymbol{v}) = (\hat{\boldsymbol{s}}_{\boldsymbol{j}}(\boldsymbol{p}) - \boldsymbol{s}_{\boldsymbol{j}})^{\boldsymbol{2}} = (\boldsymbol{g}_{\boldsymbol{j}}^{\boldsymbol{T}}\boldsymbol{p} + \boldsymbol{d}_{\boldsymbol{j}} - \boldsymbol{s}_{\boldsymbol{j}})^{\boldsymbol{2}}$, se puede expresar como:

$$\left(\begin{pmatrix} g_j g_j^T & \ddots\ 0\ \ddots & -g_j & \ddots\ 0\ \ddots \\ \ddots\ 0\ \ddots & \ddots\ 0\ \ddots & 0 & \ddots\ 0\ \ddots \\ g_j^T & \cdots 0 \cdots & 1 & \cdots 0 \cdots \\ \ddots\ 0\ \ddots & \ddots\ 0\ \ddots & 0 & \ddots\ 0\ \ddots \end{pmatrix}, \begin{pmatrix} d_j g_j \\ 0 \\ -d_j \\ 0 \end{pmatrix}, d_j^2 \right)$$

Donde el valor 1 aparece en $\boldsymbol{A_{3+j,3+j}}$ y el valor $-\boldsymbol{d_j}$ aparece en $\boldsymbol{b_{3+j}}$.

Juntando todas estas expresiones cuadráticas, queda:

$$Q^f = (\boldsymbol{A}, \boldsymbol{b}, c) =$$
$$\left(\begin{pmatrix} \boldsymbol{n}\boldsymbol{n}^T & \sum_j \boldsymbol{g}_j \boldsymbol{g}_j^T & -\boldsymbol{g}_1 & \cdots & -\boldsymbol{g}_m \\ & -\boldsymbol{g}_1^T & & & \\ & \vdots & & I & \\ & -\boldsymbol{g}_m^T & & & \end{pmatrix}, \begin{pmatrix} d\boldsymbol{n} + \sum_j d_j \boldsymbol{g}_j \\ -d_1 \\ \vdots \\ -d_m \end{pmatrix}, d^2 + \sum_j d_j^2 \right)$$

Las primeras 3 filas y las primeras 3 columnas de la matriz **A** son densas, pero el resto de la sub-matriz $\boldsymbol{m} \times \boldsymbol{m}$ es la identidad **I**. Al añadir el atributo $\boldsymbol{\lambda_j}$ de precisión $Q = \boldsymbol{Q_p} + \sum_j \boldsymbol{\lambda_j^2 Q_{s_j}}$. La sub-matriz $\boldsymbol{m} \times \boldsymbol{m}$ pasa a ser una matriz con el escalar $\boldsymbol{\lambda_j^2}$ en su diagonal, por lo que solo se necesita de una variable. Todo en conjunto requiere de **11 + 4*m*** variables, lo que hace su costo de almacenamiento lineal en $\boldsymbol{m}$.

Función vtkQuadricClustering

En esta implementación se usa un método completamente diferente a las anteriores 3 soluciones. Aquí se hace uso de la agrupación de vértices y la métrica de error cuadrática descrito en el trabajo de Lindstrom [27].

En el trabajo de Lindstrom [27] se presenta un híbrido, que separa los vértices por grupos como lo presenta el trabajo de Rossignac [28]. Con la integración del error métrico cuadrático presentado por Garland [13] y después mejorado por Lindstrom [23].

La matriz $\boldsymbol{A}$ propuesta originalmente por Garland [13], se asume es invertible, aunque en práctica esto no necesariamente es el caso, como por ejemplo cuando la superficie es plana o tiene cero curvatura gaussiana.

Para resolver esto Lindstrom [23] proponen resolver el problema asegurándose que este sobre-condicionado, quedando los valores del determinante de $\boldsymbol{A}$ lo suficientemente grandes.

En este caso por el contrario solo se utilizan tres condiciones, y se trata el problema de una manera un poco diferente, teniendo a $\boldsymbol{x}$ como la proyección ortogonal del centro de la celda en el espacio de todas las soluciones a $\boldsymbol{Ax = b}$. Para lograr esto se aplica una descomposición en valores singulares $\boldsymbol{A = U\Sigma V^T}$, que para una matriz semidefinida simétrica positiva $\boldsymbol{A}$ es equivalente a hacer una descomposición de valor propio. Esto se puede lograr utilizando rotaciones de Jacobi.

$$\sigma_i^+ \begin{cases} 1/\sigma_i & si \quad \sigma_i/\sigma_1 > \epsilon \\ 0 & si\ no \end{cases}$$

Donde $\boldsymbol{\sigma_1}$ es el mayor valor singular y $\boldsymbol{\epsilon}$ es un valor umbral, definido en dicho trabajo como $\boldsymbol{10^{-3}}$. Por lo tanto, el vértice **x**, más cercano al centro de la celda $\hat{\boldsymbol{x}}$ que satisface $\boldsymbol{Ax = b}$ es

$$x = \hat{x} + V\Sigma^+ U^T(b - A\hat{x})$$

Lo que se puede resumir en $\boldsymbol{A^{-1}b}$ siempre que $\boldsymbol{\Sigma^+ = \Sigma^{-1}}$.

Ya estudiadas dos de las bibliotecas más relevante en algoritmos geométricos, se presenta un breve resumen de algunos algoritmos modernos que cambian la forma en que se plantea la simplificación de malla tradicional.

2.3 Técnicas modernas de reducción de mallas

En la constante evolución de los algoritmos para la simplificación de mallas se han hecho importantes avances que han cambiado la forma en la que se planteaba originalmente como se debería simplificar una malla. Entre los más resaltantes trabajos que han innovado en esta área se encuentran los que se estudian a continuación.

2.3.1 Billboard clouds para simplificación extrema de modelos

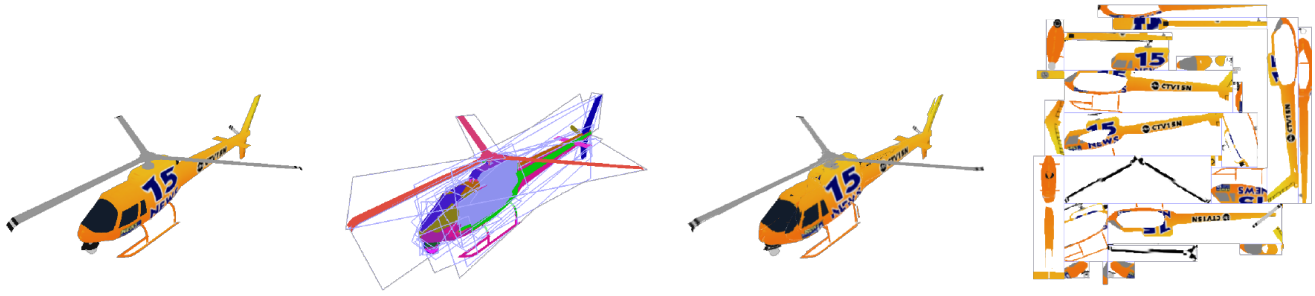
En esta técnica presentada por Décoret et al. [29] se busca tratar el problema con un enfoque diferente. Tradicionalmente las técnicas de reducción de mallas buscan eliminar vértices o aristas progresivamente para disminuir la cantidad de polígonos a desplegar y así aligerar la carga de trabajo, y la diferencia principal entre las diferentes técnicas es el impacto en el nivel de detalle y el consumo de recursos y tiempo. En la técnica presentada en este trabajo se busca transpolar los vértices de textura de las caras de la malla a unos nuevos planos que se irán definiendo según la topología de la malla, la ventaja de esta técnica es que se pueden hacer reducciones extremas de las mallas, pasando de 5000 o más caras, a solo unas 32 manteniendo un nivel aceptable de detalle.

Descripción general del algoritmo

Como se muestra en la **Figura 14**, se comienza generando un conjunto de planos parcialmente transparentes de tamaño, orientación y resolución de textura independientes. Estos planos se generan de tal forma que se acoplen a la geometría del modelo y se ira proyectando en ellos toda la información de las caras del modelo, conservando valores como el mapa de normales para ser utilizado por el píxel *shader*.

La forma en la que se generan estas *billboard clouds* se puede expresar como un problema de optimización definido por una función de error, y una función de costo. La función de error se basa en la norma euclidiana $\boldsymbol{L_\infty}$ que se computa en el espacio-objeto y es independiente de la vista.

La estrategia en la que se basa la función de error, es definir un error máximo tolerable $\boldsymbol{\varepsilon}$, y el objetivo es generar los planos que estén dentro de ese umbral tal que el costo sea mínimo, el algoritmo es una optimización exhaustiva que va seleccionando los planos capaces de proyectar la mayor cantidad de caras posibles.



(a) (b) (c) (d)

***Figura 14**: Ejemplo de Billboard Cloud, (a) Modelo original con 5138 caras, (b) Billboards unicolores para mostrar cómo se agruparon las caras, (c) Billboards con las texturas cargadas solo 32 polígonos desplegados (d)Ejemplo de las texturas generadas para los planos.*

Para estimar la relevancia de un plano con respecto de otro, se utiliza la densidad definida en base a la validez, cobertura y la penalización. La validez del plano se expresa por medio de un booleano que dicta si un plano simplifica una cara o no, la cobertura se utiliza como medida de calidad, evitando asignar a un plano una cara con mala proyección.

Penalización es la métrica para descartar planos que no entren dentro del umbral aceptable. Y así queda definida la densidad $\boldsymbol{d}(\boldsymbol{P})$ como:

$$\boldsymbol{d}(\boldsymbol{P}) = \boldsymbol{C}(\boldsymbol{P}) - \boldsymbol{Penalty}(\boldsymbol{P})$$

Donde $\boldsymbol{C}(\boldsymbol{P})$ es la cantidad de caras para las que el plano $\boldsymbol{P}$ es una representación validad, se le denomina cobertura de $\boldsymbol{P}$.

Para calcular si un plano simplifica una cara se utiliza la distancia euclidiana (La longitud de una línea recta que conecte ambos puntos) entre los vértices de la cara y el plano, y esta debe ser menor a ε, quedando que los planos válidos entran en un rango en forma de esfera alrededor de cada vértice. Los planos que son una representación valida del vértice $\boldsymbol{v}$ se describen como $\boldsymbol{valid}_{\boldsymbol{\varepsilon}}(\boldsymbol{v})$ que son los planos que interceptan la esfera de radio $\boldsymbol{\varepsilon}$, de la misma forma para un plano $\boldsymbol{P}$, $\boldsymbol{valid}(\boldsymbol{P})$ son todas las caras que cubre $\boldsymbol{P}$. En la **Figura 15** se puede ver un ejemplo del rango donde un plano es válido para simplificar un vértice.

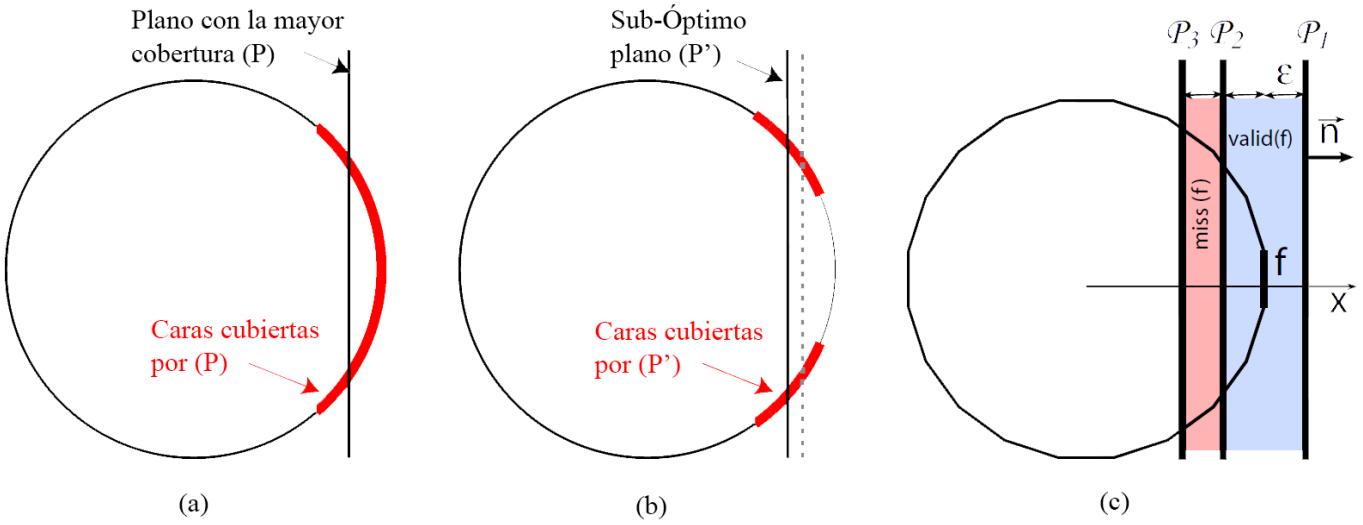


Figura 15: *Proximidad y penalización por proyecciones no adecuadas*

Una opción es utilizar $\boldsymbol{valid(P)}$ como una métrica para medir la relevancia de un plano con respecto a otro, pero si se hace esto, un plano que cubre varias caras pequeñas tendría mayor relevancia que uno que cubre una cara de mayor área. Este comportamiento podría traer consecuencias no deseadas. Por esto se calcula el área de la proyección de la cara en el plano. Dando de esta manera un mayor peso a las caras con mayor área proyectada sobre el plano.

$$\boldsymbol{C(P)} = \sum_{\boldsymbol{f \in valid_{\varepsilon}(P)}} \boldsymbol{area_p(f)}$$

Algunas veces existen planos que son óptimos para unas caras pero fallan en capturar todos los atributos de las demás caras, por esto se le da la capacidad a las caras de penalizar aquellos planos que no las proyectan adecuadamente.

Para una normal $\vec{\boldsymbol{n}}$ los planos válidos son los que se encuentran a una distancia no mayor a $\boldsymbol{\varepsilon}$, donde $\boldsymbol{\varepsilon}$ es el umbral de error que define el usuario, también se toma en cuenta los que se encuentren a $\boldsymbol{\varepsilon}$ distancia de ser válidos por la izquierda, estos se agrupan como $\boldsymbol{miss_{\varepsilon}(f)}$. Se podrían considerar $\boldsymbol{valid_{\varepsilon}(f)}$ y $\boldsymbol{miss_{\varepsilon}(f)}$ como partes contiguas del segmento $\boldsymbol{2\varepsilon}$ **y** $\boldsymbol{\varepsilon}$, quedando la penalización como:

$$\boldsymbol{Penalización(P)} = \boldsymbol{w_{penalización}} \sum_{\boldsymbol{f \in miss(P)}} \boldsymbol{area_p\,(f)}$$

Los planos se definen usando las coordenadas esféricas $\boldsymbol{(\theta, \varphi)}$ para la dirección normal, y la distancia $\boldsymbol{\rho}$ hasta el origen, en este espacio, los planos válidos para cada vértice serán representados como superficies producidas por la función $\boldsymbol{\rho = f(\theta, \varphi)}$ (ver **Figura 16**). Se calcula el dominio variando $\boldsymbol{\rho}$ en un rango $\boldsymbol{-\varepsilon}$ **y** $\boldsymbol{\varepsilon}$.

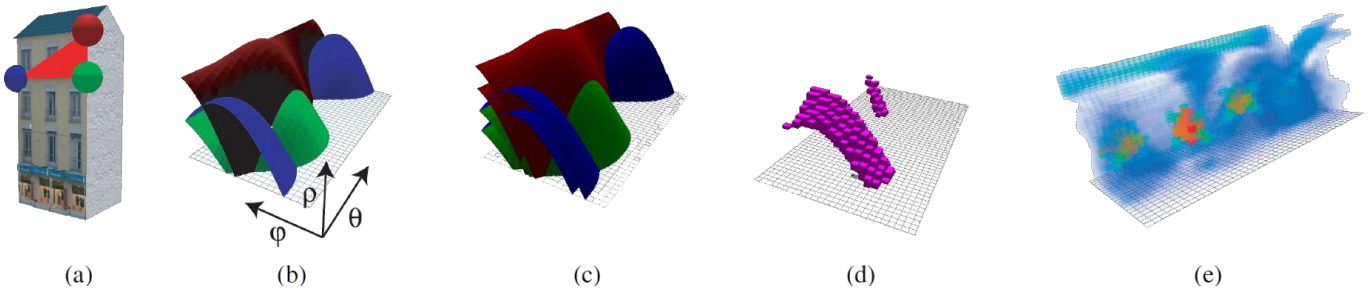


Figura 16: *(a) Se resalta la cara f y los vértices que al componen. (b) Conjunto de planos válidos para cada vértice según la función* $\boldsymbol{\rho = f(\theta, \varphi)}$. *(c) Dominio valido para cada plano (entre* $\boldsymbol{-\varepsilon}$ *y* $\boldsymbol{\varepsilon}$ *). (d) Discretización de la intercepción del dominio valido de cada plano (e) Cobertura general de la malla.*

Es espacio debe ser dividido en bins $\boldsymbol{B_{\theta_i,\varphi_j,\rho_k}}$ y en cada bins calcularse su densidad. Para una cara $\boldsymbol{f}$ dada, $\boldsymbol{valid_\varepsilon(f)}$ representa todos los bins que pertenecen al conjunto solución para la cara $\boldsymbol{f}$. Estos bins se consideran válidos para $\boldsymbol{f}$ si en ellos existe al menos un plano que sea válido para $\boldsymbol{f}$, esto es calculable iterando en las coordenadas $\boldsymbol{\theta, \varphi}$.

Con unas ciertas coordenadas $\boldsymbol{\theta, \varphi}$ y una cara $\boldsymbol{f}$ se calcula el rango $[\boldsymbol{\rho_{min}, \rho_{max}}]$ que intercepta $\boldsymbol{valid_\varepsilon(f)}$, se calcula $\boldsymbol{C(f)}$ que es constante para una dirección dada, y se suma como la cobertura de los bins entre $\boldsymbol{\rho_{min}}$ y $\boldsymbol{\rho_{max}}$ y como penalización para los bins entre $\boldsymbol{\rho_{min} - \varepsilon}$ y $\boldsymbol{\rho_{min}}$. Para tener en cuenta el antialising $\boldsymbol{C(f)}$ es ponderado por el porcentaje del bin que está cubierto.

Los planos se generan en base a las zonas más densas de estos bins, buscando intercepciones entre los $\boldsymbol{valid(f_i)}$, dando mayor prioridad al plano que cubra más área total. Luego, se itera sobre el conjunto de bins con mayor densidad, en busca de un plano donde se proyecten todas las caras del conjunto. Una vez seleccionado el plano, se actualiza la densidad removiendo los valores de las caras ya colapsadas y se prosigue a buscar el siguiente bin con mayor densidad.

Refinamiento adaptativo

Cada plano P_i tiene asignado un conjunto $\boldsymbol{valid_\varepsilon^F(P_i)}$, el cual contiene las caras que serán proyectadas en él. Por cada cara primero se conseguirá su *bounding box*, y se proseguirá por proyectarlas ortogonalmente en el plano, el tamaño de las texturas se calcula según el tamaño de su *bounding box*.

En muchas ocasiones un plano es solución de varias caras al mismo tiempo, para aumentar el detalle y reducir cualquier espacio en blanco que pueda quedar entre diferentes billboards se proyecta la textura en todos los planos que sean solución para la cara. Para añadir iluminación también se debe recalcular el mapa de normales proyectado en los bounding box para ser añadido al plano.

2.3.2 Simplificación de mallas en tiempo real utilizando la GPU

Un aspecto importante de la Computación Gráfica, es el *pipeline* gráfico, que son los pasos necesarios para desplegar una representación 2D en el monitor de la escena. En este *pipeline* existen pasos donde un desarrollador puede cargar su propio código con el fin de alterar el resultado de este *pipeline*.

Una de las principales diferencias entre la GPU y la CPU (*Central Processing Unit*) es la gran cantidad de procesos que esta puede ejecutar en paralelo, contando con hasta los 5632 núcleos. Cuando la CPU comúnmente maneja entre 2 y 32 núcleos.

Durante un prolongado periodo de tiempo la simplificación de malla se consideró un algoritmo exclusivo de la CPU, por la complejidad de las operaciones necesarias para simplificar un modelo, pero con la introducción de los *shaders* que daba la posibilidad de interactuar con la geometría de la malla durante el *pipeline*, se abrió la posibilidad de realizar estas operaciones complejas en el despliegue de la malla.

Métodos exitosos como los propuestos por Garland [13] y Lindstrom [3] y [23] están pensados para trabajar en la CPU, esto conlleva tiempos de procesamiento que hacen imposible realizar simplificaciones en tiempo real. Por esto se busca en este trabajo incluir las siguientes contribuciones:

- Reformular el trabajo de "Vertex Clúster" de Lindstrom [27] para adoptar el ambiente de trabajo paralelo de la GPU
- Una estructura de datos en forma de árbol octal (una estructuras de dato en forma de árbol que cada nodo tiene exactamente 8 hijos) para propósito general en la GPU
- Simplificación de malla adaptativa con requerimientos fijos de memoria
- Preservación de detalles según importancia utilizando *warping* no lineal

En la conferencia SIGGRAPH del año 2000, Peter Lindstrom presentó su trabajo "Out-of-Core Simplification of large Polygonal Models" [27], en donde añadía al trabajo de Rossignac y Borrel [28], la métrica cuadrática para calcular la solución y error estimado, presentado en el trabajo Lindstrom [3] y [23].

En Lindstrom [27], una vez definido el número de subdivisiones que tendrá la rejilla, se carga la malla original un triángulo por vez, se identifica en que celda cae cada vértice del triángulo y se marca con el índice de la celda. Si la celda no ha sido inicializada, se genera un índice (normalmente en orden incremental) y se crea la matriz de error cuadrático con ceros. Si dos o más vértices de un mismo triángulo caen sobre la misma celda, se colapsa y descarta el triángulo, de lo contrario se agregan los vértices al grupo V_{out} y la cara al grupo T_{out}.

Antes de cargar el siguiente triángulo se calcula el error cuadrático y se suma a la celda de cada uno de sus vértices.

El error cuadrático para un triángulo t está dado por.

$$t = (x_1, x_2, x_3)$$
$$Q = \begin{pmatrix} A & -b \\ -b^T & c \end{pmatrix} = nn^T$$
$$n = \begin{pmatrix} x_1 \times x_2 + x_2 \times x_3 + x_3 \times x_1 \\ -[x_1, x_2, x_3] \end{pmatrix}$$

Una vez sumados todos los errores cuadráticos de los vértices en los errores cuadráticos acumulados en sus respectivas celdas, se calcula el vector representativo de la celda y se despliega la malla simplificada.

En el trabajo de DeCoro y Tatarchuk [30] se utiliza esta estrategia de simplificación *out-of-core* (algoritmos elaborados para trabajar con volúmenes de información demasiado grandes para ser almacenada en la memoria del

computador) presentada por Lindstrom, y la capacidad de procesamiento paralelo de la GPU para realizar simplificaciones en tiempo real con niveles de detalle aceptables.

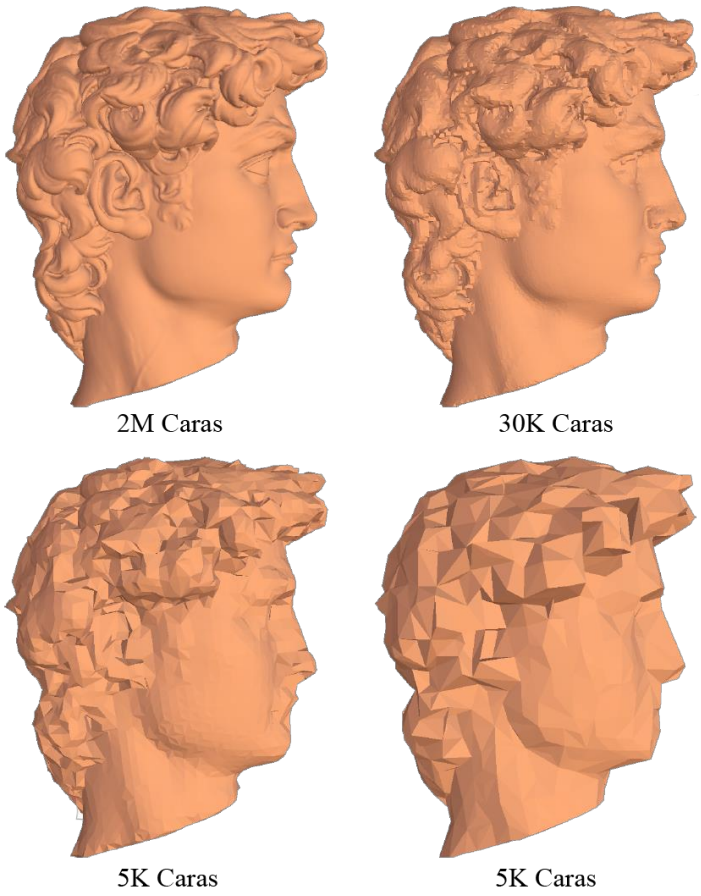


Figura 17: *Utilizando el algoritmo de simplificación en tiempo real en la GPU de DeCoro y Tatarchuk se pueden generar estos cuatro niveles de detalles un orden de magnitud más rápido que un algoritmo que reside en la CPU que genera solo 1 nivel de detalle.*

Descripción general del algoritmo

El algoritmo DeCoro y Tatarchuk se puede reducir a 3 pasos sencillos y requiere que se cargue la malla dos veces durante el *pipeline* de renderizado. El modelo se divide en una rejilla de tamaño definido por el usuario, y se genera un *off-screen buffer* (para realizar el renderizado sin que se despliegue en pantalla) que sirve de arreglo bidimensional para almacenar la data que será trabajada por el *pipeline* gráfico. El tamaño del *buffer* dependerá del nivel de detalle solamente, permitiendo que el algoritmo pueda trabajar con mallas de cualquier tamaño. Los pasos que realizará este algoritmo como se muestra en la **Figura 18** son los siguientes:

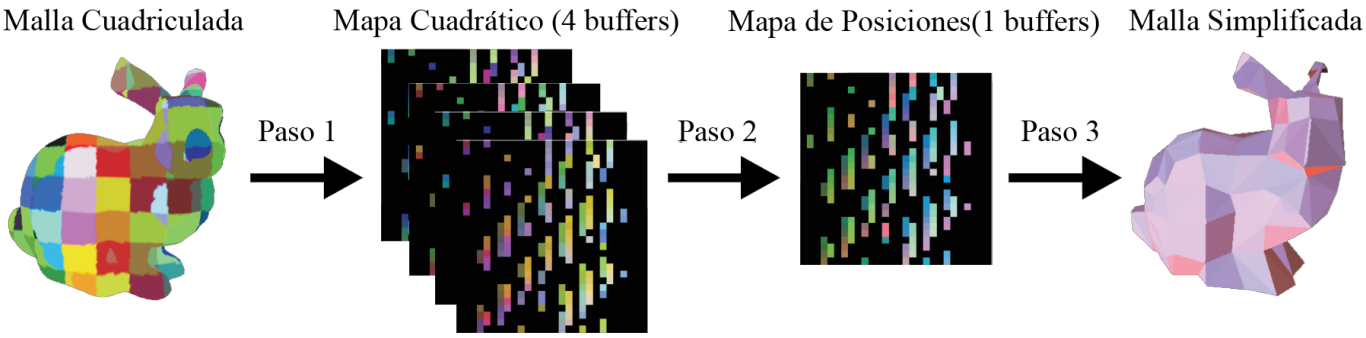


Figura 18: *La malla original es dividida en una rejilla de 9x9x9. En el paso 1 se calcula el error cuadrático para cada celda y se almacena en los buffers que contienen el error cuadrático en valores de píxel. En el paso 2 se minimiza el error cuadrático para calcular el vértice representativo de cada celda como se explicó en el algoritmo de Lindstrom [27]. En el paso 3 se genera la malla simplificada con la información del paso 2.*

Paso 1: Generación del mapa de grupo cuadrático. Dada una malla y su respectivo *bounding box*, junto con el número de divisiones para cada dimensión, se renderizan los vértices, y se le asigna un identificador único a cada celda creada al dividir el *bounding box*. Se considerara el *buffer* como un arreglo indexado por los identificadores de las celdas. Cada campo del arreglo va a almacenar la suma del error cuadrático para esa celda (10 valores punto flotante para una matriz de tamaño $\mathbf{4 \times 4}$ simétrica), la posición de vértice promedio en esa celda (3 valores punto flotante) y el conteo de vértices.

El *vertex shader* determina que celda corresponde a cada vértice y su posición en el arreglo, y el *geometry shader*, que tiene acceso a todos los vértices en el triángulo utiliza su posición global y calcula su error cuadrático de la cara, y se lo asigna a cada uno de sus vértices para que se acumule en el mapa de textura por el píxel *shader* que propaga los colores calculados con composición aditiva habilitado.

Paso 2: Calcular la posición óptima del vértice representativo. Utilizando el mapa cuadrático generado en el Paso 1, se calcula el vértice representativo de cada celda. Se dibuja un cuadrado del mismo tamaño de los *buffers* utilizados en el Paso 1 en un nuevo *buffer*.

En el píxel *shader* se toman los valores de error cuadráticos generados antes y se calcula la posición óptima que tendrá el nuevo vértice con inversión de matriz. Si el determinante de la matriz está por debajo de un umbral impuesto por el usuario (por ejemplo $\mathbf{1e^{-10}}$) se toma como singular, por lo tanto se utiliza una posición promediada y se guarda el resultado en el nuevo arreglo.

Paso 3: Generación de la malla simplificada. Se vuelve a pasar al *pipeline* la malla original para mover los vértices a su posición simplificada, y eliminar los que se hayan descartado. El *vertex shader* identifica la celda a la que pertenece cada vértice nuevamente, y el *geometry shader* verifica si los 3 vértices de la cara están en una misma celda, si es así se elimina la cara. En caso de no ser así se retorna la nueva posición según el resultado del Paso 2, que será la nueva posición del triángulo que será almacenada en un *buffer* de la GPU para uso posterior.

Se pueden calcular múltiples niveles de detalle sin tener que repetir todos los pasos. Si la resolución se reduce a la mitad se omite el Paso 1, y se crea el mapa cuadrático reduciendo la resolución del mapa cuadrático anterior. El Paso 2 se ve inalterado, pero el Paso 3 puede utilizar la malla ya simplificada anteriormente como entrada, ya que la conectividad es la misma.

Esto permite la creación de diferentes niveles de detalle (LOD), considerablemente más rápido que realizar todos los pasos nuevamente.

División en rejilla no uniforme usando una función de warping

Una manera para incrementar la adaptabilidad del algoritmo, es deformar la malla original y aumentar el nivel de detalle en zonas de mayor complejidad. Se puede aplicar una función para deformar la malla durante el proceso de simplificación, pero se aplica la operación inversa al calcular el error cuadrático. El único cambio al utilizar esta función de *warping* es que altera en que celda caen los vértices, para poder calcular con precisión el error cuadrático es importante no alterar la posición de los vértices ni el espacio ocupado por la rejilla.

Un uso importante para esta función de *warping* es para simplificaciones dependientes del punto de visión, donde el algoritmo da prioridad a las zonas más cercanas al espectador, se puede observar un ejemplo de esto en la **Figura 19**. La forma más simple de lograr esto es aplicando la matriz de transformación *World View Projection* donde los diferentes modelos son pasados a un único espacio global, luego transformados a un espacio dependiente de la posición del espectador y por ultimo transformados al espacio de proyección donde se proyecta la imagen en una escena plana, en este último comúnmente se aplica la transformación de perspectiva. Transformaciones dependientes del espectador son útiles para aplicaciones animadas en tiempo real.

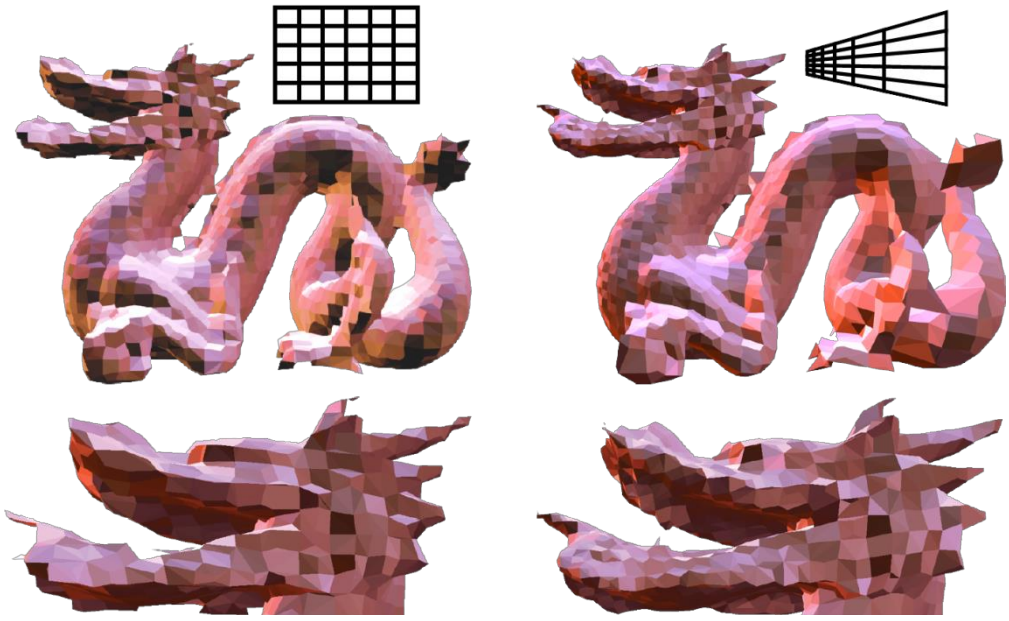


Figura 19: *A la izquierda un ejemplo de la simplificación en tiempo real sin utilizar función de warping. A la derecha el mismo ejemplo utilizando una función de warping dependiente de la posición del espectador, que en este caso está situado a su izquierda.*

Como se muestra en la **Figura 21**, otro uso importante de esta funcionalidad es el incrementar el nivel de detalle en zonas relevantes para el usuario, por lo que se debe permitir utilizar funciones de *warping* que hagan especial énfasis en las zonas importantes, para lograr esto se usa una función de contrapeso gaussiano $\boldsymbol{f}(\boldsymbol{x})$ centrada en el punto de interés quedando como

$$
\begin{aligned}
f_{\mu,\sigma,b}(x) &= (1-b)G_{\mu,\sigma}(x) + b \\
\hat{F}_{\mu,\sigma}(x) &= \int_{-\infty}^{x} G_{\mu,\sigma}(t)dt \\
&= \frac{1}{2}\left(1 + \text{erf}\frac{x-\mu}{\sigma\sqrt{2}}\right) \\
F_{\mu,\sigma,b}(x) &= \frac{\hat{F}_{\mu,\sigma}(x) - \hat{F}_{\mu,\sigma}(0)}{\hat{F}_{\mu,\sigma}(1) - \hat{F}_{\mu,\sigma}(0)}(1-b) + bx
\end{aligned}
$$

En esta definición, la ecuación $\boldsymbol{G_{\mu,\sigma}}(\boldsymbol{x})$ es la distribución normal estándar, y **erf(.)** es la función de error gaussiano. Se define un parámetro de balance ***b***, que determina un contrapeso mínimo para regiones fuera del área de interés, si se establece que ***b=1*** es equivalente a un muestreo uniforme (ver **Figura 20**).

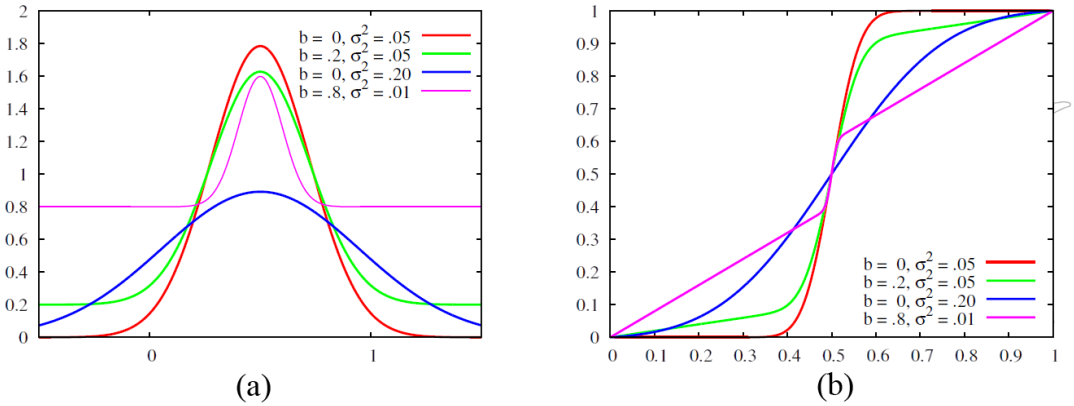


Figura 20: *Ejemplo de las funciones de warping con diferentes parámetros σ y b (a) ejemplo de la función* $f_{\mu,\sigma,b}(x)$. *(b) ejemplo de la función* $F_{\mu,\sigma,b}(x)$

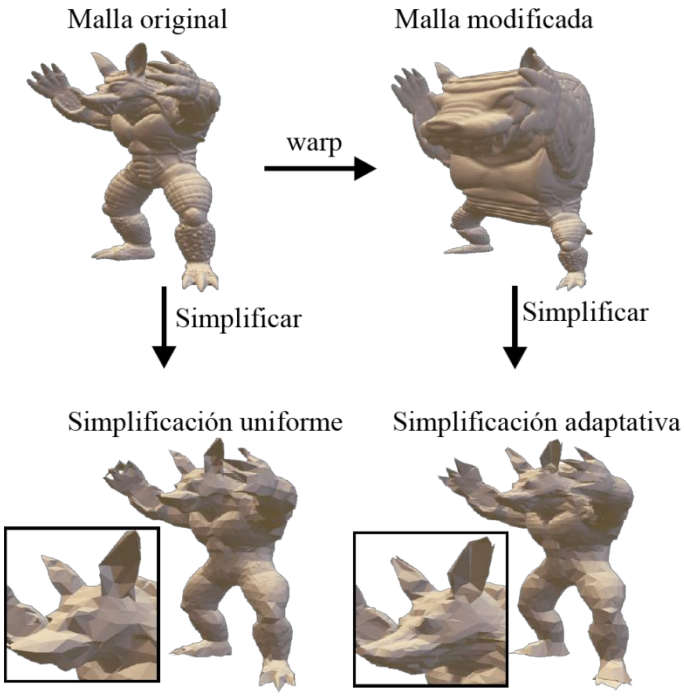


Figura 21: *Ejemplo de la simplificación adaptativa, a la izquierda un modelo simplificado tradicionalmente, y a la derecha un modelo simplificado utilizando una función de warping*

Árboles octales probabilísticos

Al usar una rejilla de tamaño definido, se debe introducir el tamaño de la rejilla antes de comenzar el proceso de simplificación, y no permite fácilmente incluir niveles impares de precisión. Y la información debe ser almacenada en un gran arreglo de tamaño fijo para que las direcciones de memoria puedan ser calculadas directamente, sin importar la cantidad de clúster que realmente tengan información.

La propuesta consiste en utilizar una estructura de datos, que permita múltiples niveles de resolución, desde la más baja hasta una más detallada, donde cada nivel contiene el doble de detalle que el anterior. Cada celda va almacenar el error cuadrático estimado de una rejilla de un nivel de detalle determinado. Cuando se mapea un vértice de la malla a la rejilla en el paso de simplificación, esta representación permite un nivel de muestro superior en zonas con más detalles.

En la **Figura 22** se aprecia un ejemplo de la distribución en un árbol octal de la celda de división.

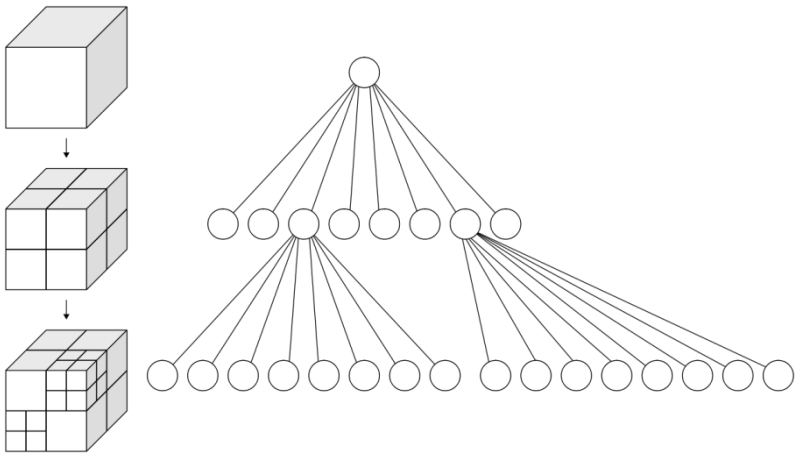


Figura 22: *Ejemplo de un árbol octal representando los diferentes niveles de subdivisión en un cubo.*

Operaciones: El octal define las operaciones de alto nivel ADDVERTEX (v) y FINDCLUSTER (v) utilizadas en el Paso 1 y Paso 3, respectivamente. Estas hacen uso de las operaciones de bajo nivel WRITE (*k*,*d*) y *d*=READ (*k*) para escribir o leer la información ***d*** en la posición ***k***.

Construcción probabilística: Durante la creación del árbol en el Paso 1, se utiliza la operación ADDVERTEX en cada vértice a insertar en el árbol. En un árbol de profundidad máxima l_{max}, un vértice tiene l_{max} niveles en donde puede ser ubicado. Una implementación de ADDVERTEX podría hacer l_{max} corridas por el árbol, para asignar cada vértice en cada posible nivel. Esto resultaría en la construcción del árbol más precisa, pero el tiempo de simplificación crecería proporcionalmente.

Alternativamente, se puede considerar el error cuadrático de la rejilla $\boldsymbol{Q_c}$ como el resultado de integrar los errores cuadráticos $\boldsymbol{Q_x}$ en cada punto $\boldsymbol{x}$ en la superficie contenida en ***C***, escalado según el área diferencial **dA**. En una malla con una gran cantidad de polígonos, se puede aproximar esta cantidad tomando la suma del error cuadrático de los vértices $\boldsymbol{Q_v}$ contenidos en ***C***, que son calculados por las caras adyacentes $\boldsymbol{Q_f}$ y sus correspondientes áreas $\boldsymbol{A_f}$.

$$Q_c = \int_{x \in C} Q_x dA \approx \sum_{v \in C} \sum_{f \in adj(v)} Q_f \frac{A_f}{3}$$

Para reducir la cantidad de muestreos, se propone asignar el nivel de los vértices de manera aleatoria. En vez de una distribución aleatoria uniforme, utiliza una función que crece exponencialmente en niveles superiores, y como existen exponencialmente menos vértices en niveles inferiores, el muestreo se mantiene relativamente igual.

Almacenamiento probabilístico: Como con el almacenamiento uniforme, se almacenan los niveles del árbol octal, en *buffers off-screen*, utilizándolos como un arreglo, y dividiéndolo en secciones por cada nivel. Una vez ADDVERTEX(*v*) calcule el nivel donde se almacenará ***v***, se genera un índice ***k*** para el vértice y se invoca WRITE(*k*,*v*) para guardarlo. La función WRITE utiliza una funciona *hash* de distribución uniforme para guardar los vértices en el *buffer*, para reducir el espacio de almacenamiento, se reserva espacio para menos nodos de los que son necesarios para el nivel entero. Por lo que la probabilidad de que WRITE(*k*,*v*) sea válido se espera sea igual al porcentaje de nodos ocupados en ese nivel.

Acceso al árbol: Después de crear el árbol en el Paso 1, se utilizará FINDCLUSTER(*v*) en el Paso 3 para determinar la celda correspondiente y el nivel del vértice ***v***. Al ser una estructura de múltiples niveles, se mitiga el efecto de valores perdidos. En este caso ADDVERTEX(*v*) mantiene la propiedad de que cada punto en el espacio está representado por un nodo en la estructura, la única incógnita es en qué nivel de detalle se encuentra.

Para acelerar la velocidad de la búsqueda, se utiliza una búsqueda binaria en los diferentes niveles. Como la profundidad del árbol es $\boldsymbol{O(\log(N_c))}$, donde $\boldsymbol{N_c}$ es el número total de celdas, una búsqueda binaria probabilística sobre la profundidad, reduciría el orden a $\boldsymbol{O(\log(\log(N_c)))}$.

Detectar colisiones de hash: Como la función WRITE utiliza tablas *hash*, se puede encontrar el caso en que dos vértices tengan como resultado el mismo índice en la tabla. Una solución es que WRITE, almacene las llaves y el valor, permitiendo a READ determinar si se consiguió o no una colisión. En la aplicación no es posible debido a las limitaciones en la composición aditiva utilizada para acumular el error cuadrático. Por lo tanto, se utiliza el modo de composición máximo para solo el componente alfa y solo se escribe ***k*** en un *buffer* y ***–k*** en el otro. La operación READ verifica que los valores sean iguales a ***k*** y ***-k*** respectivamente.

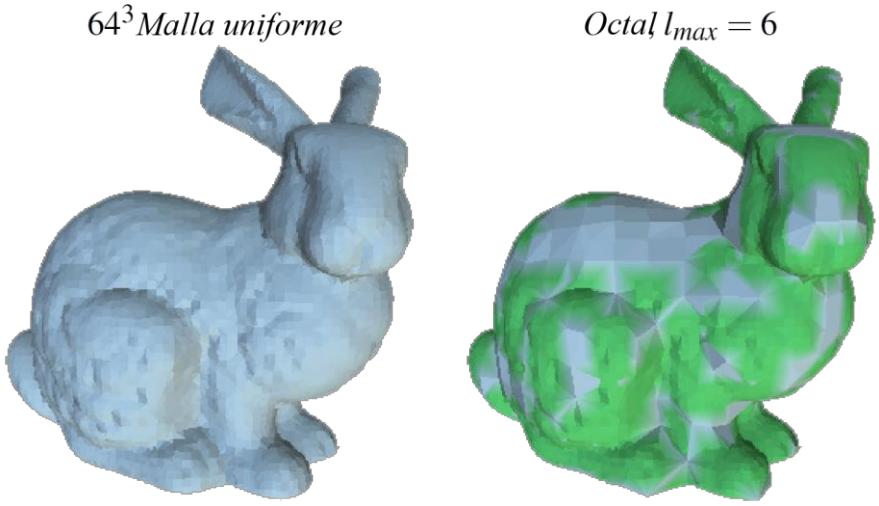


Figura 23: *Ejemplo de la simplificación con rejilla uniforme y árbol octal, a la izquierda una rejilla uniforme de tamaño* 64^3*, con 13K triángulos, a la derecha la misma malla simplificada utilizando un árbol octal con 6 niveles de detalle (equivalente en espacio a una rejilla de* 64^3*), en este caso la cantidad de triángulos es de 4K, y se mantiene un nivel de detalle similar en zonas relevantes como el contorno de las patas, piernas, orejas y ojos.*

Capítulo 3 Solución Propuesta

En este capítulo se describe el esquema general del diseño utilizado para desarrollar la solución.

3.1 Descripción general

Primero se presenta una descripción básica de la estructura del sistema de simplificación en tiempo real en la GPU.

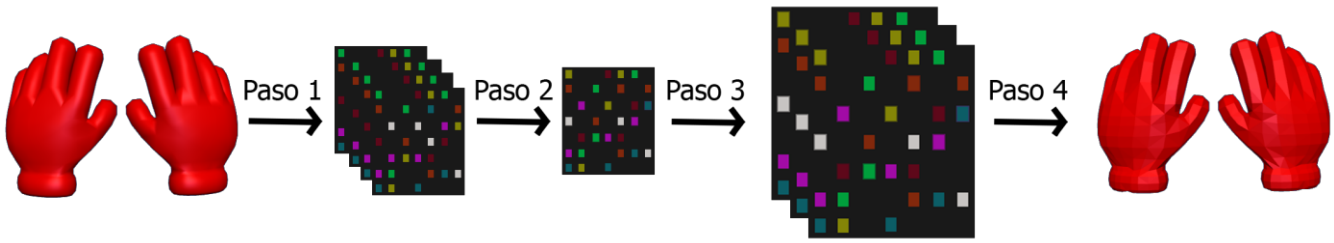


Figura 26: *Ejemplo del algoritmo de simplificación en tiempo real utilizando la GPU. Como se aprecia en la imagen, existen muchas similitudes con el algoritmo desarrollado por DeCoro, con la diferencia de un paso adicional antes de generar la malla simplificada.*

En la **Figura 26** se describen las diferentes estructuras generadas en el proceso de simplificación. El proceso se asemeja al implementado en el trabajo de DeCoro y Tatarchuk [30], las diferencias se detallan en el próximo capítulo

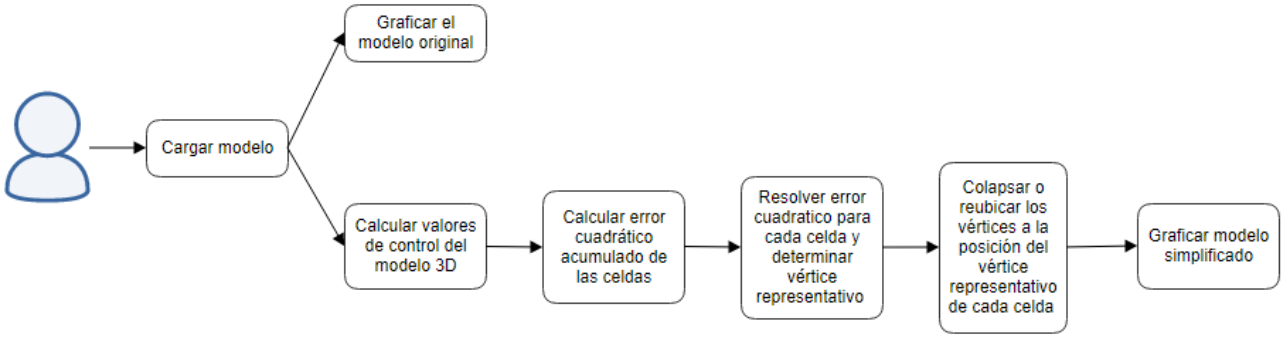


Figura 27: *Proceso de simplificación de mallas en tiempo real.*

Como se aprecia en la **Figura 27**, se diseñó una única clase encargada de procesar el modelo geométrico. Este módulo requiere de que se suministre un objeto 3D en malla y ofrece dos métodos para ejecutar sobre la malla suministrada:

Graficar Original: Este método grafica la malla original y permite realizar operaciones sobre la misma. Entre las cuales están: Dibujar un escenario, delinear la malla del modelo, mostrar información técnica de la malla, como el número de vértices y caras, así como un log de los cálculos realizados sobre la malla.

Simplificar: Este método que se describe en sus distintos pasos en la **Figura 27**, es el método que realiza los cálculos necesarios para simplificar la malla. El método Simplificar posee las mismas funciones que ofrece el

método de Graficar Original. Con la diferencia de que en este método se implementan sobre la malla resultante del proceso de simplificación.

La entrada de la aplicación es el modelo en malla que se busca simplificar. La malla es procesada utilizando la GPU para aprovechar sus capacidades de procesamiento paralelo. La salida de la aplicación será la malla original simplificada. El algoritmo de simplificación consiste en 4 pasos que se detallan a continuación:

En el primer paso, dada la malla, el *bounding box*, y una cantidad definida por el usuario de dimensiones, se envía la información al *pipeline* gráfico. Se utiliza la imagen que retorna el *pipeline* gráfico como matriz de almacenamiento, donde cada píxel representa una celda. En el *Vertex Shader* cada vértice debe calcular su posición en esta matriz. En la implementación de DeCoro y Tatarchuk [30], se utiliza el *Geometry Shader* para realizar el cálculo del error cuadrático, pero en este trabajo este cálculo se realiza en el *Vertex Shader*, por limitaciones técnicas(la *API* WebGL no posee un *Geometry Shader*). El *fragment shader* acumula el error cuadrático utilizando la función de *blending*.

En el segundo paso, se envía al *pipeline* gráfico el resultado del primer paso junto a un plano con el mismo tamaño de la matriz de almacenamiento. En el *Vertex Shader* solo se retornan los valores sin alterar. En el *Fragment Shader*, se extrae la información de la matriz de almacenamiento, se determina el vértice representativo utilizando inversión de matrices y se retorna una nueva matriz conteniendo los vértices representativos de cada celda.

En el tercer paso, se envía la malla original y la matriz de almacenamiento generada en el segundo paso al *pipeline* gráfico. En el *Vertex Shader* los vértices determinan su nueva ubicación utilizando la matriz de almacenamiento. En nuestra implementación cada vértice se reubica a su nueva posición en el *Vertex Shader*. En la implementación de DeCoro y Tatarchuk [30], los vértices son reposicionados en el *Geometry Shader*. En el *Fragment Shader* se retorna para ser almacenado en el conjunto de matrices la estructura de la malla simplificada.

En el cuarto paso, se extrae la información de las matrices de almacenamiento generadas en el tercer paso y se genera la malla simplificada, este paso es solo necesario si se desea tener la malla resultante en la memoria RAM. Esto permite luego generar un archivo distribuible y utilizar el modelo resultante en otra aplicación.

3.2 Implementación

En este capítulo se detalla la implementación del algoritmo de simplificación en tiempo real utilizando la GPU en WebGL. Para el desarrollo del algoritmo de simplificación en tiempo real utilizando la GPU en WebGL se hizo uso de la metodología *ASD (Adaptive Software Development)*, que gracias a sus ciclos rápidos de aprendizaje y mejora, lo convierte la metodología ideal para la realización de un proyecto con tantos componentes ofuscados por los múltiples *API* de desarrollo utilizados. En la implementación del algoritmo de simplificación en tiempo real, se hizo uso de los siguientes lenguajes:

HTML (*Hypertext Markup Language*), que es el lenguaje estándar para la creación de páginas y aplicaciones web, se utilizó para la creación de las etiquetas necesarias para almacenar la información requerida y dar estructura a todos los distintos elementos que conformarían la aplicación.

CSS (*Cascading Style Sheets*), es el lenguaje desarrollado para describir la apariencia de un documento escrito en HTML; este lenguaje es utilizado para darle forma y agregarles estilos a todas las etiquetas creadas con HTML, para el desarrollo del algoritmo de simplificación en tiempo real se hizo uso del *framework* de diseño web Bootstrap, que se encarga de abstraer toda la complejidad del lenguaje css y acelera el desarrollo del diseño de la página web.

Javascript, que junto con CSS y HTML, completan el conjunto de lenguajes con los que se genera todo el contenido dentro del *World Wide Web*, es el lenguaje encargado de brindar dinamismo a la página web y es donde está implementado el *framework* WebGL, con el que Javascript es capaz de tener acceso a los comandos internos

de la tarjeta gráfica. Para el desarrollo del algoritmo de simplificación se hizo uso de la biblioteca Threejs, una de la biblioteca más importante que encapsula toda la complejidad del lenguaje WebGL, facilitando el desarrollo de los *shaders* necesarios para realizar la simplificación de mallas.

Al momento de implementar el algoritmo de simplificación, debido a la necesidad de que existieran dos entornos independientes que se encargaran de manejar tanto el modelo original como el simplificado, se englobó toda la lógica con la que se procesa la malla en una clase (ver **Figura 28**) que puede ser instanciada múltiples veces. La estructura general de ésta clase se define a continuación.

Para lograr la simplificación de mallas en tiempo real, el algoritmo se dividió en los siguientes pasos.

Primero se carga el modelo en cualquiera de los formatos soportados y se traduce a las estructuras de datos que proporciona la biblioteca Threejs. El primer paso es calcular el *Bounding Box*, resultando en dos vértices representativos con el valor máximo y mínimo de cada coordenada. Utilizando los valores del *Bounding Box* aplicamos operaciones de escalado y traslación para transformar los vértices pertenecientes al objeto al espacio de valores entre los números uno (1) y dos (2). Las posiciones se trasladan con el fin de evitar errores de precisión punto flotante al momento de realizar los cálculos de error cuadrático. Se calcula la metadata que necesitaremos para poder determinar cómo se agruparán los vértices. La metadata está compuesta por los siguientes elementos:

- A cada vértice se le asigna la posición de los otros vértices de su cara, para simular el funcionamiento del *Geometry Shader* en el *Vertex Shader*.

- Se asigna como atributo a todos los vértices los valores máximo y mínimo, el número de dimensiones definidas por el usuario para dividir la rejilla, y el tamaño de la textura que se utilizara para almacenar toda la información de la rejilla.

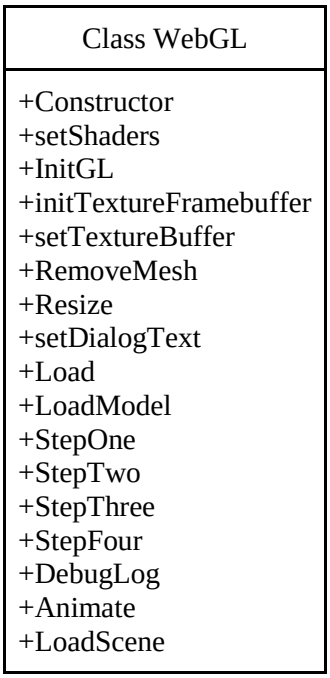


Figura 28*: Diagrama de la clase principal en el algoritmo de simplificación de mallas.*

3.3 Implementación del algoritmo de simplificación

En esta sección detalla la implementación de los pasos descritos en la sección 3.1.

En el primer paso se debe calcular el error cuadrático, primero el *Vertex Shader* determina si dos o más vértices de la cara pertenecen a una misma celda, en caso de ser así la cara debe colapsada, por lo que se descarta. Debido a que no se posee un *Geometry Shader* con el que eliminar los elementos a descartar, le asignamos al vértice una posición que se salga del rango de la cámara. En el caso de que todos los vértices de la cara pertenezcan a distintas celdas se genera la matriz de error cuadrático de la siguiente forma:

$$v = (x_1, x_2, x_3)$$

$$Q = \begin{pmatrix} A & -b \\ -b^T & c \end{pmatrix} = nn^T$$

El vértice ***v*** representa cada vértice, y el valor **n** está compuesto por la siguiente fórmula.

$$n = \begin{pmatrix} (v_1 \times v_2) + (v_2 \times v_3) + (v_3 \times v_1) \\ -(v_2 \cdot v_3) \end{pmatrix}$$

Como se muestra a continuación, con el fin de recuperar el error cuadrático acumulado en el *Vertex Shader*, se ajusta la posición del vértice a la del píxel de la textura donde se almacenará la información de la celda.

```
vec3 pos;

//Primero es necesario trasladar los valores de posición del vértice
//al rango de valores entre el [0 - Dim), con el fin de utilizarlos como
//índice de la posición en el Grid, donde Dim es el número de
//dimensiones definido por el usuario, luego utilizamos la función
//floor() que redondea hacía abajo los números. Quedando 3.4 => 3

vec3 CellIndex = floor((VertPos - min)*Dim/(max - min));

//Si el índice es igual a Dim, se sale del rango del grid por lo que se le
//resta 1 para movilizarlo a la posición del grid más cercana, quedando
//el índice en 3D de la posición dentro del Grid

if(CellIndex.x == Dim)CellIndex.x-=1.0;
if(CellIndex.y == Dim)CellIndex.y-=1.0;
if(CellIndex.z == Dim)CellIndex.z-=1.0;

//El índice 3D se transforma en un índice de 1D.

float temp = CellIndex.x + CellIndex.y * Dim + CellIndex.z*Dim*Dim ;

//El índice en 1D se transforma en 2D en el rango [0 - RTDim] para poder
//almacenar la información en una textura plana 2D, fijando la coordenada
//Z en 1.0, donde RTDim es el tamaño de la textura donde se almacenara el
//Grid

pos.y = floor(temp/RTDim);
pos.x = temp - (pos.y * RTDim);
pos.z = 1.0;

//Se suma 1 a las coordenadas X, Y para evitar los valores iguales a 0.
//ya que existe perdida de información si los pixeles se encuentran cerca
//del borde de la pantalla.
```

```
pos.x +=1.0;
pos.y +=1.0;

//Se traslada los índices al rango (-1 - 1), que es el rango de valores que utiliza
//el monitor. Al momento de hacer el cálculo se incrementa el valor del parámetro
//RTDim, con el fin de evitar los bordes, ya que causaba inconsistencia entre distintos
//dispositivos.

float trtdim = RTDim + 1.0;
pos.x = ((pos.x / trtdim)*2.0) - 1.0;
pos.y = ((pos.y / trtdim)*2.0) - 1.0;
```

El *Vertex Shader* pasa al *Fragment Shader* la información calculada a través de un arreglo de trece (13) posiciones.

La información que transmite el *Vertex Shader* al *Fragment Shader* se lista a continuación:

- Nueve (9) valores pertenecen a la diagonal superior de la matriz simétrica.
- Tres (3) valores para la posición (X,Y,Z) del vértice.
- Un (1) valor para acumular la cantidad de vértices que pertenecen a la celda.

El *Fragment Shader* acumula la información suministrada por el *Vertex Shader* sin modificarla. Para acumular la información se hace uso de la función de *blending* (gl.ONE, gl.ONE) que se suma todos los valores de los píxeles que coinciden coordenadas. Las funciones de *blending* toman en cuenta el valor alpha de los vértices, con el fin de proporcionar transparencia a las imágenes. Al utilizar la función de blending (gl.ONE, gl.ONE) se multiplica por uno los valores del vértice, ignorando el valor alpha del mismo.

El resultado del *pipeline* gráfico se almacena en una textura de tamaño igual a $\sqrt{Dimensiones^3}$. Donde las dimensiones es el número de divisiones que tendrá el *Bounding Box* determinado por el usuario. Esto con el fin de poder almacenar los 13 valores necesarios por celda, se hizo uso de un conjunto de 4 texturas, que utilizando sus 3 valores RGB y el valor Alpha, suman 16 campos donde se puede almacenar la información y queda un margen de mejora para futuros cálculos adicionales.

En el segundo *Shader*, se genera un plano con el mismo tamaño que las texturas de almacenamiento. El plano se envía junto con la metadata y las texturas generadas en el primer *shader* al *pipeline* gráfico para calcular el vértice representativo. En el *Vertex Shader* se retorna la información sin alterar y finaliza su ejecución. En el *Fragment Shader* cada píxel accede a su píxel correspondiente de las texturas generadas en el primer paso. Cada píxel construye la matriz de error cuadrático y calcula el píxel promedio utilizando el valor promedio y el conteo de vértices. Para determinar el vértice representativo se invierte la matriz de error cuadrático y se resuelve la siguiente ecuación:

$$A_x = b$$
$$x = A^{-1}b$$

La matriz deja de ser invertible cuando todos los planos sean linealmente dependientes. Al sumar un pequeño término a la diagonal se incrementa el determinante causando que la matriz sea invertible. Debido a estas situaciones y a la pérdida de precisión punto flotante al insertar y extraer datos en el *pipeline* gráfico en WebGL, el error cuadrático tiende a ser impreciso.

Para mejorar la calidad del resultado al realizar la simplificación se hace uso de dos vértices representativos, el primero y más importante es el vértice resultante al resolver la ecuación de error cuadrático. El segundo vértice se genera a partir de promediar la posición acumulada de los vértices pertenecientes a la celda. El vértice que se utilizará como vértice representativo será el promedio entre ambos vértices.

El *Fragment Shader* almacena en una textura del tamaño del plano generado, se retorna en cada píxel la posiciones (X, Y, Z) del vértice representativo en los valores RGB y se deja el valor alfa fijó en 1.0.

En el paso tres, se envía la malla original junto con la textura que contiene los vértices representativos para reubicar los vértices pertenecientes a la malla, con el fin de poder extraer la nueva malla del *pipeline* gráfico se genera una nueva textura de tamaño $\sqrt{Numeros\ de\ vértices\ en\ la\ malla}$.

En el *Vertex Shader* el vértice representante de cada cara verifica si la cara debe ser colapsada y detiene su ejecución en caso de ser así. Si es necesario reubicar los vértices, el vértice representante extrae la nueva posición de los vértices de su cara de la textura generada en pasos anteriores, y retorna las posiciones nuevas para los vértices de la cara.

En el *Fragment Shader* se retorna la información sin alterar. Con solo conocer las nuevas posiciones de los vértices no se puede determinar la estructura topológica de la malla. Es necesario poder preservar la interconectividad de las caras si se desea reconstruir la nueva malla preservando la estructura de la malla original. Para conservar la interconectividad se utilizarán 3 texturas, donde cada píxel representará una cara. Con tres (3) texturas poseemos doce (12) términos donde se puede almacenar la información de los vértices de la malla. Al conocer que el píxel correspondiente en una misma posición en las distintas texturas representa la misma cara, se pueden relacionar los vértices, y por lo tanto, determinar la topología de la malla.

En el paso 4, se debe construir la malla simplificada utilizando la información generada en los anteriores pasos. Primero se debe extraer la información de las texturas, y almacenarse en un set de arreglos en formato punto flotante. Luego se itera por los arreglos extrayendo la información posicional de los vértices que conforman la malla simplificada y almacenan en orden en una nueva estructura interna de Threejs llamada BufferGeometry, que luego es utilizado para formar la nueva malla.

3.4 Implementación de la interfaz

Para la interfaz se tuvieron que tomar en cuenta dos elementos cruciales. Como se aprecia en la **Figura 29**, se agregó el botón ***Cargar*** con el que se puede suministrar a la aplicación un archivo *obj* (la biblioteca threejs trae soporte por defecto también al formato de archivos *js*) que internamente se traduce a las estructuras internas de la biblioteca Threejs. La segunda es brindar la capacidad de comparar el modelo original con su versión simplificada dividiendo la pantalla de visualización en dos, donde a cada mitad corresponde un entorno gráfico. En el entorno izquierdo se graficará el modelo original, y en el entorno derecho se graficará el modelo resultante al modelo de simplificación.

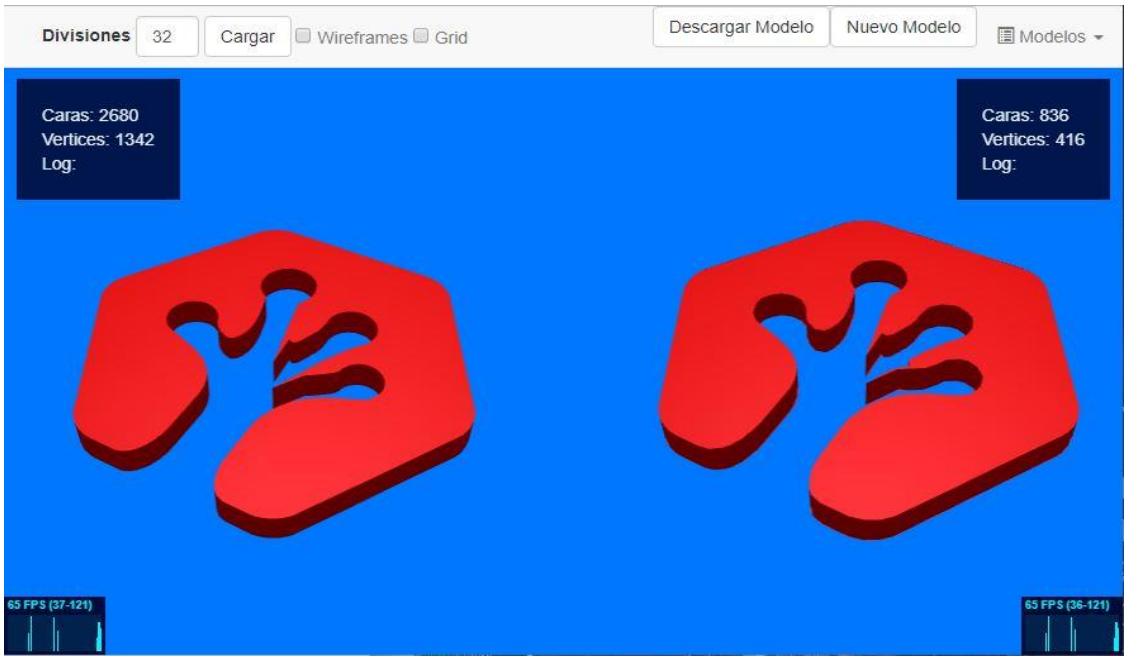


Figura 29*: Prueba del modelo TreehouseLogo (izquierda) y la malla simplificada (derecha) utilizando el algoritmo de simplificación utilizando 32 dimensiones.*

Existen múltiples elementos descriptivos en la aplicación, que se agregaron para facilitar información al usuario con los que se pueden estudiar el rendimiento del algoritmo y medir su funcionamiento. Una funcionalidad importante para apreciar cómo se afectó la cantidad y estructura de las caras pertenecientes al modelo, es el delineado de las caras. Como se aprecia en la **Figura 30** esta opción está disponible bajo el nombre de *Wireframe*, que al activarse resalta con líneas blancas los bordes de todas las caras, mostrando las interconexiones entre todos los vértices del modelo.

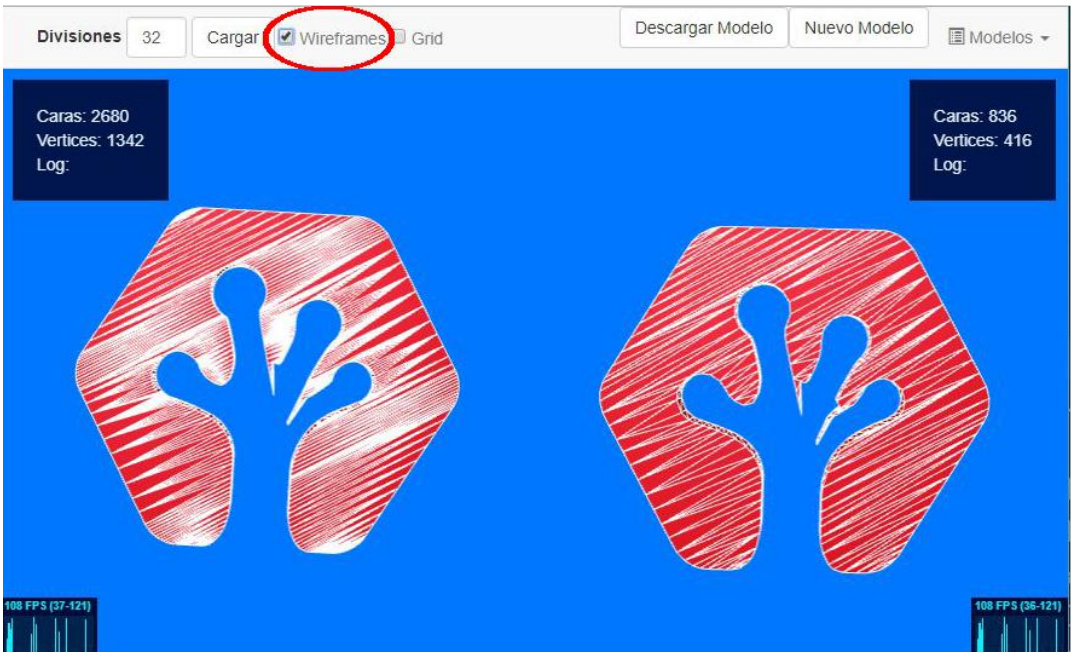


Figura 30*: Ejemplo de la funcionalidad Wireframe. Se puede apreciar como el delineado de los bordes de las caras permite apreciar la diferencia en la estructura y cantidad de las caras.*

Otra herramienta que se implementó con el fin de poder analizar los resultados del algoritmo de simplificación es el delineado de las celdas. Como se muestra en la **Figura 31** esta funcionalidad se llamó *Grid*, con ésta se puede tener

un mejor entendimiento de cómo trabaja el algoritmo internamente y el por qué simplifica las mallas de la forma en que lo hace al permitir determinar la forma en que se están agrupando los vértices, y percibir que vértice está remplazando los vértices que fueron colapsados.

La función *Grid*, se implementó principalmente para evaluar el proceso de simplificación, ya que se desconoce sus posibles beneficios para el uso práctico de la aplicación.

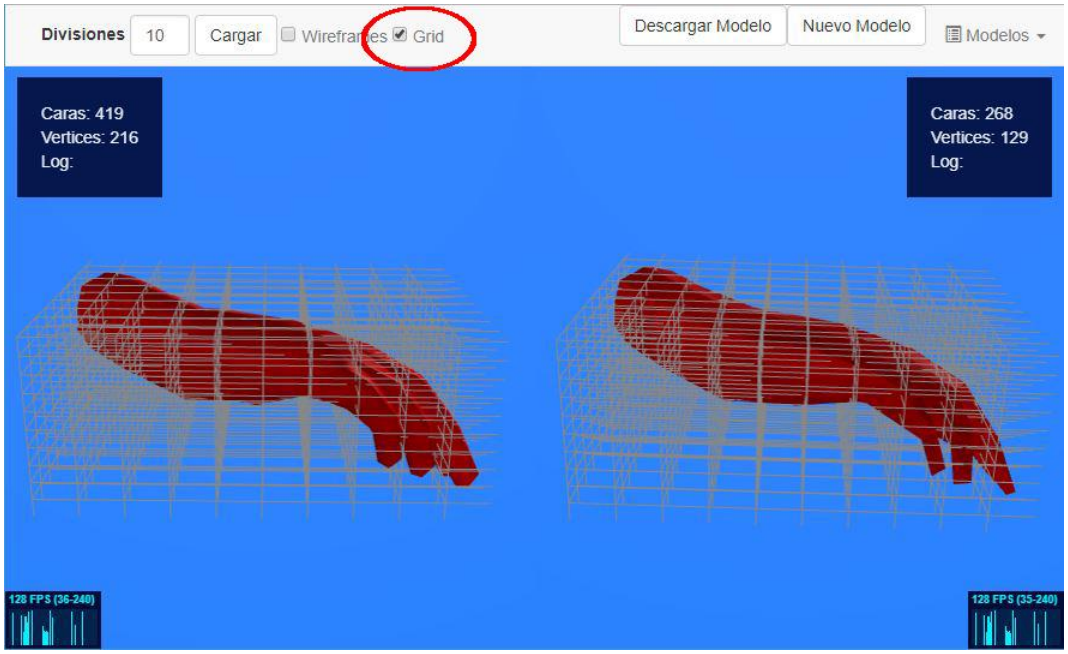


Figura 31: *Ejemplo del delineado de las celdas, que demuestra la forma en que el algoritmo agrupa los vértices, para realizar los cálculos que hace y la forma en que colapsa las caras.*

Un elemento necesario para poder medir la efectividad del algoritmo de simplificación, son las pantallas informativas, que consisten en un segmento de la página dedicada a mostrar información interna de las mallas y del proceso de simplificación. Las pantallas informativas consisten en campos de texto que se actualizan constantemente. Existen pantallas informativas para ambos entornos gráficos. Donde se muestra el número de caras y vértices que componen a la malla, adicionalmente muestra los últimos mensajes emitidos por la aplicación. Un ejemplo de estas pantallas se puede apreciar en la **Figura 32**, el elemento número 1.

Para poder hacer uso del algoritmo de simplificación es necesario definir la cantidad de subdivisiones que tendrá el *Bounding Box*. Esto se logra con un campo editable de texto (el elemento número 2 de la **Figura 32**) donde se puede definir la cantidad de divisiones que tendrá el *Bounding Box*.

El elemento número 3 de la **Figura 32** es el botón Cargar, su función es recalcular la malla simplificada, con el fin de actualizar la cantidad de divisiones del *Bounding Box* que se utiliza como parámetro durante el proceso de simplificación.

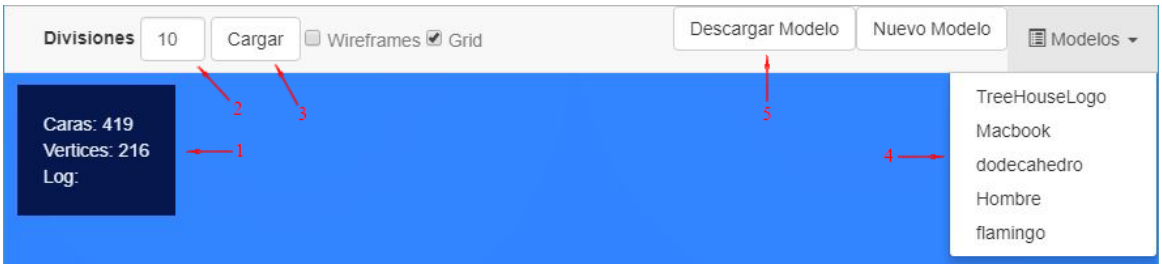


Figura 32*: Modelos predefinidos para realizar pruebas del algoritmo de simplificación*

Para facilitar realizar pruebas con la aplicación se agregó una lista de modelos predefinidos ya cargados en el servidor (Elemento número 4 de la **Figura 32**).

El botón “Descargar Modelo” qué se observa en la **Figura 32** como el elemento número 5, se implementó para permitir al usuario extraer el modelo simplificado en el formato obj. Esta funcionalidad hace uso del método OBJExporter de la biblioteca Threejs, que genera la información en texto plano que contendrá el archivo en formato obj. Para realizar la descarga se usó el siguiente código en JavaScript:

```
var element = document.createElement('a');
element.setAttribute(
'href',
'data:text/plain;charset=utf-8,'
+ encodeURIComponent(WebGL2.get_new_mesh())
);

//Se crea un elemento del tipo “a”, y se le agrega como atributo la información
//generada por la función OBJExporter de threejs en texto plano.
//Al definirle el atributo descarga, y tipo texto, el navegador por defecto
//iniciara el proceso de descarga del archivo result.ob.
element.setAttribute('download', "result.obj");

element.style.display = 'none';
document.body.appendChild(element);

element.click();

//Se simula un evento tipo Click, con el fin de iniciar el proceso de descarga

document.body.removeChild(element);

//Se elimina el elemento creado para no dejar información ya no necesaria
//en memoria.
```

Con los detalles de implementación descritos, se presenta en el siguiente capítulo los resultados de las pruebas realizadas.

4 Pruebas y resultados

En este capítulo se describen un conjunto de pruebas básicas y una comparación entre los diferentes resultados y las limitantes del desarrollo realizado.

4.1 Ambiente de pruebas

El dispositivo donde se realizaron las pruebas se compone de:

- Procesador : Intel® Core™ i7-3770 @ 3.40GHz
- Memoria : 12.00 Gb DDR3 1333 Mhz
- GPU : Asus GeForce GT 610 Graphic Card 1 GB DDR3

Para el desarrollo de la aplicación se hizo uso de las bibliotecas:

- Threejs - Biblioteca que se encarga de encapsular lógica de WebGL [31]
- OrbitControls - Biblioteca para el manejo de la cámara [32]
- Less - Biblioteca que encapsula la lógica del lenguaje CSS [33]
- Stats - Biblioteca que mide los *frames* por segundo (fps) [34]
- Loaders - En este trabajo se utilizan múltiples bibliotecas, como OBJLoader.js, MTLLoader.js y las funciones internas de Threejs, que se encargan de traducir de los formatos de objeto en malla 3D a estructuras manejables por la aplicación.
- Bootstrap – Es un *framework* para el modelado de la interfaz de la página [35]

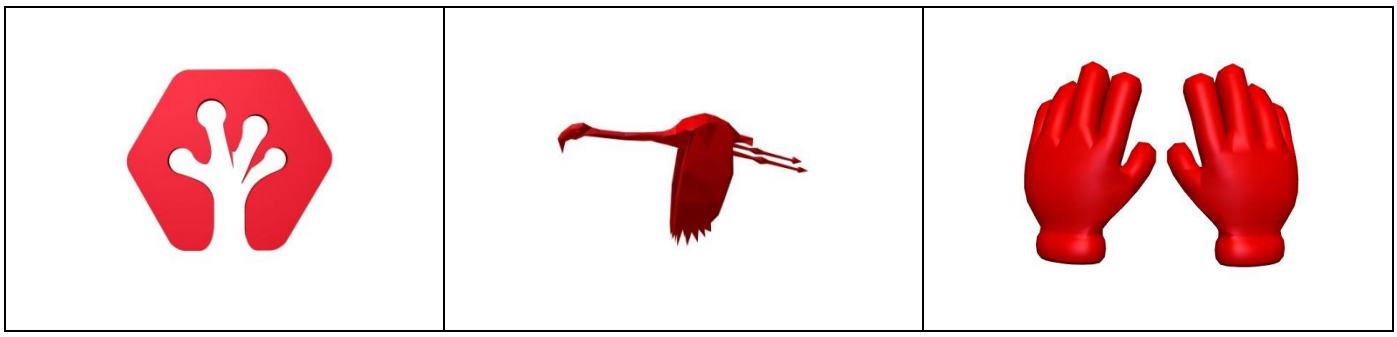
Las bibliotecas se almacenan en el mismo *hosting* desde el cual se sirve la aplicación Web, a excepción de la biblioteca de Bootstrap que se importa directamente del *hosting* de sus creadores.

Para facilitar el acceso a la aplicación se hizo uso de la plataforma *GitHub*[36] para servirla públicamente utilizando su funcionalidad llamada *GitHub pages* [37] con la que se puede publicar un proyecto web desde sus repositorios. Actualmente (Primer trimestre del año 2018) se encuentra publicado el algoritmo de simplificación a través del *url (Uniform Resource Locator) gomesandres.github.io.*

4.2 Modelos utilizados

Para la realización de las pruebas se utilizaron los siguientes modelos:

Treehouse_logo.js	Flamingo.js	GloveLow_poly.obj
Modelo que representa el logo de la página teamtreehouse.com [38]	Modelo que refleja un flamenco como su nombre indica.	Modelo que simula un par de guantes
2680 Vértices	542 Vértices	2144 Vértices
8040 Caras	1626 Caras	6432 Caras



4.3 Pruebas cuantitativas

Se realizó un conjunto de pruebas a diferentes modelos con el fin de medir el comportamiento del algoritmo de simplificación.

Modelo	Dimensiones	Vértices Originales	Caras Originales	Vértices Simplificado	Caras Simplificado
treehouse_logo.js	10	1342	2680	138	284
	15	1342	2680	214	436
	32	1342	2680	416	836
Flamingo.js	10	273	542	108	233
	15	273	542	173	347
	32	273	542	237	469
GloveLow_poly.obj	10	1076	2144	408	940
	15	1076	2144	726	1534
	32	1076	2144	1040	2073

Tabla 1: *Conjunto de pruebas para medir el desempeño según la cantidad de caras y vértices que fueron eliminadas durante el proceso de simplificación.*

Como se puede apreciar en al **Tabla 1**, el porcentaje de reducción de vértices y caras, varía entre el 3,31% de elementos eliminados hasta un total de 89,71%. Ésta variación entre la cantidad de elementos eliminados se debe a la manera en que se distribuyen los vértices de las mallas. Modelos en malla que distribuyen con mayor uniformidad los vértices en su perímetro tienden a tener una menor cantidad de caras colapsadas durante el proceso de simplificación.

Modelo	Dimensiones	Paso 1	Paso 2	Paso 3	Paso 4
treehouse_logo.js	10	47,47ms	35,09ms	29,01ms	53,62ms
	15	47,30ms	27,48ms	35,08ms	70,18ms
	32	54,77ms	38,69ms	29,47ms	50,93ms
Flamingo.js	10	29,52ms	24,96ms	27,41ms	38,21ms
	15	31,06ms	28,44ms	26,24ms	35,11ms
	32	36,50ms	32,55ms	29,60ms	48,57ms
GloveLow_poly.obj	10	36,40ms	26,32ms	28,33ms	54,77ms
	15	34,34ms	25,18ms	24,14ms	31,71ms
	32	39,37ms	30,88ms	25,83ms	37,58ms

Tabla 2: *Conjunto de pruebas para medir los tiempos de ejecución de las distintas funciones que conformar el algoritmo de simplificación.*

En la **Tabla 2**, se muestra un conjunto de pruebas de tiempo de ejecución. Estas pruebas se hicieron sobre los principales pasos del proceso de simplificación, que se detallaron en anteriores capítulos. Los tiempos que se muestran en la **Tabla 2**, están expresados en milisegundos. Debido a que el entorno WebGL aún es experimental, durante las pruebas los tiempos presentaban considerables variaciones, por lo que se tuvo que realizar múltiples pruebas y promediar los tiempos.

4.4 Pruebas cualitativas

Se presentan un conjunto de pruebas visuales, realizadas sobre los modelos de prueba.

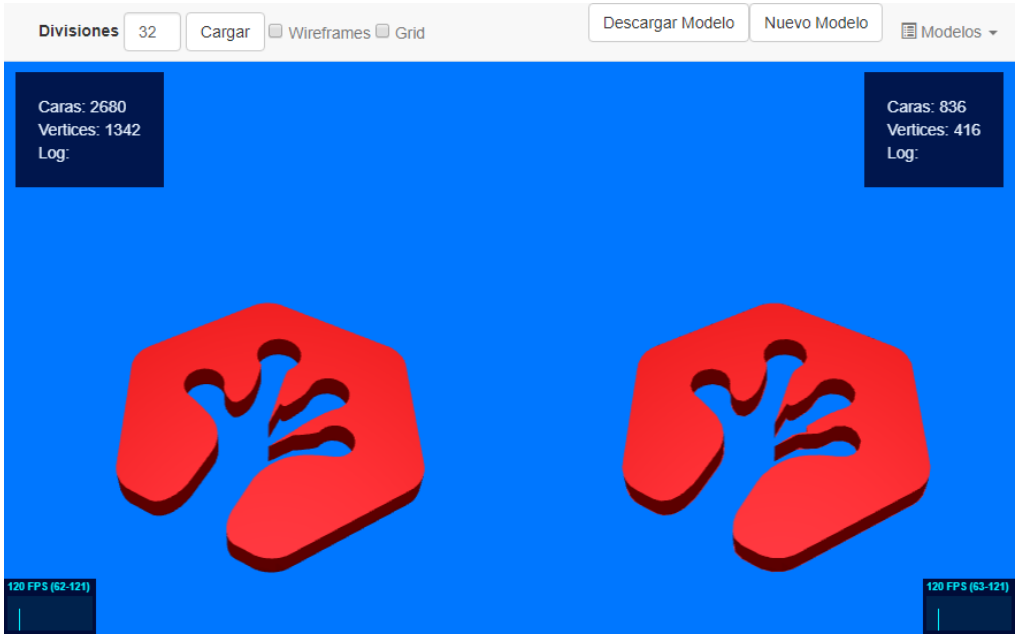


Figura 33*: Prueba cualitativa del modelo treehouse_logo.js, utilizando 32 dimensiones.*

Como se puede apreciar en la **Figura 33**. Cuando aplicamos el algoritmo de simplificación utilizando como parámetro 32 dimensiones al modelo *treehouse_logo.js*, se obtiene como resultado un modelo que conserva muchas de las características estéticas del modelo original.

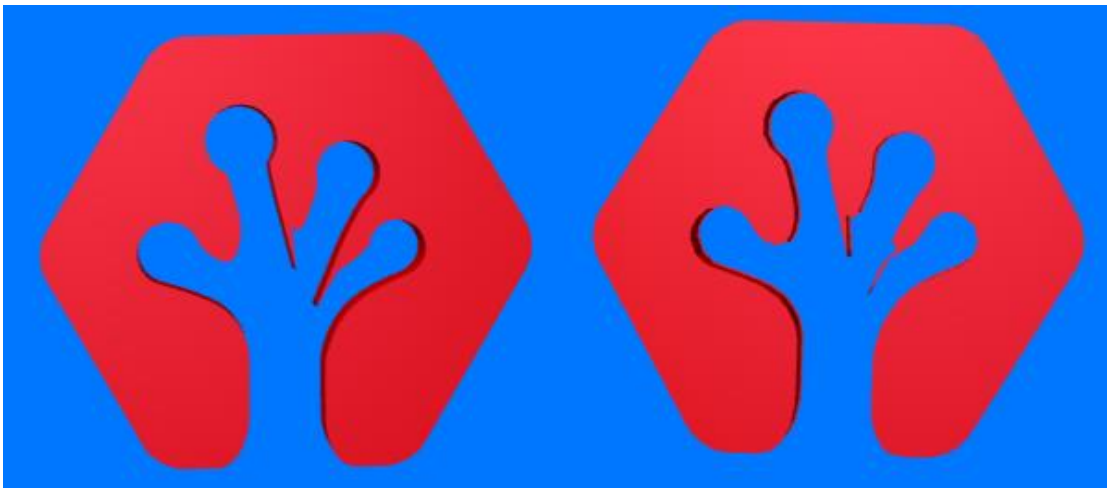


Figura 34: *Captura de la prueba con 32 dimensiones donde se nota con mayor claridad las diferencias generadas por el proceso de simplificación.*

En la **Figura 34** se pueden resaltar las diferencias entre los modelos. En el lado izquierdo del modelo se puede notar un pequeño segmento que fue colapsado por el algoritmo de simplificación.

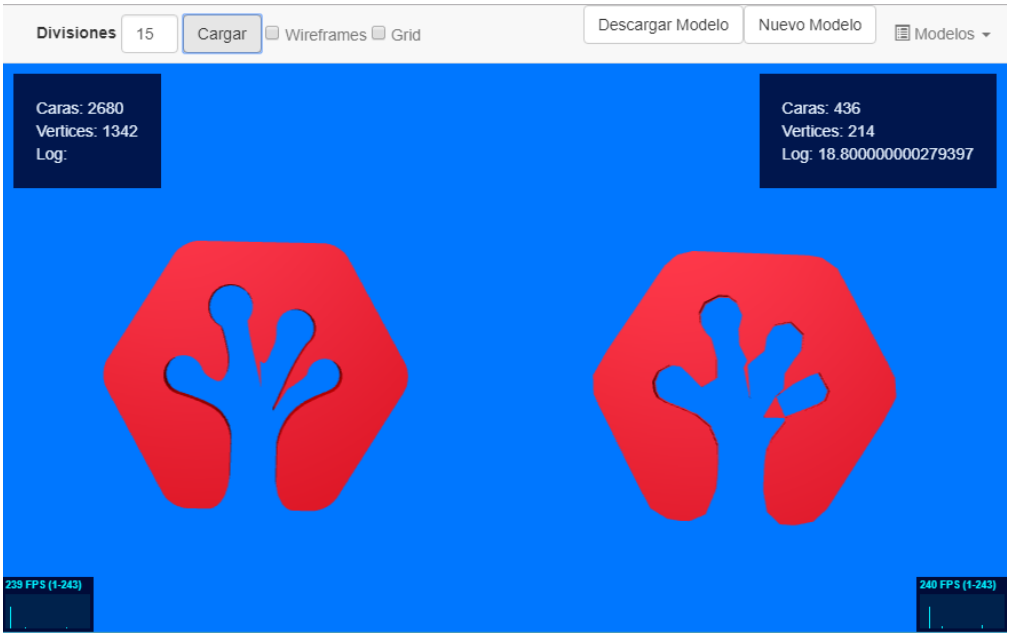


Figura 35: *Prueba cualitativa sobre el modelo treehouse_logo.js, utilizando como parámetro 15 dimensiones*

En la **Figura 35** se modificó el parámetro de dimensiones con el fin de incrementar el factor de simplificación. El número de caras y vértices resultantes se redujo a la mitad. Lo primero que se puede notar es como la apariencia estética de la malla se deformo en la parte central del modelo donde existe mayor cantidad de detalles.

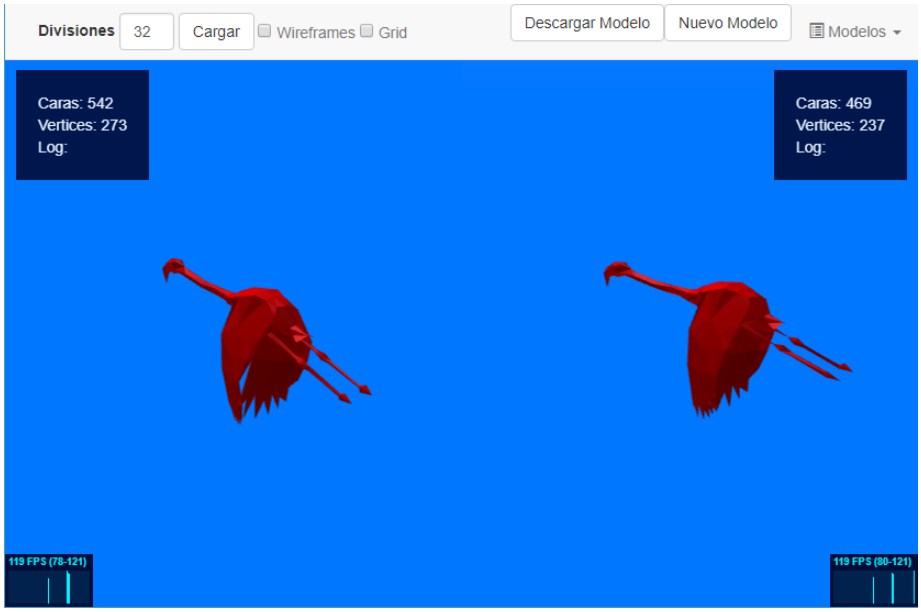


Figura 36: *Prueba cualitativa sobre el modelo Flamingo.js, utilizando como parámetro 32 dimensiones.*

Como se puede apreciar en la **Figura 36**, se utilizó el algoritmo de simplificación utilizando 32 dimensiones como parámetro, sobre el modelo *Flamingo.js*. La malla resultante conserva la mayoría de las características estéticas principales. En los segmentos del modelo con mayor cantidad de detalle se notan algunas variaciones, como lo son el extremo de las patas del flamenco.

Figura 37*: Prueba cualitativa sobre el modelo Flamingo.js utilizando como parámetro 15 dimensiones.*

En la **Figura 37**, se realizó la prueba sobre el modelo *Flamingo.js* utilizando como parámetro 15 dimensiones. En el modelo resultado se conservan la mayoría de los detalles estéticos. Los segmentos que se ven más afectados son las puntas de las patas y el sector de la cabeza. La pérdida de detalle en los sectores más afectados se debe a la cercanía de los vértices que la conforman. Causando que la mayoría de los vértices que la conforman pertenezcan a una misma celda, y sean colapsados por el algoritmo.

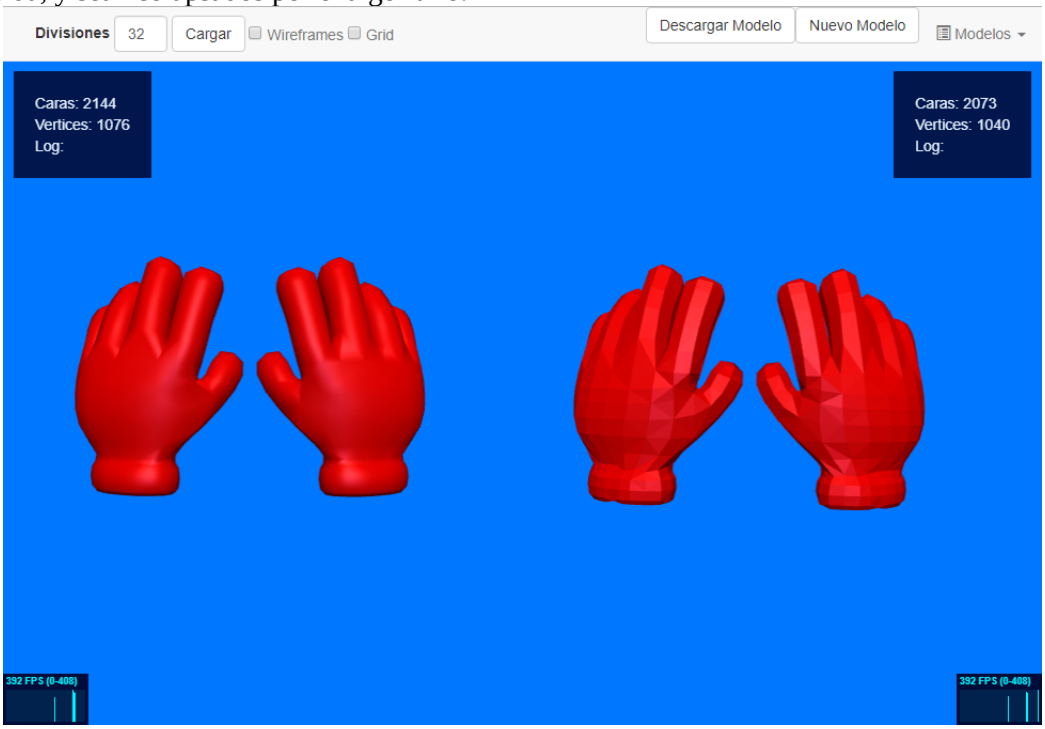


Figura 38*: Prueba cualitativa sobre el modelo GloveLow_poly.obj utilizando como parámetro 32 dimensiones*

En la **Figura 38** se puede notar el resultado al realizar la prueba cualitativa sobre el modelo *GloveLow_poly.obj*. Al utilizar 32 dimensiones se puede notar como la malla resultante conserva la mayoría de las características quedando como principal diferencia a resaltar la pérdida de suavidad.
En la **Figura 39** se muestra el resultado de la aplicación perceptualdiff [39] luego de realizar un análisis de diferencias basado en la percepción. En total existen 8736 pixeles con diferencias entre ambas imágenes.

Figura 39: *Imagen generada por la aplicación perceptualdiff, que resalta los pixeles con diferencias entre dos imágenes segun un algoritmo por percepción. Utilizando como base el modelo GloveLow_poly.obj. Con imágenes del modelo original y simplificado usando como parámetro 32 dimensiones.*

Como se aprecia en la **Figura 39** los segmentos más resaltantes son los bordes, donde se acentúa la pérdida de suavidad en el modelo. Las diferencias restantes se deben en su mayoría a los efectos de la iluminación sobre el modelo.

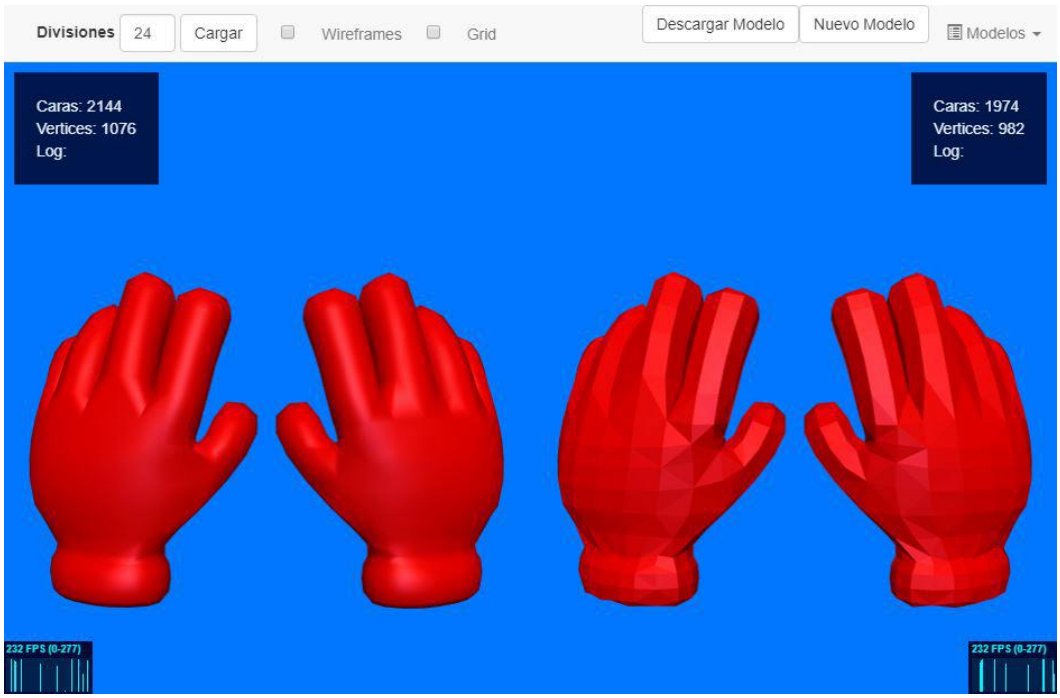


Figura 40: *Prueba cualitativa sobre el modelo GloveLow_poly.obj, utilizando como parámetro 24 dimensiones*

En la **Figura 40** se muestra la prueba cualitativa sobre el modelo *GloveLow_poly.obj*, utilizando como parámetro 24 dimensiones, se puede notar como se acentúan las diferencias en los bordes y en la suavidad, pero se conservan todas las cualidades estéticas principales. En la **Figura 41** se realizó la evaluación de percepción utilizando el algoritmo de *perceptualdiff*, donde lo principal a destacar es como se incrementa el área donde existen diferencias en los bordes del modelo, pero en contra de lo esperado, se reducen las diferencias en la zona interna de los modelos.

Figura 41: *Imagen generada por la aplicación perceptualdiff, sobre el modelo GloveLow_poly.obj, original y simplificado utilizando como parámetro 24 dimensiones.*

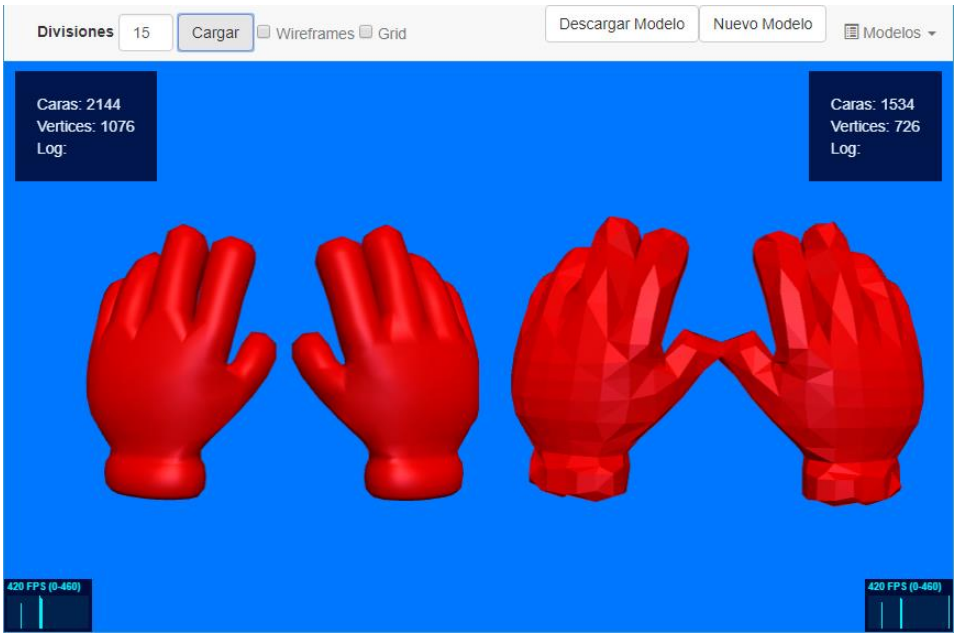


Figura 42*: Prueba cualitativa sobre el modelo GloveLow_poly.obj, utilizando como parámetro 15 dimensiones*

En la **Figura 42**, al utilizar 15 dimensiones la topología se ve alterada debido a que los vértices pertenecientes al borde de los dedos son colapsados, uniendo ambas manos. En la **Figura 43** se puede apreciar, como a pesar de utilizar una menor cantidad de dimensiones (14 dimensiones), la topología del modelo original se conserva mejor. Esto es debido a que al momento de agrupar los vértices con la rejilla, los vértices pertenecientes a los bordes de los dedos son agrupados en celdas distintas.

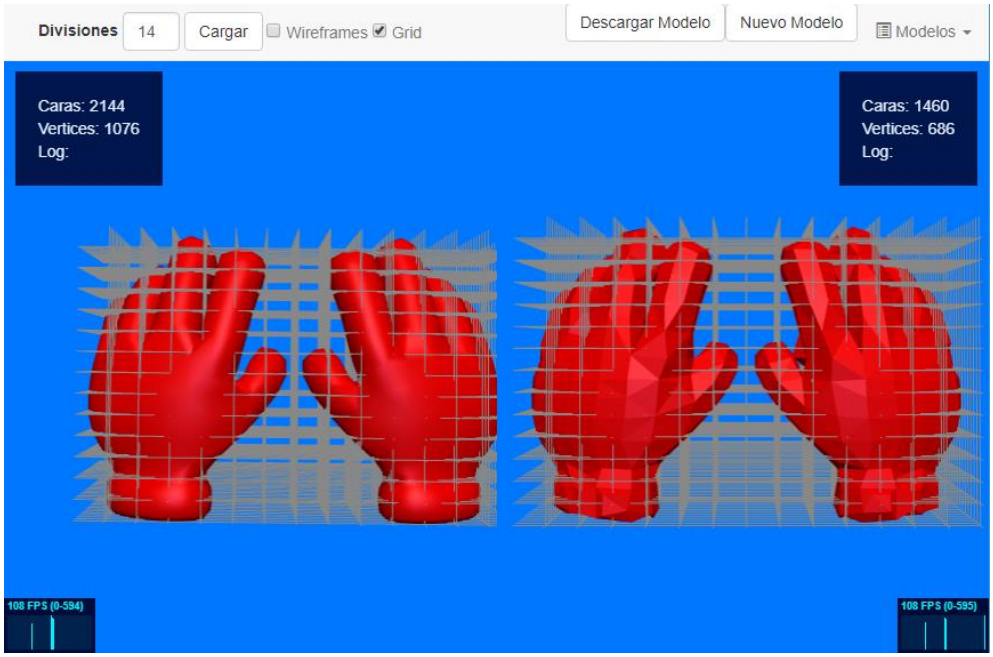


Figura 43*: Prueba cualitativa sobre el modelo GloveLow_poly.obj, utilizando como parámetro 14 dimensiones*

En la **Figura 44** se nota como se repite el incremento en el número de píxeles con diferencias cerca del borde del modelo. Una diferencia que se puede notar, es como se crearon zonas donde existen una gran cantidad de píxeles

con diferencias. Esto es debido a deformaciones en el modelo, con tan solo 14 dimensiones, cada celda colapsa una gran cantidad de vértices, llevando a que se unan segmentos de la malla que antes eran disconexos.

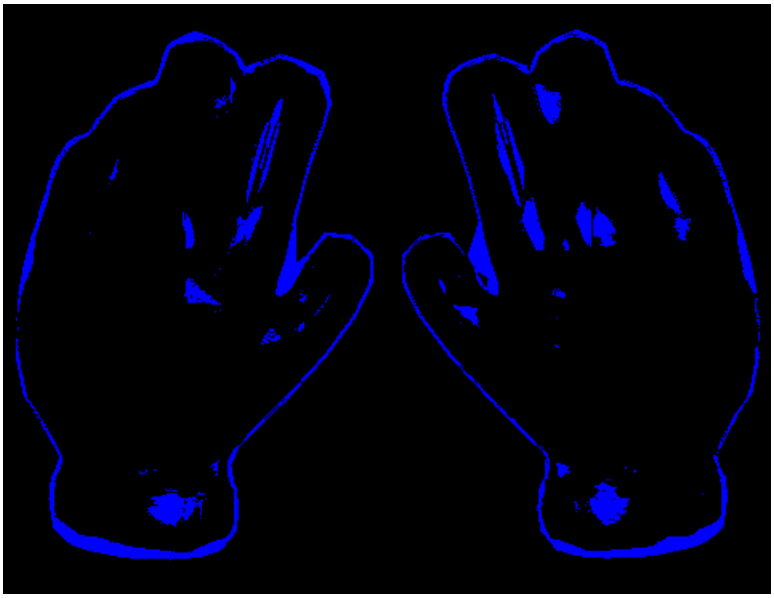


Figura 44*: Imagen generada por la aplicación perceptualdiff, sobre el modelo GloveLow_poly.obj, original y simplificado utilizando como parámetro 14 dimensiones.*

5 Conclusiones y recomendaciones

En este Trabajo Especial de Grado se realizó un algoritmo capaz de realizar simplificaciones de mallas en tiempo real en WebGL, haciendo uso de la GPU. Reduciendo la cantidad de elementos pertenecientes a una malla, reduciendo de esta manera el uso de recursos del ordenador.

Durante el desarrollo de este Trabajo Especial de Grado, existieron elementos que resaltaron ya sea por su complejidad o ser un hecho imprevisto. A continuación se detallan estos elementos más resaltantes.

La función que necesito de la mayor cantidad tiempo para ser implementada, fue la extracción de la información resultante del *pipeline* gráfico en cada uno de los pasos del proceso de simplificación detallados en el capítulo 3. Al no tener acceso a las instrucciones internas del procesador gráfico, fue necesario desarrollar una alternativa utilizando técnicas como *deferred rendering*, que consiste en almacenar el resultado del *pipeline* gráfico en texturas para su uso posterior.

Estudiando los distintos algoritmos de simplificación de mallas, resulta ser un área de estudio con bastantes alternativas e ideas muy creativas para resolver el inconveniente que representa trabajar con mallas con un nivel de detalle inmanejable en las tecnologías disponibles. El trabajo que más resalto por su originalidad, era el trabajo de Décoret et al. [29], Billboard Clouds. Donde se presenta una solución, que a diferencia de las soluciones tradicionales, busca generar una nueva malla, en vez de modificar la existente. Resultando en simplificaciones extremas de un 98% de reducción de polígonos

La *API* WebGL, es una plataforma de desarrollo versátil, que permite el desarrollo e investigación de algoritmos en una plataforma mucho más portable y con mayor soporte entre plataformas, facilitando la colaboración de múltiples desarrolladores en un mismo proyecto. La posibilidad de realizar desarrollos gráficos utilizando un elemento tan portable como lo es el navegador Web, permite masificar con facilidad las aplicaciones. Estos hechos percibidos durante la realización del trabajo de grado, resaltaron la utilidad y la importancia de éste *API* de desarrollo.

Ya descritas las principales conclusiones obtenidas durante el desarrollo del Trabajo Especial de Grado falta mencionar las principales recomendaciones para futuros desarrollos en la aplicación aquí descrita.

La principal dificultad encontrada durante el proceso de desarrollo del algoritmo de simplificación en tiempo real, eran las limitantes en la *API* WebGL, el beneficio de la movilidad que posee al estar implementado en una plataforma tan portable como los navegadores Web, viene con la desventaja de solo poder proveer de funciones genéricas que sean compatibles con la mayor cantidad de dispositivos. Esta desventaja dificultó el desarrollo de algunos pasos importantes en el proceso de simplificación que involucraban herramientas como el *Geometry Shader*. El no poseer un *Geometry Shader* priva de la posibilidad de eliminar las caras que fuesen a ser colapsadas durante el *pipeline* gráfico, causando que sea necesario un paso adicional.

Una funcionalidad indispensable en esta aplicación, es la capacidad de poder cargar cualquier modelo 3D para ser simplificado. Actualmente el algoritmo es capaz de cargar modelos en formato obj y js. Incluso para estos formatos existen modelos con dificultades para ser procesados, debido a diferencias entre diferentes aplicaciones de modelado 3D. Se recomienda ampliar y fortalecer las funcionalidades que se encargan de traducir estos formatos, con el fin de facilitar el uso de la aplicación y aumentar de esta forma su utilidad.

Durante el desarrollo del algoritmo de simplificación de mallas en tiempo real, se notó que múltiples entornos WebGL alojados en una misma página web comparten el *tick* gráfico. El resultado de que múltiples entornos WebGL compartan el mismo *tick* gráfico, es que se procese un mismo número de *frames* por segundo en cada entorno. Ésta cualidad causa que los entornos que posean una menor cantidad de información que procesar generen *frames* con igual velocidad que el entorno con mayor información. En la aplicación desarrollada en el trabajo de

grado esto evita que se pueda medir con exactitud el beneficio resultante del proceso de simplificación. Una mejora de la aplicación actual sería permitir que cada entorno genere *frames* independiente al resto. Permitiendo notar el impacto de la reducción de elementos.

Bibliografía

[1] WebGL, The Khronos Group, www.khronos.org

[2] Hoppe Hughs et al “Mesh Optimization,” Proc. SIGGRAPH 93, ACM SIGGRAPH, Aug. 1993, pp. 19–26.

[3] Lindstrom Peter and Turk Greg. Fast and memory efficient polygonal simplification. In IEEE Visualization, pages 279–286, 1998.

[4] AutoCAD, Autodesk Inc. www.autodesk.com

[5] Cinema 4D, MAXON Computer GmbH www.maxon.net

[6] Mass Effect, Electronic Arts, www.masseffect.com

[7] X-Plane, Laminar Research, www.x-plane.com

[8] Google Earth, Google, www.google.com

[9] The Digital Micheangelo Project, graphics.stanford.edu

[10] The Forma Urbis Romae Fragment, formaurbis.stanford.edu

[11] Constantin Zach, Engine Postmortem of inFAMOUS: Second Son by Adrian Bentley

[12] Schroeder William, Zarge Jonathan, and Lorensen William. Decimation of Triangle Meshes. Conference Proceedings of SIGGRAPH 1992, pp. 65-70.

[13] Garland Michael and Heckbert Paul. Surface simplification using quadric error metrics. In Proc. SIGGRAPH '97, pages 209–216, 1997.

[14] The Computational Geometry Algorithms Library (CGAL), CGAL Open Source Project, www.cgal.org/

[15] Universidad de Utrecht, www.uu.nl

[16] Instituto Federal de Tecnología de Zúrich, www.ethz.ch

[17] Universidad Libre de Berlín, www.fu-berlin.de

[18] Instituto Nacional de Investigación en Informática y Automática, www.inria.fr

[19] Universidad Martin Lutero de Halle-Wittenberg, www.international.uni-halle.de

[20] Instituto Max Planck para la Informática, www.mpi-inf.mpg.de

[21] Universidad de Linz Johannes Kepler, www.jku.at

[22] Universidad de Tel Aviv, english.tau.ac.il

[23] Lindstrom Peter and Turk Greg. Evaluation of memoryless simplification. IEEE Transactions on Visualization and Computer Graphics, 5(2):98–115, slash 1999.

[24] Dey Tamal et al. Topology preserving edge contraction. geometric combinatorics. Publ. Inst. Math. (Beograd) (N.S.), 66:23–45, 1999.

[25] The Visual Toolkit (VTK), Kitware Inc, www.vtk.org

[26] Hoppe Hugh. Progressive Meshes. Conference Proceedings of SIGGRAPH 1996, pp. 99-108.

[27] Lindstrom Peter. Out-of-Core Simplification of Large Polygonal Models. Conference Proceedings of SIGGRAPH 2000, pp. 259-262.

[28] Rossignac Jarek, Borrel Paul. Multi-Resolution 3D Approximations for Rendering Complex Scenes. In Modeling in Computer Graphics, 1993, pp. 455–465.

[29] Décoret Xavier et al. Billboard Clouds for Extreme Model Simplification, SIGG 2003

[30] DeCoro Christopher and Tatarchuk Natalya. Real-time mesh simplification using the GPU. In Proc. of the 2007 Symposium on Interactive 3D Graphics and Games.

[31] Biblioteca Threejs, threejs.org

[32] OrbitControls, github.com/mattdesl/three-orbit-controls

[33] Biblioteca less, lesscss.org/

[34] Biblioteca stats, WebGLstats.com/

[35] Biblioteca Bootstrap, getbootstrap.com/

[36] Plataforma de repositorios Github, github.com/

[37] Modulo para publicación de proyectos GitHub pages, pages.github.com/

[38] Team Tree House, teamtreehouse.com/

[39] Programa Perceptualdiff, github.com/myint/perceptualdiff