PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ

FACULTAD DE CIENCIAS E INGENIERÍA

DISEÑO DE UN ALGORITMO PARA RENDERING EFICIENTE DE ESTRUCTURAS PROTEICAS DE GRAN ESCALA

Tesis para optar el Título de **Ingeniero Informático,** que presenta el bachiller:

Fernando Antonio Moreno Valles

ASESOR: César A. Beltrán Castañón

Lima, julio de 2014

RESUMEN

El software de gráficos por computadora en 3D de hoy en día nos da la capacidad de modelar y visualizar objetos en situaciones o tamaños que antes no habría sido posible, incluso nos dan la capacidad de que la visualización de estos objetos sea generada en tiempo real lo que otorga la posibilidad de crear aplicaciones que hagan uso de esta capacidad para agregar interactividad con los objetos modelados.

Es muy importante la capacidad de poder dotar al usuario de una capacidad de interactividad con el gráfico generado, pero esto no se logra si es que el tiempo de respuesta de la aplicación es muy grande, por ejemplo una consola de videojuegos exigen como mínimo 30fps (cuadros por segundo) un valor menor ocasiona que los movimientos no fueran fluidos y se pierda la sensación de movimiento. Esto hace que la experiencia de usuario fluida sea una de las metas principales del rendering interactivo.

Uno de los mayores problemas que se encuentran en esta área es el de visualizar gran cantidad de polígonos, debido a limitaciones de memoria o capacidad de procesamiento, mientras mayor sea la cantidad de polígonos que se desea dibujar en pantalla, mayor será el tiempo de procesamiento que será necesario para generar las imágenes.

Una aplicación en particular es el de visualización de la estructura de proteínas. Existen proteínas que poseen una gran estructura, por la cantidad de polígonos que se requieren para representar todos los elementos y conexiones que poseen estas moléculas y adicionalmente la necesidad de visualizar grandes cantidades de moléculas simultáneamente, ocasiona que se disminuya el rendimiento y la interactividad al momento de la visualización.

El presente proyecto plantea utilizar una estructura algorítmica para realizar rendering eficiente de gran cantidad de proteínas haciendo uso de un visualizador 3D, que muestre la estructura tridimensional de estas y permita la interacción en tiempo real con el modelo. La estructura propuesta en este proyecto hace uso de la aceleración por hardware presente en las tarjetas gráficas modernas a través de un

API de generación de gráficos en tiempo real que es OpenGL con el cual se aplican optimizaciones que aprovechan la estructura planteada.

Para que el proceso de renderizado sea más veloz, se mantiene un número bajo de polígonos en los modelos. Debido a que los elementos son repetitivos (esferas y cilindros) se reutiliza la geometría de estos elementos haciendo uso de una estructura como el Scene Graph de modo que el uso de memoria sea menor y de otra estructura como el Octree que permite discriminar los elementos que deben ser procesados durante el rendering.

Combinando todo lo mencionado anteriormente, la estructura propuesta permite que se visualicen proteínas de gran estructura o gran cantidad de estas, manteniendo el grado necesario de interactividad para facilitar su estudio así como también manteniendo un aspecto estético que permita reconocer los elementos sin reducir el rendimiento.

Dedicado a mi familia, mis amigos y a todos los que me tuvieron paciencia al escucharme hablar de un tema que no entendían.

Gracias a mi familia que me apoyo durante toda mi carrera y a los amigos que fui haciendo en el camino

Agradezco en especial al Dr. Beltrán que aceptó ser mi asesor y me aconsejó en el desarrollo de todo el proyecto.

Tabla de Contenido

Índice de Figuras

CAPÍTULO 1

1.1 Problemática

El software de gráficos por computadora en 3D de hoy en día nos da la capacidad de modelar y visualizar objetos en situaciones o tamaños que antes no habría sido posible, incluso nos dan la capacidad de que la visualización de estos objetos sea generada en tiempo real lo que otorga la posibilidad de crear aplicaciones que hagan uso de esta capacidad para agregar interactividad con los objetos modelados.

La generación de imágenes en tiempo real o rendering interactivo, ha sufrido un gran proceso de transformación pasando de ser un conjunto fijo de funciones a un modelo programable de procesamiento paralelo [Haines: 2006], debido a este avance, técnicas como el suavizado de sombreado, iluminación, transparencia y reflectividad, logran dar mayor realismo a las imágenes generadas.

El aumento de la capacidad de procesamiento y la reducción de costos del hardware de las computadoras personales actuales ha permitido que el público en general tenga acceso a estas aplicaciones ya sea para educación o entretenimiento, pero también ha permitido a profesionales de otras áreas hacer uso de estas capacidades para la investigación.

En el campo de la biología y en la química el estudio de las moléculas es primordial, sobre estas se analiza su estructura y composición pero debido a su naturaleza microscópica, esta labor se vuelve complicada y no tan fácilmente accesible. Gracias a los gráficos en tiempo real es posible generar y visualizar la estructura tridimensional de las moléculas e incluso añadir interactividad y generar una aplicación que facilita el estudio de estas.

Es muy importante la capacidad de poder dotar al usuario de una capacidad de interactividad con el gráfico generado, pero esto no se logra si es que el tiempo de respuesta de la aplicación es muy grande, por ejemplo una consola de videojuegos exigen como mínimo 30fps (cuadros por segundo) un valor menor ocasiona que los movimientos no fueran fluidos y se pierda la sensación de movimiento. Esto hace

que la experiencia de usuario fluida sea una de las metas principales del rendering interactivo [Haines: 2006].

La mayoría de las técnicas que logran dar mayor realismo y mejoran la estética de los objetos mostrados, deben ser veloces de modo que estas no sacrifiquen el tiempo que se requiere en generar las imágenes ya que esto supondría una pérdida en la interactividad. Por otro lado, hay aplicaciones que requieren que se mantenga un grado de realismo y/o estética por lo que este aspecto tampoco puede ser descuidado.

Uno de los mayores problemas que se encuentran en el área de el rendering interactivo es el de visualizar gran cantidad de polígonos, debido a limitaciones de memoria o capacidad de procesamiento, mientras mayor sea la cantidad de polígonos que se desea dibujar en pantalla, mayor será el tiempo de procesamiento que será necesario para generar las imágenes. La comunidad de gráficos por computadora es muy consciente de este problema en particular [Autin, Johnson, Hake, Olson, Sanner: 2012].

Una aplicación en particular es el de visualización de la estructura de proteínas. Existen proteínas que poseen una gran estructura, por la cantidad de polígonos que se requieren para representar todos los elementos y conexiones que poseen estas moléculas y adicionalmente la necesidad de visualizar grandes cantidades de moléculas simultáneamente, ocasiona que se disminuya el rendimiento y la interactividad al momento de la visualización. Para esta aplicación en particular no se requiere que se mantenga un alto grado de realismo de los objetos pero sí que la estructura pueda ser visualizada con facilidad.

Para poder solucionar el problema se plantea utilizar una estructura algorítmica para realizar rendering eficiente de gran cantidad de proteínas haciendo uso de un visualizador 3D, que muestre la estructura tridimensional de estas y permita la interacción en tiempo real con el modelo.

El visualizador permitiría visualizar las moléculas y su estructura en el modelo de barras y esferas, siendo la esfera el átomo que es representado y las barras son los enlaces entre los átomos. Adicionalmente, se debería poder visualizar las moléculas en el modelo de relleno de superficies, en el cual solo muestra los

átomos representados como esferas de radio proporcional a sus radios atómicos mientras que los enlaces entre átomos no son visibles.

Dados estos 2 modelos de visualización, la aplicación mostraría independientemente los modelos así como también mostrarlos a la vez haciendo uso de transparencia de modo que el modelo de relleno de superficies permitiendo ver el modelo de barras y esferas que pudiese estar oculto.

La estructura de las proteínas se obtendría del Protein Data Bank en formato .pdb, la aplicación cargaría del archivo los datos necesarios para la generación y visualización de la estructura que se desee mostrar.

La estructura propuesta debería poder hacer uso de la aceleración por hardware presente en las tarjetas gráficas modernas a través de un API de generación de gráficos en tiempo real como OpenGL o DirectX, lo cual ayudaría en la eficiencia de la generación de las imágenes.

Usando técnicas de sombreado suavizado (smooth shading) se podría conseguir mejorar la apariencia estética sin necesidad de aumentar polígonos y así conseguir un equilibrio entre apariencia estética y rendimiento.

Para que el proceso de renderizado sea más veloz, se debería de mantener un número bajo de polígonos en los modelos. Debido a que los elementos son repetitivos (esferas y cilindros) se debería poder reutilizar la geometría de estos elementos haciendo uso de una estructura como el Scene Graph de modo que el uso de memoria sea menor y de otra estructura como el Octree que permita discriminar los elementos que deben ser procesados durante el rendering.

Combinando todo lo mencionado anteriormente, la estructura propuesta permitiría que se visualicen proteínas de gran estructura o gran cantidad de estas, manteniendo el grado necesario de interactividad para facilitar su estudio así como también manteniendo un aspecto estético que permita reconocer los elementos sin reducir el rendimiento.

1.2 Marco teórico

En este apartado se presentará los conceptos relacionados con el problema y la solución propuesta para este trabajo de fin de carrera.

1.2.1 Marco conceptual

1.2.1.1 Conceptos relacionados al problema

1.2.1.1.1 Visualización de moléculas

Existen diferentes formas para visualizar las moléculas en un entorno tridimensional, cada una con sus ventajas y desventajas, y de qué forma estos modos de visualización aportan al estudio de las moléculas [Dahlborn: 2003].

A continuación se presentan 2 modelos que son frecuentemente usados en software de visualización de moléculas.

- Barras y esferas(Ball & stick)

En esta representación, los átomos son representados por esferas de un solo color con un radio constante y con el tamaño suficiente para evitar que estas se intersecten. En este modelos se pueden apreciar los enlaces químicos por medio de barras o tubos que conectan las esferas.

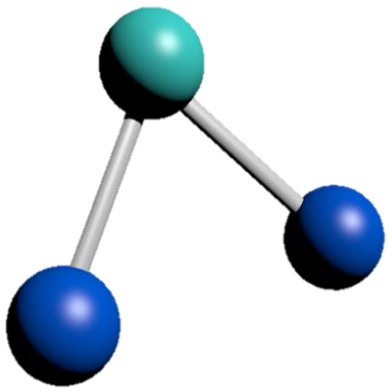


Figura 1.1 Modelo molecular de esferas y barras (ball & stick)

- Relleno de superficies(Spacefill)

En este modelo a diferencia del anterior, no se representan los enlaces entre los átomos, las esferas siguen siendo representación de los átomos pero estas son proporcionales al radio de Van Der Waals, el cuál es el radio de la esfera que representa el volumen del átomo. El modelo requiere mayor cantidad de polígonos para tener una representación correcta ya que en las intersecciones entre esferas se puede apreciar la forma geométrica real del objeto.

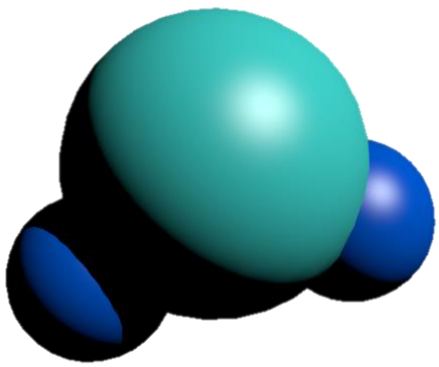


Figura 1.2 Modelo molecular de llenado de superficies (Spacefill)

1.2.1.1.2 Esquema de colores

Ambos modelos tanto el de barras y esferas como el de llenado de superficies pueden usar un esquema de colores llamado CPK en honor a sus creadores Corey, Pauling y luego mejorado por Koltun estos colores son usados por convención por los químicos [Dahlbom, 2003].

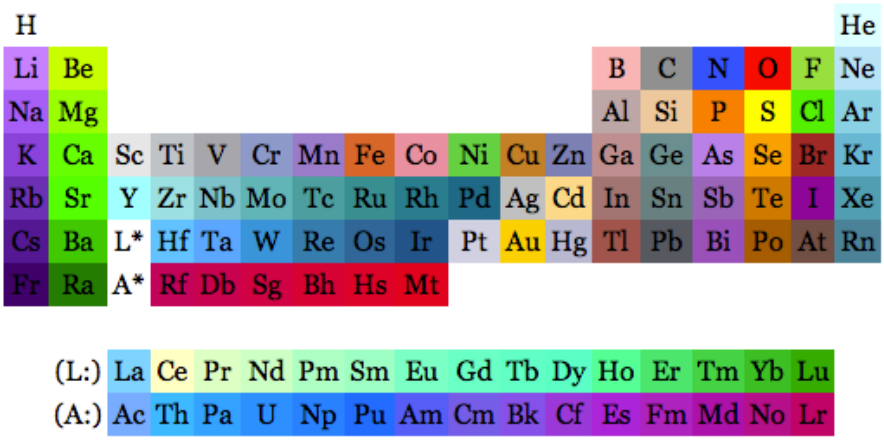


Figura 1.3 Esquema de colores CPK usado en el software Jmol

Fuente: http://jmol.sourceforge.net/jscolors/

1.2.1.1.3 Protein Data Bank

El Protein Data Bank (PDB) es un repositorio mundial para el procesamiento y la distribución de estructura biológica molecular tridimensional. Desde el sitio web de la RCSB (Research Collaboratory for Structural Bioinformatics) se puede acceder a la base de datos PDB en la cual biólogos analizan la estructura de las moléculas de proteínas. A partir del archivo ".pdb" de cada proteína podemos comprender su secuencia de aminoácidos y átomos con sus respectivas coordenadas 3D y la conectividad entre estos [Chen & Chen, 2002].

A partir de estos archivos se puede reconstruir la estructura tridimensional de la proteína y graficarla en un entorno interactivo para facilitar su estudio.

1.2.1.2 Conceptos relacionados a la propuesta de solución

1.2.1.2.1 Representaciones poligonales de objetos.

Los objetos que se quieren renderizar deben poseer una estructura para que puedan ser procesados a través del pipeline de renderizado (ver 2.1.2.2). Para esto la mayoría de objetos son representados a través de una malla compuesta de puntos llamada malla poligonal (mesh) que tiene la forma aproximada del objeto.
Cada punto representa una ubicación en el espacio, y cada uno de estos puntos está conectado con otros formando polígonos, el conjunto de polígonos forma la malla poligonal.

A pesar que es posible representar una malla poligonal haciendo uso de polígonos complejos, lo más frecuente es que estos terminen siendo descompuestos en triángulos ya que los triángulos son más fáciles y más rápidos de renderizar que los demás polígonos [Dahlbom, 2003].

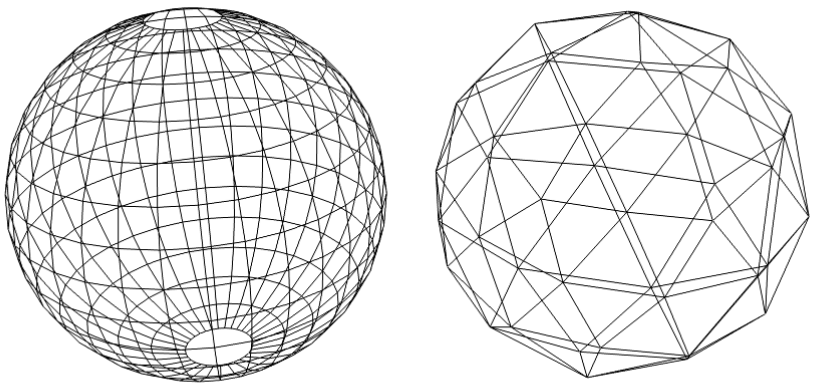


Figura 1.4 Mallas poligonales formadas por cuadrados y triángulos

1.2.1.2.2 El pipeline de renderizado.

El concepto de pipeline es aplicado a los gráficos en tiempo real, el concepto de pipeline consiste en dividir un trabajo grande en varios trabajos pequeños de modo que estos se pueden ir realizando en paralelo. De este modo el proceso de renderizado pasa principalmente por tres etapas: la etapa de aplicación, la etapa de geometría y la etapa de rasterización. La etapa de aplicación es ejecutada en el CPU y no en el hardware gráfico y no forma parte del renderizado en sí. Un ejemplo de operación realizada en esta etapa es la de animación de mallas. Como en todo pipeline, las salidas de esta etapa son las entradas de la siguiente, es decir el resultado de esta etapa es la geometría (puntos, líneas y triángulos) que terminará dibujada en la pantalla. Esta es la tarea más importante de la etapa de aplicación [Akenine-Möller, Haines: 2008].

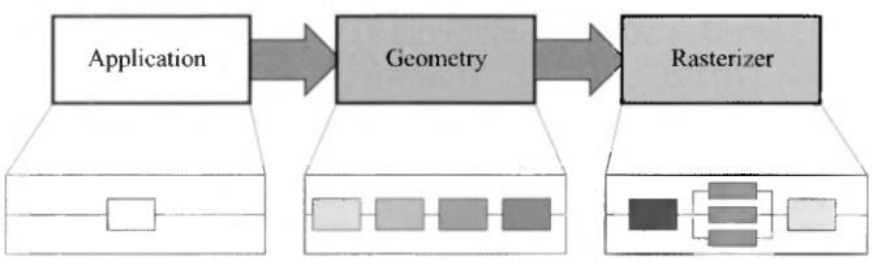


Figura 1.5 Esquema del pipeline de renderizado

Fuente: Real-Time Rendering (2008)

En la segunda etapa, la de geometría, se realizan las operaciones de transformadas de modelo y de vista, esto es, ubicar los objetos a renderizar en la escena alterando su posición, rotación y escala.

Otra operación realizada en la etapa de geometría es la de vertex shading en donde se realizan diversas operaciones a nivel de vértices como el cálculo de cómo la luz afecta a un objeto dependiendo del material del que este compuesto. Luego siguen los procesos de proyección y clipping, el proceso de proyección se encarga de dar el efecto de profundidad a la imagen que será mostrada mientras que el proceso de clipping se encarga de recortar las partes finales de la escena que no se verán en la pantalla y por lo tanto no deben ser dibujadas en pantalla para luego pasar al

proceso de screen mapping que adapta el contenido que será renderizado a el aspecto de la pantalla.

La etapa de geometría también es un pipeline que tiene como fases las anteriormente mencionadas.

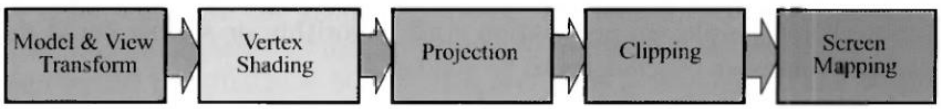


Figura 1.6 Pipeline de la etapa de geometría

Fuente: Real-Time Rendering (2008)

Finalmente se pasa a la etapa de rasterizado. En esta etapa se utiliza los datos entregados por la etapa de geometría para calcular el valor final que tendrá cada pixel que se proyectará en la pantalla así como también las operaciones que se realizan por cada pixel individualmente como el proceso de texturizado, o el de mezclar los colores en caso se desee obtener transparencia en los objetos.

En la actualidad, este pipeline ha sido modificado, tres nuevas fases programables han sido agregadas. Por un lado, se agregó el Geometry shader en la parte final de la etapa de geometría, el cual permite generar o descartar nueva geometría (vértices, líneas, triángulos) en el GPU.

Por otro lado, se agregaron las fases de tessellation control y tessellation evaluation antes y después de la fase de vertex shader respectivamente, las cuales permiten subdividir la malla poligonal en el GPU.

1.2.1.2.3 Técnicas de suavizado sombreado

En ocasiones se requiere que el acabado de un modelo tridimensional se visualice de forma suavizada y no como un objeto conformado por planos, esto se logra haciendo uso de técnicas de suavizado de sombreado [Glassner, 1997].

Gracias a la rápida evolución de las GPU presentes en las tarjetas gráficas, se pasó de un modelo de funciones fijas a un modelo completamente programable que permite mayor versatilidad para lograr los efectos requeridos en la imagen final.

Las etapas del pipeline de renderizado que poseen la capacidad de ser programables son la del vertex shader y del fragment shader los cuales realizan operaciones por cada vértice y por cada pixel (fragmento) respectivamente.

Existen dos técnicas para realizar un suavizado en el sombreado de los modelos, estos son el sombreado Gouraud y el sombreado Phong que operan a nivel de vértices y de fragmentos respectivamente. Antes de describirlos se menciona un tipo de sombreado adicional que no modifica demasiado la apariencia del objeto ya que es un tipo de sombreado muy básico.

- Sombreado Flat(plano)

El sombreado Flat utiliza un solo color por cada cara, usualmente del primer vértice de modo que se pueden identificar fácilmente las caras del modelo que se visualiza en pantalla. Este tipo de sombreado es el más sencillo y el más eficiente ya que no se realiza ninguna operación de cálculo de color por vértice o por pixel.

Figura 1.7 El modelo de un pato con sombreado plano

Fuente: Dahlbom (2003)

- Sombreado Gouraud

El sombreado Gouraud opera a nivel de vértices, por cada vértice se calcula su vector normal de superficie, y con este se calcula la interacción con la fuente o fuentes de luz para obtener el color final del vértice, luego este se interpola a través de las caras a las que pertenece el vértice. Una desventaja es que se necesita una gran cantidad de vértices para que la iluminación se muestre de forma realista.

Figura 1.8 El modelo de un pato con sombreado Gouraud

Fuente: Dahlbom (2003)

- Sombreado Phong

El sombreado Phong a diferencia del sombreado Gouraud opera nivel de fragmentos , usando las normales de los vértices para interpolarlas a través de la superficie y calcular una nueva normal por cada fragmento para luego usar esta normal y calcular el color final de este fragmento. Una desventaja es que calcular normales para cada pixel consume recursos y disminuye el rendimiento.

Figura 1.9 Modelo de un pato con sombreado Phong

Figura 9: Modelo de un pato con sombreado Phong. Fuente: Dahlbom (2003)

Se puede notar como el sombreado plano mantiene su aspecto como si fuera un conjunto de planos mientras que las otras dos técnicas presentan un acabado más suave y realista, el sombreado Phong es incluso más preciso al momento de mostrar la iluminación a diferencia del sombreado Gouraud.

1.2.1.2.4 Scene Graph

El grafo de escena o Scene Graph es una estructura utilizada para solucionar el problema de organizar jerárquicamente los elementos de una escena tridimensional compleja de modo que se facilite el posicionamiento de los elementos que la conforman [Shirley & Marchner, 2009].

Un scene graph es un grafo acíclico dirigido, lo que quiere decir que no tiene ciclos y que los nodos conectados tienen un orden específico, gracias a esta estructura, se pueden aplicar transformaciones (rotación, movimiento y escala) a nodos superiores de modo que esta transformación es aplicada también a todos los hijos de ese nodo lo que facilita la animación y fluidez de esta.

Adicionalmente, un nodo hijo puede tener diferentes nodos padre, lo que implica que 2 objetos presentes en diferentes nodos tengan la misma geometría (compartan el mismo hijo) y que tengan diferente tipo de color o sombreado (diferentes padres) lo que permite la reutilización de los recursos.

Se aplica un algoritmo de recorrido en este grafo para graficar cada elemento, determinar el orden en el cual deben graficarse e incluso para detectar colisiones. Dependiendo de este algoritmo pueden producirse diferentes resultados según el orden con el cual se recorren los nodos. [Karim, Karim, Ahmed, Rokonuzzaman: 2003]

Esta estructura es comúnmente usada en escenas donde hay un gran número de elementos en donde se pueda aprovechar el ahorro de recursos usando este tipo de estructura para luego ser cargada en una lista que otorgaría orden a los elementos que serán renderizados.

1.2.1.2.5 Octree

Octree es una estructura tipo árbol con la cual se puede optimizar el número de elementos que se dibujarán en pantalla ya que esta estructura ayuda a determinar las secciones que serán visibles y la que no, de modo que sólo se dibujan las que serán visibles. Esta estructura también suele ser utilizada para detectar colisiones ya que no hay que realizar la verificación de colisión con cada elemento de la escena, sino solamente con los elementos que se encuentren en la misma sección que el objeto al cual se le realiza las comprobaciones.

El Octree consiste en dividir la escena en ocho partes y cada una de esas en otras ocho y así consecutivamente hasta un nivel dado con lo que se consigue construir un árbol en el cual cada nodo tiene exactamente ocho nodos hijo. El primer nivel se suele subdividir haciendo uso de los ejes cartesianos de modo que la escena al ser de 3 dimensiones cuenta con 8 octantes fácilmente identificables.

El beneficio de usar una estructura de este tipo viene del hecho en que si un nodo superior no es visible, ninguno de sus hijos lo será de modo que no es necesario hacer cálculos adicionales. [Karim, Karim, Ahmed, Rokonuzzaman: 2003]

1.2.1.2.6 Nivel de detalle dinámico

Para simular la realidad, los objetos dibujados en la pantalla deben poder contar con detalles que proporcionen información sobre la forma y las características que hagan capaces de identificar un objeto como tal.

En una escena grande, se pueden presentar objetos que se encuentran a diferentes distancias del punto de vista del observador, los objetos que se encuentran a mucha distancia, no requieren un nivel de detalle muy grande ya que estos no pueden ser detectados fácilmente por el ojo humano. Por otro lado, un objeto cercano si debe presentar la mayor cantidad de detalles posibles para poder asegurar el realismo de la imagen presentada en pantalla.

Existe una técnica llamada nivel de detalle dinámico (DLOD, por sus siglas en inglés), la cual consiste en cambiar el nivel de detalle según ciertos parámetros de la escena, como distancia o iluminación, reduciendo o aumentando en tiempo real el número de polígonos necesarios de la malla poligonal de dicho objeto.

Hay varias formas de implementar esta técnica, ya sea almacenando los datos de diferentes mallas para reemplazarlas al dibujar o con una misma malla pero modificando su número de polígonos directamente.

1.3 Estado del arte

Se han realizado anteriormente algunas aplicaciones con el mismo propósito que es el de visualizar las estructuras moleculares entre otras, a continuación se presentará alternativas distintas que han sido realizadas en los últimos años.

1.3.1 Visualización molecular eficiente en sistemas de gran escala

Es una tesis de maestría realizada en Espoo, Finlandia por Matti Dahlbom. Como su nombre lo indica, se desarrolló una aplicación para la visualización de moléculas y presentar métodos para visualizar eficientemente grandes estructuras.

La aplicación se realizó en el lenguaje Java para potabilidad en diferentes sistemas operativos y usando también el API de OpenGL por lo que se aprovechó la aceleración por hardware de las tarjetas gráficas.

Para el momento que se realizó la tesis, el API de OpenGL aún ofrecía la funcionalidad fija por lo cual no se pudo implementar el sombreado Phong sino que se optó por el sombreado Gouraud que si era soportado.

La optimización se realizó usando Scene Graph estableciendo una jerarquía entre la molécula (padre) y los átomos y conexiones (hijos), de modo que se pudiera generar la estructura de la molécula ubicando cada átomo y conexión y luego poder posicionar y rotar la molécula entera.

1.3.2 uPy: un API de CG con aplicaciones en modelado de moléculas

El proyecto consistió en crear un API hecho en Python para unificar la manera de generar modelos en 3D a través de diferentes aplicaciones de modelado. Entre las aplicaciones se destaca el uso conjunto con ePMV (embedded Python Molecular Viewer) que es también un plug-in para la visualización de estructuras moleculares.

Estos plug-ins se integran con software de generación de gráficos tanto propietarios como software libre, como Blender, 3D Studio Max y maya y hace uso de las capacidades de renderizado de estas aplicaciones para generar y mostrar la molécula en el visualizador de este software. Al ser integrados con este software, la optimización usando Scene Graph se delega al software de modelado que podría usar esta técnica u otra similar.

Los plug-ins permiten que se visualicen las moléculas en diversos modelos de visualización tales como el modelo de barras y esferas, el modelo de rellenado de superficies y el modelo de representación en cintas. También hace uso de los esquemas de colores CPK y de coloreado por factor de temperatura cristalográfica [Dahlbom, 2003].

1.3.3 GLmol

GLmol es un visualizador de moléculas cuya principal virtud es funcionar en el navegador web sin hacer uso de ningún plug-in. GLmol funciona en HTML5 y JavaScript haciendo uso de WebGL.

WebGL es un API similar a OpenGL pero orientado al funcionamiento en el navegador web lo cual es un beneficio en portabilidad ya que solo es necesario contar con un navegador web y puede ser usado también en dispositivos móviles. Como desventaja presenta que tiene un API más reducida que OpenGL y también su rendimiento depende del navegador web que se esté usando y por eso no se obtiene el mismo rendimiento que OpenGL nativo.[Hoetzlein, 2012].

GLmol usa la librería THREE.js que implementa un motor para renderizar los objetos en 3D haciendo uso de Scene Graph para evitar consumir excesiva

memoria. Adicionalmente utiliza el esquema de color CPK y los modelos de barras y esferas y rellenado de superficie. Como principal limitación tiene que los modelos de barras y esferas no pueden ser visualizados con eficiencia si estos poseen un gran número de átomos.

Existen también en la actualidad productos especializados para el campo de modelado molecular, específicamente para visualización interactiva de proteínas. Algunos de estos productos son la evolución de otros que en su época pudieron ser tecnología de punta pero que con el paso del tiempo no lograron actualizarse pero sirvieron de base para una nueva generación.

1.3.4 Productos comerciales para resolver el problema

- Molsoft

Molsoft es software propietario, además de ofrecer visualizadores de proteínas tiene una amplia gama de productos que incluyen predicción de estructuras, acoplamiento molecular entre otros.

Molsoft posee también un visualizador de moléculas para dispositivos móviles con Android y iOS.

1.3.5 Productos no comerciales para resolver el problema

- RasMol

Desarrollado por Roger Sayle y publicado en el paper Rasmol: Biomolecular graphics for all en el año 1995. RasMol es un programa para la visualización de gráficos moleculares tales como proteínas, ácidos nucleicos y pequeñas moléculas

El programa se desarrolló en la universidad de Edinburgh en la unidad de investigación bio-computacional. Este software fue escrito en lenguaje C y su código es abierto. Su última versión estable es la 2.7.5.1 lanzada en julio del 2009.

Este software ofrece la posibilidad de visualizar las moléculas en diferentes esquemas de colores y visualización y define un estándar de software de visualización molecular [Dahlbom, 2003].

- Jmol

Software libre para visualización 3D de moléculas y de código abierto, fue desarrollado en lenguaje Java con lo que lo hace compatible con todos los sistemas operativos que soporten la máquina virtual de Java.

Es derivado de XMol, un software desarrollado en el Minnesota Supercomputer Center en el año 1999. Una de las características más resaltantes de este software es que no utiliza aceleración por hardware así que no es necesario tener una tarjeta gráfica potente para utilizarlo [Herráez, 2006].

1.3.6 Conclusiones

Los visualizadores de moléculas más recientes están haciendo uso de tecnologías nuevas como WebGL que permiten mayor portabilidad y un acceso más fácil a las aplicaciones al requerir solamente de un navegador, pero esto viene con una consecuencia, que es la de pérdida de rendimiento ya que depende del navegador usado.

Los visualizadores de escritorio usan un lenguaje de alto nivel o funcionan como plug-in de otro software visualizador lo que hace que dependa de la presencia de software adicional lo cual limita la portabilidad.

La solución propuesta en este trabajo de fin de carrera ofrece una estructura algorítmica que haga uso de la aceleración de hardware de modo que se mantenga la eficiencia y se permita la interactividad.

1.4 Objetivo general

Desarrollar una estructura algorítmica basada en scene graph para una eficiente visualización de estructuras proteicas de gran escala.

1.5 Objetivos específicos

1. Desarrollar un visualizador de proteínas haciendo uso de un API de gráficos.

2. Desarrollar un software de carga de archivos ".pdb" para la interpretación de la información sobre las proteínas.

3. Renderizar las imágenes usando algoritmos de sombreado que no añadan excesivo realismo de modo que se mantenga la eficiencia.

4. Implementar la estructura usando scene graph y octree como estructura de datos para visualizar gran cantidad de objetos en la escena.

5. Implementar técnicas de optimización que aprovechen la estructura propuesta

6. Evaluar la eficiencia de la estructura implementada a través de experimentación numérica.

1.6 Resultados esperados

1. Resultado 1 para el objetivo 1: Visualizador de proteínas.

2. Resultado 2 para el objetivo 2: Software de carga de archivos ".pdb"

3. Resultado 3 para el objetivo 3: Informe de experimentación numérica y algoritmo de sombreado elegido.

4. Resultado 4 para el objetivo 4: Estructura de datos basada en scene graph y octree para la representación de las proteínas.

5. Resultado 5 para el objetivo 4: Método para recorrer la estructura de datos de forma eficiente para el renderizado.

6. Resultado 6 para el objetivo 5: Aplicación de técnica de nivel de detalle dinámico, frustum culling y renderizado múltiple.

7. Resultado 7 para el objetivo 6: Informe de experimentación numérica, comparando la velocidad del visualizador utilizando la estructura y sin utilizarla.

1.7 Herramientas, métodos y procedimientos

1.7.1 Mapeo

Resultados esperado	Herramientas a usarse
Software visualizador de proteínas	-Programación orientada a objetos -Modelo de ciclo de vida incremental -OpenGL
Software de carga de archivos .pdb	-Programación orientada a objetos -Modelo de ciclo de vida incremental
Informe de experimentación numérica y algoritmo de sombreado elegido.	-Algoritmos de sombreado (shading) que aumentan el realismo sin modificar la geometría de los objetos. - OpenGL (GLSL)
Estructura de datos basada en scene graph y octree para la representación de las proteínas.	-Estructura algorítmica basada en scene graph y octree -Programación orientada a objetos
Método para recorrer la estructura de datos de forma eficiente para el renderizado	- Programación orientada a objetos
Aplicación de técnica de nivel de detalle dinámico, frustum culling y renderizado múltiple	- Tessellation Shaders - Geometry Shaders - OpenGL
Informe de experimentación numérica, comparando la velocidad del visualizador utilizando la estructura y sin utilizarla.	- Prueba de hipótesis

1.7.2 OpenGL

OpenGL (Open Graphics Library) es un API para hardware de gráficos. El API consiste de una serie de funciones que apoyan al programador a especificar los objetos y operaciones a realizar para obtener gráficos tridimensionales de alta calidad.

Un programa típico de OpenGL contiene las llamadas necesarias para inicializar el contexto de donde se dibujarán los gráficos. Una vez que esto se logra los siguientes comandos del API se usan para definir el contenido a dibujar, esto incluye la geometría de los objetos, sus texturas y sus materiales. [M. Seagal, K. Akeley: 2013]

1.7.3 GLSL

GLSL (OpenGL Shading Language) es un lenguaje de programación con el cuál se escriben los programas que serán ejecutados por la GPU más conocidos como shaders. GLSL posee una sintaxis parecida a la del lenguaje C, pero con ciertas operaciones adicionales que facilitan el manejo de vectores y matrices.

GLSL fue introducido en la versión OpenGL 2.0 Core y entre los beneficios que supuso esta introducción se encuentra la compatibilidad entre distintas plataformas y la capacidad de hacer uso de hardware de forma eficiente ya que cada fabricante puede optimizar el compilador de GLSL para aprovechar la arquitectura de un hardware en particular.

1.7.4 Algoritmos de sombreado

Los algoritmos de sombreado hacen uso de la iluminación para poder asignar un color a los objetos que son renderizados sin necesidad de aumentar vértices en su geometría lo que reduce el costo de memoria y CPU necesario para aumentar el realismo de la imagen generada.

Se escogen los algoritmos de Gouraud y Phong, que operan a nivel de vértices y a nivel de píxeles respectivamente para determinar el color final que tendrá el objeto en un punto específico en función a la dirección del vector normal.

1.7.5 Tessellation Shaders

Los tessellation shaders son un tipo particular de programas que se encargan de subdividir la geometría haciendo uso de la capacidad de paralelismo de la GPU. La etapa de tessellation consta de las siguientes tres partes: tessellation control,

en la cual se determinan la cantidad de subdivisiones que se realizará; tessellation generation, en la cual se genera la nueva geometría y tessellation evaluation, en donde se puede realizar operaciones adicionales a los nuevos vértices generados. Solamente la parte de tessellation generation no es programable.

1.7.6 Geometry Shaders

Los Geometry shaders, no operan sobre un elemento únicamente sino que pueden acceder a otros vértices, líneas o triángulos. Adicionalmente a esta capacidad, los Geometry shaders pueden generar nuevos vértices o triángulos así como también descartar algunos para que no sean procesados por las siguientes etapas del pipeline de renderizado.

1.7.7 Estructura algorítmica basada en scene graph y octree

Se utilizará una estructura algorítmica basada en scene graph de modo que se pueda reutilizar los elementos existentes en la escena. En este caso, al ser conformado solo por esferas y cilindros, se puede aprovechar esta estructura para ahorrar recursos de cómputo y a su vez poder renderizar los elementos de forma eficiente. [Reiners: 2002]

El octree ayudará para descartar los objetos que no sean visibles de la escena durante la etapa de aplicación de modo que se reduzca los cálculos necesarios para el rendering dado que sólo se procesará la información que llegue finalmente a la pantalla.

1.7.8 Modelo de ciclo de vida incremental

Se basa en ir construyendo el software por funcionalidades o incrementos, con esto se logra dividir el software en partes más simples y sencillas de implementar. Adicionalmente, permite que luego de cada incremento se realicen pruebas unitarias, de validación y de aceptación antes de comenzar con el siguiente incremento.

1.7.9 Programación orientada a Objetos

Se utilizará el paradigma de programación orientada a objetos debido a que el nivel de abstracción que provee este paradigma permite representar con mayor facilidad elementos como una malla poligonal, una textura o un material.

1.8 Alcance

Este proyecto pretende proporcionar una herramienta que facilite el estudio de las estructuras moleculares de proteínas de gran escala a través de un graficador en tres dimensiones que haga uso eficiente de alguna estructura algorítmica para mantener la eficiencia.

1.8.1 Limitaciones

La aplicación graficará haciendo uso de solamente 2 modelos de representación de moléculas los cuales son el modelo de barras y esferas y el modelo de rellenado de superficie. No se implementará el modelo de cinta ya que esta solamente muestra información muy básica sobre la estructura de una proteína.

Adicionalmente, la data se extraerá de archivos ".pdb" únicamente debido a la gran disponibilidad de estos archivos en el wwpdb (world wide protein data bank) y también debido a que está correctamente estructurado y documentado lo que facilita su uso y lectura.

La estructura algorítmica estará basada en Scene Graph y Octree para mostrar eficientemente gran cantidad de elementos porque permite reutilizar elementos comunes en la escena como las mallas poligonales, shaders y texturas y permite procesar solamente los objetos que estén en el campo de visión.

1.8.2 Riesgos

Riesgo identificado	Impacto en el proyecto	Medidas correctivas para mitigar
Indisponibilidad de base de datos de proteínas	No se contaría con los datos necesarios para la visualización.	Obtener una copia local de cierto número de archivos del wwpdb
Estructura scene graph y octree poco óptimas para su aplicación en la visualización de grandes BD	El resultado final se vería afectado en términos de eficiencia.	Aplicación de técnicas adicionales de optimización utilizando las estructuras propuestas.
No se tiene disponibilidad del hardware necesario para implementar las funcionalidades requeridas	La implementación de la estructura no sería aprovechada por completo	Se buscarán alternativas que sacrifiquen un poco de eficiencia por compatibilidad.
Demasiados algoritmos de sombreado para ser elegidos	Algoritmo elegido impacta en los recursos y disminuye la eficiencia	Limitar el número de algoritmos de sombreado a experimentar.
La implementación y la data no fueron adecuadamente implementadas.	Resultado final incompleto o que genera visualizaciones incorrectas.	Se utilizará un modelo incremental y se hará la comparación con productos o aplicativos similares.

1.9 Justificación y viabilidad

1.9.1 Justificativa del proyecto de tesis

El proyecto de fin de carrera servirá para apoyar la futura investigación en la estructura molecular de proteínas ya que proveerá la posibilidad de visualizar de manera interactiva y simultánea numerosas estructuras moleculares en dos diferentes representaciones.

El proyecto aportará con una estructura algorítmica que permitirá una visualización eficiente de las proteínas de gran escala manteniendo la interactividad y fluidez de la visualización.

La estructura podrá ser reutilizada o adaptada para otras aplicaciones de gráficos en computación.

El proyecto contribuirá con una base para futuros trabajos en los que se puede ampliar la funcionalidad de este aplicativo agregando la capacidad de comparación de estructuras o predicción de acoplamiento de dos proteínas.

1.9.2 Análisis de viabilidad del proyecto de tesis

El proyecto hará uso de metodologías que han sido probadas previamente en publicaciones académicas por lo cual es menos probable que la elección de una de estas haga inviable el proyecto.

En cuanto a la recolección de datos, se cuenta con una base de datos mundial de proteínas que tiene crecimiento diario la cual es la wwpdb, esto asegura la viabilidad en este aspecto ya que se cuenta con gran cantidad de datos de pruebas que son fácilmente accesibles y están validados.

Se cuenta con facilidad en el acceso a las herramientas necesarias, estas se encuentran documentadas y también se cuenta con cierta experiencia su manejo. En el caso de OpenGL este estándar es desarrollado por el grupo Khronos es actualizado frecuentemente y las funciones obsoletas no son

removidas de la especificación por lo que las aplicaciones implementadas no dejan de funcionar con versiones posteriores de OpenGL.

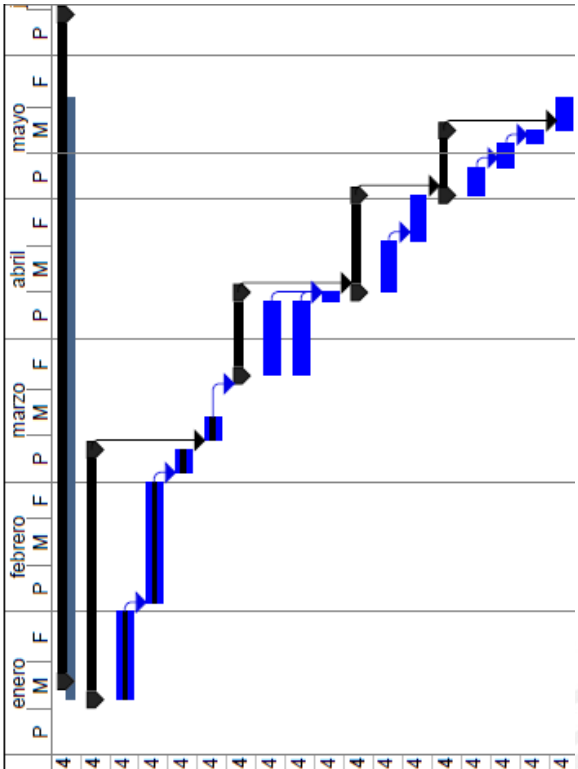
El desenvolvimiento de este proyecto en todas fases del ciclo de vida de software será independiente de factores externos, por lo que esto no presenta una amenaza a la viabilidad.

En cuanto a las necesidades, se requiere hardware necesario para correr el renderizador de OpenGL en tarjetas gráficas que soporten la especificación 4.3, las cuales son cualquier tarjeta gráfica dedicada que haya sido lanzada al mercado desde el año 2010, lo cual no afecta necesariamente la viabilidad económica del proyecto considerando que no se necesitará adquirir algún recurso extra.

En cuanto a necesidades de software, se dividirán en el software a usarse y la documentación. Con respecto a la disponibilidad de los recursos de software a utilizarse, no hay un impacto en la viabilidad debido a que estas herramientas son de licencia libre. Asimismo la documentación requerida se obtiene sin dificultad y de fuentes oficiales lo que garantiza que sean actualizadas, relevantes y correctamente estructurada.

1.10 Plan de actividades

Id	Nombre de tarea	Duración	Comienzo	Fin
1	**Software Visualizador de Proteínas**	**102 días**	**vie 17/01/14**	**lun 09/06/14**
2	**Implementación del visualizador**	**40 días**	**lun 13/01/14**	**vie 07/03/14**
3	Diseño del visualizador	15 días	lun 13/01/14	vie 31/01/14
4	Implemetación del visualizador	20 días	lun 03/02/14	vie 28/02/14
5	pruebas del visualizador	5 días	lun 03/03/14	vie 07/03/14
6	Software de lectura de archivos .pdb	5 días	lun 10/03/14	vie 14/03/14
7	**Implementación de algoritmo de sombreado**	**14 días**	**lun 24/03/14**	**jue 10/04/14**
8	Implementación de Gouraud Shading	12 días	lun 24/03/14	mar 08/04/14
9	Implementación de Phong Shading	12 días	lun 24/03/14	mar 08/04/14
10	Informe de comparación de algoritmos	2 días	mié 09/04/14	jue 10/04/14
11	**Implemetación de estrucutra de datos basada en**	**15 días**	**vie 11/04/14**	**jue 01/05/14**
12	Diseño e implementación de la estructura	7 días	vie 11/04/14	lun 21/04/14
13	Diseño e implementación del método para recorrer la	8 días	mar 22/04/14	jue 01/05/14
14	**Implementación de optimizaciones a la estructura**	**10 días**	**vie 02/05/14**	**jue 15/05/14**
15	Implementación de nivel de detalle dinámico	4 días	vie 02/05/14	mié 07/05/14
16	implementación de frustum culling paralelo	3 días	jue 08/05/14	lun 12/05/14
17	implementación de rederizado múltiple	3 días	mar 13/05/14	jue 15/05/14
18	Evaluación de desempeño	5 días	vie 16/05/14	jue 22/05/14



CAPÍTULO 2

VISUALIZADOR 3D

La estructura algorítmica que se desarrollará necesita de un visualizador en donde pueda ser aplicada y posteriormente poder apreciar si es que esta mejora el rendimiento al renderizar gran cantidad de objetos, en este caso proteínas.

El visualizador 3D se desarrolló de modo que se pudiera representar una escena básica, esto significa contar con objetos, geometrías, materiales, luces, cámaras y los procedimientos para convertir estos elementos en una escena visible.

2.1 Estructura del visualizador

El visualizador está compuesto de un conjunto de clases, entre ellas, dos de las más importantes son la clase Scene y la clase Renderer. La clase Scene es aquella que contiene los datos necesarios para representar la escena, mientras que la clase Renderer es aquella que contiene la lógica necesaria para convertir los datos de Scene en el dibujo final en la pantalla.

La separación de la lógica y los datos brinda flexibilidad al visualizador, gracias a esta separación, la clase Renderer podría hacer uso de un API de gráficos distinto de OpenGL como lo es DirectX y no habría necesidad de cambiar nada en los datos.

2.1.1 Scene

La clase Scene es la que contiene los datos acerca de la escena, los objetos, luces, materiales y cámaras. Esta clase posteriormente pasa a convertirse en el Scene Graph, el cual forma parte de la estructura algorítmica que se propone.

Cada escena tiene una cámara, la que se encarga de determinar el volumen que se visualizará; una lista de luces, que son las que en combinación da color a la escena y una lista de objetos, que son los objetos que conforman la escena.

Todo elemento de la escena se representa como una instancia de la clase Object3D. Estos elementos pueden ser visibles o no, pero lo que los identifica es

que son objetos que cuentan con una posición en la escena. Para dar mayor orden a la escena, existen clases derivadas de Object3D que se encargan de representar de mejor forma a los objetos de una escena.

Los objetos visibles y dibujables están representados mediante la clase Mesh, esta clase es la que contiene referencias a la geometría y material del objeto que se dibujará. Los datos acerca de la posición, rotación y escala los heredan de la clase Object3D. Otras clases que heredan de Object3D son las cámaras y las luces, ya que requieren tener una posición en la escena, pero estos objetos no se dibujan.

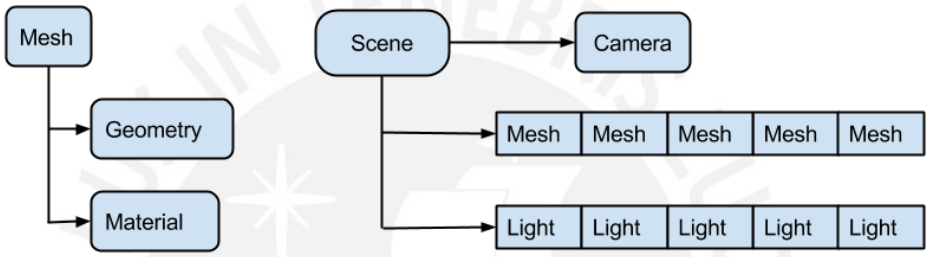


Figura 2.1 Representación básica de la escena

La clase Renderer es la que se encarga de tomar los datos presentes en la escena, almacenados en un objeto scene, y realizar las llamadas a la API de OpenGL necesarias para dibujar los elementos en la pantalla.

Otra de sus tareas es estructurar los datos de cada uno de los objetos en una representación que OpenGL pueda procesar, así como también de realizar las operaciones de renderizado que sea realizan en la CPU, esta es conocida como la etapa de aplicación.

La independencia entre esta clase y los datos hace que sea sencillo implementar e integrar la estructura que se diseñó en este proyecto ya que la estructura no modifica esta clase. Adicionalmente, cualquier cambio en la API o nueva versión de OpenGL puede ser integrada con el visualizador haciendo cambios en la clase Renderer solamente sin tener que modificar la estructura algorítmica de la escena.

2.2 Bucle de renderizado

Para dibujar los objetos en la pantalla es necesario realizar una serie de operaciones sobre los datos para que OpenGL pueda finalmente procesar la escena en una imagen visible. En la versión moderna de OpenGL, a partir de la versión 3.2, ya no existe el "modo inmediato" de OpenGL, en el cual solo había que especificar los parámetros de la escena y dibujar objeto por objeto.

Para poder dibujar, en la versión moderna de OpenGL, es necesario primero reservar la memoria en la tarjeta gráfica en los llamados buffers para que OpenGL pueda utilizarlos. Cada geometría de la escena es procesada, se crean buffers independientes para los vértices, que son los datos dibujables; para los vectores normales, que ayudan a determinar el color final de cada pixel y para los índices, que son los que indican que vértices componen cada triángulo.

El siguiente paso es reservar en la memoria los parámetros generales de la escena como las matrices que generan la perspectiva en la escena y los datos acerca de cada una de las fuentes de luz que interactúan con esta. Estos datos globales son almacenados en buffers independientes.

Los materiales son los siguientes en ser procesados, por cada material se crea también un buffer donde almacenar las propiedades. Un siguiente paso, el cual es muy importante, es crear y compilar los shaders por cada tipo de material que exista en la escena, estos shaders son los que procesarán las geometrías y materiales de cada objeto.

Una vez que todos los buffers y shaders están creados, se comienza a procesar objeto por objeto. Por cada objeto se actualizan sus buffers y se calcula la matriz que posiciona el objeto en la escena a partir de su posición, rotación y escala. A continuación se indica a la API los buffers que se utilizarán para la próxima llamada "draw", que es la encargada de dibujar.

Finalmente, se asignan los shaders que se utilizarán y se indica cual es la distribución de los vértices en los buffers y se realiza la llamada draw. Se procede a realizar estos pasos por el resto de los objetos que pertenezcan a la escena.

2.3 Bucle principal del visualizador

Existe también una serie de pasos que se realizan en la aplicación para que sea posible visualizar e interactuar con la escena que será renderizada. Los datos deben ser cargados a la aplicación y se debe procesar la entrada del usuario para poder lograr esto.

Para que todo esto sea posible, debe existir una ventana y un contexto donde se pueda dibujar el resultado de la renderización. Este es el primer paso que se realiza en el visualizador ya que las llamadas a la API de OpenGL no funcionarán si es que no existe un contexto.

Una vez que se tiene un contexto y ventana donde visualizar los resultados, es necesario cargar los datos que se quieren dibujar. Se crea un objeto escena para almacenar los componentes de esta y deben crearse y añadirse también los objetos, geometrías, luces y materiales.

A continuación comienza el bucle principal del visualizador, es un bucle debido a que se necesita que se genere una imagen varias veces por segundo para poder tener interactividad. Se comienza procesando la entrada del usuario, cada movimiento o click del mouse y cada pulsación de teclado genera un evento que puede ser procesado o no según sea requerido, en caso si lo sea, se actualizan los datos de la escena para reflejar la acción deseada, por ejemplo, rotar la cámara.

Luego de que la entrada del usuario fue procesada, si es que esta no es la señal para terminar el programa, se procede a llamar al objeto Renderer que se encargará de realizar los pasos especificados en la sección anterior y presentar el resultado en la pantalla.

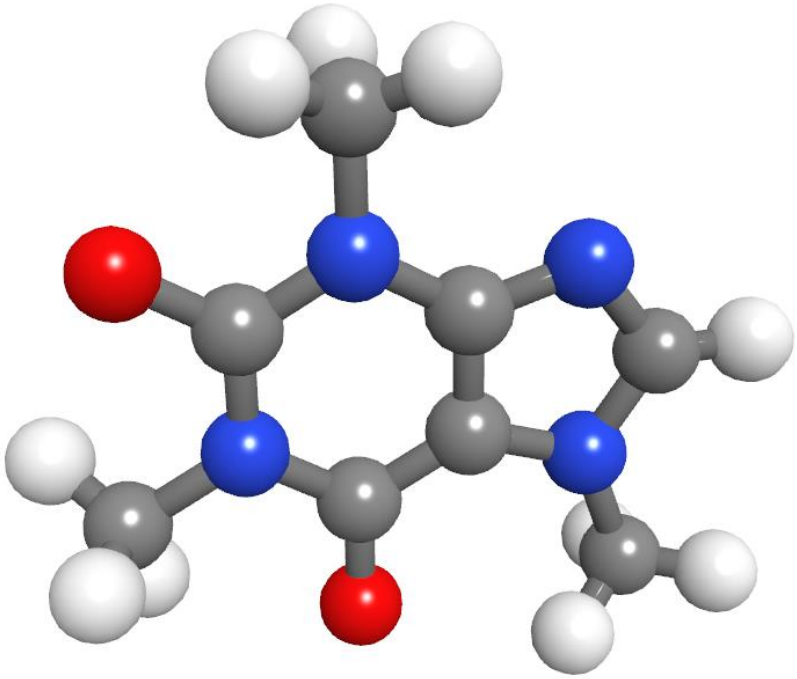


Figura 2.2 Imagen generada por el visualizador 3D

CAPÍTULO 3

SOFTWARE DE CARGA DE ARCHIVOS PDB

3.1 El formato PDB

El archivo PDB contiene la información necesaria para comprender la estructura de las proteínas, sus secuencias de aminoácidos y sus átomos con sus posiciones en el espacio tridimensional así como también la conectividad que existe entre estos la cual puede estar expresada de forma explícita a través un registro específico y también de forma implícita, teniendo que realizar algunos cálculos para obtenerla. A continuación se presenta como ejemplo un extracto de un archivo PDB.

```
HEADER        OXYGEN TRANSPORT                12-SEP-12   4H2L
TITLE  DEER MOUSE HEMOGLOBIN IN HYDRATED FORMAT

ATOM      1  N   VAL A   1      -9.426 -36.528  22.645  1.00 41.73           N
ATOM      2  CA  VAL A   1      -9.143 -37.100  21.337  1.00 38.88           C
ATOM      3  C   VAL A   1     -10.323 -36.812  20.406  1.00 31.57           C
ATOM      4  O   VAL A   1     -11.299 -36.184  20.810  1.00 35.22           O
ATOM      5  CB  VAL A   1      -8.877 -38.626  21.449  1.00 41.02           C

HETATM 2174  CHA HEM A 201       6.677 -29.294   2.095  1.00 17.27           C
HETATM 2175  CHB HEM A 201       2.451 -30.785   3.955  1.00 14.65           C
HETATM 2176  CHC HEM A 201       2.401 -26.978   7.030  1.00 14.39           C
HETATM 2177  CHD HEM A 201       6.341 -25.221   4.788  1.00 16.58           C

CONECT  635 2216
CONECT 1757 2259
CONECT 2174 2178 2205
```

Lo archivos .pdb son archivos de texto, cada línea de este archivo es un registro. Estos registros tienen sus campos de ancho fijo, esto es, cada campo tiene una longitud de caracteres fija que se completará con espacios en blanco si es que los datos no ocupan toda la longitud. Los primeros seis caracteres son los que determinan la información que contiene cada línea del archivo.

3.1.1 Tipos de registros

El registro HEADER es para definir la proteína contiene la clasificación de esta, su fecha de ingreso a la base y su código.

Los registros ATOM son los que brindan la información acerca de los átomos que conforman la proteína. Contiene un índice, el nombre del átomo, el nombre del aminoácido al que pertenece. Las columnas 6, 7, y 8 son las coordenadas X, Y, Z respectivamente en Angstroms. Los registros HETATM contienen la misma información que ATOM pero son los de átomos heterogéneos.

Los registros CONECT representan la conectividad entre los átomos, están compuestos por varios números. Para interpretar el registro, se toma el primer número que es el índice del átomo, los siguientes números son los índices de los a los que el primer átomo está conectado.

3.1.2 Conectividad

La conectividad de los átomos no siempre está representada por algún registro y debe ser calculada en términos de las posiciones relativas entre dos átomos e incluso los elementos de cada átomo.

Para calcular la conectividad primero hay que determinar si alguno de los dos átomos es un átomo de hidrógeno, en caso sea así, habrá conectividad si se cumple lo siguiente:

$$0.4A < d < 1.9A$$

Siendo d la distancia entre los dos átomos en Armstrong

En caso ninguno de los átomos sea de hidrógeno, habrá conectividad si se cumple lo siguiente:

$$0.4A < d < 1.2A$$

Siendo d la distancia entre los dos átomos en Armstrong

3.2 Estructura para almacenar la proteína

3.2.1 Consideraciones previas

Antes de poder cargar el archivo .pdb se deben tener en cuenta que se necesita información adicional que no se encuentra en un .pdb, esto es el color del átomo y su radio de Van der Waals.

Ya que estos valores no están incluidos en los .pdb, se crearon 2 archivos adicionales que contenían los radios de Van der Waals y los colores de los elementos según el esquema CPK. Estos archivos son cargados antes de cargar el archivo .pdb y se realiza en contenedores asociativos, el contenedor map de la librería STL de C++, utilizando como valor identificador el nombre del elemento químico.

3.2.2 Lectura y almacenamiento

La geometría utilizada por las proteínas está conformada por esferas y cilindros que solo se diferencian por su posición, dimensiones y orientación. Estos parámetros pueden ser alterados sin necesidad de alterar la geometría original, por lo cual se crea una geometría de cilindro y una de esfera que serán compartidas por todos los átomos.

El archivo .pdb se lee línea por línea, cuando se encuentra un registro ATOM o HETATM, se crea un objeto Atom la cual es una clase derivada de Mesh que tiene atributos adicionales como el radio de Van der Waals y el nombre del elemento.

Se crea un segundo objeto Atom que será el que contenga la información de la representación ball & stick, así se cuenta con los dos objetos Atom, ambos hacen referencia a la misma geometría y al mismo material pero su escala es diferente.
Cada uno de estos objetos Atom es luego agregado a un objeto de la clase Molecule, la cual es una clase derivada de Object3D que adicionalmente cuenta con tres listas: la lista de átomos para la representación Spacefill, una lista de

átomos para la representación ball & stick y una lista de objetos Mesh que son las conexiones entre átomos.

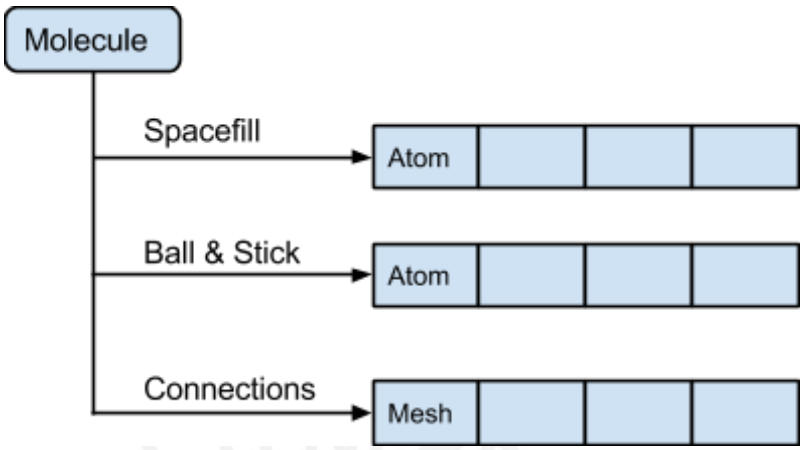


Figura 3.1 Estructura de un objeto Molecule

Una vez que se terminó de leer los registros ATOM se leen los registros CONECT y se calculan las conexiones no explícitas, se crean los objetos Mesh que los representan y se añaden al objeto Molecule.

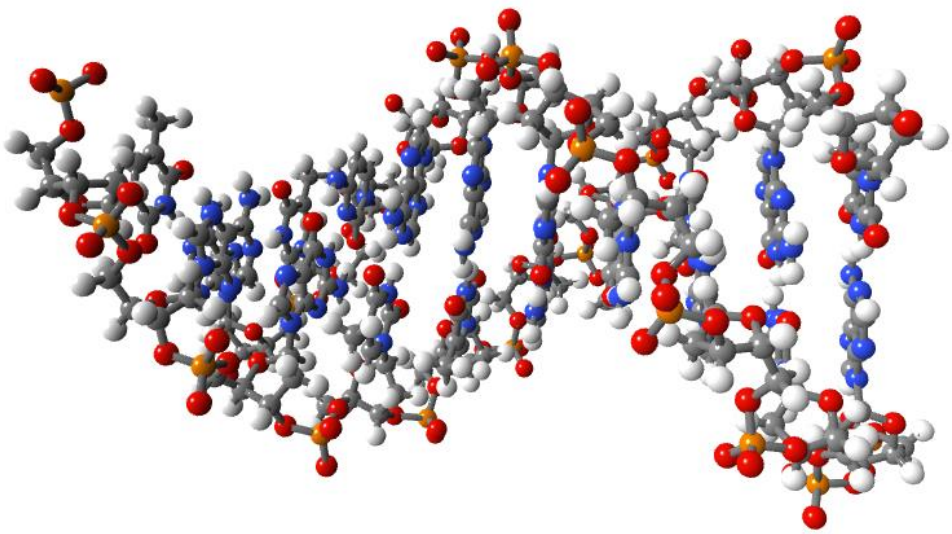


Figura 3.2 Resultado final luego de renderizar un objeto Molecule

CAPÍTULO 4

COMPARACIÓN DE ALGORITMOS DE SHADING

4.1 Algoritmos de Shading

Los shaders son programas que se ejecutan de forma paralela haciendo uso de la GPU para realizar operaciones en la geometría de forma eficiente. Una de estas operaciones es la de determinar el color final de cada pixel que se muestre en la pantalla, tomando en consideración todos los elementos presentes en una escena como las cámaras, las luces, la perspectiva y por supuesto la superficie del objeto.

Existe gran cantidad de algoritmos que se encargan de dar diferentes efectos a los objetos, los que tratan de aproximar una imagen real se llaman fotorealistas. Para este proyecto se escogieron los dos algoritmos de sombreado fotorealista más utilizados.

Estos algoritmos son conocidos como Gouraud Shading y Phong Shading o Per-Vertex Shading y Per-Fragment Shading, los cuales aplican los cálculos una vez por vértice y una vez por pixel respectivamente.

Para elegir el algoritmo adecuado para el proyecto, se busca contar con un resultado que no impacte en la eficiencia del renderizado ya que esto haría que la estructura propuesta también pierda eficiencia y adicionalmente este algoritmo debe renderizar con un grado de realismo suficiente.

4.2 Resultado visual

Para las pruebas fue necesario utilizar un modelo donde se pueda apreciar los efectos de los algoritmos de sombreado. El modelo que se utilizó está compuesto por 8694 vértices que conforman 17200 triángulos y posee algunas partes de su estructura con mucha densidad de vértices mientras otras partes tienen baja densidad de vértices. Esta diferencia se generó de forma intencional de modo que se pueda observar los resultados en diferentes tipos de superficies haciendo uso de un mismo modelo.

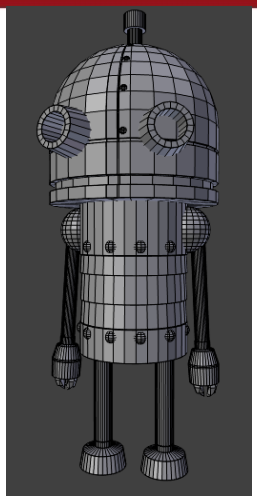


Figura 4.1 Modelo Original usado para la experimentación

Los brazos y piernas tienen mayor densidad de vértices

En la siguiente imagen se puede apreciar la diferencia de los resultados de los algoritmos en una zona con pocos vértices. En el caso del sombreado por vértice, la luz es evaluada por cada vértice y luego interpolada a través de los pixeles del triángulo, si no hay vértices en el punto donde la luz es aplicada el brillo no aparece como en el caso del torso.

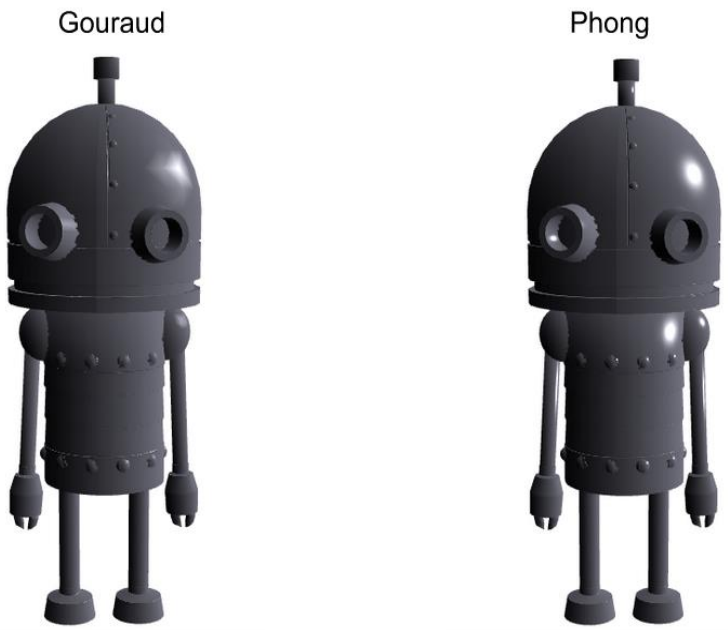


Figura 4.2 Comparación visual de los algoritmos de shading

Para evitar que ocurran casos como el mencionado anteriormente, es necesario que exista mayor cantidad de vértices en donde se pueda evaluar la interacción con la luz, en la siguiente imagen el brillo del brazo es bastante similar en ambos casos.

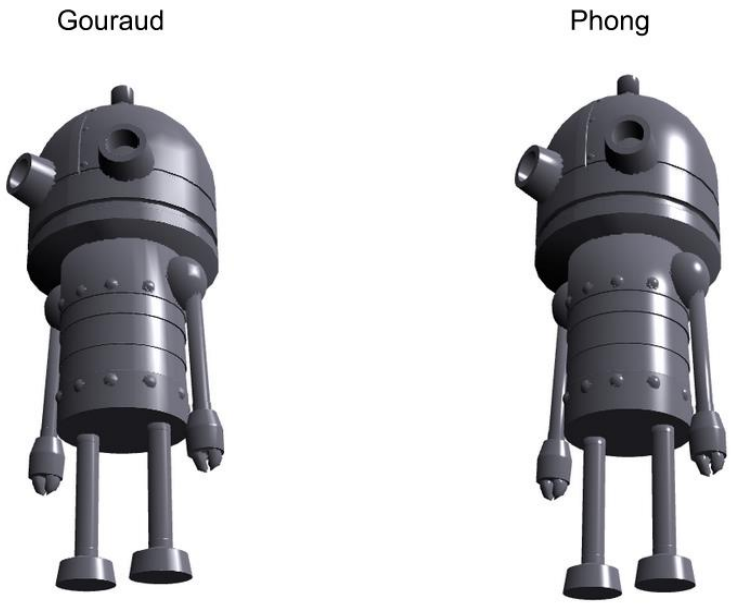


Figura 4.3 Comparación en otro ángulo de los dos algoritmos de shading

4.3 Evaluación de rendimiento

4.3.1 FPS y ms/frame

Para medir el rendimiento de una aplicación de rendering interactivo, un aspecto importante que se debe tener en cuenta es la fluidez de las animaciones, esto es que tan fluido se ve el movimiento de modo que no de sensación de lentitud y se genere un ambiente más inmersivo e interactivo.

Una medida que se usa para determinar la eficiencia del rendering son los fotogramas por segundo o FPS por sus siglas en inglés. Los FPS indican cuantas veces se generó una imagen por cada segundo de animación. A pesar que los FPS nos dan una idea de la velocidad total de la aplicación como un todo, se hace necesario utilizar otra unidad de medida para la eficiencia.

Otra medida que se utiliza es la inversa de los FPS, la cual es el tiempo que se demora en generar un fotograma. Usando esta medida, ms/frame, se puede determinar cuáles son las partes del programa que están demorando más

tiempo. Por ejemplo, si tenemos un tiempo de 33ms, podríamos determinar que renderizar cierto objeto demora 5ms del tiempo total.

4.3.2 Objetivo del experimento

El objetivo de este informe es comparar dos algoritmos de sombreado o shading: por vértice y por pixel para poder determinar cuál utilizar para el rendering sin afectar el rendimiento de forma considerable, teniendo en consideración también la apariencia estética final.

4.3.3 Diseño del experimento

La variable que se utilizará en el experimento será milisegundos por fotograma, esto se debe a que esta variable es más precisa que los FPS. Adicionalmente, al usar FPS también es necesario saber el número de fotogramas en los cuales se alcanzó los FPS dados haciendo más complejo el trabajar matemáticamente con ellos.

Para la experimentación se medirá el tiempo que toma la ejecución de un ciclo de rendering, esto incluye la reserva de memoria en la tarjeta gráfica, el cálculo de parámetros adicionales para el rendering y la llamada glDrawElements() la cual es la responsable de renderizar los objetos ejecutando los shaders en la tarjeta gráfica que es donde se encuentra la implementación de los algoritmos que se evaluaron.

Se eligió medir todo el ciclo de rendering en comparación con medir solamente la llamada glDrawElements ya que lo que nos interesa evaluar es el impacto que tienen los dos diferentes algoritmos en el tiempo total de un ciclo de rendering el cual es lo que finalmente determina la interactividad y fluidez de la animación.

4.3.4 Ejecución del experimento

El experimento se realiza para determinar si es que realmente existe una diferencia de rendimiento al usar un sombreado por píxel en vez de un sombreado por vértice.

La variable que analizaremos y será nuestra variable dependiente es la cantidad de milisegundos que toma generar un fotograma (ms/frame). Esta variable es dependiente del algoritmo que se utilice para shading.

Hipótesis: Hay diferencia en el tiempo de generación de un fotograma al usar el algoritmo de sombreado por vértice que al usar el algoritmo de sombreado por píxel.

$$H_0: \mu_v = \mu_p$$
$$H_0: \mu_v \neq \mu_p$$

Donde:

μv: Promedio de ms/frame usando sombreado por vértice.
μp: Promedio de ms/frame usando sombreado por píxel.

4.3.5 Resultados del experimento

Se tomaron 150 muestras de cada algoritmo para el análisis de los resultados.

	Nº de muestras	Promedio (ms)	Varianza(ms)
Phong (Per-pixel)	150	16.23	5.79
Gouraud (Per-vertex)	150	15.65	1.58

Para analizar la hipótesis de comparación de dos medias con varianza desconocida, primero debemos determinar si las muestras tienen distribución normal para luego poder escoger el test que se realizará para la comparación. Para esto se aplican los test de Kolmogorov-Smirnov y Shapiro-Wilk

Pruebas de normalidad

	Kolmogorov-Smirnova			Shapiro-Wilk		
	Estadístico	gl	Sig.	Estadístico	gl	Sig.

Phong	,192	150	,000	,822	150	,000
Gouraud	,178	150	,000	,938	150	,000

a. Corrección de la significación de Lilliefors

La pruebas de bondad de ajuste nos devuelven un valor de significancia igual a cero lo cual indica que las distribuciones no son normales y se deberá aplicar pruebas no paramétricas que no asuman la normalidad de las muestras, en este caso se aplica la prueba U de Mann-Whitney con nivel de significancia de 95%.

Rangos

	Tipo	N	Rango promedio	Suma de rangos
Tiempo	Phong	150	158,19	23729,00
	Gouraud	150	142,81	21421,00
	Total	300		

Estadísticos de contrastea

	Tiempo
U de Mann-Whitney	10096,000
W de Wilcoxon	21421,000
Z	-1,573
Sig. Asintót. (bilateral)	,116

a. Variable de agrupación: Tipo

El test nos arroja un valor de 0.116 el cual es mayor que 0.05 por lo cual se acepta la hipótesis nula que indica que no hay diferencia en el tiempo de generación de un fotograma si se usa un algoritmo de sombreado por vértice a un algoritmo de sombreado por pixel.

4.4 Conclusiones

Este análisis ha servido para determinar que ambos algoritmos son igualmente eficientes. Dado que el algoritmo de sombreado por píxel produce resultados más realistas y no afecta en la eficiencia del rendering, se ha elegido este algoritmo para la implementación de la estructura algorítmica.

CAPÍTULO 5

ESTRUCTURAALGORÍTMICA

5.1 Estructuras base

Para acelerar el renderizado de las moléculas, utilizaremos una estructura algorítmica, que realmente es una adaptación y combinación de dos estructuras ya existentes que han sido usadas en el campo de los gráficos por computadora: el scene graph y el octree. Una de las estructuras nos permite calcular la visibilidad de los objetos en la escena mientras la otra nos ayuda a organizar de manera jerárquica los objetos para una fácil manipulación.

5.1.1 Octree

El octree es una estructura de datos de tipo árbol, que se encarga de subdividir la escena o mundo 3D en ocho partes, inicialmente están suelen ser los ocho octantes del sistema cartesiano, luego cada una de estas partes en otras ocho y así sucesivamente hasta alcanzar un límite definido ya sea por el tamaño de la región, el número de subdivisiones o el número de elementos que están contenidos, entre otros.

Subdividir la escena en varios sectores nos ayuda a determinar algunos parámetros de estos como por ejemplo la visibilidad de un sector. Al recorrer el árbol por profundidad, se puede ir calculando si un sector es visible o no, de no serlo, se puede afirmar que todos sus subsectores tampoco son visibles, por lo que se ahorra gran cantidad de cálculos.

Cabe resaltar que si los elementos de la escena son estáticos, no hay necesidad de regenerar el árbol en ningún momento, a diferencia de tener una escena dinámica en la cual al cambiar un elemento de posición o de escala, puede ya no pertenecer al sector que estaba asignado, por lo que hay que calcular nuevamente su sector.

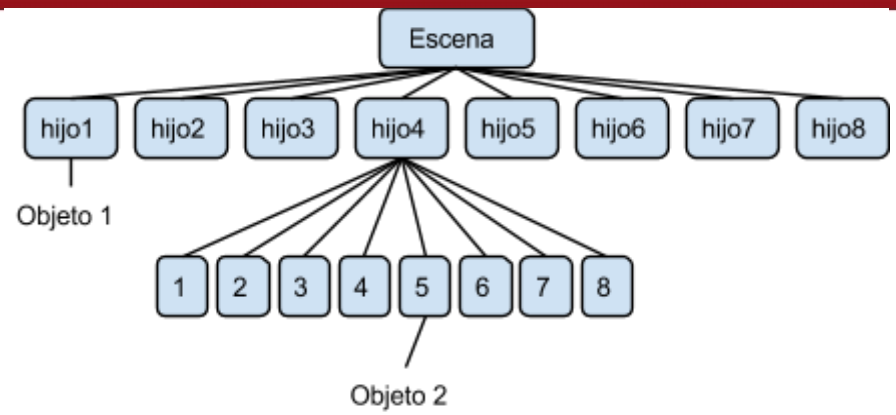


Figura 5.1 Representación del Octree

5.1.2 Scene Graph

El scene graph es un grafo estructurado en forma de árbol donde se encuentra representada la escena que se renderizará, tomando en cuenta todos los componentes como geometrías o conjuntos de vértices, materiales, iluminación y cámaras. Estos componentes están organizados en una jerarquía, donde existen elementos padre que condicionan la posición y orientación de sus elementos hijos, a su propia posición y orientación.

Una ventaja de utilizar un scene graph, es que esta provee mayor orden en la escena, pudiendo identificar los elementos que dependan de la posición de un elemento padre hace más fácil el poder manipular matemáticamente estos objetos. Adicionalmente, esta estructura permite la reutilización de recursos de una escena, las geometrías y materiales pueden pertenecer a más de un elemento del grafo.

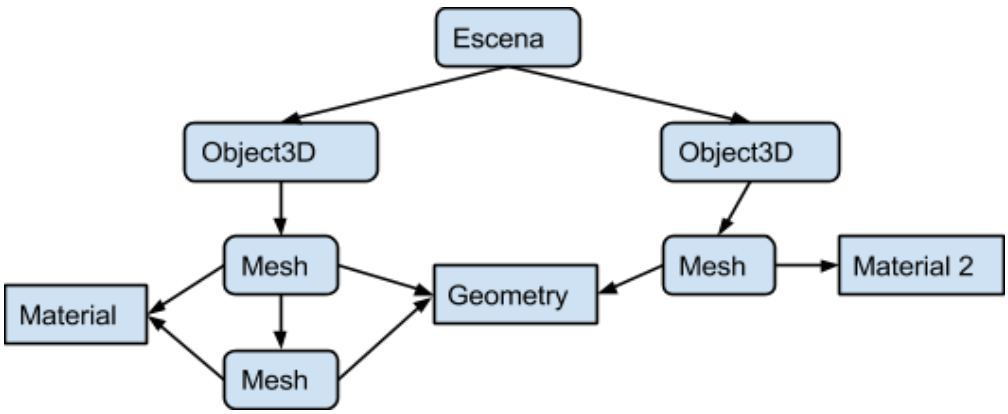


Figura 5.2 Representación del Scene Graph

5.2 Estructura propuesta

Para el proyecto se utilizarán ambas estructuras de datos ya que ofrecen distintas mejoras cada uno y se intentará aprovechar las ventajas de ambas.

5.2.1 Estructura de datos

Para el caso del octree, se comienza con definir la escena en ocho partes, luego uno a uno se ingresa los objetos en el octree. Al ingresar un objeto, se ubica en la partición en la cual encaje, en caso de encajar, se subdivide la partición si es que aún no se encuentra dividida y se analiza si el objeto encaja en alguna de las nuevas particiones hijo, así sucesivamente hasta que no se encaje completamente con alguna partición hijo.

```
insertarObjeto (objeto, arbol)
inicio
  si no(arbol subdividido):
    subdividir(arbol)
  fin si
  para cada hijo de arbol:
    si encaja(objeto, hijo):
      insertarObjeto(objeto, hijo)
      terminar
    fin si
  fin para
  añadir objeto a arbol. listaObjetos
fin
```

Para la generación del scene graph, se deberá contar con listas de todas las geometrías y materiales a ser usados en la escena y se debe agregar los objetos respetando la jerarquía de la escena, cada nodo tendrá una referencia al nodoOctree al cual está asignado.

Cada nodo del scene graph será un elemento de la clase Object3D. Una instancia de Object3D no necesariamente es un elemento visible o renderizable

de la escena, lo que la hace particular es que cuenta con atributos como posición, rotación y escala. Los atributos de Object3D se utilizan para poder establecer la jerarquía, los atributos de un hijo de este tipo de objeto tendrán como referencia inicial a los atributos de su padre.

Las instancias visibles son de la clase Mesh la cual hereda de Object3D pero contiene también como atributos, la geometría del objeto, el material y el nivel de detalles que se empleará para el renderizado las cuales son características que un objeto visible posee.

5.2.2 Recorrido de la estructura

Una vez generadas las estructuras tanto del scene graph como del octree, se debe indicar la secuencia que se utilizará para tomar toda esa data estructurada y utilizarla para el renderizado.

Se debe empezar por el octree, recorriendo por profundidad los nodos y determinando su visibilidad, la cual dependerá del límite que tenga el volumen de la pirámide trunca que representa la cámara, también conocido como frustum, así como también de la posición u orientación de esta en la escena. Al recorrer el árbol, también se calcula el valor de nivel de detalle de cada objeto, dependiendo de la distancia y el tamaño de este con respecto a la posición de la cámara.

```
recorrerOctree (nodo)
inicio
  si visible(nodo):
    para cada objeto en nodo:
      objeto es visible
    fin para
    para cada hijo en nodo:
      recorrerOctree(hijo)
    fin para
  fin si
fin
```

Una vez calculados estos valores, se procede a realizar el recorrido por profundidad del scene graph, actualizando los valores de las posiciones del hijo con respecto al padre. No hay necesidad de procesar los objetos no visibles que fueron calculados durante el recorrido del octree.

```
listaGeometrias ← ∅
listaMateriales ← ∅
recorrerScenegraph(nodo, matrizPadre = matrizIdentidad):
inicio
  matriz ← calcularMatriz(nodo, matrizIdentidad)
  añadir nodo.material a listaMateriales
  añadir nodo.geometria a listaGeometrias
  para cada hijo de nodo:
    recorrerScenegraph(hijo, matriz)
  fin para
fin
```

Para el caso de las geometrías y materiales, el scene graph contará con listas en las cuales se contará con estos elementos agrupados de modo que solo requiera una llamada a la tarjeta gráfica para movilizar estos elementos y una sola llamada de renderizado para todos los objetos.

5.2.3 Actualización de la estructura

El movimiento de la cámara de la escena afecta los objetos que están visibles, una vez que esta cambia de posición o rotación, sectores de la escena se vuelven visibles mientras otros sectores se vuelven invisibles. Para determinar cuáles de estos sectores han cambiado de visibilidad simplemente es necesario recorrer el octree.

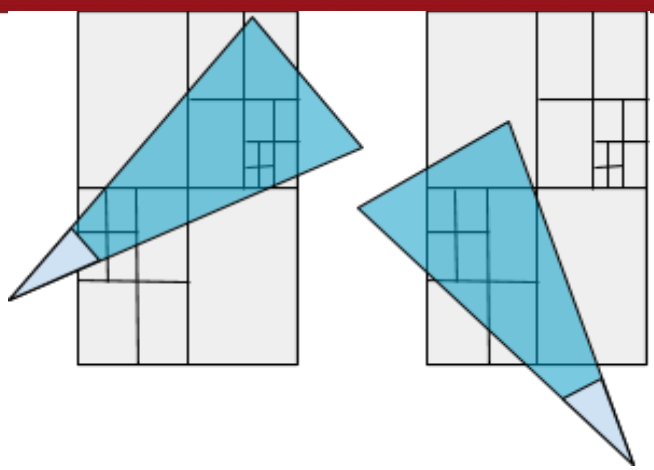


Figura 5.3 Efectos en la visibilidad por mover la cámara

Finalmente, el mover los objetos también requiere recalcular el octree, esto se debe a que una vez que el objeto se mueve, lo más probable es que ya no se encuentre en el mismo sector y sea necesario reubicarlo, pero no es necesario actualizar todo el árbol para lograr esto.

Para obtener el nuevo sector del objeto lo que se hace es primero evaluar si aún se encuentra en la misma partición, si es así evaluar si está contenido en alguno de las particiones hijas, sino probar si está contenido en la partición padre, este proceso se repite hasta que el objeto no esté contenido en ningún hijo en cuyo caso significa que el objeto pertenece a dicha partición.

```
RecalcularSector(objeto)
inicio
  particionActual = objeto.particionActual
  si encaja(objeto.particionActual):
    si no(particionActual subdividido):
      subdividir(particion actual)
    fin si
    para cada hijo en particionActual:
      si encaja(objeto, hijo):
        objeto.particionActual = hijo
      fin si
    fin para
```

```
    si no
      objeto.particionActual = objeto.particionActual.padre
    fin si
    si objeto.particionActual <> particionActual
      RecalcularSector(objeto)
    fin si
  fin
```

5.3 Resultados

Para visualizar el resultado de la aplicación de la estructura algorítmica y su recorrido en el proyecto, se agregaron cubos que ocupaban el volumen de cada sector subdividido del octree.

Los cubos fueron renderizados en modo wireframe, esto significa que sólo se grafican las aristas mientras que las caras de los polígonos son totalmente transparentes de modo que se pueda observar los objetos que cada nodo del octree contiene.

Para probar la funcionalidad del scene graph, se creó un Object3D que representaba una molécula completa, cuyos hijos eran los átomos y las conexiones entre ellos. Usando esta organización de los elementos se puede cambiar la posición de toda la molécula moviendo solo el objeto padre y no tener que mover cada átomo o conexión independientemente.

5.4 Conclusión

Las estructuras utilizadas apoyan en el proceso de renderizado, acelerando o simplificando algunos pasos importantes como la detección de visibilidad o el manejo jerárquico de la escena. La manera en que se recorren estas estructuras influye en la eficiencia en que se dibujan los objetos.

CAPÍTULO 6

OPTIMIZACIONES PARA LA ESTRUCTURA

6.1 Optimizaciones a la estructura

Una vez que se cuenta con la estructura algorítmica y la forma de recorrerla, aún se pueden realizar optimizaciones de renderizado. Estas optimizaciones deben aprovechar la estructura propuesta para poder realizar los cálculos necesarios sin tener que afectar la eficiencia.

Las optimizaciones realizadas no solo aprovecharon la estructura planteada, sino que también aprovechan las funcionalidades introducidas en las especificaciones nuevas de OpenGL a partir de la versión 4.0. Por ello es necesario contar con hardware moderno que soporte esta especificación, a la fecha que se realizó el proyecto, las tarjetas compatibles con esta especificación son a partir de las series GeForce 600 y las Radeon HD 5000 de los fabricantes Nvidia y AMD respectivamente.

6.1.1 Nivel de detalle dinámico

La cantidad de moléculas presente en la escena incrementa de forma significativa la cantidad de triángulos que deben ser procesados al momento de realizar el rendering. Una manera de reducir este número es contar con una geometría bastante básica, en este caso un icosaedro que es un poliedro conformado por veinte triángulos, y a partir de este sólido base ir subdividiendo para conseguir el nivel de detalle requerido.

El nivel de detalle requerido se determinará en función de la distancia en la que se encuentre un átomo al punto visual que es la cámara de la escena. Este valor se calculará al momento de recorrer el octree para determinar la visibilidad, durante ese recorrido se asignará a cada malla poligonal la distancia hacia la cámara. El límite inferior de subdivisiones será el icosaedro y el límite superior será 4 subdivisiones.

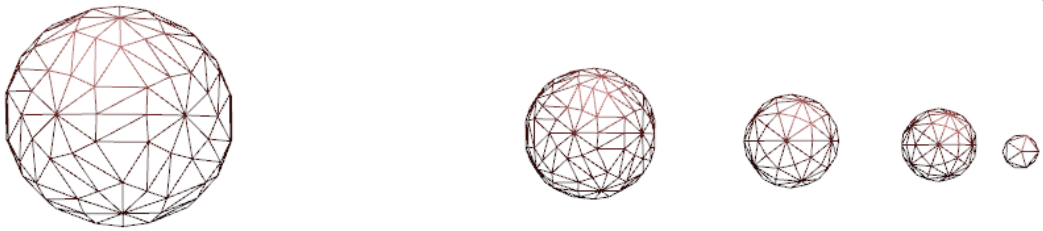


Figura 6.1 Múltiples subdivisiones de un mismo objeto dependiendo de la distancia

La subdivisión se implementó haciendo uso de los tessellation shaders, esto permite que se pueda realizar la subdivisión directamente desde el buffer donde se encuentran almacenados los puntos a través de la GPU.

En caso de no contar con el hardware requerido para soportar los tessellation shaders, existen dos alternativas para tener nivel de detalle dinámico. Por un lado la subdivisión se puede realizar con el CPU, esto resulta ineficiente ya que se tendría la necesidad de actualizar el contenido de los buffers, lo cual afectaría severamente el rendimiento.
Por otro lado, una alternativa más viable sería contar con múltiples versiones de una misma geometría guardadas en el buffer, de este modo no hay necesidad de realizar subdivisión alguna, solo habría que indicar cuál de las geometrías usar. La desventaja de esto es que se requiere mayor cantidad de memoria para almacenar las diferentes versiones de una geometría.

6.1.2 Renderizado múltiple

El renderizado de grandes cantidades de objetos también está limitado por el driver de la tarjeta gráfica. Puede que el CPU y la GPU estén en la capacidad de renderizar aún más pero el driver ya está en el límite de su capacidad lo que genera un cuello de botella para el rendimiento.

El driver se encuentra limitado ya que las llamadas a la API de OpenGL son costosas pues estas deben cumplir con la especificación y realizar las validaciones correspondientes. Ya que modificar el driver para volverlo más eficiente esta fuera del alcance de este proyecto, una solución a este problema es usar la API de manera eficiente intentando reducir el número de llamadas que se realicen a esta.

La versión 4.3 de la especificación OpenGL logra hacer más eficiente el uso de esta API utilizando llamadas que ejecuten el renderizado de múltiples objetos de forma simultánea. Con esto se ahorra el costo de tener que estar cambiando de parámetros cada vez que se necesite renderizar un objeto.

```
Crear buffers de geometría
Asignar datos a los buffers de geometría
Calcular data constante a toda la escena

Para cada objeto de la escena:
  Asignar buffer a usar
  Calcular data independiente de cada objeto
  Asignar data a los programas
  Especificar la estructura de la data
  Renderizar (draw)
Fin para
```

Bucle de renderizado clásico

La función glMultiDrawElementsIndirect() permite renderizar múltiples elementos con una sola llamada, los parámetros individuales de cada objeto son guardados en un buffer común para todos los objetos, de este modo se evita llamadas sucesivas a la API para establecer los parámetros de cada objeto.

```
Crear buffers (1 buffer -> toda la escena)
Asignar datos a los buffers
Calcular data constante a toda la escena
Asignar buffer a usar
Para cada objeto de la escena:
  Calcular data independiente de cada objeto
     Actualizar los buffers
Fin para
Renderizar (draw)
```

Bucle mejorado, solo tiene una llamada draw

La desventaja que trae usar esta nueva función es que todos los objetos que serán renderizados por esta función deben usar los mismos shaders ya que no hay posibilidad de especificar qué shader usará cada objeto.

Si no se cuenta con el hardware necesario que soporte la especificación 4.3 de OpenGL lamentablemente no se podrá usar esta función. Por otro lado, los parámetros de cada objeto aún pueden ir agrupados en un mismo buffer y así se reduce la cantidad de llamadas a la API ocasionadas por modificar los parámetros de renderizado de cada objeto.

6.1.3 Frustum culling paralelo

El octree proporciona la posibilidad de determinar qué elementos no son visibles para la cámara de modo que estos no sean procesados por las siguientes etapas del pipeline de renderizado. Dependiendo de la sección de la escena donde se encuentre el objeto, se determina su visibilidad, pero hay casos en los que esto no necesariamente aplica.

Existe la probabilidad de que una sección de la escena sea incluida debido a que sólo una pequeña porción de esta era visible ante la cámara lo cual ocasiona que todos los objetos de esa sección sean procesados, independientemente de si son visibles o no.

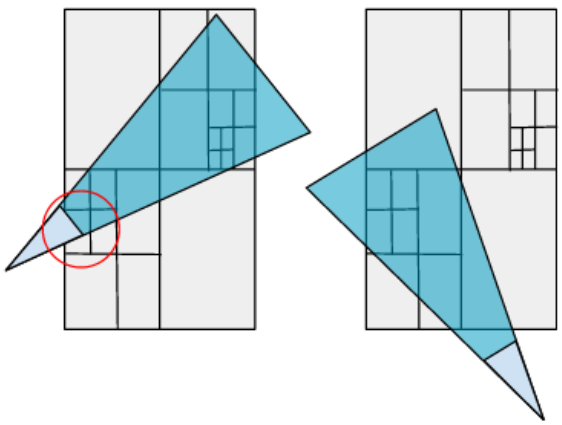


Figura 6.2 Visibilidad limitada de algunos sectores de la escena

Para reducir un poco los cálculos se puede descartar los vértices fuera de la pantalla de forma paralela haciendo uso del Geometry shader, este descarte no

puede realizarse en una etapa anterior ya que se está utilizando el nivel de detalle dinámico, el cual puede generar nuevos vértices que pueden o no estar dentro del campo de visibilidad.

En caso de no contar con hardware que soporte esta funcionalidad se puede implementar una solución alternativa, se puede subdividir el octree para que tenga más niveles o se puede convertir el octree en un icoseptree que consiste en subdividir la escena en 20 secciones. En ambos casos la complejidad de la estructura aumenta y debido a que la estructura se procesa en la CPU, esto puede afectar el rendimiento.

CAPÍTULO 7

EXPERIMENTACIÓN NUMÉRICA Y COMPARACIÓN DE RENDIMIENTO

7.1 Comparación de rendimiento

Una vez implementada la estructura e integrada con las optimizaciones propuestas, se debe evaluar si estas realmente tienen efecto en el rendimiento final de la aplicación, esto significa poder visualizar gran cantidad de moléculas de forma simultánea sin perder rendimiento, en comparación a usar el visualizador sin las mejoras implementadas.

7.2 Resultado visual

Para el experimento se utilizó una molécula sencilla compuesta por 40 objetos entre esferas y cilindros. Esta molécula fue duplicada 64 veces a través de la escena para conseguir una situación que lograra que la versión del visualizador sin la estructura comenzara a perder eficiencia.

En esta situación, la versión con la estructura implementada debería mostrar un mejor rendimiento y de este modo demostrar que en efecto cumple con su objetivo.

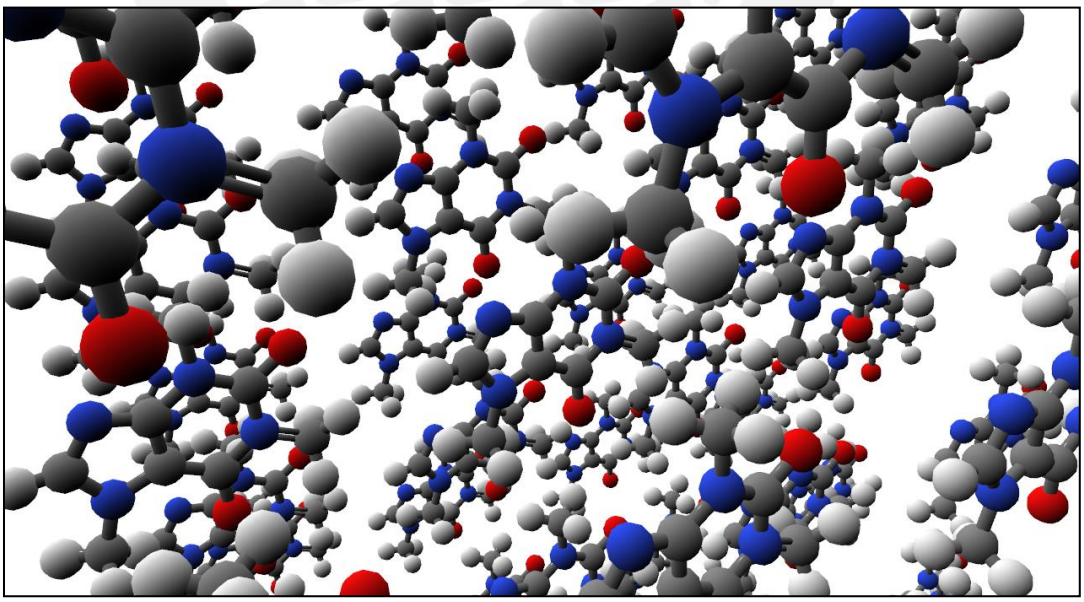


Figura 7.1 Escenario utilizado para la experimentación

7.3 Experimentación numérica

7.3.1 Objetivo del experimento

El objetivo de este informe es comparar el rendimiento de la nueva estructura implementada utilizando un visualizador con una versión de este que no implemente la estructura.

7.3.2 Diseño del experimento

La variable que se utilizará en el experimento será milisegundos por fotograma, luego esto podrá ser expresado como fotogramas por segundo para poder observar visualmente la mejora en el rendering.

El tiempo medido tendrá en consideración toda una iteración de dibujado, obteniendo el tiempo utilizado por cada etapa del pipeline de renderizado, incluyendo la etapa de aplicación, ya que una de las optimizaciones implementadas actúa en esta fase.

7.3.3 Ejecución del experimento

La variable que analizaremos y será nuestra variable dependiente es la cantidad de milisegundos que toma generar un fotograma (ms/frame). Esta variable es dependiente de si se está utilizando o no la estructura. Se tomará como valor de intervalo de confianza 95% en todas las pruebas.

Hipótesis nula: No hay diferencia entre el tiempo que toma generar un fotograma, utilizando la estructura propuesta, al tiempo requerido sin utilizar la estructura.

$$H_0: \mu_o = \mu_{so}$$
$$H_1: \mu_o < \mu_{so}$$

Donde

μo = promedio de tiempo de generación de un fotograma en milisegundos utilizando la estructura

μso = promedio de tiempo de generación de un fotograma en milisegundos sin utilizar la estructura

La prueba realizada es una prueba de una cola.

7.3.4 Resultados del experimento

Se tomaron 1100 muestras de cada una de los aplicativos y se obtuvieron los siguientes resultados:

	N	Media	Desv. Est.	Var	Min	Max
Sin la estructura	1100	20.56	4.64	21.55	16	40
Con la estructura	1100	12.71	2.78	7.75	10	32

Para poder comparar las medias entre sí primero debemos comprobar si las muestras siguen una distribución normal o no para saber qué tipo de prueba de hipótesis aplicar. Para ello aplicamos la prueba de Kolmogorov-Smirnov y Shapiro-Wilk a cada una de las muestras independientemente.

Pruebas de normalidad

	Kolmogorov-Smirnova			Shapiro-Wilk		
	Estadístico	gl	Sig.	Estadístico	gl	Sig.
SO	,316	1100	,000	,744	1100	,000
O	,294	1100	,000	,708	1100	,000

a. Corrección de la significación de Lilliefors

Los resultados de la prueba nos indican que ambas muestras no son normales dado que la significancia es igual a cero, por lo cual debemos aplicar una prueba no paramétrica que no asuma la normalidad de las muestras, en este caso usamos las prueba U de Mann-Whitney

Rangos

	versión	N	Rango promedio	Suma de rangos
time	SO	1100	1602,31	1762535,50
	O	1100	598,70	658564,50
	Total	2200		

Estadísticos de contrastea

	time
U de Mann-Whitney	53014,500
W de Wilcoxon	658564,500
Z	-37,470
Sig. Asintót. (bilateral)	,000

a. Variable de agrupación: versión

El resultado de la prueba arroja un valor de significación 0, al ser menor que 0.5 se rechaza la hipótesis nula y se acepta la hipótesis alternativa que indica que el tiempo de generación de un fotograma es menor utilizando la estructura implementada.

7.4 Conclusiones

Los resultados del experimento demuestran una evidente mejora en la eficiencia del visualizador al utilizar la estructura implementada, se obtienen mejoras de aproximadamente 20 FPS.

La mejora se debe a que la estructura y sus optimizaciones ayudan a disminuir el número de elementos a renderizar, así como también reducir las llamadas a la API de gráficos y evitar procesar escenas que no formarán parte de la imagen final.

CAPÍTULO 8

8.1 Conclusiones y recomendaciones

Las técnicas implementadas en este proyecto han sido posibles gracias al avance tecnológico de hardware y software para generación de gráficos. El trabajo de Dahlbom se vio limitado por el hardware de la época, motivo por el cual tuvo que descartar técnicas como el nivel de detalle dinámico acelerado por hardware o incluso el sombreado por pixel. Debido a que algunas de las técnicas no eran posibles ser paralelizadas a nivel de GPU, tuvo que optar por realizarlas de forma serial en el CPU lo que impactó la eficiencia.

Es posible integrar el scene graph con el octree, esto se logró a través de tener las estructuras enlazadas por medio de referencias desde los nodos de una hacia los nodos de la otra, de este modo se pudo escoger con mayor libertad cuál de las dos estructuras se recorre al momento de renderizar.

Si bien el octree nos provee de un método para recorrer los objetos en la escena, ignorando gran cantidad de ellos, esta posibilidad se tuvo que descartar ya que hubiera sido imposible realizar la optimización de renderizado múltiple ya que recorrer de esta manera el octree obliga a realizar varias llamadas a la API, lo cual puede impactar la eficiencia.

La característica que se conservó del octree fue la posibilidad que esta estructura brinda para determinar los objetos visibles a la cámara de modo que no se tenga que procesar los no visibles.

El uso de scene graph aportó también mejoras debido a la alta compatibilidad del escenario que es el renderizado de estructuras moleculares. Esto se debe a que los objetos a dibujar fueron repetitivos, principalmente se utilizó solamente dos geometrías que fueron compartidas por todos los objetos de la escena.

Se sacrificó la portabilidad de la aplicación al implementar las técnicas de optimización aplicadas a las estructuras, muchas de estas se encuentran actualmente disponibles en hardware moderno, e inclusive algunas no estaban disponibles en el hardware utilizado en los experimentos de este proyecto, por lo

cual se tuvo que recurrir a implementar una alternativa que devuelva el mismo resultado.

Se espera que en el futuro esta limitación se mejore y se pueda mantener la compatibilidad mediante los avances de la especificación de OpenGL y la disposición de los fabricantes de hardware a adoptar estas nuevas funcionalidades.

Como recomendación, se puede continuar el trabajo realizado en este proyecto adaptando la estructura a otros escenarios, de modo que no esté limitada a elementos repetitivos o similares sino a escenarios más complejas que demanden una gestión de los recursos de forma aún más eficiente y que pueden tomar como base la estructura propuesta.

Referencias bibliográficas

AKENINE-MOLLER, Tomas; MOLLER, Tomas; HAINES, Eric.
2002 Real-time rendering. AK Peters, Ltd.

SHIRLEY, Peter
2005 Fundamentals of computer graphics. Wellesley: AK Peters

AUTIN, L.; JOHNSON, G.; HAKE, J.; OLSON, A; SANNER, M.
2012 "uPy: A Ubiquitous CG Python API with Biological-Modelling Applications," Computer Graphics and Applications, IEEE, vol.32, no.5, pp.50, 61

CHEN, Shann-Ching; CHEN, Tsuhan.
2002 Retrieval of 3D protein structures. En Image Processing. 2002. Proceedings. 2002 International Conference on. IEEE, 2002. p. 933-936.

HAINES, Eric
2006 "An introductory tour of interactive rendering," Computer Graphics and Applications, IEEE, vol.26, no.1, pp.76, 87

GLASSNER, Andrew
1997 Situation normal [Gourand and Phong shading]. Computer Graphics and Applications, IEEE, vol. 17, no 2, p. 83-87.

SAYLE, Roger
1994 RasMol 2.5: Molecular Graphics Visualisation Tool. BioMolecular Structures Group, GlaxoResearchandDevelopment, Greenford, Middlesex, UK

HERRÁEZ, Angel
2006 Biomolecules in the computer: Jmol to the rescue. Biochemistry and Molecular Biology Education, vol. 34, no 4, p. 255-261

HOETZLEIN, Rama C

2012 Graphics Performance in Rich Internet Applications.Computer Graphics and Applications, IEEE, vol. 32, no 5, p. 98-104.

KARIM, M. S., KARIM, A. M., AHMED, E. & ROKONUZZAMAN, M

2003 Scene Graph Management for OpenGL Based 3D Graphics Engine. In Proc. International Conference on Computer & Information Technology (ICCIT) (Vol. 1, pp. p395-400)

DAHLBOM, M

2003 Efficient Molecular Visualization in Large Scale Systems Espoo, Finland

SEAGAL, Mark & AKELEY Kurt

2013 Kurt The OpenGL Graphics System: A Specification (Version 4.3 (Core Profile))

REINERS, Dirk

2002 Scene Graph Rendering