**UNIVERSIDAD AUTÓNOMA METROPOLITANA - IZTAPALAPA**

**DIVISIÓN DE CIENCIAS BÁSICAS E INGENIERÍA**

# VISUALIZACIÓN ÓPTIMA DE CAMPOS ESCALARES Y VECTORIALES UTILIZANDO GPUS

Tesis que presenta

**Apolinar Martinez Melchor**

Para obtener el grado de

**Maestro en ciencias y tecnologías de la información**

Directores de tesis

M. en C. Oscar Yañez Suárez

Dr. Jorge Garza Olguín

Tesis presentada el 5 de Marzo de 2015 ante los siguientes sinodales:

Presidente: Dr. Julio Manuel Hernández Pérez, BUAP

Secretario: M. en C. Oscar Yañez Suárez, UAM-I

Vocal: Dra. Graciela Román Alonso, UAM-I

Ciudad de México, D.F. Febrero de 2015

UNIVERSIDAD AUTÓNOMA METROPOLITANA - IZTAPALAPA

DIVISIÓN DE CIENCIAS BÁSICAS E INGENIERÍA

# VISUALIZACIÓN ÓPTIMA DE CAMPOS ESCALARES Y VECTORIALES UTILIZANDO GPUS

Tesis que presenta

**Apolinar Martinez Melchor**

Para obtener el grado de

**Maestro en ciencias y tecnologías de la información**

Directores de tesis

M. en C. Oscar Yañez Suárez

Dr. Jorge Garza Olguín

Tesis presentada el 5 de Marzo de 2015 ante los siguientes sinodales:

Presidente: Dr. Julio Manuel Hernández Pérez, BUAP

Secretario: M. en C. Oscar Yañez Suárez, UAM-I

Vocal: Dra. Graciela Román Alonso, UAM-I

Ciudad de México, D.F. Febrero de 2015

# Resumen

El presente trabajo continúa con la línea de investigación del proyecto 155070 de CONACYT cuyo título es "Uso de tarjetas gráficas para el análisis de la función de onda".

En el Área de Fisicoquímica Teórica se realizan estudios sobre la estructura electrónica, las propiedades físicas y químicas, la relación estructura-actividad y la reactividad de las especies químicas.En consecuencia, este trabajo de investigación se centró en la Visualización Óptima de Campos Escalares y Vectoriales utilizando GPUs; con el fin de proveer de una herramienta de *software*.

molS, es la herramienta *software* desarrollada, que permite realizar las tareas de calcular sistemas de moléculas sobre tarjetas gráficas y visualizar en tiempo real campos escalares.

# Abstract

The work presented in this document is entitled "Implementation/Use of graphics cards in weave function analysis". This research work is comprised within the research lines of Project 155070, a wider project registered under National Council for the Sciences and Technology (CONACYT, for its acronym in Spanish).

This research work revolved around the issues of developing and implementing Optimal Visualization of Scalar and Vector Fields through GPUs. The goal was to provide the Physicochemistry Department of The Autonomous Metropolitan University Campus Iztapalapa (UAM - I, for its acronym in Spanish) with a software tool for the analysis of molecular chemical bonds.

The aforementioned software tool has been developed and named "molS". This tool allows the user to operate graphics cards to calculate the geometry of molecular systems and to obtain real- time visualization of the molecular systems.

This document contains a description for the theoretical framework, the design and the development of "molS". Also, this document presents the process trough which "molS" was refined into an efficient software tool. Thus, this work contains the exploration on different possible schemes of graphics card's memory usage during the development and implementation of "molS". Finally, this document presents different example cases of ussage and aplication of "molS".

# Agradecimientos

A mis asesores el M. en C. Oscar Yañez y Dr. Jorge Garza que gracias a sus lecciones, consejos, confianza y tiempo dedicado me ayudaron a lograr la finalización de este trabajo de investigación.

Al Consejo Nacional de Ciencia y Tecnología (CONACyT) por financiar el presente proyecto de investigación, y a la Universidad Autónoma Metropolitana Unidad Iztapalapa por otorgarme la oportunidad de seguirme preparando.

Al Dr. Jorge Garza, un ejemplo a seguir, que gracias a sus consejos me han guiado durante toda mi formación académica.

# Índice general

Resumen . . . . . . . . . . iii
Abstract . . . . . . . . . . v

**Agradecimientos** **vii**

**1. Introducción** **1**
1.1. Motivación . . . . . . . . . . 2
1.2. Contribución . . . . . . . . . . 3
1.3. Objetivos . . . . . . . . . . 4
1.3.1. Objetivo general . . . . . . . . . . 4
1.3.2. Objetivos Específicos . . . . . . . . . . 4

**2. Visualización científica** **7**
2.1. Proceso de Visualización . . . . . . . . . . 8
2.2. Transformación de datos . . . . . . . . . . 11
2.2.1. Transformaciones basadas en las variables independientes . . . . . . . 12
2.2.2. Transformaciones a partir de las variables dependientes . . . . . . . . 13
2.2.3. Transformaciones basadas en todas las variables . . . . . . . . . . 13
2.3. Mapeo de datos . . . . . . . . . . 13
2.4. Despliegue de datos: bibliotecas para la visualización 3D . . . . . . . . . . 18
2.5. Campos escalares y vectoriales en la química cuántica . . . . . . . . . . 22
2.6. Visualización de superficie . . . . . . . . . . 25
2.7. El proceso de visualización . . . . . . . . . . 26
2.8. Unidades de procesamiento gráfico, GPU . . . . . . . . . . 29

2.8.1. Arquitectura Unificada de Dispositivos de Cómputo, CUDA . . . . . 30
2.8.2. Interoperabilidad de OpenGL y CUDA . . . . . . . . . . . . . . . . . . 33
2.8.3. OpenGL Shading Language . . . . . . . . . . . . . . . . . . . . . . . . 34
2.9. Aplicaciones para visualización en química cuantica . . . . . . . . . . . . . . 36

**3. Diseño del programa molS 39**
3.1. Ambiente molS . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 41
3.2. Arquitectura molS . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 41
3.3. Funcionamiento de molS . . . . . . . . . . . . . . . . . . . . . . . . . . . . 42
3.4. Descripción de componentes principales de molS . . . . . . . . . . . . . . . . 44
3.4.1. Componentes de la Capa 1: Procesamiento . . . . . . . . . . . . . . . 44
3.4.2. Implementación de algoritmo *marching cubes* . . . . . . . . . . . . . 45
3.4.3. Componentes de la Capa 2: Sistema molecular y Campo escalar . . . 48
3.4.4. Componentes de la Capa 3: Simulador . . . . . . . . . . . . . . . . . 50
3.4.5. Componentes de la Capa 4: Manejador de Ventanas . . . . . . . . . 50
3.5. Implementación de los componentes del ***software*** molS . . . . . . . . . . . . 51

**4. Resultados 61**
4.1. Acoplamiento de núcleo matemático y procesamiento visual . . . . . . . . . 61
4.2. Despliegue de campos escalares . . . . . . . . . . . . . . . . . . . . . . . . 63
4.3. Programación eficiente sobre los hilos de la tarjeta gráfica . . . . . . . . . . . 65

**5. Conclusiones y trabajo futuro 69**

**A. Tablas de referencía marching cubes 77**

**B. Visualización con molS 89**

**C. Interoperabilidad entre CUDA y OpenGL 95**

# Índice de figuras


1.1. Procesamiento y visualización de simulaciones . . . . . . . . . . . . . . . . . . 3

1.2. Procesamiento y visualización de simulaciones en tiempo real. . . . . . . . . 4

2.1. Abstración del cause de visualización (*visualization pipeline*) . . . . . . . . . 9

2.2. Proceso típico de visualización, donde la interacción proporciona los principales medios para reducir el espacio de búsqueda en la exploración visual. *Cdata*, *Cctrl* y *Cimage* denotan los datos de entrada, parámetros de control y visualización de resultados almacenados en la memoria del ordenador, respectivamente. Pinfo y Pknow representan la información y los conocimientos adquiridos por el usuario [5]. . . . . . . . . . . . . . . . . . . . . . . . . . 10

2.3. Visualización de información asistida, donde un cause adicional muestra información acerca de los datos de entrada reduciendo al usuario espacio de búsqueda en el principal proceso de visualización[5]. . . . . . . . . . . . . . . 12

2.4. Espacios cartesianos bidimensional y tridimensional[20]. . . . . . . . . . . . . 14

2.5. Transformaciones lineales. a)Traslación b)Rotación c)Escalamiento [20]. . . . 16

2.6. Ejemplo de translación en el espacio 3D. . . . . . . . . . . . . . . . . . . . . . 17

2.7. Representación de la rotación tomando como eje de giro $OY$ y matriz de rotación $R(y, \varphi)$. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 18

2.8. Representación de la rotación tomando como eje de giro $OX$ y matriz de rotación $R(x, \alpha)$. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 19

2.9. Representación de la rotación tomando como eje de giro $OZ$ y matriz de rotación $R(z, \theta)$. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 20

2.10. Cause GPU [16]. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 20

2.11. Isosuperficies dibujadas con OpenGL. . . . . . . . . . . . . . . . . . . . . . . . 22
2.12. Visualización de un campo escalar y geometría de una molécula de agua. . . 23
2.13. Casos base del algoritmo *marching cubes* (los vértices negros tienen un valor mayor al isovalor). . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 26
2.14. GPU Architecture. TPC: Texture/processor cluster; SM: Streaming Multiprocessor; SP: Streaming Processor. . . . . . . . . . . . . . . . . . . . . . . . . . 32
2.15. Rendimiento entre GPU y CPU. . . . . . . . . . . . . . . . . . . . . . . . . . . 33
2.16. Las Etapas *Vertex Shader* y *Fragment Shader* del cause de visualización suelen ser reemplazados. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 35

3.1. Herramientas, Bibliotecas y conceptos para la construcción del *software* para visualización molS. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 41
3.2. Arquitectura del *software* de visualización molS. . . . . . . . . . . . . . . . . . 43
3.3. Componente *SystemMolecular*. . . . . . . . . . . . . . . . . . . . . . . . . . . . 44
3.4. Vector de densidad electrónica . . . . . . . . . . . . . . . . . . . . . . . . . . . 44
3.5. Ilustración de los pasos que se siguen para generar la malla del visualizador molS . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 45
3.6. a) Clase ScalarField b) isosuperficie que describe la densidad electrónica. . . 45
3.7. Ejemplo del archivo molécula.cub utilizado para la visualidor molS. . . . . . 46
3.8. Diagrama de clases del visualizador molS. . . . . . . . . . . . . . . . . . . . . 47
3.9. Clases ScalarFieldCPU y ScalarFieldGPU que heredan de la clase ScalarField. 48
3.10. Numeración de los vértices y aristas de un *marching cube* . . . . . . . . . . . 49
3.11. Ejemplo de marching cube, caso 1. . . . . . . . . . . . . . . . . . . . . . . . . 49
3.12. Cálculo referente a un vértice de una faceta *marching cubes* . . . . . . . . . . 51
3.13. Ejemplo del archivo molecula.xyz utilizado para el visualizador molS . . . . 51
3.14. El arreglo que almacena la densidad electrónica es a) apuntado a los módulos externos de procesamiento b) cargado desde un archivo *.cub*. . . . . . . . . . 55
3.15. El arreglo *vectorScarlarField* en la GPU. . . . . . . . . . . . . . . . . . . . . 56
3.16. Arreglo de posicionamiento de facetas. . . . . . . . . . . . . . . . . . . . . . . 57
3.17. *kernel* de tamaño $dimBlock * dimGrid$ . . . . . . . . . . . . . . . . . . . . . 57

3.18. Cada *thread* procesa un *marching cube* y almacena sobre una posición de un arreglo. . . . . . 58

4.1. Representación de dos moléculas de agua sin conecciones entre los átomos . . 62
4.2. Representación de dos moléculas de agua con conecciones entre los átomos . 62
4.3. Gráfica molecular de una fracción de una proteína. . . . . . 63
4.4. Visualización de la densidad electrónica de la molécula $H_2$. . . . . . 64
4.5. Visualización del sistema molecular $H_2O$. . . . . . 66
4.6. Visualización del sistema molecular Cubano. . . . . . 67
4.7. Visualización del sistema molecular $\beta$-ciclodextrina. . . . . . 68

B.1. Geometrías moleculares . . . . . . 89
B.2. Densidad electrónica de una molécula de agua . . . . . . 90
B.3. Laplaciano de la molécula de agua . . . . . . 91
B.4. Densidad electrónica de una molécula benzeno . . . . . . 92
B.5. Densidad electrónica de una molécula cubano . . . . . . 93
B.6. Densidad electrónica de una molécula $\beta$-ciclodextrina . . . . . . 94

# Índice de tablas

2.1. Biblioteca gráficas . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 21
2.2. Algoritmo para visualizar una isosuperficie . . . . . . . . . . . . . . . . . . . 24
2.3. Algoritmo *marching cubes* . . . . . . . . . . . . . . . . . . . . . . . . . . . . 27
2.4. Directivas del preprocesador soportadas por GLSL . . . . . . . . . . . . . . . 36
2.5. Aspectos de las aplicaciones para la visualización de química cuántica . . . . 38

3.1. Casos: *marching cubes* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 50

4.1. Tiempos requeridos para generar una isosuperficie en tres sistemas moleculares 65

5.1. Aspectos de molS para la visualización de algunos campos de la QC . . . . . 70

# Capítulo 1

# Introducción

El presente trabajo de investigación se enmarcó en el ámbito de la Química Computacional (QC) que requiere de la simulación de estructuras de sistemas moleculares. Durante años, el área de investigación de Química Teórica ha realizado simulaciones computacionales en cúmulos de Unidades de Procesamiento (*Central Processing Unit, CPU*). Estas simulaciones funcionan como un "microscopio computacional", que permite a los investigadores analizar las estructuras de los sistemas moleculares. Así, la Química Computacional requiere de infraestructura de cómputo de alto rendimiento como los cúmulos de servidores conocidos como *Clusters*; con el fin de calcular, procesar y visualizar estructuras electrónicas de átomos y moléculas para el análisis de resultados. Sin embargo, las simulaciones realizadas en un cluster implican un alto costo y limitado acceso. En años recientes la industria de procesamiento gráfico y video juegos ha diversificado el uso de las Unidades de Procesamiento Gráfico (*Graphics Processing Unit, GPU*), que comumente eran utilizadas para procesamiento visual y actualmente se han empleado para procesamiento de datos. Las GPUs se han convertido en dispositivos masivos de cómputo en paralelo, y es posible programarlas en los lenguajes de programación como C/C++. La potencia de cálculo proporcionada por las GPUs dan lugar al uso de técnicas para el análisis que antes requerían de cálculos en la computación de alto rendimiento (*High Performance Computing, HPC*). Por consiguiente, las tarjetas gráficas han creado una gran oportunidad para emplear nuevas técnicas de simulación y análisis que antes eran demasiado exigentes computacionalmente.

## 1.1. Motivación

En el contexto de las aplicaciones de visualización, la química teórica es uno de los campos del conocimiento donde cotidianamente se hace uso de recursos de procesamiento númerico en un centro de supercómputo para generar campos escalares y vectoriales. Los campos generados posteriormente se analizan en una estación de trabajo con *software* de visualización. El proceso se divide en dos tareas:

1. Procesamiento de los datos: esta tarea se ejecuta en un cluster y tendrá una duración dependiendo del tamaño del sistema molecular, así como de las características de cómputo del mismo.

2. Visualización de los resultados: esta tarea recurre comúnmente al esquema (Ver Figura 1.1) de visualizar campos escalares, o vectoriales, en una estación de trabajo.

El equipo de químicos teóricos de la Universidad Autónoma Metropolitana cuenta con equipos de cómputo con tarjetas gráficas. El seguir los pasos de la Figura 1.1 para el procesamiento y visualización de simulaciones impide que se aprovechen las capacidades de las GPU embebidas. Así, en muchas ocasiones no se aprovecha la capacidad de la tarjeta gráfica para realizar operaciones de punto flotante, además de requerir la transferencia de datos del CPU a la tarjeta gráfica para generar el despliegue gráfico. Por consiguiente, una de las preocupaciones actuales del área de investigación de Química Teórica es contar con tarjetas gráficas en sus estaciones de trabajo -con la capacidad de efectuar operaciones de punto flotante- para explotar simultáneamente las capacidades de cálculo numérico y de visualización. Naturalmente, la memoria física de la tarjeta gráfica será una limitante y es ahí donde se tienen que explorar diversos esquemas del uso de memoria, lo que implica buscar una manera eficiente de transferir datos entre la tarjeta gráfica y el CPU. Por lo que se identifica la necesidad de contar con el *software* adecuado para explotar todas las facetas de una tarjeta gráfica de última generación.

Por lo tanto, este trabajo de investigación se enfocó en el diseño y desarrollo de una aplicación, llamada molS (por las siglas en inglés de molecular System), para visualizar en tiempo real campos escalares y vectoriales procesados en tarjetas gráficas.

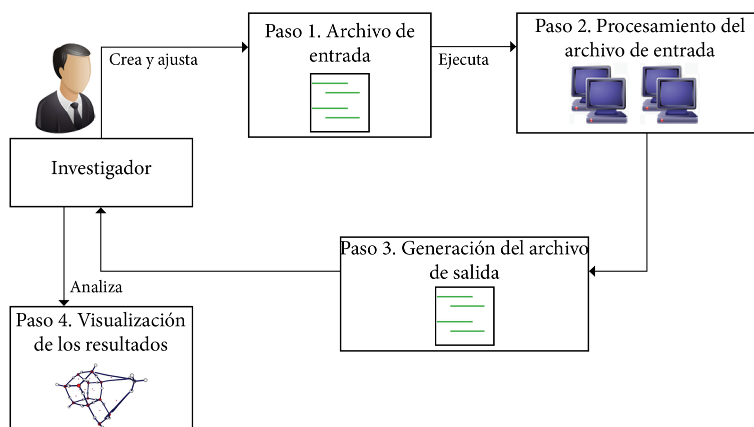



Figura 1.1: Procesamiento y visualización de simulaciones

## 1.2. Contribución

Este trabajo de investigación se centró en el desarrollo de la aplicación molS para satisfacer una necesidad del área de Fisicoquímica Teórica de visualizar y calcular simultáneamente -en tiempo real- campos escalares y/o vectoriales, sobrepuestos en las geometrías moleculares. La planeación de este trabajo permitió identificar los siguientes puntos como contribución al área de Química Teórica:

1. El desarrollo del *software* molS acelerará el procesamiento y visualización de campos escalares y vectoriales generados por la Química Teórica, mejorando el procesamiento de datos con la explotación de las capacidades de diseño en las GPUs de nueva generación, y aprovechamiento de la interoperabilidad entre la GPU y el lenguaje de programación OpenGL.

2. El investigador cambiará el procedimiento para visualizar sus simulaciones, el nuevo proceso de visualización requiere de un equipo de cómputo con una unidad de procesamiento gráfico y el *software* molS, la herramienta molS integrará los pasos de la Figura 1.2 para contribuir a un proceso ágil de visualización.

3. Se contará con una línea de productos *software* del Departamento de Química de la UAM Iztapalapa. El *software* molS se diseñará como una aplicación modular, flexible y escalar para que el código fuente pueda ser reutilizado y agregar funcionalidad o realizar

mantenimiento con un mínimo esfuerzo.

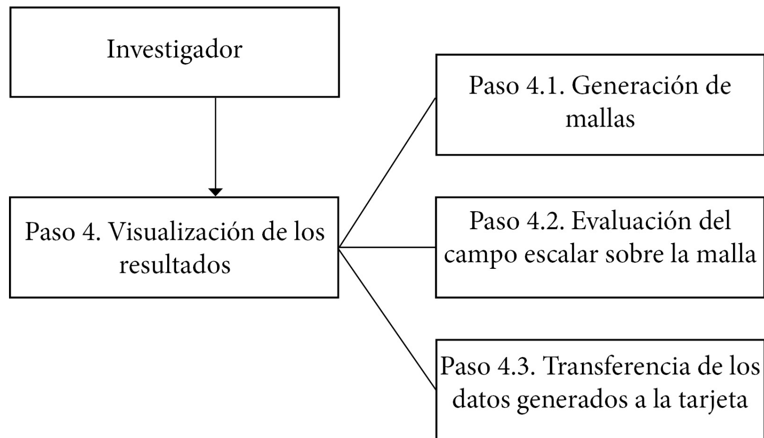



Figura 1.2: Procesamiento y visualización de simulaciones en tiempo real.

## 1.3. Objetivos

### 1.3.1. Objetivo general

**Generar una aplicación para visualizar en tiempo real campos escalares y vectoriales, de la química cuántica, procesados en tarjetas gráficas.**

### 1.3.2. Objetivos Específicos

1. Realizar una revisión de las fuentes de información relacionadas con el tema de investigación para establecer el contexto actual del tema.

2. Evaluar la densidad electrónica y sus derivadas de sistemas moleculares considerando mallas de orden medio.

3. Distribuir una malla tridimensional sobre los hilos contenidos en una tarjeta gráfica para garantizar el procesamiento uniforme.

4. Diseñar e implementar una herramienta *software* para la visualización de campos escalares y vectoriales.

5. Generar una interfaz de programación de aplicaciones para desplegar gráficamente campos escalares y vectoriales.

La estructura de la presente tesis se basa en cuatro capítulos; que son detallados a continuación.

Capítulo 1. *Introducción.* En este capítulo se explica de forma concisa el tema a desarrollar, por lo que se presentará en forma abreviada lo que se desea lograr con este trabajo, su motivación y la contribución; así como la justificación de porqué la necesidad de construir una aplicación que combina la visualización y el procesamiento numérico.

Capítulo 2. *Visualización Científica.* En este capítulo se presentan los antecedentes teóricos y las líneas actuales de investigación acerca del trabajo relacionado en esta área, y en particular de los visualizadores que existen.

Capítulo 3. *Diseño de molS.* En este capítulo se describe la arquitectura de la aplicación molS, y los módulos que integran la arquitectura. Además se detallan las decisiones de diseño tomadas durante el desarrollo de la arquitectura, y se analizan los módulos principales que caracterizan a la aplicación molS.

Capítulo 4. *Resultados.* En este capítulo se describe tres casos de estudio, que buscan recolectar las experiencias y resultados de los usuarios de molS. Las evaluaciones se orientan a comparar el procesamiento para la visualización utilizando el CPU y la GPU. Además de mostrar gráficas comparando la velocidad del visualizador con diferentes sistemas moleculares, utilizando como variables la cantidad de átomos, la distancia entre los puntos de evaluación sobre la malla, también se muestran resultados para comparar el rendimiento utilizando las diversas memorias que componen a la GPU.

# Capítulo 2

# Visualización científica

La visualización se emplea normalmente como un mecanismo de observación, para ayudar a los investigadores con las comparaciones intuitivas y para mejorar la comprensión de los datos estudiados[24]. La visualización científica se centra principalmente en los datos físicos, tales como los asociados al cuerpo humano, la tierra, las moléculas, etc. Por consiguiente, la visualización científica es el proceso de transformar los datos generados por un modelo matemático, en una forma visual, y es un elemento clave de la ciencia computacional [5, 34]. La visualización científica mejora la percepción de características y patrones en los datos, facilita la navegación a través de la interacción con conjuntos complejos y dispares de datos y mejora la comunicación de los resultados científicos[34]. Por ende, este tipo de visualización usa la forma, la posición en el espacio, el color, la brillantez y el movimiento para ayudar a los investigadores a comprender relaciones que de otra manera podrían pasar inadvertidas. Así, las imágenes resultantes de la visualización son parte de la investigación misma. La visualización científica propone facilitar la interpretación visual de los datos y define las siguientes dos áreas específicas: la visualización de volúmenes y la visualización de flujo [31].

1. La visualización de volúmenes: Se refiere generalmente a campos escalares; esta área se extiende desde el análisis de datos científicos a la reconstrucción de datos dispersos y a la representación de objetos geométricos sin una descripción matemática de su superficie. Permite el análisis del interior de un volumen utilizando técnicas como las de slicing y transparencias.

2. La visualización de flujo: Este campo se utiliza para la visualización en general de sistemas dinámicos en los que hay datos involucrados que evolucionan con el tiempo. El comportamiento cualitativo de dichos sistemas puede comprenderse adecuadamente a partir de la estructura de la evolución temporal de sus trayectorias. Estos tienen implícitamente una gran cantidad de datos que no es directa ni fácilmente observable. Ejemplos de visualización de flujo lo constituyen los campos vectoriales y tensoriales de distintos órdenes.

## 2.1. Proceso de Visualización

El proceso de visualización científica es fundamentalmente el mismo que un proceso de búsqueda normal. En la visualización científica, las herramientas para las tareas de "búsqueda" suelen ser de aplicación específica (por ejemplo, la red, el flujo, el volumen de visualización). Mientras, que el espacio de los parámetros de la "búsqueda" es normalmente enorme (por ejemplo, la exploración de muchas posiciones de visualización o el probar muchas funciones de transferencia). Este proceso se describe a menudo por medio de un cause de visualización (*visualization pipeline*). El objetivo del cause de visualización es servir de guía para estructurar la gran variedad de métodos de visualización [5]; por consiguiente puede modificarse de acuerdo al método de visualizacion. En la Figura 2.1 se presentan los pasos esenciales del cause de visualización. La entrada del cause de visualización se adquiere a partir de una fuente de datos, tal como una simulación numérica, una medición de datos físicos, o una base de datos. Estos datos deben ser transformados hasta lograr la representación visual de los mismos. Los pasos que conforman el cause de visualización son descritos a continuación:

1. Análisis de datos: Es la etapa de transformación de datos (En la Figura 2.2 véase $Cdata$) que convierte los datos en una forma más adecuada para la visualización, por ejemplo, interpolaciones para datos faltantes, correcciones de mediciones erróneas, etc. Esto puede implicar el remuestreo, los cambios de tipos de datos, creación de subconjuntos, y la derivación de nuevas cantidades.

2. Filtrado: En esta etapa, de acuerdo a los objetivos que se buscan con la visualización, se hace una selección de datos a ser visualizados (véase Figura 2.1).

3. Mapeo: Los datos seleccionados, son mapeados a geometrías primitivas (puntos, líneas, etc.), así como sus atributos (color, posición, tamaño). El mapeo representa una o más variables sobre el conjunto o un subconjunto del dominio (véase Figura 2.1).

4. Render: Esta es la última etapa donde finalmente los datos geométricos son transformados en imágenes. Se obtiene una colección satisfactoria de los resultados de visualización (En la Figura 2.2 véase $Cimage$).

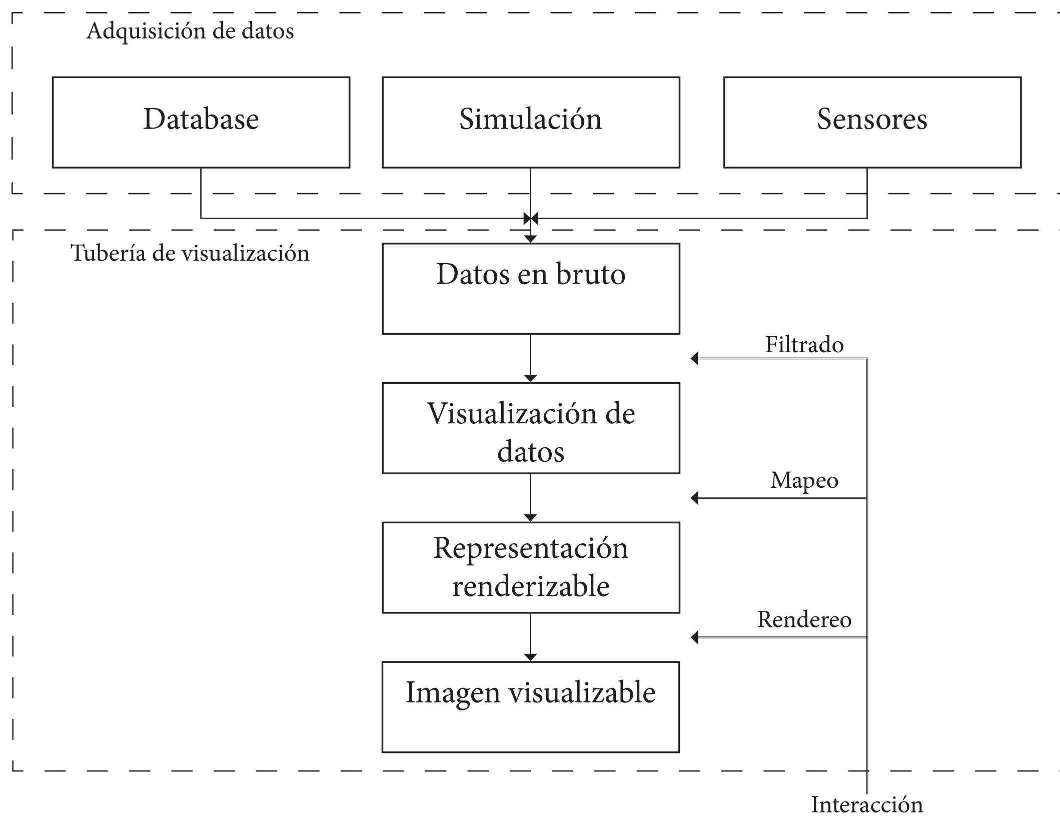



Figura 2.1: Abstración del cause de visualización (*visualization pipeline*)

El proceso de la visualización científica es inherentemente iterativo (ver Figura 2.2). Un usuario puede interactuar con las tres etapas del cause de visualización. Por otra parte, la interacción juega un papel crucial debido a que el espacio de parámetros sólo puede ser explorado de forma interactiva. Una buena interpretación viene de experimentar con los parámetros de visualización y representación de los datos más relevantes. Así un requisito previo será una técnica de visualización eficiente para lograr la aplicación en tiempo real.

Cuanto más eficiente sea la aplicación, menor será el número de iteraciones necesarias para la selección de parámetros [5].

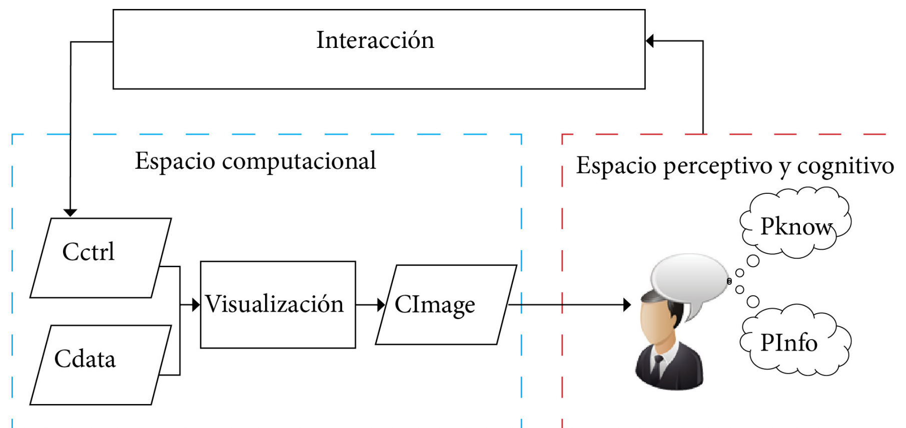



Figura 2.2: Proceso típico de visualización, donde la interacción proporciona los principales medios para reducir el espacio de búsqueda en la exploración visual. *Cdata*, *Cctrl* y *Cimage* denotan los datos de entrada, parámetros de control y visualización de resultados almacenados en la memoria del ordenador, respectivamente. Pinfo y Pknow representan la información y los conocimientos adquiridos por el usuario [5].

Un punto importante a mencionar del proceso de visualización, es que al ser un procesamiento lineal, cada paso depende del resultado del paso anterior, y al ser volumenes de información muy grandes, que pueden provenir de diversas fuentes, el tiempo de inicio a fin puede ser muy grande y el costo computacional muy alto, por lo que la visualización demanda constantemente técnicas y metodologías más rápidas y eficientes. En este sentido, en las últimas dos décadas se ha buscado mejorar el desempeño (*performance*) de las herramientas de visualización para que los usuarios puedan realizar *búsquedas* interactivas de manera eficiente y explorar espacios de parámetros más grandes. Sin embargo, con la creciente cantidad de datos y el aumento de la disponibilidad de las distintas técnicas de visualizacion, el espacio de búsqueda de un proceso de visualización también se expande. Para algunos algoritmos de visualización, puede ser difícil encontrar una asignación clara y única de elementos algorítmicos de iteración, mapeo, o de rendereo. En particular los métodos basados en GPU, a menudo requieren una reestructuración del cause de visualización para lograr una aplicación eficiente. El proceso de transformación de los datos numéricos en una o más imágenes involucra las

siguientes áreas: artes gráficas, interacción humano-computadora y ciencias cognitivas [22], involucrando a investigadores e ingenieros especializados en modelado, *software* y *hardware*. Para cumplir con estos objetivos, la visualización científica integra técnicas concernientes a la presentación de datos científicos; empleando un proceso comprensible y reproducible [22]. Los datos de entrada en el proceso de visualización son importantes para lograr una representación exitosa. En la Figura 2.3 se considera una segundo cause de visualización que típicamente muestra la información sobre el conjunto de datos de entrada.

El usuario utiliza dicha información para reducir el espacio de búsqueda para los parámetros de control óptimos; esto da lugar al proceso de visualización asistida. Estas técnicas utilizan la información capturada en el proceso de visualización para mejorar la eficiencia y la eficacia de visualización. En la visualización asistida, el sistema proporciona al usuario un segundo subproceso de visualización (ver Figura 2.3), que típicamente muestra la información sobre el conjunto de datos de entrada. El usuario utiliza dicha información para reducir el espacio de búsqueda de los parámetros de control óptimos, por lo que la interacción es mucho más amigable. Los investigadores han introducido una variedad de técnicas para la visualización de características complejas de datos basándose en la información extraída de los datos. En este proyecto de investigación se considera vital el uso del cause agregado en la Figura 2.3 para mejorar la interacción del usuario con la herramienta.

## 2.2. Transformación de datos

Los métodos de visualización están organizados de acuerdo al tipo de datos, a partir de campos escalares o campos vectoriales. Un conjunto de datos multidimensional ${L^n}_m$ consiste de m variables independientes que representan el dominio de datos y n variables dependientes definidas sobre el dominio. En la mayoría de las aplicaciones de variables independientes se define un dominio espacial bidimensional o tridimensional, y el conjunto de datos se llama simplemente datos 2D y 3D, respectivamente [34]. Una variable independiente se puede introducir para considerar el tiempo. Ambas variables dependientes e independientes pueden ser discretas o continuas y pueden tener un número finito o un intervalo infinito de valores. Los ejemplos más comunes de las variables independientes son campos escalares (n = 1) de

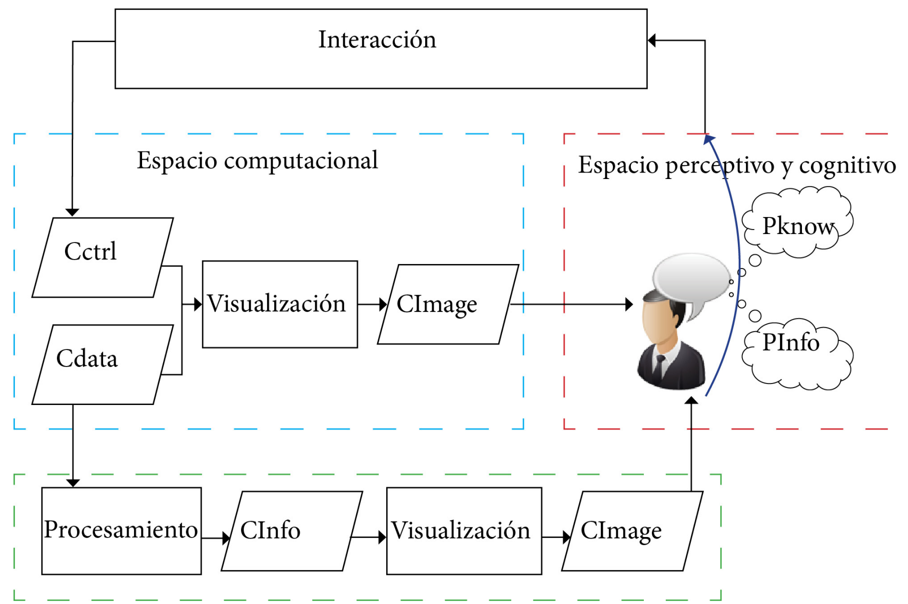



Figura 2.3: Visualización de información asistida, donde un cause adicional muestra información acerca de los datos de entrada reduciendo al usuario espacio de búsqueda en el principal proceso de visualización[5].

temperatura, campos vectoriales (n = 3) como la velocidad y campos de tensores simétricos (n = 6) tal como stress. Muchos conjuntos de datos científicos consisten en múltiples campos definidos en el mismo dominio que resulta en un espacio de alto orden de las variables dependientes. Los conjuntos de datos son muy distintos y se clasifican de acuerdo a su distribución espacial. De esta manera, en la visualización se trabaja con datos de tipo escalares, vectores, tensores y datos con variables múltiples. Con el fin de asignar los datos a atributos visuales es necesario convertir los datos primero en una forma más adecuada. Esto se logra mediante el uso de una transformación de datos que modifica las variables independientes, dependientes o ambas.

### 2.2.1. Transformaciones basadas en las variables independientes

En general, las variables independientes de un conjunto de datos científicos definen el espacio y el dominio temporal. Este tipo de transformación a menudo se llama selección subespacial y es un ejemplo de reducción de datos. La transformación de los datos considerando la variable independiente suele mejorar la percepción de los datos establecidos por el

descubrimiento de las características ocultas y, por tanto, facilita la exploración del conjunto de datos. Sin embargo, la comprensión del conjunto de datos no puede ser mejorada puesto que la información estructural contenida en las variables independientes no se extrae.

### 2.2.2. Transformaciones a partir de las variables dependientes

Las transformaciones de las variables dependientes pueden ser representados como operadores matemáticos. Las técnicas se clasifican en la reducción de datos, expansión de datos, o algoritmos de modificación de datos, dependiendo de si la dimensión de la variable dependiente reduce, aumenta o permanece sin cambios.

### 2.2.3. Transformaciones basadas en todas las variables

Los conjuntos de datos científicos, que son dados por los valores de muestras discretas, deben ser transformados en datos continuos antes de la transformación. Este tipo de transformación de datos es un ejemplo de enriquecimiento de datos y se puede lograr mediante el uso de técnicas de interpolación, como la interpolación de datos dispersos o la interpolación de elementos finitos.

## 2.3. Mapeo de datos

Para generar una visualización, es necesario un mapeo de los datos en el espacio cartesiano de dos o tres dimensiones, que represente las relaciones contenidas en los mismos de manera tan intuitiva como sea posible. Los pixeles en una ventana de dos dimensiones se describen utilizando coordenadas cartesianas: un punto X (horizontal) y un punto Y (vertical) (véase Figura 2.4). La visualización 3D consiste en una ilusión lograda a partir de intensos cálculos matemáticos que proyectan objetos en tres dimensiones sobre una pantalla, en la cual sólo pueden percibirse dos dimensiones, ancho y altura. La profundidad no existe en la pantalla, es la dimensión que debe simularse al graficar objetos 3D. Para simular esta tercera dimensión, se agregan efectos visuales y se proyectan los objetos 3D sobre 2D a través del proceso conocido como rendering. En el espacio de tres dimensiones se considera un tercer eje, conocido como

el eje Z, y se refiere a la profundidad de un punto determinado. Así, el uso del eje Z creará la ilusión del espacio tridimensional en la pantalla plana de la computadora. Si bien la idea de una tercera dimensión en un monitor plano de la computadora puede parecer imaginaria, es muy real para el procesamiento. El procesamiento considera la perspectiva, y selecciona los atributos de las primitivas adecuadas con el fin de crear el efecto tridimensional. Para especificar puntos en tres dimensiones, las coordenadas se especifican en el orden: X, Y, Z. Tan pronto entramos en el mundo de las coordenadas 3D, una cierta cantidad de control debe ser abandonado al proceso de transformación. No se controlan las ubicaciones exactas del píxel con las coordenadas XY, porque la ubicación XY se ajustará para tener en cuenta la perspectiva 3D.

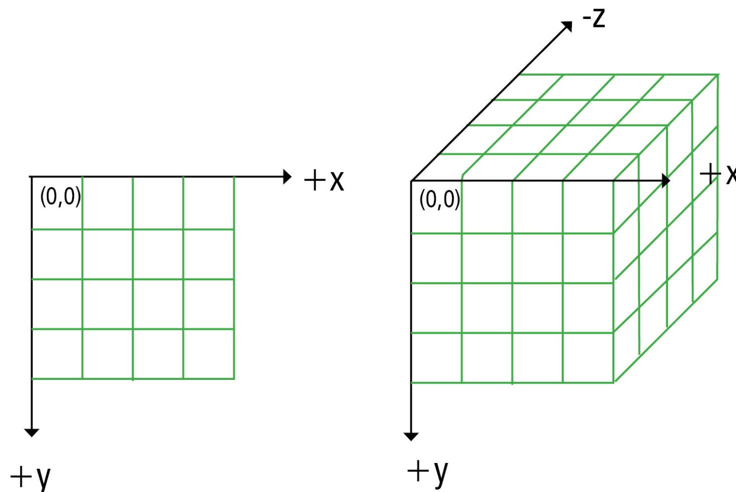



Figura 2.4: Espacios cartesianos bidimensional y tridimensional[20].

Imaginemos que estamos en una habitación con una cámara en las manos y comenzamos a tomar fotos. Cada uno de los objetos tiene una forma y geometría que se describe en un sistema de coordenadas local, que es único para cada objeto, se centra en el objeto, y no depende de ningún otro objeto. Tienen una posición y orientación con respecto al observador. La imagen final de cada objeto será el producto de la posición del observador, del lente de su cámara y los ajustes de la cámara que interactúan para mapear características tridimensionales[20], por lo que debemos considerar los siguientes conceptos:

1. Espacio de coordenadas locales: Espacio geométrico tridimensional con respecto al cual

se tiene la representación geométrica de un objeto.

2. Espacio de coordenadas del mundo: Espacio geométrico tridimensional con respecto al cual se tiene la representación geométrica de una escena.

3. Espacio de visualización: Espacio geométrico tridimensional que contiene al plano de visualización y en el que se realiza la proyección geométrica.

4. Punto de referencia visual o centro de proyección: Origen del espacio de visualización y el lugar geométrico donde se sitúa la lente de una cámara o el ojo de un observador.

5. Ventana de visualización: Es la región del plano de visualización sobre la que se realiza la proyección.

6. Volumen de visualización: Es la parte del espacio de coordenadas del mundo que intervienen en el proceso de visualización.

Ahora bien, la variación de la posición y/o el tamaño de los objetos, con respecto a los sistemas de referencia, se hace mediante transformaciones lineales. Las transformaciones permitirán mover, girar y manipular objetos en un mundo en 3D (véase Figura 2.5), además de permitir proyectar coordenadas 3D en una pantalla 2D. Aunque las transformaciones parecen modificar un objeto directamente, en realidad, son simplemente transformaciones del sistema de coordenadas local del objeto en otro sistema de coordenadas. En el espacio 3D se pueden realizar las siguientes transformaciones lineales:

- Translación: La traslación de un objeto consiste en moverse cierta distancia, en una dirección determinada. En 2D, se efectúa la traslación de un objeto para cambiar su posición a lo largo de una línea recta. Un punto bidimensional se convierte al agregar las distancias de traslación $t_x$ y $t_y$ a la posición de coordenadas original $(x, y)$, quedando $x' = x + t_x$ y $y' = y + t_y$. El par $(t_x, t_y)$ es conocido como vector de traslación. La traslación en 3D es una extensión de esta transformada en 2D. Utilizando la Figura 2.6, se supone un sistema de coordenadas $O'UVW$ que se encuentra trasladado con

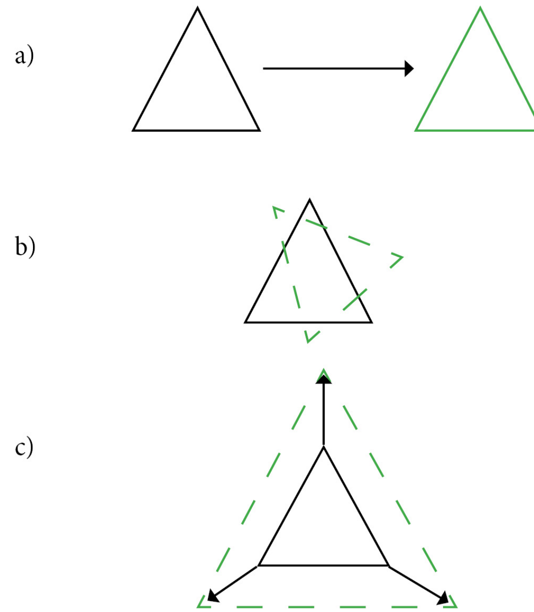



Figura 2.5: Transformaciones lineales. a)Traslación b)Rotación c)Escalamiento [20].

respecto al sistema $OXYZ$, $\vec{p} = p_x\vec{i} + p_y\vec{j} + p_z\vec{k}$. La matriz de traslación está dada por:

$$T(p) = \begin{bmatrix} 1 & 0 & 0 & p_x \\ 0 & 1 & 0 & p_y \\ 0 & 0 & 1 & p_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

El punto referenciado con respecto al eje $OXYZ$ es: $r_{xyz} = T(p)r_{uvw}$.

El punto referenciado con respecto al eje $O'UVW$ es:

$r_{uvw} = [r_u, r_v, r_w]^T = r_u i_u + r_v j_v + r_w j_w$

- Rotación: Además de la posición es necesario definir la orientación con respecto al sistema de referencia, por lo que surge la representación de la orientación. En el mundo físico, la rotación es un concepto bastante sencillo e intuitivo. Si se toma el timón de un barco y lo giramos, se tiene una idea de lo que significa rotar un objeto. Pero la rotación en programación, por desgracia, no es tan simple. Surgen preguntas del tipo. ¿Alrededor de qué eje debe girar? ¿A qué ángulo se debe girar? ¿Alrededor de qué punto de origen? Para una rotación en 2D, se aplica la transformación de rotación bidimensional cuando se cambia la posición de un objeto a lo largo de una trayectoria de una circunferencia

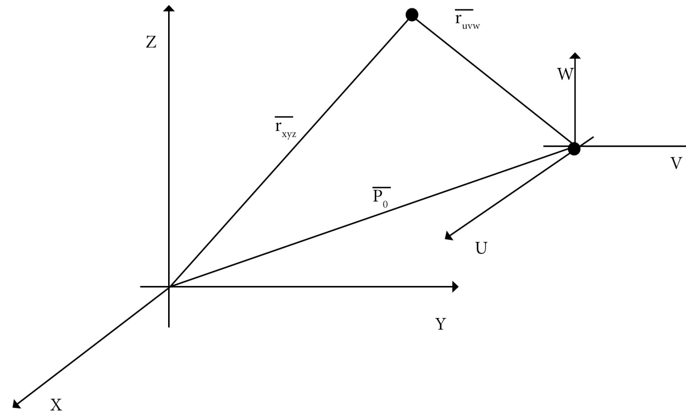



Figura 2.6: Ejemplo de translación en el espacio 3D.

en el plano $xy$. Se especifica el ángulo de rotación $\theta$ y la posición $(x_r, y_r)$ del punto de rotación (pivote). Para realizar el giro de los objetos en 3D, el usuario ha de establecer un eje de rotación, así como el ángulo y el sentido de giro alrededor de dicho eje. Por otro lado, los giros en 3D normalmente se realizan aprovechando la base trigonométrica de las rotaciones en 2D, es decir, descomponiendo los giros 3D en sus componentes ortogonales. Por esta razón, primero recordaremos cómo se realizan los giros en 2D. $x' = xcos\theta - ysen\theta$, $y' = xsen\theta + ycos\theta$

$$r = \begin{bmatrix} cos\theta & -sen\theta \\ sen\theta & cos\theta \end{bmatrix}$$

$P' = r \bullet P$

Las siguientes matrices de rotación realizan rotaciones de vectores alrededor de los ejes x, y, o z, en el espacio de tres dimensiones:

- Si se hace una rotación tomando como eje de giro al $OY$, la matriz de rotación $R(y, \varphi)$ se muestra en la Figura 2.7.
- Si se hace una rotación tomando como eje de giro al $OX$, la matriz de rotación $R(x, \alpha)$ se muestra en la Figura 2.8.
- Si se hace una rotación tomando como eje de giro al $OZ$, la matriz de rotación $R(z, \theta)$ se muestra en la Figura 2.9.

Cada una de estas tres rotaciones básicas se realiza en sentido antihorario alrededor del

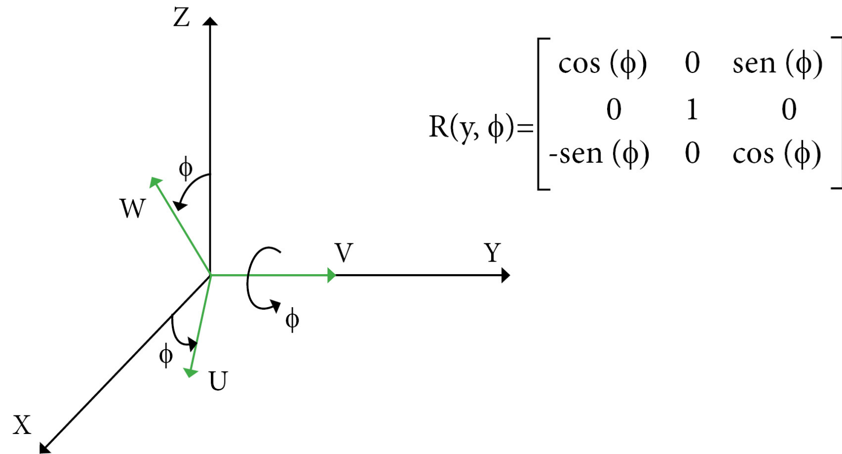



Figura 2.7: Representación de la rotación tomando como eje de giro $OY$ y matriz de rotación $R(y, \varphi)$.

eje y considerando un sistema de coordenadas con la regla de la mano derecha.

- Escalamiento: Dentro de un espacio de referencia los objetos pueden modificar su tamaño relativo en uno, dos, o los tres ejes. Se multiplican los valores de coordenadas por los factores de escalamiento $s_x$, $s_y$ y $s_z$. Para ello se ha de aplicar la matriz de escalamiento, y se realizar la siguiente operación:

$$V' = (x' \; y' \; z' \; 1) = (x \; y \; z \; 1) \bullet \begin{bmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

## 2.4. Despliegue de datos: bibliotecas para la visualización 3D

Un denominador común en todos los dominios de *software* de visualización es el uso de lenguajes de programación que permiten el uso de bibliotecas gráficas para el acceso directo a las unidades de procesamiento gráfico y la memoria de dispositivos de cómputo. La gran mayoría de estas aplicaciones están usando C/C++, Java o Python, así como la biblioteca gráfica OpenGL para el procesamiento de los gráficos. Las Bibliotecas gráficas y sus inter-

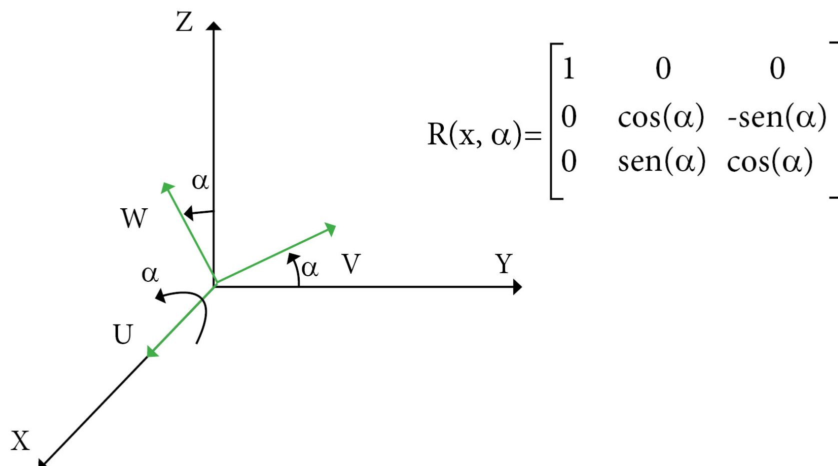



Figura 2.8: Representación de la rotación tomando como eje de giro $OX$ y matriz de rotación $R(x, \alpha)$.

faces de programación de aplicaciones (*Application Programming Interface, API*) ayudan a construir los objetos e imágenes que se muestran en la pantalla. Una aplicación de gráficos 3D se comunica con el *hardware* de gráficos a través de una API 3D (ver Figura 2.10). Las figuras 3D se describen generalmente por una representación de contorno formada por una colección de primitivas. En la mayoría de los casos, las primitivas son triángulos definidos por vértices. Así, la información del vértice consiste de posición geométrica y otros atributos como vector normal, coordenadas de textura, color u opacidad. Es posible actualmente enviar un conjunto de primitivas desde la CPU a la GPU como un flujo de vértices, que luego son transformados por el procesamiento de vértices (ver Figura 2.10). Las primitivas se convierten en fragmentos después del procesamiento de vértices. Las GPUs actuales procesan exclusivamente triángulos en estas etapas y son de interés para la química computacional por la capacidad para dar respuesta a un alto número de operaciones. Para cumplir con el objetivo de este trabajo, fue necesario investigar de qué manera se podían dibujar las isosuperficies resultantes de los cálculos de los campos moleculares. Por lo que se decidió estudiar el lenguaje OpenGL para ser utilizado en molS. A continuación, se explicarán brevemente sus características principales.

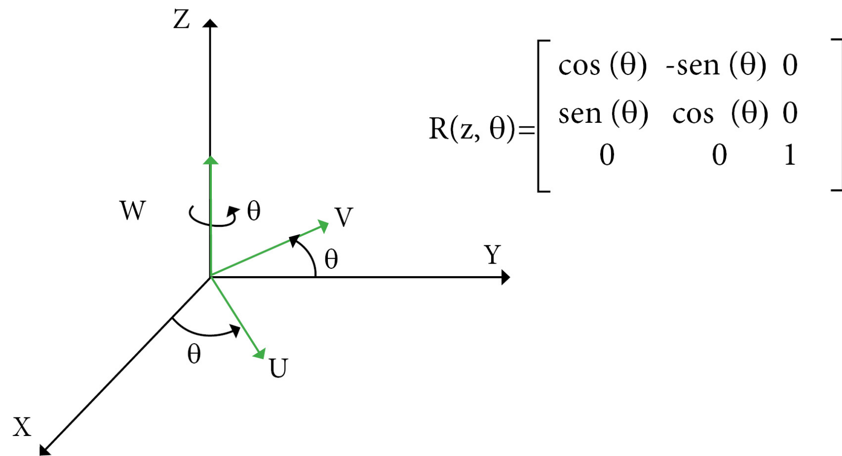



Figura 2.9: Representación de la rotación tomando como eje de giro $OZ$ y matriz de rotación $R(z, \theta)$.

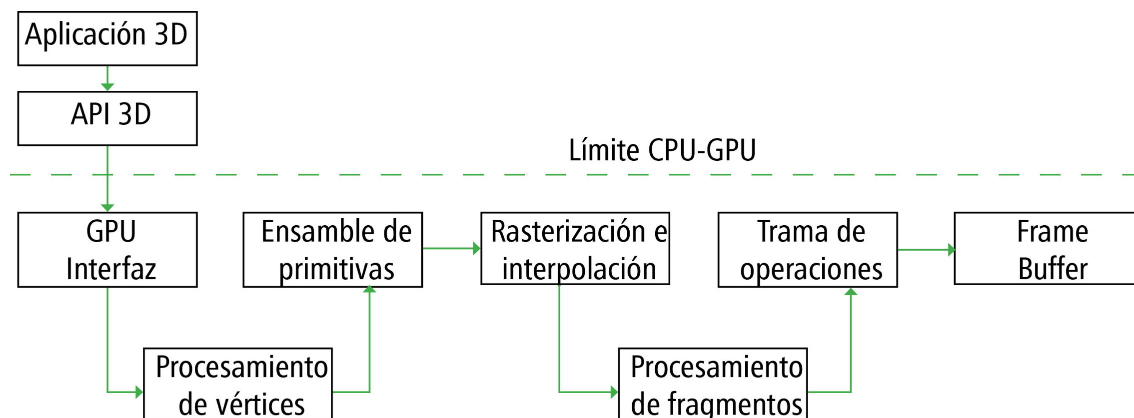



Figura 2.10: Cause GPU [16].

## OpenGL

En 1992 *Silicon Graphics* presentó *Open Graphics Library* (OpenGL), una interfaz de programación de aplicaciones multilenguaje y multiplataforma, que permite el trazado de gráficos de alto rendimiento. OpenGL es un conjunto de primitivas geométricas: puntos, líneas y polígonos, y comandos que actúan sobre estas primitivas y que permiten realizar tareas como sombreado, iluminación y texturizado[20]. Se ha consolidado como la biblioteca por excelencia para desarrollar aplicaciones 2D y 3D con independencia de la plataforma o el *hardware* gráfico. OpenGL está dividido en varias bibliotecas, cada una de ellas con funcionalidades específicas que facilitan el trabajo de programar. En la Tabla 2.1 se muestran las

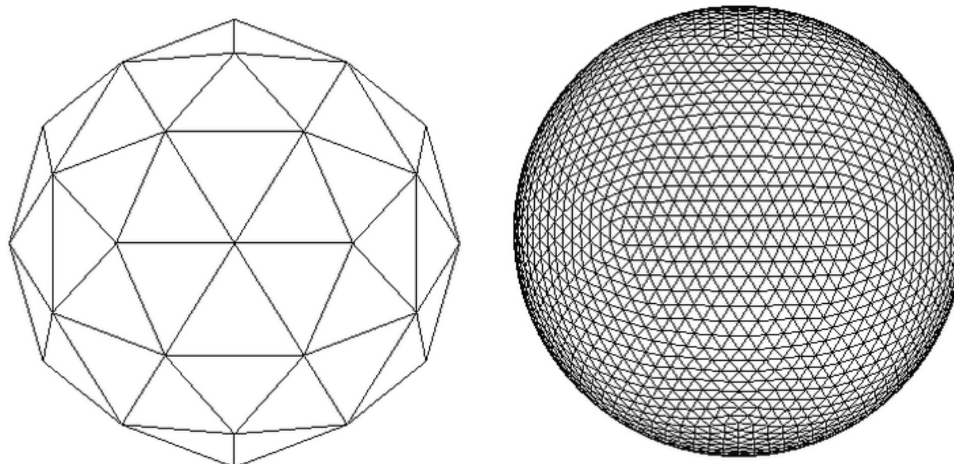
tres bibliotecas utilizadas para desarrollar gráficos con OpenGL. OpenGL puede ser usado para todo tipo de aplicaciones que necesiten visualización en tiempo real de alto rendimiento y calidad (aplicaciones científicas, médicas, militares, simuladores y videojuegos). Se puede trabajar con él desde diferentes lenguajes como C/C++, Java, Python. OpenGL es simplemente una biblioteca de *software* para acceder a características del *hardware* de gráficos.

Tabla 2.1: Biblioteca gráficas

| Biblioteca | Descrpción |
|---|---|
| OpenGL | Integrada por primitivas (vértice, líneas y polígonos) para la construcción de objetos más complejos. |
| OpenGL Utility (GLU) | Compuesta por una serie de funciones de dibujo de alto nivel que, a su vez, se basan en las primitivas de OpenGL. |
| OpenGL Utility ToolKit (GLUT) | Es una biblioteca de utilidades para programas OpenGL que principalmente proporciona diversas funciones de entrada/salida con el sistema operativo. Entre las funciones que ofrece se incluyen declaración y manejo de ventanas y la interacción por medio de teclado y ratón. |

La versión 4.3 de OpenGL tiene cerca de 500 funciones que permiten especificar objetos, imágenes y operaciones necesarias para producir aplicaciones computacionales 3D interactivas. Es independiente del *hardware* y es implementado sobre diferentes tipos de *hardware* para gráficos o como *software*, además es independiente del sistema operativo y el manejador de ventana. La Figura 2.11 es el resultado de dibujar isosuperficies utilizando OpenGL como serán utilizadas por este trabajo de investigación.

Existen varias aplicaciones que han sido desarrolladas para un despliegue gráfico de moléculas y algunos campos escalares utilizando OpenGL, tal es el caso de la herramien-



(a) 80 triángulos con 240 vértices. (b) 640 triángulos con 1920 vértices

Figura 2.11: Isosuperficies dibujadas con OpenGL.

ta, MacMolPlt (descrita en el sub capítulo 2.10). MacMolPlt utiliza OpenGL para presentar un procesamiento 3D de alta calidad.

## 2.5. Campos escalares y vectoriales en la química cuántica

La visualización de volúmenes es una técnica ampliamente utilizada en muchas ramas de la ciencias y las ingenierías, desde la visualización de campos escalares de ingeniería hasta las imágenes médicas, y la representación de nubes u otros fenómenos gaseosos en simulaciones y entornos visuales. En los últimos años la representación mediante texturas se ha convertido en un método popular para la visualización del volumen. En la química cuántica las variables básicas involucradas en el estudio de átomos y moléculas son el número de electrones, el tipo de núcleos y su posición. Por ejemplo, la molécula de agua consta de tres átomos: un átomo de oxígeno, dos átomos de hidrógeno, y 10 electrones. Una representación de esta molécula se muestra en la Figura 2.12. Lo que le hace falta a esta gráfica es la distribución de los electrones alrededor de los átomos. Una forma de conocer la distrubución electrónica alrededor de los

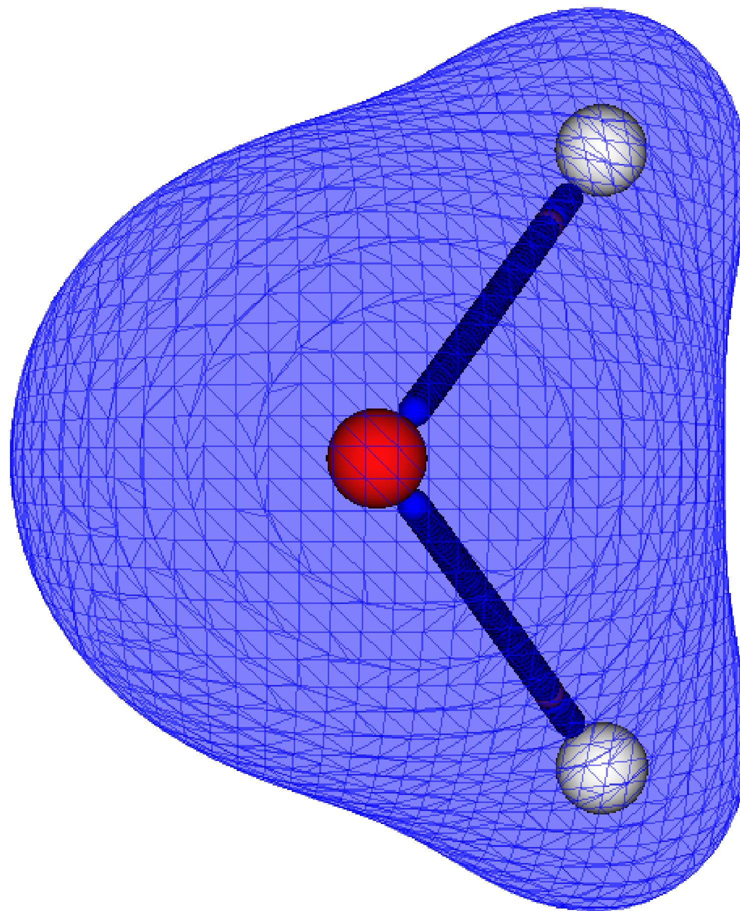


Figura 2.12: Visualización de un campo escalar y geometría de una molécula de agua.

núcleos es a través del estudio de la densidad electrónica, $\rho$, la cual es obtenida a partir de

$$\rho(\vec{r}) = \sum_{i=1}^{ocupado} n_i \psi_i(\vec{r})^* \psi_i(\vec{r}), \tag{2.1}$$

donde $\psi$ representa un orbital y $n_i$ su ocupación. En este proyecto usaremos una representación de los orbitales en términos de un conjunto de funciones de base, $\{g_\mu(\vec{r})\}$ con

$$g_\mu(\vec{r}) = (x - X)^l (y - Y)^n (z - Z)^m \exp(-\zeta(r - R)^2). \tag{2.2}$$

En esta expresión $X, Y, Z$ representan las coordenadas donde la función está centrada y $\zeta$ depende del centro que se esté usando. Así, los orbitales quedan descritos en términos de estas funciones de base de la siguiente manera

$$\psi_i(\vec{r}) = \sum_{\mu=1}^{K} c_\mu^{(i)} g_\mu(\vec{r}), \tag{2.3}$$

Tabla 2.2: Algoritmo para visualizar una isosuperficie

| ALGORITMO GENERAR MALLA(arregloDensidadElectronica) |
|---|
| 1. Leer el arreglo con densidad electrónica<br>2. Describir la malla 3D<br>3. Calcular isosuperficies en cada cubo interno con marching cubes en función de un isovalor<br>▪ Calcular vértices para la isosuperfice<br>▪ Calcular las normales para la isosuperficie<br>▪ almacenar los vértices y normales<br>4. Cargar vértices y normales al buffer de bibliotecas de visualización |

donde $K$ representa al número de funciones a usar y los coeficientes $c_\mu^{(i)}$ se obtienen al hacer un cálculo de la química cuántica. Generalmente en este tipo de cálculo es donde se requiere de cómputo de alto rendimiento. Por ejemplo, se pueden tener sistemas moleculares con cientos de orbitales y miles de funciones de base. Pensemos que deseamos obtener la gráfica de una densidad electrónica. Para esta situación, requerimos del archivo donde se encuentren los coeficientes $c_\mu^{(i)}$ y definir una malla para hacer la gráfica. En nuestra aplicación hemos decidido hacer la malla de la siguiente manera: El sistema de estudio se envuelve por una malla tridimensional ó paralelepípedo, que se interpreta de la formación de cubos dentro de él. En cada vértice de los cubos internos se tiene la evaluación de densidad electrónica[23]. En la química cuántica, para visualizar una isosuperficie que describe la densidad electrónica de cierto valor ó densidad electrónica (isovalor), se reduce la malla utilizando el algoritmo marching cubes (descrito en la Tabla 2.3). La isosuperficie se genera siguiendo el algoritmo de la Tabla 2.2.

## 2.6. Visualización de superficie

Dentro de la investigación en visualización científica, la representación de datos volumétricos se destaca por las dificultades computacionales que plantea, pero al mismo tiempo concentra la mayor atención en la investigación actual. La técnica de visualización volumétrica permite representar cualquier fenómeno en una figura geométrica que pueda ser reconstruida como un sólido con características de continuidad, distribución y estructura en capas o niveles.

### Algoritmo *Marching Cubes*

Uno de los algoritmos de *rendering* de volúmenes mediante isosuperficies difundido y utilizado es el denominado *marching cubes*, propuesto por Lorensen y Cline en 1987 [19]. El algoritmo de marching cubes se basa en atravesar el volumen de un cubo. Un cubo se compone de ocho vértices y los valores de cada uno de los vértices determinan si las aristas del cubo son interceptadas por la superficie dada. La condición para que una arista intercepte la superficie es que uno de los valores de un vértice debe ser menor que el isovalor y el otro debe ser mayor a éste. El punto de intersección de las aristas se calcula como una media ponderada de las intensidades de los vértices. Los puntos de intersección forman un conjunto de triángulos que representan la superficie. Hay 256 posibles casos de intersección entre un cubo y la superficie, pero por consideraciones de simetría se reducen en principio a solo 15 (Ver Figura 2.13)[19, 9]. Para cada triángulo de la malla, la norma unitaria a su superficie se calcula para los propósitos de sombreado. El algoritmo de *marching cubes* adaptado para este trabajo de investigación se describe en la Tabla 2.3. A diferencia del algoritmo original de *marching cubes* el descrito en la Tabla 2.3 es implementado sobre GPUs que lo hace ser más veloz y menos demandante en los recursos de CPU.

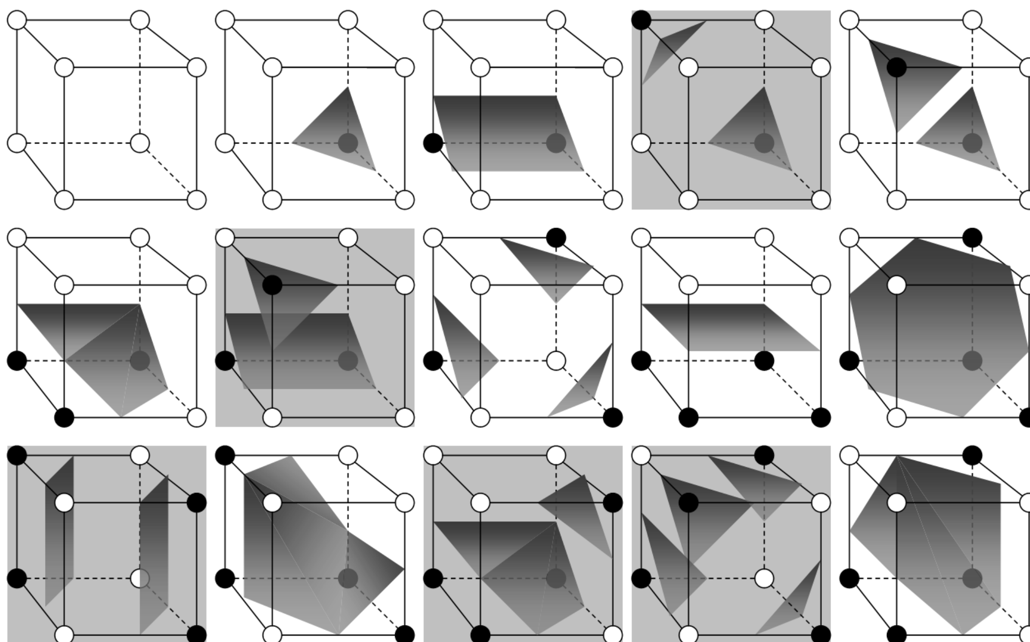


Figura 2.13: Casos base del algoritmo *marching cubes* (los vértices negros tienen un valor mayor al isovalor).

## 2.7. El proceso de visualización

Los modelos para describir los sistemas de moléculas se calculan en una red (o malla) tridimensional, y los datos resultantes se utilizan para el trazado de isocontornos e isosuperficies para su visualización, así como para otros tipos de análisis. Existen paquetes de *software* que permiten el rendereo (graficación) de sistemas moleculares realizando cálculos en la CPU y requieren tiempos de ejecución de decenas a cientos de segundos dependiendo de la complejidad del sistema molecular [17]. Existen implementaciones de algoritmos en el popular programa de visualización molecular VMD, que ahora produce alta calidad de representaciones gráficas para grandes sistemas de moléculas en menos de un segundo, alcanzando por primera vez animaciones interactivas de simulaciones de trayectorias de química cuántica en cálculos de tiempo real. Las herramientas de visualización existentes realizan cálculos de sistemas moleculares en la CPU y requieren tiempos de ejecución de decenas a cientos de segundos, dependiendo de la complejidad del sistema molecular y la resolución de las mallas de evaluación. Las herramientas que utilizan la CPU se limitan a la característica

Tabla 2.3: Algoritmo *marching cubes*

```
ALGORITMO MARCHINGCUBES(Mesh, isoLevel)
Load Mesh
FOR ALL marchingCube
  cube = verticesMarchingCube(Malla)
  case = caseFindMarchingCube(cube)
  totalTriangles = findTotalTriangles(case)
  FOR ALL triangle
    vertexTriangle = computeTriangle(triangle)
    normalTriangle = computeNormal(vertexTriangle)
    vertexArray[i] = vertexTriangle
    normalArray[i++] = normalTriangle
```

del procesamiento en forma serial inherente a esta tecnología. Un procesador secuencial, se basa en la arquitectura Von Neumann, ejecuta instrucciones secuencialmente en cada ciclo de reloj. Cada instrucción es ejecutada por la CPU de una en una. Pocos de estos paquetes son implementados para hacer uso de procesadores multinúcleos, y ninguno para unidades de procesamiento gráfico. Las GPU, a diferencia de los CPU, hacen procesamiento *stream*. Un *stream* ejecuta una función (*kernel*) sobre un conjunto de datos de entrada de forma simultánea. Los elementos de entrada se pasan al *kernel* y se procesan de forma independiente sin dependencias entre otros elementos(datos). Esto permite que el programa se ejecute de forma paralela nativamente. Naturalmente, existe una gran oportunidad de mejorar las capacidades en términos de interactividad (procesamiento en tiempo real de cálculo y visualización), calidad de visualización, y escalabilidad a sistemas más grandes y complejos [12]. Actualmente, las GPU se han convertido en dispositivos de cómputo masivamente paralelo, y más recientemente se ha hecho posible programarlos en lenguajes de programación populares como C/C++ y Fortran. Esto ha creado una gran oportunidad para emplear nuevas técnicas de simulación y análisis que antes eran demasiado demandantes computacionalmente. En otros casos, la potencia de cálculo proporcionada por las GPU puede hacer factible las técnicas de análisis que antes requerían de cálculo en la computación de alto rendimiento (HPC) y

que sean accesibles a los científicos que en muchas ocasiones carecen de experiencia con clusters, sistemas de colas, y similares o que no tengan acceso a recursos de supercómputo [15]. Dado el costo y disponibilidad limitada de *hardware* de cómputo de alto desempeño, se han buscado mejores opciones para realizar simulaciones más grandes, con una mayor velocidad para la comunidad científica. El último gran avance en este sentido fue la evolución a clusters de alto desempeño Linux basados en equipos de cómputo ordinarios. Como siguiente avance, se requiere una tecnología de productos básicos con un fuerte apoyo comercial, se cree que este avance se encuentra en los aceleradores gráficos 3D inspirados en la demanda pública de realismo visual en los juegos de computadoras [24]. Las GPU están diseñadas como dispositivos orientados a rendimiento proporcionando decenas de miles de cálculos independientes. Esta clave de diseño permite a las GPU disponer de una matriz de unidades aritméticas en lugar de memorias caches, sacrificando el uso de decodificadores de instrucciones independientes, generando un procesador de arquitectura SIMD (Una instrucción - Multiples Datos) que comparten decodificadores de instrucciones. Esta elección de diseño maximiza el número de unidades aritméticas por mm cuadrado. Con la multiplexión se logra la ejecución de miles de hilos en cada unidad de procesamiento físico, gestionados por un programador de *hardware* que puede intercambiar los grupos de hilos activos e inactivos así como las operaciones de memoria en cola. De esta manera, las operaciones de la memoria de un hilo se traslapan con las operaciones aritméticas de los demás. GPU recientes puede ejecutar hasta 67 millones de hilos de ejecución por cada *kernel* lanzado [24]. Las GPU han traído grandes beneficios para la aceleración y aumento de rendimiento en simulaciones de las ciencias básicas. Haciendo cálculos hoy en día, más rápidos y con mayor comodidad, proporcionando resultados durante el almuerzo en lugar de durante la noche, permitiendo la comprobación de resultados en menos tiempo. Este trabajo considera específicamente el uso de las unidades de procesamiento de gráficos rápidos para obtener un alto rendimiento - la ganancia de rendimiento puede ser tanto como de dos o más órdenes de magnitud en comparación con una unidad central de procesamiento.

## 2.8. Unidades de procesamiento gráfico, GPU

En años recientes la industria de las tarjetas gráficas ha diversificado su uso, pasando de ser utilizadas para procesamiento visual y a ser aprovechados para procesamiento general de datos numéricos. Existen tarjetas gráficas que están conformados de miles de procesadores para ejecutar concurrentemente programas y efectuar operaciones de punto flotante (en precisión simple o precisión doble), con tal magnitud que pueden competir con servidores, o agregados de servidores (Clusters). Esta cualidad de las tarjetas gráficas ha sido blanco para que los centros de supercómputo [25] y los investigadores las utilicen en el procesamiento de simulaciones, logrando reducir la cantidad de servidores por cluster e incrementar el poder de cómputo reduciendo el consumo de energía y espacio físico para su instalación. Una limitante para el uso de tarjetas gráficas como unidad de procesamiento de datos es la memoria física de la que disponen estos dispositivos ya que no comparten su memoria interna con la del sistema de cómputo donde se encuentran instaladas; debido a que la memoria del GPU y CPU no están unificadas existen consideraciones que se deben realizar para lograr sacar provecho de las GPU. Al brindar las tarjetas de video la capacidad de procesamiento de datos, resulta conveniente calcular y visualizar sobre el mismo dispositivo. Por otro lado son escasas las aplicaciones libres y comerciales que hacen uso de estaciones de trabajo o computadoras personales con tarjetas gráficas para explotar simultáneamente capacidades de cómputo científico y de visualización[13]. Las unidades de procesamiento gráfico son dispositivos presentes en computadoras personales actuales, proveen de operaciones básicas al CPU, tales como el rendereo de imágenes en memoria y el despliegue de imágenes en pantalla [6]. Las nuevas arquitecturas (Figura 2.14) de las tarjetas GPU tienen desde decenas hasta millares de unidades de procesamiento o SP (Stream Processors) que a su vez se organizan en SM(Stream Multiprocessors), estos se encuentran organizados de manera matricial lo cual lo asemeja a la taxonomía de Flynn SIMD [16]. Estos dispositivos disponen de memoria (global, compartida, constante, textura y registros) para diferentes niveles de acceso, lo que, lo hace análogo a cluster de alto rendimiento. Estas tarjetas se conectan a computadoras personales y servidores multicores mediante un bus de comunicaciones PCIe 2/3. En los últimos años la mejora en el rendimiento en los procesadores de propósito general ha disminuido, a cambio,

el rendimiento en las GPU ha aumentado. La razón entre las GPU de múltiples núcleos y procesadores multinúcleo en operaciones de punto flotante es superior de 10 a 1 [16, 18]. El rendimiento de las tarjetas gráficas se emplea en muchas aplicaciones lo que ha permitido que los desarrolladores abran nuevas ramas del conocimiento como el procesamiento de vídeo e imágenes, la biología y la química computacional, la simulación de la dinámica de fluidos, la reconstrucción de imágenes de Tomografía computalizada, el análisis sísmico o el trazado de rayos (ray casting ), entre otras.

Entre las principales virtudes que tienen las GPU y que serán utilizadas en este proyecto, se resumen las siguientes: tarjetas programables, altamente paralelas, multicores (soporte a múltiples hilos), altísimo poder de cálculo y gran ancho de banda. Por otra parte, existen diversas plataformas para la computación de GPU, dos de las más populares, son: *Open Computing Library (OpenCL)* y *Compute Unified Device Architecture (CUDA)*. Las dos plataformas están muy cerca de ofrecer el mismo *performance*, pero CUDA ofrece un mejor soporte del manejo de las tarjeta NVIDIA.

### 2.8.1. Arquitectura Unificada de Dispositivos de Cómputo, CUDA

CUDA es una arquitectura de cálculo paralelo generado por la compañía NVIDIA que aprovecha la gran potencia de las GPU para proporcionar un incremento extraordinario del rendimiento del sistema[16, 25, 6, 22]. Para programar en CUDA hay que realizar un enfoque diferente del problema a resolver, explotando sobre todo el paralelismo de datos. Es decir, los problemas resueltos con secuencia de código que se realiza sobre muchos datos diferentes son ideales para explotar el paralelismo que ofrece CUDA. Estos programas paralelizados se ejecutarán en una tarjeta gráfica con un procesador masivamente paralelo. La tarjeta gráfica consta de varios multiprocesadores, cada uno de los cuales esta formado a su vez por varios procesadores. Estos procesadores son los que se encargan de ejecutar los hilos de ejecución (*threads*) que vienen agrupados en bloques. Así se estará ejecutando la misma aplicación (*kernel*) en todo el multiprocesador pero actuando sobre diferentes datos[6, 16]. Una arquitectura típica CUDA consta de los componentes como se ilustra en la Figura 2.14[16]. Es así que la arquitectura CUDA permite abrir al programador la tarjeta gráfica para utilizarla como un co-procesador matemático. Los componentes *Host CPU, Bridge y System memory*

son externos a la tarjeta gráfica, y se denominan colectivamente como el *host*. Todos los componentes restantes forman la GPU y la arquitectura CUDA, y se denominan colectivamente como dispositivo[14, 13]. La unidad de interfaz de host es responsable de la comunicación, de responder a los comandos, y de facilitar la transferencia de datos entre el *host* y el dispositivo. Por ende, la programación en CUDA es un tipo de programación heterogéneo que implica la ejecución de código en dos plataformas diferentes: el *host* y el dispositivo. El sistema *host* se compone principalmente de la CPU, la memoria principal y su arquitectura de soporte. El dispositivo es generalmente la tarjeta de video que consiste de una GPU con soporte de la arquitectura CUDA. El código fuente de un programa CUDA consiste tanto en el código para el *host* y el código para el dispositivo; mezclados en el mismo archivo. Debido a que el código fuente se dirige a dos arquitecturas de procesamiento diferentes, se requieren pasos adicionales en el proceso de compilación. El NVidia C Compiler (NVCC) primero analiza el código fuente y crea dos archivos separados. El fichero *host* se compila con un compilador estándar de C / C ++ que produce archivos de objeto de la CPU estándar. El fichero de dispositivo se compila con el CUDA C Compiler (CUDACC) que produce archivos objeto CUDA. Estos ficheros objeto están en un lenguaje ensamblador conocido como Parallel Thread eXecution o archivos PTX. Los archivos PTX son reconocidos por los controladores de dispositivos que se instalan con tarjetas gráficas NVIDIA. El conjunto de archivos resultantes se vinculan y se crea un ejecutable CPU-GPU.

El parámetro de configuración más importante requerido por cada *kernel* es el particionamiento de hilos. El hilo es una unidad abstracta primaria, que se utiliza para modelar el cálculo simultáneo. Su concepto es muy similar a los hilos de la CPU, donde cada hilo tiene su identificador único y memoria local para variables. Para el caso de CUDA, los hilos siempre se organizan en bloques. Un bloque puede tener hasta tres dimensiones, donde la extensión de las dos primeras puede ser como máximo 512, mientras que la última no debe exceder de 64. A pesar de que los límites permiten mucho más, los bloques no pueden tener más de 1.024 hilos. Para permitir el procesamiento de gran número de hilos, los bloques se dividen en una malla, que puede tener de manera similar hasta tres dimensiones con un tamaño que va desde 1 hasta 65,536. Para distinguir los hilos dentro del núcleo podemos utilizar variables incorporadas para determinar el ID de cada hilo. La plataforma de cálculo paralelo CUDA

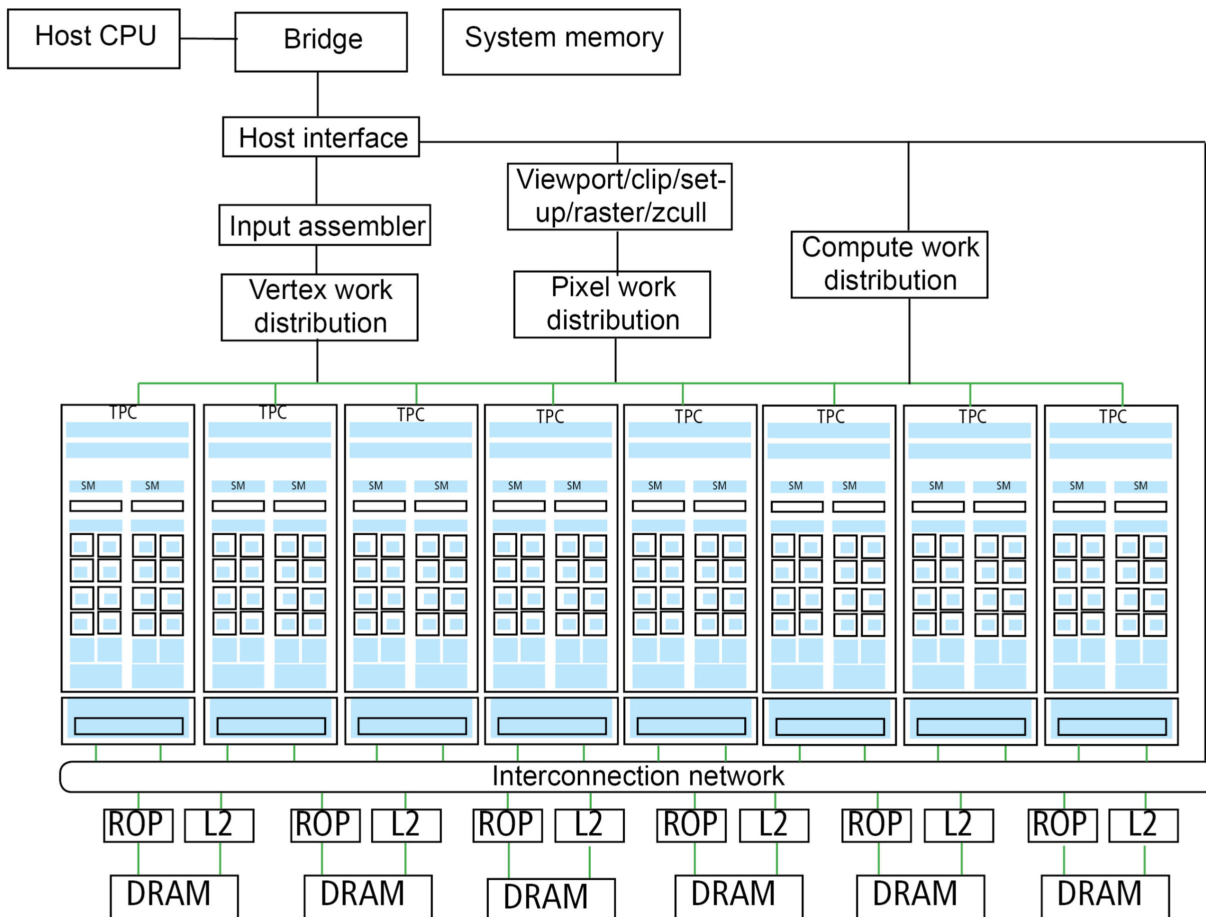



Figura 2.14: GPU Architecture. TPC: Texture/processor cluster; SM: Streaming Multiprocessor; SP: Streaming Processor.

proporciona extensiones de C y C++ que permiten implementar el paralelismo en el procesamiento de tareas y datos con diferentes niveles de granularidad, sin tener que aprender un nuevo lenguaje de marca. El programador puede expresar ese paralelismo mediante diferentes lenguajes de alto nivel como C, C++ y Fortran o mediante estándares abiertos como las directivas de OpenACC. En la actualidad, la plataforma CUDA se utiliza en miles de aplicaciones aceleradas en la GPU y en miles de artículos de investigación publicados[16]. En la Figura 2.15 se compara el rendimiento en GFLOPS de las tres principales compañías que dedican sus unidades de procesamiento numérico a brindar mayores capacidades de operaciones flotantes por ciclo de reloj.

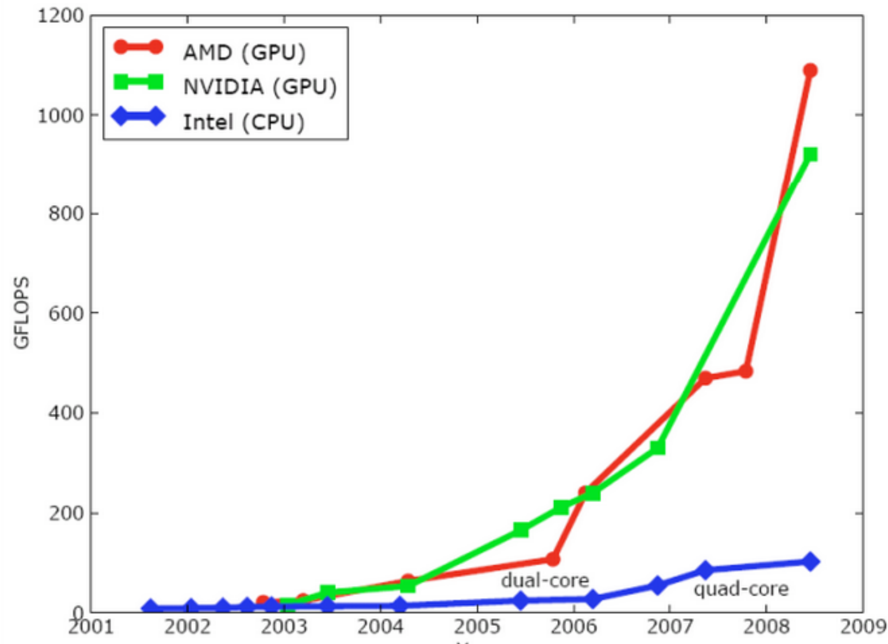



Figura 2.15: Rendimiento entre GPU y CPU.

### 2.8.2. Interoperabilidad de OpenGL y CUDA

OpenGL y CUDA pueden trabajar de la mano para agregar mayor rendimiento a las aplicaciones interactivas o animaciones, NVIDIA ha desarrollado bibliotecas para la interoperabilidad entre estos, a continuación se describe el procedimiento para hacer uso:

- Declaración, configuración de los objetos en el contexto de OpenGL.
- Registro de los objetos en el contexto de CUDA.
- Mapeo de los objetos al contexto de CUDA.

Los objetos buffers de OpenGL pueden ser mapeados dentro del espacio de direcciones de CUDA y entonces ser usados como memoria global (Vertex buffer objects y Pixel buffer objects), esto permite la visualización de los datos calculados. En el Apéndice C se presentan las instrucciones necesarias para la interoperabilidad entre OpenGL y CUDA.

### 2.8.3. OpenGL Shading Language

GLSL o el *OpenGL Shading Language* se utiliza para escribir programas que se ejecutan en la GPU. Permite especificar segmentos de programas gráficos que serán ejecutados sobre el dispositivo gráfico [28]. El lenguaje se define en un documento de especificación desarrollada por el ARB[1] que se puede encontrar en el registro opengl.org junto a la especificación OpenGL. GLSL es un lenguaje de alto nivel que toma prestados muchos elementos de C y C ++, por lo que la sintaxis es familiar a C. A diferencia de C, GLSL en su main () no toma ningún argumento, sino los datos que entran y salen de una etapa de shading se pasan con el uso de variables globales especiales en el shader[2], esto es diferente a las variables globales que se definen en los programas. Hay dos tipos principales de shaders: *Vertex Shaders y Fragment Shaders*. Cuando se usan *shaders*, se tomará el control de las secciones del cause de la Figura 2.16; esto proporciona una enorme cantidad de flexibilidad y control, pero implica que la responsabilidad de algunas tareas se maneja a través del uso de shaders.

1. *Vertex* shaders son programas que operan en cada vértice que se envía a la tarjeta gráfica por un comando de renderizado como *glDrawArrays()*. Es la responsabilidad de este shader calcular la posición final del vértice y también calcular los atributos de los vértices tales como los colores. Una cosa importante a tener en cuenta es que un vertex shader sólo sabe de un solo vértice a la vez; no es posible extraer información sobre vértices vecinos. La única salida que se requiere por un vertex shader es la posición. Esto se debe almacenar en una variable incorporada, que es un vector de punto flotante de cuatro elementos (vec4) llamado *gl_Position*. Una vez que la etapa de procesamiento de vértices es completa, la tarjeta gráfica toma de nuevo el control hasta la etapa de procesamiento de los fragmentos. Cuando se utiliza un vertex shader, tendrá que manejar manualmente las siguientes actividades: transformación de vértices, generación de coordenada de texturas, cálculos de iluminación, aplicación del color, transformación de la normal.

2. *Fragment Shaders* cuya función es calcular el color de salida final de un píxel que se

[1] Architecture Review Board, comunidad dedicada al soporte y desarrollo de OpenGL

[2] Los programas escritos en GLSL se les denomina *shader*.

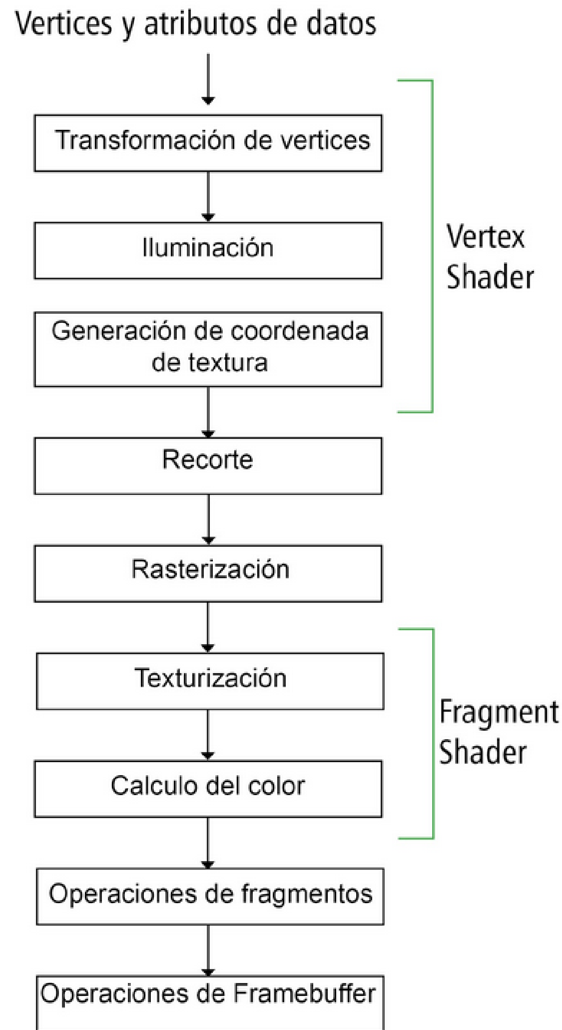



Figura 2.16: Las Etapas *Vertex Shader* y *Fragment Shader* del cause de visualización suelen ser reemplazados.

almacena en la memoria de vídeo. Los *Fragment Shaders* toman las salidas del vertex shader como variables de entrada; se pueden usar estas entradas para colorear y darle la textura al fragmento, o para alcanzar efectos más avanzados como el *bump mapping* o la iluminación por pixel. A la salida de los *Fragment Shaders* se tiene un único dato, un arreglo de cuatro elementos para formar el color del píxel. Cuando se utiliza un *Fragment Shaders* debe manejar las siguientes partes del cause: Computing per-pixel colors, aplicación de texturas, cálculo de fog por pixel, aplicando iluminación por pixel. GLSL toma prestada la idea de un preprocesador de C. Las directivas del preprocesador

se evalúan antes de la compilación del código fuente y puede alterar el código final que se compila. La lista completa de las directivas de preprocesador y sus funciones se muestran en la Tabla 2.4. Para utilizar GLSL en el código, son necesarias las funciones de OpenGL 4.3.

Tabla 2.4: Directivas del preprocesador soportadas por GLSL

| Directiva | Función |
|---|---|
| # | Hace caso omiso |
| #define | Define una constante o macro |
| #undefine | Elimina le definición una constante previamente definida |
| #if #ifdef #ifndef #else #elif #endif | Compilación condicional |
| #error | Inserta un mensaje de error dentro de la bitacora de salida |
| #pragma | Declarar opciones específicas de la implementación |
| #line | Se usa para sobrescribir la organización de las lineas del compilador |
| #version | Elija la versión de GLSL a utilizar ( la versión actual es la 4.30) |
| #extension | Utilice para habilitar la funcionalidad extendida a disposición de la aplicación GLSL. |

## 2.9. Aplicaciones para visualización en química cuantica

Existe una gran variedad de aplicaciones para la visualización de sistemas moleculares, todas ellas con enfoques y posicionamientos diferentes.

1. ***Visual Molecular Dynamics (VMD)[13]:*** Desarrollado por *Theoretical and Computational Biophysics Group*, es un *software* que permite realizar las tareas de desplegar, animar y analizar sistemas biomoleculares de gran tamaño. Es usado principalmente como una herramienta científica para la visualización interactiva en 3D [24, 29, 14].

Está diseñado para hacer uso de diversos sistemas de *hardware*, tal como, laptops, desktops, clusters. VMD puede mostrar simultáneamente cualquier número de estructuras utilizando una amplia variedad de estilos de rendering y métodos de coloración. Las moléculas se muestran como una o más *representaciones*, en el que cada representación encarna un método de representación y esquema de color particular para un subconjunto seleccionado de átomos. Entre las ventajas de este *software* se encuentran que es gratuito, tiene una excelente interfaz gráfica, esta disponible para GNU/Linux y Windows, y es apropiado para macromoléculas y dinámica molecular. Sin embargo, VMD no es muy bueno para visualizar moléculas pequeñas.

2. ***A Graphical User Interface for Computational Chemistry Software (Gabedit) [2]:*** Desarrollado por Abdul-Rahman Allouche. Es una interfaz gráfica para el usuario, sin costo alguno, que ofrece preprocesamiento y postprocesamiento adaptado a nueve paquetes de *software* de la química computacional. Interfaz para el postprocesamiento de aplicaciones de química cuántica. Incluye herramientas para editar, desplegar, analizar, convertir y animar sistemas de moléculas [2]. Los archivos de entrada pueden ser generados por el *software* de química computacional apoyada por Gabedit. Algunas propiedades moleculares de interés se procesan directamente de la salida de los programas de química computacional; otros se calculan en Gabedit antes de la visualización. Lo que destaca de este *software* es la construcción y visualización de moléculas cuánticos.

3. **Molden[26]:** Es un paquete de *software* para pre y post-procesamiento de datos de los programas de química computacional. Permite visualizar propiedades moleculares del sistema a partir de los datos obtenidos con los programas de cálculo ab initio GAMESS y GAUSSIAN, y los de cálculo semiempírico MOPAC/AMPAC, leyendo directamente la información de los ficheros de salida. Molden puede visualizar la geometría, los datos volumétricos como superficie o como forma de contornos.

4. **MacMolPlt[4]:** Es definido por sus desarrolladores como un programa de gráficos modernos para el trazado de estructuras moleculares en 3D y de los modos normales (vibraciones). MacMolPlt incluye una interfaz estándar para abrir archivos, guardar

archivos, imprimir, copiar, comandos de visualización y movimiento de ventana. Una característica muy agradable es la ayuda proporcionada por *tooltip*.

Tabla 2.5: Aspectos de las aplicaciones para la visualización de química cuántica

| | VMD | Gabedit | Molden |
|---|---|---|---|
| Manejo de campos escalares | Si | Si | Si |
| Manejo de campos vectoriales | No | - | No |
| Se ejecuta en *hardware* | GPU | CPU | CPU |
| Disponible en ISO | - | - | - |
| Visualización de grandes moléculas ($>$ 128 átomos) | Si | - | - |
| Visualización de pequeñas moléculas ($<$ 64 átomos) | Si | - | - |
| Manejo de archivos de entrada/salida | Si | Si | Si |
| Interfaz de usuario | Si | Si | Si |

# Capítulo 3

# Diseño del programa molS

Con la química cuántica se realizan diferentes estudios sobre la estructura electrónica de átomos, moléculas, superficies y/o sólidos. Las funciones que describen la geometría y densidad electrónica de sistemas moleculares se calculan en una malla tridimensional, y los datos resultantes se pueden entonces utilizar para el trazado de isocontornos e isosuperficies para su visualización, así como para otros tipos de análisis, lo que demanda el uso de cómputo de alto rendimiento para la ejecución del cálculo y la visualización. En [1], se explica que los métodos empleados requieren evaluar integrales bielectrónicas, realizar sumas de elementos, entre otras operaciones para efectuar el estudio de la estructura electrónica de átomos y moléculas, lo que da lugar a una cantidad de operaciones en el orden de millones de operaciones por segundo. El tamaño y la complejidad de este conjuntos de datos hace que sea cada vez más difícil de entender, comparar, analizar, comunicar y visualizar los datos. Mejorar el proceso de visualización en aplicaciones de la química cuántica, es una preocupación latente para los investigadores y que ha propiciado el desarrollo de esta tesis. A partir de la investigación realizada en los Capítulos 2 y 3, se compararon las aplicaciones para la visualización. Con base al estudio realizado en el capítulo 2, resulta conveniente integrar las siguientes características a la herramienta molS:

1. **Visualización de sistemas moleculares:** La visualización de los sistemas moleculares se limita a obtener desde un arreglo las posiciones en el espacio cartesiano y el tipo de átomo o punto a representar mediante una esfera. Dado el átomo (H para Hidrogenos, Li para Litio, etc.) este se buscará en una base donde se tienen registrados

todos los átomos conocidos, en esta base tambien se define el color y el radio de cada esfera (átomo). Además de los átomos registrados tambien se registran puntos como: PCE punto critico de enlace, PE punto de enlace, etc. Posteriormente se listarán todas las posibilidades.

2. **Visualización de campos escalares:** Para la visualización de campos escalares se identificaron las siguientes tareas:

   - Generación de la malla de evaluación (dominio).
   - Evaluación del campo escalar sobre la malla,
   - Paso de los datos generados a la tarjeta gráfica[1, 10].

   A través de las tres tareas, molS permitirá visualizar un campo escalar calculado por el módulo externo "Procesamiento de campo" (véase Figura 3.2). El módulo entrega un archivo en formato plano con un arreglo de valores de un campo escalar dado en un espacio tridimensional. molS lee los valores de dicho campo como un arreglo unidimensional. El cual procesa como una malla tridimensional al cual aplica el algoritmo *marching cubes* para obtener una colección de valores que representarán una isosuperficie. De esta manera molS muestra en la pantalla un campo escalar que puede rotarse, trasladarse y/o escalarse.

3. **Visualización de campos vectoriales:** Para la visualización de campos vectoriales se identifican las mismas tareas que con el campo escalar.

4. Ejecución en el *hardware* GPU-CPU: La visualización de campos escalares y vectoriales recurre comúnmente al esquema de evaluación sobre el CPU (o en un conjunto de CPUs) y después transferir el campo generado para su visualización en una estación de trabajo con capacidades gráficas. En este sentido se está perdiendo la cualidad de la tarjeta gráfica de hacer operaciones de punto flotante, además de resultar necesario realizar la transferencia de datos del CPU a la tarjeta gráfica para generar el despliegue gráfico. Por lo que molS pretende hacer las operaciones de punto flotante y la visualización en las GPU, además que resulte transparente al usuario.

## 3.1. Ambiente molS

Este trabajo se enmarca dentro de la línea de Cómputo de Alto Rendimiento, para construirlo se requirio del aprendizaje de las herramientas, bibliotecas, conceptos mostrados en la Figura 3.1. Se muestran en forma de pirámide para obviar como se soportan entre ellas. Para

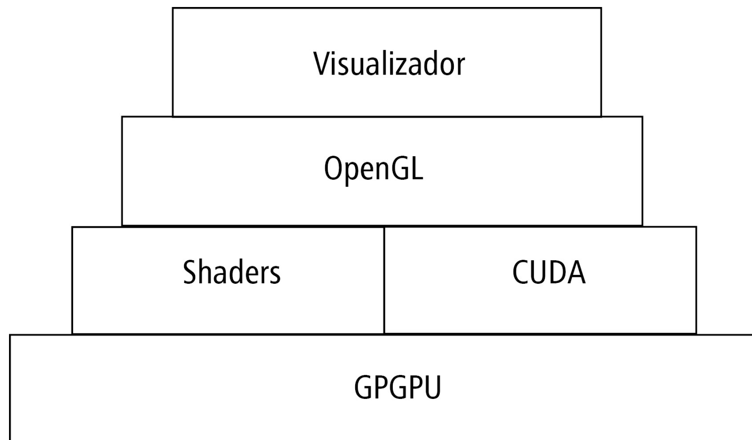



Figura 3.1: Herramientas, Bibliotecas y conceptos para la construcción del *software* para visualización molS.

la visualización de los campos vectoriales y escalares se estudió la Arquitectura de las Unidades de Procesamiento Gráficas, la Arquitectura de Unificada de Dispositivos de Cómputo (CUDA), OpenGL, la interoperabilidad de OpenGL y CUDA y el Lenguaje de Sombreado de OpenGL (GLSL). El *software* molS fue desarrollado con el lenguaje C++, la biblioteca de visualización GLUT, la biblioteca de procesamiento CUDA y bajo el sistema operativo Centos 6.

## 3.2. Arquitectura molS

La arquitectura de molS (Ver Figura 3.2) se construyó como una instancia de la arquitectura de referencia: Arquitectura en capas. La arquitectura en capas ayudará a que molS sea estructurado en grupos lógicos que están a un nivel particular de abstracción y de *hardware*. De esta manera en molS se soporta el desarrollo independiente y la evolución de diferentes partes del sistema. molS se integra por cuatro capas y cada capa tiene una responsabilidad específica:

1. Capa 1: Procesamiento. En esta capa se realiza el procesamiento de la información con la ayuda de operaciones primitivas para manipular el *hardware*.

2. Capa 2: Sistema molecular y Campo escalar. Esta capa se comunica con la capa 3 para entregarle los datos que se van a desplegar y se compone de los siguientes grupos lógicos:

   - Sistema molecular: Procesa y acomoda la información proporcionada del procesamiento externo para situar la geometría del sistema molecular en escena.
   - Campo escalar: Procesa y trata la malla tridimensional mediante el algoritmo de *marching cubes* para obtener los valores que describirán una isosuperficie. Se comunica con la capa "Simulador" y "OpenGLbuffer".

3. Capa 3: Simulador. Esta capa integra los componentes a visualizar. Se gestionan las instancias de procesamiento numérico y los arreglos a visualizar, así como determina la operación de molS, esto es, que trabaje con archivos de entrada o interactúe con los módulos externos de procesamiento escalar y vectorial.

4. Capa 4: Manejador de ventanas. Esta capa realiza la renderización que comprende las tareas de desplegar, rotar, escalar y transladar escenas con sistemas de moléculas y/o campos escalares y vectoriales. Además, permite la interacción del usuario con molS mediante el teclado y mouse para manipular la ventana de visualización.

## 3.3. Funcionamiento de molS

Como ya se describió en los capítulos anteriores, para la visualización de los campos se recurre a una malla. El componente Campo Escalar se encargará de gestionar la malla de evaluación. Por lo que recibe un apuntador al vector (ver Figura 3.3) de densidad electrónica.

El vector es una colección de parámetros que describen los valores para la densidad electrónica de la malla tridimensional: cantidad de átomos, posición de cada átomo, distancias y cantidad de puntos evaluados en la malla tridimensional y valores de la densidad eléctrica. A continuación se describen los parámetros que integran la malla de evaluación de densidad electrónica:

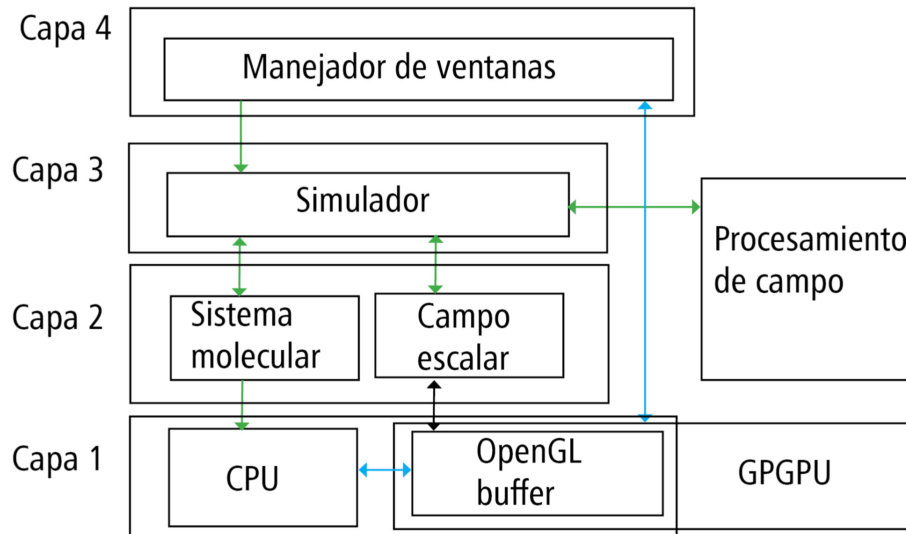



Figura 3.2: Arquitectura del *software* de visualización molS.

1. $x_0$,$y_0$,$z_0$: Representan la posición inicial en un espacio cartesiano donde se iniciará la evaluación.

2. $strideX$, $strideY$, $strideZ$: Indican la distancia entre cada posición de evaluación.

3. $nStrideX$, $nStrideY$, $nStrideZ$: Indican el paso con el parámetro nStride para $X$, $Y$ y $Z$.

4. $\rho$: Cada posición $X$, $Y$, $Z$ de la malla tiene un valor de densidad electrónica (Vea Figura 3.4)
   $(\rho_0,\ ...,\ \rho_i,\ \ldots,\ \rho_{nStrideX*nStriideY*nStrideZ}\ )$.

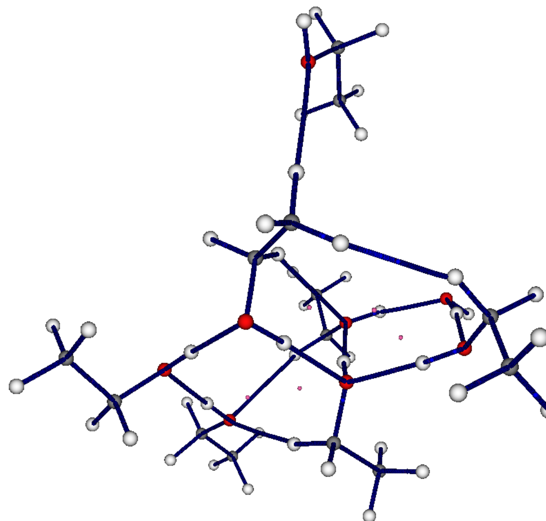
5. El vector describe la malla en el siguiente orden de posiciones (Vea Figura 3.5):
   $(x_0, y_0, z_0)$, $(x_0, y_0, z_1)$, $\ldots$, $(x_0, y_0 z_n)$, $(x_0, y_1, z_0)$, $(x_0, y_2, z_0)$, $\ldots$, $(x_n, y_n, z_n)$

6. Las posiciones tendrán las distancias dadas por los vectores de translación
   $(0, 0, nStrideZ)$, $(0, nStrideY, 0)$ y $(nStrideX, 0, 0)$.

La información del vector es utilizada por la clase *ScalarField* (ver Figura 3.6), que es la encargada de proveer al componente Simulador de un arreglo de posiciones para visualizar la isosuperficie que describe el campo escalar, en este caso densidad electrónica.

| SystemMolecular |
| --- |
| numAtoms: long int<br>atomsSystemMolecularStruct: float *<br>atomsSystemMolecular: float *<br>nameSystemMolecular: char * |
| SystemMolecular(char *)<br>loadAtomsDefinitions(void)<br>mallocAtoms(void)<br>mallocSystemMolecular(void)<br>getAtomsSystemMolecular(void): float *<br>getAtomsSystemMolecularColor(void): float * |

(a) Clase *SystemMolecular*



(b)

Figura 3.3: Componente *SystemMolecular*.

| $\rho_0$ $x_0y_0z_0$ | $\rho_1$ $x_0y_0z_1$ | ... | $\rho_\iota$ $x_iy_jz_k$ | ... | $\rho_{nStrideX*nStrideY*nStrideZ}$ $x_ny_nz_n$ |
| --- | --- | --- | --- | --- | --- |

Figura 3.4: Vector de densidad electrónica

En la Figura 3.6.b. se presenta el tipo isosuperficies que se visualizarán con la información almacenada.

## 3.4. Descripción de componentes principales de molS

En la Figura 3.8, se muestra un diagrama de entidades principales que integran molS, para explicar la descomposición funcional del *software* desarrollado.

### 3.4.1. Componentes de la Capa 1: Procesamiento

Las clases *ScalarFieldGPU* y *ScalarFieldCPU*, reciben el arreglo numérico de evaluación de la densidad electrónica. La Capa 1 es la encargada de realizar operaciones como la asignación de memoria utilizando las funciones *malloc* y *cudaMalloc*. En esta capa se realizan el

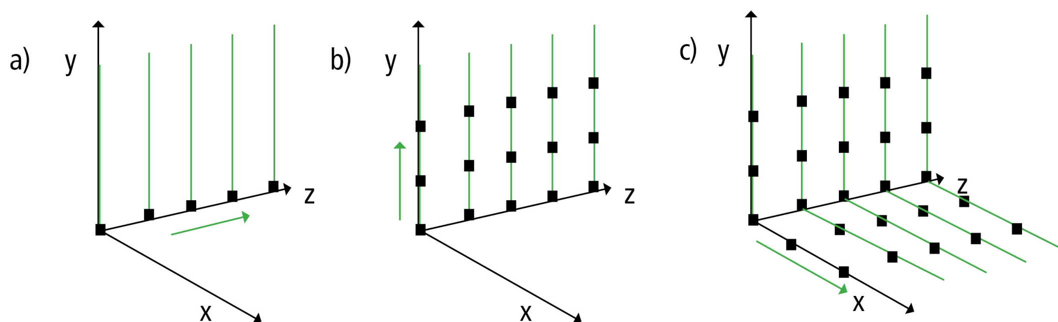



Figura 3.5: Ilustración de los pasos que se siguen para generar la malla del visualizador molS

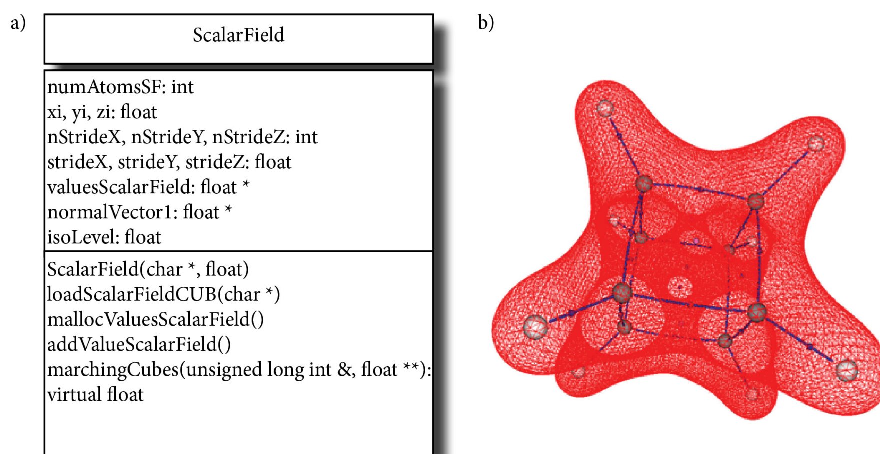



Figura 3.6: a) Clase ScalarField b) isosuperficie que describe la densidad electrónica.

lanzamiento de los *kernels* para procesar la malla en *marching cubes*. Para implementar el algoritmo *marching cubes* en diferentes tipos de unidades de procesamiento o incluso utilizar diferentes bibliotecas, se puede realizar herencia sobre la clase *ScalarField*. Para este trabajo de investigación se realizó una implementación sobre CPU y otra sobre GPU.

### 3.4.2. Implementación de algoritmo *marching cubes*

La implementación del algoritmo *marching cubes* está basado en el análisis de todos los vértices de un cubo y la adquisición de isosuperficies alrededor de los vértices. El análisis que se realiza sobre un cubo se basa en un isovalor que describa una isosuperficie. El primer paso para iniciar con al algoritmo *marching cubes* es analizar cubo por cubo, en cada cubo se visita cada uno de sus vértices, buscando sea menor el valor al isovalor buscado. Para

```
Agua
Densidad electronica total



3   4.000000   5.474484   4.862068
40    0.200000    0.000000    0.000000
56    0.000000    0.200000    0.000000
50    0.000000    0.000000    0.200000
8    8.000000   0.000000    0.000000    0.215517
1    1.000000   0.000000    1.474485   0.862068
1    1.000000   0.000000   1.474485   0.862068
4.194007e10 6.802469e10 1.071454e09 1.638897e09 2.434454e09
3.511732e09 4.919417e09 6.692324e09 8.841203e09 1.134274e08
1.413179e08 1.709809e08 2.008955e08 2.292269e08 2.539992e08
```

Figura 3.7: Ejemplo del archivo molécula.cub utilizado para la visualidor molS.

interpretar un cubo o *marching cube* se deben numerar los vértices y las aristas (ver Figura 3.10). La numeración de los vértices ayudará a identificar en cual de los 256 casos posibles se encuentra el cubo analizado. Para identificar el caso asociado al cubo analizado, se puede seguir dos posibles caminos:

- Procesar los vértices del cubo con un algoritmo inteligente para que determine cuantas isosuperficies calculará y la dirección en la que se calcularán las normales.

- Utilizar las tablas definidas en [23], que implica visitarlas con base en un índice para identificar el caso.

El tiempo de procesamiento sobre un solo *marching cube* puede variar, debido a que puede calcularse de 0 a 5 facetas[1]. Si se añade al cálculo de los vértices de cada faceta, el procesamiento para encontrar qué aristas forman cada triángulo y que una malla tiene un número potencia cúbica de *marching cubes*, el tiempo de procesamiento total puede dispararse. Por lo que la segunda opción resulta atractiva, además que puede pensarse en visualizar mallas con un *stride* entre evaluación muy pequeña, lo que hace posible tener una isosuperficie con facetas muy pequeñas que describirán con *suavidad* una isosuperficie.

Para encontrar a qué caso corresponde el cubo analizado, se realiza lo siguiente: Se visita cada vértice, desde el 0 al 7, comparando el valor de densidad electrónica $\rho$ en cada vértice

[1]Una faceta es un polígono que ayuda a describir una isosuperficie.

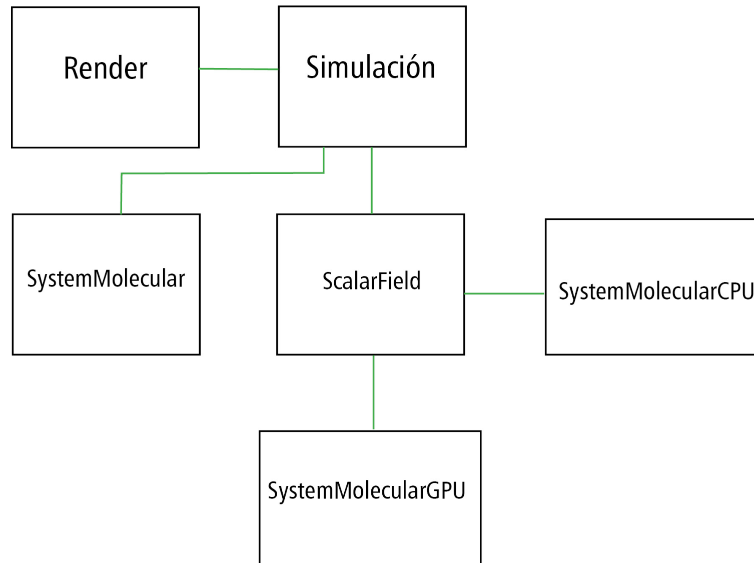



Figura 3.8: Diagrama de clases del visualizador molS.

con el isovalor buscado.

```
1 for i=0 to 7
2     index += cube[i] > isovalor ? 2^i : 0;
```

Este recorrido entregará valores entre 0 y 255, que son las combinaciones de *marching cubes* existentes. Calculado el valor *index* del *marching cube*, se consulta la Tabla 3.1 para obtener la combinación de aristas donde se calcularán los vertices de cada faceta (Tabla 3.1).

La Tabla 3.1 describe el orden en que deben procesarse las aristas para poder calcular las normales correspondientes a cada faceta (En el arreglo *triTable* del Apéndice A están todos los casos de *marching cubes*).

Al finalizar el procesamiento de toda la malla, se obtienen dos vectores, que corresponden a los vértices de las facetas y a los vectores normales de cada faceta. Cabe mencionar que para calcular las posiciones de los vértices de las facetas, se realiza una interpolación (Figura 3.11, 3.12) sobre cada arista involucrada. La implementación del algoritmo *marching cubes*, requiere de tres estructuras :

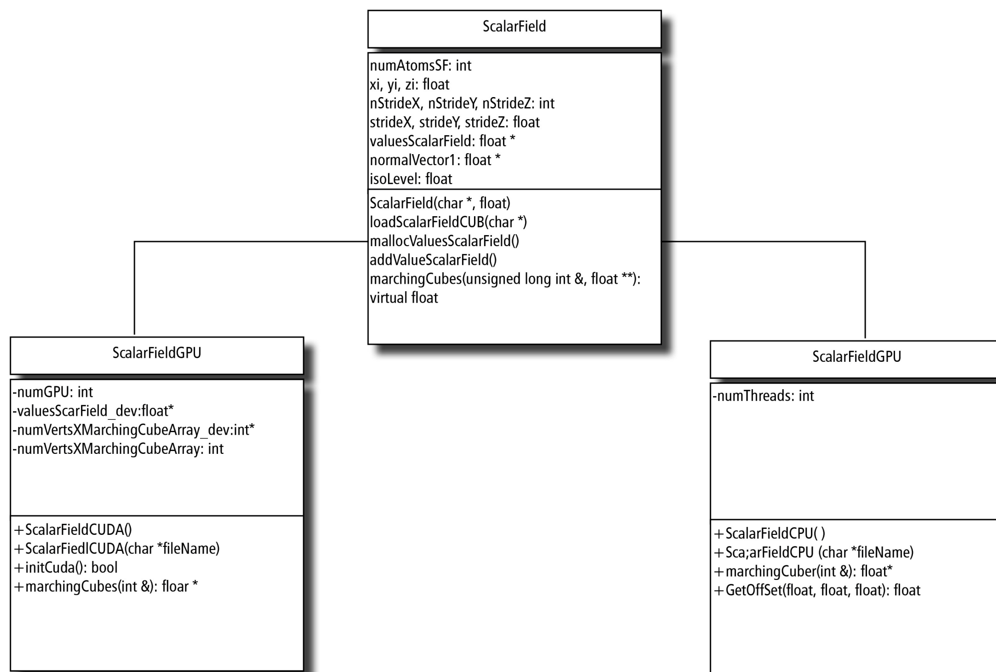



Figura 3.9: Clases ScalarFieldCPU y ScalarFieldGPU que heredan de la clase ScalarField.

1. triTable: Colección de las aristas de los 256 casos de *marching cubes*.

2. numVertsTable: Este arreglo almacena la cantidad de vertices que aportará cada *marching cubes*.

3. edgeConnection: Almacena la lista de conexiones (vertice-a-vertice) de las 12 aristas de un *marching cubes*.

En el Apéndice A puede encontrar los arreglos $triTable$, $numVertsTable$, $edgeConnection$.

### 3.4.3. Componentes de la Capa 2: Sistema molecular y Campo escalar

El primer componente de la Capa 2 se denomina Sistema molecular y se implementó a través de la clase "SystemMolecular" escrita en el lenguaje C. Este componente solicita información referente a la geometría del sistema molecular a visualizar, tal como: cantidad de átomos, posición de cada átomo, tipo de átomo y puntos característicos. Esta información es utilizada por la clase SystemMolecular para cumplir con su función de proveer al componente

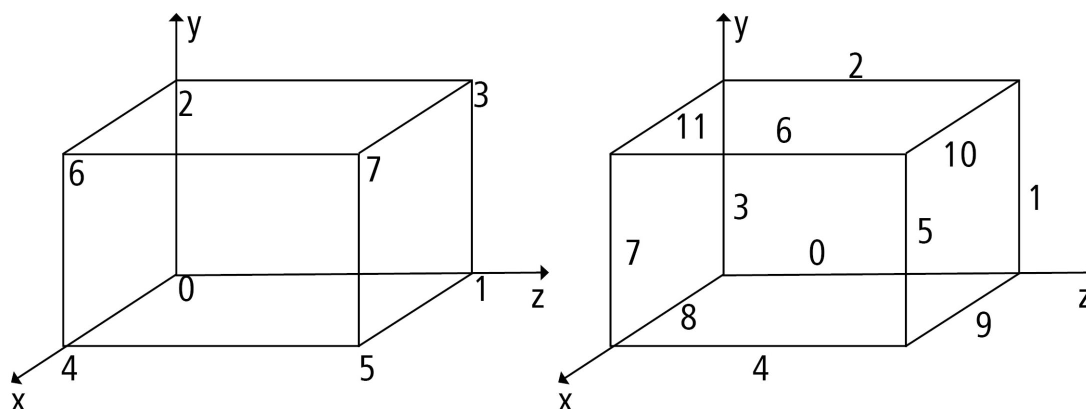



Figura 3.10: Numeración de los vértices y aristas de un *marching cube*

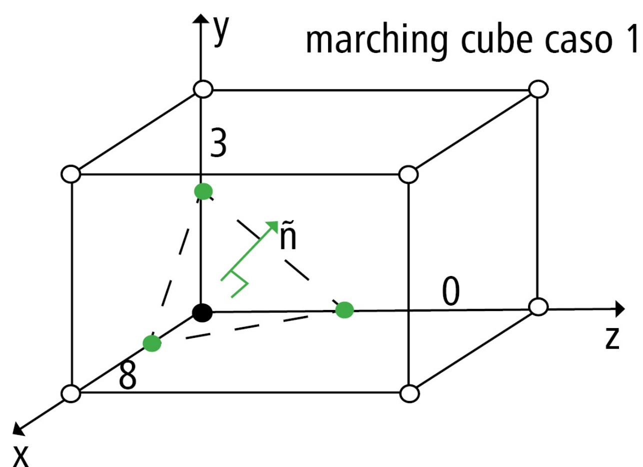



Figura 3.11: Ejemplo de marching cube, caso 1.

Simulador un arreglo de posiciones y tipos de átomos a renderizar. En la Figura 3.3(a).b se presenta el tipo de estructuras que almacena esta clase. En el ámbito de las aplicaciones para química cuántica, se utiliza el tipo de archivos *xyz* (ver Figura 3.3(b).a) donde se puede almacenar de forma persistente información referente a la geometría de un sistema molecular. El segundo componente de la Capa 2 se denomina Campo Escalar y se implementó a través de la clase "ScalarField" escrita en el lenguaje C.

Tabla 3.1: Casos: *marching cubes*

| vértices > isovalor | triángulos (aristas) |
|---|---|
| 0 | $-$ |
| $1(vertice = 0 > isovalor)$ | $0-8-3$ |
| $2(vertice = 1 > isovalor)$ | $0-1-9$ |
| $3(vertice = 0,1 > isovalor$ | $1-8-3, 9-8-1$ |
| ... | ... |
| $100(vertice = 2,5,6 > isovalor)$ | $1-4-9, 1-2-4, 2-6-4$ |
| ... | ... |
| 253 | $0-9-1$ |
| 254 | $0-3-8$ |
| 255 | $-$ |

### 3.4.4. Componentes de la Capa 3: Simulador

La clase Simulador es la encargada de gestionar el comportamiento de la aplicación, con la ayuda de un pequeño módulo de análisis determina cual será el comportamiento. La aplicación molS es capaz de trabajar de manera independiente de los módulos externos de procesamiento numérico, utilizando archivos de entrada, así como también tiene la habilidad para manipular apuntadores para visualizar en tiempo real los resultados de los módulos externos. Entre las tareas que realiza podemos destacar la comunicación de los resultados del procesamiento numérico necesario para tratar la malla y visualizar la isosuperficie.

### 3.4.5. Componentes de la Capa 4: Manejador de Ventanas

El primer componente del *software* molS esta compuesto por la clase Render, esta clase es la encargada de la interacción de los usuarios con los componentes del *software*. La inicialización del ambiente gráfico, el comportamiento del espacio para las rotaciones, traslaciones y el escalamiento, es parte del trabajo que realiza este componente. Si bien el tratamiento de las tareas antes mencionadas son esenciales para el comportamiento de la aplicación molS, el trabajo más importante de esta clase se da en las operaciones para poner los vectores de

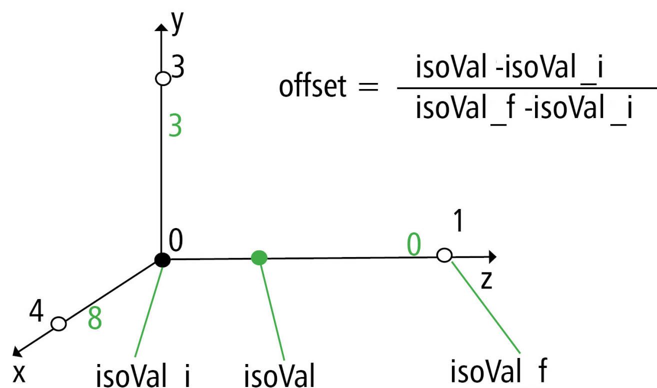



Figura 3.12: Cálculo referente a un vértice de una faceta *marching cubes*

128

Agua File Critical Points in Amstrong

| | | | |
|---|---|---|---|
| O | 0.000000 | 0.000000 | 0.215517 |
| H | 0.000000 | 1.474485 | 0.862068 |
| H | 0.000000 | 1.474485 | 0.862068 |
| PCE | 0.000000 | 1.108291 | 0.603216 |
| PCE | 0.000000 | 1.108291 | 0.603216 |

Figura 3.13: Ejemplo del archivo molecula.xyz utilizado para el visualizador molS

datos en el buffer de OpenGL. Las operaciones de GenBuffer, BindBuffer, DrawArray, etc son cruciales para el buen desempeño de la aplicación.

## 3.5. Implementación de los componentes del *software* molS

En el subcapítulo anterior se describió la funcionalidad de las cuatro capas que componen el *software* molS. Durante la descripción de la funcionalidad de las capas se mencionaron las clases más importantes que las componen. A continuación, se describe la manera en que se

implementaron para mostrar la interoperabilidad de OpenGL, CUDA y GLSL.

**La clase Render:** Está basada en los siguientes *callbacks* (Código 3.1, que son necesarios para lograr la interactividad del usuario con el entorno visual.

```
1 glutDisplayFunc(display);
2 glutReshapeFunc(reshape);
3 glutMouseFunc(mouse);
4 glutMotionFunc(motion);
5 glutKeyboardFunc(key);
6 glutMainLoop();
```

Código 3.1: callbacks para manejo de ventanas con GLUT

Además utiliza las funciones gráficas para lograr una visualización correcta y suave; la biblioteca gráfica debe inicializarse con funciones y parámetros del Código 3.2.

```
1 glutInitDisplayMode(GLUT_RGBA | GLUT_DEPTH | GLUT_DOUBLE);
2 glBlend(ENABLED);
3 glBlendFunc (GL_DST_ALPHA, GL_ONE_MINUS_DST_ALPHA);
4 glEnable(GL_DEPTH_TEST);
5 glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```

Código 3.2: Inicialización de OpenGL

El sistema molecular, campo escalar y vectorial son almacenados en arreglos de tipo flotantes, estos arreglos deben ser cargados a los *Vertex Buffer Objects* para ser manejados directamente sobre la tarjeta gráfica. El código 3.3 es el necesario para los arreglos que almacenan los vértices que describen una isosuperficie con sus respectivas normales.

**La clase Simulación** tiene como atributos los componentes a visualizar; en esta clase se lleva el control de los componentes que se desea visualizar, la visualización puede ser

```
glGenBuffers(1, &vboTriangles);
glBindBuffer(GL_ARRAY_BUFFER, vboTriangles);
glBufferData(GL_ARRAY_BUFFER, 3 * sizeof(float) * nVertices, 0,\
GL_DYNAMIC_DRAW);
glBindBuffer(GL_ARRAY_BUFFER, 0);
glBindBuffer(GL_ARRAY_BUFFER, vboTriangles);
glBufferSubData(GL_ARRAY_BUFFER, 0, 3 * nVertices * sizeof(float),\
vertices);
glBindBuffer(GL_ARRAY_BUFFER, 0);
glGenBuffers(1, &vboNormals);
glBindBuffer(GL_ARRAY_BUFFER, vboNormals);
glBufferData(GL_ARRAY_BUFFER, sizeof(float) * (nVertices / 3), 0,\
GL_DYNAMIC_DRAW);
glBindBuffer(GL_ARRAY_BUFFER, 0);
glBindBuffer(GL_ARRAY_BUFFER, vboNormals);
glBufferSubData(GL_ARRAY_BUFFER, 0, sizeof(float) * (nVertices / 3),\
normales);
glBindBuffer(GL_ARRAY_BUFFER, 0);
```

Código 3.3: Manejo de Vertex Buffer Object

combinada o bien de cada uno, además se encarga de recibir información desde módulos externos o desde archivos.

Una característica importante del *software* molS es que puede trabajar con información proveniente de los módulos de procesamiento externo (Linea 1: Código 3.4) o de manera independiente; para este control de información se utiliza la clase *Simulation*.

**La clase SystemMolecular**: Es la encargada de manipular la información de la geometría del sistema molecular a visualizar, esta clase prepara dos arreglos de valores: el primer arreglo tiene la posición en el espacio cartesiano de los átomos o trayectorias, además de la dimensión del radio de la esfera a visualizar. El segundo arreglo de valores tiene almacenado el color de cada átomo o trayectoria. Estos dos vectores son usados por la clase Render para visualizar el sistema molecular. El radio y color de cada átomo o puntos característicos de

```
SystemMolecular *systemMolecular;
ScalarField *scalarField;
VectorField *vectorField;
```

Código 3.4: Componentes manejados por la Clase Simulator.

los sistemas moleculares son definidos en el archivo *atoms* (Código 3.5), el formato de dicho archivo es: La primera línea establece el número de registros (átomos y puntos) registrados en el archivo, cada registro de átomo debe estructurarse con los siguientes campos: una etiqueta, el radio del átomo, y color del átomo en formato RGB.

El sistema molecular es cargado al arreglo *atomSystemMolecuar* y los colores de cada *áto-*

```
20
H                    1.520     255 255 255
Li                   1.520     204 128 255
He                   1.520     217 255   255
O                    1.520     255 013 013
N                    1.520     000 255 000
C                    1.520     144 144 144
...
PCE                  0.5       087 023 143
PCA                  0.5       255 105 180
PCC                  0.5       255 000 000
PCNN                 0.5       184 076 000
PT                   0.5       000 000 255
```

Código 3.5: Archivo de definiciones atoms.

*mo/punto* son cargados al arreglo *atomsSystemMolecularColor* (código 3.6) estos dos son atributos de la clase *SystemMolecular*. Dichos arreglos se envían a la clase Simulador, que a su vez los envía a la clase Render, donde se pasan a los Buffers de OpenGL y se visualizan.

**La clase ScalarField**: Es la encargada de manipular el arreglo que almacena la densidad electrónica en la malla (Figura 3.14). Dentro de esta clase se tienen los parámetross necesarios

```
1 r = SystemMolecular::getRadio(sTmp);
2 atomsSystemMolecular[i * 4 + 0] = x;
3 atomsSystemMolecular[i * 4 + 1] = y;
4 atomsSystemMolecular[i * 4 + 2] = z;
5 atomsSystemMolecular[i * 4 + 3] = r * 0.152917721;
6 SystemMolecular::getColor(sTmp, Color);
7 atomsSystemMolecularColor[i * 4 + 0] = Color [0] / 255.f;
8 atomsSystemMolecularColor[i * 4 + 1] = Color [1] / 255.f;
9 atomsSystemMolecularColor[i * 4 + 2] = Color [2] / 255.f;
10 atomsSystemMolecularColor[i * 4 + 3] = 1.f;
```

Código 3.6: Componentes manejados por la Clase Simulator.

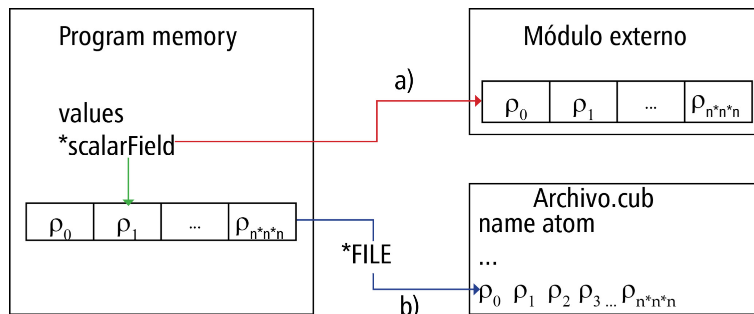



Figura 3.14: El arreglo que almacena la densidad electrónica es a) apuntado a los módulos externos de procesamiento b) cargado desde un archivo *.cub*.

para describir la malla de evaluación, además se encarga de solicitar a la clase de simulación el isovalor para visualizar el campo escalar. La clase ScalarFieldCUDA implementa el algoritmo *marching cubes* utilizando dispositivos de procesamiento gráfico, esta clase calcula dos arreglos: el primer arreglo tiene todos los vértices que describirán una isosuperficie y el segundo arreglo los vectores normales a cada una de las facetas de la isosuperficie. La cantidad de facetas que describirán una isosuperficie es díficil de predecir por lo que el primer *kernel* (Código 3.7) de procesamiento sobre la malla debe calcular la cantidad de facetas que se tendrán, esto para poder reservar el arreglo exacto de vértices que forman la isosuperficie a visualizar. Resultaría sencillo reservar memoria considerando la cantidad máxima de facetas (5) con que puede contribuir cada cubo, en el caso de mallas con muchos cubos, necesitaría reservarse

el total de cubos multiplicado por 15 posiciones de tipo real para almacenar 5 facetas por cubo. Esto último puede funcionar para sistemas pequeños, en el caso de sistemas medianos o grandes los recursos de memoria RAM y memoria global en la GPU deberían ser de gran magnitud, lo que eliminaría el uso del *software* en equipos de cómputo tipo *workstation*.
Para lanzar el *kernel* que calcula el número de facetas para la isosuperficie con el isovalor dado, el arreglo de la densidad electrónica (Figura 3.15) debe estar presente en la memoria global de la GPU.

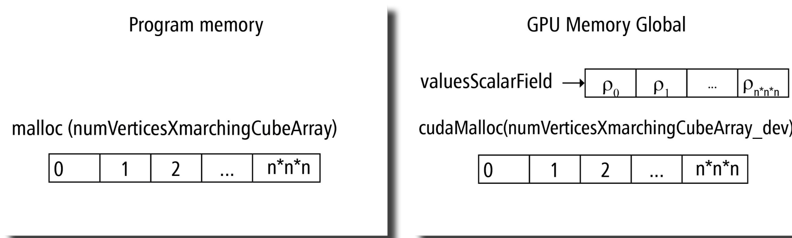
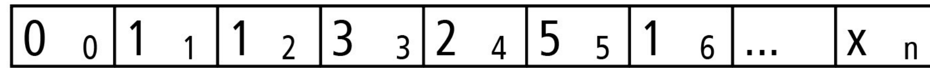



Figura 3.15: El arreglo *vectorScarlarField* en la GPU.

```
1 calculeMarchingCubeCUDA_kernel<<<dimGrid,dimBlock>>>(valuesScalarField_dev,\
2 numVertsXMarchingcubeArray_dev,getIsoLevel(),nStrideX,nStrideY,nStrideZ,\
3 strideX,strideY,strideZ);
```

Código 3.7: *kernel* para calcular número de facetas

Al finalizar la ejecución del *kernel*, el apuntador $numVertsXMarchingcubeArray_dev$ apunta a un arreglo como el de la Figura 3.16(a), este vector habrá que procesarlo para que tome la forma del arreglo de la Figura 3.16(b). Al tomar esta forma, cada hilo del siguiente *kernel* tendrá la posición donde iniciar a almacenar los vertices de las facetas de cada *marching cube*. Procesado el arreglo $numVertsXMarchingcubeArray_dev$, la posición $n * n * n$ tendrá la cantidad de vertices que describirán la isosuperficie, la cual debemos reservar en el dispositivo de procesamiento $(cudaMalloc((void)\&vertsIsoSurface_dev, n * n * n * 3 * sizeof(float)))$.




(a)

(b)

Figura 3.16: Arreglo de posicionamiento de facetas.

El arreglo *vertsIsoSurface_dev* almacenará todos los vertices que compondran los poligonos(facetas) de la isosuperficie, cada vertice tiene tres componentes $(x, y, z)$.
El *kernel marchingCubeCUDA_kernel* (Código 3.8) calculará en cada *marching cube* las posiciones de los vértices y los almacenará en el arreglo *vertsIsoSurface_dev*.
Los *kernel* que se han implementado para procesar la isosuperficie deben lanzar una cantidad de hilos igual o mayor (Figura 3.17 y 3.18) al número de *maching cubes* en la malla, para ellos es importante definir las dimensiones de bloque y grid que tendrá los parámetros para lanzar dichos *kernel*(Vea Código 3.9).

En gran parte, el buen rendimiento del algoritmo *Marching Cubes* es debido al uso de Tablas

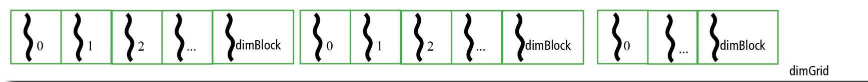



Figura 3.17: *kernel* de tamaño $dimBlock * dimGrid$

con las combinaciones de las facetas con que cada caso de cubo contribuye. La memoria de texturas de GPU resulta ser la mejor opción para precargar dichas tablas, esta decisión de implementación mejora el rendimiento de la aplicación molS. El Código 3.10 es el encargado de precargar dichas tablas a memoria de texturas de la unidad de procesamiento gráfico, este código debe ejecutarse antes de lanzarse cualquier kernel para procesar las isosuperficies, además que solo se carga una sola vez.

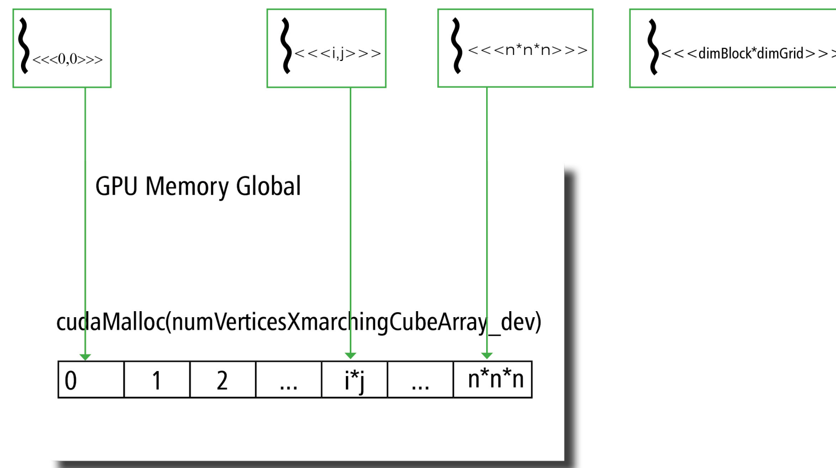



Figura 3.18: Cada *thread* procesa un *marching cube* y almacena sobre una posición de un arreglo.

```
1  marchingCube = threadIdx.x + blockIdx.x * blockDim.x;
2  posI0SaveVerts = numberVert_dev[marchingCube];
3  cpf = nStrideZ - 1;
4  cpp = cpf * (nStrideY - 1);
5  ppTmp = nStrideY * nStrideZ;
6  fila = marchingCube / cpf;
7  piso = nStrideZ * (marchingCube / cpp);
8  cub[8] = {
9  marchingCube + fila + piso, marchingCube + fila + piso + 1,
10 marchingCube + fila + piso + nStrideZ + 1, marchingCube + fila + piso + nStrideZ,
11 marchingCube + fila + piso + ppTmp, marchingCube + fila + piso + ppTmp + 1,
12 marchingCube + fila + piso + nStrideZ + ppTmp + 1, marchingCube + fila + piso + nStrideZ +
       ppTmp
13 };
14 for( recorrido = 0; recorrido < 8; recorrido ++)
15   if( valuesScalarField_dev[ cub [ recorrido ] ] < isoLevel )
16     cubeIndex |= pow_dev(2, recorrido);
17 cubeIndex = 255 - cubeIndex;
18 if( tex1Dfetch(edgeTable_Tex, cubeIndex) != 0 && marchingCube < cpp * (nStrideX - 1)){
19   for( index = 0; index  < tex1Dfetch(numVertsTable_Tex, cubeIndex); index ++){
20     iEdge = tex1Dfetch(triTable_Tex, cubeIndex * 16 + index);
21     i0 = cub[ tex1Dfetch(edgeConnection_Tex, iEdge * 2 + 0) ];
22     i1 = cub[ tex1Dfetch(edgeConnection_Tex, iEdge * 2 + 1) ];
23     isoVal0 = valuesScalarField_dev[ i0 ];
24     isoVal1 = valuesScalarField_dev[ i1 ];
25     z0 = i0 % nStrideZ; y0 = (i0 / nStrideZ) % nStrideY;
26     x0 = i0 / (nStrideZ * nStrideY); z1 = i1 % nStrideZ;
27     y1 = (i1 / nStrideZ) % nStrideY; x1 = i1 / (nStrideZ * nStrideY);
28     x0 = xi + x0 * strideX; y0 = yi + y0 * strideY; z0 = zi + z0 * strideZ;
29     x1 = xi + x1 * strideX; y1 = yi + y1 * strideY; z1 = zi + z1 * strideZ;
30     float fOffset = fGetOffSet_dev( isoVal0, isoVal1, isoLevel);
31     if ( fOffset == 0.0 ) continue;
32     x0 = x0 + (x1 - x0) * fOffset; y0 = y0 + (y1 - y0) * fOffset;
33     z0 = z0 + (z1 - z0) * fOffset;
34     vertsIsoSurface_dev[posI0SaveVerts * 3 + 0] = x0;
35     vertsIsoSurface_dev[posI0SaveVerts * 3 + 1] = y0;
36     vertsIsoSurface_dev[posI0SaveVerts * 3 + 2] = z0;
37     posI0SaveVerts ++;
38     }
39 }
```

Código 3.8: *kernel* para calcular facetas de todos los *marching cubes*

```
1 unsigned long int numberMarchingCubes = (nStrideX -1) * (nStrideY - 1) * (nStrideZ - 1);
2 unsigned long int numberThreads = numberMarchingCubes;
3 unsigned long int threads = 1024;
4 unsigned long bloques = (numberThreads - 1) / threads + 1;
5 dim3 dimGrid(bloques, 1, 1);
6 dim3 dimBlock(threads, 1 ,1);
```

Código 3.9: Código para calcular las dimensiones del *kernel*

```
1 cudaMalloc((void **) &numVertsTable_dev, 256 * sizeof(int));
2 cudaMemcpy(numVertsTable_dev, numVertsTable, 256 * sizeof(int),\
3 cudaMemcpyHostToDevice);
4 cudaBindTexture(NULL, numVertsTable_Tex, numVertsTable_dev, 256 * \
5 sizeof(int));
6 cudaMalloc((void **) &triTable_dev, 256 * 16 * sizeof(int));
7 cudaMemcpy(triTable_dev, triTable, 256 * 16 * sizeof(int),\
8 cudaMemcpyHostToDevice) );
9 cudaBindTexture(NULL, triTable_Tex, triTable_dev, 256 * 16 * \
10 sizeof(int));
11 cudaMalloc((void **) &edgeConnection_dev, 12 * 2 * sizeof(int));
12 cudaMemcpy(edgeConnection_dev, edgeConnection, 12 * 2 * sizeof(int),\
13 cudaMemcpyHostToDevice) );
14 cudaBindTexture(NULL, edgeConnection_Tex, edgeConnection_dev, 12 * 2 *\
15 sizeof(int));
16 cudaMalloc((void **) &edgeTable_dev, 256 * sizeof(int));
17 cudaMemcpy(edgeTable_dev, edgeTable, 256 * sizeof(int),\
18 cudaMemcpyHostToDevice);
19 cudaBindTexture(NULL, edgeTable_Tex, edgeTable_dev, 256 *\
20 sizeof(int));
```

Código 3.10: Carga de tablas de referencia para maching cubes en GPU

# Capítulo 4

# Resultados

## 4.1. Acoplamiento de núcleo matemático y procesamiento visual

Una parte importante del quehacer de la química cuántica es la solución de la ecuación de Schrödinger o el uso de la teoría de funcionales de la densidad para la obtención de la densidad electrónica. Sin embargo, otra parte importante de esta disciplina es la visualización de la densidad electrónica o de los orbitales que la constituyen tanto de átomos como de moléculas. En este trabajo de investigación, además de la densidad electrónica se ha evaluado el gradiente o el laplaciano de la densidad electrónica, la función de localización electrónica, el gradiente de densidad reducido y otros campos de interés para la comunidad química. La aplicación no se límta a la visualización de la densidad electrónica, que es a la que se hace referencia a lo largo de la discusión. La primera parte de este proyecto fue la de visualizar la posición de los átomos que conforman un sistema molecular. En la Figura 4.1 se presentan dos moléculas de agua.

La visualización de la posición de los átomos en estas moléculas quita en menor medida la intuición química que nos proporciona una gráfica molecular. Es importante mencionar que la conectividad entre los átomos de una molécula es un invento de los químicos y con esa intuición muchos de los programas de visualización trabajan. Sin embargo, existe un modelo donde se puede encontrar la conectividad entre los átomos. La Teoría cúantica de átomos en

Figura 4.1: Representación de dos moléculas de agua sin conecciones entre los átomos

moléculas (QATIM), está basado en encontrar los puntos críticos de la densidad electrónica, [3] esto es

$$\nabla\rho(\vec{r}) = \vec{0}. \tag{4.1}$$

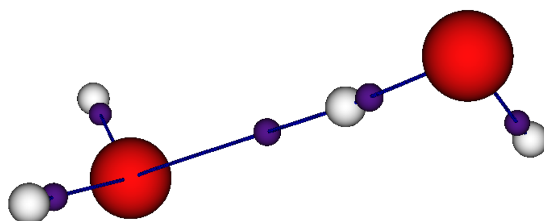


Figura 4.2: Representación de dos moléculas de agua con conecciones entre los átomos

A partir de estos puntos se buscan las trayectorias donde la densidad electrónica se maximiza (sobre las posiciones de los núcleos) y así es como se puede encontrar la conectividad entre los átomos en una molécula. Este proceso fue acelerado con la ayuda de una GPU en nuestra aplicación y fue motivo de publicación en la revista *Journal of Computational Chemistry* [11]. Acoplando este proceso a la aplicación, se obtiene la molécula de agua con sus conexiones como se muestra en la Figura 4.2.

Es evidente de esta figura que cambia radicalmente el aspecto de una gráfica molecular. En este caso, las esferas azules(oscuras, vea Figura 4.2) representan los puntos críticos de la densidad electrónica. Además es claro que hay interacción entre las dos moléculas de agua porque existe un punto crítico de enlace y a partir de éste se generan las trayectorias de enlace que unen a las dos moléculas.

Naturalmente, la aplicación funciona también con sistemas de tamaño considerable, de hecho todo el esfuerzo realizado es para trabajar con sistemas de centenas o miles de átomos. En la Figura 4.3 se muestra un ejemplo donde la gráfica molecular implica ya un esfuerzo computacional al graficar las trayectorias de enlace de una molécula que representa una fracción de una proteína .

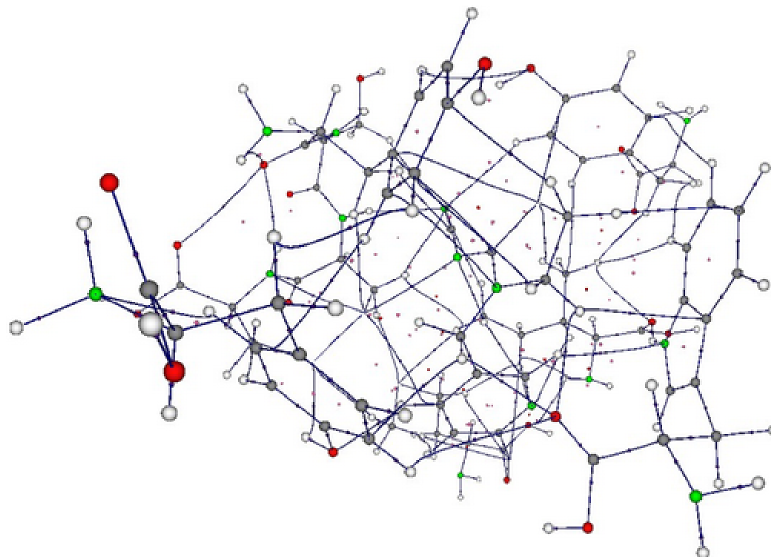


Figura 4.3: Gráfica molecular de una fracción de una proteína.

## 4.2. Despliegue de campos escalares

Una vez que se visualiza la posición y los puntos críticos que conforman un sistema molecular y sus enlaces; el siguiente paso es envolver al sistema en un malla tridimensional, en particular, en un paralelepípedo, y evaluar sobre cada punto de la malla al campo escalar. En el caso de la densidad electrónica se usa la ecuación 2.1. Como ya se mencionó, el algoritmo marching cubes se utiliza para la extracción de una malla poligonal que aproxima una iso-superficie de un campo escalar tridimensional. La Figura 4.4 se representa el campo escalar (iso-superficie), que representa a la densidad electrónica de una molécula de hidrógeno. Este fue uno de los primeros resultados alcanzados por este trabajo de investigación. La represen-

tación se obtuvo haciendo el procesamiento y visualización sobre la GPU a partir del arreglo de densidad electrónica con la ayuda del algoritmo marching cubes. Con la construcción de la aplicación molS se logra la visualización de dichos campos escalares y sistemas moleculares. La escalabilidad de nuestro visualizador se verá limitado por la memoría global del dispositivo de procesamiento gráfico que se tenga disponible. Para visualizar una isosuperficie se debe tener en cuenta el tamaño(bytes) que tienen los siguientes arreglos:

1. densidad electrónica $(nStrideX * nStrideY * nStrideZ * sizeof(float))$.

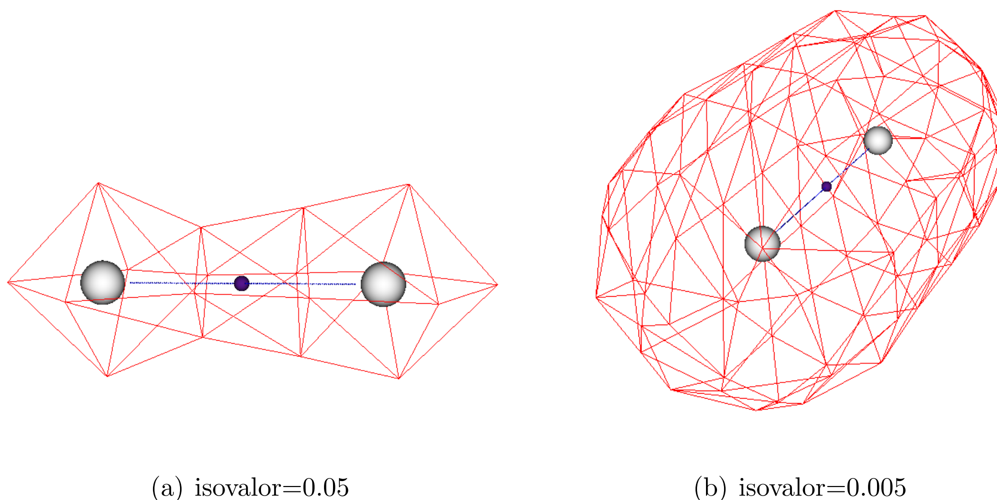
2. posiciones de facetas por marching cubes $((nStrideX-1)*(nStrideY-1)*(nStrideZ-1)*sizeof(shortint))$.

3. vertices de iso superficie $(3 * totalFacetas(isovalor) * sizeof(float)$.

4. normales a las facetas $(totalFacetas(isovalor) * sizeof(float)$.

El total de vértices y normales de la iso superficie se calcula con el *kernel* 3.7.

El campo escalar es procesado bajo el enfoque de “divide y vencerás” (*marching cube por marching cube de manera simultanea ó paralela*) para encontrar los polígonos que conforman la isosuperficie. molS lográ la visualización de campos escalares mas allá de veintidos millones de marching cubes gracias a la interación entre la memoria global de la GPU y los Buffer de OpenGL.



(a) isovalor=0.05 (b) isovalor=0.005

Figura 4.4: Visualización de la densidad electrónica de la molécula $H_2$.

## 4.3. Programación eficiente sobre los hilos de la tarjeta gráfica

Si bien las capacidades de visualización están disponibles en unidades de procesamiento actuales, son las unidades de procesamiento gráfico las que disponen de la capacidades para poder visualizar conjuntos grandes de datos, a esta capacidad se le conoce como aceleración gráfica. En el Capítulo 2 se hablaba de que estas unidades fueron diseñadas para realizar procesamiento visual exclusivamente. Sin embargo, en los últimos años se ha permitido el procesamiento de datos básicos (*float, int, long, etc*). Esto permite mejorar el procesamiento siempre y cuando el problema soporte la técnica SIMD. Debido a que en este trabajo de investigación requerimos procesar campos escalares, el uso de unidades de procesamiento gráfico resulta de brindar un rendimiento de hasta 10 veces mayor, respecto al uso de CPUs. Para soportar esta afirmación, a continuación se enlistan los resultados obtenidos con algunos sistemas moleculares.

Tabla 4.1: Tiempos requeridos para generar una isosuperficie en tres sistemas moleculares

| Sistema molecular | $H_2$ | Agua | Cubano | $\beta$-ciclodextrina |
|---|---|---|---|---|
| Número de átomos | 2 | 3 | 16 | 138 |
| Total de puntos de densidad electrónica | 768 | 768 | 8400000 | 2323200 |
| Total de cubos en la malla | 539 | 539 | 8276609 | 22984269 |
| isovalor | 0.05 | 0.01 | 0.1 | 0.1 |
| Tiempo CPU | 1.us | 1.s | 0.77s | 2.28s |
| Tiempo GPU | 60.4us | 1.3ms | 84ms | 279ms |

De este primer resultado es importante resaltar que la capacidad de obtener mediciones del tiempo empleado por la CPU para la visualización de esta molécula radica en la cantidad de datos a procesar, y que el parámetro para juzgar el tamaño de una malla está determinada por el número de cubos que tenga ésta. La cantidad de átomos no representa una medida para determinar si el sistema a visualizar es grande o pequeño, lo que lo determina es el número de

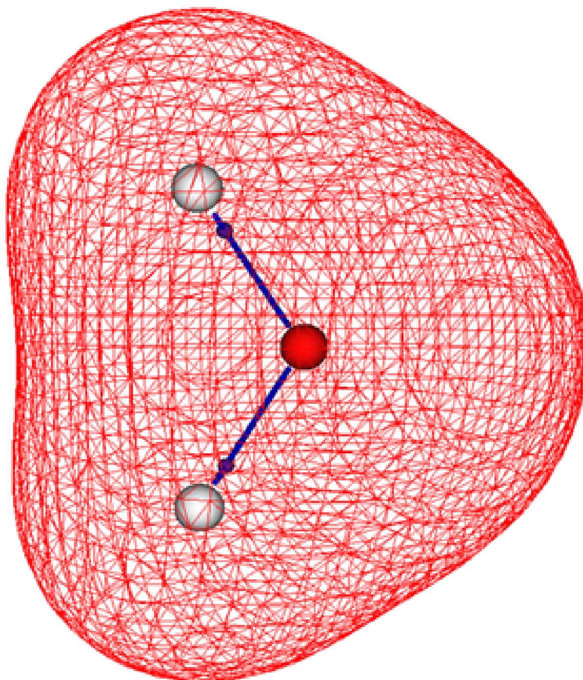


Figura 4.5: Visualización del sistema molecular $H_2O$.

funciones de base y la extensión en el espacio del sistema molecular. Las siguientes estructuras moleculares requieren de un procesamiento de aproximadamente sesenta y cuatro millones de operaciones de punto flotante. La molécula cubano (Figura 4.6) requiere de procesar 8,276,609 cubos para representar su densidad electrónica, cada cubo consta de 8 vértices, lo que significa que deben analizarse alrededor de sesenta y cuatro millones de puntos. Al *software* molS le toma 84 ms para generarla, que en comparación al procesamiento realizado con la CPU mejora el tiempo de procesamiento diez veces.

La molécula conocida como $\beta$-ciclodextrina (Figura 4.7) requiere de procesar aproximadamente veintitrés millones de cubos para representar su densidad electrónica. Al *software* molS le toma 279 ms para generarla, que en comparación al procesamiento realizado con la CPU mejora el tiempo de procesamiento diez veces, la visualización de este tamaño de malla con procesamiento sobre CPU requiere tiempo en el orden de segundos, lo que se refleja significativamente en la interactividad de la aplicación con el usuario. Aunque el uso de los buffers de OpenGL ayuda al desempeño en la visualización, el tiempo de procesamiento para visualizar la isosuperficie es significativamente alto.

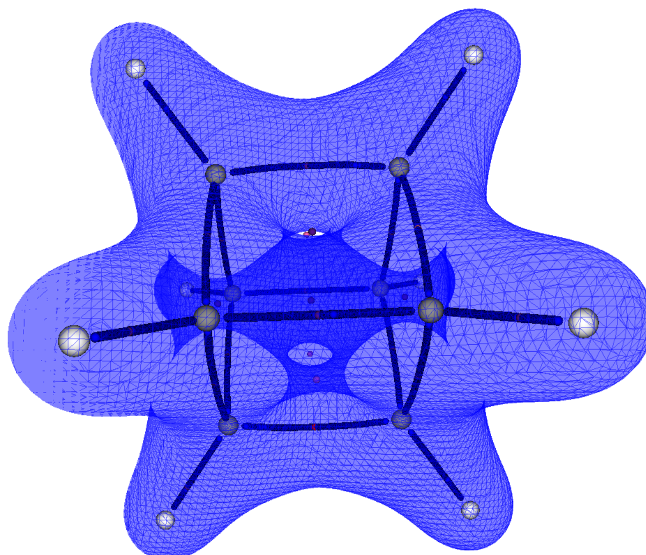


Figura 4.6: Visualización del sistema molecular Cubano.

Además de la densidad electrónica y los tres sistemas discutidos, se ha trabajado con otros campos y otras moléculas. En el Apéndice B se presenta una galería de algunos sistemas que han sido estudiados y que han servido para el análisis requerido en algunas apliaciones del área de Fisicoquímica teórica de la Universidad Autónoma Metropolitana-Iztapalapa.

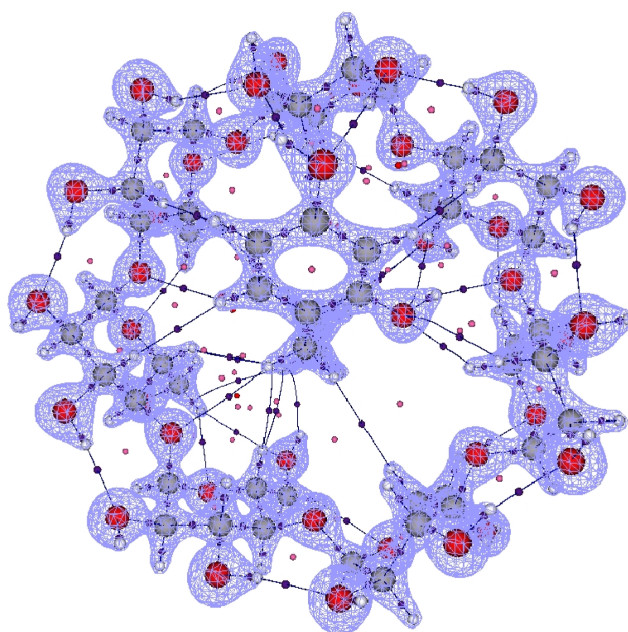


Figura 4.7: Visualización del sistema molecular $\beta$-ciclodextrina.

# Capítulo 5

# Conclusiones y trabajo futuro

La contribución del trabajo realizado en esta Tesis es proponer una solución eficiente para visualizar y calcular simultáneamente -en tiempo real- campos escalares y/o vectoriales, sobrepuestos en las geometrías moleculares. Por consiguiente, para alcanzar el objetivo de la Tesis se trabajó en la visualización en tarjetas gráficas (GPU) lo que implicó estudiar, la Arquitectura de Unificada de Dispositivos de Cómputo (CUDA), OpenGL, la interoperabilidad de OpenGL y CUDA y el Lenguaje de Sombreado de OpenGL (GLSL), para maximizar el rendimiento de la solución propuesta.

El modelo de programación de la arquitectura CUDA supone que los hilos se pueden ejecutar en una unidad de procesamiento gráfico que permite desarrollar cómputo de alto desempeño. Esta característica se explotó en este trabajo para alcanzar un rendimiento adecuado del software desarrollado molS, tanto en tiempo como en el tamaño de datos procesados. El *software* de visualización molS cumple con las características resumidas en la Tabla 5.1.

Con la implementación del visualizador moIS se logró en forma satisfactoria la visualización de campos escalares de la química cuántica, permitiendo reducir el tiempo y agilizar el proceso de análisis.

El software moIS fue diseñado por capas lo que permite que sea modificable, por lo que se ha identificado como trabajo futuro, refinar el conjunto de componentes que integran moIS mediante la experimentación en diferentes casos de uso.

Tabla 5.1: Aspectos de molS para la visualización de algunos campos de la QC

| | molS |
|---|---|
| Manejo de campos ecalares | Si |
| Manejo de campos vectoriales | Si |
| Se ejecuta en el *hardware* de | GPU |
| Disponible en el SO | Linux |
| Visualización de grandes moléculas (> 128 átomos) | Si |
| Visualización de pequeñas moléculas (< 64 átomos) | Si |
| Manejo de archivos de entrada/salida | Si |
| Interfaz de usuario | No |

# Bibliografía

[1] Alcalde Segundo. 2013. “Evaluación de campos escalares de la química cuántica sobre unidades de procesamiento gráfico”. Estado de México: Universidad Autónoma del Estado de México.

[2] Allouche, Abdul-Rahman. 2011. “Gabedit—A Graphical User Interface for Computational Chemistry Softwares“. Journal of Computational Chemistry 32 (1): 174–182. doi:10.1002/jcc.21600.

[3] Bader, R. F. W., ”Atoms in molecules: A Quantum Theory“, Oxford University Press, New York, 1990.

[4] Brett M. Bodea, Mark S. Gordona, Macmolplt: ”A graphical user interface for GAMESS“, Journal of Molecular Graphics and Modelling, 1998.

[5] Chen, M. Ebert, D.; Hagen, H.; Laramee, R.S. ; van Liere, R. ; Ma, K.-L. ; Ribarsky, W. ; Scheuermann, G. ; Silver, D. ”Data, Information, and Knowledge in Visualization“. IEEE Xplore. Accessed July 21, 2014».

[6] Cook, Shane. 2013. ”CUDA Programming a Developer’s Guide to Parallel Computing with GPUs“. Amsterdam; Boston: Morgan Kaufmann. http://www.sciencedirect.com/science/book/9780124159334.

[7] Friedrichs, Mark S., Peter Eastman, Vishal Vaidyanathan, Mike Houston, Scott Legrand, Adam L. Beberg, Daniel L. Ensign, Christopher M. Bruns, and Vijay S. Pande. ”Accelerating Molecular Dynamic Simulation on Graphics Processing Units.“ Journal of Computational Chemistry 30, no. 6 (2009): 864–72.

[8] Hardy, David J., John E. Stone, Kirby L. Vandivort, David Gohara, Christopher Rodrigues, and Klaus Schulten. "Fast Molecular Electrostatics Algorithms on GPUs." Urbana 51 (2010): 61801.

[9] Heiden, W., T. Goetze, and J. Brickmann. "Fast Generation of Molecular Surfaces from 3D Data Fields with an Enhanced marching Cube Algorithm." Journal of Computational Chemistry 14, no. 2 (February 1993): 246–50. doi:10.1002/jcc.540140212.

[10] Hernández Esparza. 2011. "Análisis del enlace de hidrógeno con la matriz de densidad de orden uno: aplicación sobre tarjetas gráficas". Benemérita Universidad Autónoma de Puebla.

[11] Hernández-Esparza, R., Mejía-Chica,S.- M., Zapata-Escobar, A. D., Guevara-García, A., Martínez-Melchor, A., Hernández-Pérez, J.- M., Vargas, R. y Garza, J., "Grid-based algorithm to search critical points, in the electron density, accelerated by graphics processing units", Journal of Computational Chemistry, **35**, pp. 2272-2278, 2014.

[12] Henkelman, G.; Arnaldsson, A.; Jónsson, H.; " A fast and robust algorithm for Bader decomposition of charge density", Comput. Mater Sci. 36, 254 (2006).

[13] Humphrey, W., A. Dalke, and K. Schulten. "VMD: visual molecular dynamics." Journal of Molecular Graphics 14, no. 1 (February 1996): 33–38, 27–28.

[14] Hwu, Wen-mei W.,"GPU Computing Gems, Emerald Edition.", 2011.

[15] Hwu, W.-M. W. "GPU Computing Gems Jade Edition". Morgan Kaufmann. Amsterdam, 2011.

[16] Kirk, David, y Wen-mei Hwu. 2010. "Programming Massively Parallel Processors: a Hands-on Approach.", Burlington, Massachusetts: Morgan Kaufmann Elsevier.

[17] Kozlowski, David, and Julien Pilmé. "New insights in quantum chemical topology studies using numerical grid-based analyses.", Journal of Computational Chemistry 32, no. 15 (November 30, 2011): 3207–17. doi:10.1002/jcc.21903.

[18] Kumar, D., y M. A. Qadeer. 2010. "Fast heterogeneous computing with CUDA compatible Tesla GPU computing processor (personal supercomputing).", En Proceedings of the International Conference and Workshop on Emerging Trends in Technology, 925–930. ICWET '10. New York, NY, USA: ACM. doi:10.1145/1741906.1742121. http://doi.acm.org/10.1145/1741906.1742121.

[19] William E. Lorensen, Harvey E. Cline. 1987. "Marching Cubes: A High Resolution 3D surface construction Algotithm." Schenectady, New York 12301.

[20] Luke Benstead, "Beginning OpenGL Game Programming, Second Edition", http://ebook-dl.com/item/beginning_opengl_game_programming_second_edition_luke_benstead/

[21] Nickels, S., D. Stockel, S.C. Mueller, H.-P. Lenhof, A. Hildebrandt, and A.K. Dehof. "A Powerful Package for Presentations and Lessons in Structural Biology." In 2013 IEEE Symposium on Biological Data Visualization (BioVis), 33–40, 2013. doi:10.1109/BioVis.2013.6664344.

[22] Pavel Karas. "GPU Acceleration of Image Processing Algorithms.", Masaryk University, 2010. http://is.muni.cz/th/106808/fi_r/teze-final.pdf

[23] Paul Bourke, "Poligonising a scalar field", http://paulbourke.net/geometry/polygonise/, 1994.

[24] Phillips, James C., y John E. Stone. 2009. "Probing biomolecular machines with graphics processors". Commun. ACM 52 (10) (octubre): 34–41. doi:10.1145/1562764.1562780.

[25] Sanders, Jason, y Edward Kandrot. 2011. "CUDA by Example: An Introduction to General-purpose GPU Programming.", Upper Saddle River, NJ: Addison-Wesley.

[26] Schaftenaar, G., and J. H. Noordik. "Molden: a Pre- and Post-processing Program for Molecular and Electronic Structures*.", Journal of Computer-Aided Molecular Design 14, no. 2 (February 01, 2000): 123–34. doi:10.1023/A:1008193805436.

[27] Shiffman, Daniel. "Learning Processing: A Beginner's Guide to Programming Images, Animation, and Interaction.", 1st ed. San Francisco, Calif.; Oxford: Morgan Kaufmann, 2008.

[28] Shreiner, Dave. 2013. "OpenGL Programming Guide the Official Guide to Learning OpenGL, Versions 4.1“. http://proquest.safaribooksonline.com/?fpi=9780132748445.

[29] Stone, John E., Jan Saam, David J. Hardy, Kirby L. Vandivort, Wen-mei W. Hwu, y Klaus Schulten. 2009. "High performance computation and interactive display of molecular orbitals on GPUs and multi-core CPUs“. En Proceedings of 2nd Workshop on General Purpose Processing on cs Processing Units, 9–18. GPGPU-2. New York, NY, USA: ACM. doi:10.1145/1513895.1513897. http://doi.acm.org/10.1145/1513895.1513897.

[30] Stone, John E., David J. Hardy, Ivan S. Ufimtsev, and Klaus Schulten. "GPU-accelerated Molecular Modeling Coming of Age." Journal of Molecular Graphics and Modelling 29, no. 2 (September 2010): 116–25. doi:10.1016/j.jmgm.2010.06.010.

[31] T. Todd, Elvins, "A survey of Algorithms for Volume Visualization", SIGGRAPH'91.

[32] Ufimtsev , S.; Martínez, T. J., "Quantum chemistry on graphical processing units. 1. Strategies for two-electron integral evaluation", Journal of Chemical Theory and Computation. 4, 222 (2008).

[33] Weiguo, Bertil Schmidt, Gerrit Voss, and Wolfgang Müller-Wittig. "Accelerating Molecular Dynamics Simulations Using Graphics Processing Units with CUDA." Computer Physics Communications179, no. 9 (November 01, 2008): 634–41. doi:10.1016/j.cpc.2008.05.008.

[34] Wünsche, Burkhard Claus, and Burkhard Claus Wuensche. "A Toolkit for the Visualization of Tensor Fields in Biomedical Finite Element Models." Thesis, ResearchSpace@Auckland, 2004. https://researchspace.auckland.ac.nz/handle/2292/1225.

# Acrónimos

| | | |
|---|---|---|
| API | — | Application Programming Interface |
| CPU | — | Central Processing Unit |
| CUDA | — | Compute Unified Device Architecture |
| Gabedit | — | A Graphical User Interface for Computational Chemistry Softwares |
| GLSL | — | OpenGL Shading Language |
| GPGPU | — | General Purpose Computing on Graphics Processing Unit |
| GPU | — | Graphics Processing Unit |
| HPC | — | High Performance Computing |
| molS | — | molecular System. |
| OpenGL | — | Open Graphics Library |
| QC | — | Química Computacional |
| QT | — | Química Teórica |
| VMD | — | Visual Molecular Dynamics |

# Apéndice A

# Tablas de referencía marching cubes

```
 #define X -1
const int triTable[256][16] =
{
{ X, X, X, X, X, X, X, X, X, X, X, X, X, X, X, X},
{ 0, 8, 3, X, X, X, X, X, X, X, X, X, X, X, X, X},
{ 0, 1, 9, X, X, X, X, X, X, X, X, X, X, X, X, X},
{ 1, 8, 3, 9, 8, 1, X, X, X, X, X, X, X, X, X, X},
{ 1, 2,10, X, X, X, X, X, X, X, X, X, X, X, X, X},
{ 0, 8, 3, 1, 2,10, X, X, X, X, X, X, X, X, X, X},
{ 9, 2,10, 0, 2, 9, X, X, X, X, X, X, X, X, X, X},
{ 2, 8, 3, 2,10, 8,10, 9, 8, X, X, X, X, X, X, X},
{ 3,11, 2, X, X, X, X, X, X, X, X, X, X, X, X, X},
{ 0,11, 2, 8,11, 0, X, X, X, X, X, X, X, X, X, X},
{ 1, 9, 0, 2, 3,11, X, X, X, X, X, X, X, X, X, X},
{ 1,11, 2, 1, 9,11, 9, 8,11, X, X, X, X, X, X, X},
{ 3,10, 1,11,10, 3, X, X, X, X, X, X, X, X, X, X},
{ 0,10, 1, 0, 8,10, 8,11,10, X, X, X, X, X, X, X},
{ 3, 9, 0, 3,11, 9,11,10, 9, X, X, X, X, X, X, X},
{ 9, 8,10,10, 8,11, X, X, X, X, X, X, X, X, X, X},
{ 4, 7, 8, X, X, X, X, X, X, X, X, X, X, X, X, X},
```

```
{ 4, 3, 0, 7, 3, 4, X, X, X, X, X, X, X, X, X, X},
{ 0, 1, 9, 8, 4, 7, X, X, X, X, X, X, X, X, X, X},
{ 4, 1, 9, 4, 7, 1, 7, 3, 1, X, X, X, X, X, X, X},
{ 1, 2,10, 8, 4, 7, X, X, X, X, X, X, X, X, X, X},
{ 3, 4, 7, 3, 0, 4, 1, 2,10, X, X, X, X, X, X, X},
{ 9, 2,10, 9, 0, 2, 8, 4, 7, X, X, X, X, X, X, X},
{ 2,10, 9, 2, 9, 7, 2, 7, 3, 7, 9, 4, X, X, X, X},
{ 8, 4, 7, 3,11, 2, X, X, X, X, X, X, X, X, X, X},
{11, 4, 7,11, 2, 4, 2, 0, 4, X, X, X, X, X, X, X},
{ 9, 0, 1, 8, 4, 7, 2, 3,11, X, X, X, X, X, X, X},
{ 4, 7,11, 9, 4,11, 9,11, 2, 9, 2, 1, X, X, X, X},
{ 3,10, 1, 3,11,10, 7, 8, 4, X, X, X, X, X, X, X},
{ 1,11,10, 1, 4,11, 1, 0, 4, 7,11, 4, X, X, X, X},
{ 4, 7, 8, 9, 0,11, 9,11,10,11, 0, 3, X, X, X, X},
{ 4, 7,11, 4,11, 9, 9,11,10, X, X, X, X, X, X, X},
{ 9, 5, 4, X, X, X, X, X, X, X, X, X, X, X, X, X},
{ 9, 5, 4, 0, 8, 3, X, X, X, X, X, X, X, X, X, X},
{ 0, 5, 4, 1, 5, 0, X, X, X, X, X, X, X, X, X, X},
{ 8, 5, 4, 8, 3, 5, 3, 1, 5, X, X, X, X, X, X, X},
{ 1, 2,10, 9, 5, 4, X, X, X, X, X, X, X, X, X, X},
{ 3, 0, 8, 1, 2,10, 4, 9, 5, X, X, X, X, X, X, X},
{ 5, 2,10, 5, 4, 2, 4, 0, 2, X, X, X, X, X, X, X},
{ 2,10, 5, 3, 2, 5, 3, 5, 4, 3, 4, 8, X, X, X, X},
{ 9, 5, 4, 2, 3,11, X, X, X, X, X, X, X, X, X, X},
{ 0,11, 2, 0, 8,11, 4, 9, 5, X, X, X, X, X, X, X},
{ 0, 5, 4, 0, 1, 5, 2, 3,11, X, X, X, X, X, X, X},
{ 2, 1, 5, 2, 5, 8, 2, 8,11, 4, 8, 5, X, X, X, X},
{10, 3,11,10, 1, 3, 9, 5, 4, X, X, X, X, X, X, X},
{ 4, 9, 5, 0, 8, 1, 8,10, 1, 8,11,10, X, X, X, X},
{ 5, 4, 0, 5, 0,11, 5,11,10,11, 0, 3, X, X, X, X},
```

```
{ 5, 4, 8, 5, 8,10,10, 8,11, X, X, X, X, X, X, X},
{ 9, 7, 8, 5, 7, 9, X, X, X, X, X, X, X, X, X, X},
{ 9, 3, 0, 9, 5, 3, 5, 7, 3, X, X, X, X, X, X, X},
{ 0, 7, 8, 0, 1, 7, 1, 5, 7, X, X, X, X, X, X, X},
{ 1, 5, 3, 3, 5, 7, X, X, X, X, X, X, X, X, X, X},
{ 9, 7, 8, 9, 5, 7,10, 1, 2, X, X, X, X, X, X, X},
{10, 1, 2, 9, 5, 0, 5, 3, 0, 5, 7, 3, X, X, X, X},
{ 8, 0, 2, 8, 2, 5, 8, 5, 7,10, 5, 2, X, X, X, X},
{ 2,10, 5, 2, 5, 3, 3, 5, 7, X, X, X, X, X, X, X},
{ 7, 9, 5, 7, 8, 9, 3,11, 2, X, X, X, X, X, X, X},
{ 9, 5, 7, 9, 7, 2, 9, 2, 0, 2, 7,11, X, X, X, X},
{ 2, 3,11, 0, 1, 8, 1, 7, 8, 1, 5, 7, X, X, X, X},
{11, 2, 1,11, 1, 7, 7, 1, 5, X, X, X, X, X, X, X},
{ 9, 5, 8, 8, 5, 7,10, 1, 3,10, 3,11, X, X, X, X},
{ 5, 7, 0, 5, 0, 9, 7,11, 0, 1, 0,10,11,10, 0, X},
{11,10, 0,11, 0, 3,10, 5, 0, 8, 0, 7, 5, 7, 0, X},
{11,10, 5, 7,11, 5, X, X, X, X, X, X, X, X, X, X},
{10, 6, 5, X, X, X, X, X, X, X, X, X, X, X, X, X},
{ 0, 8, 3, 5,10, 6, X, X, X, X, X, X, X, X, X, X},
{ 9, 0, 1, 5,10, 6, X, X, X, X, X, X, X, X, X, X},
{ 1, 8, 3, 1, 9, 8, 5,10, 6, X, X, X, X, X, X, X},
{ 1, 6, 5, 2, 6, 1, X, X, X, X, X, X, X, X, X, X},
{ 1, 6, 5, 1, 2, 6, 3, 0, 8, X, X, X, X, X, X, X},
{ 9, 6, 5, 9, 0, 6, 0, 2, 6, X, X, X, X, X, X, X},
{ 5, 9, 8, 5, 8, 2, 5, 2, 6, 3, 2, 8, X, X, X, X},
{ 2, 3,11,10, 6, 5, X, X, X, X, X, X, X, X, X, X},
{11, 0, 8,11, 2, 0,10, 6, 5, X, X, X, X, X, X, X},
{ 0, 1, 9, 2, 3,11, 5,10, 6, X, X, X, X, X, X, X},
{ 5,10, 6, 1, 9, 2, 9,11, 2, 9, 8,11, X, X, X, X},
{ 6, 3,11, 6, 5, 3, 5, 1, 3, X, X, X, X, X, X, X},
```

```
{ 0, 8,11, 0,11, 5, 0, 5, 1, 5,11, 6, X, X, X, X},
{ 3,11, 6, 0, 3, 6, 0, 6, 5, 0, 5, 9, X, X, X, X},
{ 6, 5, 9, 6, 9,11,11, 9, 8, X, X, X, X, X, X, X},
{ 5,10, 6, 4, 7, 8, X, X, X, X, X, X, X, X, X, X},
{ 4, 3, 0, 4, 7, 3, 6, 5,10, X, X, X, X, X, X, X},
{ 1, 9, 0, 5,10, 6, 8, 4, 7, X, X, X, X, X, X, X},
{10, 6, 5, 1, 9, 7, 1, 7, 3, 7, 9, 4, X, X, X, X},
{ 6, 1, 2, 6, 5, 1, 4, 7, 8, X, X, X, X, X, X, X},
{ 1, 2, 5, 5, 2, 6, 3, 0, 4, 3, 4, 7, X, X, X, X},
{ 8, 4, 7, 9, 0, 5, 0, 6, 5, 0, 2, 6, X, X, X, X},
{ 7, 3, 9, 7, 9, 4, 3, 2, 9, 5, 9, 6, 2, 6, 9, X},
{ 3,11, 2, 7, 8, 4,10, 6, 5, X, X, X, X, X, X, X},
{ 5,10, 6, 4, 7, 2, 4, 2, 0, 2, 7,11, X, X, X, X},
{ 0, 1, 9, 4, 7, 8, 2, 3,11, 5,10, 6, X, X, X, X},
{ 9, 2, 1, 9,11, 2, 9, 4,11, 7,11, 4, 5,10, 6, X},
{ 8, 4, 7, 3,11, 5, 3, 5, 1, 5,11, 6, X, X, X, X},
{ 5, 1,11, 5,11, 6, 1, 0,11, 7,11, 4, 0, 4,11, X},
{ 0, 5, 9, 0, 6, 5, 0, 3, 6,11, 6, 3, 8, 4, 7, X},
{ 6, 5, 9, 6, 9,11, 4, 7, 9, 7,11, 9, X, X, X, X},
{10, 4, 9, 6, 4,10, X, X, X, X, X, X, X, X, X, X},
{ 4,10, 6, 4, 9,10, 0, 8, 3, X, X, X, X, X, X, X},
{10, 0, 1,10, 6, 0, 6, 4, 0, X, X, X, X, X, X, X},
{ 8, 3, 1, 8, 1, 6, 8, 6, 4, 6, 1,10, X, X, X, X},
{ 1, 4, 9, 1, 2, 4, 2, 6, 4, X, X, X, X, X, X, X},
{ 3, 0, 8, 1, 2, 9, 2, 4, 9, 2, 6, 4, X, X, X, X},
{ 0, 2, 4, 4, 2, 6, X, X, X, X, X, X, X, X, X, X},
{ 8, 3, 2, 8, 2, 4, 4, 2, 6, X, X, X, X, X, X, X},
{10, 4, 9,10, 6, 4,11, 2, 3, X, X, X, X, X, X, X},
{ 0, 8, 2, 2, 8,11, 4, 9,10, 4,10, 6, X, X, X, X},
{ 3,11, 2, 0, 1, 6, 0, 6, 4, 6, 1,10, X, X, X, X},
```

```
{ 6, 4, 1, 6, 1,10, 4, 8, 1, 2, 1,11, 8,11, 1, X},
{ 9, 6, 4, 9, 3, 6, 9, 1, 3,11, 6, 3, X, X, X, X},
{ 8,11, 1, 8, 1, 0,11, 6, 1, 9, 1, 4, 6, 4, 1, X},
{ 3,11, 6, 3, 6, 0, 0, 6, 4, X, X, X, X, X, X, X},
{ 6, 4, 8,11, 6, 8, X, X, X, X, X, X, X, X, X, X},
{ 7,10, 6, 7, 8,10, 8, 9,10, X, X, X, X, X, X, X},
{ 0, 7, 3, 0,10, 7, 0, 9,10, 6, 7,10, X, X, X, X},
{10, 6, 7, 1,10, 7, 1, 7, 8, 1, 8, 0, X, X, X, X},
{10, 6, 7,10, 7, 1, 1, 7, 3, X, X, X, X, X, X, X},
{ 1, 2, 6, 1, 6, 8, 1, 8, 9, 8, 6, 7, X, X, X, X},
{ 2, 6, 9, 2, 9, 1, 6, 7, 9, 0, 9, 3, 7, 3, 9, X},
{ 7, 8, 0, 7, 0, 6, 6, 0, 2, X, X, X, X, X, X, X},
{ 7, 3, 2, 6, 7, 2, X, X, X, X, X, X, X, X, X, X},
{ 2, 3,11,10, 6, 8,10, 8, 9, 8, 6, 7, X, X, X, X},
{ 2, 0, 7, 2, 7,11, 0, 9, 7, 6, 7,10, 9,10, 7, X},
{ 1, 8, 0, 1, 7, 8, 1,10, 7, 6, 7,10, 2, 3,11, X},
{11, 2, 1,11, 1, 7,10, 6, 1, 6, 7, 1, X, X, X, X},
{ 8, 9, 6, 8, 6, 7, 9, 1, 6,11, 6, 3, 1, 3, 6, X},
{ 0, 9, 1,11, 6, 7, X, X, X, X, X, X, X, X, X, X},
{ 7, 8, 0, 7, 0, 6, 3,11, 0,11, 6, 0, X, X, X, X},
{ 7,11, 6, X, X, X, X, X, X, X, X, X, X, X, X, X},
{ 7, 6,11, X, X, X, X, X, X, X, X, X, X, X, X, X},
{ 3, 0, 8,11, 7, 6, X, X, X, X, X, X, X, X, X, X},
{ 0, 1, 9,11, 7, 6, X, X, X, X, X, X, X, X, X, X},
{ 8, 1, 9, 8, 3, 1,11, 7, 6, X, X, X, X, X, X, X},
{10, 1, 2, 6,11, 7, X, X, X, X, X, X, X, X, X, X},
{ 1, 2,10, 3, 0, 8, 6,11, 7, X, X, X, X, X, X, X},
{ 2, 9, 0, 2,10, 9, 6,11, 7, X, X, X, X, X, X, X},
{ 6,11, 7, 2,10, 3,10, 8, 3,10, 9, 8, X, X, X, X},
{ 7, 2, 3, 6, 2, 7, X, X, X, X, X, X, X, X, X, X},
```

```
{ 7, 0, 8, 7, 6, 0, 6, 2, 0, X, X, X, X, X, X, X},
{ 2, 7, 6, 2, 3, 7, 0, 1, 9, X, X, X, X, X, X, X},
{ 1, 6, 2, 1, 8, 6, 1, 9, 8, 8, 7, 6, X, X, X, X},
{10, 7, 6,10, 1, 7, 1, 3, 7, X, X, X, X, X, X, X},
{10, 7, 6, 1, 7,10, 1, 8, 7, 1, 0, 8, X, X, X, X},
{ 0, 3, 7, 0, 7,10, 0,10, 9, 6,10, 7, X, X, X, X},
{ 7, 6,10, 7,10, 8, 8,10, 9, X, X, X, X, X, X, X},
{ 6, 8, 4,11, 8, 6, X, X, X, X, X, X, X, X, X, X},
{ 3, 6,11, 3, 0, 6, 0, 4, 6, X, X, X, X, X, X, X},
{ 8, 6,11, 8, 4, 6, 9, 0, 1, X, X, X, X, X, X, X},
{ 9, 4, 6, 9, 6, 3, 9, 3, 1,11, 3, 6, X, X, X, X},
{ 6, 8, 4, 6,11, 8, 2,10, 1, X, X, X, X, X, X, X},
{ 1, 2,10, 3, 0,11, 0, 6,11, 0, 4, 6, X, X, X, X},
{ 4,11, 8, 4, 6,11, 0, 2, 9, 2,10, 9, X, X, X, X},
{10, 9, 3,10, 3, 2, 9, 4, 3,11, 3, 6, 4, 6, 3, X},
{ 8, 2, 3, 8, 4, 2, 4, 6, 2, X, X, X, X, X, X, X},
{ 0, 4, 2, 4, 6, 2, X, X, X, X, X, X, X, X, X, X},
{ 1, 9, 0, 2, 3, 4, 2, 4, 6, 4, 3, 8, X, X, X, X},
{ 1, 9, 4, 1, 4, 2, 2, 4, 6, X, X, X, X, X, X, X},
{ 8, 1, 3, 8, 6, 1, 8, 4, 6, 6,10, 1, X, X, X, X},
{10, 1, 0,10, 0, 6, 6, 0, 4, X, X, X, X, X, X, X},
{ 4, 6, 3, 4, 3, 8, 6,10, 3, 0, 3, 9,10, 9, 3, X},
{10, 9, 4, 6,10, 4, X, X, X, X, X, X, X, X, X, X},
{ 4, 9, 5, 7, 6,11, X, X, X, X, X, X, X, X, X, X},
{ 0, 8, 3, 4, 9, 5,11, 7, 6, X, X, X, X, X, X, X},
{ 5, 0, 1, 5, 4, 0, 7, 6,11, X, X, X, X, X, X, X},
{11, 7, 6, 8, 3, 4, 3, 5, 4, 3, 1, 5, X, X, X, X},
{ 9, 5, 4,10, 1, 2, 7, 6,11, X, X, X, X, X, X, X},
{ 6,11, 7, 1, 2,10, 0, 8, 3, 4, 9, 5, X, X, X, X},
{ 7, 6,11, 5, 4,10, 4, 2,10, 4, 0, 2, X, X, X, X},
```

```
{ 3, 4, 8, 3, 5, 4, 3, 2, 5,10, 5, 2,11, 7, 6, X},
{ 7, 2, 3, 7, 6, 2, 5, 4, 9, X, X, X, X, X, X, X},
{ 9, 5, 4, 0, 8, 6, 0, 6, 2, 6, 8, 7, X, X, X, X},
{ 3, 6, 2, 3, 7, 6, 1, 5, 0, 5, 4, 0, X, X, X, X},
{ 6, 2, 8, 6, 8, 7, 2, 1, 8, 4, 8, 5, 1, 5, 8, X},
{ 9, 5, 4,10, 1, 6, 1, 7, 6, 1, 3, 7, X, X, X, X},
{ 1, 6,10, 1, 7, 6, 1, 0, 7, 8, 7, 0, 9, 5, 4, X},
{ 4, 0,10, 4,10, 5, 0, 3,10, 6,10, 7, 3, 7,10, X},
{ 7, 6,10, 7,10, 8, 5, 4,10, 4, 8,10, X, X, X, X},
{ 6, 9, 5, 6,11, 9,11, 8, 9, X, X, X, X, X, X, X},
{ 3, 6,11, 0, 6, 3, 0, 5, 6, 0, 9, 5, X, X, X, X},
{ 0,11, 8, 0, 5,11, 0, 1, 5, 5, 6,11, X, X, X, X},
{ 6,11, 3, 6, 3, 5, 5, 3, 1, X, X, X, X, X, X, X},
{ 1, 2,10, 9, 5,11, 9,11, 8,11, 5, 6, X, X, X, X},
{ 0,11, 3, 0, 6,11, 0, 9, 6, 5, 6, 9, 1, 2,10, X},
{11, 8, 5,11, 5, 6, 8, 0, 5,10, 5, 2, 0, 2, 5, X},
{ 6,11, 3, 6, 3, 5, 2,10, 3,10, 5, 3, X, X, X, X},
{ 5, 8, 9, 5, 2, 8, 5, 6, 2, 3, 8, 2, X, X, X, X},
{ 9, 5, 6, 9, 6, 0, 0, 6, 2, X, X, X, X, X, X, X},
{ 1, 5, 8, 1, 8, 0, 5, 6, 8, 3, 8, 2, 6, 2, 8, X},
{ 1, 5, 6, 2, 1, 6, X, X, X, X, X, X, X, X, X, X},
{ 1, 3, 6, 1, 6,10, 3, 8, 6, 5, 6, 9, 8, 9, 6, X},
{10, 1, 0,10, 0, 6, 9, 5, 0, 5, 6, 0, X, X, X, X},
{ 0, 3, 8, 5, 6,10, X, X, X, X, X, X, X, X, X, X},
{10, 5, 6, X, X, X, X, X, X, X, X, X, X, X, X, X},
{11, 5,10, 7, 5,11, X, X, X, X, X, X, X, X, X, X},
{11, 5,10,11, 7, 5, 8, 3, 0, X, X, X, X, X, X, X},
{ 5,11, 7, 5,10,11, 1, 9, 0, X, X, X, X, X, X, X},
{10, 7, 5,10,11, 7, 9, 8, 1, 8, 3, 1, X, X, X, X},
{11, 1, 2,11, 7, 1, 7, 5, 1, X, X, X, X, X, X, X},
```

```
{ 0, 8, 3, 1, 2, 7, 1, 7, 5, 7, 2,11, X, X, X, X},
{ 9, 7, 5, 9, 2, 7, 9, 0, 2, 2,11, 7, X, X, X, X},
{ 7, 5, 2, 7, 2,11, 5, 9, 2, 3, 2, 8, 9, 8, 2, X},
{ 2, 5,10, 2, 3, 5, 3, 7, 5, X, X, X, X, X, X, X},
{ 8, 2, 0, 8, 5, 2, 8, 7, 5,10, 2, 5, X, X, X, X},
{ 9, 0, 1, 5,10, 3, 5, 3, 7, 3,10, 2, X, X, X, X},
{ 9, 8, 2, 9, 2, 1, 8, 7, 2,10, 2, 5, 7, 5, 2, X},
{ 1, 3, 5, 3, 7, 5, X, X, X, X, X, X, X, X, X, X},
{ 0, 8, 7, 0, 7, 1, 1, 7, 5, X, X, X, X, X, X, X},
{ 9, 0, 3, 9, 3, 5, 5, 3, 7, X, X, X, X, X, X, X},
{ 9, 8, 7, 5, 9, 7, X, X, X, X, X, X, X, X, X, X},
{ 5, 8, 4, 5,10, 8,10,11, 8, X, X, X, X, X, X, X},
{ 5, 0, 4, 5,11, 0, 5,10,11,11, 3, 0, X, X, X, X},
{ 0, 1, 9, 8, 4,10, 8,10,11,10, 4, 5, X, X, X, X},
{10,11, 4,10, 4, 5,11, 3, 4, 9, 4, 1, 3, 1, 4, X},
{ 2, 5, 1, 2, 8, 5, 2,11, 8, 4, 5, 8, X, X, X, X},
{ 0, 4,11, 0,11, 3, 4, 5,11, 2,11, 1, 5, 1,11, X},
{ 0, 2, 5, 0, 5, 9, 2,11, 5, 4, 5, 8,11, 8, 5, X},
{ 9, 4, 5, 2,11, 3, X, X, X, X, X, X, X, X, X, X},
{ 2, 5,10, 3, 5, 2, 3, 4, 5, 3, 8, 4, X, X, X, X},
{ 5,10, 2, 5, 2, 4, 4, 2, 0, X, X, X, X, X, X, X},
{ 3,10, 2, 3, 5,10, 3, 8, 5, 4, 5, 8, 0, 1, 9, X},
{ 5,10, 2, 5, 2, 4, 1, 9, 2, 9, 4, 2, X, X, X, X},
{ 8, 4, 5, 8, 5, 3, 3, 5, 1, X, X, X, X, X, X, X},
{ 0, 4, 5, 1, 0, 5, X, X, X, X, X, X, X, X, X, X},
{ 8, 4, 5, 8, 5, 3, 9, 0, 5, 0, 3, 5, X, X, X, X},
{ 9, 4, 5, X, X, X, X, X, X, X, X, X, X, X, X, X},
{ 4,11, 7, 4, 9,11, 9,10,11, X, X, X, X, X, X, X},
{ 0, 8, 3, 4, 9, 7, 9,11, 7, 9,10,11, X, X, X, X},
{ 1,10,11, 1,11, 4, 1, 4, 0, 7, 4,11, X, X, X, X},
```

```
{ 3, 1, 4, 3, 4, 8, 1,10, 4, 7, 4,11,10,11, 4, X},
{ 4,11, 7, 9,11, 4, 9, 2,11, 9, 1, 2, X, X, X, X},
{ 9, 7, 4, 9,11, 7, 9, 1,11, 2,11, 1, 0, 8, 3, X},
{11, 7, 4,11, 4, 2, 2, 4, 0, X, X, X, X, X, X, X},
{11, 7, 4,11, 4, 2, 8, 3, 4, 3, 2, 4, X, X, X, X},
{ 2, 9,10, 2, 7, 9, 2, 3, 7, 7, 4, 9, X, X, X, X},
{ 9,10, 7, 9, 7, 4,10, 2, 7, 8, 7, 0, 2, 0, 7, X},
{ 3, 7,10, 3,10, 2, 7, 4,10, 1,10, 0, 4, 0,10, X},
{ 1,10, 2, 8, 7, 4, X, X, X, X, X, X, X, X, X, X},
{ 4, 9, 1, 4, 1, 7, 7, 1, 3, X, X, X, X, X, X, X},
{ 4, 9, 1, 4, 1, 7, 0, 8, 1, 8, 7, 1, X, X, X, X},
{ 4, 0, 3, 7, 4, 3, X, X, X, X, X, X, X, X, X, X},
{ 4, 8, 7, X, X, X, X, X, X, X, X, X, X, X, X, X},
{ 9,10, 8,10,11, 8, X, X, X, X, X, X, X, X, X, X},
{ 3, 0, 9, 3, 9,11,11, 9,10, X, X, X, X, X, X, X},
{ 0, 1,10, 0,10, 8, 8,10,11, X, X, X, X, X, X, X},
{ 3, 1,10,11, 3,10, X, X, X, X, X, X, X, X, X, X},
{ 1, 2,11, 1,11, 9, 9,11, 8, X, X, X, X, X, X, X},
{ 3, 0, 9, 3, 9,11, 1, 2, 9, 2,11, 9, X, X, X, X},
{ 0, 2,11, 8, 0,11, X, X, X, X, X, X, X, X, X, X},
{ 3, 2,11, X, X, X, X, X, X, X, X, X, X, X, X, X},
{ 2, 3, 8, 2, 8,10,10, 8, 9, X, X, X, X, X, X, X},
{ 9,10, 2, 0, 9, 2, X, X, X, X, X, X, X, X, X, X},
{ 2, 3, 8, 2, 8,10, 0, 1, 8, 1,10, 8, X, X, X, X},
{ 1,10, 2, X, X, X, X, X, X, X, X, X, X, X, X, X},
{ 1, 3, 8, 9, 1, 8, X, X, X, X, X, X, X, X, X, X},
{ 0, 9, 1, X, X, X, X, X, X, X, X, X, X, X, X, X},
{ 0, 3, 8, X, X, X, X, X, X, X, X, X, X, X, X, X},
{ X, X, X, X, X, X, X, X, X, X, X, X, X, X, X, X}};

 const int numVertsTable[256] =
```

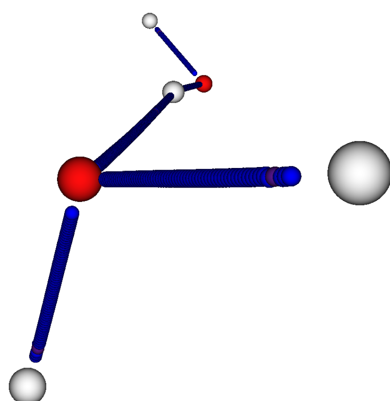
```
{
     0, 3, 3, 6, 3, 6, 6, 9, 3, 6,
     6, 9, 6, 9, 9, 6, 3, 6, 6, 9,
     6, 9, 9,12, 6, 9, 9,12, 9,12,
    12, 9, 3, 6, 6, 9, 6, 9, 9,12,
     6, 9, 9,12, 9,12,12, 9, 6, 9,
     9, 6, 9,12,12, 9, 9,12,12, 9,
    12,15,15, 6, 3, 6, 6, 9, 6, 9,
     9,12, 6, 9, 9,12, 9,12,12, 9,
     6, 9, 9,12, 9,12,12,15, 9,12,
    12,15,12,15,15,12, 6, 9, 9,12,
     9,12, 6, 9, 9,12,12,15,12,15,
     9, 6, 9,12,12, 9,12,15, 9, 6,
    12,15,15,12,15, 6,12, 3, 3, 6,
     6, 9, 6, 9, 9,12, 6, 9, 9,12,
     9,12,12, 9, 6, 9, 9,12, 9,12,
    12,15, 9, 6,12, 9,12, 9,15, 6,
     6, 9, 9,12, 9,12,12,15, 9,12,
    12,15,12,15,15,12, 9,12,12, 9,
    12,15,15,12,12, 9,15, 6,15,12,
     6, 3, 6, 9, 9,12, 9,12,12,15,
     9,12,12,15, 6, 9, 9, 6, 9,12,
    12,15,12,15,15, 6,12, 9,15,12,
     9, 6,12, 3, 9,12,12,15,12,15,
     9,12,12,15,15, 6, 9,12, 6, 3,
     6, 9, 9, 6, 9,12, 6, 3, 9, 6,
    12, 3, 6, 3, 3, 0,
};

 /*
 * cubo
```

```
* x
* |
* 4---------7 --- y
* |\        |\
* | \       | \
* |  \      |  \
* |   \     |   \
* 0----5----3----6
*  \   |     \   |
*   \  |      \  |
*    \ |       \ |
*     \|        \|
*      1---------2
*       \
*        \
*         z
* x
* |    7
* *---------* --- y
* |\        |\
* | \4      | \
*8|  \      |  \6
* |   \     |   \
* *--3-*----*-5--*
*  \   |     \   |
*   \  |9     \2 |10
*   0\ |       \ |
*     \|        \|
*      *---------*
*       \   1
```

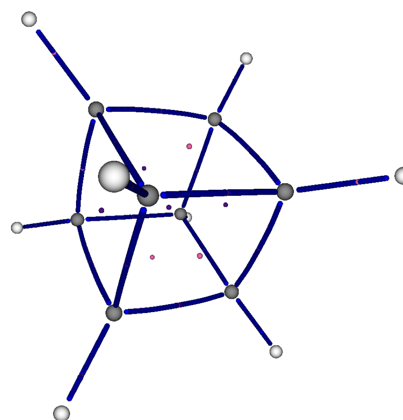
```
 *        \
 *        z
 */
const int edgeConnection[12][2] =
{
        {0,1}, {1,2}, {2,3}, {3,0},
        {4,5}, {5,6}, {6,7}, {7,4},
        {0,4}, {1,5}, {2,6}, {3,7}
};
```

# Apéndice B

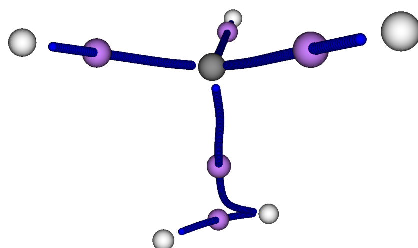
# Visualización con molS



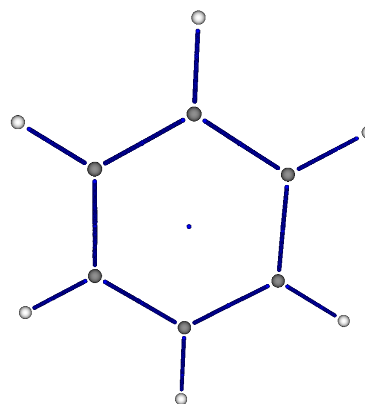
(a) Dímero de agua



(b) Cubano

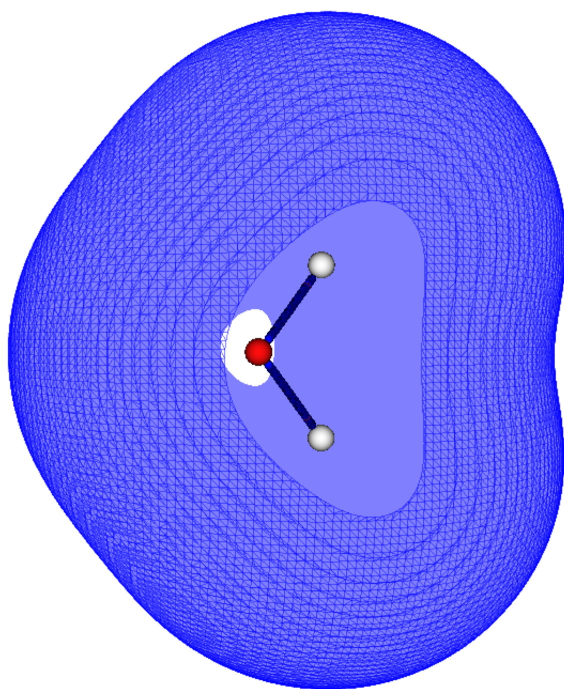


(c) $CH_5$

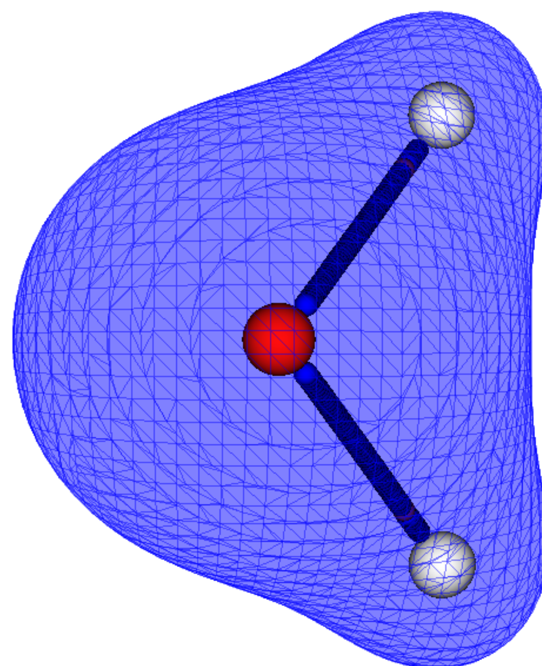


(d) Benzeno

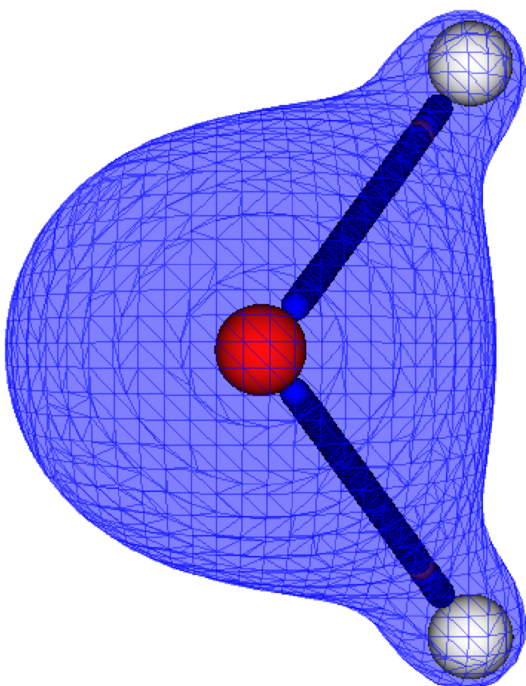
Figura B.1: Geometrías moleculares



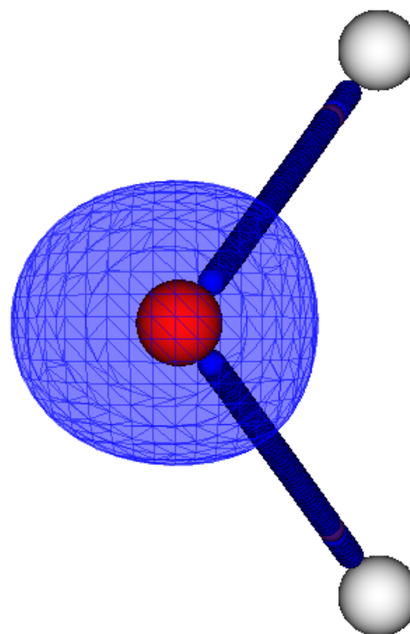
(a) IsoValor: 0.000015



(b) IsoValor: 0.1

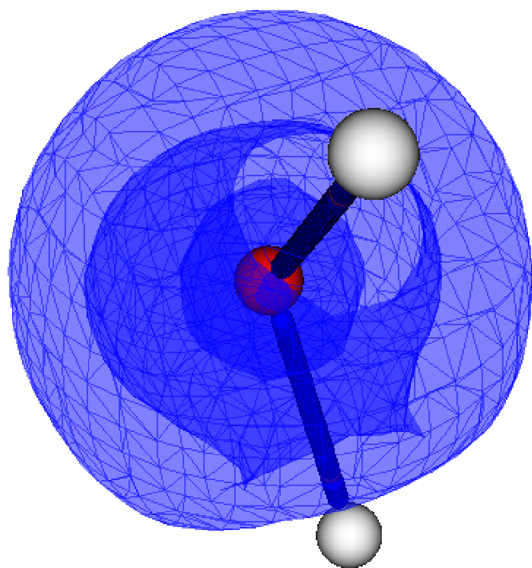


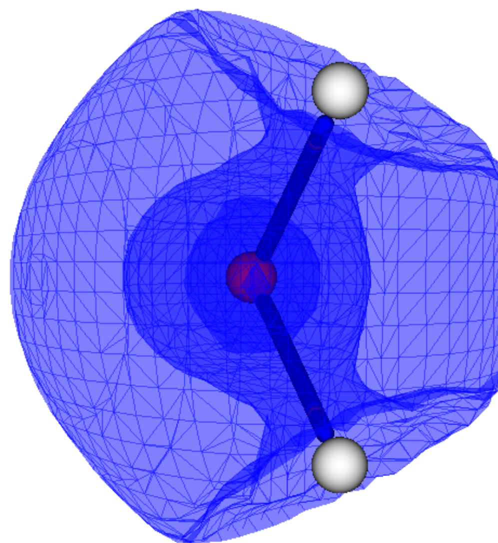
(c) IsoValor: 0.248832

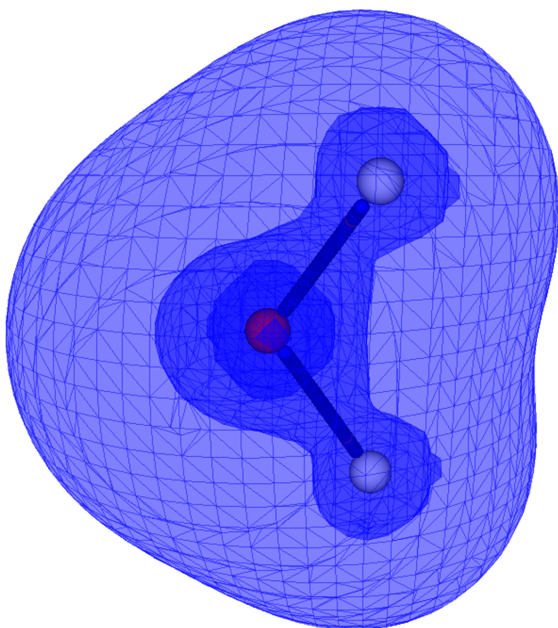


(d) IsoValor: 0.594407

Figura B.2: Densidad electrónica de una molécula de agua


(a) IsoValor: 0.285866


(b) IsoValor: 0.1


(c) IsoValor: 0.057552

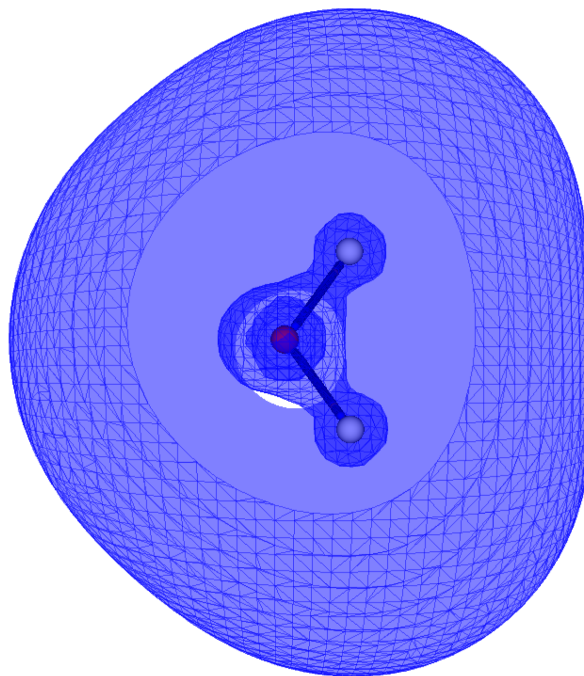
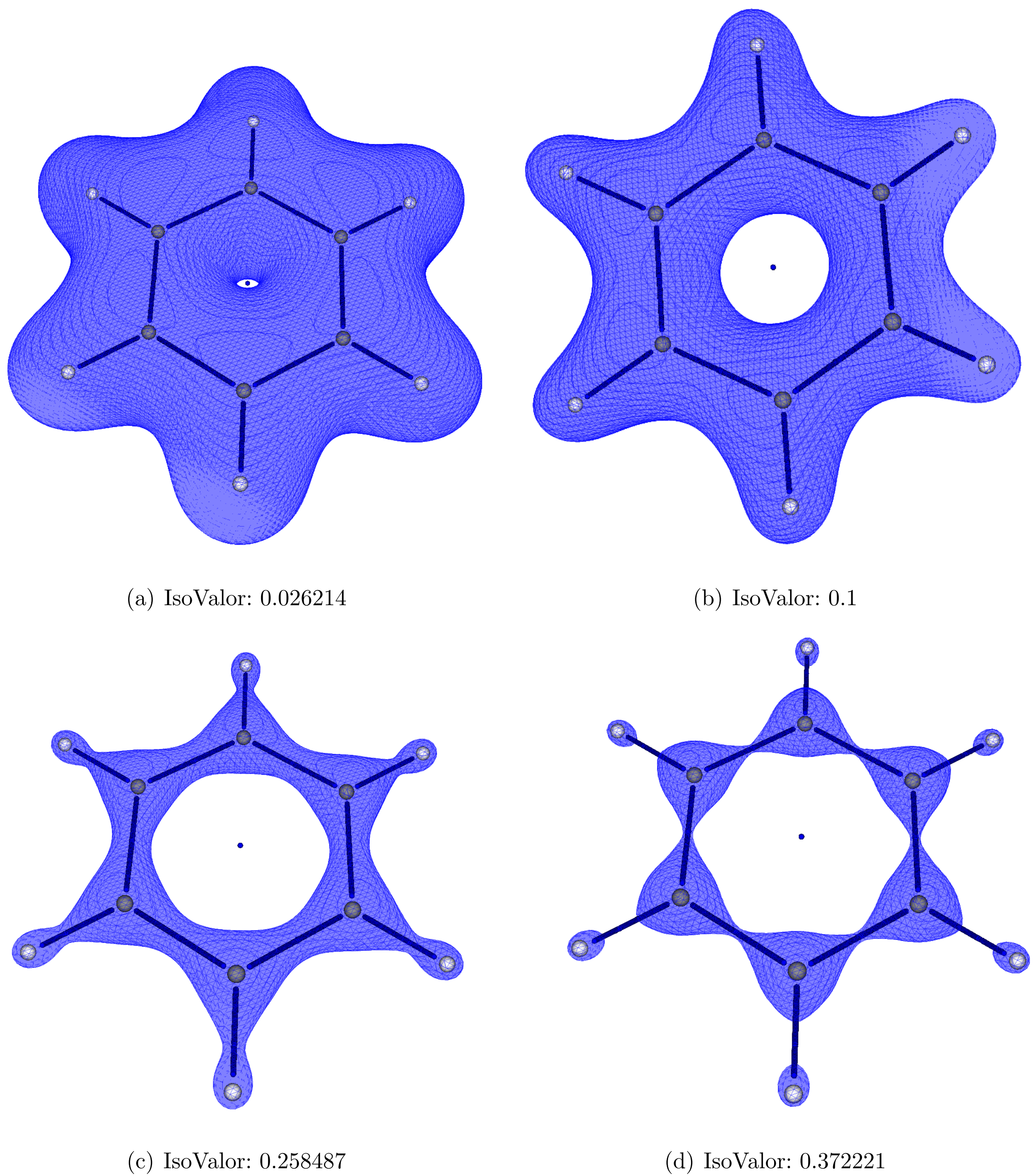

(d) IsoValor: 0.000177

Figura B.3: Laplaciano de la molécula de agua



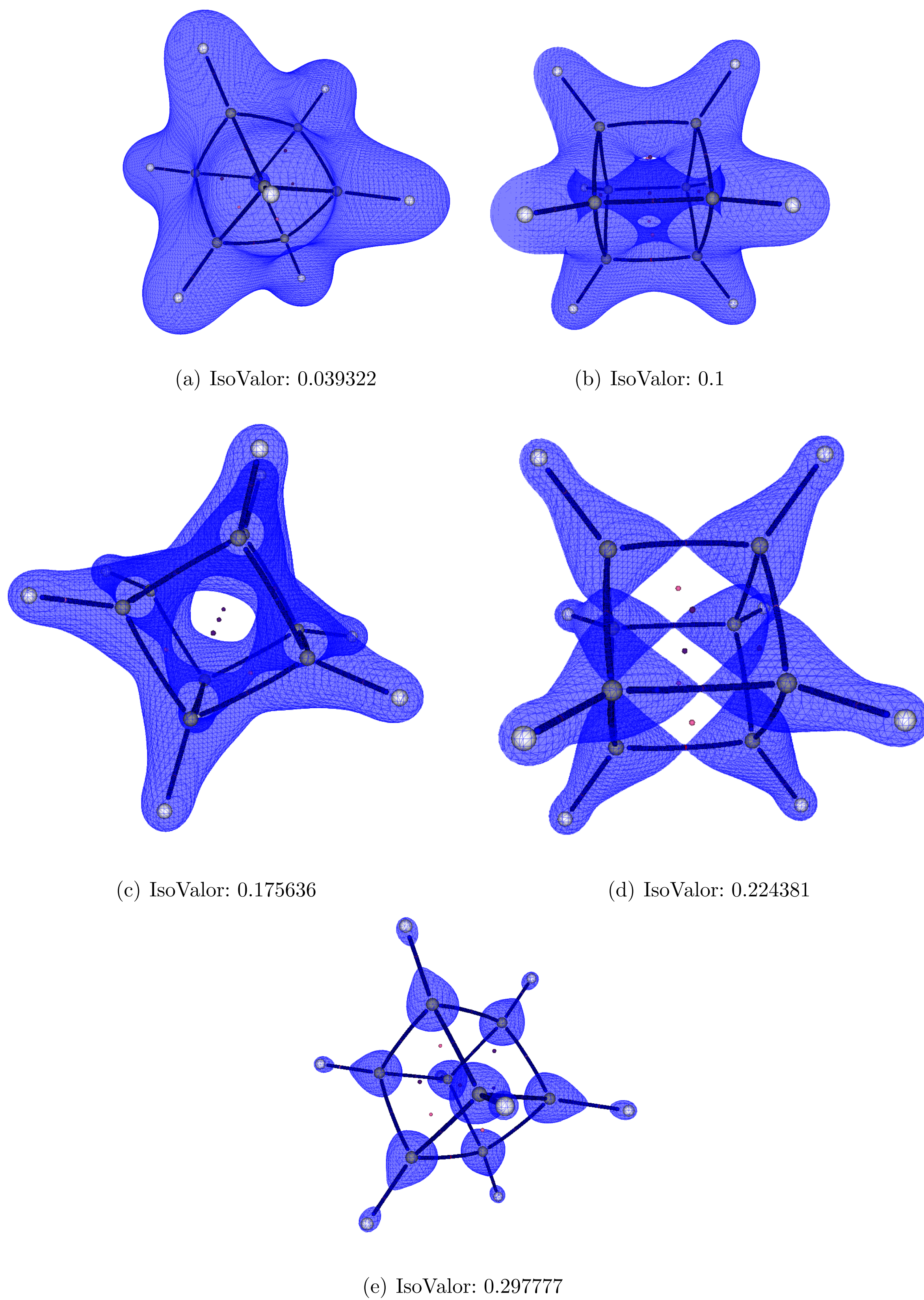
(a) IsoValor: 0.026214

(b) IsoValor: 0.1

(c) IsoValor: 0.258487

(d) IsoValor: 0.372221

Figura B.4: Densidad electrónica de una molécula benzeno



(a) IsoValor: 0.039322

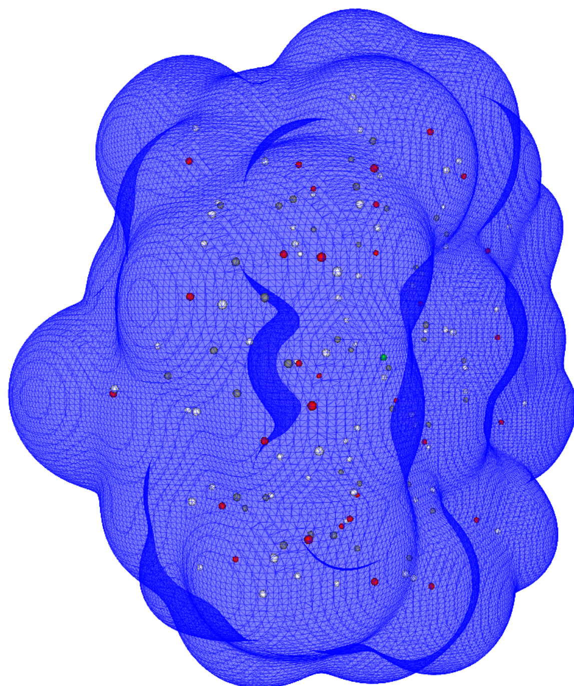
(b) IsoValor: 0.1

(c) IsoValor: 0.175636

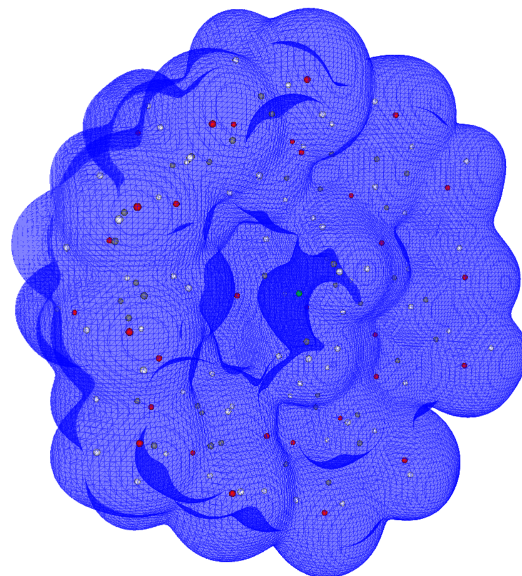
(d) IsoValor: 0.224381

(e) IsoValor: 0.297777

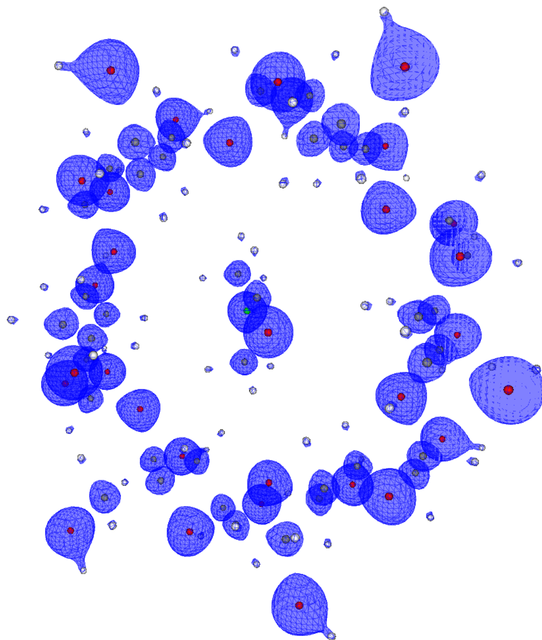
Figura B.5: Densidad electrónica de una molécula cubano



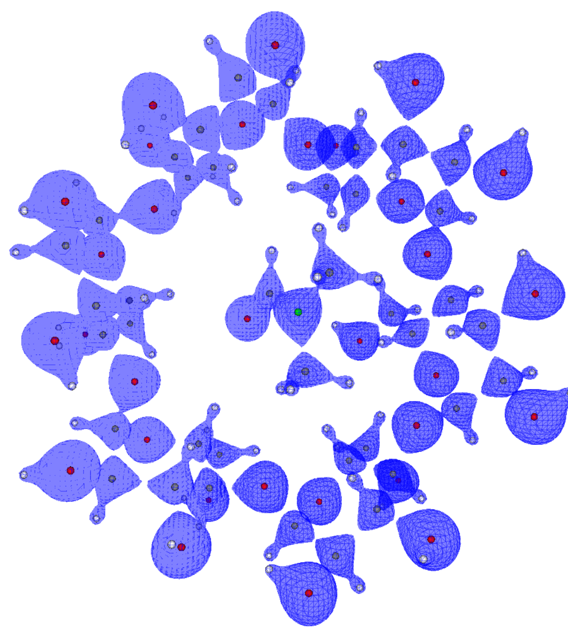
(a) IsoValor: 0.000142



(b) IsoValor: 0.002339



(c) IsoValor: 0.275188



(d) IsoValor: 0.330226

Figura B.6: Densidad electrónica de una molécula $\beta$-ciclodextrina

# Apéndice C

# Interoperabilidad entre CUDA y OpenGL

```
 1.- Segmento de código para la interoperabilidad de OpenGL y CUDA
// Creación e inicialización de buffers , y registro en CUDA
// Creación de un buffer object
glGenBuffers( 1, vbo);
glBindBuffer( GL_ARRAY_BUFFER, *vbo);
// Inicialización de un buffer object
unsigned int size = mesh_width*mesh_height*4*sizeof( float)*2;
glBufferData( GL_ARRAY_BUFFER, size, 0,GL_DYNAMIC_DRAW);
glBindBuffer( GL_ARRAY_BUFFER, 0);
// Registro del buffer object en CUDA
cudaGLRegisterBufferObject(*vbo);

2. Mapeo de OpenGL buffer object en CUDA
float4 *dptr;
cudaGLMapBufferObject( (void**)&dptr, vbo));

3. Ejecución sobre dispositivo gráfico con CUDA
dim3 block(8, 8, 1);
```

```
dim3 grid(mesh_width / block.x, mesh_height block.y, 1);
kernel<<< grid, block>>>(dptr, mesh_width,mesh_height, anim);
```