



UNIVERSIDAD
DE MÁLAGA

UNIVERSIDAD DE MÁLAGA
ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA
INGENIERÍA INFORMÁTICA

VISOR EN TCL3D PARA VISUALIZACIÓN DINÁMICA
TRIDIMENSIONAL DE MOLÉCULAS QUÍMICAS

Realizado por
ÓSCAR NOEL AMAYA GARCÍA

Dirigido por
FRANCISCO NÁJERA ALBENDÍN

Director Académico
JOSÉ MARÍA ÁLVAREZ PALOMO

Departamento
LENGUAJES Y CIENCIAS DE LA COMPUTACIÓN

MÁLAGA, Julio de 2010

UNIVERSIDAD DE MÁLAGA
ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA
INGENIERÍA INFORMÁTICA

Reunido el tribunal examinador en el día de la fecha, constituido por:

Presidente/a D./D^a. _____

Secretario/a D./D^a. _____

Vocal D./D^a. _____

para juzgar el proyecto Fin de Carrera titulado:

VISOR EN TCL3D PARA VISUALIZACIÓN DINÁMICA TRIDIMENSIONAL DE MOLÉCULAS QUÍMICAS

del alumno D. ÓSCAR NOEL AMAYA GARCÍA

dirigido por D. FRANCISCO NÁJERA ALBENDÍN

y dirigido académicamente por D. JOSÉ MARÍA ÁLVAREZ PALOMO

ACORDÓ POR _____ OTORGAR LA CALIFICACIÓN

DE _____

Y PARA QUE CONSTE, SE EXTIENDE FIRMADA POR LOS COMPARCIENTES
DEL TRIBUNAL, LA PRESENTE DILIGENCIA.

Málaga, a _____ de _____ de 2010

El/La Presidente/a

El/La Secretario/a

El/La Vocal

Fdo.

Fdo.

Fdo.

Índice general

1. Introducción	1
1.1. Antecedentes	1
1.2. Situación Inicial del Proyecto	2
1.3. Objetivos y Aportación	3
2. Preparación y Requisitos	4
2.1. Preparación Previa	4
2.1.1. Elección del lenguaje y librerías	4
2.1.2. Pruebas de Rendimiento	6
2.2. Análisis de Requisitos	7
2.2.1. Requisitos Funcionales	7
2.2.2. Requisitos No Funcionales	9
2.3. Estructura General	9
3. Diseño e Implementación	11
3.1. Metodología de Trabajo	11
3.2. Fases de Trabajo	12
3.3. Recursos Disponibles	12
3.4. Metodología de Implementación	13
3.5. Desarrollo Incremental	17
3.6. Implementación de VisorGL	21
3.6.1. Estructura Global. Motor Gráfico	25
3.6.2. Representación de Moléculas	31
3.6.3. Interacción y Selección	34
3.6.4. Edición	41

3.6.5. H.U.D	45
3.6.6. Medidas	47
3.6.7. Puentes de Hidrógeno	48
3.6.8. Etiquetas	49
3.6.9. Sistema de Referencia	50
3.6.10. Orbitales Moleculares	51
3.6.11. Animación	58
3.7. Integración	59
4. Problemas, Conclusiones y Trabajos Futuros	61
4.1. Problemas	61
4.2. Conclusiones	62
4.3. Trabajos Futuros	63
A. Instalación	64
A.1. Creación del instalador	64
A.2. Tutorial de la Instalación	65

CAPÍTULO 1

Introducción

1.1. Antecedentes

En el año 2007 se desarrolló el software *BrandyMol v1.0* [5], una completa aplicación de Modelización Molecular de la que no había antecedentes en el mercado. Desde entonces se ha estado utilizando para investigación en los departamentos de la Universidad de Málaga, y en docencia en asignaturas del *Campus Andaluz Virtual*. Sus características principales puede resumirse en:

- Representación de Moléculas. Diferentes modelos.
- Interacción a nivel de cámara. Rotación, desplazamiento, zoom.
- Interacción a nivel de moléculas. Selección y edición de las mismas.
- Representación de medidas. Distancias, ángulos y torsiones.
- Representación de puentes de hidrógeno
- Visualización de etiquetas.
- Visualización de Orbitales Moleculares.
- Automatización de cálculos semiempíricos.

BrandyMol fue implementado en *Tcl/Tk*[4], un lenguaje interpretado especializado en interfaces gráficas. Actúa como *front-end* de aplicaciones de cálculo químico de las

que se cuenta con los fuentes originales y son compilados según la plataforma destino. Para el visor molecular, se utilizó también el lenguaje *Tcl/Tk*, ya que *VTK*[11] puede ser compilado a modo de envoltorio, generando librerías para este lenguaje, resultando una integración nativa que facilita su sustitución. Para una información más detallada puede consultarse [5]

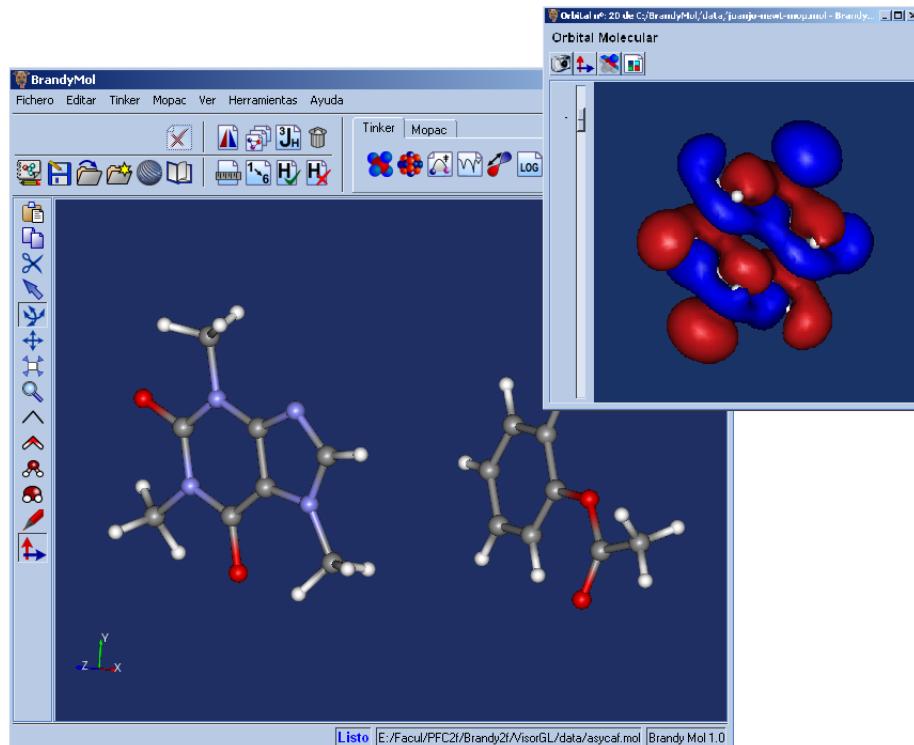


Figura 1.1: BrandyMol v1.0

1.2. Situacion Inicial del Proyecto

En estos años ha recibido un uso más interactivo del esperado, manifestando respuestas lentas con moléculas que superan el tamaño medio. Aun comportándose correctamente, hace el trabajo diario incómodo, especialmente en el ámbito de investigación.

Contamos por tanto, con todo el material de la aplicación anterior como punto de partida y el apoyo de **Gregorio Torres** y **Francisco Nájera**, los químicos que participaron activamente en la primera versión, que validarán y explotarán la nueva aportación, para asegurar que el desembolso en tiempo y dinero merece la pena.

1.3. Objetivos y Aportación

Se propone el desarrollo de un nuevo visor centrado en el **rendimiento**. Debe aportar al menos la misma funcionalidad que el anterior y reutilizar todo lo posible de la aplicación original, ser totalmente transparente al usuario y multiplataforma.

Queda por decidir el lenguaje y las librerías en las que se desarrollará el visor. Lo más apropiado, a priori, parece ser utilizar *Tcl/Tk* como lenguaje, para favorecer la integración con el resto de la aplicación y posteriores ampliaciones. No obstante, la elección de las librerías gráficas y el rendimiento en un intérprete es un tema crucial al que habrá que dedicar el tiempo necesario para asegurar que el funcionamiento en circunstancias extremas sea aceptable.

Como detalle personal, quiero destacar que al ser yo el encargado de ambos proyectos, el tiempo de adaptación al ámbito químico es mínimo. A pesar de contar con un diseño cuidado, nadie mejor conoce los aspectos a optimizar. La relación y compenetración con los químicos no pudo ser mejor tanto a nivel profesional como personal, lo que transforma este proyecto de un trámite, a una nueva aventura en la que sé que voy a divertirme y aprender. Y como cualquiera que lea esto sabrá, pocas cosas mas satisfactorias hay que trabajar en lo que te gusta.

CAPÍTULO 2

Preparación y Requisitos

2.1. Preparación Previa

Anteriormente se comentó que sería deseable que el nuevo visor estuviera implementado en *Tcl/Tk*, pero debemos encontrar las librerías gráficas adecuadas que satisfagan nuestras necesidades de rendimiento. De no obtener resultados satisfactorios, deberá tenerse en cuenta el desarrollo en otro lenguaje, y exportarlo posteriormente como un componente para el intérprete. *Tcl/Tk* permite esto mediante la creación de librerías que respeten su *API*, concretamente en *C/C++*. En *BrandyMol v1.0* esto fue necesario para tratar casos particulares de comunicación con el Sistema Operativo *Windows*, por lo que no es una alternativa desconocida.

2.1.1. Elección del lenguaje y librerías

El mercado actual lo dominan dos tecnologías principales. *DirectX*[9] de *Microsoft* y *OpenGL*[10] de distribución libre. La primera sólo funciona en entornos *Windows*, por lo que queda descartada dada la naturaleza multiplataforma de nuestra aplicación. *OpenGL* se oferta como librerías para el lenguaje *C* y es multiplataforma. Existen más librerías gráficas, pero la mayoría son una abstracción por encima de alguna de las dos anteriores, como *VTK*, pero ya que buscamos sobre todo rendimiento, delegaremos lo menos posible en terceros.

El elegido es por tanto *OpenGL*, del que ofrecemos su descripción:

OpenGL[10] (*Open Graphics Library*) es una especificación estándar que define una *API* multilenguaje y multiplataforma para escribir aplicaciones que produzcan gráficos 2D y 3D. La interfaz consiste en más de 250 funciones diferentes que pueden usarse para dibujar escenas tridimensionales complejas a partir de primitivas geométricas simples, tales como puntos, líneas y triángulos. Fue desarrollada originalmente por *Silicon Graphics Inc. (SGI)* en 1992 y se usa ampliamente en *CAD*, realidad virtual, representación científica, visualización de información y simulación de vuelo. También se usa en desarrollo de videojuegos, donde compite con *Direct3D* en plataformas *Microsoft Windows*.

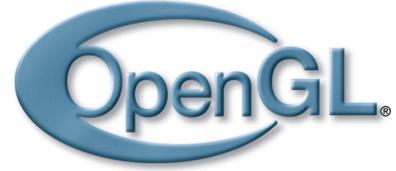


Figura 2.1: OpenGL Logo

Para nuestra suerte, encontramos en internet un envoltorio de las librerías *OpenGL* para el lenguaje *Tcl/Tk*. Se le conoce por **Tcl3D** y podemos obtener toda la información y *software* en [12]. Esto es, tratar como instrucciones nativas de nuestro intérprete los mismos comandos de la *API OpenGL*. Proporciona las librerías y código fuente para *Windows*, *Linux* y *MacOS*, toda la documentación necesaria y todos los ejemplos que la versión original de *OpenGL* proporciona en *C*, la sumistran en código *Tcl*, lo que facilita mucho el acomodamiento al envoltorio.

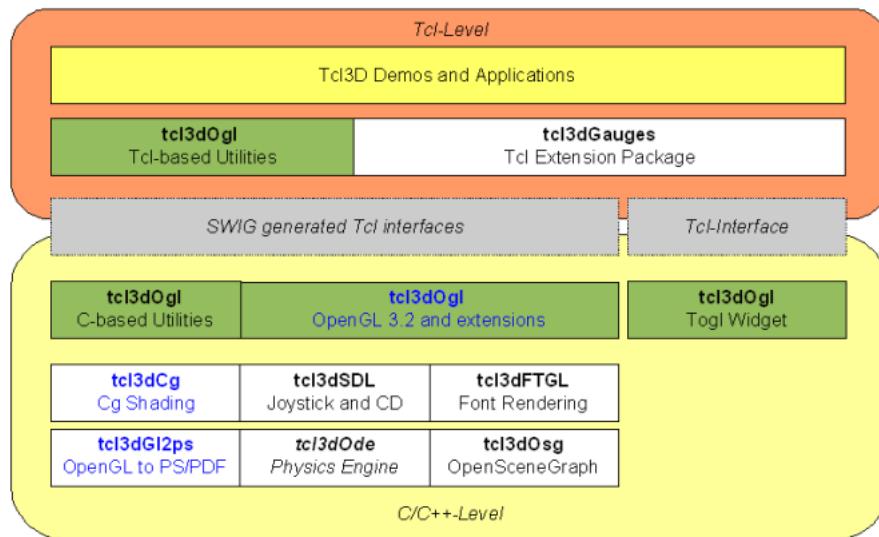


Figura 2.2: Arquitectura Tcl3D

Como podemos ver, no solo implementa la funcionalidad básica de *OpenGL*, sino técnicas avanzadas como la programación de *sombreadores*

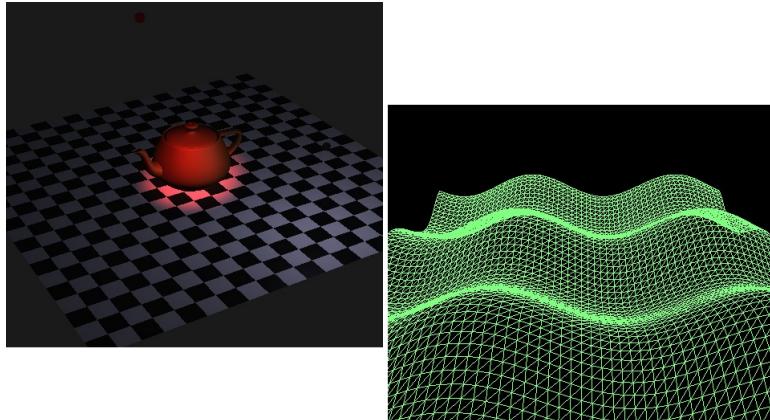


Figura 2.3: Ejemplos Tcl3D

2.1.2. Pruebas de Rendimiento

Contamos entonces con *Tcl3D*, que parece ser el candidato perfecto como lenguaje para el visor molecular, llevando al extremo la integración con la aplicación anterior. No obstante, recordemos que estamos en un intérprete donde los tiempos de ejecución no suelen ser los que estamos acostumbrados a ver utilizando lenguajes compilados. Es por ello que previo al diseño formal, se desarrollan pequeñas aplicaciones con objeto de medir el rendimiento en situaciones parecidas a un visor sobrecargado.

Algunas de las pruebas realizadas fueron:

- Sobrecarga de esferas y cilindros aleatorios simulando moléculas.
- Sobrecarga de mallas poligonales simulando orbitales moleculares.
- Sobrecarga de texto simulando etiquetas.
- Tiempos de respuesta en animación.
- Tiempos de respuesta en selección.
- Pruebas de integración como componente.

Los resultados obtenidos fueron satisfactorios en todos los aspectos, acercándose mucho al esperado en un programa compilado. Esto es porque el núcleo *OpenGL* está realmente compilado como librería. Es en el manejo de las estructuras de datos y comunicación con el exterior donde deberemos prestar mayor atención. *Tcl/Tk* no es demasiado rico en cuanto a variedad de tipos de datos, ni eficiente en el uso de los mismos, por lo que la implementación deberá ser especialmente cuidadosa e intercalar pruebas durante todo el desarrollo.

2.2. Análisis de Requisitos

La recolección de requisitos no será una labor complicada puesto que tratamos de crear un visor con al menos la misma funcionalidad que el anterior. Asimismo, para el visor anterior contábamos con la experiencia de los químicos, los cuales llevan años utilizando visores similares. Tienen ahora una vez más la oportunidad de refinarlo a su gusto en cuestiones de usabilidad, comodidad y alguna característica adicional.

Desglosaremos los requisitos en funcionales y no funcionales:

2.2.1. Requisitos Funcionales

- **Representación de moléculas.** Dar al usuario una representación tridimensional de las moléculas con que trabaja el visor.
- **Carga y descarga de moléculas.** Permitir añadir y eliminar moléculas del visor.
- **Diferentes modos de resolución.** Representar las moléculas con mayor o menor calidad visual para favorecer el rendimiento o la interactividad.
- **Modos representación total y parcial.** Permitir varios modos de representación simultáneos en diferentes secciones de la molécula.
 - Líneas
 - Cilindros
 - Cilindros y Bolas
 - CPK
- **Modo Superficie y Alambres.** Permitir alternar entre una representación de la molécula como superficie y un modo alámbrico.

- **Ocultación de partes de las moléculas.** Permitir que partes seleccionadas de las moléculas no sean visibles.
- **Interacción de cámara.** Permitir la interacción del usuario con la cámara del visor 3D permitiendo diferentes acciones.
 - **Rotación.** La cámara gira acimutalmente alrededor del centro de la escena.
 - **Desplazamiento.** La cámara realiza desplazamientos horizontales y verticales al centro de la escena.
 - **Zoom.** La cámara se aleja o acerca al centro de la escena.
 - **Centrar.** La cámara se centra en el contenido visible del visor.
- **Selección y Edición.** Permitir la selección interactiva de átomos y enlaces del contenido del visor y su posterior edición.
 - Selección de partes de las moléculas mediante click
 - Selección de partes de las moléculas mediante cuadro de selección
 - Rotación de contenido total o parcial del visor
 - Desplazamiento de contenido total o parcial del visor
- **Representación de Medidas.** Permitir la representación visual de medidas tomadas entre diferentes átomos del contenido del visor
 - **Distancias.** Entre dos átomos del visor.
 - **Ángulos.** Entre tres átomos del visor, siendo el central el átomo común.
 - **Torsiones.** Ángulo entre los planos formados por los tres primeros átomos seleccionados y los tres últimos.
- **Representación de Etiquetas.** Permitir que se muestre información genérica junto a cada átomo del contenido del visor.
- **Representación de Puentes de Hidrógeno.** Permitir que se muestren líneas discontinuas entre los átomos que produzcan puentes de hidrógeno.
- **Representación de un Eje de Coordenadas.** Mostrar un sistema de coordenadas que esté orientado al sistema de referencia en todo momento.
- **Representación de Orbitales Moleculares.** Mostrar superficies que representen valores de densidad en el espacio.
- **Modo Animación.** Permitir que la cámara gire acimutalmente de manera continua mostrando un contador de *FPS*.

2.2.2. Requisitos No Funcionales

- Incorporarse al visor anterior como componente de *Tcl/Tk*
- Multiplataforma
- Ofrecer un tiempo de respuesta considerablemente menor a su predecesor

2.3. Estructura General

Aunque este proyecto se centra en el desarrollo del visor, haremos una breve descripción de la estructura general de la aplicación. *BrandyMol* cuenta principalmente con un visor 3D capaz de representar moléculas orgánicas, medidas, etiquetas, puentes de hidrógeno, orbitales moleculares, sistemas de referencia... Asimismo cuenta con un sistema de interacción que nada tiene que envidiar a aplicaciones comerciales.

Las moléculas vienen definidas según el formato estándar *MOL*[13], para facilitar la comunicación con otros programas, especialmente con *MDL Isis/Draw*[14], programa de dibujo de moléculas en 2D muy conocido y del que se dispone de licencia de libre distribución.

Tras la visualización e interacción, otro de los puntos fuertes es la comunicación con las aplicaciones *Tinker*[15] y *Mopac*[16]. Ambos son distribuciones libres de *software* para realizar cálculos semiempíricos moleculares. Se presentan como ejecutables para línea de comandos que toman ficheros de entrada junto a parámetros y devuelven resultados.

BrandyMol se encarga de proporcionar una interfaz sencilla entre ellos y la interpretación de resultados que se muestran de manera automática en el visor. Tarea nada sencilla debido a las conversiones de formato necesarias y el complejo control de errores.

Además, incorpora ciertas características de modelado simple, como mover partes de la molécula, adición de hidrógenos o imágenes especulares, que resultan necesarias en el trabajo diario.

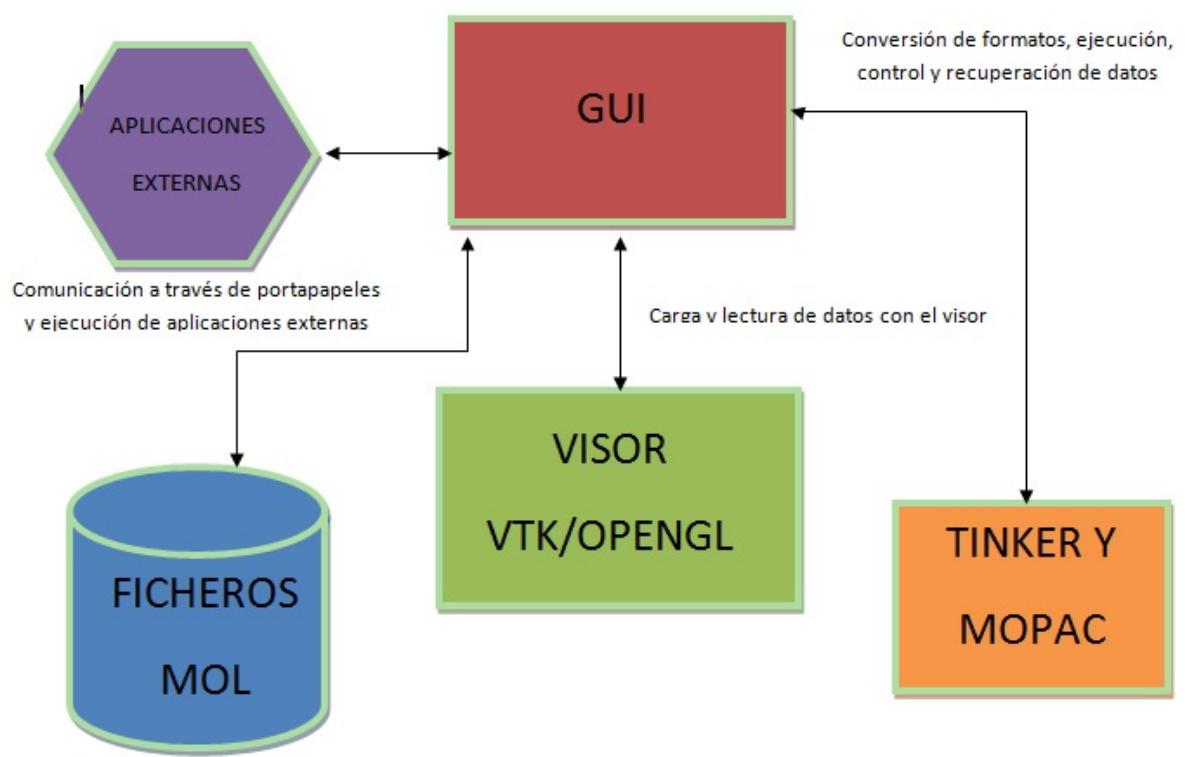


Figura 2.4: Esquema General de la Aplicación

CAPÍTULO 3

Diseño e Implementación

3.1. Metodología de Trabajo

Para la realización de este proyecto se ha utilizado el desarrollo incremental. Se implementó un visor con funcionalidad básica de representación e interacción, y posteriormente se fueron agregando características. Esto se hizo como un componente individual aislado de la aplicación principal. De este modo nos aseguramos que toda comunicación se hace a través de una interfaz perfectamente delimitada. Una vez finalizada es necesario realizar una integración, sustituyendo el visor *VTK* por el nuevo en *OpenGL*. Se hace así porque es más sencilla la automatización de ciertas labores de desarrollo y depuración en un entorno puramente *script*, en lugar de lidiar con toda la interfaz gráfica de *BrandyMol*.

Algo a tener en cuenta es la complejidad de las pruebas. Recordemos que *Tcl/Tk* es un intérprete y que no disponemos de ningún analizador sintáctico ni semántico, por lo que los errores se producirán en tiempo de ejecución. Y que estamos en una aplicación gráfica *3D*, donde muchas de las situaciones se producirán solo ante eventos concretos en su manejo. Se hace difícil en ocasiones provocar el paso por todos los posibles flujos de ejecución. A modo de ejemplo, puedo recordar que tras semanas de correcto funcionamiento, al estar interaccionando con el visor se produjo un error. Reinicié el programa y fue complicado reproducirlo. El problema se debió que al mover una molécula a una posición concreta, ésta se alineó exactamente con un plano que provocaba una división por cero solo si se miraba perpendicularmente con la cámara a dicho plano. Como puede imaginarse, esto hace temblar a cualquier testador de *software*. No obstante me atrevo a decir que una vez finalizado el desarrollo, es bastante fiable.

3.2. Fases de Trabajo

Podemos desglosar las fases de trabajo de este proyecto, en las siguientes:

- Estudio del lenguaje de comandos *Tcl*, y su extensión *Tk* para *GUI's*.
- Estudio de la *API* gráfica *OpenGL* para la representación e interacción con escenas tridimensionales.
- Comprensión del problema y familiarización con aspectos de la rama de Química Orgánica hacia la que está orientada el *software*.
- Diseño de alto nivel de abstracción identificando las estructuras globales necesarias para afrontar el problema desde un punto de vista informática.
- Desarrollo e implementación incremental basada en prototipos, contando con el punto de vista de los químicos, para verificar la corrección del programa en cada iteración. Pruebas parciales
- Pruebas de la aplicación, evaluando resultados y satisfacción por parte de los clientes.
- Optimización del código, y corrección de las posibles incidencias ocurridas en las pruebas.

3.3. Recursos Disponibles

- Físicos
 - Ordenador personal. *Intel Core 2 Duo @ 2000 Mhz, 2GB Ram, Ati Mobility Radeon X1600*
- Lógicos
 - Sistemas Operativos *Windows XP/Vista/7*, y *Ubuntu 9.04*
 - Librerías *Tcl3D* (*Wrapper* de *OpenGL* para *Tcl*)
 - Intérprete *Tcl/Tk ActiveState 8.5.5.0*
 - Compilador *C++* de *Visual Studio 2005*
 - Herramientas de cálculo químico *GNU, Tinker* y *Mopac*

- *Software de modelado molecular 2D MDL Isis Draw 2.5*
- *BrandyMol v1.0* (Aplicación anterior, desarrollada como *Proyecto Fin de Carrera* en la Ingeniería Técnica en Informática de Sistemas)

Se hace una mención especial de las características del ordenador personal, y específicamente del procesador gráfico, puesto que es particularmente potente. Sin embargo, el visor debe tener un rendimiento aceptable en ordenadores más modestos. Por lo tanto será necesario probar el visor en otras máquinas.

3.4. Metodología de Implementación

Durante la etapa de formación del anterior *Proyecto Fin de Carrera* se ideó una metodología de implementación para asemejar *Tcl/Tk* a un lenguaje orientado a objetos. Recordamos que este es un lenguaje de *scripts* imperativo, pero personalmente me gusta trabajar con *Orientacion a Objetos*. La metodología se bautizó como **MACTON** (*Metodología de Adaptación de Clases a Tcl por Oscar Noel*). Toda la aplicación se implementó siguiendo una serie de pautas que hacen posible un manejo similar a *C++* o *Java*.

Para representar una clase, se genera un espacio de nombres (*namespaces*) con el nombre de la clase. Dentro de ese espacio de nombres se generarán *arrays*, los cuales representarán las instancias de la clase. Los procedimientos de creación recibirán como mínimo un parámetro, **base** que será el nombre del array que representará al objeto. Éstos comprobarán que no exista ya uno con el mismo nombre dentro del **namespace**, de lo contrario devolverán que ya existe. Los métodos son implementados como funciones de *Tcl*, cuyo primer parámetro es **base** con el que hemos creado el objeto. La eliminación de estos objetos será llevada a cabo por una función que eliminará el *array*, así como los recursos que pudiera tener asignados.

Para cada clase existirán dos procedimientos, **new<NombreClase>** que hará de constructor y **del<NombreClase>** de destructor, que tomarán como mínimo un parámetro, **base**, que representará la instancia de nuestro objeto. Al crear con **new** se comprobará que en el *namespace* correspondiente no exista otra variable con el mismo nombre, en caso de existir devolveríamos -1 que se interpretará como error, y de lo contrario declararíamos una nueva variable con el comando *Tcl variable*, que se usará como *array* contenedor de los parámetros de nuestro objeto. La eliminación de objetos debe realizarse manualmente mediante **del**, la cual eliminará el *array* y liberará los recursos que pudiera tener asignados. Los métodos son definidos como funciones cuyo primer parámetro es el nombre de

la instancia deseada, `base`, y cuya primera línea será un enlace para tratar localmente al objeto. El prototipo sería `upvar #0<Nombre-Clase>::$base <nombre-local>`, donde realmente `<NombreClase>` es el nombre del *namespace*, que hace referencia al nivel de anidación, y `<nombre-local>` es el nombre que le queramos dar para usar el objeto dentro de nuestro método. Para mayor detalle sobre los comandos `upvar` y `namespace` puede consultarse la documentación de *Tcl/Tk*[4].

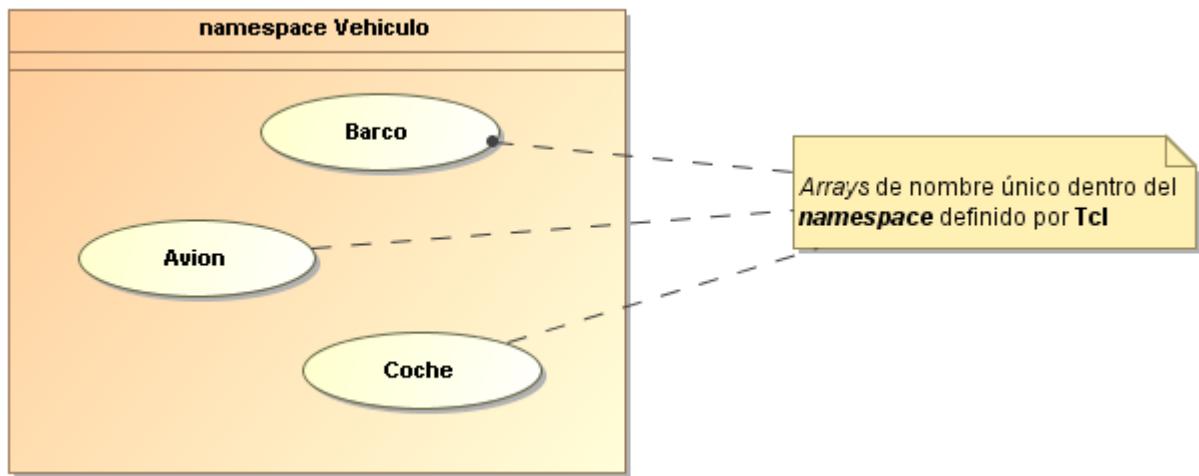


Figura 3.1: Implementación del modelo *OO* a *Tcl+MACTON*

Veamos a continuación un ejemplo simple de uso de *Macton*. En primer lugar diseñaremos una clase con varios métodos, y después haremos uso de ella creando objetos. Observaremos la similitud con lenguajes nativamente orientados a objetos.

```
class Coche {

    private String nombre;

    Coche(String nombre) {
        this.nombre= new String(nombre);
    }//fincontructor

    public String devuelveNombre() {
        return this.nombre;
    }//finmetodo

    public void cambiaNombre(String nombre) {
```

```

        this.nombre=nombre;
    };//finmetodo

}//finclass

namespace eval Coche {
    proc newCoche { base nombre } {
        if {[info exists Coche::$base]==0} {
            variable $base
            set ${base}(nombre) $nombre
            return 0
        } else {
            #"La variable ya existe"
            return -1
        }
    }; #finproc

    proc delCoche { base } {
        upvar #0 Coche::$base coche
        if {[array exists coche]} { unset coche }
    }; #finproc

    proc devuelveNombre { base } {
        upvar #0 Coche::$base coche
        return $nombre
    }; #finproc

    proc modificaNombre { base nombre } {
        upvar #0 Coche::$base coche
        set coche(nombre) $nombre
    }; #finproc
};

}; #finnamespace

```

JAVA

```

Vehiculo cs = new Vehiculo("Coche");
cs.cambiarNombre("Barco");

```

Capítulo 3. Diseño e Implementación

```
System.out.println(cs.devuelveNombre());
```

Tcl con MACTON

```
Vehiculo::newVehiculo cs "Coche"  
Coche::cambiarNombre cs "Barco"  
puts [Coche::devuelveNombre cs]
```

3.5. Desarrollo Incremental

En esta sección se mostrará de manera resumida y mediante diagramas, la evolución en el tiempo de la aplicación. En secciones posteriores podremos ver un análisis más exhaustivo de la funcionalidad con múltiples ejemplos. Cada iteración de las mostradas a continuación se compone de múltiples subiteraciones y de una pequeña batería de pruebas

- **Iteración 1.** Se desarrolla el núcleo del visor con la inicialización de los parámetros de la máquina *OpenGL* y la interfaz mínima *Macton* para poder integrarlo dentro de una ventana simple de *Tcl/Tk*.
- **Iteración 2.** Se incorpora la carga y descarga de objetos *Data* que contienen moléculas, y su representación en el visor. Los modos de representación (Líneas, Cilindros, Cilindros y Bolas y CPK) son aún globales al visor, aunque funcionales.
- **Iteración 3.** Se incorpora un manejador de eventos de teclado y ratón así como la interacción con la cámara gracias a éstos. Rotación, Traslación, Zoom y Centrado.

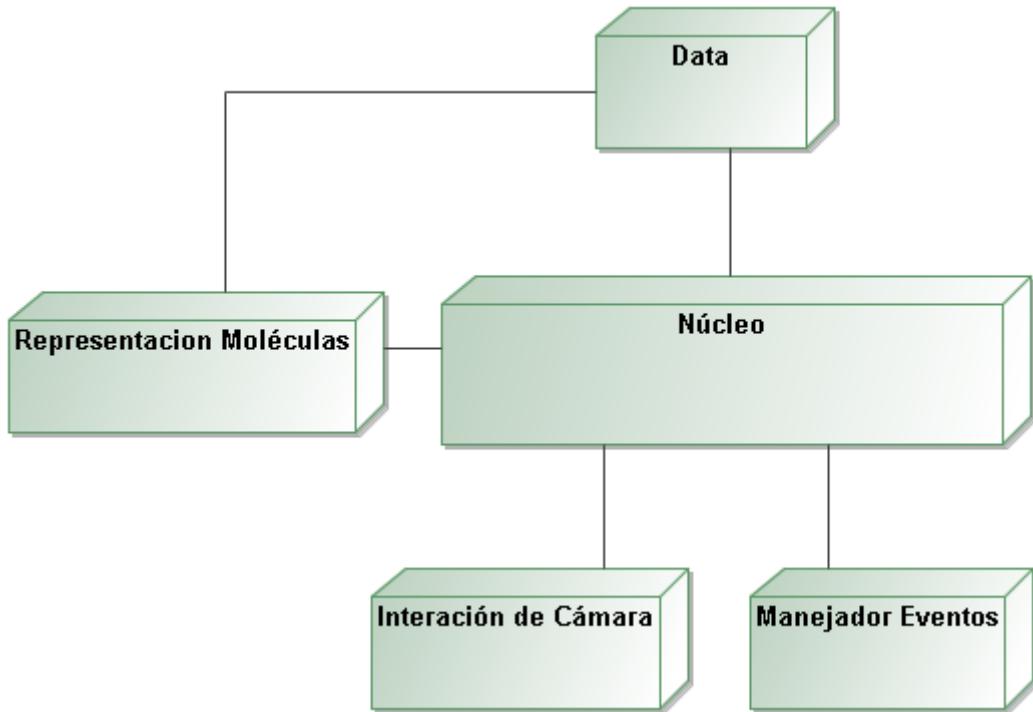


Figura 3.2: Esquema *VisorGL* tras la 3^a iteración

- **Iteración 4.** Se mejora el manejador de eventos y se incorpora la posibilidad de seleccionar partes concretas de moléculas. Esto permite labores de edición sobre las

mismas. Rotación y translación de partes o moléculas enteras. Además se permite ahora que el modo de representación no sea general al visor, sino local a cada átomo y enlace. Esto facilita visualmente tareas concretas como los *mapas de reacción*, y las reorientaciones necesarias entre *conformaciones moleculares*.

- **Iteración 5.** Una vez implementado el sistema de selección, se implementan los sistemas de Medidas (Distancias, Ángulos y Torsiones) entre cualesquiera átomos y enlaces del visor, Puentes de Hidrógeno y Etiquetas.

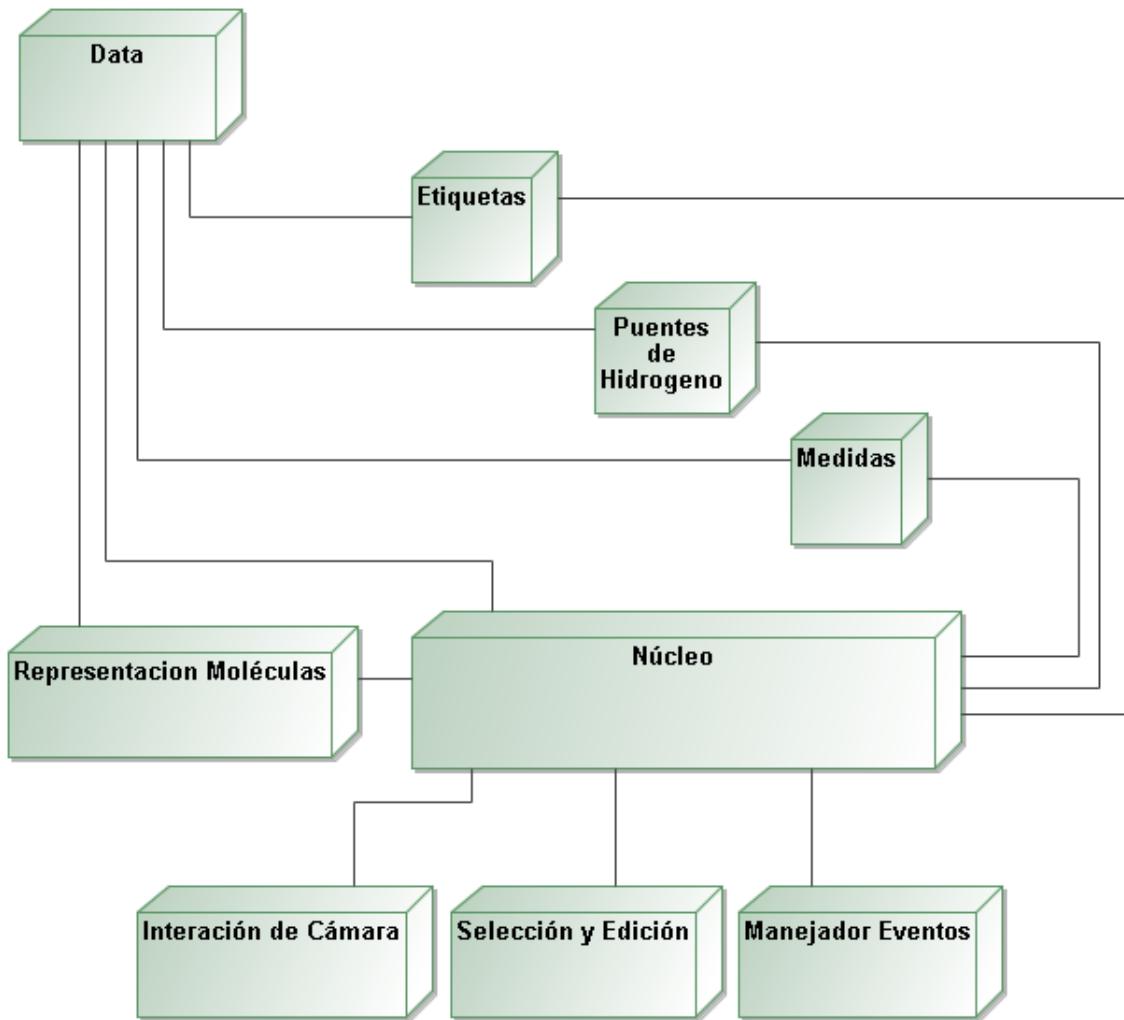


Figura 3.3: Esquema *VisorGL* tras la 5^a iteración

- **Iteración 6.** Se incorpora el modo de animación con un contador de *FPS* (*Frames por segundo*) para pruebas de rendimiento, y una *HUD* (*Head-Up Display*) para información adicional como modos activos o información sobre el adaptador gráfico.

- **Iteración 7.** Se incorpora la capacidad de representar orbitales moleculares. Éstos son superficies de triángulos que es necesario generar a partir de la información de densidad en el espacio próxima a las moléculas. Una de las iteraciones más largas y complicadas donde era necesario tener muy presente el rendimiento.

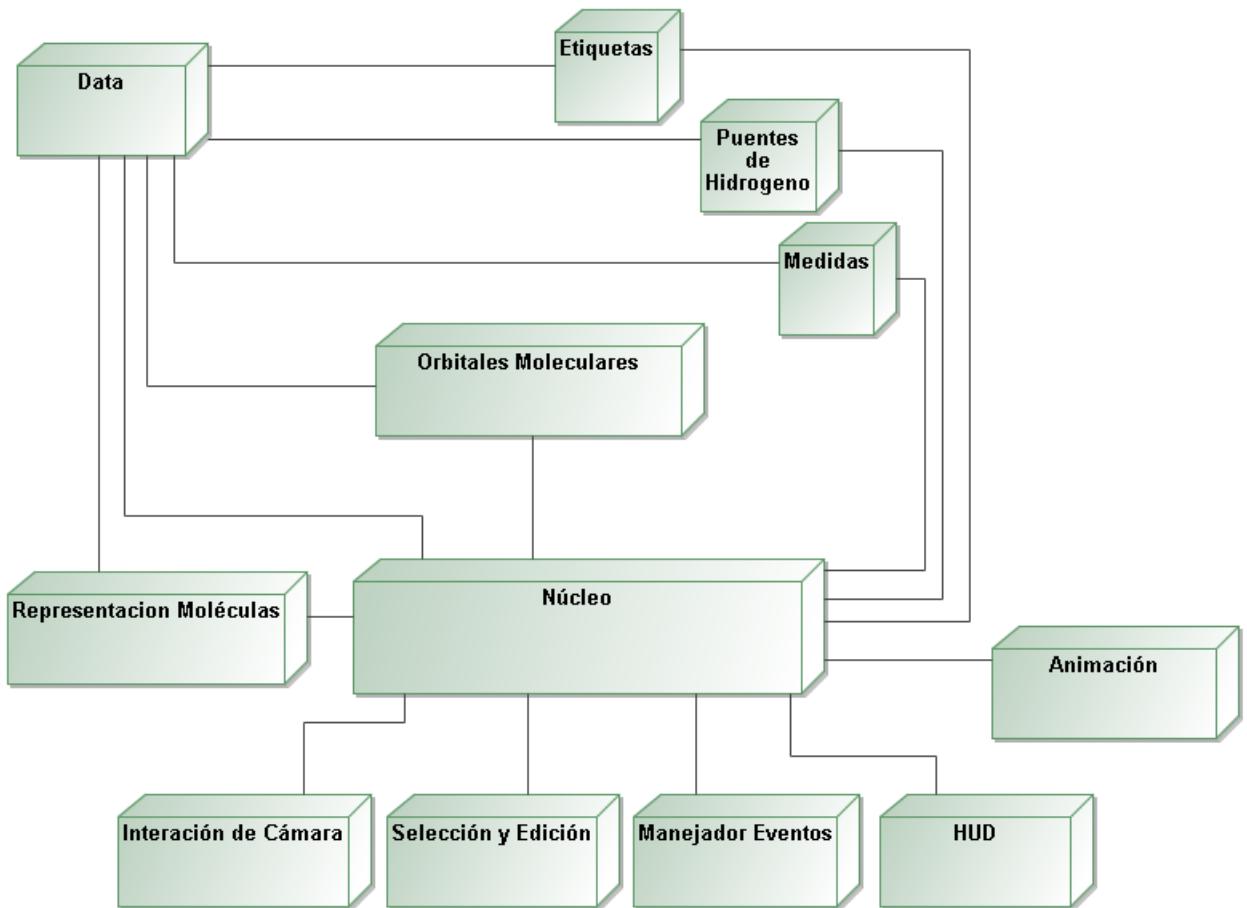


Figura 3.4: Esquema *VisorGL* tras la 8^a iteración

- **Iteración 8.** Se realiza una iteración completa centrada en pruebas funcionales y pruebas de rendimiento, previas a la integración en *BrandyMol*. Variando todo tipo de diversidad y tamaños de moléculas. Verificando todos las posibles acciones del visor, que como recordamos, se desarrolla en un intérprete, con todos los problemas de depuración que ello conlleva.
- **Iteración 9.** Integración del visor en *BrandyMol*, sustituyendo el anterior en *VTK*.
- **Iteración 10.** Se realiza otra iteración completa centrada en pruebas. Éstas ahora versarán sobre la correcta integración con el *fron-end*. Deben poder realizarse

correctamente todas las tareas de alto nivel que éste ofrece.

- **Iteración 11.** Se prepara la aplicación para ser usada en otros sistemas. Se generan ejecutables para lanzar los scripts sobre un intérprete de *Tcl/Tk* y se generan instaladores para *Windows* mediante *InnoSetup*, donde esta incluido el propio intérprete, *scripts*, librerías y todo lo necesario.
- **Iteración 12.** Se generan ayudas y se distribuye a grupos de usuarios finales para su testeо.
- **Iteración 13.** Corrección de incidencias aparecidas en la fase anterior y verificación del sistema-

3.6. Implementación de VisorGL

Tras haber comprendido el problema, habiendo hecho un rápido viaje en la evolución del proyecto y comprendido la metodología de desarrollo, presentaremos su estructura interna, así como detalles de implementación. Será el apartado más extenso de esta memoria donde describiremos los principales procedimientos e ilustraremos con ejemplos del visor 3D.

VisorGL ha sido implementado en un único fichero *VisorGL.tcl*, no obstante requiere de algunos otros de la versión anterior, puesto que su funcionamiento no varía en absoluto. Puede consultarse [5]

Árbol del directorio de BrandyMol

- Directorio de instalación. Ej:
c:/BrandyMol
 - bin
 - chm
 - data
 - tmp

Ficheros de VisorGL 1.5

- Conf.tcl
- Fich.tcl
- Dataf.tcl
- VisorGL.tcl

Figura 3.5: Esquema de directorios y ficheros de la aplicación

Nombre del Fichero: VisorGL.tcl

Librerías Requeridas: tcl3d

Directorio: Dir. Aplicación/bin

Define el Namespace: VisorGL

Descripción: Módulo que define una clase que implementa el Visor Molecular 3D. Carga moléculas contenidas en instancias de *Data*. Incorpora una completa interacción, y una amplia interfaz de comunicación con el exterior.

```
base(): array tcl que representara la instancia del visor
```

```
base(baseConf)
```

```
#GENERAL
```

```
base(backColor)
```

```

#CAMARA
    base(camaraPos)
    base(camaraFocal)
    base(camaraUp)
    base(viewAngle)
    base(zoomPersp)
    base(zoomParal)
    base(viewport)
    base(proyeccion)

#RATON
    base(ratonPosAnt)
    base(ratonPos)

#DATOS
    base(moleculas)
    base(moleculasMostrar)

#TCL3D
    base(vec3i)
    base(vec4i)
    base(vec3f)
    base(vec4f)
    base(vec4d)
    base(mat16)
    base(mat16v)

#INTERACCION
    base(modoAct)
    base(SHIFT)
    base(CTRL)
    base(ratonIzqPuls)
    base(ratonDchoPuls)
    base(dobleClick)
    base(ratonMov)
    base(agregaSeleccion)
    base(actorUpcMover)

```

```
base(ficheroSeleccionado)

#ANIMACION
base(idleId)
base(idleTick)
base(idleClockFrame1)
base(idleClockFrame2)
base(fpsMostrar)

#OPENGL DISPLAY LISTS
base(useDisplayList)
base(molList)
base(molListSelect)
base(hudList)
base(medidasList)
base(etqList)
base(phList)
base(ejesList)
base(orbitalList)
base(cuboList)
base(seleccionList)
base(rotacionList)
base(displaysOrbitales)

#CAMBIOS
base(molModificadas)
base(molModificadasSelect)
base(hudModificada)
base(medidasModificadas)
base(etqModificadas)
base(phModificados)
base(seleccionModificadas)
base(rotacionModificadas)

#SELECCION
base(colorSeleccion)
base(rectanguloSeleccion)
```

```
base(posInicioSeleccion)
base(posFinSeleccion)

#MOVIMIENTO MOLECULAS
base(posInicioMovimiento)
base(posFinMovimiento)
base(moverSeleccion)

#ROTACION MOLECULAS
base(listaMolRotar)
base(matrizRotar)

#REPRESENTACION
base(radioAtC)
base(radioAtCB)
base(wireframe)
base(resolucion)
base(escalaCPK)

#MEDIDAS
base(atomosMedidas)
base(distancias)
base(angulos)
base(torsiones)

#EDICION
base(moverAtomos)
base(listaMoverAtomos)

#ETIQUETAS
base(etqId)
base(etqCodB)
base(etqCarga)
base(etqCodTink)
base(etqQuira)

#PUENTES DE HIDROGENO
```

```

base(listaPuentesH)
base(elemsPuentesH)
base(distPuentesH)
base(phMostrar)

#ORBITALES MOLECULARES
base(orbitalMostrado)
base(modoTranspOrbital)
base(modoRepOrbital)

#HUD
base(hudMostrar)

#EJES COORDENADAS
base(ejesMostrar)

#ANIMACION PRESENTACION
base(presentacion)
base(speedPres)
base(progressPres)

```

3.6.1. Estructura Global. Motor Gráfico

Las librerías *tcl3d* proveen un *widget Tk* que maneja una ventana gráfica *OpenGL*. Se puede utilizar como cualquier otro componente de *Tk* para embeberlo en nuestras aplicaciones. Habitualmente se le provee 3 rutinas: **createproc** encargada de la inicialización de parámetros y variables, **reshapeproc** invocada ante redimensionamientos en la ventana y **displayproc** ejecutada cada vez que sea necesario renderizar la ventana. Una llamada típica tendría la siguiente forma:

```

togl $window.visor
-width 650
-height 650
-double true
-depth true
-createproc    "VisorGL::inicializarVisorGL <window>"
-reshapeproc   "VisorGL::reshapeVisorGL      <base>"
```

```
-displayproc "VisorGL::displayVisorGL      <base>"
```

Préviamente será necesario crear una instancia de *VisorGL*, de nombre *<base>* mediante una llamada a su constructor. Posteriormente en la sección de integración se explicará en más detalle.

Veremos a continuación una descripción de los principales procedimientos que definen la estructura global y el motor gráfico.

proc newVisorGL { base baseConf }

Constructor. Genera una nueva instancia de *VisorGL* declarando un *array* de nombre **base**. Devolverá -1 en caso de ya existir o 1 en caso contrario. Inicializa todas las variables necesarias para el posterior uso del visor.

```
if {[info exists VisorGL::$base]==0} {
    variable $base

    set ${base}(baseConf) $baseConf

    #GENERAL
    set ${base}(backColor)      "0.1 0.2 0.4"

    #CAMARA
    set ${base}(camaraPos)      "0.0 0.0 5.0"
    set ${base}(camaraFocal)     "0.0 0.0 0.0"
    set ${base}(camaraUp)        "0.0 1.0 0.0"
    set ${base}(viewAngle)       60.0

    ...
}
```

proc delVisorGL { base }

Destructor. Elimina la instancia **base** del *VisorGL*, en caso de no existir, no realizará ninguna acción. Elimina tanto los datos moleculares, como toda la estructura del visor. Libera todos los recursos reservados.

proc inicializarVisorGL { base w }

Este procedimiento es llamado por el *widget* de *Tk togl*. Se encarga de inicializar las

variables internas de *OpenGL* necesarias. Colores, materiales, luces, *buffers* a utilizar, fuentes de texto, funciones de mezclado...

```
#Configuro widget
$visor(togl) configure
    -width [lindex $visor(viewport) 0] \
    -height [lindex $visor(viewport) 1] \
    -displayproc "VisorGL::displayVisorGL $base" \
    -reshapeproc "VisorGL::reshapeVisorGL $base" \
    -double true \
    -depth true

#Inicializo fuentes
set visor(fontText3D) [$w loadbitmapfont [format "-*-%...
set visor(fontText2D) [$w loadbitmapfont [format "-*-%...

#Limpio Buffer de color
glClearColor [lindex $visor(backColor) 0] \
    [lindex $visor(backColor) 1] \
    [lindex $visor(backColor) 2]
    0

#Limpio buffer de profundidad
 glEnable GL_DEPTH_TEST

#Activo materiales
 glEnable GL_COLOR_MATERIAL

#Establezco funcion de mezclado para transparencias
 glEnableFunc GL_SRC_ALPHA GL_ONE_MINUS_SRC_ALPHA

#Establezco luces
 establecerLuces $base

 reshapeVisorGL $base $visor(togl) [lindex $visor(viewport) 0] \
    [lindex $visor(viewport) 1]
```

...

proc reshapeVisorGL { base toglwin w h }

Este procedimiento se ejecuta ante los eventos de redimensionamiento de ventana. Actualiza el *viewport* de *OpenGL* para adaptar la ventana al nuevo tamaño.

```
set visor(viewport) "$w $h"
glViewport 0 0 $w $h
set visor(hudModificada) 1
$toglwin postredisplay
```

proc displayVisorGL { base w }

Procedimiento que se ejecuta para el *renderizado* de la pantalla. Establece la función de proyección, limpia los *buffers* necesarios, y chequea los *flags* para todas las características del visor. Moléculas, medidas, selecciones... en busca de cambios. De no haberlos, ejecuta la *display list* correspondiente. En caso contrario, la actualiza según corresponda. Las *display list* son secuencias de llamadas a instrucciones *OpenGL* agrupadas y compiladas para mayor rendimiento. Una vez empaquetadas no pueden modificarse, tan solo reemplazarse. El rendimiento del visor desactivando el uso de estas listas es totalmente inviable, por lo que deben minimizarse las actualizaciones de estas listas. La estructura general de este procedimiento es:

```
#Modo matriz de proyeccion
glMatrixMode GL_PROJECTION
glLoadIdentity

#Establece perspectiva
gluLookAt [lindex $visor(camaraPos) 0] \
            [lindex $visor(camaraPos) 1] \
            [lindex $visor(camaraPos) 2] \
            [lindex $visor(camaraFocal) 0] \
            [lindex $visor(camaraFocal) 1] \
            [lindex $visor(camaraFocal) 2] \
            [lindex $visor(camaraUp) 0] \
            [lindex $visor(camaraUp) 1] \
            [lindex $visor(camaraUp) 2]

#Limpia buffer de color
```

```

glClearColor [lindex $visor(backColor) 0] \
    [lindex $visor(backColor) 1] \
    [lindex $visor(backColor) 2] \
    0

glClear [expr $::GL_COLOR_BUFFER_BIT | $::GL_DEPTH_BUFFER_BIT]

#Modo matriz de modelo-vista
glMatrixMode GL_MODELVIEW
glLoadIdentity

#Moleculas
if { $visor(molModificadas) == 1 || $visor(molList) == -1 } {
    crearDisplayListMol $base
    set visor(molModificadas) 0
}
glCallList $visor(molList)

...
#Selecciones Multiples para translaciones
if { $visor(seleccionModificadas) == 1 } {
    set visor(seleccionModificadas) 0
    crearDisplayListSeleccion $base
}

#Orbitales
if {$visor(orbitalMostrado) != -1} {
    set dl [lindex $visor(displaysOrbitales) $visor(orbitalMostrado)]
    glPolygonMode GL_FRONT_AND_BACK $visor(modorepOrbital)
    glCallList $dl
    glPolygonMode GL_FRONT_AND_BACK GL_FILL
}

...
#Medidas

```

```

if { $visor(medidasModificadas) == 1 || $visor(medidasList) == -1 } {
    crearDisplayListMedidas $base
    set visor(medidasModificadas) 0
}
glCallList $visor(medidasList)

```

...

```

#Intercambio de buffers y volcado a pantalla
$w swapbuffers
glFinish

```

```

proc crearDisplay<caracteristica> { base } y
proc delDisplay<caracteristica> { base }

```

Estos procedimientos actualizan la *display list* de la *característica* concreta. Por ejemplo, si el *flag* **molModificadas** se encuentra a **1**, será necesario destruir la *display list* correspondiente, y crearla pintando las moléculas manualmente. Posteriormente y mientras no se modifique nuevamente el *flag* se pintará directamente la *display list*.

```

proc crearDisplayListMol { base } {
    upvar #0 VisorGL::$base visor
    if { $visor(molList) != -1 } {
        glDeleteLists $visor(molList) 1
    }
    set visor(molList) [glGenLists 1]
    glNewList $visor(molList) GL_COMPILE

    #dibujo manual
    dibujarMoleculas $base $visor(baseConf)

    glEndList
}


```

```

proc delDisplayListMol { base } {
    upvar #0 VisorGL::$base visor
    glDeleteLists $visor(molList) 1
    set visor(molList) -1
}

```

Con estas funciones tenemos la base del visor. En las siguientes secciones iremos añadiendo más funcionalidad.

3.6.2. Representación de Moléculas

Es la principal característica. Las moléculas se componen de átomos químicos y enlaces entre ellos. Cada átomo pertenece a un elemento químico, y a cada uno se le ha asociado un color determinado en el módulo *Conf*. Se proporcionan cuatro modos de representación diferentes:

- **Modo Líneas:** Los átomos no tienen representación. Los enlaces se representan como líneas simples con extremos del color de los átomos que unen.
- **Modo Cilindros:** Los átomos no tienen representación. Los enlaces se representan como cilindros con extremos del color de los átomos que unen.
- **Modo Cilindros y Bolas:** Los átomos se representan como esferas de radio y color variable según el elemento. Los átomos se representan igual que el modo *Cilindros*.
- **CPK:** Los átomos se representan como esferas de tamaño variable según el elemento y escalados según un factor constante que responde a cuestiones químicas.

El estado de representación anterior se aplica a cada átomo de la molécula por separado. Siendo posible representaciones mixtas, que serán de importante ayuda en tareas de más alto nivel como los *mapas de reacción*. La función `proc dibujarMolecula { ... }` es la encargada del dibujado.

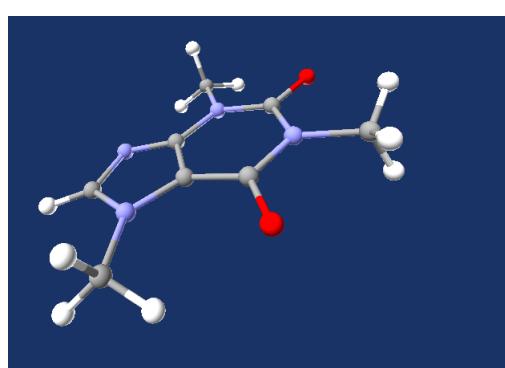


Figura 3.6: Cafeína

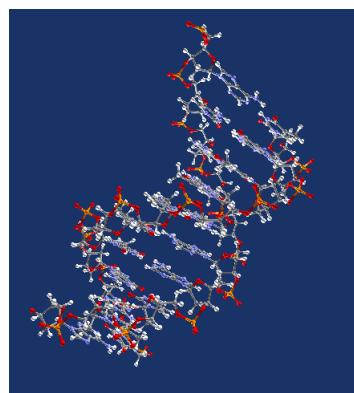


Figura 3.7: Segmento de ADN

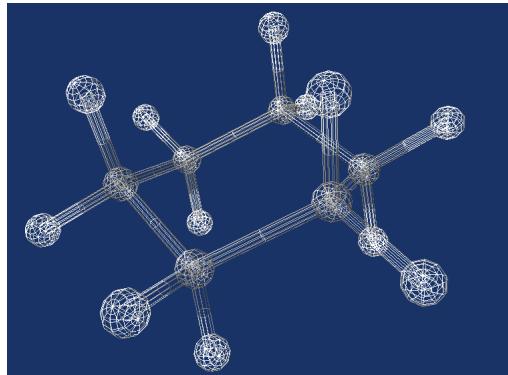


Figura 3.8: Ciclohexano en modo alambre

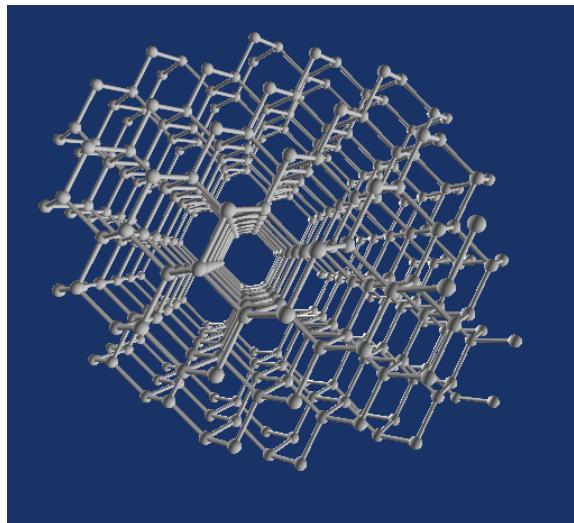


Figura 3.9: Diamante

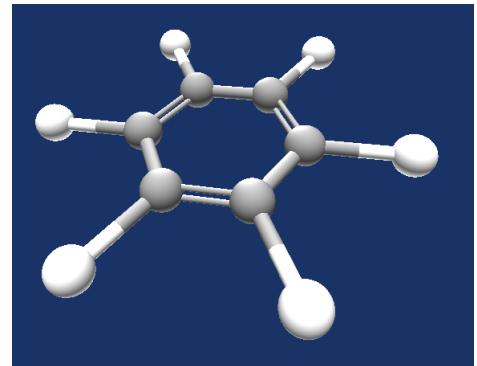


Figura 3.10: Benceno

Representación de Átomos

Para cada átomo de la molécula, dibujamos una esfera en la posición que indique su *Data*, seleccionamos el color correspondiente al elemento, o el color de selección en caso de que éste se haya seleccionado y dibujamos una esfera del radio y resolución que corresponda a la representación local de ese átomo. Nótese el cuarto parámetro de la función `glColor4f` es **0.5**. Éste representa la transparencia, que solo es tenida en cuenta si es ejecutada entre las primitivas `glEnable GL_BLEND` y `glDisable GL_BLEND` como veremos más adelante.

```
for {set x 0} {$x < $datos(numAtomos)} {incr x} {
    set radio [seleccionaRadioRepresentacionAt $base $baseData $x]
    if { $visor(visible,$baseData,$x) && $radio > 0} {
```

```

glPushMatrix
glTranslatef $datos(coordX,$x) $datos(coordY,$x) $datos(coordZ,$x)

#selecciona color
if {[lsearch -exact $visor(atomosSeleccionados,$baseData) $x] \
    == -1} {
    set color [colorToRGB $datos(colorAtom,$x)]
} else {
    set color $visor(colorSeleccion)
}

glColor4f [lindex $color 0] [lindex $color 1] \
    [lindex $color 2] 0.5
gluSphere $quadObj $radio $visor(resolucion) $visor(resolucion)
glPopMatrix
}
}

```

Representacion de Enlaces

Los enlaces se representan mediante cilindros o líneas, según el modo de representación activo. Se dibujan dos secciones de colores correspondientes a los extremos de los átomos. Se sitúan y escalan de manera que queden orientados en la línea recta que uniría a ambos elementos. Los enlaces pueden ser de tres tipos. Simples, dobles ó triples, y serán representados mediante una, dos ó tres líneas o cilindros según correspondan.

```

for {set i 0} {$i < $datos(numAtomos)} {incr i} {
    foreach j $datos(conect,$i) {
        #para no repetir los enlaces
        if { $i < $j && $visor(visible,$baseData,$i) && \
            $visor(visible,$baseData,$j)} {

            ... Calculo para escalar y posicionar los enlaces correctamente
            glTranslatef ...
            glRotatef ...

            #segun el tipo de enlace, simple, doble o triple

```

```

switch $datos(tipoConect,$i,$j) {
    ....
    pintaEnlace $base $quadObj $radio $longSeg
    ....
}
...
}

```

Donde el procedimiento `pintaEnlace` distingue entre los diferentes modos, pintando cilindros o líneas

```

proc pintaEnlace { base quad radio long } {
    upvar #0 VisorGL::$base visor
    if { $radio == 0 } {
        glBegin GL_LINES
        glVertex3f 0.0 0.0 0.0
        glVertex3f 0.0 0.0 $long
        glEnd
    } else {
        gluCylinder $quad $radio $radio $long $visor(resolucion) 1
    }
}

```

En las siguientes figuras observamos varios ejemplos de los diferentes modos de representación

3.6.3. Interacción y Selección

Hasta ahora tenemos un visor con buena capacidad representativa pero una nula interacción. Incorporaremos para ello manejadores de teclado y ratón que permitirán al usuario interaccionar con el visor a varios niveles. Se asocian los eventos de teclado y ratón a funciones *Tcl* del siguiente modo:

```

bind $frameright.visor <ButtonRelease-3> \
    "GUI::manejadorLBRE $base"
bind $frameright.visor <ButtonPress-1> \
    "VisorGL::manejadorLBPE $gui(visor) %x %y"

```

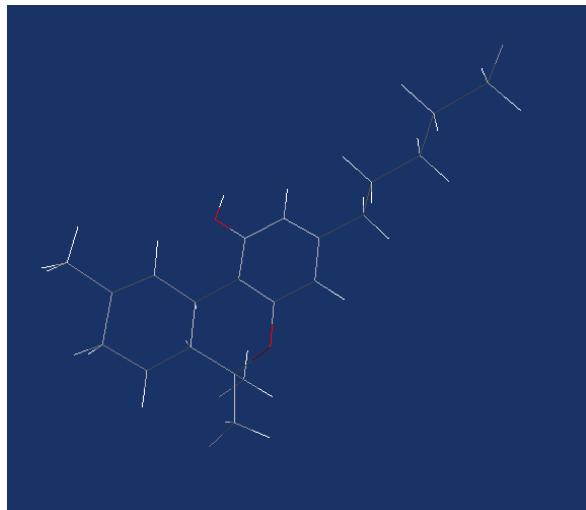


Figura 3.11: Molécula de THC en modo líneas

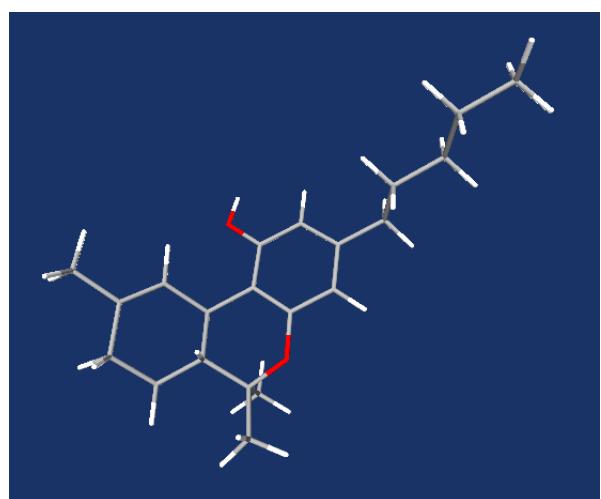


Figura 3.12: Molécula de THC en modo cilindros

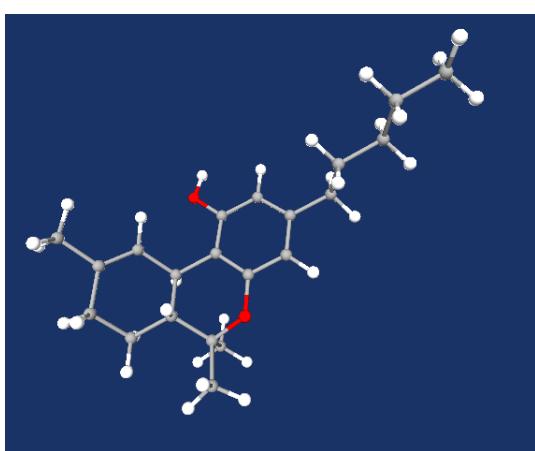


Figura 3.13: Molécula de THC en modo Cilindros y Bolas

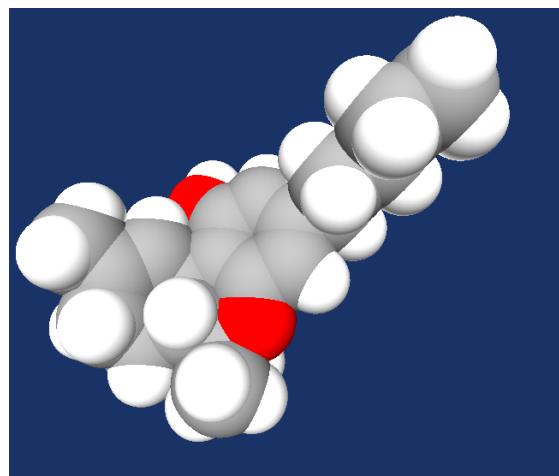


Figura 3.14: Molécula de THC en modo CPK

```

bind $frameright.visor <ButtonRelease-1> \
    "VisorGL::manejadorLBRE $gui(visor) %x %y"
bind $frameright.visor <B1-Motion> \
    "VisorGL::manejadorMME $gui(visor) %x %y"
bind $frameright.visor <B3-Motion> \
    "VisorGL::manejadorMME $gui(visor) %x %y"
bind . <MouseWheel> \
    "VisorGL::manejadorScroll $gui(visor) %D"
bind $frameright.visor <2> \
    "VisorGL::camaraReset $gui(visor)

```

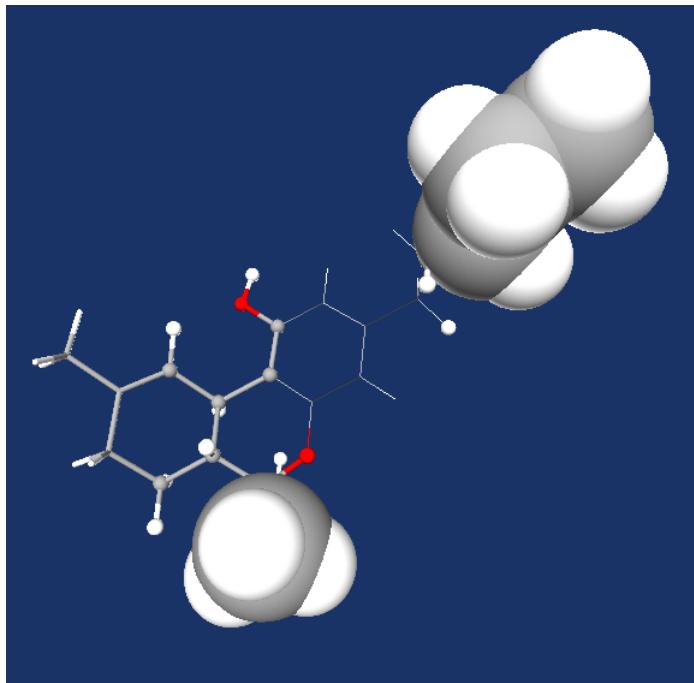


Figura 3.15: Molécula de THC en modo de representación Híbrido

```

bind . <KeyPress> \
    "VisorGL::manejadorKPE $gui(visor) %K"
bind . <KeyRelease> \
    "VisorGL::manejadorKRE $gui(visor) %K"
bind $frameright.visor <Double-Button-1> \
    "VisorGL::manejadorDCE $gui(visor)"

```

Interacción de cámara

El modo de interacción de cámara no realiza cambios sobre las estructuras moleculares, tan solo en la cámara que visualiza la escena. Se permiten las siguientes acciones:

- **Rotación:** El ratón permite realizar movimientos acimutales (la cámara se mueve alrededor del centro de la escena conservando el mismo radio). Es la opción por defecto que permite la visualización más sencilla.
- **Traslación:** El movimiento horizontal y vertical del ratón desplaza el centro de la escena, el cual es utilizado para posteriores acciones como rotaciones.
- **Zoom:** El movimiento vertical del ratón acerca y aleja la cámara al centro de la escena. En el caso de la proyección en perspectiva se abre el ángulo del campo visual

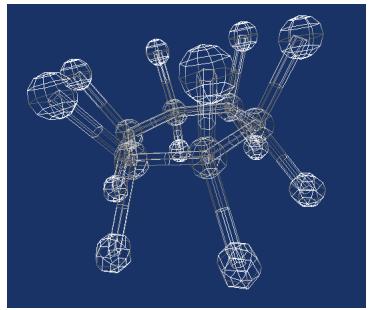


Figura 3.16: Molécula en resolución baja

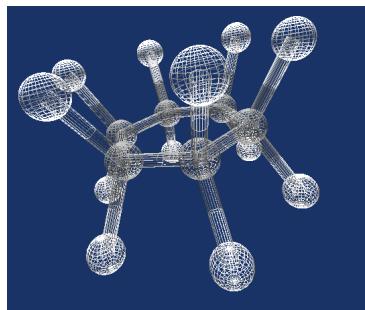


Figura 3.17: Molécula en resolución media

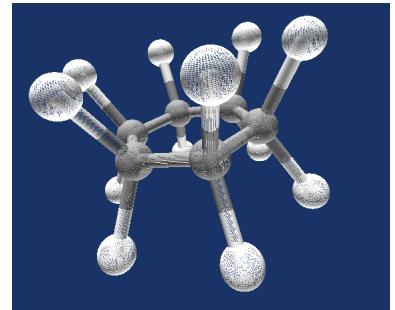


Figura 3.18: Molécula en resolución alta

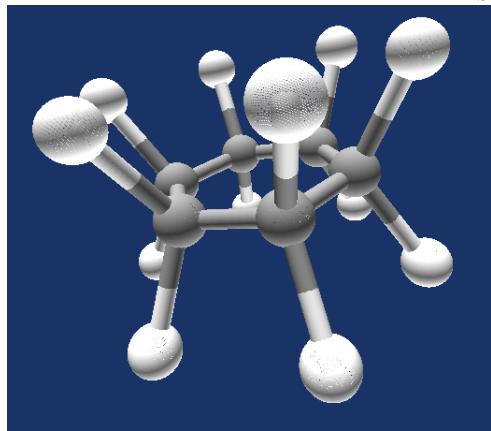


Figura 3.19: Molécula en resolución ultra

del *frustum*.

```
gluPerspective [expr $visor(viewAngle)*$visor(zoomPersp)] \
[expr double([lindex $visor(viewport) 0])/ \
double([lindex $visor(viewport) 1])] \
0.2 100.0
```

En la proyección ortográfica se amplía el volumen de visualización:

```
set rel double([lindex $visor(viewport) 1])/ \
double([lindex $visor(viewport) 0])
glOrtho [expr -4.0*$visor(zoomParal)] \
[expr 4.0*$visor(zoomParal)] \
[expr -4.0*$visor(zoomParal)*$rel] \
[expr 4.0*$visor(zoomParal)*$rel] \
0.1 100.0
```

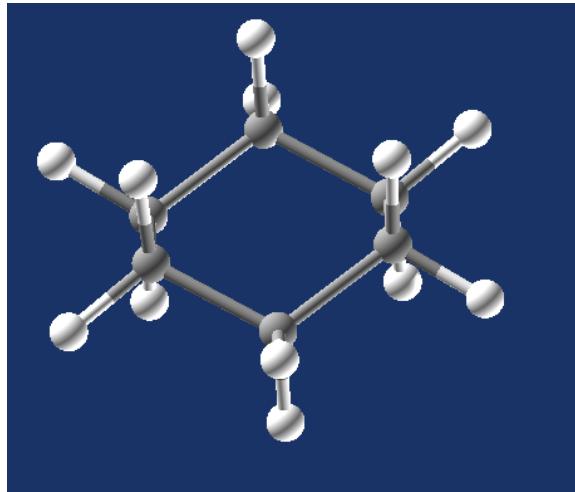


Figura 3.20: Proyección Paralela

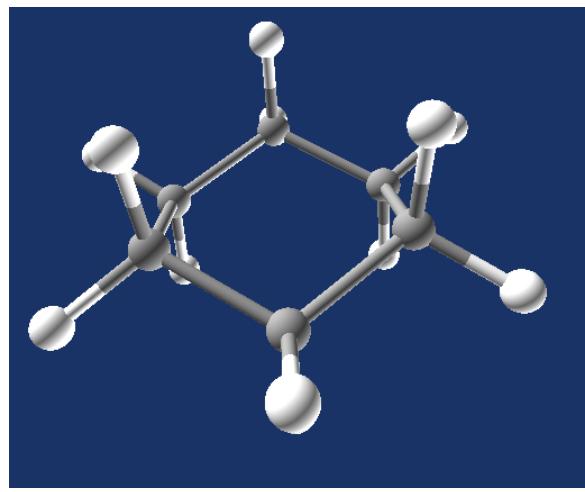


Figura 3.21: Proyección Cónica

- **Centrar:** Esta acción posiciona la cámara de modo que sea visible todo el contenido del visor y sitúa el centro de la escena en el centro del contenido visible. Para ello se calcula el centro del paralelepípedo que encierra a todos los átomos del visor, a partir del cual se aleja la cámara una distancia algo superior a la mitad de su diagonal.

```

set bB [boundingBox $base $visor(moleculas)]

#Calculo posiciones de la camara y centro de escena
...
set visor(camaraPos) ...
set visor(camaraFocal)
...

#Situo la camara
gluLookAt  [lindex $visor(camaraPos) 0] \
            [lindex $visor(camaraPos) 1] \
            [lindex $visor(camaraPos) 2] \
            [lindex $visor(camaraFocal) 0] \
            [lindex $visor(camaraFocal) 1] \
            [lindex $visor(camaraFocal) 2] \
            [lindex $visor(camaraUp) 0] \
            [lindex $visor(camaraUp) 1] \
            [lindex $visor(camaraUp) 2]
    
```

Selección

El visor permite la selección de moléculas, átomos y/o enlaces mediante dos modos. *Click* sobre los elementos del visor individuales, o generación de un rectángulo para selección múltiple. Para realizar la selección *OpenGL* proporciona un *buffer* sobre el que se actúa de la siguiente forma.

1. Habilitamos el *buffer*

```
glSelectBuffer $BUFSIZE $selectBuffer
glRenderMode GL_SELECT
```

2. Recuperamos la matriz de proyección que tengamos en la escena que estamos visualizando para tener la misma orientación.

```
glMatrixMode GL_PROJECTION
glGetDoublev GL_PROJECTION_MATRIX $proyeccion
glPushMatrix
glLoadIdentity

glGetIntegerv GL_VIEWPORT $viewport
```

3. Calculamos el área de selección centrada en *centro* y de amplitud *delta* que variará según sea selección simple o múltiple. Para la selección múltiple se visualiza en tiempo real un rectángulo de líneas discontinuas donde el usuario encierra los elementos que desee.

```
set centro ...
set delta ...
```

4. Se genera una matriz de picado que restringe el dibujado a una pequeña porción del *viewport* y se multiplica por la matriz de proyección recuperada anteriormente.

```
gluPickMatrix [lindex $centro 0] [lindex $centro 1] \
              [lindex $delta 0] [lindex $delta 1] $viewport
glMultMatrixd [tcl3dVectorToList $proyeccion 16]

glMatrixMode GL_MODELVIEW
glPushMatrix
```

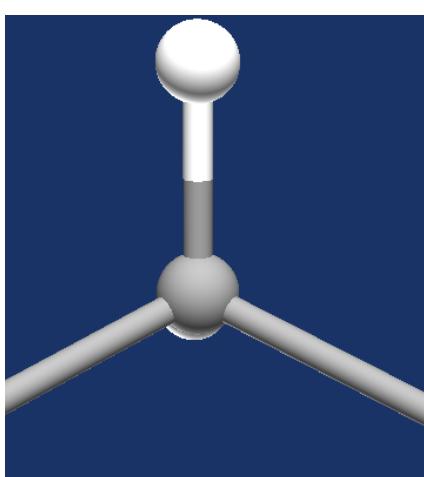


Figura 3.22: Sin selección

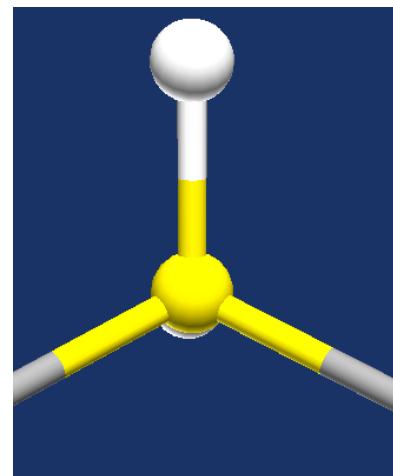


Figura 3.23: Átomo con enlaces múltiples seleccionado

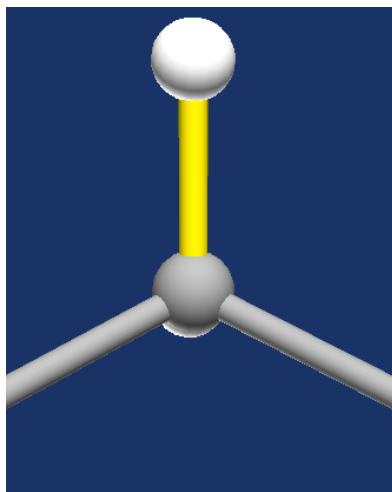


Figura 3.24: Enlace seleccionado

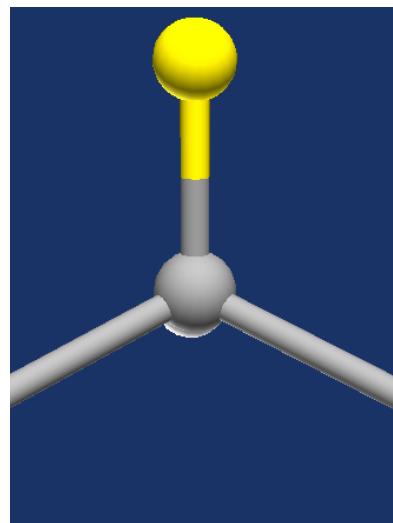


Figura 3.25: Átomo con enlace simple seleccionado

5. Se redibuja la escena generando nombres para cada uno de los elementos. Estos nombres serán los que se recuperen tras la selección para saber sobre qué elementos se ha picado.

```
#Para atomos
glLoadName $n
incr n
gluSphere $quadObj 0.25 10 10
...
#Para enlaces
```

```

glLoadName $n
gluCylinder $quadObj 0.12 0.12 $longSeg 15 1
...

```

6. Se calculan los elementos seleccionados y se ordenan según la profundidad, interpretando como seleccionados los elementos más próximos al *eje Z* si la selección es simple, o todos si la selección es múltiple

```

set hits [glRenderMode GL_RENDER]
if {!$multiple} {
    return [processHits $hits $selectBuffer]
} else {
    return [processHitsMultiple $hits $selectBuffer]
}

```

Una vez finalizada la selección se colorean los elementos de un color predeterminado para su fácil identificación. Es posible también deseleccionar elementos haciendo simple *click* sobre zonas del visor que no contengan nada, o sobre los propios elementos seleccionados. La selección permite acciones de edición dentro del propio visor, o recuperación de información de las partes de la molécula seleccionada para acciones posteriores, por ejemplo toma de medidas, mostrar datos de elementos concretos...

3.6.4. Edición

El visor permite pequeñas labores de edición para su uso habitual. No es en absoluto una aplicación de modelado en 3D, el diseño de moléculas se suele realizar en programas especializados y en 2D como *MDL Isis Draw*. Las acciones posibles son las siguientes:

- **Ocultación de Hidrógenos:** Se muestran u ocultan los hidrógenos seleccionados o los que estén conectados a elementos seleccionados. Esto permite una fácil visualización en ámbitos químicos en los que solo se desea ver la estructura básica de la molécula. Es necesario ocultar los enlaces a dichos hidrógenos.
- **Rotación Relativa:** A diferencia de la rotación de cámara alrededor de la escena, se permite la rotación de moléculas respecto de las demás. Para ello se crea una *display list* con el contenido seleccionado y se muestra activando la transparencia para una fácil distinción. Ello se consigue mediante la activación de la función de mezclado.

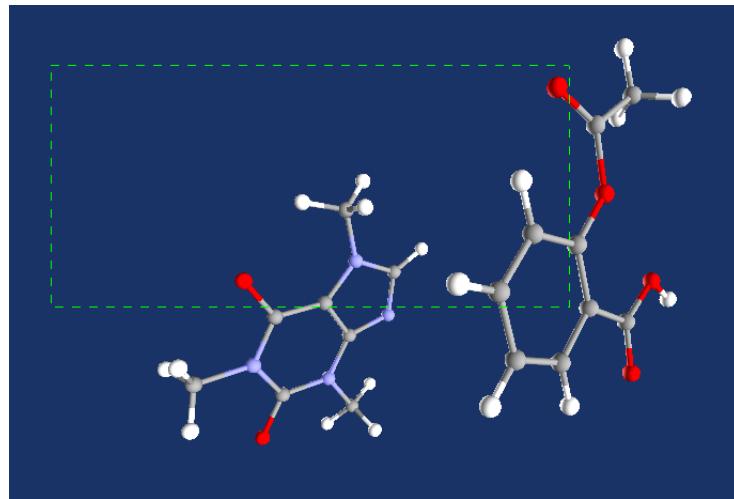


Figura 3.26: Rectángulo de selección múltiple

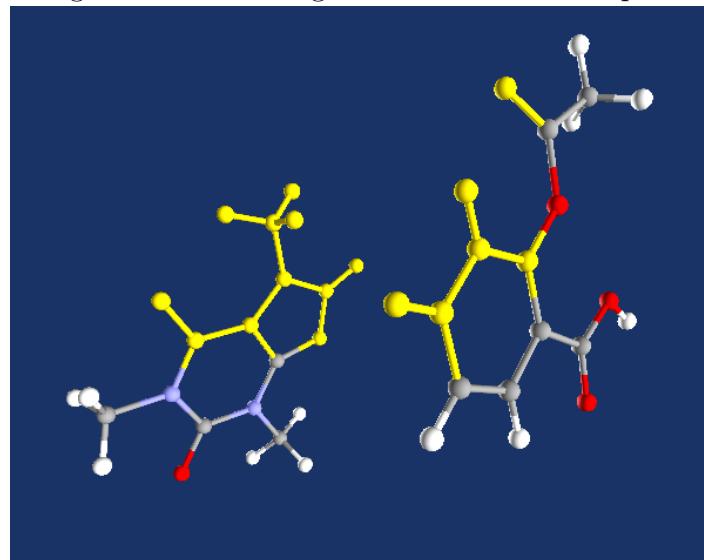


Figura 3.27: Selección Múltiple

```

glEnable GL_BLEND
glBlendFunc GL_SRC_ALPHA GL_ONE_MINUS_SRC_ALPHA

... dibujado de la molécula

glDisable GL_BLEND

```

Las rotaciones se acumulan en la matriz `visor(matrizRotar)` y una vez se suelte el botón del ratón, se efectuarán los cambios en la molécula original. Es posible la rotación en los tres ejes *X*, *Y* y *Z*. La combinación de diversas teclas modifican los

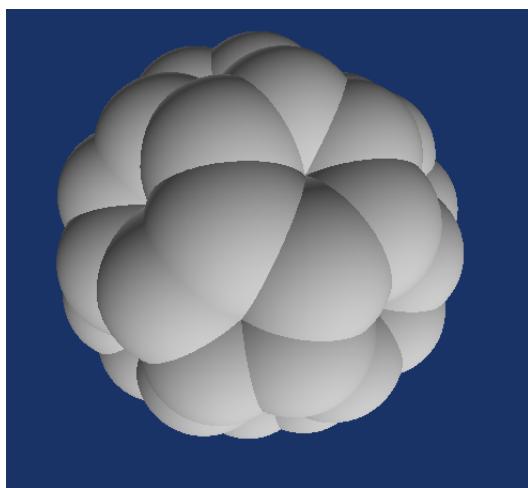


Figura 3.28: Enlace seleccionado

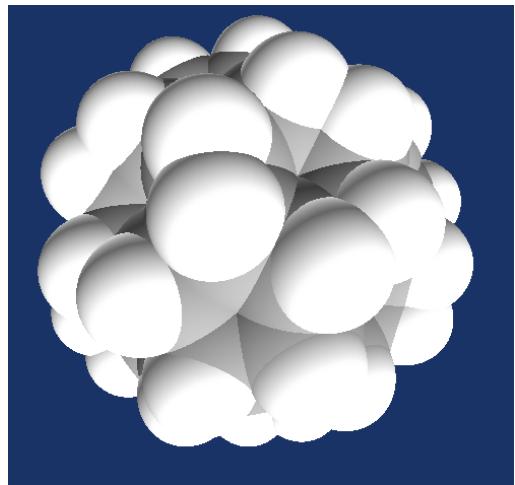


Figura 3.29: Átomo con enlace simple seleccionado

ejes sobre los cuales se rota. Puede verse con más detalle en [5].

```
#Rotacion en X e Y de lo seleccionado
set mX [calculaRotacionEjeAng $base "X" [expr -$difY / 2.0]]
set mY [calculaRotacionEjeAng $base "Y" [expr -$difX / 2.0]]
set mRXY [math::linearalgebra::matmul $mY $mX]
set visor(matrizRotar) [math::linearalgebra::matmul \
                        $mRXY \
                        $visor(matrizRotar)]
```

...

```
#Aplicacion de cambios
foreach mol $listBaseData {
    upvar #0 Data::$mol datos
    for {set x 0} {$x < $datos(numAtomos)} {incr x} {
        set pos [list $datos(coordX,$x) \
                  $datos(coordY,$x) \
                  $datos(coordZ,$x)
                  1.0]
        set npos [math::linearalgebra::matmul $Res $pos]

        set datos(coordX,$x) [lindex $npos 0]
        set datos(coordY,$x) [lindex $npos 1]
        set datos(coordZ,$x) [lindex $npos 2]
```

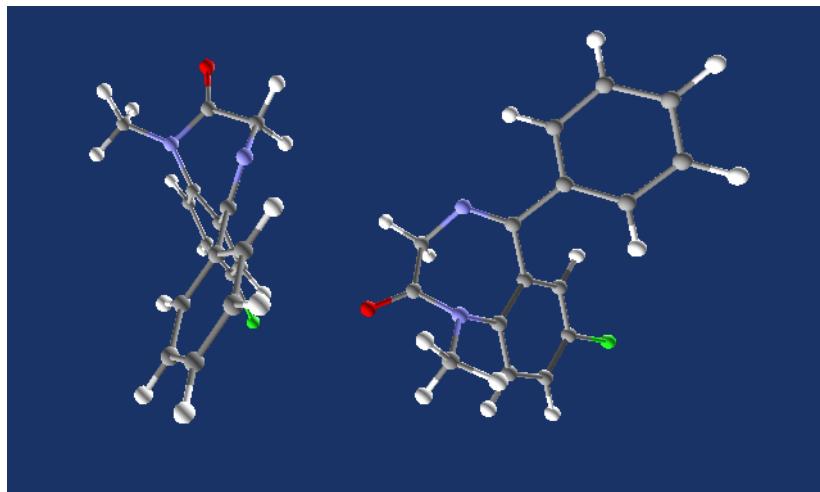


Figura 3.30: Moléculas desorientadas

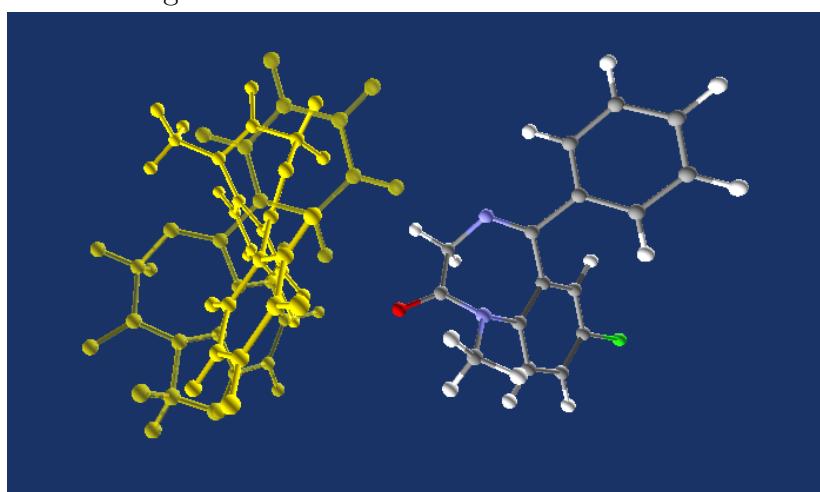


Figura 3.31: Reorientación

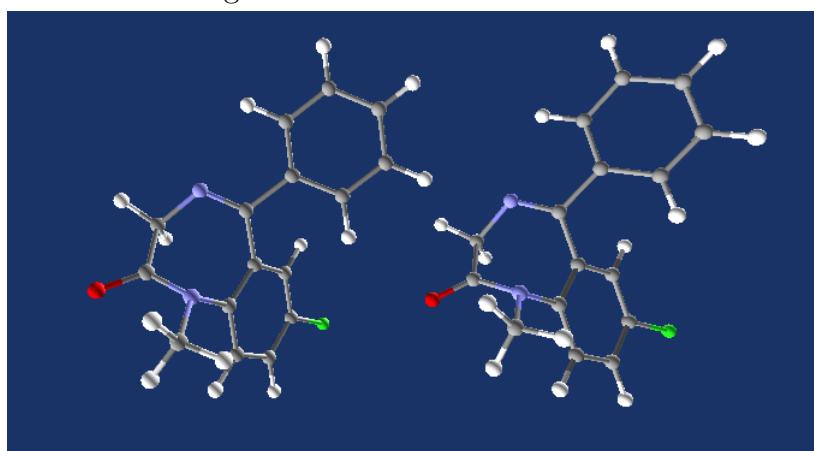


Figura 3.32: Moléculas Reorientadas

```

    }
}

```

- **Desplazamiento Relativo:** Se permite el desplazamiento de partes de la molécula seleccionada o de moléculas completas. El método es similar al anterior acumulando los desplazamientos del ratón del visor, en lugar de las rotaciones realizadas y aplicándolos a la molécula original al soltar el ratón.

```

set visor(moverSeleccion) [math::linearalgebra::add \
$visor(moverSeleccion) \
[math::linearalgebra::scale
-2 \
[vectorDifPosRaton $base \
$visor(ratonPosAnt) $visor(ratonPos)]]]

```

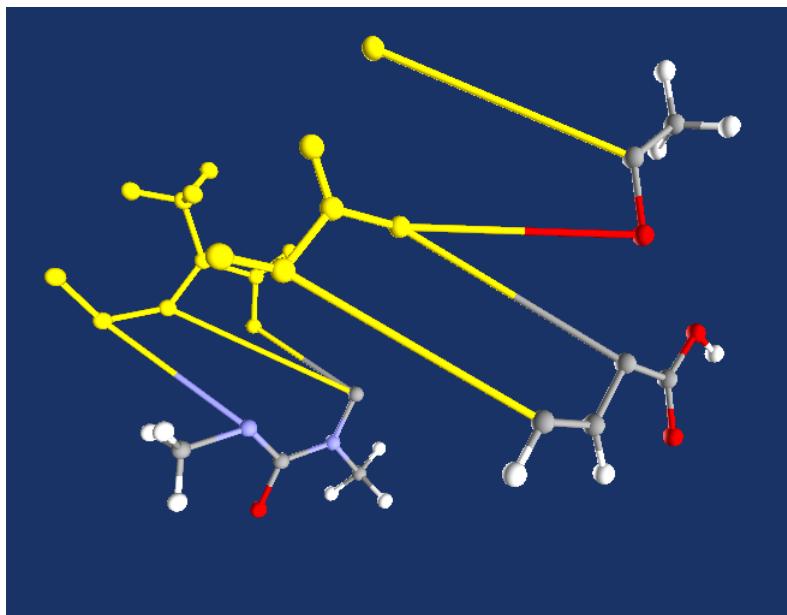


Figura 3.33: Desplazamiento Relativo

3.6.5. H.U.D

El visor cuenta con una *H.U.D* (*Head-Up Display*), un término adoptado de los videojuegos y aplicaciones 3D. Consiste en mostrar información en todo momento en el visor en

forma de letras y números. Se destaca el modo en que se encuentra (rotar, mover, trasladar o zoom), información sobre la versión actual del visor, información sobre el adaptador gráfico y un contador de *FPS* (*frames* por segundo) para el modo animación.

Desde el punto de vista de *OpenGL* es necesario hacer cambios en las matrices de proyección y utilizar primitivas de dibujado de texto. La *H.U.D.* debe estar visible en todo momento haya lo que haya en el visor. Para ello definimos un procedimiento estándar en el que establecemos la posición, el color y el texto en concreto que deseamos pintar.

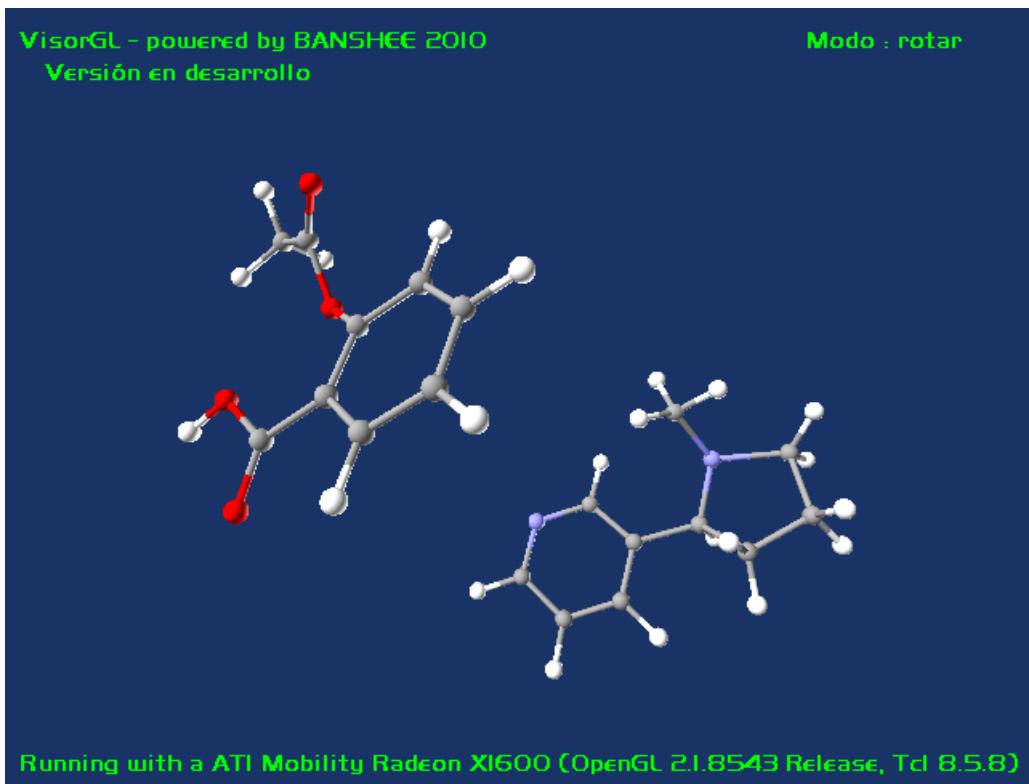


Figura 3.34: Head-Up Display

```

proc escribirCad2D { base pos color cad } {
    upvar #0 VisorGL::$base visor

    glMatrixMode GL_PROJECTION
    glPushMatrix
    glLoadIdentity

    glOrtho 0.0 [expr double([lindex $visor(viewport) 0])] \
        [expr double([lindex $visor(viewport) 1])] \

```

```

0.0 0.0 1.0

glMatrixMode GL_MODELVIEW
glPushMatrix
glLoadIdentity
glDisable GL_LIGHTING

glColor3f [lindex $color 0] [lindex $color 1] [lindex $color 2]
glRasterPos2i [lindex $pos 0] [lindex $pos 1]

glListBase $visor(fontText2D)
set len [string length $cad]
set sa [tcl3dVectorFromString GLubyte $cad]
glCallLists $len GL_UNSIGNED_BYTE $sa
$sa delete

 glEnable GL_LIGHTING
 glPopMatrix

glMatrixMode GL_PROJECTION
glPopMatrix

glMatrixMode GL_MODELVIEW
}

```

3.6.6. Medidas

Haciendo uso de los métodos de selección es posible efectuar diversas medidas sobre elementos del visor. Dependiendo del número de átomos seleccionados de la misma o diferentes moléculas, se calculará una u otra de entre las siguientes.

- **Distancia:** Distancia en *Armstrongs* entre dos átomos seleccionados
- **Angulo:** Ángulo en grados entre tres átomos seleccionados
- **Torsion:** Seleccionados cuatro átomos, la torsión es el ángulo entre el plano que forman los tres primeros átomos seleccionados con los tres últimos.

Destacaremos el modo de dibujado de los semiarcos que representan los ángulos. Para ello se utilizan los *splines* que implementa *OpenGL* de manera nativa. Se le proporcionan tres puntos de control correspondientes a los extremos del semiarco y el punto medio, manteniendo el mismo radio. *OpenGL* se encargará de generar los puntos intermedios sobre los que pintaremos segmentos de líneas.

```
glEnable GL_MAP1_VERTEX_3
glMap1f GL_MAP1_VERTEX_3 0.0 1.0 3 [llength $ctrlPoints] \
[join $ctrlPoints]
glBegin GL_LINE_STRIP
for { set i 0 } { $i <= 30 } { incr i } {
    glEvalCoord1f [expr double ($i)/30.0]
}
glEnd
glDisable GL_MAP1_VERTEX_3
```

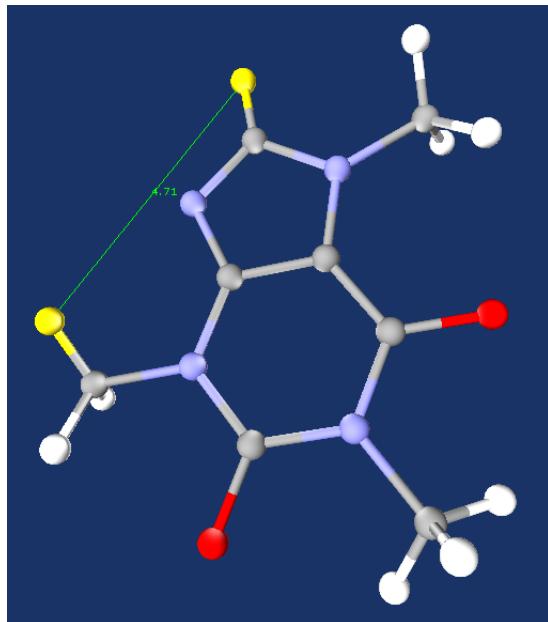


Figura 3.35: Distancia

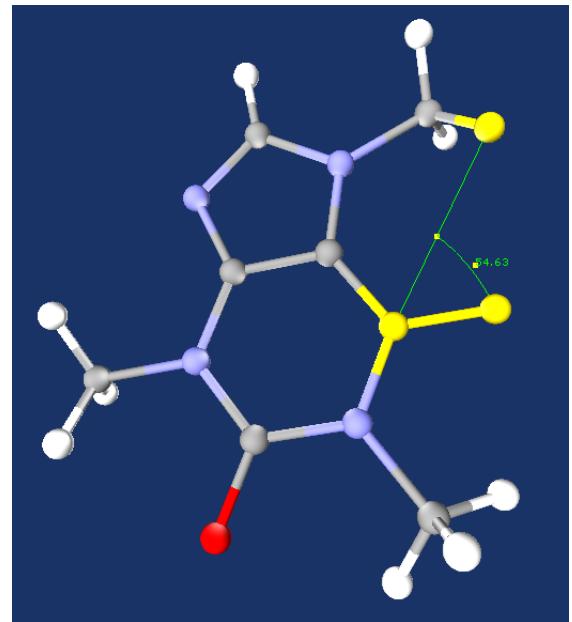


Figura 3.36: Ángulo

3.6.7. Puentes de Hidrógeno

El módulo *Data* calcula los átomos en que se dan las circunstancias para que haya un puente de hidrógeno. El visor representa esta información como líneas discontinuas. Los puentes son recalculados automáticamente si se edita la molécula en el visor.

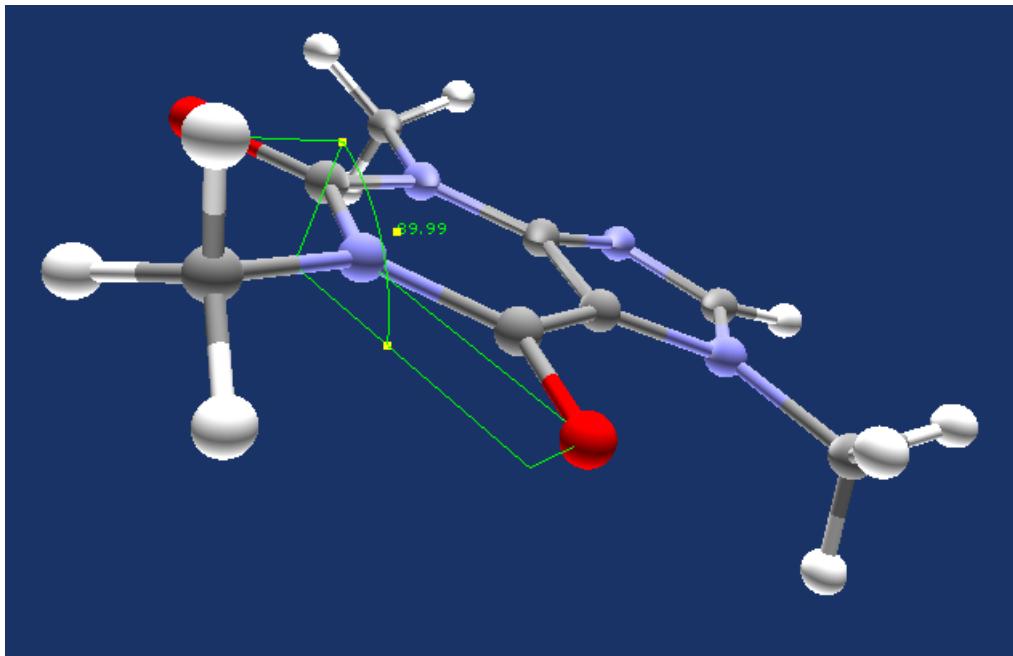


Figura 3.37: Torsión

```

glEnable GL_LINE_STIPPLE
glLineStipple 1 0x00FF

glBegin GL_LINES
    glVertex3f [lindex $p1 0] [lindex $p1 1] [lindex $p1 2]
    glVertex3f [lindex $p2 0] [lindex $p2 1] [lindex $p2 2]
glEnd

glDisable GL_LINE_STIPPLE

```

3.6.8. Etiquetas

El visor permite la visualización de diversas etiquetas (*Identificador del átomo, Código BrandyMol, Carga del átomo, Código Tinker y Quiralidad*). Éstas son representadas como cadenas de texto 3D próximas a la posición de cada átomo. Para ello se define una función estándar en la que se especifica posición, color y cadena de texto.

```

proc escribirCad3D { base pos color cad } {
    upvar #0 VisorGL::$base visor
    glDisable GL_LIGHTING

```

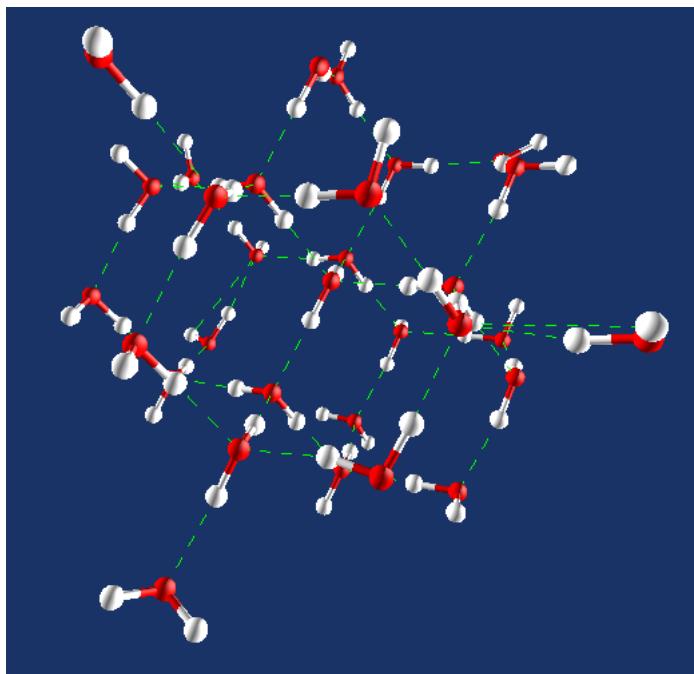


Figura 3.38: Moléculas de agua con Puentes de Hidrógeno

```

glColor3f [lindex $color 0] [lindex $color 1] [lindex $color 2]
glRasterPos3f [lindex $pos 0] [lindex $pos 1] [lindex $pos 2]

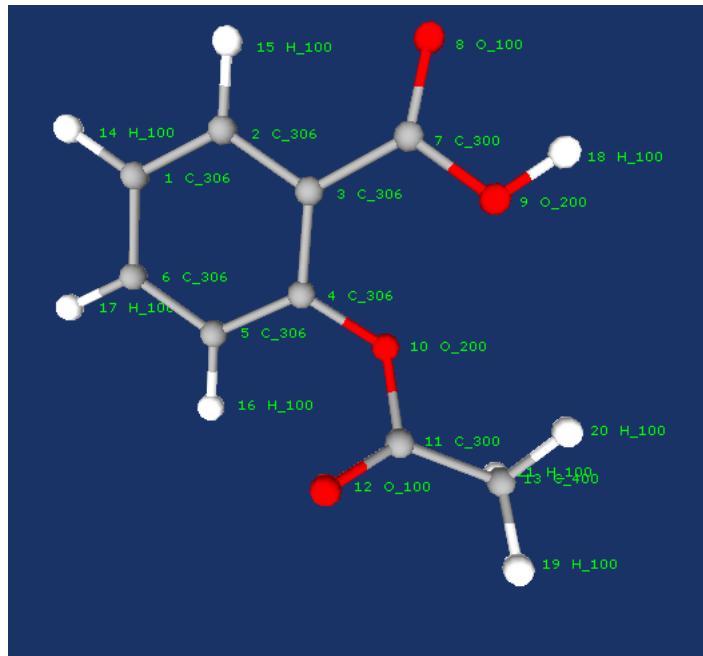
glListBase $visor(fontText3D)
set len [string length $cad]
set sa [tcl3dVectorFromString GLubyte $cad]
glCallLists $len GL_UNSIGNED_BYTE $sa
$sa delete

 glEnable GL_LIGHTING
}

```

3.6.9. Sistema de Referencia

Se incorpora un pequeño eje de referencia en la esquina inferior izquierda del visor para una fácil orientación. Puede activarse o desactivarse a elección del usuario. Son dibujados de tal forma que nunca son ocultados por ninguna molécula, orbital, medida o cualquier objeto creado desde el visor.

Figura 3.39: Molécula con etiquetas *Identificador y Código BrandyMol*

3.6.10. Orbitales Moleculares

Este apartado es uno de los más complejos del visor. Para empezar, se cuenta con un código escrito en lenguaje *Fortran* en el que se define una función de densidad para un espacio cartesiano $f(x, y, z)$, y el objetivo es representar isosuperficies para cada uno de los valores de densidad posibles. En la versión anterior del visor en *VTK* esto era prácticamente automático gracias al alto nivel de las librerías gráficas, pero en este caso queda todo a disposición del programador.

La solución pasa por combinar varios programas:

1. Retocar el código *Fortran* **plotBrandyGL**, para que genere como salida un fichero de texto que representará un cubo de resolución variable que envuelva a la molécula, donde cada punto interior del cubo contendrá el valor de densidad positiva y negativa del orbital que se le pase como parámetro. El cubo se define mediante un origen (x, y, z) , un espaciado (dx, dy, dz) y un número de espacios en cada dimensión (DX, DY, DZ) .

```
WRITE (24,*) 'DIMENSIONS'
WRITE (24,*) INT(SPACES), INT(SPACES), INT(SPACES)
WRITE (24,*) ' '
```

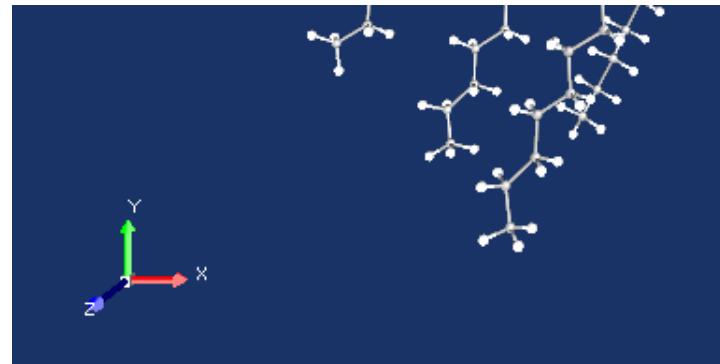


Figura 3.40: Eje de Referencia

```

WRITE (24,*) 'ORIGIN'
WRITE (24,*) XMIN,YMIN,ZMIN
WRITE (24,*) ' '
WRITE (24,*) 'SPACING'
WRITE (24,*) XINC,YINC,ZINC
WRITE (24,*) ' '
WRITE (24,*) 'POINT_DATA ', INT(SPACES)**3
WRITE (24,'(A)') 'SCALARS POS float'
WRITE (24,'(A)') 'LOOKUP_TABLE default'
WRITE (24,'(I5)') (((ABS(INT((DENSITP(NX,NY,NZ)*256)/VALMAX)),
*   NX=1,INT(SPACES)),NY=1,INT(SPACES)),NZ=1,INT(SPACES))
WRITE (24,'(A)') 'SCALARS NEG float'
WRITE (24,'(A)') 'LOOKUP_TABLE default'
WRITE (24,'(I5)') (((ABS(INT((DENSITN(NX,NY,NZ)*256)/
*   VALMAX)),NX=1,INT(SPACES)),NY=1,INT(SPACES)),NZ=1,INT(SPACES))

```

Lo que generará un fichero similar al siguiente, donde cada valor de las *LOOKUP_TABLE* corresponde al valor de densidad en cada punto del cubo definido anteriormente

```

...
DATASET STRUCTURED_POINTS
DIMENSIONS
51 51 51

ORIGIN
-4.66783047 -4.05339289 -5.12696171

```

```

SPACING
0.212765768 0.202441633 0.178248167

POINT_DATA 132651
SCALARS POS float
LOOKUP_TABLE default
0
0
3
0
1
...
SCALARS NEG float
LOOKUP_TABLE default
1
7
5
0
0
...
END

```

2. Implementación del algoritmo **Marching Cubes** [17] para generar mallas de triángulos. El fichero generado anteriormente contiene datos para generar 256 isosuperficies diferentes siendo necesario crear una malla de triángulos entre aquellos puntos que tengan el mismo valor. Una técnica lo suficientemente eficiente para ello es el algoritmo de *Marching Cubes*. Aunque todo el visor se desarrolla en *Tcl*, el algoritmo siguiente se implementa en *C*. El motivo es que *Tcl* resulta terriblemente lento manejando grandes volúmenes de datos, tan solo la carga de los datos del fichero ya lo deja en evidencia, por lo que se descarta la implementación del algoritmo completo. La solución pasa por implementarse en código *C* y posteriormente generar una librería dinámica **MarchingDll** que defina nuevos comandos para el intérprete.

Una primera aproximación es que este programa *C* genere ficheros de texto con información sobre vértices y normales de la superficie, y sea el visor *Tcl* quien los lea e invoque las primitivas *OpenGL* al igual que en el resto de la aplicación. No obstante nuevamente el rendimiento cae de forma drástica debido al volcado de datos en

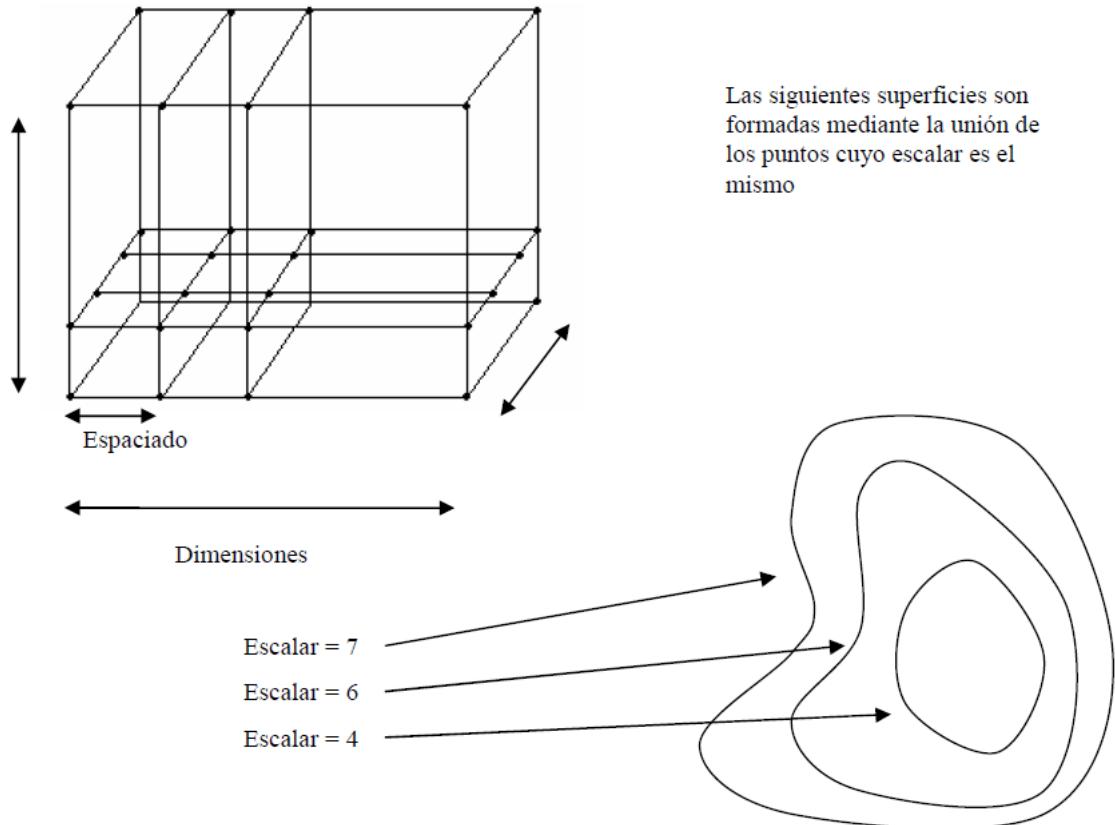


Figura 3.41: Esquema conceptual del cubo de densidad

memoria a ficheros de texto, lectura de estos ficheros por parte de *Tcl* e invocación de las primitivas. Debido a los requerimientos finales de la aplicación es necesario cargar las 256 isosuperficies para permitir modos de visualización interactivos muy rápidos. Los usuarios finales deben poder ver prácticamente en tiempo real como varía la superficie mientras ellos mueven una pequeña *scroll bar*. Se soluciona finalmente invocando directamente las primitivas de *OpenGL* desde el código *C*, ya que *Tcl* no es más que un envoltorio para éstas y los efectos son los mismos.

A pesar de que el algoritmo en sí se puede adaptar fácilmente, el cálculo de las normales debe hacerse a mano. Se utiliza para ello *la técnica del gradiente*. Consiste en calcular las variaciones de densidad en los puntos circundantes al vértice que se está tratando y promediarlos. Como podrá comprobarse en las imágenes adjuntas, los efectos de iluminación que se consiguen son más que suficientes para los requerimientos del visor.

A modo de resumen, se genera una *Dll* para intérpretes *Tcl* que aporta dos nuevos comandos `cargarFicheroOrbital` que lee un fichero con el formato definido en el punto anterior, y `pintarSuperficieIsovalor` que ejecutará el algoritmo *Mar-*

ching Cubes sobre el conjunto de datos e invocará directamente a las primitivas `glNormal` y `glVertex`. Será tarea del visor *Tcl* invocarlo dentro de las primitivas `glBegin GL_TRIANGLES ... glEnd GL_TRIANGLES` o alguna otra, siendo de esta manera más reutilizable.

Parte del código correspondiente a la implementación del algoritmo:

```
GLvoid vMarchCube1(int iX, int iY, int iZ, int orb){
    ...
    //Find which vertices are inside of the surface \
    //and which are outside
    iFlagIndex = 0;
    for(iVertexTest = 0; iVertexTest < 8; iVertexTest++){
        if(afCubeValue[iVertexTest] <= fTargetValue)
            iFlagIndex |= 1<<iVertexTest;
    }
    ...
    glNormal3f( ... );
    glVertex3f ( ... );
    ...
}
```

Parte del código correspondiente a la creación de la librería dinámica donde puede observarse la declaración de nuevos comandos para el intérprete *Tcl*.

```
...
EXTERN_C int DECLSPEC_EXPORT
Banmarchingdll_Init(Tcl_Interp* interp){
#define USE_TCL_STUBS
Tcl_InitStubs(interp, "8.5", 0);
#endif

if (version == NULL)
    return TCL_ERROR;
int r = Tcl_PkgProvide(interp, "Banmarchingdll",
                       Tcl_GetString(version));
Tcl_CreateCommand(interp, "cargarFicheroOrbital",
                  cargarFicheroOrbital, NULL, NULL);
Tcl_CreateCommand(interp, "pintarSuperficieIsovalor",
                  pintarSuperficieIsovalor, NULL, NULL);
```

...

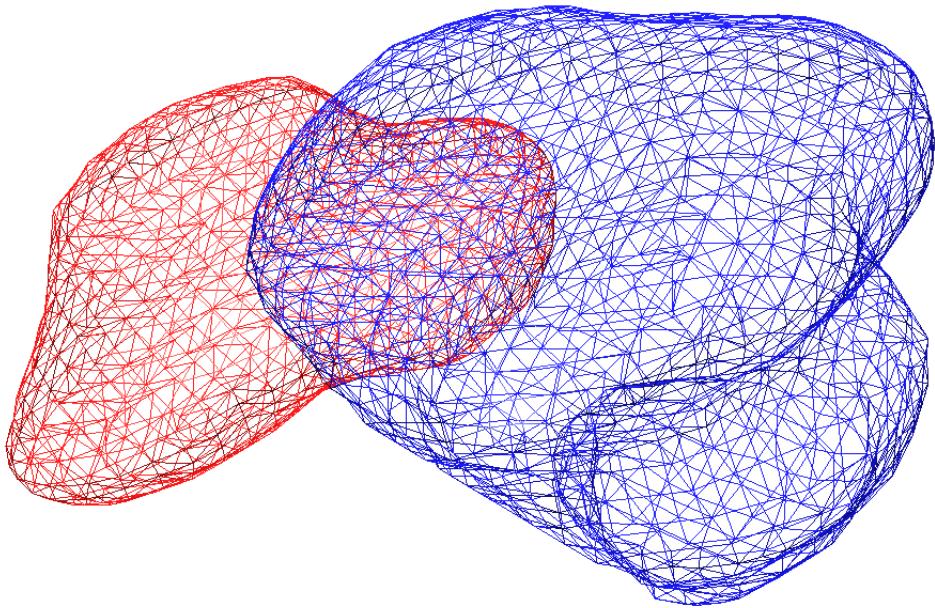


Figura 3.42: Mallas de triangulos generadas por *Marching Cubes*

3. Desde el intérprete *Tcl* se ejecutan los nuevos comandos aportados por la librería dinámica para cada isovalor, almacenandolos en *displays lists*. Posteriormente es posible elegir cuál de ellos mostrar. Se consiguen efectos visuales bastante atractivos al variarlos rápidamente desde la aplicación final *BrandyMol*.

```
#creo los display Lists
for {set x 0} {$x <= 255} {incr x} {
    #display
    set dl [glGenLists 1]
    lappend visor(displaysOrbitales) $dl
    glNewList $dl GL_COMPILE

    pintarSuperficieIsovalor $x

    glEndList
#end display
}
```

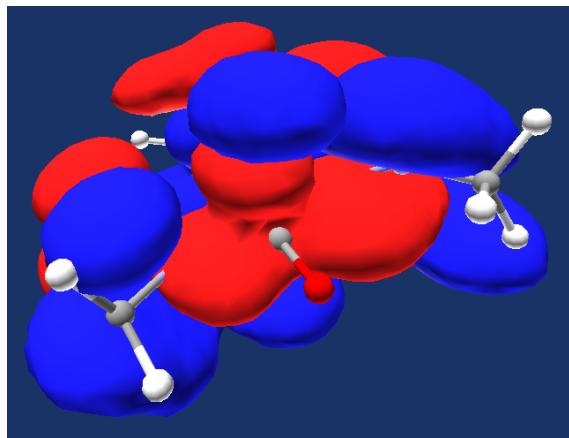


Figura 3.43: Orbital en modo superficie

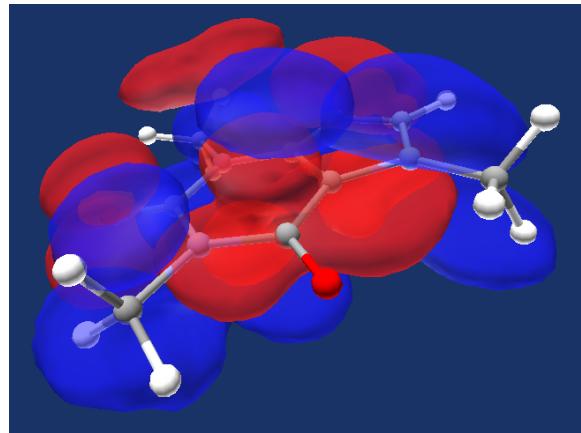


Figura 3.44: Orbital en modo superficie y transparencia

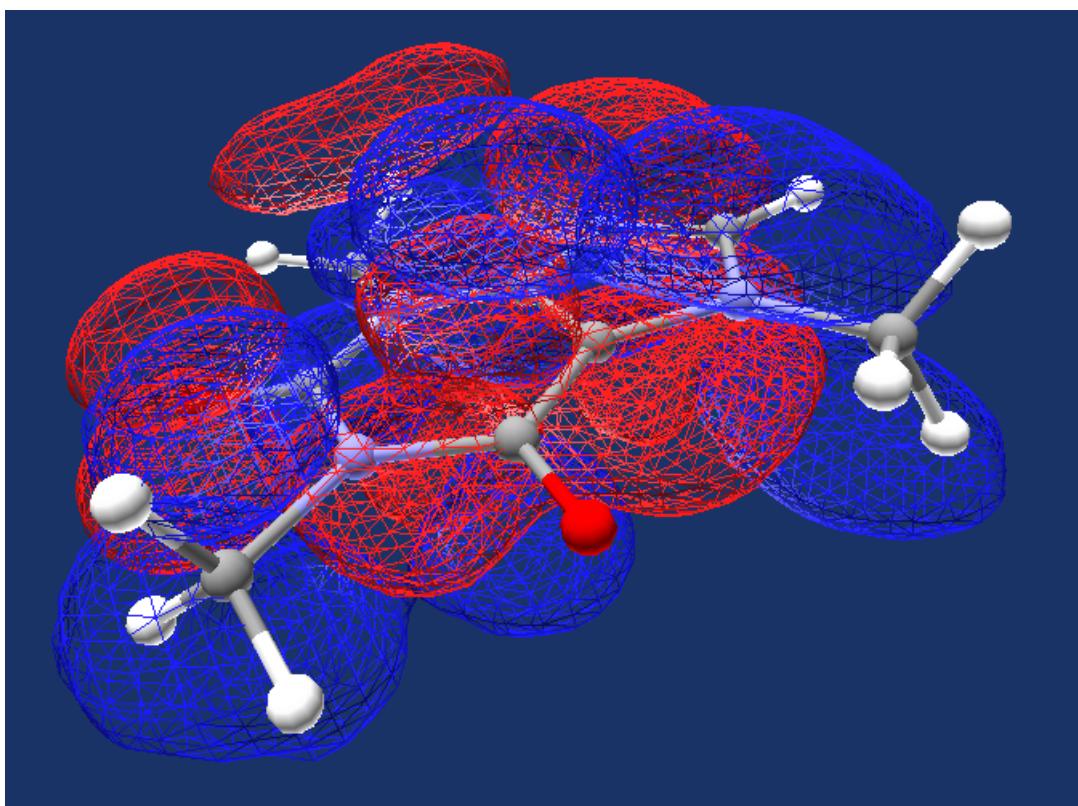


Figura 3.45: Orbital en modo malla alámbrica

3.6.11. Animación

Con objeto de los test de rendimiento preliminares se implementó un modo de animación del visor. Una vez activado, la cámara rota indefinidamente acimutalmente alrededor de la escena siempre que no haya acciones pendientes. Para ello hacemos uso del comando `after idle` de *Tcl* que ejecuta un comando siempre que el intérprete esté ocioso. La *H.U.D.* incorpora un contador de *FPS* siempre que se habilite el modo animación para tener estimaciones de rendimiento.

```
proc idle { base } {
    upvar #0 VisorGL::$base visor

    camaraAzimuth $base 3
    $visor(togl) postredisplay
    set visor(idleClockFrame1) $visor(idleClockFrame2)
    set visor(idleClockFrame2) [clock microseconds]
    set visor(idleId) [after idle "VisorGL::idle $base"]
}
```

Se implementa además una presentación donde el logotipo de *OpenGL* se recompone antes de dejarlo funcional al usuario. No persigue ningún fin en particular salvo sorprender a los usuarios.



Figura 3.46: Animación inicial

3.7. Integración

El visor se ha desarrollado como un componente independiente que puede integrarse en cualquier ventana *Tk*. Para su desarrollo se ha integrado en una ventana simple, y se han ejecutado manualmente instrucciones de su *API* para las pruebas: cargas de ficheros, de orbitales, tomas de medidas, etc... Una vez cerca de su versión final, es necesario integrarlo dentro de *BrandyMol*, dando como resultado *BrandyMol v1.5*. Dado que el visor *VTK* fue desarrollado como un componente *Macter* con una interfaz de comunicación claramente definida, la adaptación no resulta demasiado compleja. Es necesario sustituir las llamadas del tipo *VisorVTK::nombreProcedimiento* por *VisorGL::nombreProcedimiento*, y algunos ajustes adicionales que no merecen mayor atención. *BrandyMol* utiliza el visor para tareas de más alto nivel, recuperando información del visor y dándole un uso más amigable incluso. Para más detalles puede consultarse la documentación de *BrandyMol v1.0* [5]

Puede verse a continuación como se crea el visor y se agrega la aplicación:

```
#Se crea la instancia de VisorGL
VisorGL::newVisorGL v c

#Se define en componente de manejo de OpenGL
togl $frameright.visor
    -createproc "VisorGL::inicializarVisorGL $baseVisor"

#Se empaqueta como componente de Tk
pack $frameright.visor -expand 1 -fill both
```

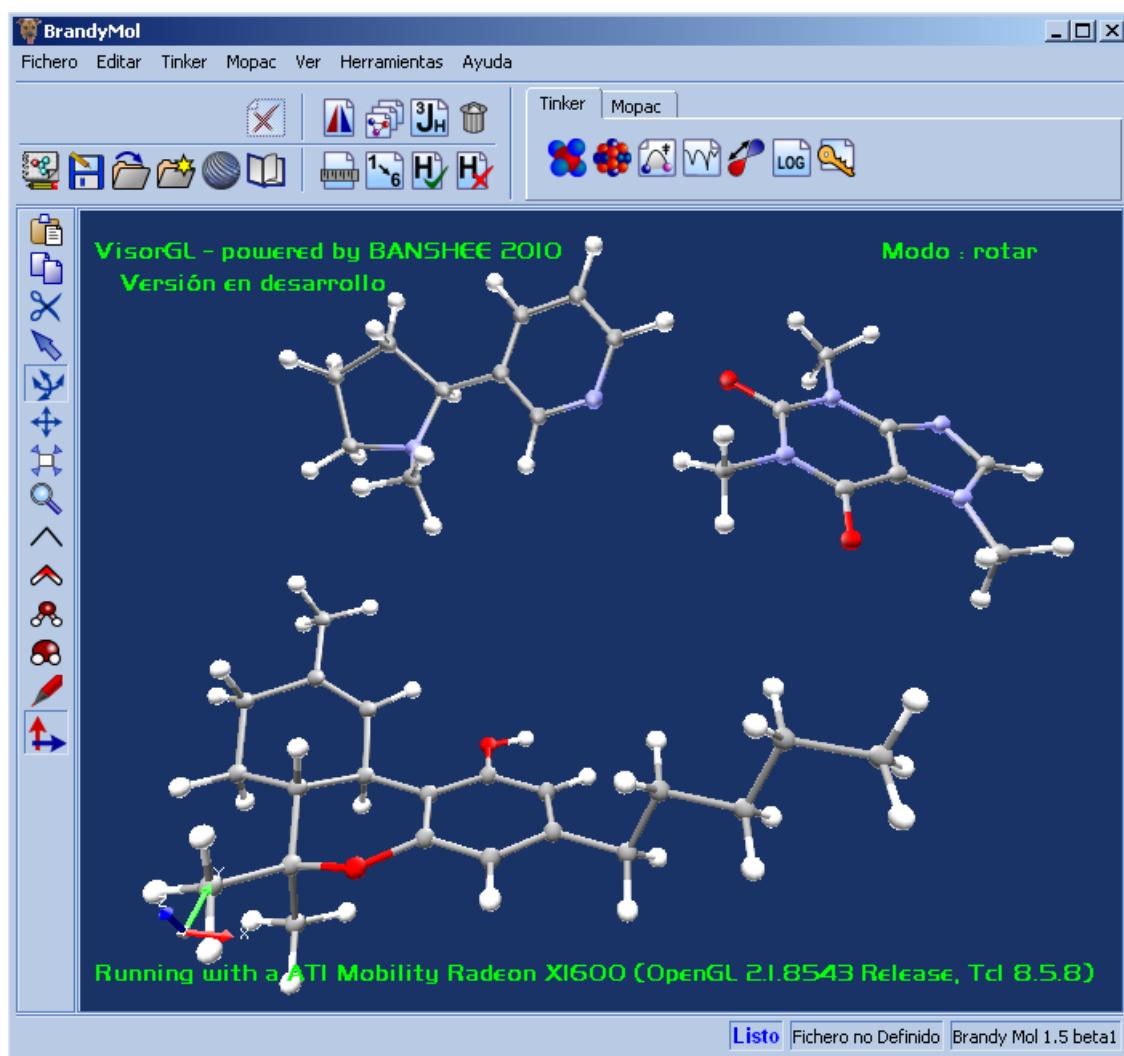


Figura 3.47: VisorGL integrado en BrandyMol

CAPÍTULO 4

Problemas, Conclusiones y Trabajos Futuros

4.1. Problemas

Los problemas surgidos a lo largo del proyecto no han sido muchos, pero si especialmente frustrantes.

El problema principal arrastra el desarrollo al completo. *Tcl/Tk* es un intérprete y no dispone de analizadores sintácticos gratuitos por lo que los errores más simples no se producen hasta que se ejecuta la correspondiente línea de código. Además es una aplicación *3D* interactiva donde en ocasiones el flujo de ejecución depende de posiciones concretas de moléculas y cámara que no siempre es fácil reproducir varias veces. Como ejemplo, se producía una división por cero solo cuando dos átomos de una molécula estaban perfectamente alineados a uno de los ejes de referencia, y la cámara se posicionaba perpendicularmente al plano. Una situación especial que es difícil reproducir mientras no se conozca el motivo. Por ello se ha tenido que dedicar especial atención y tiempo a pruebas.

Otro problema importante fue un error en la implementación de las librerías *Tcl3D*. Concretamente en la función `tcl3DRotate` que devuelve una matriz de rotación de cierto ángulo, algunos de los elementos de la matriz no correspondían con la definición formal. Lo complicado no fue solucionarlo, sino encontrar el foco del problema, ya que normalmente se suele achacar a un error propio antes que a unas librerías compiladas que usan cientos de personas. La realidad superó a la humildad. *VisorGL* implementa un modo de animación en el que la cámara gira acimutalmente de manera continua, para ello hace uso de la función `tcl3dRotatef` entre otras. A priori no se nota nada anómalo, pero tras un tiempo

(en mi caso el correspondiente a una merecida merienda), la cámara ha aumentado de manera considerable su radio al centro de la escena. No detallaré el número de horas dedicadas ni de pruebas diferentes hasta dar con el motivo, ni la frustración sentida al ver que algo conceptualmente correcto no funciona como debiera, pues cualquier informático podrá imaginarse. Lo que realmente me extrañó es como no se dio cuenta antes nadie de un error así en una función tan trivial en toda aplicación *3D*.

```

DESCRIPTION
glRotate produces a rotation of angle degrees around the
vector (x,y,z). The current matrix (see glMatrixMode) is
multiplied by a rotation matrix with the product replacing
the current matrix, as if glMultMatrix were called with the
following matrix as its argument:

$$\begin{pmatrix}
 xx(1-c)+c & xy(1-c)-zs & xz(1-c)+ys & 0 \\
 | & | & | & | \\
 | yx(1-c)+zs & yy(1-c)+c & yz(1-c)-xs & 0 \\
 | xz(1-c)-ys & yz(1-c)+xs & zz(1-c)+c & 0 \\
 | & | & | & | \\
 | 0 & 0 & 0 & 1
 \end{pmatrix}$$

Where c =  $\cos(\text{angle})$ , s =  $\sin(\text{angle})$ , and  $\|(x,y,z)\| = 1$ 
(if not, the GL will normalize this vector).

```

If the matrix mode is either **GL_MODELVIEW** or **GL_PROJECTION**,
all objects drawn after **glRotate** is called are rotated. Use
glPushMatrix and **glPopMatrix** to save and restore the
unrotated coordinate system.

Definición OpenGL

```


/*
 * Build a rotation matrix based on angle and axis.
 * Angle is given in degrees.
 */
void tcl3dMatfRotate (float angle, float *axis, float *m)
{
    float s, c;
    float ux, uy, uz;
    s = sin(DEGTORAD(angle));
    c = cos(DEGTORAD(angle));
    tcl3dVec3fNormalize (axis);
    ux = axis[0];
    uy = axis[1];
    uz = axis[2]; pow(ux, 2) !!!
    m[0] = c + (1-c) * ux;
    m[1] = (1-c) * ux*uy + s*uz;
    m[2] = (1-c) * ux*uz - s*uy;
    m[3] = 0;
    m[4] = (1-c) * uy*ux - s*uz;
    m[5] = c + (1-c) * pow(uy, 2);
    m[6] = (1-c) * uy*uz + s*ux;
    m[7] = 0;
    m[8] = (1-c) * uz*ux + s*uy;
    m[9] = (1-c) * uz*uz - s*ux;
    m[10] = c + (1-c) * pow(uz, 2);
    m[11] = 0;
    m[12] = 0;
    m[13] = 0;
    m[14] = 0;
    m[15] = 1;
}


```

Implementación tcl3D

Figura 4.1: Error de implementación en tcl3D

Por último ha resultado complicado el generar una aplicación eficiente utilizando un lenguaje (*Tcl*) que no está enfocado a ello. Su pobreza en estructuras de datos ha provocado que algunas partes críticas sean desarrolladas en *C++* y se hayan exportado como librerías dinámicas al intérprete.

4.2. Conclusiones

A nivel profesional se ha mejorado sustancialmente el rendimiento de *BrandyMol* que llevaba usándose tres años sin incidencias y que ha evolucionado gracias a los usuarios que le han dado más uso del que se pensó, siendo necesarias nuevas capacidades. Dejamos por tanto una aplicación que utiliza librerías más afines a su rendimiento y haciendo más sencillo su mantenimiento. Además se ha puesto a prueba el buen diseño empleado en el proyecto anterior ya que no ha sido ninguna complicación integrar el nuevo visor.

A nivel personal he podido vivir la evolución de mi *software*. Saber que realicé algo útil que la gente ha utilizado a diario y comprobar sus necesidades. Cómo muchas de

las decisiones de diseño que apliqué eran o no correctas, y profundizar en temas como la eficiencia, algo que muchas veces los ingenieros inexpertos no tenemos en cuenta, pero que el mundo real si demanda, y haber ampliado enormemente mi conocimiento de *OpenGL* lo cual sé que me servirá en un futuro cercano pues es el campo al que deseo dedicarme dentro del mundo de la informática.

Y por último agradecer a **Gregorio Torres** y **Francisco Nájera** el tiempo dedicado no solo a mí, sino al proyecto en general, pues ellos le han dado vida a diario en sus laboratorios y clases. Ojalá este *software* sea la herramienta clave para algún descubrimiento importante y forme a muchos investigadores a lo largo de los años.

Desde aquí solo puedo estar agradecido por haber sido parte en estos proyectos.

4.3. Trabajos Futuros

Dada la evolución de internet y de las aplicaciones en la nube, sin duda el transformar *BrandyMol* de una aplicación de escritorio a una aplicación *Web* sería una de las mejores opciones. Además están surgiendo envoltorios para *OpenGL* en *javascript* como *WebGL*[18], que dotan a aplicaciones *3D* dentro del navegador de todos los recursos de bajo nivel de las *GPU*. Se podría también integrar con el *Campus Andaluz Virtual* mucho mejor siendo parte de él y facilitando la labor docente de entrega de trabajos y prácticas.

Utilizar *clusters* de computación para permitir cálculos sobre grupos de moléculas que serían imposibles en computadores personales. Para ello sería necesario modificar todo el *software* de cálculo científico utilizado para que soporte las más modernas técnicas de computación paralela.

APÉNDICE A

Instalación

A.1. Creación del instalador

Para la distribución de *BrandyMol* era necesario que el proceso de instalación fuera algo sencillo y sin complicaciones, sin embargo en principio es necesario instalar un intérprete de *Tcl/Tk*, las librerías *tcl3D*, el programa *MDL Isis Draw* ...

También hay que destacar que el fichero *conf.ini* contiene los directorios de trabajo de la aplicación y que estos pueden diferir de un equipo a otro. Y muy importante es el caso del software *Tinker*, éste es de libre uso, pero no de libre distribución siendo necesario para cada usuario acceder a la dirección *FTP* correspondiente, descargarlos y situarlos en el directorio apropiado.

Era necesario automatizar todo esto, y para ello se utilizó el *software* libre *InnoSetup Compiler*[19], *ISTool*[20] y la librería *Download DLL*. Estas tres herramientas permiten crear asistentes de instalación personalizados y descargar archivos desde la red.

El instalador final copia todos los archivos y librerías necesarias dentro del directorio especificado por el usuario manteniendo la estructura jerárquica adecuada. Y genera el archivo *conf.ini* según el directorio de instalación., accede de forma automática a la dirección `ftp://dasher.wustl.edu/pub/tinker-bin/windows`, de donde descarga todos los ejecutables *Tinker* necesarios, los descomprime y copia en el lugar adecuado.

El instalador también contiene *MDL Isis Draw*, el plugin *CHIME*[21], las ayudas y un identificador de nombres moleculares. Además de diversas opciones de instalación al usuario, y como en casi todo instalador es posible generar iconos de acceso directo variados,

y un desinstalador.

En resumen, en un único archivo empaquetamos todo lo necesario, y tras realizar un pequeño asistente dejamos *BrandyMol* listo para ser usado.

Personalmente me han gustado estas herramientas, por la razón de que todo el instalador puede expresarse en forma de *script*. De hecho *InnoSetup* es el compilador, mientras que *ISTool* es un asistente gráfico que escribe parte del *script* por nosotros. Esto deja grandes posibilidades de ampliación.

Finalmente se han producido cuatro distribuciones diferentes del programa dependiendo del usuario al que vaya destinado.

- **NETFULL:** Es la distribución por defecto que se usará en la asignatura *Introducción a la Modelización Molecular*, contiene *BrandyMol*, los programas de dibujo y descarga *Tinker* de Internet.
- **NETLITE:** Igual a la anterior salvo porque no contiene los programas de dibujo. Indicada para los que ya los tengan o prefieran descargarlos por su cuenta. Es la distribución de menor tamaño.
- **OWNFULL:** Esta distribución lo contiene todo, *BrandyMol*, los programas de dibujo y los ejecutables *Tinker*. No es apropiada para distribución por los problemas ya comentados. Solo se utiliza para uso interno.
- **OWNLITE:** Igual a la anterior pero sin incluir los programas de dibujo. Utilizada principalmente en la etapa de depuración del instalador.

La Universidad de Málaga cuenta con los permisos de distribución de los programas de dibujo *MDL IsisDraw 2.5*, Ayudas de *IsisDraw*, *MDL CHIME* y el módulo de nomenclatura de *IsisDraw*.

A.2. Tutorial de la Instalación

A continuación mostramos paso a paso como instalar la distribución *OWNFULL*.

Apéndice A. Instalación



Figura A.1: Paso 1

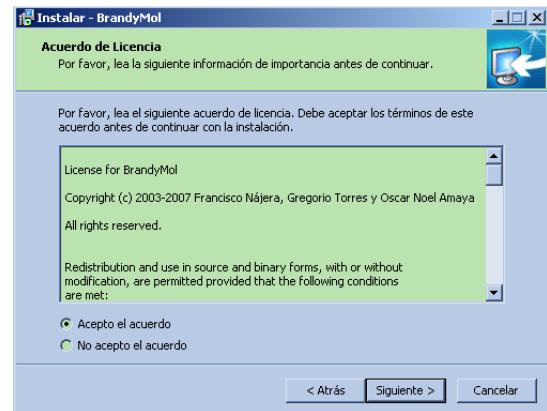


Figura A.2: Paso 2

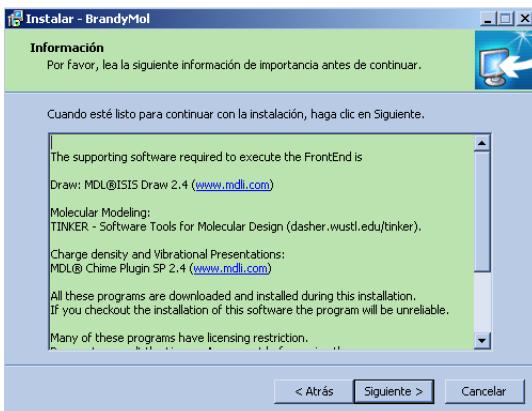


Figura A.3: Paso 3

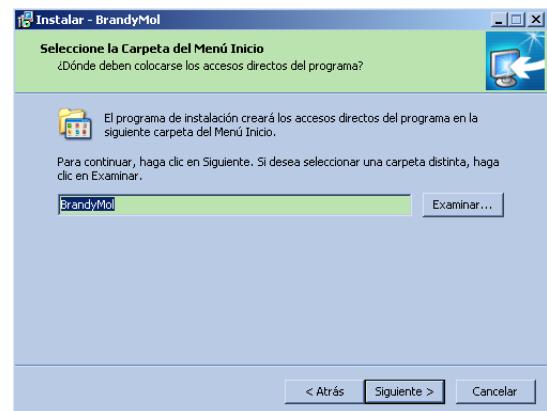


Figura A.4: Paso 4



Figura A.5: Paso 5



Figura A.6: Paso 6

Apéndice A. Instalación



Figura A.7: Paso 7



Figura A.8: Paso 8

Bibliografía

- [1] **OpenGL Programming Guide** Third edition, The Official Guide. Addison Wesley, 1999.
- [2] **OpenGL Reference Manual** Fourth edition, The Official Reference. Addison Wesley, 2004.
- [3] **Gráficos por Computadora con OpenGL**. Prentice Hall, 2006.
- [4] **Practical Programming in Tcl and Tk**, Third Edition. Prentice Hall, 1999.
- [5] **Entorno Gráfico Modular de Moléculas 3D en VTK**. Proyecto Fin de Carrera de Ingeniería Técnica en Informática de Sistemas, EI PF/1881. Oscar Noel Amaya García, 2007. http://jabega.uma.es/search*spi?/aAmaya+Garc{226}ia%2C+{226}Oscar+Noel/aamaya+garcia+oscar+noel/-3%2C-1%2C0%2CB/frameset&FF=aamaya+garcia+oscar+noel&1%2C1%2C#
- [6] **Química Orgánica, Estructura y Función** Tercera Edición. Omega, 2000.
- [7] **Win32 Programmer's Reference**, Microsoft Corporation, 1996.
- [8] **The VTK User's Guide** version 4.4, Kitware 2004.
- [9] **DirectX** Microsoft. <http://msdn.microsoft.com/en-us/directx/default.aspx>
- [10] **OpenGL** Microsoft. <http://www.opengl.org/>
- [11] **Visualization Toolkit**. <http://www.vtk.org/>
- [12] **Tcl3D** <http://www.tcl3d.org/>
- [13] **Formato MOL**. <http://www.symyx.com/downloads/public/ctfile/ctfile.pdf>
- [14] **MDL Isis/Draw**. www.mdli.com

Bibliografía

- [15] **Tinker** Software Tools for Molecular Design. <http://dasher.wustl.edu/tinker>
- [16] **Mopac** Molecular Orbital PAckage. <http://openmopac.net/>
- [17] **Algoritmo Marching Cubes**. http://en.wikipedia.org/wiki/Marching_cubes
- [18] **WebGL** OpenGL ES 2.0 for the Web. <http://www.khronos.org/webgl/>
- [19] **Inno Setup**. <http://www.jrsoftware.org/isinfo.php>
- [20] **ISTool**. <http://www.istool.org>
- [21] **Chime Plugin**. <http://www.mdli.com>