UNIVERSIDAD NACIONAL AUTÓNOMA
DE MÉXICO

FACULTAD DE CIENCIAS

GLNEBULA: UN SISTEMA PARA LA VISUALIZACIÓN DE NEBULOSAS PLANETARIAS

T E S I S

QUE PARA OBTENER EL TÍTULO DE:

LICENCIADO EN CIENCIAS DE LA COMPUTACIÓN

P R E S E N T A :

JUAN JOSÉ AJA FERNÁNDEZ

TUTOR: DR. CHRISTOPHE MORISSET

2007

FACULTAD DE CIENCIAS
UNAM

Universidad Nacional Autónoma de México
UNAM
Biblioteca Central
Dirección General de Bibliotecas de la UNAM

UNAM – Dirección General de Bibliotecas
Tesis Digitales
Restricciones de uso

DERECHOS RESERVADOS ©
PROHIBIDA SU REPRODUCCIÓN TOTAL O PARCIAL

Todo el material contenido en esta tesis esta protegido por la Ley Federal del Derecho de Autor (LFDA) de los Estados Unidos Mexicanos (México).

El uso de imágenes, fragmentos de videos, y demás material que sea objeto de protección de los derechos de autor, será exclusivamente para fines educativos e informativos y deberá citar la fuente donde la obtuvo mencionando el autor o autores. Cualquier uso distinto como el lucro, reproducción, edición o modificación, será perseguido y sancionado por el respectivo titular de los Derechos de Autor.

1. Datos del alumno
   Aja
   Fernández
   Juan José
   55 34 15 77
   Universidad Nacional Autónoma de México
   Facultad de Ciencias
   Ciencias de la Computación
   401109184

2. Datos del tutor
   Dr
   Christohpe
   Morisset

3. Datos del sinodal 1
   Dr
   Luis Miguel
   de la Cruz
   Salas

4. Datos del sinodal 2
   Dr
   Wolfgang
   Steffen

4. Datos del sinodal 3
   Dr
   Alejandro
   Aguilar
   Sierra

4. Datos del sinodal 4
   Dr

2

Alfredo Javier
Santillán
González

5. Datos del trabajo escrito
GLNebula: Un sistema para la visualización de nebulosas planetarias
104 p
2007

Dedicatoria y Agradecimientos

Dedico el presente trabajo enteramente a mis padres, que con silenciosa paciencia observaron el lento desarrollo del mismo.

Agradezco tremendamente el apoyo y paciencia de Christophe y de Luis, así mismo un agradecimiento muy especial a Daniel por sus ideas que son pilares de la construcción de GLNebula. A Tonatiuh, Pamela y el resto de los que directa o indirectamente influenciaron este trabajo y me apoyaron.

Índice general

Capítulo 1

Introducción

Las herramientas computacionales para asistir a la investigación científica han probado su utilidad desde hace algunos años, sirviendo en un principio como apoyo y recientemente, en muchas áreas, demostrando ser indispensables para el desarrollo del conocimiento científico. La adquisición masiva de datos y la creación de modelos computarizados de fenómenos naturales son sumamente importantes para el estudio científico, pero considerando la cantidad de información involucrada y que no tiene un significado real hasta ser interpretada, es necesario contar con herramientas que permitan facilitar la extracción de datos útiles, o bien que permitan la reconstrucción visual del modelo, para tener una idea más clara del mismo y así tal vez deducir información que de otra manera hubiera sido muy complicado si sólo se tuvieran los datos. Así se decidió crear una herramienta: GLNebula, que asistiera a los astrónomos en la visualización de nebulosas planetarias y en general cualquier región ionizada[1], a partir de datos obtenidos de modelos computacionales.

Actualmente no hay una manera de visualizar modelos de nebulosas planetarias en tiempo real que permita la observación desde cualquier punto de vista del objeto, debido a la gran cantidad de datos y en parte por el desconocimiento que muchos investigadores especializados poseen de las herramientas computacionales.

[1]Regiones gaseosas en donde los átomos se desprenden de algunos de sus electrones

Así se pensó en GLNebula para que permitiera cumplir con dos objetivos:

- **Investigación:** Debido a la lejanía de los objetos estudiados y a los métodos empleados para estos fines, los astrónomos están acostumbrados a tener una única imagen del cuerpo en observación. Hay sistemas que permiten conocer otros aspectos de la morfología nebular, pero normalmente requieren realizar un procesamiento de los datos previo al despliegue de las imágenes, lo que se traduce en un consumo de tiempo considerable y evita la interactividad. Por ello sería útil tener un sistema que permitiera la visualización en tiempo real de objetos astronómicos a partir de datos obtenidos de observaciones o simulaciones, de manera que sea posible tener una proyecctión desde cualquier posición que el observador desee y así despejar dudas sobre su morfología, que como se mencionará más adelante, suelen presentarse a la hora de tratar con una única vista de la nebulosa. Con esto en mente, GLNebula se diseñó para asistir al investigador en la creación de una imagen mental más precisa del objeto modelado.

- **Divulgación:** Las nebulosas están entre los cuerpos astronómicos más coloridos, comúnmente presentado imágenes, que aunque espectaculares[2], no permiten la inmersión. Por otro lado es posible crear animaciones sumamente complejas de objetos astronómicos empleando herramientas como Autodesk-Maya, Blender y 3D Studio, pero de nuevo es imposible interactuar con los resultados. Es por ello que GLNebula fue diseñado para permitir un nivel de interactividad mayor, haciendo posible que el usuario controle la cámara a placer y manipule otros parámetros como la transparencia de la nube o su brillo. La idea final que rodea a la meta de divulgación es presentar modelos de alta resolución en el observatorio de visualización Ixtli[3] al público interesado y en un futuro extender al sistema para implementar estereoscopía activa y hacer la experiencia aún más envolvente.

A continuación se pretende proveer un panorama general de los objetos a visuali-

[2] $http://nssdc.gsfc.nasa.gov/photo\_gallery/photogallery-astro-nebula.html$
[3] $http://www.ixtli.unam.mx/$

zar por GLNebula, se hablará sobre sus características principales y su clasificación. Posteriormente se tratará el tema de los modelos empleados como datos de entrada, siguiendo con un capítulo dedicado a la visualización científica para luego pasar a los aspectos técnicos del desarrollo del sistema, lo cual es lo más importante del trabajo realizado y son temas de mayor dominio por parte del autor.

1.1. Nebulosas

Una nebulosa es un cuerpo astronómico formado por nubes de gas y polvo estelar. Está compuesta principalmente por Hidrógeno ($\sim 90\,\%$) y Helio ($\sim 10\,\%$), los cuales se encuentran comúnmente ionizados debido a la cercanía de fuentes de fotones energéticos y en menor medida a las altas temperaturas en su interior[4] .

El término proviene del latín *nebula* (literalmente nube), e históricamente fue aplicado a cualquier objeto astronómico difuso o menos definido que un planeta o estrella pero igualmente visible con un telescopio. Actualmente el significado es un poco más especializado, refiriéndose a un cuerpo en particular y no a cualquier objeto o "manchón" estelar, como podría ser una fuente brillante que se presentara borrosa debido a condiciones atmosféricas.

La importancia de las nebulosas dentro de la astronomía reside principalmente en que se presentan en etapas clave de la evolución de las estrellas: por ejemplo el caso de las nebulosas de emisión que son un semillero de nuevas estrellas, o bien las Novas, Supernovas y Nebulosas Planetarias que aparecen al final del ciclo vital estelar.

Dentro de una nebulosa se dan las condiciones adecuadas para la formación de nuevas estrellas: debido a las altas temperaturas y a la acumulación de energía el material nebular comienza a unirse en formaciones cada vez mayores, si su gravedad es suficiente procederá a condensarse en una o más estrellas. La contracción de estrellas en proceso de formación eleva considerablemente la temperatura interna del sistema, hasta el punto

[4]Secciones 2.3 y 2.3.1

de desencadenar la fusión del hidrógeno que las compone.

Al llegar a este punto la gravedad no es lo suficientemente fuerte para continuar con la contracción, debido a que la fusión interna estabiliza el cuerpo y no permite que la disminución de tamaño continúe (hacia el final de su vida el proceso se invierte: la gravedad comienza a ganar terreno a la fusión, desencadenando una implosión y dependiendo de la masa de la estrella se definirá su destino como una estrella moribunda, de neutrones o un hoyo negro), así que se observa un crecimiento paulatino de la estrella.

La relación entre estrellas y nebulosas es muy estrecha ya que el origen de las mismas está determinado en parte por la muerte de las estrellas, este asunto se tratará en la sección referente a las nebulosas planetarias.

1.1.1. Clasificación general de nebulosas

Aunque la meta del sistema es permitir la visualización de nebulosas planetarias, no está de más proveer una clasificación de las nebulosas en general, de cualquier manera es trivial extender GLNebula para permitir la visualización de otro tipo de nebulosas o inclusive cualquier tipo de cuerpo astronómico, ya que todo depende de los datos de entrada (siempre y cuando éstos cumplan con tener un escalar por celda y se presenten en forma de mallas cúbicas). El criterio empleado para esta clasificación en particular se basa principalmente en el tipo de iluminación que las compone, además toma en cuenta su evolución y su tipo de progenitora.

Tipos de nebulosas:

- Nebulosa de Emisión: Es una nube de gas ionizado que emite luz en varias longitudes de onda, la ionización procede de fotones de alta energía despedidos por estrellas cercanas o en su interior sumamente calientes. Junto con los residuos de supernova suelen ser las más coloridas entre los tipos de nebulosas. La mayor parte de su iluminación procede del gas ionizado en su interior, aunque también

presentan reflexión de la luz emitida por estrellas que se están formando dentro de ella. El color (y espectro de emisión) depende de la composición química y la cantidad de elementos ionizados. Debido a que el Hidrógeno es el elemento prevaleciente en el gas interestelar ya que requiere relativamente poca energía para su ionización (ver la sección 2.3), en la mayoría de las nebulosas de emisión el color rojo es dominante, aunque también presentan líneas de Oxígeno y Nitrógeno.
De entre este tipo se destacan: M42[5] (*nebulosa de Orión*), NGC2261[6] (*nebulosa de Hubble*) y M20 (conocida como *nebulosa trífida*), la cual cae también dentro de la categoría de nebulosa de reflexión.

- Nebulosa de Reflexión: También llamadas difusas. Las nebulosas de reflexión son grandes nubes de polvo que reflejan la luz emitida por estrellas cercanas. La diferencia con las nebulosas de emisión es que en este caso las estrellas cercanas no despiden la misma cantidad de energía (calor), así los elementos en el gas no se ionizan, pero las partículas de polvo están lo suficientemente esparcidas como para permitir el paso de luz y reflejarla. El polvo tiende a dispersar con mucha facilidad la luz azul, lo cual provoca que la mayoría de las nebulosas de reflexión tengan una coloración añil.
  Se destacan IC 2118 [7], conocida como la *nebulosa cabeza de bruja*, NGC 1432 que es parte del cúmulo de las Pléyades y la anteriormente mencionada M20.

- Nebulosas Oscuras: A veces llamadas de absorción, son el opuesto a las nebulosas de emisión. En este caso las partículas de polvo dentro de la nube absorben casi completamente la luz emitida por las estrellas cercanas, hecho que las hace ver como manchones sin luz contrastando con el entorno cuando está iluminado. Dentro de ellas está contenido mucho del medio interestelar, a veces siendo un millón de veces más masivas que nuestro sol, suelen tener 150 años luz de largo

[5]La M se refiere al número de Messier, astrónomo francés que en 1774 publicó un catálogo de 45 objetos astronómicos a manera de guía para los observadores

[6]Del término *New General Catalogue*

[7]Acrónimo de *Index Catalog*

y una temperatura sumamente baja de entre 7 y 15 grados Kelvin. Debido a que no emiten luz su observación se complica, por lo cual se recurre a técnicas de detección de radiación (microondas emitidas por sus partículas).
Entre las más conocidas están: IC 434 (*Cabeza de Caballo*), *Saco de Carbón* (sin número)

- Nebulosas Planetarias: El término está en cierto modo mal aplicado: los telescopios que los astrónomos empleaban anteriormente no eran lo suficientemente poderosos como para distinguir claramente un objeto lejano, por lo que algunos cuerpos eran vistos como gigantes gaseosos (como Júpiter, por ejemplo). Por ello al ver una nebulosa relativamente grande (y lejana), se podía catalogar como un planeta gaseoso, así el término fue heredado a lo que hoy se denomina nebulosa planetaria.
  Casi todas las nebulosas planetarias poseen una característica esfera gaseosa que las envuelve, esta envoltura no es más que restos de lo que fue una estrella: al llegar a cierta etapa de su vida, una estrella comienza a agotar el combustible de su núcleo, provocando que la región de Helio en su interior se contraiga debido a la gravedad, incitándolo a que comience a fusionarse y a expandirse alrededor del antiguo (y ya quemado) núcleo. Esta expansión provoca el enfriamiento en la superficie, y en este momento es cuando la estrella se cataloga como gigante roja. A partir de este punto comienza el proceso de creación de la nebulosa planetaria. Cuando casi toda la envoltura de Hidrógeno se ha agotado, el núcleo queda expuesto, y gases sumamente calientes son expedidos de él a distintas velocidades e intervalos. Mientras tanto la estrella continúa su evolución, procede a transformarse en una enana blanca (un cuerpo extremadamente caliente de tamaño pequeño) la cual emite grandes cantidades de fotones altamente energéticos, lo que ioniza la nube de gas que la rodea y provoca que brille. Entre los ejemplos más famosos destacan: NGC 6543 (*Ojo de Gato*), IC 4406 (*Retina*) y NGC 6751 (*Diente de león*).

- Residuos de Nova y Supernova: De origen similar a las nebulosas planetarias, están igualmente formadas a partir de desechos de estrellas, pero en este caso la eyección de material es considerablemente más violenta, producto de la transformación de una o dos estrellas en nova y supernova.

  Hacia el final de la vida de una estrella masiva, cuando la fusión comienza a perder terreno ante la atracción gravitatoria, la estrella colapsa. En este momento se producen ondas de choque de inmensa fuerza que proyectan las capas hacia el espacio, dejando expuesto el núcleo. Al material expulsado se le conoce como el residuo de una supernova, y normalmente se encuentra ionizado debido a la onda de choque, lo que destaca a estos cuerpos en el cielo y convierte a sus miembros en algunas de las nebulosas más vistosas. Existen dos tipos de supernova: el tipo I corresponde a dos estrellas y presenta patrones destructivos para ambas, el tipo II es sólo una estrella con masa mayor a 30 solares y también termina con la destrucción de la estrella. A su vez las novas pueden presentarse en parejas: una estrella "normal" acompañada de una enana blanca, la cual gradualmente consume el material de la primera, este tipo es conocido como nova recurrente.

  Entre las más destacadas se encuentran: M1 (*nebulosa del cangrejo*) y NGC6960 (*nebulosa del velo*).

ID	**Nombre**	**Tipo**[a]	**Distancia**[b]	**A : D**[c]
M20	Nebulosa Trífida	E/R	1.59	18h 2.6m : -23°2’
M42	Nebulosa de Orión	E	0.459	5h 35.4m : -5°27’
NGC2261	Nebulosa de Hubble	E	0.919	6h 39.2m : 8°44’
IC2118	Cabeza de Bruja	R	0.306	5h 6.9m : -7°13’
NGC1432,35	Pléyades	R	0.122	3h 47m : 24°67’
IC434	Cabeza de Caballo	O	0.49	5h 40.9m : -2°28’
S/N	Saco de Carbón	O	0.613	19h 5.9m : 6°00’
IC4406	Retina	P	0.582	14h 22.4m : -44°09’
NGC6543	Ojo de Gato	P	0.919	17h 58.6m : 66°38’
NGC6751	Diente de león	P	1.992	17h 58.6m : 66°38’
GQ Mus 83	Nova Muscae	N	4	11h 52m : -67°12’
M1	Nebulosa del Cangrejo	S	1.931	5h 34.4m : 22°01’
NGC6960	Nebulosa del Velo	S	0.613	20h 45.6m : 30º 42’

[a]E = Emisión, O = Oscura, R = Reflexión, P = Planetaria, N = Residuo de Nova S= Supernova
[b]kpc - Kiloparsecs
[c]Ascensión derecha : Declinación

Cuadro 1.1: Algunos ejemplos de nebulosas

Capítulo 2

Nebulosas Planetarias

Una nebulosa planetaria es producto de la fase final del ciclo de vida de una estrella de masa mediana (similar a nuestro sol y hasta 8 veces su masa), debido a que la cantidad de la misma es adecuada para los eventos que se presentan posteriormente, cuando la gravedad y las reacciones nucleares entran en conflicto para determinar qué forma tomará la estrella en su etapa final. Cuando entra en esta etapa de su vida (llamada AGB, *Asymptotic Giant Branch*), la estrella comienza a perder masa rápidamente: el combustible en su núcleo ha sido fusionado (el Hidrógeno se fusiona en Helio, que hace lo propio en Carbono), poniendo un alto a las reacciones nucleares en este nivel, mientras que continúan en una capa exterior con el consumo del Helio, siendo la capa producto de la condensación de la masa expedida en vientos salientes de la estrella.

La pérdida de masa en forma de vientos estelares es tan acelerada, que la envoltura de Hidrógeno es expulsada en unos 10^5 años[6]. En este punto el núcleo estelar está casi completamente expuesto y la estrella entra a la etapa conocida como PPN (del inglés *Proto-Planetary Nebula*). Así mientras que la capa exterior está en constante expansión, el núcleo permanece en forma de una estrella central extremadamente caliente, que continuamente expulsa radiación altamente energética (aquí aparecen las líneas de emisión, que se explicarán más adelante).

Estos pulsos de radiación ionizan el gas en la capa exterior de la nebulosa, haciendo

que brille y acelerando aún más su expansión. Las pulsaciones continúan durante un periodo de unos 10^6 años [6]. A esta cáscara junto con su borde brillante es a lo que para fines prácticos comúnmente se le considera como la nebulosa planetaria, aunque hay características presentes en la mayoría de las observaciones que sugieren un modelo "promedio"de estos cuerpos. Rózyczka[3] propone considerar principalmente, además de cáscara y borde:

- **Halo:** Frecuentemente visto en forma de aros concéntricos rodeando a la nebulosa. Están compuestos del gas y polvo despedido de la estrella central durante su etapa final. Aparentemente una parte de la materia dentro del halo es arrastrada por ondas de choque producidas por la interacción de vientos y elementos ionizados, la cual se cree pasa a formar la cáscara.

- **Anillo:** Presente en el borde interior de la cáscara, de un brillo característico que lo hace destacar de su entorno. Se considera que denota la presencia de una onda de choque sumamente fuerte.

- **Cavidad interior o burbuja:** Este elemento es una región compuesta por una onda de choque de alta temperatura y velocidad, comúnmente emitiendo rayos X (recientemente observados con los satélites Chandra y XMM).

Estas características junto con otros rasgos comunes, aunque no presentes en todas las nebulosas, como: lóbulos bipolares (estructuras de reloj de arena), chorros salientes, nudos, flujos de Hubble, extrusiones y ondas (patrones presentes dentro de la burbuja); podrían ser consideradas como genéricas.

Conforme la nebulosa evoluciona en su vida llega un punto en que la estrella central ya no es capaz de emitir pulsos energéticos debido a la pérdida de calor, en esta etapa el brillo de la nebulosa decae considerablemente ya que el gas deja de ser ionizado por la enana blanca en su interior, dificultando su observación. La ionización se detiene, pero el gas nebular sigue expandiéndose debido a la energía cinética de expulsión experimentada

antes de la etapa final, por ello eventualmente la nebulosa dispersará toda su materia en el medio interestelar, enriqueciéndolo con Carbono, Oxígeno y otros elementos. El enfriamiento de la estrella central continúa hasta que toda la energía térmica ha sido agotada, en este punto la estrella entra en la última etapa de su vida como una "enana oscura".

2.1. Clasificación de Balick

No hay un consenso general con respecto a la clasificación de las nebulosas planetarias, pero diversos enfoques se han empleado con distintos propósitos (clasificarlas por su morfología, su composición, etc.).

El esquema *Vorontsov-Velyaminov* propuesto en 1934 es uno de los más empleados para clasificar nebulosas planetarias tomando en cuenta su morfología. Está compuesto por seis categorías generales, dos de las cuales poseen subcategorías:

- **I - Imagen estelar**
- **II - Disco suave**
  - **IIa - Más brillante hacia el centro**
  - **IIb - Brillo uniforme**
  - **IIc - Vestigios de estructura anillada**
- **III - Disco irregular**
  - **IIIa - Con distribución irregular del brillo**
  - **IIIb - Con vestigios de estructura anillada**
- **IV - Estructura anillada**
- **V - Forma irregular, similar a nebulosa difusa**
- **VI - Forma anómala**

El sistema no prohíbe que un cuerpo caiga dentro de múltiples clases, de tal forma que morfologías relativamente complejas pueden ser clasificadas por la combinación de

categorías, así se pueden tener tipos compuestos como IV+IIa. Las clases son autoexplicativas y, aunque es un esquema útil, es posible que sea obsoleto debido a que no toma en cuenta características morfológicas como la bipolaridad. Uno de los primeros modelos visualizados con GLNebula corresponde a una nebulosa con estas características, por lo cual aunque el esquema es lo suficientemente importante como para ameritar una breve descripción, tal vez no sea el más adecuado para profundizar.

Probablemente el esquema de clasificación más apropiado para el presente trabajo sea el propuesto por Balick[1], informal pero útil, el cual también se concentra en la morfología de las nebulosas planetarias. Propone que su forma puede ser reducida a tres tipos básicos, llamados clases: esféricas (*Round*), elípticas (*Elliptical*) y bipolares/en forma de mariposa (*Bipolar, Butterfly*) y una menor de tipo irregular/peculiar (*Peculiar*). Además emplea una noción de periodo: temprano (*Early*), medio (*Middle*) y tardío (*Late*) relacionada con la proximidad de anillos interiores brillantes al núcleo de la nebulosa y que provee una idea sobre la etapa en la que se encuentra. Las transiciones entre clases no están prohibidas y las de periodo se dan por hecho, de manera que una nebulosa planetaria puede pasar por más de un tipo en su evolución (aunque la escala de tiempo no permite observar el movimiento de clases, es posible inferir a qué tipo perteneció el cuerpo anteriormente).

- **Esféricas:** En esta clase el envoltorio gaseoso que la nebulosa heredó de su etapa como gigante roja (llamado por sus siglas en inglés RGE, *Red Giant Envelope*) presenta una forma circular, debido a que fue expulsado y acelerado por vientos estelares del núcleo en forma más o menos uniforme (isotrópico), formando una burbuja. En su etapa temprana suelen presentar halos internos y comúnmente una distribución uniforme de densidades a lo largo de anillos dentro de la burbuja. Cuando entran en la etapa tardía la onda de choque inicial llega al borde externo del RGE, perforándolo y dejando escapar el gas que estaba dentro de la burbuja, esparciendo su contenido en el medio interestelar.

- **Elípticas:** Presentan características similares a las esféricas, sólo que poseen dos ejes: mayor y menor mucho más definidos. En el periodo temprano el envoltorio es despedido con diferentes velocidades hacia los polos y el ecuador, de manera que adquiere una forma ovalada y, si la densidad es mayor en el ecuador (como comúnmente sucede), presentan discos ecuatoriales característicos. Se cree que la diferencia de velocidades presentes en la expulsión del envoltorio es producto del momento angular adquirido gracias a la rotación de la estrella central. Si la diferencia de densidades entre los polos y el ecuador es suficientemente grande es posible que tome forma de mariposa, cayendo en la categoría siguiente. Su etapa tardía es similar al caso esférico, solamente que la perforación del RGE se presenta primero en el eje polar, y así la salida del gas aparece como dos lóbulos, cada uno en forma de burbuja a lo largo de este eje.

- **Bipolares/Butterfly:** En su etapa juvenil el envoltorio suele presentarse en forma de un disco delgado de alta densidad, que debido a la interacción con los vientos estelares sufre una reducción gradual de masa, perdiéndola en la dirección de salida de los vientos, finalmente tomando una forma que se considera similar a la de una mariposa. Conforme la nebulosa evoluciona hacia su etapa tardía los vientos continúan con la erosión al interior del disco, formando un agujero central que crece poco a poco. Esta expansión desde el centro del disco produce un abultamiento en los lóbulos que previamente eran delgados, acentuando las características de mariposa que anteriormente eran más sutiles. En la etapa media de su vida suelen presentar halos exteriores bipolares y muchas veces, debido a la forma de estos últimos y a la estructura cercana al núcleo, se puede inferir qué tipo de nebulosa era antes de tomar forma de mariposa.

- **Peculiares/Irregulares:** En esta categoría caen aquellas nebulosas que no pueden ser clasificadas debido a que sus rasgos son demasiado sutiles o bien no presentan características distintivas del todo, así que normalmente cuando una nebulosa es categorizada como peculiar o irregular suele ser acompañada de una descrip-

ción compleja que combina lóbulos bipolares, halos irregulares, RGEs amorfos y simetrías complejas (de punto) o inexistentes.

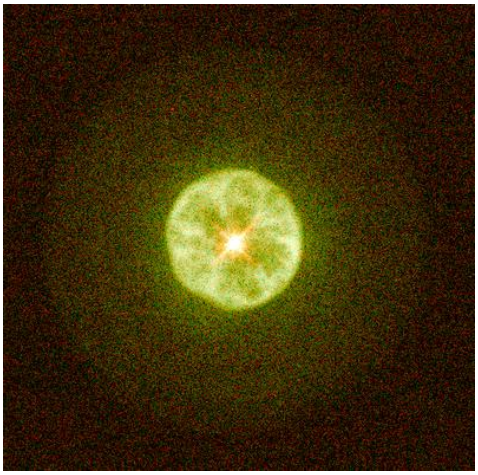

Figura 2.1: IC3568: Esférica en etapa temprana - *Howard Bond (ST Scl), NASA*

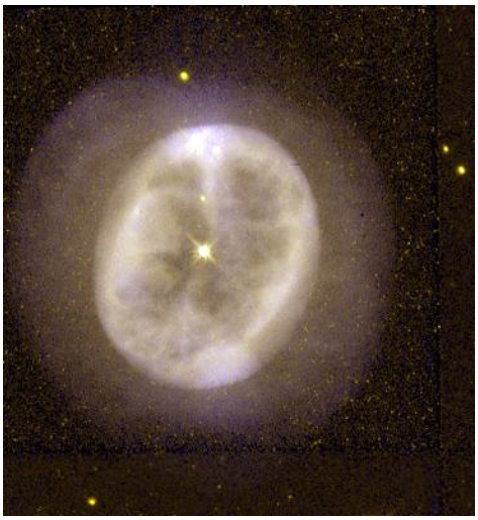

Figura 2.2: NGC2022: Elíptica en etapa media - *Howard Bond (ST Scl), NASA*

Es claro que la clasificación apela al juicio del observador, por lo que podrían presentarse infinidad de ambigüedades al momento de reportar nuevos hallazgos de nebulosas o tratar de clasificar objetos ya conocidos, es por ello que rara vez es un tema central de estudio y solamente se emplea para dar una idea de la estructura morfológica de un

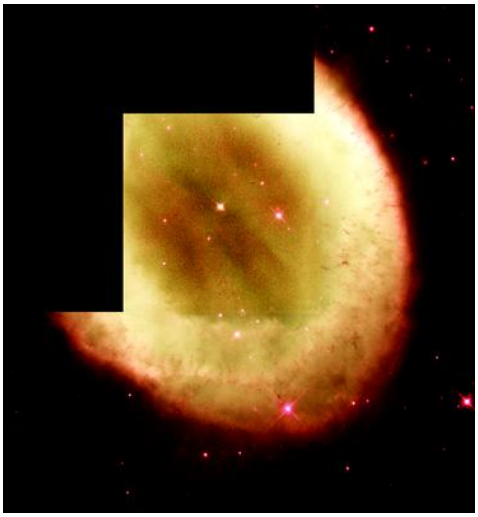

Figura 2.3: NGC6720: Elíptica en etapa media con halo esférico - *Howard Bond (ST Scl), NASA*

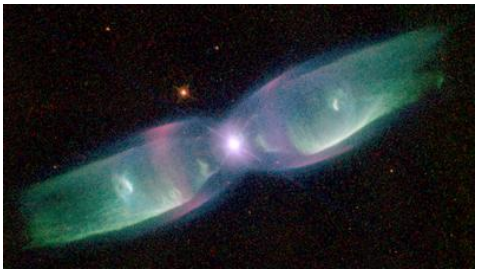

Figura 2.4: M2-9: Forma de mariposa en etapa media - *Bruce Balick (University of Washington), Vincent Icke (Univ. Leiden), Garrelt Melema (Univ. Estocolmo), NASA*

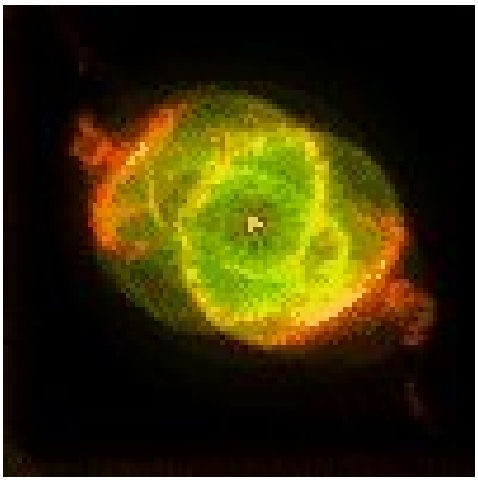

Figura 2.5: NGC6543: Peculiar con halo esférico - *Bruce Balick (University of Washington), NASA*

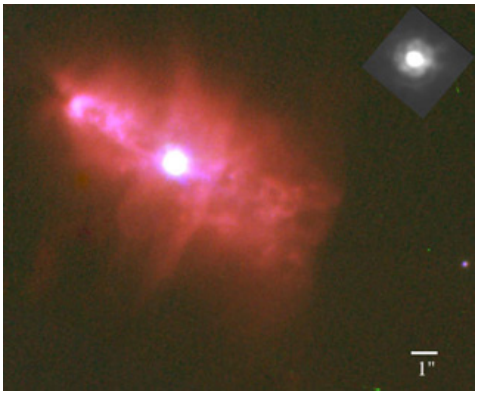

Figura 2.6: IC2149: Peculiar sin rasgos predominantes - *Patrick A. Young, Donald W. McCarthy, Craig Kulesa, Karen A. Knierman, Jacqueline Monkiewicz,(Steward Obs.), Guido Brusa, Douglas Miller, Matthew Kenworthy (Center For Astronomical Adaptative Optics)*

cuerpo dado. Si por ejemplo se emplea un filtro IR para una primera observación se podrían presentar discrepancias de la morfología al momento de cambiar de observatorio para detectar rayos X, ya que no figurarán las mismas partes del gas en ambos casos. Además, la forma de las nebulosas suele cambiar (aunque lentamente) dependiendo del estado actual de la misma, por lo que Meixner[2] propone añadir a esta clasificación un criterio que permita considerar modelos cinemáticos, para así tener en cuenta la posibilidad de cambio en los cuerpos.

2.2. Debate binaridad/campo magnético

La morfología de las nebulosas presenta patrones que corresponden directamente con el movimiento de los vientos en el interior de las mismas, conociendo estos patrones se podría tener una idea precisa de los cambios que sufre una estrella desde su etapa como gigante roja hasta convertirse en una nebulosa planetaria. Desafortunadamente su estructura es lo suficientemente compleja para que aún no se tenga un marco teórico completo de las interacciones que ocurren en el interior, de manera que no se conoce con certeza el proceso que lleva a tomar a la nebulosa su morfología final, aunque hay varios puntos de vista que permiten enfocar el estudio para tomar en cuenta diversos aspectos y así tratar de explicar parte por parte.

Si se toma un enfoque hidrodinámico, entonces es razonable decir que la nebulosa planetaria es producto de las interacciones de vientos estelares con distintas densidades, velocidades y temperaturas. Este esquema ha probado ser adecuado para explicar la forma general de la capa exterior de material estelar en el caso de morfologías esféricas, pero no es suficiente para explicar el origen de características más complejas como pueden ser simetrías de punto, bipolaridad o chorros salientes.

Así, a grandes rasgos, es posible reducir a tres el número de fuerzas que influyen en la formación de una nebulosa planetaria: la gravedad, electromagnetismo y fricción con

el medio interestelar u otros cuerpos. En recientes observaciones de nebulosas planetarias (particularmente las que provee el Telescopio Espacial Hubble) se ha encontrado una presencia magnética importante y ha llevado a pensar a muchos ([7], [8]) que la existencia de un campo magnético puede ser un factor determinante de la morfología de la nebulosa en formación. Con esto en mente se ha extendido el modelo hidrodinámico para tomar en cuenta interacciones con un campo magnético, llamado MHD (Magnetic Hydro-Dynamics). Se ha propuesto que el campo magnético puede ser responsable del direccionamiento del flujo de material estelar en la fase PPN, creando muchos de los chorros salientes característicos de algunas nebulosas y propiciando la pérdida acelerada de masa en regiones donde se presentan filamentos magnéticos. El esquema ha probado ser útil para ciertos casos en los que aplicando un modelo MHD se obtienen morfologías similares a las observadas, pero de nuevo no es suficiente para explicar todas las formas.

Si ahora se recurre a la gravedad y fricción es posible que la interacción de la masa estelar en expansión y el medio interestelar o un objeto cercano en forma de compañero, sea un factor influyente en la morfología final de la nebulosa. Al momento de la eyección de materia en la fase de gigante roja puede entrar en juego la gravedad de un compañero de la estrella, atrayendo gran parte del material hacia él mientras gira y tal vez producir una forma de mariposa. Así otros investigadores (principalmente Soker[9]) sostienen que es posible ignorar del todo la influencia del campo magnético y que la morfología puede ser atribuida casi exclusivamente a éstas interacciones, señalando por ejemplo que no es necesaria la presencia de un campo magnético para la formación de chorros.

Ninguna de las dos corrientes ha mostrado ser infalible y posiblemente la explicación final, si un día se conoce, involucre ideas de ambas partes. De cualquier manera es un tema de debate actual y merecedor de mención.

A continuación se presenta una breve explicación del por qué del brillo nebular y más adelante se desarrollará la idea detrás de los modelos de fotoionización empleados para producir los datos que GLNebula recibe como entrada.

2.3. Estructura de ionización

La nebulosa planetaria actúa como un filtro, absorbiendo parte de los fotones y permitiendo el libre tránsito de otros de manera que la luz que se detecta es producto de un sinnúmero de interacciones que ocurren a lo largo del cuerpo.

La emisión de fotones altamente energéticos por parte de la estrella central llevan al sistema a un estado de ionización: los fotones emitidos por radiación que excedan una energía de alrededor de 13.6 electrovolts (energía requerida para ionizar el Hidrógeno) y que en su camino choquen con un átomo, lo desprenderán (con cierta probabilidad) de un electrón llevándolo a un nivel mayor de energía (figura 2.7). Por ejemplo, si el átomo es de Hidrógeno (HI) al ser impactado por el fotón pasará a un nuevo estado de ionización HII, esto puede ocurrir más de una vez en elementos más cargados que el Hidrógeno, que sólo puede ser ionizado una vez.

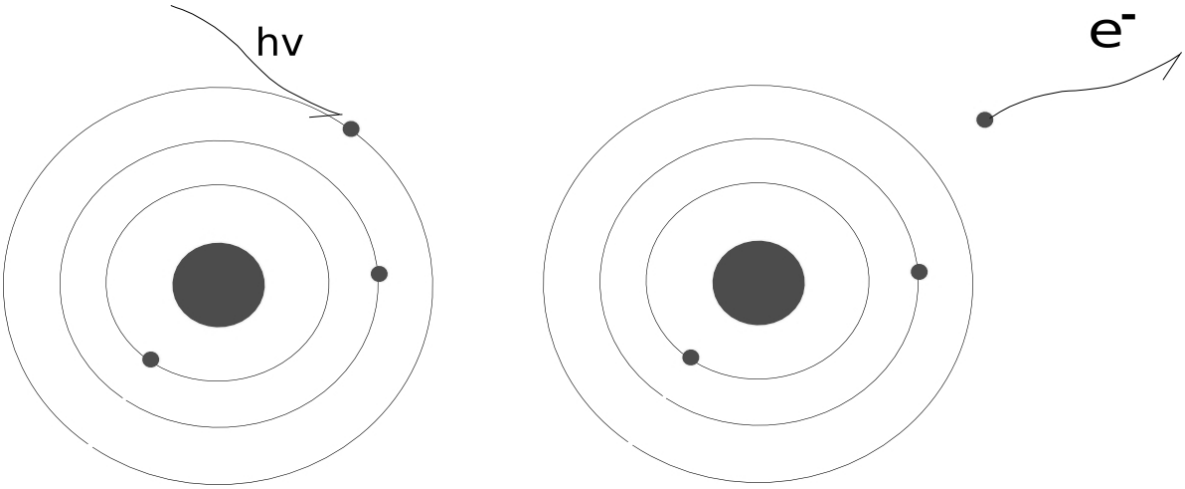

Figura 2.7: Fotoionización 1 : Se aproxima un fotón, 2 : Desprende al átomo de un electrón, ionizándolo

Los elementos fotoionizados tienden a presentarse en capas alrededor del núcleo nebular, debido a que la abundancia de fotones decrece con la distancia a la fuente,

dando paso a regiones donde predominan elementos ionizados de un tipo y escasean de otro, además de permitir el empleo de filtros de observación para atenuar o resaltar alguna región de interés.

La fotoionización ocurre cerca de estrellas con temperaturas bastante altas (mayores a 20 000 K) y en cuerpos como: estrellas gigantes, núcleos de nebulosas planetarias, novas y súper novas, y en núcleos activos de galaxias.

2.3.1. Recombinación en el modelo de Hidrógeno de Bohr

Existe otra interacción conocida como recombinación (figura 2.8), en donde los electrones libres pasarán a formar parte de algún ión que encuentren en su camino, eventualmente los electrones capturados regresarán a su nivel energético base emitiendo un fotón (o más si el regreso se da en cascada) con una longitud de onda particular dependiendo del estado energético en el que se encontraban previo a la recombinación, estas emisiones de fotones son las denominadas líneas de recombinación.

Empleando el modelo de Bohr para el Hidrógeno podemos enumerar las líneas de recombinación que ocurren en este átomo, según el tipo de luz emitida (ver figura 2.9):

- **Región Ultravioleta:** A esta región pertenece la llamada *Serie de Lyman* y la componen las líneas que aparecen en las transiciones que ocurren desde cualquier nivel energético al primero. Se indexan con el prefijo L seguido de $\alpha, \beta, \gamma, \delta, ...$ dependiendo del nivel del que provengan. Su longitud de onda va de los 121 nanómetros y converge a 91.15 nm, que es su límite inferior.

- **Región Visible:** En la región visible del espectro aparece la *Serie de Balmer*. Son las transiciones de cualquier nivel hacia el segundo, denotadas por una H, indexadas también por letras griegas (aunque sin usar prefijo) y comenzando en 656 nm (H_{alpha} correspondiente al rojo) hasta 410 nm (H_{delta}, violeta).

- **Región Infrarroja:** Tres series ocurren en esta parte del espectro, todas deno-

tadas con el nombre de su descubridor: La *Serie de Paschen* está compuesta por las líneas que aparecen en transiciones hacia el tercer nivel y van de los 1874 nm decreciendo hasta converger en el límite inferior de 820.1 nm; la *Serie de Brackett* engloba las transiciones hacia el cuarto nivel y ocurre de los 4522 nm a su límite inferior de 1459 nm; las líneas pertenecientes a la *Serie de Pfund* aparecen en movimientos hacia el quinto nivel y van desde los 7460 a 2280 nanómetros; finalmente la *Serie de Humphreys* va de los 12368 a 3282 nm, con líneas que aparecen en las transiciones hacia el sexto nivel.

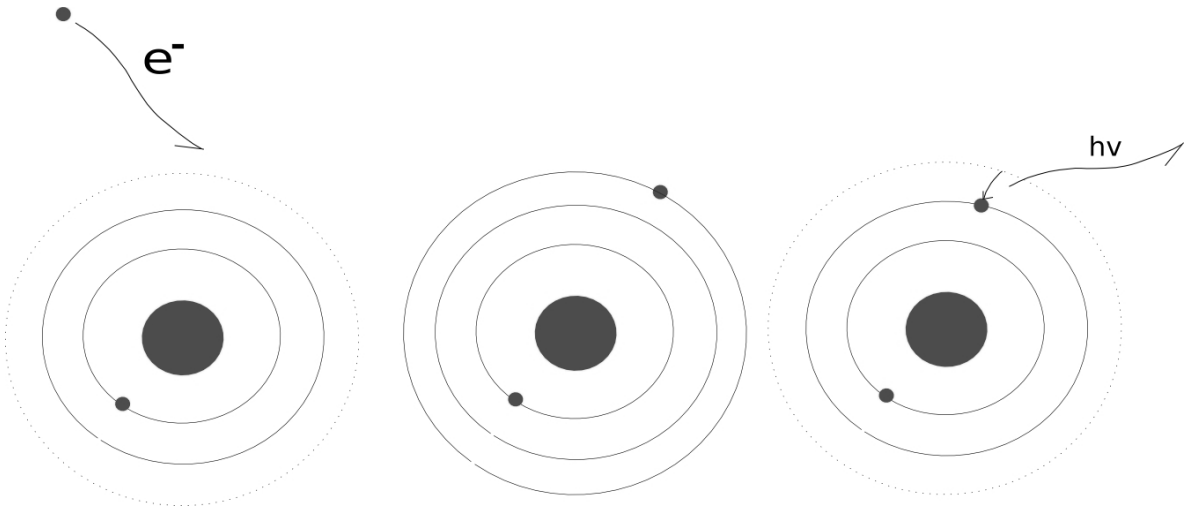

Figura 2.8: Recombinación 1 : Se aproxima un electrón libre, 2 : Se recombina con el átomo en un nivel energético mayor a su nivel base, 3 : Regresa a su nivel base, emitiendo un fotón

2.3.2. Ionización y Excitación colisional

Existe otro tipo de ionización, la ionización colisional, en la que en el interior del gas hay partículas con tremenda energía cinética, las cuales al chocar con sus vecinos provocan el desprendimiento de sus electrones. La excitación colisional con emisión se da de manera similar: con sólo aproximarse a sus vecinos pueden excitar a un electrón llevándolo a un nivel energético mayor y emitir uno o más fotones al regresar a su nivel

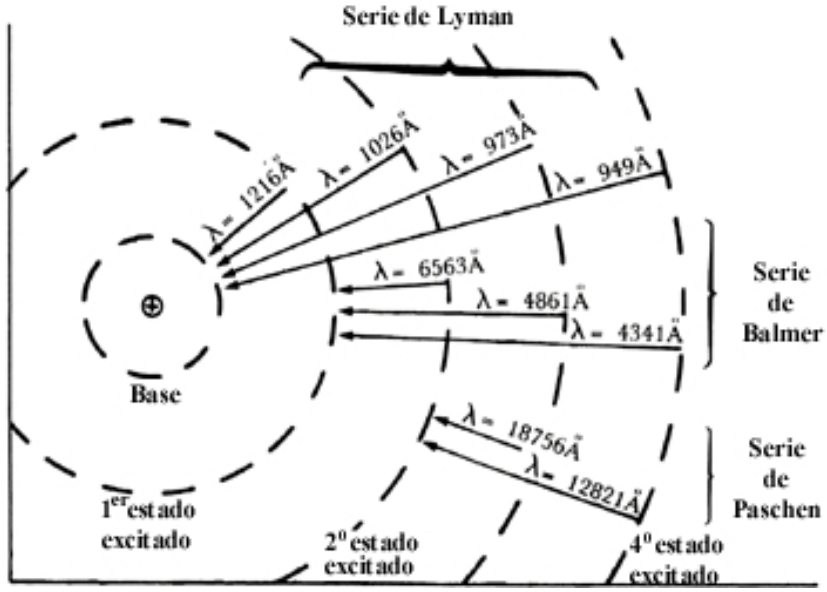

Figura 2.9: Modelo de Bohr del átomo de Hidrógeno y sus respectivas líneas de recombinación

energético base (figura 2.10). La ionización colisional amerita mención como miembro de la estructura de ionización, aunque no posee relación con los códigos empleados para los modelos y sólo se presenta en gases extremadamente calientes como la corona solar.

2.3.3. Equilibrios

Además de un estado de ionización las nebulosas presentan dos equilibrios: de ionización y térmico.

En el primer caso se estipula que los eventos de fotoionización se producen en igual número que la recombinación de electrones liberados por medio de los primeros, mientras que en el equilibrio térmico las ganancias aportadas por la energía cinética de los electrones libres, equivalen a las pérdidas presentes en forma de emisión de fotones. Estas dos nociones de equilibrio permiten crear modelos discretizando el proceso de recombinación y fotoionización (lo que en la nebulosa ocurre en cada parte infinitesimal se traduce en las celdas de una malla de resolución arbitraria), infiriendo así el número de fotones emitidos en cada proceso de recombinación y excitación por unidad de volumen

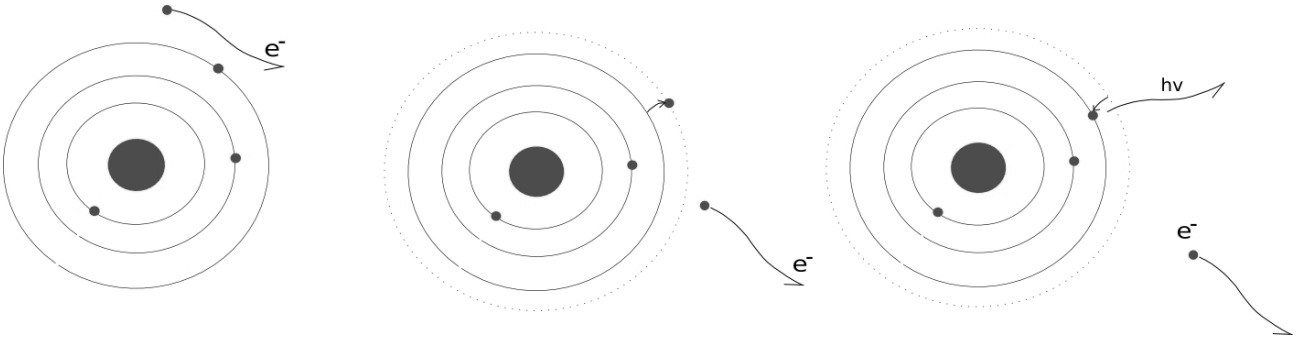

Figura 2.10: Excitación colisional con emisión 1 : Se aproxima un electrón libre, 2 : Excita a un electrón del átomo y éste incrementa su nivel energético, 3 : Al regresar a su estado base emite un fotón

en una nebulosa en particular.

2.4. Líneas de Emisión y Colores

La emisión luminosa de muchos cuerpos astronómicos, particularmente estrellas y objetos tipo *cuerpo negro*[1], se presenta en forma de un contínuo en el espectro y como una campana si se grafica la intensidad en función de la longitud de onda. Las nebulosas planetarias en cambio registran en el espectro una serie de líneas, llamadas *líneas de emisión*, debido a que los fotones emitidos por cada elemento presentan variaciones minúsculas en sus respectivas longitudes de onda, esto hace que las líneas salten a la vista en las mediciones y se presenten como picos muy angostos en las gráficas (casi monocromáticos, ver figura 2.11).

Existen dos tipos de líneas de emisión en una nebulosa planetaria y se distinguen por el tipo de interacción que produce los fotones: líneas de recombinación y líneas de excitación colisional, ambas tratadas en la sección anterior. Las líneas de excitación

[1] *Black-body:* Cuerpos que absorben la radiación electromagnética que pasa a través de ellos y producen luz si su temperatura es alta, como nuestro sol

colisional son conocidas como líneas prohibidas[2] (denotadas por un par de paréntesis cuadrados envolviendo al elemento).

Estas líneas son denominadas así debido a que el tiempo de vida en el nivel excitado colisionalmente es mucho mayor que en el caso de líneas permitidas: en condiciones de densidad normales la desexcitación colisional es lo suficientemente probable para que ocurra en un tiempo relativamente corto posterior a la excitación, de manera que no se emite un fotón; en cambio en ambientes con densidades muy bajas la probabilidad de que ocurra una colisión con un electón libre que transforme la energía del nivel excitado en energía cinética propia es sumamente baja, así al electrón le da tiempo de regresar a su nivel base y emitir un fotón. En los laboratorios se tiene una densidad tal que no permite la desexcitación radiactiva con emisión ya que la probabilidad de que no ocurra una colisión antes de que el electrón regrese a su estado base es casi cero, por ello el término de líneas prohibidas.

El hecho de que las líneas de emisión sean picos casi monocromáticos (como el caso de las lámparas de Sodio de muchos túneles) facilita la posibilidad de asignarle un color particular a cada línea observada, de manera que es posible producir una imagen en RGB[3] de la emisión escogiendo 3 líneas de interés. Este hecho es el que está detrás de la coloración en GLNebula: se toma una capa representada por un cubo de datos de emisión, digamos H_α, se le asigna un color y una intensidad particular, digamos rojo y se hace lo mismo para otras dos capas, digamos $HeII$ y $[OIII]$ coloreadas con verde y azul respectivamente; al final se obtiene una imagen compuesta con los tres colores empleados.

En la sección correspondiente al programa se trata el procedimiento con más detalle.

[2]Observadas por William Huggins: Astrónomo británico que llevó a cabo muchas observaciones de líneas de emisión y absorción de varios objetos celestes. En ese momento no se conocía un elemento capaz de emitir en la frecuencia de 5007 Å (línea que predomina en intensidad y ahora es atribuída al Oxígeno dos veces ionizado), por lo que se le atribuyó a un nuevo elemento: el *nebulium*

[3]Del inglés:Red, Green, Blue. Imagen compuesta por tres colores primordiales: Rojo, Verde, Azul

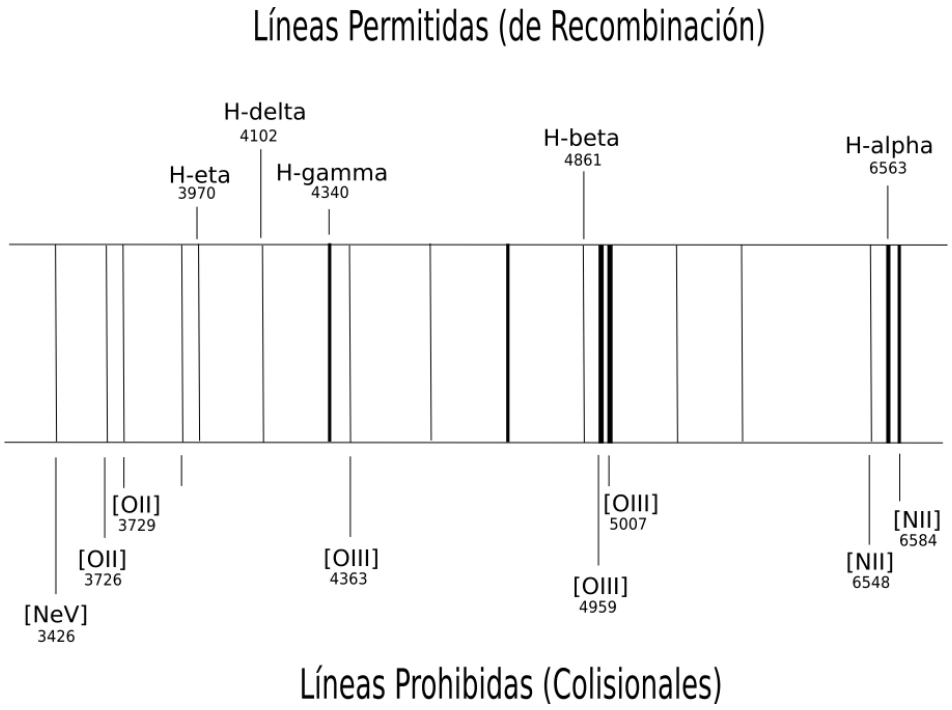

Figura 2.11: Algunas líneas de emisión y sus longitudes de onda (en Ångstroms)

Capítulo 3

Modelos de Fotoionización y Visualización Científica

Este capítulo tiene como propósito servir de transición entre los mundos astronómicos y computacionales. Se pretende cerrar lo pertinente a la física nebular con un enfoque algorítmico, para comenzar a ver las cosas desde la perspectiva computacional. Posteriormente se dará una breve introducción a la visualización científica para finalizar con la descripción de algunas técnicas empleadas en este ámbito.

3.1. Modelos de Fotoionización

La idea detrás del desarrollo de modelos de fotoionización de nebulosas planetarias radica en tener una forma de describir completamente un cuerpo a partir de ciertos parámetros para así tener un mayor entendimiento de la física del objeto y tal vez encontrar el menor número de variables que lo determinen completamente. Esto permitiría no sólo mejorar el conocimiento actual de las nebulosas planetarias sino también desarrollar nuevas teorías sobre la evolución estelar, o complementar las ya existentes.

Ercolano[10] asevera: “Las nebulosas planetarias [...] suelen ser transparentes a la

radiación en forma de líneas de emisión producida en su interior y, por ende, una predicción de su espectro requiere solamente un conocimiento de la temperatura electrónica y de la estructura de ionización en cada posición del volumen ionizado. Esto puede lograrse resolviendo numéricamente el par de ecuaciones para los equilibrios de ionización y térmico".

De manera que se requiere resolver un sistema de ecuaciones para cada unidad de volumen de la nebulosa, así es natural pensar en el uso de una malla de resolución arbitraria para encapsularla (en el caso tridimensional) o bien trazar una línea y subdividirla en muestras (en el caso de modelos con simetría esférica) y, mediante la aplicación de técnicas estadísticas, realizar un seguimiento de las reacciones que ocurren en cada celda o punto. Al final se busca obtener una descripción del gas ionizado, particularmente la intensidad de las líneas emitidas y su temperatura. Obviamente la calidad del modelo y el tiempo de procesamiento están ligados a la resolución de la malla y el número de cálculos a realizar en cada celda, requiriendo recursos computacionales inmensos que en el pasado no estaban disponibles, por lo que las simulaciones tardaban un tiempo a veces prohibitivo, los modelos resultantes no eran tan detallados y además unidimensionales.

Para realizar los cálculos es necesario contar con parámetros que describan las propiedades de los elementos que componen el gas, como: probabilidades de transición de nivel energético de los electrones, probabilidad de recombinación, cantidad de energía cedida en cada interacción, etc. De manera que se requiere además una base de datos de física atómica, mismas que a finales de los sesenta (cuando empezaron a surgir códigos de fotoionización) no existían o estaban sumamente incompletas. Hoy en día el gran desempeño y el relativo bajo costo del hardware, así como la existencia de bases de datos extensas permiten el desarrollo de códigos muy completos, de muy alta resolución, con posibilidad de realizar los cálculos en 3D y con tiempos de procesamiento mucho menores a los que anteriormente se tenían.

3.1.1. Unidimensionales

Hasta hace poco los códigos de fotoionización disponibles trabajaban sobre modelos unidimensionales, realizando sus cálculos a lo largo de una única línea, principalmente por limitaciones asociadas con los tiempos y el hardware. La tendencia en estos momentos apunta a tener códigos tridimensionales, pero algunos de los actuales emplean versiones 1D para sus cálculos, así que su importancia sigue vigente.

La estrategia a seguir consiste en emplear como datos de entrada las características del flujo ionizante (dirección, tipo, temperatura, entre otras), del gas a ionizar (composición, densidad, distribución, etc.) y se complementa con información obtenida de una base de datos de física atómica. Se procede a calcular los equilibrios térmicos y de ionización para un conjunto de puntos sobre la línea, resolviendo un sistema de ecuaciones para cada caso y así obtener al final una descripción de características de la nebulosa como: temperatura del gas, densidad electrónica, intensidad de las líneas emitidas, etc.

A continuación se presenta una descripción de los códigos 1D y tridimensionales, enumerando algunos de los miembros que pertenecen a cada categoría y sus características y desventajas principales.

Cloudy

Cloudy[1] figura entre los más completos códigos de fotoionización en existencia. Su desarrollo comenzó en 1978 y continúa a la fecha. Es abierto al público, originalmente escrito en FORTRAN, en 1989 se tradujo a C y actualmente está desarrollado en C++. El autor original es Gary Ferland aunque ahora hay varios colaboradores que constantemente introducen nuevos parámetros y características al sistema.

[1] http://www.nublado.org/

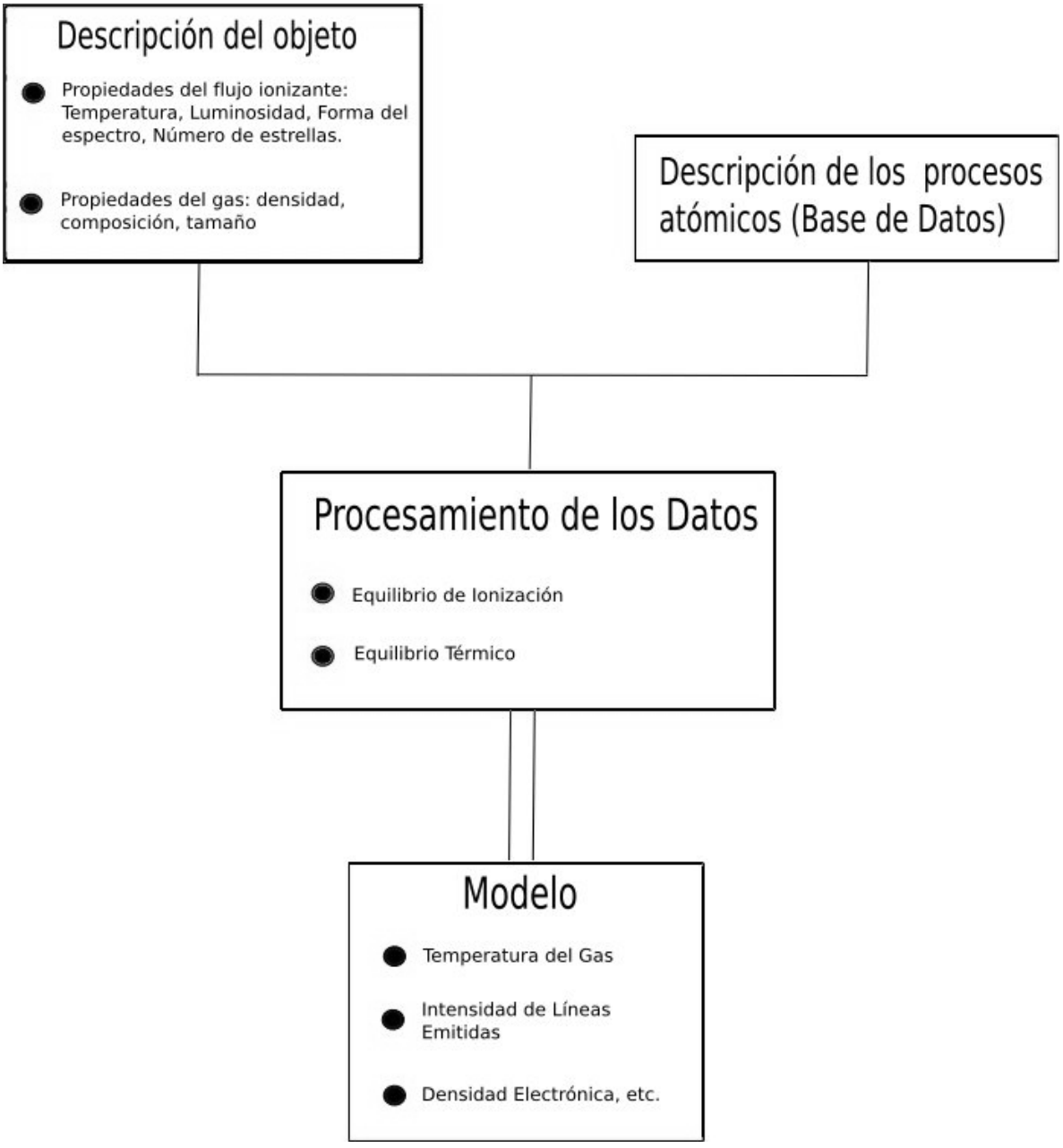

Figura 3.1: *Pipeline* de los códigos 1D

NEBU

NEBU es un código de fotoionización desarrollado por Péquignot, Stasinska y Viegas, escrito en FORTRAN y creado en la década de los setentas. Emplea como entradas descripciones del flujo ionizante y del gas a ionizar, además de la longitud del radio interno (distancia del núcleo al envoltorio), composición química y ley de densidad (constante, lineal o más compleja) del gas y se complementa con información de una base de datos atómica. A la salida se obtiene: la densidad y temperatura electrónicas del gas y las emisividades de líneas.

3.1.2. Tridimensionales

El inmenso poder computacional disponible actualmente y el relativo bajo costo del hardware han permitido el desarrollo de códigos más sofisticados, siendo el avance más notable el paso de códigos unidimensionales a tridimensionales, aunado a esto las bases de datos de física atómica han crecido considerablemente, por lo que los cálculos obtenidos son ahora más precisos. A continuación se presentan dos de los códigos de fotoionización tridimensionales existentes, explicando brevemente sus características principales y su ventajas y desventajas.

CLOUDY_3D

Cloudy_3D[2]([4]) es la extensión a tres dimensiones de NEBU, aunque puede ser usado con cualquier código de fotoionización unidimensional. Emplea IDL[3], una plataforma que provee muchas herramientas para el análisis y visualización de datos. Trabaja sobre la suposición de que el objeto tiene simetría axial: se toman muestras regulares de la nebulosa empleando el radio interno r y dos ángulos: θ y φ. Para una muestra dada de ambos ángulos, se corren en paralelo códigos unidimensionales a lo largo del radio r, al final se obtiene una malla tridimensional en donde cada celda cúbica corresponde a

[2] $http://132,248,1,102/Cloudy\_3D/$

[3] http://www.rsinc.com/idl/

una terna (r, θ, φ) y contiene la información de salida (de nuevo: intensidad de líneas, temperatura, etc.) en cada cubo.
El hecho de emplear códigos unidimensionales lo pone en desventaja frente a otros códigos como Mocassin: debido a que se itera un código 1D los cálculos de transferencia de radiación sólo se realizan a lo largo de cada radio, de manera que no se hace un seguimiento de las interacciones entre celdas vecinas que no se encuentren a lo largo de una misma línea, pero tiene la ventaja de ser sumamente veloz, obteniendo resultados en minutos y siendo de 3 a 4 órdenes de magnitud más rápido que Mocassin[4].

Asociadas a Cloudy_3D existen un par de herramientas desarrolladas también en IDL: VISNEB_3D usada para la visualización de la salida que permite realizar rotaciones, acercamientos y proyecciones para obtener gradientes de brillo; y VELNEB_3D que puede ser aplicado a los resultados de los modelos para generar perfiles de líneas de emisión, así como mapas Posición-Velocidad, asumiendo un campo de velocidad arbitrario. Ambas pueden trabajar con la salida de cualquier código de fotoionización, siempre y cuando se encuentren estructurados en cubos con el orden de crecimiento adecuado: primero X, luego Y, al final Z.

Mocassin

Mocassin[4] es un código desarrollado por Ercolano[11] y otros, que emplea métodos estocásticos[5] en sus cálculos y, a diferencia de otros sistemas, permite la reconstrucción de nebulosas planetarias sin restricciones morfológicas, es decir: no presupone una simetría esférica de los cuerpos a modelar, lo que es sumamente útil si se considera que sólo el 10 % de las nebulosas planetarias observadas presentan una morfología de este tipo[12]. Además es posible alimentarle funciones de densidad complejas y puede resolver modelos que posean una o más estrellas ionizantes en posiciones arbitrarias (no sólo centrales, aunque rara vez se presentan en nebulosas planetarias). De nuevo se intenta

[4] *http : //hea − www.harvard.edu/ bercolano/*

[5] Concretamente de Monte Carlo, de ahí el nombre: MOnte CArlo SimulationS of Ionised Nebulae

simular localmente los equilibrios térmicos y de ionización, encapsulando a la nebulosa en una malla tridimensional (con la posibilidad de ser no-uniforme, dependiendo de la distribución de densidad) y resolviendo para cada celda. Todas estas características lo hacen un código muy completo, aunque desgraciadamente también hacen que no sea tan veloz como se desearía.

Actualmente se está tratando de juntar los códigos de fotoionización con algunos sistemas de modelación hidrodinámica, con esto se espera obtener cálculos mucho más precisos de las reacciones que ocurren en el interior y además crear modelos que tomen en cuenta el tiempo, para así poder simular la evolución de la nebulosa en sus distintas etapas.

3.2. Visualización Científica

La faceta didáctica de GLNebula no requiere mayor justificación, se pretende atraer al público hacia la astronomía (y tal vez a la forma de hacerla) presentando imágenes en tiempo real de objetos coloridos. Por otro lado si pretende ser una herramienta de visualización científica, entonces es pertinente explicar en qué consiste la materia, mencionando brevemente algunas de las técnicas empleadas, una de las cuales fue la elegida para el diseño original.

En 1987 en un congreso titulado *"Visualization in Scientific Computing Workshop"* la *National Science Foundation* se refirió a la visualización científica como: "[...] un método computacional. Transforma lo simbólico en lo geométrico, permitiendo al investigador observar sus simulaciones y cómputos. La visualización ofrece un método para ver lo invisible. Enriquece el proceso de descubrimiento científico y proporciona entendimientos profundos e inesperados. En muchos campos está revolucionando la manera en que los científicos hacen ciencia[...] La meta de la visualización es apuntalar métodos científicos existentes usando nuevo entendimiento científico obtenido a partir de métodos visuales".

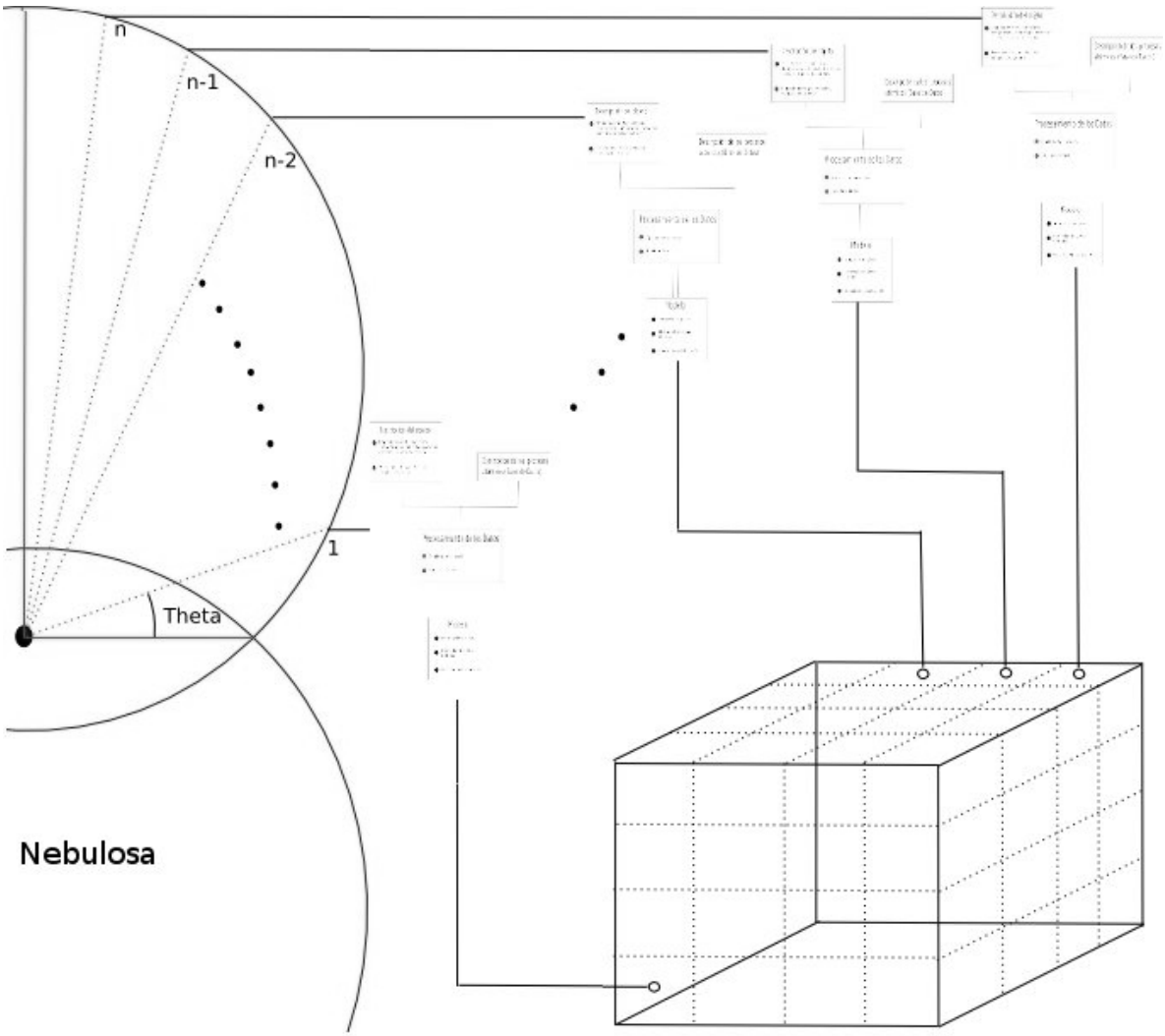

Figura 3.2: *Pipeline* de Cloudy_3D: Se corre el código unidimensional múltiples veces y se obtiene un cubo con una muestra por celda, correspondiente a cada (r, θ, φ). La imagen es intencionalmente borrosa y sólo busca ser representar múltiples ejecuciones del caso unidimensional empleando la figura 3.1

No se espera revolucionar la manera de hacer ciencia, pero sí es de la incumbencia de GLNebula probar ser una herramienta útil para asistir a los investigadores en la visualización de sus objetos de estudio, tal vez ayudando a la creación de una imagen mental más detallada de la nebulosa que pueda proveer algún entendimiento extra que de otra manera no hubiera salido a la luz.

La visualización científica es una rama de la graficación por computadora encargada de aplicar transformaciones a datos científicos para obtener un despliegue visual que permita entender el proceso representado por la información. Así su herramienta principal son las transformaciones en forma de algoritmos, las que normalmente aparecen en forma de filtros a lo largo del *pipeline* de manera que la entrada de un filtro puede ser la salida de su antecesor y así consecutivamente, transformando los datos hasta obtener la visualización.

Es posible catalogar las transformaciones dependiendo de su tipo:

- **Transformaciones Geométricas:** En las que se altera la geometría pero no la topología de los datos de entrada, es decir: la esencia del objeto (coordenadas relativas de sus puntos) se mantiene. Rotaciones, traslaciones y escalamiento uniforme de las coordenadas son un ejemplo.

- **Transformaciones Topológicas:** La topología de un conjunto de datos es la forma en la que están estructurados sus miembros: mallas uniformes, rectilíneas, etc. Normalmente los cambios en la topología implican cambios geométricos, aunque hay excepciones: transformar una malla triangular en una no estructurada, donde las celdas no tienen un espaciamiento uniforme y no poseen la misma forma entre sí, es un ejemplo.

- **Transformaciones de Atributos:** Tienen que ver con el tipo de atributos o datos asociados a la topología o geometría del conjunto: temperatura, presión, intensidad, y no con su estructura. La manipulación y creación de escalares basándose en alguna propiedad como datos de elevación, pertenecen a este tipo.

- **Transformaciones Mixtas:** Se encargan de cambiar tanto la estructura de los datos como sus atributos, siendo el cálculo de isosuperficies, que más adelante se tratará, un ejemplo.

También se pueden catalogar de acuerdo al tipo de datos sobre los que actúan:

- **Escalares:** Trabajan con un único valor asociado por cada punto o celda del conjunto de datos, por ejemplo la coloración de regiones de un mapa según su elevación.
- **Vectoriales:** Manipulan representaciones tridimensionales de dirección y magnitud, la visualización de flujos y campos vectoriales son ejemplos de este tipo.
- **Tensoriales:** Operan en matrices tensoriales, por ejemplo un modelo que visualice las fuerzas que actúan sobre determinado material.

Además de éstas categorías están los algoritmos de modelado: aquellos que generan la topología y la geometría de los datos, la construcción de una malla triangular a partir de un conjunto de puntos aleatorios en el espacio es un ejemplo.

3.2.1. Técnicas y algoritmos

Triangulación de Delaunay

La triangulación de Delaunay es una técnica muy usada para la construcción de mallas triangulares a partir de un conjunto de puntos en el espacio. Tiene la propiedad de que ningún punto sobre los que se va a triangular cae dentro del círculo que engloba a los vértices que definen una celda triangular, es decir: no está contenido en el circuncírculo de cualquier triángulo. Además se ha demostrado que una triangulación es única para cada conjunto de puntos siempre y cuando no se tengan posiciones degeneradas: tres puntos colineales o cuatro dentro de un mismo círculo para el caso bidimensional y cuatro puntos coplanares y cinco dentro de una esfera para el caso 3D (es decir: $n+1$ puntos en el mismo hiperplano y $n+2$ en la misma hiperesfera para el caso n-dimensional).

La parte central del algoritmo requiere una forma de decidir si agregar o no un vértice a la triangulación actual, esto se logra revisando que el vértice en cuestión no se encuentre contenido en el circuncírculo de cada triángulo existente. En caso positivo (y también asegurándose de que la región no sea degenerada) se procede a guardar las aristas de los triángulos que lo cumplan (quitando las repeticiones) y se retiran de la triangulación, para posteriormente formar un triángulo con cada arista de las almacenadas y el vértice a agregar. Al final se añade la nueva triangulación realizada a la ya existente, repitiendo el procedimiento para cada vértice que aún no se encuentre en la triangulación.

En resumen, si se desea agregar el vértice v a T y llamamos E al conjunto de aristas:

1. Revisar que v no esté en el circuncírculo de ningún $t \epsilon T$
2. Si v no está contenido, se triangula y continúa con otro vértice
3. Si v está contenido, entonces remover todos los t que lo cumplan, quitando las aristas que no afecten al resto de los triángulos
4. Se guardan las aristas que se removieron previamente en $E^{'}$, quitando aristas dobles
5. Se forman nuevos triángulos entre cada $e \epsilon E^{'}$ y v, a la triangulación nueva se le llama $T^{'}$
6. Se agrega cada $t \epsilon T^{'}$ a T
7. Se repite hasta que cada v haya sido agregado a T

Algunas implementaciones emplean un valor de tolerancia que determina la distancia mínima entre dos puntos, de manera que los que se consideren coincidentes sean descartados del algoritmo.

Isosuperficies

El cálculo de contornos o superficies con valor escalar constante es una de las técnicas más empleadas en la visualización científica, usadas por ejemplo para resaltar isotermas[6] en mapas climatológicos y determinadas ondas de choque en el caso de modelos

[6]Líneas de temperatura constante

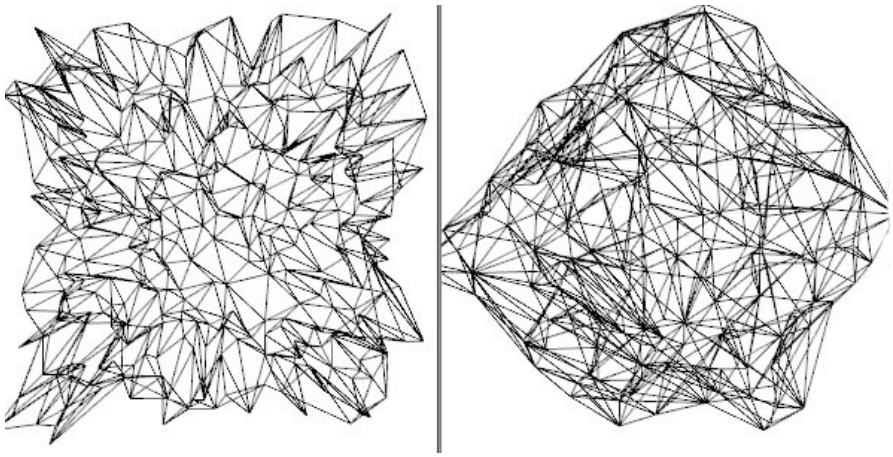

Figura 3.3: Triangulación de Delaunay con tolerancia del 0.01 aplicada a 500 puntos aleatorios. 2D (izquierda), 3D (derecha)

hidrodinámicos de aeronaves. La idea consiste en leer los datos (deben estar en forma de una malla) y localizar los escalares que sean iguales al valor deseado, para extraer un borde en el caso bidimensional o una superficie en el 3D, aunque muchas veces el denominado *isovalor* se encuentra a la mitad de los escalares disponibles, de manera que se requiere interpolar para aproximar la superficie, operación que además ayuda al cálculo de normales para sombreado cuando se tiene un modelo de iluminación.

Uno de los algoritmos más conocidos de extracción de isosuperficies en mallas poligonales es Marching Cubes[14] y su variación bidimensional Marching Squares. Se comienza construyendo cubos imaginarios tomando los valores del campo escalar de ocho en ocho. El algoritmo asume que una superficie puede pasar por una celda en un número finito de formas poligonales, así que se construye una tabla de *estados topológicos* compuesta por las posibles combinaciones. En un cubo hay $2^8 = 256$ formas en las que una superficie puede pasar a través de una celda, por ende la tabla de estados para el caso tridimensional tendría 256 entradas, afortunadamente muchas son rotaciones y reflexiones de otras así que se pueden reducir a 15 casos básicos (ver figura 3.4).

Para determinar la primitiva que corresponde a determinada superficie es necesario buscar el patrón en la tabla, esto se logra indexando los cubos de acuerdo a un criterio

de contención: si el valor de un vértice excede al isovalor se considera dentro de la superficie, en caso contrario está fuera. Los vértices contenidos se marcan con 1 y los ajenos con 0, así el índice correspondiente a un cubo en particular está compuesto por 8 bits representando los estados de sus vértices (4 en el caso 2D).

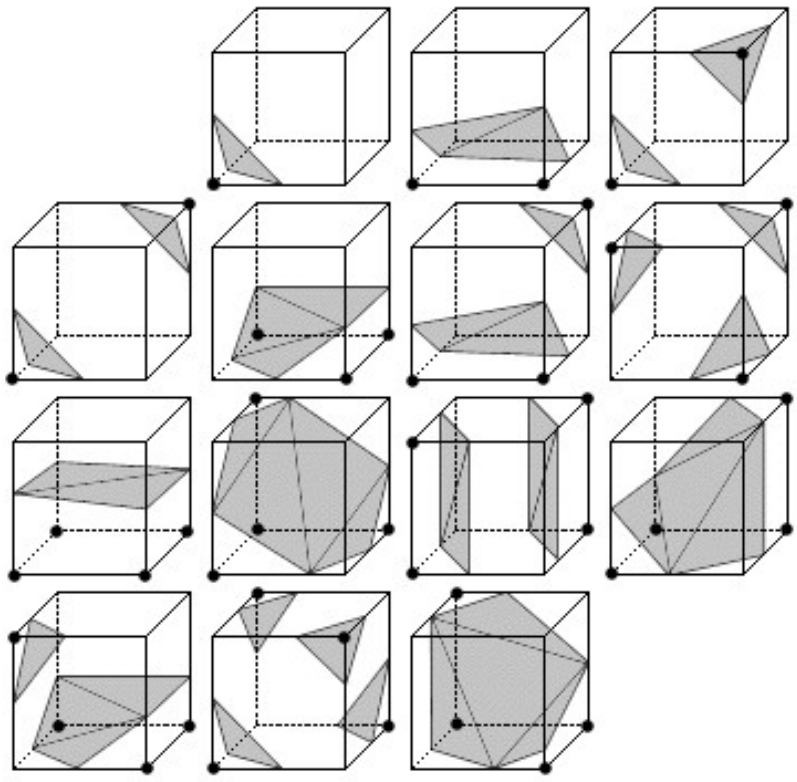

Figura 3.4: Estos 14 casos más el vacío son las maneras en las que una superficie puede pasar por cada cubo (cortesía de Martin Franc, *University of West Bohemia*)

Así el algoritmo procede de esta manera:

1. Se selecciona una celda
2. Se calcula el estado de sus vértices: 1 si su valor es mayor al escalar, 0 en otro caso

3. Se crea el índice del cubo usando el estado de sus ocho vértices
4. Se busca en la tabla de estados topológicos
5. Usando interpolación se calcula la locación de la superficie para cada arista en la tabla de casos
6. Se repite con todas las celdas

Finalmente se obtienen primitivas independientes dentro de cada celda (comúnmente triángulos), las cuales pasarán a ser visualizadas.

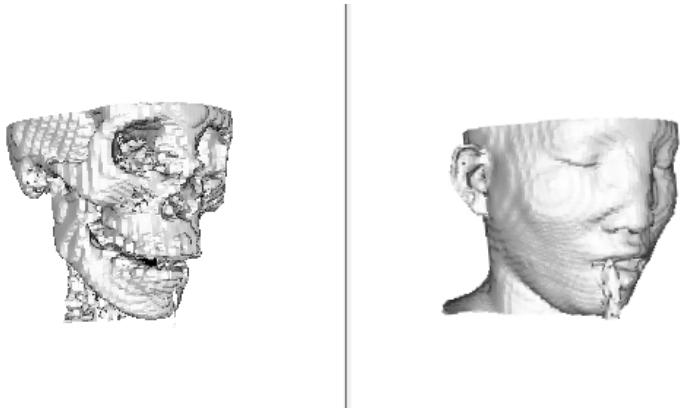

Figura 3.5: Extracción de isosuperficies aplicadas a un modelo, primero con valor escalar 1150 y luego 500

Decimación

La decimación es una técnica de reducción de datos que se aplica sobre mallas para disminuir el número de celdas de acuerdo con cierto criterio (si éstas son poligonales, la técnica se conoce como reducción de polígonos). Para lograrla se definen tres operaciones: el colapso de aristas, la reducción de vértices y el borrado de celdas.
En el primer caso se emplea un valor de tolerancia: si la longitud de determinada arista es menor a dicho valor la arista colapsará hacia su vértice más cercano, de manera que los vértices en sus extremos pasarán a ser sólo uno. La reducción de vértices trabaja de la misma forma, borrando el vértice que se considera innecesario y por lo tanto sus celdas asociadas, para después retriangular el hueco que queda. El borrado de celdas

opera de nuevo con la noción de tolerancia, borrando una celda y retriangulando otra vez.

El valor de tolerancia está usualmente determinado por una métrica que trata de minimizar el error que produciría el cambio en cuestión con respecto a los datos originales, probablemente la más usada se basa en la acumulación del error representado por una función cuadrática: midiendo la distancia a un conjunto de planos, cada uno correspondiente a una celda en la malla original, minimizando esta distancia es como se decide qué operación utilizar para obtener el cambio más adecuado.

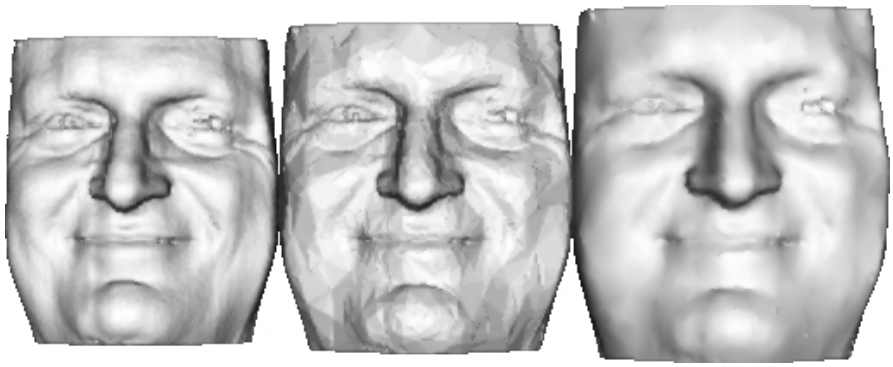

Figura 3.6: Decimación del 80 % (centro) aplicada al modelo original (izquierda), decimación con sombreado suave (derecha)

Volume Rendering

Volume rendering directo es una técnica empleada para obtener proyecciones 2D a partir de datos volumétricos: normalmente un conjunto de muestras (x, y, z, v) llamadas *voxeles* que representan un valor v (temperatura, densidad, etc.) dada una posición espacial (x, y, z), aunque hay casos donde v puede ser un vector (representando por ejemplo velocidad) o bien las muestras pueden variar con el tiempo, siendo 4D. Las muestras son tomadas a intervalos regulares y, debido a que es una técnica muy empleada en visualizaciones médicas, normalmente son obtenidas a partir de lecturas de un escáner MRI o CT. Para hacer la visualización es necesario tener una manera de colorear cada voxel, definiendo sus valores en una tripleta RGB más un canal Alpha

que determina la transparencia de cada uno, esto se logra empleando una función de transferencia en donde en cada entrada se corresponde un valor determinado a un color en particular.

Existen varias formas de hacer *volume rendering*:

- **Volume Ray Casting:** Este método es computacionalmente muy intensivo, pero permite el uso de iluminación y transparencia para alcanzar mayor realismo que con otros algoritmos. La idea es trazar un rayo por cada pixel que inicie en el centro de perspectiva del observador y atraviese todo el volumen, realizando cálculos en cada muestra. En la implementación de Blinn/Kajiya tenemos un modelo con cierta densidad que va a ser penetrado por un rayo, además de una o más fuentes de luz. En cada muestra es necesario calcular el nivel de intensidad de luz, dependiendo de la densidad del volumen en ese punto. Al final para obtener la intensidad luminosa que le llega al observador desde la dirección del radio se suma las intensidades individuales de cada muestra, haciendo la integral:

$$B = \int_{t_1}^{t_2} e^{(-\tau \int_{t_1}^{t} D(s)ds)} I(t)D(t)P(cos\theta)dt$$

Donde D corresponde a la densidad del volumen, I la luz que llega a una muestra, $P(cos\theta)$ el ángulo del vector de luz desde cada punto y τ una constante para convertir densidad en atenuación. Esto lo hace una técnica computacionalmente muy intensiva, sacrificando tiempo y poder de procesamiento por realismo, aunque hay varias formas de optimizar el proceso que usualmente emplean formas para distinguir los objetos invisibles al observador.

Resumiendo:

1. Se toma un pixel y se traza un rayo
2. Se toman muestras (puntos) en el rayo uniformemente espaciados
3. Se calcula la opacidad y el color de cada punto usando interpolación
4. Se acumula la contribución de color y transparencia del punto actual al rayo (integrando)

5. Al pixel se le asigna el color del rayo correspondiente

6. Se continúa el proceso para cada pixel

La técnica tiene la ventaja de que la calidad depende del número de rayos, así que no se requiere una imagen de muy alta resolución, es posible obtener la visualización en un tiempo mucho menor reduciendo la cantidad de los mismos. Además, debido a sus características, es una técnica ideal para realizar procesamiento en paralelo, dividiendo la imagen en regiones y alimentado cada subconjunto a *pipelines* distintos, reduciendo el tiempo de generación de imágenes considerablemente.

- **Texture Mapping:** Desde hace unos años las tarjetas de video se han vuelto muy veloces para operaciones de mapeo de texturas, esta característica puede ser aprovechada para aplicar volume rendering empleándolas para modelar los polígonos. Existen dos aproximaciones dependiendo del tipo de texturas soportada por el hardware: la primera consiste en tomar cortes del volumen y mapearlos como texturas bidimensionales en polígonos, tomando en cuenta sólo aquellos perpendiculares al observador. La segunda aproximación requiere soporte para texturas tridimensionales, también empleando un conjunto de polígonos perpendiculares dentro del volumen a los cuales se les aplicará una textura 3D, empleando el color y opacidad del volumen original. Posteriormente se hace un mezclado de las aportaciones de cada polígono texturizado de atrás hacia adelante, obteniendo la imagen final. En ambos casos se suelen emplear tres conjuntos de polígonos y texturas perpendiculares, para así poder escoger el que posea mayor número de polígonos ortogonales a la dirección del observador y evitar desplegar esquinas. Aún siendo casi completamente basada en hardware, la calidad de esta técnica suele ser menor que ray casting y está ligada al número de cortes que se le realicen al volumen original, de nuevo sacrificando realismo por velocidad y conveniencia.

- **Splatting:** De entre las tres mencionadas splatting es la que posee mayor balance entre velocidad y realismo. El procedimiento consiste en proyectar los voxeles

en un plano tomándolos como discos para obtener la *huella* del voxel, siendo un círculo para proyección paralela o con forma elipsoidal cuando la proyección es en perspectiva. El color y transparencia que el voxel contribuye a la imagen está determinado por una distribución gaussiana en torno al centro de la huella, por ello comúnmente se le conoce como *Gaussian Splatting*. La proyección se realiza de atrás hacia adelante, tomando cada elemento de volumen, obteniendo los discos de proyección, calculando sus propiedades (color y transparencia) de acuerdo con la función elegida y acumulando su contribución para así obtener la imagen final haciendo una composición de las aportaciones acumuladas.

Figura 3.7: Volume Rendering aplicado a cortes del cerebro extraídas por CAT. Se empleó *ray casting*.

A continuación se presenta el capítulo más importante del trabajo: el correspon-

diente a la herramienta final.

Capítulo 4

GLNebula

En este capítulo se presentan las dos aproximaciones tomadas para el desarrollo del sistema: *volume rendering* y *billboarding*[1], empleando VTK y OpenGL respectivamente. El nombre de la herramienta deja sin dudas qué aproximación fue por la que se optó, pero antes de tratar con la versión final se presenta el primer método, argumentando la elección de la biblioteca VTK para su implementación, así como los problemas que aparecieron durante su uso. Posteriormente se trata el segundo camino tomado, el por qué de usar OpenGL, por qué satisfizo las metas propuestas y las características de la última versión.

4.1. Objetivos

Recapitulando: el objetivo principal de este trabajo era obtener una visualización en tiempo real de una nebulosa planetaria a partir de datos producidos por modelos de fotoionización. Se reciben tres datos de entrada, correspondientes a tres capas modeladas de la nebulosa según el elemento ionizado. Los datos son texto ASCII organizados en forma de cubos uniformes (misma distancia entre celdas), creciendo primero con respecto al eje X, luego Y y por último Z. Cada capa (cubo) debía corresponder con un

[1]Que al no ser una técnica de visualización científica, no fue explicada en el capítulo anterior y se presenta más adelante

canal de color: RGB, además de presentar transparencia (canal Alpha) para permitir el paso de la luz y así dar la apariencia de nube. Cada entrada en los datos corresponde a una celda en el cubo, cada valor en la celda representa el número de fotones emitidos cada segundo por unidad de volumen (que al ser todos los cubos del mismo tamaño es constante y por lo tanto se supone igual para cada celda, ver 5.1.2 para planes futuros sobre este tema) y la transparencia se asume la misma para todas.

Con esto en mente se tenía que construir la visualización de la nebulosa, permitiendo libre movimiento a la cámara, control de parámetros (correspondencia RGB de los datos, transparencia) además de mantener el uso de recursos en un nivel aceptable de manera que la imagen se redibujara 5 o 6 veces por segundo y así proveer interactividad.

4.2. Primera aproximación: Volume Rendering

El propósito original del proyecto era la aplicación de *volume rendering* mediante *ray casting* sobre una malla tridimensional, de manera que el modelo de transferencia de color y opacidad se mantuviera lo más apegado posible a su contraparte física, obteniendo idealmente la apariencia de una nube luminosa o un "plasma", que tal vez de otra manera no fuera posible de conseguir (lo cual probó ser falso, afortunadamente). Debido a que la complejidad que presenta desarrollar un sistema de trazado de rayos desde cero excede los conocimientos y tiempo disponible, se buscaron sistemas que tuvieran una implementación o al menos facilitaran las herramientas para desarrollarlo. En la búsqueda aparecieron algunas posibilidades, pero la de mejores características resultó ser VTK.

4.2.1. Visualization Toolkit

Visualization Toolkit (VTK) es un sistema diseñado por Kitware[2] para el desarrollo de aplicaciones dentro del campo de la graficación por computadora, particularmente la

[2]http://kitware.org

visualización científica y la manipulación de imágenes médicas. Consiste en una serie de clases escritas en C++ que proveen estructuras de datos, funciones y varias utilidades de muy alto nivel cuyo propósito es facilitar la transformación de los datos en imágenes y así permitir que el desarrollador se enfoque en el tratamiento de los datos más que en cuestiones técnicas (aunque debido a la pobre documentación existente es a veces un poco complicado su desarrollo, hecho que se trata de compensar con guías de uso a la venta). Gracias a estas características es posible realizar visualizaciones aparentemente complicadas en unas cuantas líneas de código, que de otra manera serían casi imposibles de desarrollar por una sola persona en un tiempo razonable. Además VTK es multiplataforma, gratuito y en su mayor parte libre (existiendo clases patentadas).

La biblioteca está dividida en dos módulos generales, caracterizados por su uso:

Módulo de Visualización:

El módulo de visualización es el encargado de transformar la información en datos gráficos de manera que puedan ser desplegados por el módulo gráfico, es decir, es el responsable de construir la representación geométrica a visualizar a partir de los datos de entrada.
Lo componen dos tipos de objetos:

- **vtkDataObject**: Es la representación interna de los datos en VTK, las estructuras geométricas y topológicas que contendrán los datos en arreglos de puntos, mallas uniformes, etc.; además de contener a los atributos asociados a la información como escalares y vectores.

- **vtkProcessObject**: A este conjunto pertenecen todas aquellas operaciones que se pueden realizar sobre los objetos tipo dato y que filtran la información para transformarla en otro tipo. Las implementaciones de algoritmos como decimación y *marching cubes* son algunos de sus miembros.

Módulo Gráfico

Compuesto por las clases que permiten tomar la información encapsulada en estructuras de datos, crear una escena y desplegarla. Las principales clases de este módulo son:

- **vtkProp, vtkActor2D, vtkActor**: Definen los objetos que se verán en la escena, de manera que a cada elemento visible le corresponde un actor. La primera es la que define las propiedades elementales de todos los objetos a desplegar, para 2D se tiene vtkActor2D y en el caso tridimensional vtkActor, ambos heredan muchas de sus características de vtkProp. vtkActor2D son todos aquellos objetos que representan datos en dos dimensiones, por ejemplo datos de imagen, mientras que vtkActor define el resto de los objetos como volúmenes (vtkVolume es la que realmente los maneja, pero hereda mucho de vtkActor) los que emplean una matriz de 4x4 para definir su posición, orientación y escalamiento en la escena.

- **vtkLight**: En una escena tridimensional vtkLight define y controla la iluminación, cantidad de fuentes, color, etc. Para el caso bidimensional no se requieren luces.

- **vtkCamera**: Define los objetos de tipo cámara, encargados de manipular la proyección de la geometría tridimensional en el plano 2D. Posee varios métodos para orientarla hacia el objetivo, además de soporte para proyección en perspectiva y visualización estereoscópica pasiva. Innecesaria para el caso 2D.

- **vtkProperty:** Define las apariencia de los actores en escena, como: color, efectos de luz ambiental, *rasterización* (*wireframe* o sólida), etc.

- **vtkTransform:** Define las transformaciones lineales de los objetos en forma de matrices de 4x4 asociadas a los actores.

- **vtkLookupTable, vtkColorTransferFunction:** Proveen métodos para colorear objetos de acuerdo a los valores escalares asociados a los datos. La diferencia es que la segunda construye una función definiendo algunos de sus puntos, de

manera que los valores que correspondan a puntos que no fueron explícitamente insertados al momento de la construcción, toman un color determinado implícitamente por cómo se comporta la función en ese punto.

- **vtkMapper:** Los actores no pueden representar su propia geometría, así que recurren a los objetos de este tipo que son los encargados de encapsular la geometría y de proveer métodos para desplegarla. Actúan como mediadores entre el módulo de visualización y el gráfico.

- **vtkRenderer, vtkRenderWindow, vtkRenderWindowInteractor:** Los dos primeros sirven de intermediarios entre el módulo gráfico y el sistema operativo. El primero provee métodos para transformar geometría, luces y ángulo de cámara en una imagen. vtkRenderWindow provee un ambiente para que el anterior realice su tarea de dibujado, mientras que el último tipo de objetos son aquellos que manipularán las interacciones con el usuario, permitiendo por ejemplo rotar la cámara cuando se mantiene presionado el botón izquierdo del ratón y se mueve horizontalmente.

Este par de módulos actúan juntos para formar lo que se conoce como el *pipeline* de visualización, con un trabajo parecido a una línea de ensamblaje, en donde a lo largo del flujo se encuentran los filtros de transformación. Para demostrar el poder y facilidad de VTK en cuestiones de visualización, se presenta un ejemplo:
Se desea realizar una triangulación de Delaunay sobre un conjunto de puntos en el plano, es necesario:

1. Leer los datos, construyendo puntos a partir de las coordenadas leídas y guardándolos en una estructura como vtkPoints.

2. Se procede a alimentar a un filtro vtkPolyData el conjunto de puntos, transformándolos en el tipo de datos que vtkDelaunay2D, la implementación de la triangulación de Delaunay bidimensional en VTK, requiere para trabajar.

3. Alimentar a la instancia de vtkDelaunay2D el conjunto de datos apuntado por vtkPolyData

4. Declarar un *mapper* genérico: vtkDataSetMapper y asignarle como entrada la salida de la triangulación.

5. Declarar un actor (vtkActor si se desea posicionar el objeto en el espacio, o vtkActor2D en caso de que sólo se desee el plano) para encapsular a la triangulación, asignándole como *mapper* el que se instanció anteriormente.

6. Crear un vtkRenderer, un vtkRenderWindow que apunte hacia él y tal vez un vtkRenderWindowInteractor para manipular eventos del ratón (para la mayoría de las visualizaciones con la cámara y luces creadas por defecto por vtkRenderWindow basta, así que normalmente no es necesario hacer mayores manipulaciones en ese sentido)

7. Comenzar la visualización, llamando al método start() con el objeto vtkRenderer

Para realizar todo el procedimiento desde cero no sólo habría que definir las estructuras de datos pertinentes, también implementar el algoritmo de triangulación de Delaunay, encontrar la manera de desplegar la imagen y manejar los eventos con el usuario, todo esto además dependería del sistema operativo donde se esté trabajando. Así es fácil ver por qué se empleó VTK para realizar el sistema.

A continuación se presenta una descripción del programa resultante de la primera aproximación.

4.2.2. vtkNebula

VTK Provee funciones para aplicar volume rendering por software a muchas estructuras de datos. Para vtkNebula se requería que lo pudiera hacer sobre una malla regular estructurada, ya que los datos vienen en esa forma. Dicha clase no existe directamente,

pero como una malla de este tipo es un caso particular de una malla no estructurada, entonces vtkUnstructuredGrid (la estructura de datos) y vtkUnstructuredGridHomogeneousRayIntegrator (la clase que define el lanzador de rayos) son suficientes.

Lo que en volume rendering corresponde a las unidades de volumen (voxeles), en vtkNebula serán las celdas tridiemsnionales de la malla, las cuales en VTK se definen por sus vértices y su valor escalar asociado, así que cada elemento de la malla debía contener las posiciones de sus 8 vértices y dentro, como datos de celda (vtkCellData), el valor de intensidad correspondiente.

El primer paso es leer los datos, así que se crean tres vectores: r, g, b que contendrán la información de cada una de las tres capas de datos, y uno más: *ceros*, que indicará en qué posiciones el valor de intensidad es menor a un valor de corte definido por el usuario, empleando un 1 si es inferior o 0 en otro caso.

Acto seguido se realiza la construcción de la malla:

Primero se calculan las dimensiones que tendrá y el número de puntos que la formarán, usando el último índice antes del fin del archivo. Posteriormente los puntos se almacenan en una estructura vtkPoints, primero creciendo en X, luego Y, luego Z. Se crean las etiquetas de las celdas y se les nombra.

Ciclando sobre el número de puntos y usando tres enteros para asegurar que el ciclo no construya celdas con los puntos de los extremos, se procede a insertar celdas a la malla: se reetiqueta un vtkVoxel temporal para que en cada iteración posea los índices correspondientes a la celda actual, si ésta no es nula, es decir: si su índice en el vector *ceros* indica que es válida entonces el voxel se agregará a la malla, en caso contrario se descartará. Al final del procedimiento se le asigna a la malla los puntos creados inicialmente.

Normalmente para colorear cuando se usa volume rendering se crea una función de transferencia y se agregan puntos a la misma, cada uno correspondido por un valor RGBA. Para el funcionamiento del sistema fue necesario desviarse del método usual: en

vez de emplear una función de transferencia o *lookup table* y saturar la función con un punto por celda, se optó por usar un componente escalar de cuatro entradas (RGBA) por voxel, asociando a cada etiqueta una tupla. Previo a insertar los valores de color se busca el escalar máximo de cada canal, de manera que el n-ésimo voxel se colorea dividiendo cada una de las tres entradas n-ésimas de intensidad entre el mayor de cada capa, normalizando la intensidad en el intervalo [0,1]. La transparencia de cada voxel será la misma (definida por la variable *alpha*) la que varía normalmente en proporción inversa a la densidad del volumen: si el valor alpha es muy grande, en volúmenes densos se saturará muy rápido, en cambio si alpha es pequeña para densidades bajas el volumen se verá opaco.

Todo lo anterior se resume a asociar a cada celda con una tupla:

```
1  //Se insertan los valores según el índice
2  for(i=0; i < no_voxels; i++)
3          cellsID->InsertNextTuple4(r.at(i)/rmax,
4                                    g.at(i)/gmax,
5                                    b.at(i)/bmax,
6                                    alpha);
```

Lo que resta es visualizar la malla: para esto se declara una ventana, un actor del tipo vtkVolume que la represente, un *mapper* (vtkUnstructuredGridRayCastMapper) y un filtro que triangule las celdas (vtkDataSetTriangleFilter), debido a que el *mapper* requiere como entrada celdas triangulares. Además se emplea vtkVolumeProperty para asignarle propiedades al volumen que encapsula a la malla y así indicarle a VTK que no busque una función de transferencia como normalmente lo haría cuando emplea volume rendering, sino que use un componente escalar de cuatro valores para colorear cada voxel individualmente. Si se desea conocer más a fondo el procedimiento, se recomienda revisar el código localizado en: *http* : *//glnebula.sf.net/vtkNebula.cpp*.

Desempeño

Volume Rendering es una técnica computacionalmente intensiva, más aún: VTK no emplea funciones directamente de la tarjeta gráfica, por lo que el desempeño no es el mejor que podría ser.

Se probaron modelos de 50, 100 y 150 de lado, con resolución de 400x400 pixeles en la misma máquina:

- Procesador: Pentium D 805 (Smithfield) 2.66GHz
- Memoria: 1Gb DDR2 RAM
- Tarjeta de video: Nvidia GeForce 7600GT 256MB
- Sistema Operativo: Ubuntu Linux 6.06
- VTK: 5.0

Dimensiones	**Valor de Corte**	**Voxeles Visibles**	**Cuadros por segundo promedio**
50x50x50	1e-20	11024	12
100x100x100	1e-21	33056	$< 1{,}0$
150x150x150	1e-25	668984	$< 1{,}0$

Cuadro 4.1: Desempeño de vtkNebula

Cabe aclarar que VTK reduce el número de rayos trazados cuando el usuario interactúa con la ventana, así que si se está rotando la nebulosa se mostrará un modelo de menor realismo hasta que cesen los eventos de ratón. En cuanto terminan se recalcula empleando todos los rayos, lo que toma algo de tiempo.

Problemas

Además que el desempeño no era del todo aceptable, surgieron un par de problemas al emplear *volume rendering*. En las versiones iniciales de vtkNebula se obtenía un

coloreado sumamente extraño: parecía que muchos colores se repetían y en general los voxeles no eran pintados adecuadamente. Para descartar problemas particulares a la implementación usada se realizaron pruebas con cubos coloreados aleatoriamente, con los mismos resultados. También se probó colorear alternadamente las celdas nones y pares con verde y rojo respectivamente: cuando el número de puntos crecía por ejemplo de 125 (5x5x5) a 6859 (19x19x19) se perdía el patrón de tablero de ajedrez, repitiéndose el mismo color a lo largo de 6 o 7 celdas.

En este momento se sospechó que el límite de puntos que una función de transferencia puede tener, es mucho menor al número de voxeles empleados. Se realizaron varias preguntas a la lista de correo[3] y así se descubrió que, en efecto, el límite era algo reducido: cuando se emplea una función de transferencia el encargado de hacer el trazado de rayos es vtkUnstructuredGridHomogeneousRayIntegrator, el cual en su constructor define como 1024 el máximo número de puntos que la función puede tener (mediante TransferFunctionTableSize = 1024). De esta manera al tener tres componentes de color, vtkNebula estaba restringido a 1024/3 = 341 entradas de color para la función, muy por debajo del número de voxeles necesario. Desgraciadamente esto no estaba debidamente documentado y encontrar el problema tomó mucho tiempo, solucionándose con la respuesta de uno de los desarrolladores de VTK (Randall Hand), prometiendo actualizar la documentación sobre el tema. En este punto había dos caminos a tomar: aumentar el tamaño de la función de transferencia o desecharla y en cambio utilizar cuatro componentes escalares por voxel. La primera solución presentaba el problema de que al incrementar la función de transferencia para que correspondiera al número de voxeles, la memoria iba a crecer desmesuradamente, teniendo un impacto negativo considerable en el desempeño que era de por sí pobre. Por ello se optó colorear directamente los voxeles sin usar vtkColorTransferFunction, resolviendo así el problema de coloreado y manteniendo el uso de memoria en niveles aceptables.

[3] *vtkusers@vtk.org*

Además de ésto se tiene un problema inherente a *ray casting*: los parámetros como transparencia y saturación de canales[4] no se pueden actualizar en tiempo real, debido a que se tienen que reinsertar los nuevos valores a cellsID, por lo que se pierde bastante tiempo cuando se trata de refinar la visualización.

4.2.3. Conclusiones

El desarrollo del sistema no presentó mayor reto intelectual que la construcción de una malla de tamaño arbitrario, el problema radica en que aprender a usar VTK no es sencillo, como se ha insistido previamente su documentación está sumamente incompleta y enfocada principalmente a los usuarios experimentados de la biblioteca. Una vez que se aprende a usar, VTK puede ser una herramienta muy poderosa: vtkNebula tiene alrededor de 400 líneas líneas de código, las cuales son bastante sencillas de entender debido a la naturaleza explícita de los nombres de las clases y a la manera de ensamblar el *pipeline* empleando filtros, uno tras otro; la dificultad es acostumbrarse a su uso. El desarrollo de vtkNebula hubiera llevado una fracción del tiempo total dedicado si se hubiera tenido mayor experiencia previa con la herramienta y si se tuviera una documentación apropiada.

El problema real lo presentaban *volume rendering* y *ray casting* en sí: el desempeño dejó mucho de qué desear, la interacción sólo era posible con los cubos de menor resolución (50x50x50) y si se desea llevar a cabo la presentación en la sala Ixtli, en donde se requiere objetos con mucho mayor resolución debido al tamaño de la pantalla, la aproximación no prometía mucho.

En retrospectiva es fácil ver que volume rendering no es la mejor aproximación cuando se requiere modelar objetos de tamaño considerable y mantener interactividad

[4]Tres escalares que multiplican los valores de color de cada canal, para resaltar o atenuar a placer las capas

en tiempo real con el usuario, es una técnica muy útil cuando el nivel de realismo requerido opaca la necesidad de obtener visualizaciones instantáneas, como en el caso de imágenes médicas.

Recapacitando se pensó una segunda aproximación, que por diseño era mucho más rápida que volume rendering y, hasta cierto punto, se creía capaz de dar la apariencia de nube y de usar cubos de mayor resolución.

4.3. Segunda aproximación: Billboarding

Con la experiencia adquirida en la implementación anterior y considerando que los modelos más pequeños a emplear tendrán 125 mil celdas visibles en el peor caso, se comenzó a idear un camino alterno que solucionara el problema de desempeño: Si se va a tener un único escalar por unidad de volumen (celda), entonces tal vez no haya necesidad de construir un volumen propiamente, cuando con piezas bidimensionales semi opacas se puede crear el mismo efecto, simplemente orientándolas siempre hacia el observador de manera que aparenten grosor y haciendo que mezclen su color con el de sus vecinas de atrás y adelante. De esta manera, reduciendo el número de caras de seis a una por cada celda y el número de vértices de 8 a 4, se esperaba tener una mejora considerable en el desempeño sin perder la apariencia de ser una nube volumétrica.

La técnica es conocida como *billboarding* y es muy usada en sistemas de partículas y en general en aplicaciones donde se requiera disminuir el número de polígonos a usar o tener objetos orientados hacia un lugar en particular.

Un *billboard* es una primitiva geométrica plana (usualmente cuadrada) orientada de manera perpendicular con respecto a cierto punto de referencia (en muchos casos respecto a las coordenadas de la cámara). Normalmente van acompañados de textura para maquillar la sencillez de las primitivas y dar la apariencia de ser objetos sólidos.

Existen dos formas de orientar las primitivas: con respecto a un punto y a un eje. La primera centra cada partícula en un punto de referencia sobre el que se realizarán

las rotaciones para preservar la orientación deseada, mientras que la segunda define un eje local de la primitiva para ser orientado con respecto a un eje global, realizando las rotaciones sólo con respecto al elegido (de manera que el *billboarding* funciona de forma cilíndrica, desapareciendo el efecto si por ejemplo se orienta con respecto al eje Y y la cámara mira al objeto desde arriba), aunque puede funcionar si se rotan también los ejes locales restantes (trabajando como una esfera, así manteniendo el efecto desde cualquier punto de vista). Además la orientación puede ser realizada colectiva o individualmente. La orientación colectiva es muy veloz ya que sólo requiere una única matriz de rotación para todos los puntos, construida calculando la dirección opuesta a la que está viendo la cámara (es decir: el *right vector*[5] de la cámara y de los *billboards* son contrarios). El problema de usar esta aproximación aparece cuando se tiene una cámara con apertura de campo muy grande, de manera que los elementos cercanos a las orillas se orientan erróneamente. La orientación individual es un poco más compleja, requiriendo calcular primero el vector de visión de la partícula (*look vector*) restando las coordenadas de ésta a las de la cámara. Acto seguido se calcula el *right vector* del *billboard* calculando el producto cartesiano del vector de visión de éste y el *up vector*[6] de la cámara, para al final calcular el *up vector* de la primitiva aplicando el producto cruz de sus vectores *right* y *look*. Este acercamiento tiene la desventaja de ser ligeramente más lento que el anterior ya que se deben realizar todos estos cálculos para cada una de las partículas a orientar, pero por lo mismo es más preciso y no presenta el mismo problema que la orientación colectiva.

Para llevar a cabo la implementación no hubo que buscar demasiado, la biblioteca OpenGL fue inmediatamente considerada para el desarrollo.

[5]Vector que indica hacia dónde queda la derecha de la partícula

[6]Vector que apunta en la dirección “arriba”

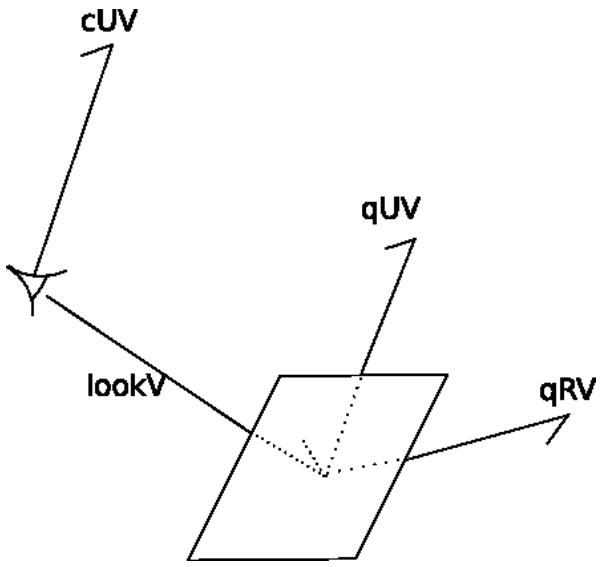

Figura 4.1: Orientación individual: se mantiene el *right vector* de la partícula (qRV) perpendicular al vector de visión de la cámara (lookV) y el *up vector* de la primera (qUV) paralelo al correspondiente de la última (cUV)

4.3.1. Open Graphics Library

OpenGL[7] es una interfase de programación de aplicaciones (API por sus siglas en inglés) creada por SGI[8] en 1992. En un principio estaba enfocado a ser un sistema propietario que funcionara con otros APIs gráficos de la compañía, pero más tarde, en colaboración con algunos fabricantes de dispositivos, decidió abrirse. La biblioteca pretende servir de intermediaria entre el hardware gráfico y un lenguaje de programación, manteniendo su nivel de funcionamiento lo suficientemente bajo para lograr compatibilidad con cualquier dispositivo, independientemente de sus cualidades y manufacturador. Se creó un consorcio independiente llamado OpenGL Architecture Review Board, encargado de la certificación de compatibilidad y la aprobación de nuevas extensiones a la plataforma. Además OpenGL es multiplataforma y cuenta con una vasta documentación de sus características, lo que aunado a su poder, extensibilidad y escalabilidad

[7]http://www.opengl.org
[8]http://www.sgi.com/

(pudiendo ser usado desde computadoras de escritorio hasta máquinas de muy alto rendimiento) lo convierten en el estándar para el desarrollo de aplicaciones en el campo de la graficación por computadora.

OpenGL funciona como una máquina de estados, comportándose de acuerdo a los cambios en los mismos y manteniéndolos a lo largo del *pipeline*, así según el estado en el que se encuentre, realizará operaciones adicionales (o dejará de hacerlas). Para habilitar estados normalmente se emplea el comando $glEnable(estado)$ y su contraparte $glDisable$, además es posible pedirle a OpenGL que indique si un estado en particular está activo haciendo $glGet(estado)$.

El *pipeline* de OpenGL está compuesto por 4 pasos principales:

El primero es el evaluador, donde se toman los comandos de polinomios y se traducen a una serie de comandos de vértices y atributos. Posteriormente viene la etapa de operaciones de vértices y ensamblado de primitivas, realizando trabajos como: transformaciones, cálculo de iluminación, *clipping*[9], cálculo de proyección, etc. Continúa con la *rasterización*, traduciendo las primitivas a fragmentos en el *frame buffer*[10]. La cuarta etapa consiste de las operaciones de fragmentos: antes de ser enviados al *frame buffer* pueden ser sometidos a pruebas de profundidad, cálculo de transparencia, etc. El *pipeline* se puede retroalimentar con la salida de objetos contenidos en el *color buffer*[11], organizados en forma de rectángulos de pixeles y, si el mapeo de texturas está habilitado, se emplea memoria de textura para modificar el color de los pixeles en el paso de rasterización.

[9]Operación que calcula la porción visible de cada elemento en la escena, para ignorar los cálculos de aquellas primitivas que no serán observadas

[10]Locación en memoria donde se almacena información como: localización de pixeles, color, coordenadas de textura, profundidad, etc.

[11]Lugar de la memoria en donde se almacena el color de cada pixel

4.3.2. GLUT

Por diseño OpenGL no provee funciones para manejar ventanas ni para responder a eventos de usuario, esto con el fin de no intervenir con la idea de independencia de plataforma al estarse preocupando por los detalles individuales de los sistemas operativos. El problema es que las aplicaciones gráficas deben estar acompañadas de al menos una ventana para desplegarse, con esto en mente se desarrolló el OpenGL Utility Toolkit, GLUT: un conjunto de utilidades que permiten realizar operaciones de creación y manejo de ventanas, trabajar con eventos de usuario y dibujar primitivas que OpenGL no posee (esferas y cubos sólidos o en *wireframe*).

Para su funcionamiento GLUT requiere que se definan funciones de llamada para entregarles el control cuando se detecta un evento, esto se realiza registrando las funciones dependiendo del tipo de eventos que manejen: por ejemplo si se quiere registrar una función para atrapar los eventos de teclado y modificar la escena dependiendo de la tecla presionada, se usará *glutKeyboardFunc(funcTeclado)* para indicarle que *funcTeclado()* resolverá los eventos. Sólo existe una condición: que la función de *callback* reciba los parámetros que le regrese la de GLUT, en este caso la tecla presionada, y que le indique si fue usado algún modificador (shift, control, etc.).

Tal vez el *callback* más importante a registrar es el de despliegue: glutDisplayFunc(), con esto se le indica a GLUT qué función será la encargada de redibujar la pantalla cuando se considere necesario, es en la función a llamar donde se encuentra la escena definida con el código OpenGL. Después de registrar las funciones manejadoras de eventos es necesario llamar a *glutMainLoop()*, la cual mostrará todas las ventanas creadas y comenzará a procesar los eventos que aparezcan en las mismas. Este ciclo continuará hasta que la aplicación sea terminada.
GLUT fue empleado para desplegar la ventana y manejar los eventos de usuario en las versiones iniciales de GLNebula.

4.3.3. GLNebula

Para llevar a cabo el *billboarding* se desarrolló una clase auxiliar Vector3 que definiera vectores tridimensionales y algunas operaciones a realizar entre ellos, como: suma, multiplicación por escalar, verificación de igualdad, etc. De manera que la manipulación y almacenamiento de los vectores requeridos para orientar las partículas se haga más sencillo.

Además se construyó una clase para el manejo de la cámara, permitiendo la creación de un objeto de este tipo indicando sus coordenadas de origen, de objetivo y su *up vector*. Así se pudieron asignar eventos del ratón a rotaciones y acercamientos del punto de vista, cumpliendo con un objetivo más.

El *pipeline* de GLNebula es muy parecido al de su antecesor, leyendo y guardando los datos en cuatro vectores, tres almacenando los datos de las tres capas y uno que indica las celdas nulas, de nuevo empleando un criterio de corte. Acto seguido se encuentran los valores mínimos, máximos y promedio de cada canal, almacenándolos en 9 variables (tres tipos de valores por cada canal). Los valores máximos son los únicos que se usarán después al momento del coloreado, los otros dos le servirán al usuario como guía para localizar posibles valores de corte.

Posteriormente se crea la malla, sólo que esta vez se definen los cuatro vértices de cada celda (hay que recordar que se usan objetos bidimensionales, por lo que en vez de ocho se tienen cuatro puntos) empleando los cálculos obtenidos para lograr la orientación deseada. Se colorea cada celda, se le asigna transparencia y si se desea se le aplica textura, para al final mostrar la malla construida (ver figura 4.3.5).

A continuación se explica con más detalle los pasos involucrados en la obtención de un modelo que difieren de la implementación de vtkNebula.

4.3.4. Billboards

Para orientar las partículas es necesario determinar sus vectores *up* y *right*. OpenGL emplea matrices de 4x4 entradas para manejar las transformaciones y posicionamiento

de los objetos en la escena: una para la proyección de cámara, otra para los modelos y una más para las texturas. GLNebula no emplea los métodos mencionados anteriormente para hacer *billboarding*, en vez de eso los vectores de orientación se obtienen directamente de la matriz de modelo.

La dirección en la que las partículas deben mirar es opuesta al vector de visión de la cámara, así que habría que calcular la matriz inversa de la de modelo. Afortunadamente, si ésta no ha sufrido cambos drásticos, es una matriz ortogonal, es decir: su inversa es igual a su transpuesta. Así sólo es necesario copiar las entradas de orientación de la matriz transpuesta a dos objetos tipo Vector3 y usar esa información para definir los cuatro vértices del GL_QUAD (primitiva de OpenGL que representa un cuadrado o rectángulo) correspondiente, centrándolo en los puntos que indican los datos.
En resumen, si la matriz de modelo es la siguiente:

$$\begin{pmatrix} r_1 & u_1 & l_1 & p_x \\ r_2 & u_2 & l_2 & p_y \\ r_3 & u_3 & l_3 & p_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

su transpuesta es:

$$\begin{pmatrix} r_1 & r_2 & r_3 & 0 \\ u_1 & u_2 & u_3 & 0 \\ l_1 & l_2 & l_3 & 0 \\ p_x & p_y & p_z & 1 \end{pmatrix}$$

que al ser idéntica a la matriz inversa, se ahorra el cálculo de la misma y se proceden a tomar los vectores necesarios de las entradas correspondientes en la matriz transpuesta. Es decir, lo que en la matriz de modelo corresponde a las entradas: $m[1,1], m[2,1], m[3,1]$ (*right vector* de la cámara) y $m[1,2], m[2,2]$ y $m[3,2]$ (*up vector*); en su inversa corresponden a las entradas de la transpuesta: $m[1,1], m[1,2], m[1,3]$ y $m[2,1], m[2,2], m[2,3]$ (vectores *up* y *right* del *billboard*).

GLNebula emplea *billboarding* esférico, de manera que los objetos pudedan ser vistos desde cualquier lugar sin perder la sensación de representar un volumen.

Blending

Cada partícula tiene un color particular, determinado por su índice propio y la información que los datos tengan del mismo, dividiendo de nuevo entre el máximo de cada canal para normalizar. A las celdas se les asigna una transparencia de manera que se pueda mezclar su color con el de los vecinos. OpenGL requiere de un cambio de estado para habilitar la funcionalidad de transparencia, haciendo *glEnable(GL_BLEND)*, de esta manera se realizará un mezclado del color acumulado en el *frame buffer* con el que va entrando. La manera en que OpenGL decide esto se basa en la siguiente ecuación:

$$Cf = (Ce * FBe) + (CFB * FBd)$$

Donde *Ce* es el color actual, *CFB* es el color almacenado en el *frame buffer*, *Cf* es el color que tendrá el pixel en cuestión después de hacer el *blending* y *FBe* y *FBd* son dos factores de atenuación de color. Cada variable de la ecuación anterior es realmente un vector de 4 entradas, representando los cuatro canales: RGB y Alpha. Los factores de atenuación se le indican a OpenGL usando *glBlendFunc*, que recibe dos argumentos, el primero controla el factor de entrada y el segundo el destino. Pueden ser: GL_ZERO, GL_ONE, GL_SRC_ALPHA, GL_DST_COLOR, GL_ONE_MINUS , entre otros. Para GLNebula se usó GL_SRC_ALPHA y GL_ONE, de manera que sólo se toma en cuenta el canal alpha del color entrante y le suma el color acumulado.

Una vez que las celdas están coloreadas se les aplica (opcionalmente) textura.

Texturas

Una textura es una imagen bidimensional que será sobrepuesta a una primitiva. Las texturas no son una parte vital del funcionamiento de GLNebula, pero permiten la creación de modelos con una apariencia tal que se asemeje más a sus contrapartes reales. Además, debido a que se están usando cuadrados, las esquinas de los mismos suelen formar patrones indeseables que se pueden atenuar mediante la aplicación de texturas, así mismo la transparencia puede ser controlada usando imágenes en blanco y negro y mapeando los gradientes de gris a distintas opacidades.

Así después de que las celdas han obtenido color y transparencia, se cargan las texturas y se especifican sus cuatro coordenadas. Para cargar las imágenes en texturas GLNebula emplea Simple Directmedia Layer (SDL) y SDL_Image.

Al final de la construcción de la malla sólo resta desplegarla, en este punto del desarrollo se usó GLUT.

4.3.5. Point Sprites

Se implementó otro tipo de *billboarding* de manera que se tuviera una opción más para dibujar las partículas: *point sprites*. Este tipo de *billboards* se distingue de los *quads* orientados en dos formas: se orientan y definen con respecto a un punto (así que no hay que trabajar con la matriz de modelo) y, en OpenGL, son una extensión que emplea hardware puramente (lo que los hace más rápidos que los *billboards* tradicionales). Para definirlos sólo hay que cambiar el estado de OpenGL para que use la extensión, haciendo *glEnable(GL_POINT_SPRITE_ARB)* y luego especificando el vértice central con *glVertex3f(x, y, z)*.

Otra ventaja de los *sprites* es que pueden ser utilizados en conjunto con un método de dibujado que acelera tremendamente el desempeño, presentado a continuación.

Display Lists

GLNebula satisfacía casi todas las necesidades planteadas inicialmente pero si se planeaba trabajar con mallas de mucho mayor resolución se tenía que realizar optimizaciones (ver 4.3.7 Desempeño), entonces se pensó en el uso de listas de despliegue. Una *display list* es un conjunto de comandos de OpenGL que se guardan con el fin de ser ejecutados posteriormente. La misma idea puede ser aplicada a comandos repetitivos como dibujar los mismos objetos en distintas posiciones, así se puede guardar la creación de la geometría en una lista y cada vez que se requiera redibujar, cambiar las coordenadas y llamar a la lista que contiene los comandos, haciendo muy eficiente el proceso. Esto encajaba perfectamente con la forma de trabajar de GLNebula, así que se decidió meter toda la construcción de la malla en una lista, de manera que cada vez que se necesitaba redibujar sólo había que llamarla. Así la creación de la geometría, la coloración, transparencia y la aplicación de textura a las celdas se hace sólo una vez, se introduce a una lista y posteriormente se ejecuta llamándola. El único momento en el que se tiene que recrear la malla es cuando alguno de los comandos dentro de la lista cambia, como el tamaño de las partículas, la saturación de color o la transparencia de las celdas.

Para crear una *display list* primero se le asigna espacio en memoria, haciendo $glGenLists()$ y dándole como argumento el número de listas a crear (en este caso una), la función regresa el indicador de la lista en forma de un GLuint. Posteriormente se comienza a definir haciendo $glNewList()$ y especificando que guarde los comandos en la localidad de memoria apuntada por el indicador de la lista que se creó previamente. Además la función requiere un modo de operación, que puede ser: GL_COMPILE para sólo crearla sin ejecutar los comandos hasta ser llamada, o GL_COMPILE_AND_EXECUTE que crea la lista y ejecuta los comandos inmediatamente. GLNebula usa el primero.

Ahora es cuando se introducen todos los comandos que se quieren repetir, para después finalizar la lista con $glEndList()$ y llamándola con $glCallList(list)$ en la función

de dibujado.

Se intentó usar *display lists* para acelerar el desempeño de los quads orientados, pero desgraciadamente no fue posible ya que es necesario recalcular manualmente la orientación en cada cuadro, a diferencia de los *sprites* en donde OpenGL se encarga de la orientación.

4.3.6. Interfaz gráfica

Para facilitar el uso de GLNebula se decidió implementar una interfaz gráfica que permitiera además mayor control sobre los parámetros de la visualización.

Desafortunadamente la funcionalidad de GLUT está limitada a la creación de aplicaciones sencillas, sin soporte para cuestiones de interacción con el usuario como menús, botones, etc. por lo que el desarrollo de extensiones que permitan programas más sofisticados es natural, una de ellas se presenta a continuación.

GLUI

Debido al tiempo dedicado a VTK y a la necesidad de tener una herramienta funcional lo más pronto posible, se optó por emplear la biblioteca GLUI[12], que aunque sencilla, proveía los widgets y controles suficientes para manejar GLNebula, además por su ligereza no tiene un efecto negativo considerable en el desempeño. GLUI es una biblioteca escrita en C++ y basada en GLUT (por ende multiplataforma) para el desarrollo de aplicaciones gráficas sencillas. Provee varios tipos de controles con la posibilidad de definir funciones de respuesta (*callbacks*) que van a ser llamadas cuando el usuario interactúe con ellos, también permite el uso de variables que se actualizan al momento de modificar los controles (*live variables*). Éstos son sumamente sencillos de agregar y se autoarreglan en la ventana conforme van siendo instanciados, tomando como referencia una malla de coordenadas con los controles haciendo las veces de celdas.

[12]http://www.cs.unc.edu/ rademach/glui/

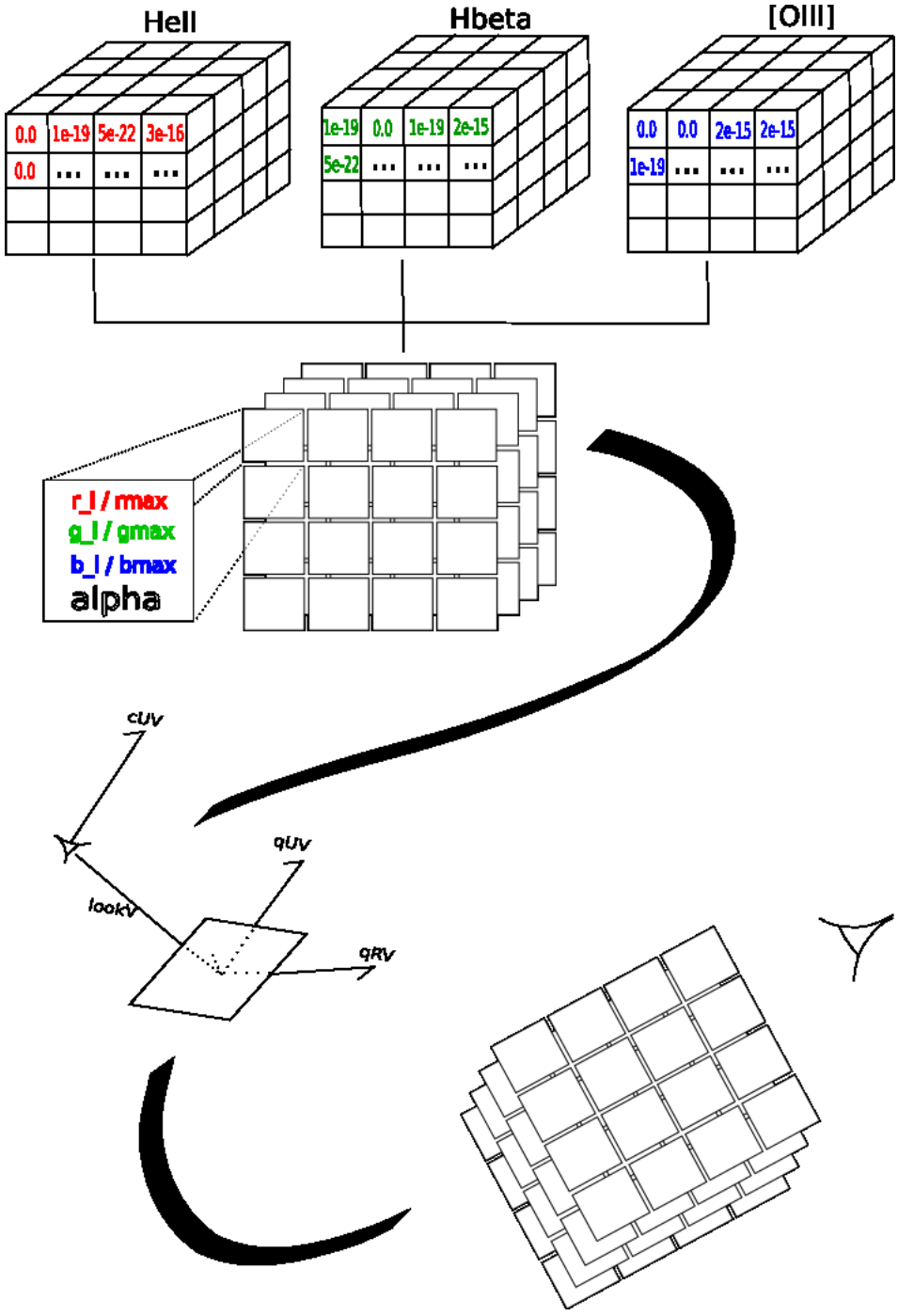

Figura 4.2: Procedimiento de creación de los modelos, de arriba a abajo: Se toman los tres datos, a cada uno se le asigna un canal de color. Se construye una malla de elementos planos (*billboards* o *sprites*), se colorea cada celda combinando el color de cada canal normalizando (dividiendo entre el máximo de cada color) y se le asigna transparencia. Se orientan las celdas hacia la cámara.

.

Convenientemente GLUI subordina el ciclo principal de GLUT al propio, de manera que la ventana que despliega la nebulosa depende de la de controles y así no hubo que realizar otra modificación que cambiar el orden de llamada a GLUT y registrarlo con GLUI, lo mismo para las funciones de teclado y ratón.

Se presentaron un par de problemas por el uso de esta biblioteca, en principio no fue posible reducir el uso de procesador al cesar la interacción: en la versión anterior es posible dormir el proceso si no se registra ningún evento de usuario (o bien si se presiona la barra espaciadora), desafortunadamente por la manera de trabajar de GLUI y su necesidad de constante refresco para atrapar cualquier cambio en sus widgets, no fue posible implementarlo, de esta manera el programa está usando recursos del CPU aún en la inactividad. Otra desventaja es que no provee widgets de alto nivel, concretamente diálogos para la apertura de archivos (*File Choosers*), esto es un inconveniente ya que hay que introducir los nombres de los archivos a mano, aunque la molestia se reduce con la posibilidad de guardar y cargar sesiones, es decir: los parámetros que proporcionan cierta visualización pueden ser guardados para su uso posterior, así si se desea replicar una visualización particular sólo es necesario cargar la sesión pertinente.

El constante uso de GLNebula dejó ver que si se empleara diálogos de carga de sesión la usabilidad mejoraría bastante, por ello se buscaron alternativas a GLUI y se encontró un candidato, presentado a continuación.

Fltk

Fast Light Toolkit[13] es un sistema para el desarrollo de interfases gráficas, es multiplataforma y está escrito en C++. Contiene además de los *widgets* usuales como botones y cajas de texto, controles de muy alto nivel como *file choosers* .

La idea es que cada control tenga su propia función de regreso a la que llamará al ser activado (entre otros eventos) así que no se tuvo que realizar ningún cambio fundamental a la estructura de GLNebula y solamente se borró cualquier referencia a GLUT/GLUI y

[13] http://www.fltk.org

se ajustaron las funciones de callback que usaba GLUI para ahora usar FLTK. Aunque FLTK tiene compatibilidad parcial con GLUT, el manejo de los eventos de usuario fue reescrito debido a que no se sabía hasta qué punto la falta de un soporte completo iba a afectar a GLNebula. La conversión llevó poco tiempo y probó ser muy útil, además siendo fiel a su nombre FLTK no impactó de manera notoria en el desempeño, que era uno de los temores antes de la migración.

Normalmente al programar interfases gráficas se dedica la mayor parte del tiempo a agregar los controles y posicionarlos sobre la ventana, afortunadamente existe Fast Light User Interface Designer, FLUID. FLUID es un programa que permite diseñar la interfase escogiendo los controles de una lista y agregarlos arrastrándolos hacia la ventana que los contendrá, también permite definir las funciones de *callback* particulares a cada control y escribir código relacionado con la interfase. Cuando el usuario está satisfecho con los cambios simplemente se le indica a FLUID que escriba el código y se generarán los fuentes de C++ apropiados. El uso de FLUID ahorró mucho tiempo que se enfocó a conectar el motor existente de GLNebula con los controles diseñados.

El cambio fue relativamente rápido y se reflejó en mayor rapidez y facilidad de uso, características que se desean en cualquier sistema.

4.3.7. Desempeño

Probando GLNebula en la misma máquina que vtkNebula, a la misma resolución (400x400 pixeles) y con la misma textura, se obtiene un desempeño mucho mejor que con su antecesor. De la misma manera usando *point sprites* se obtuvieron resultados aún mejores (ver el cuadro comparativo 4.2).

En ambos casos las mejoras son considerables y gracias al uso de *display lists* se espera que GLNebula pueda trabajar con modelos de mayor tamaño.

Dimensiones	Corte	Voxeles Visibles	Quads: Textura/Sin	Sprites:Textura/Sin
50x50x50	1e-20	11024	177/175	250/252
100x100x100	1e-21	33056	12/11	256.74[a]/286.71
70x70x70	1e-09	221808	10.61/10.49	65/67
150x150x150	1e-25	668984	3.95/3.96	50.15/50.6

Cuadro 4.2: Desempeño de GLNebula en cuadros por segundo promedio usando quads orientados y display lists con point sprites sin textura, la distancia de la cámara al centro del modelo (parámetro muy influyente en el desempeño está dada por: $sqrt((dimx/2)^2 + (dimy/2)^2 + (dimz/2)^2)$.

[a]Debido a la textura usada, a la poca densidad de este modelo en particular y a que el tamaño máximo de los puntos en la tarjeta usada es de 64, muchas de las partículas desaparecen completamente, por lo que los resultados son engañosos

4.3.8. Características

Gracias a su desempeño la versión actual de GLNebula permite la visualización en tiempo real de nebulosas planetarias a partir de intensidades de emisión, permitiendo observar el cuerpo desde cualquier punto de vista, cumpliendo con el objetivo principal. Además posee características extras que mejoran su funcionalidad. La versión actual tiene las siguientes:

- **Control de parámetros:** Es posible controlar el nivel de saturación de cada canal, resaltando o disminuyendo la presencia de cierto ion. Además permite controlar la transparencia y tamaño de las partículas, así como una función sencilla de submuestreo que dibuja las celdas de acuerdo a un índice correspondiente a potencias de dos, dibujando cada dos, cuatro u ocho celdas. Todo esto en tiempo real (si se usan *quads* orientados, en el caso de sprites es necesario reconstruir la malla presionando un botón y aún así toma una fracción de segundo).

- **Distintos modos de dibujado de partículas:** Por el momento se emplean dos maneras para producir las partículas, el uso de *quads* orientados, si bien

más lento que *point sprites* con listas de despliegue, es compatible con cualquier tarjeta de video. Mientras que, por el momento, sprites con textura no funcionan adecuadamente en tarjetas ATI, aunque es un excelente método para visualizar mallas de muy alta resolución.

- **Manejo de sesiones:** Es posible crear archivos de sesión que contengan los parámetros de la visualización actual (correspondencia RGB de los datos, textura, saturación de canales, etc.), para poder recuperar el estado con sólo cargar la sesión correspondiente.

- **Uso de texturas y captura de imágenes:** Con el uso de *SDL_Image* y de *gd graphics library* se ha podido implementar el cargado de texturas a partir de imágenes de varios tipos, así como la posibilidad de tomar instantáneas de la visualización actual y crear imágenes png con ellas, pudiendo además escribir en ellas una sinopsis con información como: nombre de sesión, nombres y rutas de los datos y dimensiones de la malla.

4.3.9. Problemas

El principal inconveniente que se presenta al usar GLNebula no es un problema del sistema en sí, sino de un soporte incompleto de OpenGL por parte de uno de los mayores fabricantes de tarjetas gráficas: ATI. Desgraciadamente el manufacturador no ha completado el soporte para la especificación completa de la versión 1.5 de OpenGL, en la cual se introdujeron *point sprites*. Por el momento si se trata de usar este tipo de partículas en tarjetas fabricadas por ATI es muy posible que el modelo no aparezca, en algunos casos esto se resuelve deshabilitando las texturas. Para tarjetas fabricadas por NVIDIA el problema no se presenta y afortunadamente en el Ixtli se tienen tarjetas de este tipo.

Otro problema que surge al usar *sprites* es que el tamaño máximo de puntos soportado por la tarjeta varía de fabricante a fabricante, así mientras que una ATI puede

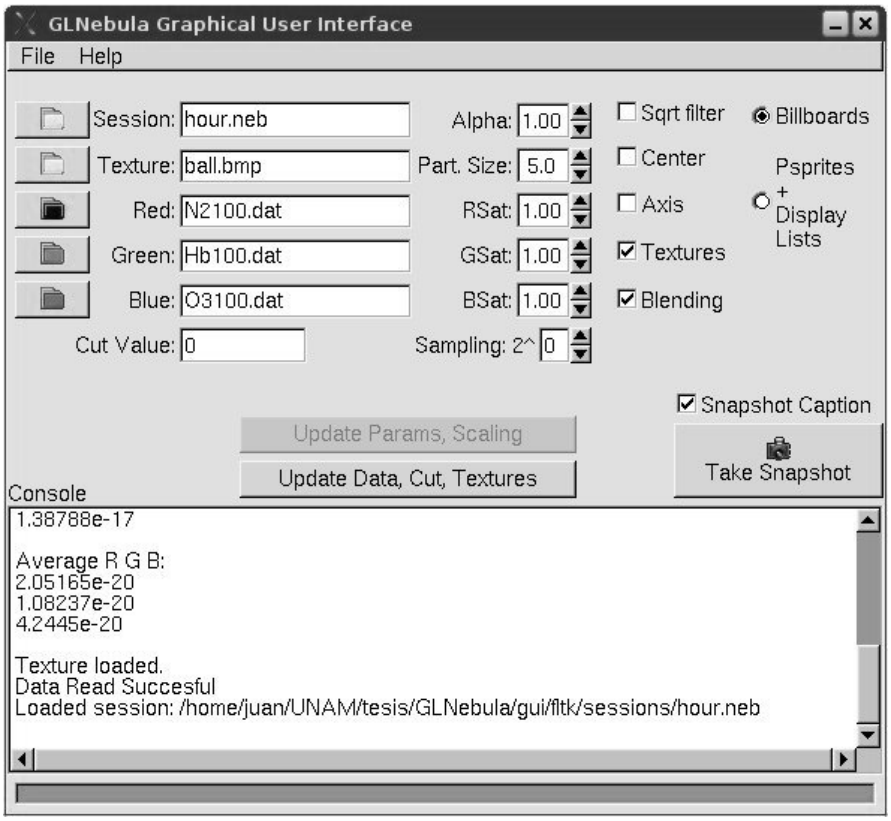

Figura 4.3: Ventana de controles de GLNebula

.

definir GL_POINT_SIZE_MAX_ARB como 1024, una NVIDIA de la misma generación no pasará de 64. Así, si el tamaño de los puntos es pequeño y la resolución de la ventana pasa de cierto punto, el modelo perderá solidez y se verán huecos entre las partículas.

4.4. Ejemplos de visualizaciones

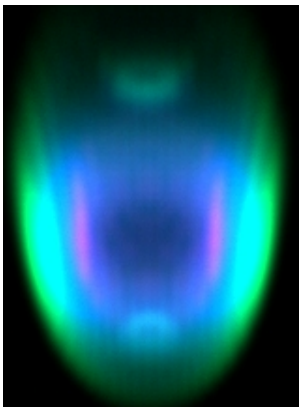

Figura 4.4: Nebulosa elipsoidal ficticia usando quads orientados, la malla es de 50x50x50. R = HeII, G = [OII], B = Hbeta. Datos cortesía de *Christophe Morisset, IA*

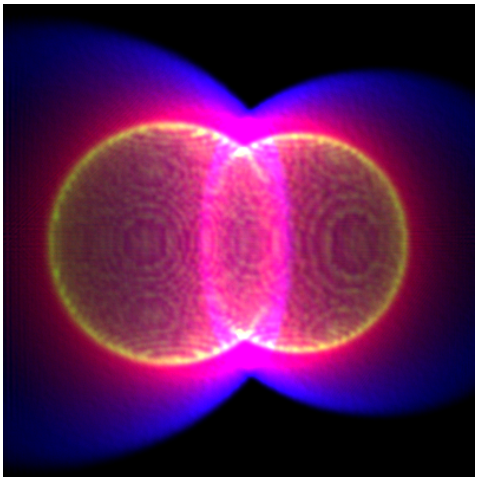

Figura 4.5: Nebulosa bipolar ficticia, empleando point sprites con textura. Malla de 150x150x150 R = Hbeta, G = HeII, B = [OII]. Datos cortesía de *Christophe Morisset, IA*

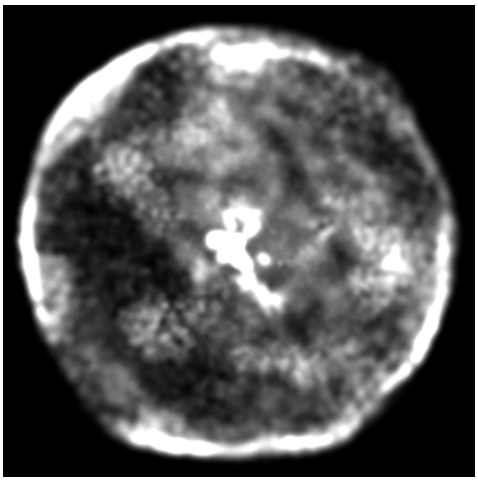

Figura 4.6: Abell 30, empleando quads Malla de 100x100x100. Se utilizó la misma capa para los tres colores. Datos cortesía de *Wolfgang Steffen, IA*

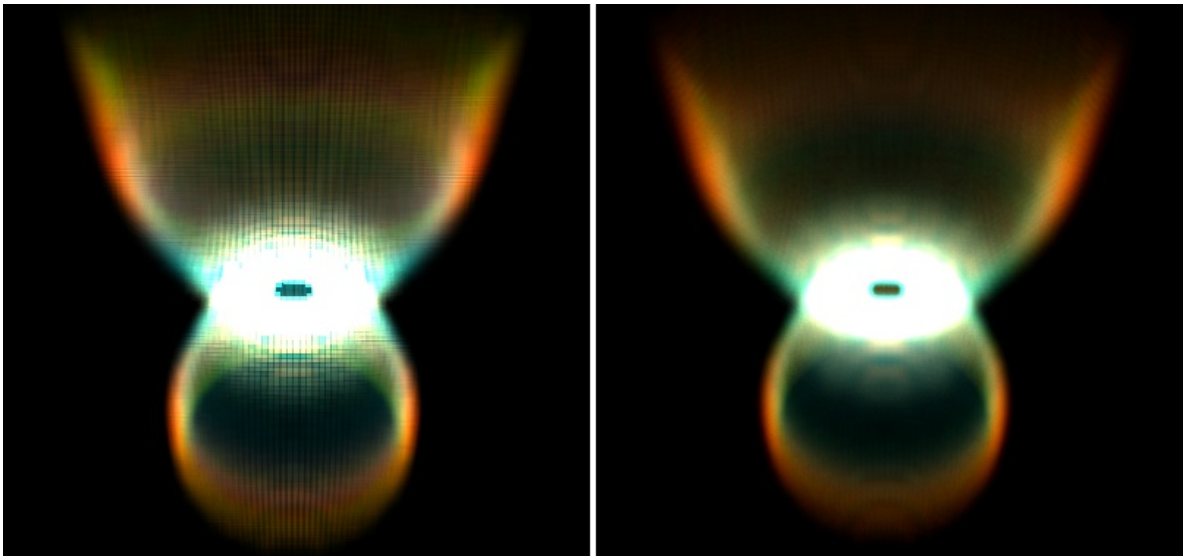

Figura 4.7: Modelo de MyCn18. En el lado izquierdo se emplean point sprites sin textura, del lado derecho el mismo modelo con quads texturizados. Malla de 100x100x100 R = [NII], G = Hbeta, B = [OIII]. Datos cortesía de *Christophe Morisset, IA*

Capítulo 5

Futuro y Conclusiones

La sección final de éste documento presenta los planes a futuro que se tienen para el desarrollo de GLNebula, así como las posibles optimizaciones que se le podrían aplicar para mejorar su desempeño, explorar el impacto que podría tener su aplicación en otras áreas de investigación que requieran visualización en tiempo real de partículas y por último se presentan las conclusiones obtenidas a partir de la finalización del trabajo.

5.1. Futuro

GLNebula es por el momento una herramienta usable, aún así fue necesario fijarse metas concretas para la primera versión. Se espera que con el uso surjan sugerencias y peticiones para características nuevas que hagan más completo y sencillo su funcionamiento. Por el momento se tienen planeados dos tipos: el primero está enfocado hacia la faceta didáctica de GLNebula, mientras que con el segundo se espera proveer mejor usabilidad.

5.1.1. Estereoscopía

En una versión futura se planea implementar estereoscopía activa para la visualización en la pantalla Ixtli, mediante la separación de canales empleando los lentes Crystal

Eyes. Se espera que la conversión no presente mayores inconvenientes, aunque existe la posibilidad de utilizar Chromium[1]: un sistema que permite paralelizar cualquier programa, además de que atrapa streams de OpenGL para hacer modificaciones directamente a los comandos que se llaman y así proveer stereo activo sin la necesidad de realizar modificación alguna a GLNebula. Cualquiera de las dos aproximaciones permitirán una sensación de inmersión en la nebulosa.

5.1.2. Mejoras y Optimizaciones

El desempeño actual de GLNebula es sumamente aceptable, aunado a esto es muy difícil superar el rendimiento de las display lists, aunque siempre queda espacio para el refinamiento. Por ello se espera que principalmente dos optimizaciones permitan una mejora en la cantidad de cuadros por segundo dibujados, incrementando la sensación de interactividad y tal vez permitiendo la visualización de modelos de mayor tamaño a los actuales, o al menos tener otros métodos para dibujar las partículas. La tercera optimización planeada mejoraría la usabilidad, permitiendo explorar sólo ciertas regiones del volumen, y el desempeño al reducir el número de celdas sin perder demasiado detalle. Por último se presenta una característica que se espera permita utilizar a GLNebula con topologías más complejas y la posibilidad de visualizar cambios a través del tiempo.

Vertex Arrays y Vertex Buffer Objects

Por el momento se realiza una llamada a glVertex cada vez que se define un punto de un GL_QUAD (4 por celda), además de una llamada a glTexCoord para cada coordenada de textura (4 más) y una llamada a glColor por celda. Esto se traduce en unas 9 llamadas a funciones de OpenGL por celda y si se toma en cuenta el gran número de éstas, la eficiencia del proceso es cuestionable. Para ello se crearon los Vertex Arrays: escencialmente se le pasa a OpenGL un apuntador dirigido hacia el conjunto de vértices, índices de color, coordenadas de textura, etc. de manera que se dibujen en una

[1] http://chromium.sourceforge.net/

sola pasada llamando a *glDrawElements()*. Además es posible indicarle qué vértices se deben pintar en determinado momento y cuáles deben ser ignorados. Posiblemente el impacto en el desempeño no sea tan grande, pero se espera que se encuentre entre los quads orientados y el uso de *display lists* con *point sprites*.

Recientemente se agregó la extensión ARB_vertex_buffer_object (VBO) a muchas tarjetas de video comerciales. La idea detrás de los VBOs es la misma que con *vertex arrays*, sólo que se emplea la memoria de alta velocidad en la tarjeta gráfica para almacenar los datos, permitiendo reducir considerablemente el tiempo de dibujado. Al ser una extensión relativamente nueva no podrá ser usada con cualquier tarjeta, pero se espera que los beneficios obtenidos usando hardware moderno sean considerables, tal vez poniéndolo a la par con el desempeño de las *display lists* sin la necesidad de utilizar exclusivamente tarjetas NVIDIA (aparentemente ATI soporta la extensión ARB_vertex_buffer_object sin mayor problema).

Subvolumen

En mallas de muy alta resolución se podría mejorar el desempeño empleando un modelo reducido del actual al momento de realizar las rotaciones, desplegando en tiempo real una malla con menor número de celdas y redibujando el modelo completo en cuanto cese la interacción con el usuario. Esto permitiría aumentar considerablemente el número de cuadros por segundo, tal vez trabajando con un modelo de la mitad o 1/3 del tamaño original, sin sacrificar la sensación de interacción al estar presentando un modelo similar. Se podría accionar automáticamente para mallas que excedan ciertas dimensiones, con la posibilidad de sobrecargar su activación por parte del usuario.

Debido a la cantidad de celdas involucradas en los modelos grandes es posible que la interpretación de la imagen se torne dificil, por se podría implementar un método de poda para reducir la malla y así poder dibujar sólo las áreas interés para el usuario,

omitiendo cualquier estorbo. Esto podría realizarse definiendo los extremos de la malla que se quiere visualizar, en forma de valores mínimos y máximos para cada coordenada; o bien haciendo un submuestreo más refinado que el actual, eliminando celdas de acuerdo a algún patrón en particular (por ejemplo convertir cada submalla de 3x3 a una especie de cruz, eliminando todas las celdas excepto la del centro y las que están a su alrededor, que si fueran cubos correspondrían a las que comparten una cara con ella). Otro acercamiento, si bien más complicado, consistiría en tomar en cuenta los valores de los vecinos que fueran cercanos de acuerdo con cierto valor de tolerancia, si los seis vecinos de una celda en particular poseen valores similares, podrían colapsarse en uno sólo, interpolando su intensidad, aunque habría que reestructurar la malla después de la poda y tal vez presente un reto mayor sin aportar demasiado.

Mallas no uniformes

Actualmente GLNebula requiere que los datos estén organizados en mallas con longitud de celdas constante para cada unidad, esto funciona con datos resultantes de códigos como NEBU_3D, pero como se mencionó en la sección correspondiente a los códigos de fotoionización, Mocassin puede trabajar con estructuras de mallas más complejas, resultando en modelos de mayor refinamiento. Se podría resolver este problema recibiendo un archivo de texto con una estructura en particular que contenga los espaciamientos de cada celda, así bastaría con construir la malla centrando cada una en la posición que indique el archivo.

Animación

Si se implementa el uso de mallas no uniformes definiendo las coordenadas de las celdas en un archivo, es posible extenderlo para usar n archivos representando los cambios en los escalares a visualizar en n estados de tiempo. De esta manera la estructura de la malla permanecería constante y se podría realizar una animación sobre los cambios de los valores. El único problema que presenta es que posiblemente la animación no fun-

cione con las listas de despliegue, debido a que en cada instante de tiempo es necesario reconstruir la malla (o por lo menos recalcular los cambios que sufrió), pero se espera que con *quads* orientados y las optimizaciones futuras trabaje satisfactoriamente.

5.1.3. Aplicación en otras áreas

Actualmente sólo se ha probado el programa con datos observacionales y de modelos de fotoionización de nebulosas planetarias, pero en principio cualquier campo escalar estático puede ser visualizado (nebulosas y regiones ionizadas en general), siempre y cuando se provean tres mallas uniformes con un sólo escalar por celda (aunque sería sencillo modificar el programa para que en un sólo archivo se especifiquen los tres valores en cada celda). Más aún, si se implementa la funcionalidad de animación, se podrían visualizar los cambios que sufre el campo escalar en cierto intervalo de tiempo, permitiendo la visualización de flujos en tiempo real. Por el momento ya hay planes para utilizar GLNebula en la visualización de campos escalares, se ha platicado con algunos interesados y se espera adaptar el programa a sus necesidades. Si se logra la adaptación, se podría emplear a GLNebula en áreas como: ciencias de la atmósfera (tal vez visualizando movimiento de frentes y cambios de temperatura), dinámica de fluídos (visualización de flujos y mezclado de sustancias) y convertise en una herramienta multidisciplinaria, posiblemente cambiando de nombre y de características, pero siempre tratando de mantener el uso de partículas para mantener la ilusión de volumen en tiempo real.

5.2. Conclusiones

El desarrollo de GLNebula tuvo sus altibajos: si se hubieran diseñado bien las cosas desde el principio y estudiado a fondo las aproximaciones antes de implementarlas, se hubiera ahorrado mucho tiempo y frustración. Ahora parece obvio que *volume rendering* fue una pésima elección y tal vez se pudo haber discriminado con dedicarle unas horas

a investigar a fondo las ventajas y desventajas, en vez de enfocarse a los detalles de la implementación en VTK. Por otro lado la segunda aproximación se desarrolló relativamente rápido y afortunadamente funcionó, aunque también tuvo sus errores, siendo la elección de GLUI como biblioteca para desarrollar la interfase gráfica uno de ellos. Es claro entonces, que la conclusión más importante que se obtuvo del trabajo fue la importancia de tener un diseño inicial bien pensado, si bien es imposible predecir todos los problemas que se presentarán, sí se pueden identificar posibles trabas o defectos escenciales del diseño antes de llevar a cabo la implementación.

Afortunadamente después de estar trabajando en el programa surgieron posibles aplicaciones del mismo y se espera que con algunas modificaciones se tenga una herramienta útil no sólo para los astrónomos, sino para cualquier investigador interesado en la visualización de campos escalares (estáticos por el momento, dinámicos en el futuro) en tiempo real. Evidentemente con el uso saldrán nuevas inquietudes sobre nuevas características y errores, pero por el momento GLNebula ha demostrado funcionar y ha cumplido con los objetivos propuestos al inicio del trabajo.

Apéndice A

Manual

http://glnebula.sourceforge.net/

A.1. Requerimientos e Instalación

La manera más sencilla de ejecutar GLNebula es en Windows, sólo es cuestión de bajar el zip que contiene el ejecutable y los .dlls necesarios y dar doble click en la aplicación.

La compilación del código fuente requiere varias bibliotecas y sus encabezados respectivos para funcionar:

- **OpenGL**[1], **GLUT**[2] Ambos incluídos en el controlador de la tarjeta de video
- **GLEW**[3]
- **Fast Light Toolkit (fltk) 1.1.7**[4]

[1] *http : //www.opengl.org/*
[2] *http : //www.opengl.org/resources/libraries/glut.html*
[3] *http : //glew.sourceforge.net/*
[4] *http : //www.fltk.org*

- **GD Graphics Library**[5]

- **SDL, SDL_Image**[6]

Además se recomienda una tarjeta de video reciente, el desempeño mejora mucho y los precios de hardware para consumidor son relativamente bajos (en versiones futuras GLNebula empleará *shaders*, que permitirán la visualización de modelos grandes con mayor detalle y desempeño).

A.1.1. Compilación en Linux

Compilar es tan sencillo como *./configure*, *make* (el ejecutable quedará en el directorio src/) y opcionalmente *make install* que copiará el binario en /usr/local/bin.

El Makefile adjunto buscará los encabezados en sus directorios respectivos comenzando en /usr/include y /usr/local/include (por ejemplo el encabezado gl.h deberá estar en /usr/include/GL/gl.h o /usr/local/include/GL/gl.h y así con el resto).

También se provee un archivo .kdevelop el cual puede ser abierto en el editor kdevelop[7] para construir el proyecto.

A.1.2. Compilación en Windows

Para windows hay un archivo de proyecto para DevC++[8], sólo hay que instalar los *devpaks* apropiados (todos están disponibles en los devpaks comunitarios, dentro del manejador de paquetes del DevC++). y compilar el proyecto.

[5] *http://www.boutell.com/gd/*
[6] *http://www.libsdl.org/*, *http://www.libsdl.org/projects/SDL_image/*
[7] *http://www.kdevelop.org/*
[8] http://www.bloodshed.net/devcpp.html

A.2. Uso de GLNebula

A.2.1. Ventana de Control

Aquí es donde se definen los parámetros de la visualización, revisando cada control (ver figura 4.3):

- **Menú**: El menú posee 4 submenús:

  Load Session: Abre un diálogo para seleccionar la sesión a cargar

  Save Session: Salva la sesión actual

  Save Session As: Abre un diálogo para escribir el nombre de la sesión a guardar

  Quit: Sale del programa

- **Botones de Carga**: Cuando son presionados, los botones del lado izquierdo de la ventana abrirán un selector de archivos que permitirá escoger el archivo a cargar: el primero cargará una sesión (similar al submenú *Load Session*), el siguiente permite escoger la textura (en formato bmp) que será aplicada a las partículas (esta característica es opcional, el modelo puede ser visualizado sin textura), por último los tres botones restantes corresponden a los datos (en orden RGB). El nombre de los archivos seleccionados aparecerá en las cajas de texto a la derecha de cada botón. Cuando los tres datos han sido seleccionados, es necesario apretar el botón *Update Data, Cut, Textures* para desplegar el modelo en la ventana principal (este paso no es necesario si una sesión ha sido cargada, en el momento de seleccionar el nombre de la sesión, los datos serán cargados automáticamente).

- **Cajas de Texto**: Las primeras cinco cajas despliegan el nombre de los archivos cargados con su botón de carga asociado. La última, etiquetada *Cut Value*, es editable. En esta caja va el valor mínimo permitido de los escalares en los datos: celdas que posean un valor igual o menor a este número, serán consideradas invisibles y por lo tanto no serán dibujadas. Es de destacar que cuando los datos son

cargados, en la consola aparecen los valores máximo, mínimo y promedio de cada canal, lo que da una idea de valores posibles de corte.

- **Parámetros**: Estos controles manipulan los distintos parámetros que definen la visualización.

  Alpha: Controla la transparencia de las partículas, va de 0 a 1 en incrementos de 0.01 cada vez que la flecha es presionada.

  Particle Size: Modifica el tamaño de las partículas en el modelo, el tamaño máximo está ligado a un parámetro particular a cada tarjeta de video, así que el límite es variable (usualmente 63.4 en tarjetas nvidia).

  Saturation: Los siguientes tres controles modifican la saturación de cada color, si por ejemplo se quiere ver sólo la presencia del verde y el azul, es necesario poner RSAT en 0.

  Sampling: Controla el muestreo de la malla en potencias de 2, si está puesto en 1 sólo la mitad de las celdas en la malla serán desplegadas, si se sube a 2 sólo una de cada cuatro celdas aparecerán.

- **Casillas de Selección**: Controlan el despliegue de la caja que indica las dimensiones de malla, así como los ejes y el número de cuadros por segundo que están siendo pintados. También es posible habilitar o deshabilitar el mapeo de texturas y mezclado de colores con ellas.

- **Selectores de Tipo de Partícula**: Mutuamente exclusivos, determinan el tipo de partícula a usar.

- **Botones de Actualización**: Redibujan la ventana de despliegue para que los cambios en la visualización sean aplicados.

  Update Data/Cut/Textures: Este botón reconstruye la malla, actualiza el valor de corte y carga las texturas.

Update Params/Scaling: Cuando se emplean *billboards* los cambios en los parámetros se realizan en tiempo real, así que este botón es desactivado. Sin embargo al usar sprites es necesario presionarlo para reflejar los cambios.

- **Captura de Imágenes (Take Snapshot)**: Al presionarse este botón la sesión será salvada y se capturará una imagen (png) de la visualización mostrada en la ventana principal. La caja de selección localizada arriba del botón habilita o deshabilita escribir información relevante a la imagen (dimensiones de la malla y nombre de los archivos).

- **Consola**: Aquí se imprime toda la información que el programa regresa, cualquier error que se presente en el procedimiento es reportado en la consola.

A.2.2. Ventana Principal

Es aquí donde el modelo aparecerá una vez que los datos hayan sido cargados. Además aquí se capturan los eventos de ratón que controlan la cámara: para rotar alrededor del modelo se debe mantener presionado el botón izquierdo mientras se mueve el puntero de lado a lado, si se desea acercar la cámara es necesario ahora presionar el botón derecho mientras se realiza un movimiento sea hacia arriba para acercarse o hacia abajo para alejarse. Esta última operación también puede ser realizada con la rueda del ratón.

Índice de cuadros

Índice de figuras

Bibliografía

[1] Balick, B. "The evolution of planetary nebulae. I - Structures, ionizations, and morphological sequences" *Astronomical Journal*, 94:671–678, 1987.

[2] Meixner, M. " Classification of Planetary Nebulae Morphology" *ASP Conf. Ser. 313: Asymmetrical Planetary Nebulae III: Winds, Structure and the Thunderbird*, 313:24-+, 2004.

[3] Rózyczka, M. "Planetary Nebulae Dynamics In a Nutshell" *AIP Conference Proceedings 804:Planetary Nebulae As Astronomical Tools*, 804:68–75, 2005

[4] Morisset, C. et. al. "NEBU_3D: A Fast Pseudo-3D Photoionization Code for Aspherical Planetary Nebulae and HII Regions" *AIP Conference Proceedings 804:Planetary Nebulae As Astronomical Tools*, 804:44–47, 2005

[5] Balick, B. "The Challenges of the Last Decade of Observations of PNe" *AIP Conference Proceedings 804:Planetary Nebulae As Astronomical Tools*, 804:77–80, 2005

[6] Kwok, S. "Evolution from the Asymptotic Giant Branch to Planetary Nebulae" *AIP Conference Proceedings 804:Planetary Nebulae As Astronomical Tools*, 804:187–192, 2005

[7] Frank, A. "PNe as Laboratories of Astrophysical MHD" *AIP Conference Proceedings 804:Planetary Nebulae As Astronomical Tools*, 804:81–84, 2005

[8] García-Segura, et. al. "Three-dimensional Magnetohydrodynamic Modeling of Planetary Nebulae. II. The Formation of Bipolar and Elliptical Nebulae with Point-symmetric Structures and Collimated Outflows" *The Astrophysical Journal: 544*, 544:336-346, 2000

[9] Soker, N. "Can We Ignore Magnetic Fields in Studies of PN Formation, Shaping and Interaction with the ISM" *AIP Conference Proceedings 804:Planetary Nebulae As Astronomical Tools*, 804:89–92, 2005

[10] Ercolano, B. "New Advances In Photoionization Codes: How And What For?" *http://arxiv.org/abs/astro-ph/0508683*, 2005

[11] Ercolano, B. et. al. "Mocassin: A fully three-dimensional Monte Carlo photoionization code" *http://arxiv.org/abs/astro-ph/0209378*, 2003

[12] Soker, N. "Properties that Cannot Be Explained by the Progenitors of Planetary Nebulae" *Astrophysical Journal Supplements*, 112:487, 1997

[13] Schroeder, W. & Martin, K. "Overview of Visualization" *Visualization Handbook*, 3–39, 2004

[14] Lorensen, W. & Cline, H. "Marching cubes: A high resolution 3D surface construction algorithm" *Proceedings of the 14th annual conference on Computer graphics and Interactive techniques*, 163-169, 1987