UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

Escuela Nacional de Estudios Profesionales

Campus Aragón

IDL COMO LENGUAJE PARA LA VISUALIZACIÓN DE VOLÚMENES

T E S I S

PARA OBTENER EL TÍTULO DE

INGENIERO EN COMPUTACIÓN

P R E S E N T A:

ISMAEL SALAS RÍOS

Asesor: Ing. Liliana Hernández Cervantes

San Juan de Aragón, Estado de México, Mayo 2003

TESIS
CON
FALLA DE
ORIGEN

PAGINACION

DISCONTINUA

Agradecimientos

A mis Padres:

Por que me han dado dos grandes regalos:
la vida y las herramientas para hacer algo con ella.

A mi Hermano:

Por que no supe ser un Hermano Mayor.

A mis demás Familiares:

Por que siempre estuvieron ahí.

A mi Alma Mater:

Por brindarme los espacios necesarios para mi superación.

A mi Asesor:

Por brindarme su tiempo y dedicación.

Al Doctor Alfredo Santillán:

Por brindarme su ayuda.

Dedicatoria

A ti...
Que siempre estas conmigo y, sin embargo, me haces sentir tan solo.

A ti...
Que me animas cuando caigo y me haces tropezar cuando avanzo.

A ti...
Que me das razones para amar cuando siento odio,
y me das razones para odiar cuando siento amor.

A ti...
Que me das alegría en mi tristeza y tristeza en mi alegría.

A ti...
Que iluminas mi oscuridad y ensombreces mi luz.

A ti...
Que ante la certidumbre me das la duda y ante la duda me das indiferencia.

A ti...
Que enriqueces mis fantasías y empobreces mis realidades.

A ti...
Tú sabes quien.

I. S. R.

Índice

Índice de Ilustraciones

Abreviaturas Utilizadas

ASCII	Código Estándar Americano para el Intercambio de Información (*American Standard Code for Information Intercange).*
CAD	Diseño Asistido por Computadora *(Computer Aided Design).*
CAL	Enseñanza Asistida por Computadora *(Computer Aided Learning).*
CMY	Cian, Magenta y Amarillo, Modelo de Color *(Cyan, Magenta, Yellow).*
CRT	Tubo de Rayos Catódicos *(Cathode Ray Tube).*
DDA	Algoritmo Digital Diferencial *(Digital Diferencial Algorithm).*
DMIP	Proyección de Intensidad Máxima con Sombreado Profundo *(Depth-Shaded Maximum Intensity Projection).*
GUI	Interface Gráfica de Usuario *(Graphical User Interface).*
HDF	Formato de Datos Jerárquicos *(Hierarquical Data Format).*
HSI	Matiz, Saturación e Intensidad, Modelo de Color *(Hue, Saturation, Intensity).*
IDL	Lenguaje Interactivo de Datos *(Interactive Data Language).*
IDLDE	*Interactive Data Language Developed Enviroment.*
KB	Kilo Bytes.
LED	Diodo Emisor de Luz *(Light Emisor Diode).*
LMIP	Proyección de Intensidad Máxima Local *(Local Maximum Intensity Projection).*
MDI	Interface de Múltiples Documentos *(Multiple Document Interface).*
MIP	Proyección de Intensidad Máxima *(Maximum Intensity Projection).*
MV	MainVolume.
NDRE	Instituto Noruego de la Defensa *(Norwegian Defence Research Establishment).*
NURBS	B-Splines No Uniformes y Racionales *(No-uniform, Rational B-Spline).*
PDI	Procesamiento Digital de Imágenes.
RC	Lanzamiento de Rayos *(Ray Casting).*
RGB	Rojo, Verde y Azul, Modelo de Color *(Red, Green, Blue).*

VC	**Visualización Científica.**
VR	**Volume Rendering.**
VMS	**Sistema de Memoria Virtual** (*Virtual Memory System*).
XDR	**Representación Externa de Datos** (*External Data Representation*).
YIQ	**Luminiscencia, Fase y Cuadratura, Modelo de Color** (*Luminance, Inphase, Quadrature*).

Introducción

En el ámbito académico y de investigación la tarea de interpretar los resultados de una investigación conlleva al análisis de una gran cantidad de datos, siendo la Visualización Científica la principal proveedora de herramientas y soluciones para facilitar estas tareas. En este sentido, los académicos e investigadores utilizan las técnicas más adecuadas para realizar una mejor interpretación de sus resultados, por ejemplo, los investigadores del Instituto de Astronomía de la UNAM se auxilian de las herramientas que proporcionan la técnica de *Volume Rendering* para visualizar espacios volumétricos que representan densidad, forma y constitución de estrellas, galaxias y demás cuerpos celestes.

Sin embargo, no existen aplicaciones generales que manejen volúmenes, más bien existen conjuntos de herramientas que permiten crear aplicaciones específicas para implementar un *Volume Rendering* de acuerdo a las necesidades del investigador; no obstante, esto implica que el interesado deba poseer conocimientos de programación o conozca perfectamente el funcionamiento de ese conjunto de herramientas para llevar a cabo su objetivo. Ante esta situación, la mayoría de los investigadores han optado por aprender lenguajes que no involucren entender paradigmas de programación, sino que, más bien, sean flexibles y fáciles de entender, pero sin que pierdan las potencialidades que ofrecen lenguajes de alto nivel como C o Java. Así pues, uno de los lenguajes que cumple con tales características es IDL, él cual ofrece una gran flexibilidad y poder de programación, además, proporciona una gran cantidad de librerías gráficas y recursos equiparables a otras librerías gráficas, sin que esto le obligue a perder su sencillez de programación, en resumen, IDL es un lenguaje sencillo para crear aplicaciones poderosas.

La unión de los conceptos de la técnica de *Volume Rendering* y la aplicación de las herramientas que proporciona IDL, dieron como origen una aplicación que provee de los elementos necesarios e indispensables para la manipulación de volúmenes, así como de sus propiedades, de una manera rápida, sencilla y eficiente, obteniendo los resultados esperados. Esta aplicación la he llamado

MainVolume y es el objetivo fundamental del presente tema de investigación y cuya finalidad es proporcionar a los investigadores del Instituto de Astronomía una herramienta que realice despliegues gráficos de espacios volumétricos con una calidad satisfactoria. En este sentido, la presente tesis se ha divido, por conveniencia de la investigación, en cuatro capítulos que irán introduciendo al lector a partir de un panorama general sobre la Graficación por Computadora hasta particularizar con los temas que envuelve la técnica de *Volume Rendering*, en otras palabras, este trabajo de investigación parte de lo general a lo particular.

El primer capítulo está dedicado a mostrar un vistazo general del mundo de la Graficación por Computadora partiendo con la narración de la historia de los sucesos que dieron origen a esta ciencia, seguida por la definición de algunos conceptos básicos que son indispensables para el entendimiento de la terminología de este ambiente. Después, se introduce al lector, sin entrar en detalles, a una serie de fundamentos matemáticos que forman parte de la teoría general la Graficación por Computadora, y que son indispensables para el despliegue y manipulación de cuerpos geométricos a través de primitivas de dibujo. La siguiente sección está dedicada a describir las características esenciales y funcionamiento básico de los principales sistemas de despliegue que existen en la actualidad. Finalmente, este capítulo culmina con una breve descripción de las principales actividades de las áreas de estudio derivadas de la Graficación por Computadora.

El segundo capítulo se enfoca en el análisis del *Volume Rendering*, sin embargo, comienza con la exploración de la Visualización Científica, que es una de las áreas de estudio descrita en el capítulo anterior. Por conveniencia, este capítulo se dividió en tres secciones: Visualización Científica, Procesamiento Digital de Imágenes y *Volume Rendering*. En la primer sección, son descritos los conceptos básicos que se utilizan en la VC, así como de la línea de trabajo que sigue esta área de estudio, a partir de su metodología. La segunda sección está dedicada a la explicación de algunos conceptos, fundamentos y teoremas del Procesamiento Digital de Imágenes, iniciando con la descripción de las diferentes disciplinas que engloba esta área de estudio y otorgándole más peso al análisis de la imagen en sus dos diferentes dominios, mostrando a su vez, las diferentes herramientas aplicables en cada uno de ellos. Por último, en la tercera sección, se desarrollan los conceptos fundamentales del *Volume Rendering*; dicha sección incluye la definición de esta técnica, así como de los elementos más preponderantes para llevarla a cabo, como por

ejemplo, la función de composición, los *kernels* de interpolación, el cálculo del color y la opacidad, etc.

El tercer capítulo inicia otra línea de estudio totalmente diferente a los dos anteriores capítulos, pues introduce al lector en los conceptos básicos de la sintaxis de IDL, mostrando las ventajas y características principales de este lenguaje. Primeramente, muestra al lector las nociones fundamentales de este lenguaje, como por ejemplo, la declaración de variables, el manejo de arreglos y punteros y demás conceptos; conforme avanza en el capítulo se introduce en un nivel superior de programación en este lenguaje, mostrando el uso de instrucciones más complejas y las diferentes clases de paradigmas de programación que IDL soporta. Con el uso de ejemplos y tablas es más fácil la comprensión de las particularidades de este lenguaje a fin de ayudar a cualquier persona, interesada en el tema, a crear aplicaciones básicas. Así mismo, este capítulo introduce al uso de los controles o widgets, que posee IDL, para la construcción de interfaces gráficas de usuario y al uso de los objetos gráficos para llevar a cabo los despliegues de escenas de *render*.

Finalmente, el cuarto capítulo conjunta la investigación de los tres anteriores capítulos para describir el funcionamiento de la aplicación *MainVolume*, detallando las herramientas IDL que utiliza para llevar a cabo sus tareas. En cada una de las tres secciones en las que se dividió este capítulo, se describe la funcionalidad de cada uno de los módulos que componen la interfaz de *MainVolume*. En la primera sección se describe el funcionamiento de cada uno de los menús, así como de las herramientas que utilizan de IDL para llevar a cabo su cometido. La segunda sección de este cuarto capítulo, está dedicada a explicar el funcionamiento del área de *render* principal de la interfaz de *MainVolume*, profundizando en el tema de los objetos gráficos, su sintaxis, clasificación y forma de utilizarlos para construir un escenario gráfico utilizando una jerarquía de estos objetos, conocida como árbol de objetos gráficos. En la tercera sección, figura el funcionamiento de los controles dedicados a manejar la transformación de coordenadas y el porcentaje de corte que se pueden aplicar al volumen contenido en la escena principal.

Espero sinceramente, que este trabajo de investigación sea del agrado del lector y, además, aporte una gota de agua al gran océano de conocimientos que existen alrededor del *Volume Rendering*, de la Visualización Científica y de la Graficación por Computadora.

Capítulo I.

Introducción a la Graficación por Computadora

1.1. Historia.

En 1824, *Peter Mark Roget* hacia hincapié en la persistencia de las imágenes en la visión y la posibilidad de usarlas para llevar a cabo una mejor explicación de datos. Bajo este concepto, en el año de 1920, se intentaba mejorar la calidad de las imágenes recibidas por cable submarino usando métodos incipientes de digitalización. El surgimiento de las computadoras originó que el procesamiento de datos y la generación de información agudizara la necesidad de representar, de alguna forma, toda esa información generada; para ello se emplearon distintos métodos, siendo el más exitoso el uso del Tubo de Rayos Catódicos (CRT, *Cathode Ray Tube*), introducido por *Tektronix Inc.*, que ofrecía la posibilidad de desplegar imágenes usando un equipo electrónico y eléctrico.

A pesar de los adelantos tecnológicos de aquella época, las computadoras trabajaban a través del procesamiento por lotes (*batch*) haciéndolas realmente lentas y poco factibles para cálculos complicados y extensos, esta situación llevó a los investigadores a crear un nuevo modelo de procesamiento, dando como resultado la computadora TX-2, lanzada al mercado a principios de 1960 y cuyas características principales eran: procesamiento en línea, capacidad de memoria de 320 KB, teclado en línea, almacenamiento en citas magnéticas y un CRT de nueve pulgadas.

Usando la computadora TX-2, en la primavera de 1963, *Ivan Sutherland*, presentó su tesis doctoral *"Sketchpad: A Man-machine Graphical Communications System"*, donde presentaba, por primera vez en el mundo, un sistema donde se podían dibujar puntos, líneas, arcos, polígonos y formas geométricas básicas, usando solamente un *display* de CRT, una pluma emisora de luz (*lightpen*) y un banco de interruptores. Entre las características más sobresalientes de este nuevo sistema destacaba su alta precisión de dibujo, manipulación, duplicado y almacenamiento, además, su software permitía escalamientos de 2000:1 adicionándole un gran espacio de dibujo. Sin embargo, este sistema tenía una seria desventaja ya que utilizaba el método llamado *vector refresh* el cual consiste en mover un rayo de luz a lo largo de un *display* y dejar una línea iluminada, ello implica que el software debe re-dibujar la información del *display* de manera cíclica en el orden en que han aparecido las figuras, por lo que el ciclo crecía enormemente y debido a la raquítica velocidad de procesamiento las primeras figuras comenzaban a

desvanecerse aún cuando el ciclo no terminaba de dibujar las últimas. Pese a estas dificultades, las innovaciones técnicas utilizadas para desarrollar este sistema le otorgaron el título del primer sistema interactivo de gráficos, además de ser la primera interfaz gráfica de usuario reconocida. Sin duda muchos autores ponen a este sistema como pionero en el desarrollo de la graficación por computadora.

Una década anterior, *Kristen Nygaard* había detectado la necesidad de simular sistemas dinámicos en la computadora mientras realizaba estudios sobre armas nucleares en el Instituto de Investigación de Defensa Noruego (*NDRE, Norwegian Defence Research Establishment*). Al mismo tiempo, *Ole-Johan Dahl*, en el Instituto de Computación Noruego, llevaba a cabo otros experimentos sobre los lenguajes de programación. Ambos investigadores unieron sus esfuerzos para dar origen a *SIMULA I*. Esta herramienta era capaz de simular sistemas dinámicos y al mismo tiempo ser un lenguaje de programación compatible en ambientes comerciales y de investigación. A pesar de los esfuerzos de estos investigadores, las dificultades tecnológicas y financieras truncaron el proyecto de *SIMULA*. Ante esta situación, *Nygaard* buscó apoyo financiero convocando a una serie de conferencias invitando a varios representantes de *UNIVAC*. *James Nickitas*, representante de *UNIVAC* Europa, solventó las necesidades financieras de estos investigadores y fue, entonces, cuando el desarrollo de SIMULA finalizó.

Regresando a la década de 1960, surgieron algunos dispositivos mecánicos para realizar trabajos de graficación sobre impresiones; uno de los más famosos y costosos de su época fue el *plotter* de pluma mecánica. Este dispositivo utilizaba, principalmente, dos mecanismos: el *plotter* de tambor (*drum plotter*) y el *plotter* de cama (*Flat Bed Plotter*). Paralelamente, surgieron sistemas de despliegue que empezaron a utilizar *píxeles*, para dar una gran calidad en el manejo de tonalidades de colores blanco y negro en las imágenes, sin embargo, los requerimientos de memoria eran altísimos. Con la aparición del semiconductor y de la memoria integrada, alrededor de 1970, comenzó a disminuir el costo de este tipo de sistemas y se comercializaron ampliamente en el mercado. Este tipo de tecnología, llamada *raster* posteriormente, consiste en dividir el área de despliegue en una malla rectangular con un determinado número de líneas equidistantes entre sí; en cada intersección, el dispositivo de despliegue ilumina un punto, llamado *pixel*. El número de líneas empleadas determina la resolución del dispositivo, asimismo, cada dispositivo tiene

asociado un *frame buffer*[1] para almacenar los cambios de valores ocurridos en él, ello determina el estado actual del *frame buffer* que es desplegado renglón por renglón en el dispositivo y es actualizado varias veces en un segundo para darle claridad a la imagen.

Es evidente que el desarrollo de software de graficación está profundamente ligado con el desarrollo del hardware apropiado. Así pues, estos nuevos avances tecnológicos permitieron a los investigadores del Centro de Investigación de Palo Alto de *Xerox* la oportunidad de crear interfaces gráficas que sirvieran de comunicación entre el usuario y la máquina, utilizando imágenes y texto. En este sentido, se desarrolló el sistema *Smalltalk-80*, el cual era todo un ambiente gráfico de comunicación entre la computadora y el usuario, y no simplemente un sistema de despliegue. La razón del éxito de *Smalltalk-80* fue que se desarrolló bajo los conceptos y experiencias de los sistemas *Sketchpad*, *SIMULA*, *Flex* y *Dynabook*, además de que el hardware requerido por este sistema ya estaba disponible. *Smalltalk* evolucionó constantemente aportando valiosas contribuciones a la Graficación por Computadora en cada nueva versión que salía al mercado; tales aportaciones son la utilización de gráficas tipo "tortuga", la manipulación del ratón para el manejo de un editor de gráficas estructuradas, la introducción de un sistema de animación y de música, la administración de la memoria, el uso de gráficas a través de bits, y lo más importante fue el concepto de portabilidad. Este último se dividió en dos componentes: la Imagen Virtual (*Virtual Image*) y la Máquina virtual (*Virtual Machine*). La Imagen Virtual consiste en una definición de estructuras de datos y clases, manejadores de texto y gráficas, compiladores, decompiladores, depuradores y *GUI* utilizando *byte codes* (un lenguaje intermedio entre el software y los códigos nativos de la máquina), mientras que la Máquina Virtual, es la interprete de *byte codes* y dependiente del código nativo de la máquina.

En la actualidad, los dispositivos de hardware y software han evolucionado tanto que la mayoría de los sistemas operativos y aplicaciones utilizan interfaces amigables; lenguajes de programación que explotan al máximo las características de graficación por computadora para la investigación, educación, entretenimiento, etc.

[1] Un *Frame Buffer* es una memoria de video cuya función es mantener el valor de los *pixeles* mientras el dispositivo de despliegue es actualizado.

1.2. Conceptos Básicos.

Actualmente, cualquier proceso de despliegue que utilice las herramientas de la Graficación por Computadora debe pasar por una metodología ó *pipeline*. Esta metodología consiste en una secuencia de pasos basados en el tipo de tarea que se desea llevar a cabo, aunque, la mayoría de la documentación de este tema, reconocen tres pasos esenciales que a continuación se describen.

El primer paso consiste en tener un modelo de la escena, el cual describe de manera abstracta lo que será desplegado, aunque no siempre, el modelo puede ser asociado al mundo real, pero es necesario tener un concepto más o menos definido del modelo principal. El segundo paso consiste en describir el modelo de la escena en objetos básicos y sus relaciones lógicas unos con respecto a otros, así también los colores y dimensiones del modelo. El último paso corresponde a la representación gráfica del modelo, esto es, la utilización de primitivas de dibujo para aproximarse al modelo original.

1.2.1. Píxeles.

Las variaciones de intensidades en cada uno de los puntos que conforman la matriz empleada por los tubos de rayos catódicos, originan modificaciones en los atributos de la imagen final. Estos puntos, invisibles para el ojo humano, se denominan elementos de la imagen (*picture elements*) o *píxeles*, los cuales están regularmente espaciados en dirección horizontal y vertical, aunque no necesariamente el espacio entre las líneas horizontales y verticales es el mismo, además, pueden adquirir distintos valores de intensidad dependiendo del hardware y software utilizado, por ejemplo, en sistemas monocromáticos, el valor mínimo de un *píxel* es cero y el valor más alto es uno. También es posible asociar arreglos de memoria matriciales para representar el valor de cada *píxel*, así los sistemas que utilizan millones de colores se auxilian de estos arreglos de memoria bajo el nombre de *frame buffers*. La representación física de los *píxeles* puede variar de un sistema a otro, habitualmente suelen ser representaciones cuadráticas o rectangulares. Finalmente, los dispositivos que utilizan *píxeles* para sus despliegues se denominan dispositivos *raster*.

1.2.2. Vectores.

Otra forma de desplegar imágenes en dispositivos gráficos es a través del uso de líneas, debido a que estas primitivas poseen longitud y dirección y pueden ser tratadas como vectores. Una imagen vectorial está compuesta por miles de pequeños vectores y cada uno de ellos posee información especifica del área de la imagen que representan. Además, es posible escalarlos al tamaño requerido por el dispositivo de despliegue.

1.2.3. Escala de grises.

Tanto los vectores como los *píxeles* pueden variar sus valores de intensidad de acuerdo a las necesidades de la imagen, este rango de valores se denomina nivel de intensidades o escala de grises. Dependiendo del software y hardware empleado, se asocia cierta cantidad de bits para representar las diferentes tonalidades de grises, así por ejemplo, un sistema de tres bits sólo puede asociar 6 diferentes tonos grises, y dos más correspondientes al blanco y al negro.

1.2.4. Color.

El color ha sido un poderoso medio de información y comunicación a través de las diferentes etapas de la historia del ser humano; en cada cultura, el color adquiere gran importancia ya que se relaciona con diferentes significados, es por esta razón que los colores utilizados para desplegar imágenes deben ser elegidos de manera óptima, ya que una mala interpretación puede ocasionar conclusiones erróneas.

El ojo humano responde a las frecuencias generadas por cada color, físicamente sólo existen tres colores, llamados colores primarios o colores aditivos, que originan todos los demás a partir de la mezcla entre ellos; bajo este principio se sustenta el modelo RGB (*red, green, blue*), que es una representación tridimensional de estos colores para crear una aproximación a la

realidad y a la forma en como el ojo humano interpreta los colores, así, la mayoría de los dispositivos de despliegue que se han construido utilizan los principios del modelo RGB. No obstante, existen otros modelos, como el modelo CMY (*cyan, magenta, yellow*), utilizado en impresoras y *plotters*, siendo la parte complementaria del modelo RGB, es decir, está conformado por colores substractivos o complementarios.

A pesar de que el modelo RGB otorga una calidad satisfactoria existe otro que brinda mejores aproximaciones al color real, este modelo se conoce como HSI (*hue, saturation, intensity*) el cual define el color en términos de matiz, saturación e intensidad. El matiz de un color se refiere a la mezcla de las longitudes de onda de los colores primarios, es importante hacer hincapié de que la percepción del matiz de un color puede variar de un individuo a otro. La saturación del color se refiere a su pureza, matemáticamente hablando, es la inversa de la cantidad de gris que tiene el color, por ejemplo, un color que no tiene gris se dice que está completamente saturado, no así los colores blanco, negro y gris, que se dice que tiene cero saturación. Y, por último, la intensidad de un color se refiere a la brillantez del mismo. El sistema HSI se usa sistemas especializados de dibujo. Por otro lado, existen fórmulas de conversión entre cada uno de los modelos vistos anteriormente.

1.2.4.1. Paleta de Colores.

Cada imagen tiene asociada una paleta de color única y diferente a las demás; esta paleta se obtiene a partir de una tabla interna del software donde a cada *pixel* se le asocia un color de acuerdo a la intensidad que esté adquiera. El tamaño de esa tabla puede variar de sistema en sistema y de acuerdo a la cantidad de bits utilizados en esa tabla se tiene un conjunto finito de colores, por ejemplo, en los sistemas actuales se dedican 8 bits para crear un universo de 256 colores diferentes y para crear una paleta de colores para una imagen se seleccionan ciertos colores de ese universo de tal forma que sean los convenientes para que la imagen se aproxime a la realidad lo más posible.

Como una breve conclusión, dependiendo del *hardware* y *software*, así como de los requerimientos de graficación, cada uno de los anteriores conceptos proporciona beneficios y dependerá del usuario final el uso de estas

definiciones, ya que la mayoría de los paquetes de graficación ofrecen estas y otras herramientas.

1.3. Transformación de Coordenadas.

Las aplicaciones gráficas requieren aplicar transformaciones de coordenadas para tener una mejor vista de los objetos desplegados. Básicamente existen tres transformaciones que son: la translación, rotación y escalamiento. Algunos autores especializados en el tema introducen dos transformaciones más, la reflexión y el corte. Sin embargo, en este trabajo de investigación sólo tratará las tres primeras. La translación consiste en cambiar la posición original de un objeto geométrico a otra posición arbitraria. La rotación consiste en rotar el objeto geométrico un cierto ángulo a partir de un punto de referencia. Y, por último, el escalamiento consiste en modificar el tamaño del cuerpo geométrico.

Para empezar es necesario definir que es una transformación, así pues, en términos matemáticos, la transformación es una correspondencia uno a uno de un conjunto de puntos, es decir, a cada punto del objeto original sólo le corresponde un punto del objeto transformado, de esta manera, una transformación implica modificar la posición, orientación o forma del objeto.

Las transformaciones pueden aplicarse a cualquier tipo de geometrías usando únicamente su representación paramétrica. Para simplificar las operaciones de transformación sobre la geometría de los objetos, es necesario utilizar matrices que contengan los valores paramétricos de las geometrías y reducirlas a una sola matriz de transformación. Dependiendo del espacio *n* dimensional[2] se necesitarán $n+1$ elementos para llevar a cabo la transformación, a estos elementos se les conoce como coordenadas homogéneas y deben estar contenidos dentro de un vector, al cual se le denomina vector homogéneo y su única restricción es que no pueden contener

[2] Un espacio *n* dimensional es un concepto abstracto e idealizado para generalizar el concepto del espacio, sin embargo, puede decirse que un espacio de esta naturaleza es aquel donde se necesitan *n* números de ejes perpendiculares entre sí para representar la posición de un punto en ese espacio.

valores ceros como elementos. Así, para un espacio tridimensional será necesario un vector con 4 coordenadas homogéneas, a decir, un punto cualquiera en el espacio tridimensional será representado como en la ecuación 1.1:

$$[xh,\ yh,\ zh,\ h] \qquad \textbf{(Ec. 1.1)}$$

Donde xh, yh, y zh representan la posición de un punto en el espacio tridimensional, multiplicado por una constante h que representa un factor de homogeneidad y cuyo valor, por lo regular, es igual a la unidad. La normalización del vector homogéneo, elimina el factor multiplicativo para regresar a la representación original, tal como se muestra en la ecuación 1.2:

$$[xh / h,\ \ yh / h,\ zh / h,\ h / h] \qquad \textbf{(Ec. 1.2)}$$

En los desarrollos subsiguientes de esta sección tomaré como valor de h a la unidad, ya que esto nos evitará confusiones y cálculos innecesarios, además de que la mayoría de los paquetes de graficación toman, por convención, este valor.

1.3.1. Translación.

La translación de un objeto geométrico implica mover a cada uno de los puntos que lo conforman en igual magnitud y dirección de un vector T dado, entonces, cada punto P será movido por un monto T para obtener una nueva posición P', representada en la ecuación 1.3:

$$P' = P + T \qquad \textbf{(Ec. 1.3)}$$

Debido a que cada punto debe ser representado como un vector homogéneo, para un espacio bidimensional, los puntos P y P' serán representados como en las ecuaciones 1.4 y 1.5, respectivamente:

$$P = [x,\ y,\ 1] \qquad \textbf{(Ec. 1.4)}$$
$$P' = [x',\ y',\ 1] \qquad \textbf{(Ec. 1.5)}$$

Donde x y y representan la posición de un punto con su factor de homogeneidad correspondiente. En un espacio tridimensional, las ecuaciones 1.6 y 1.7 muestran los vectores homogéneos correspondientes a P y P':

$$P = [\, x,\ y,\ z,\ 1 \,] \qquad \textbf{(Ec. 1.6)}$$
$$P' = [\, x',\ y',\ z',\ 1 \,] \qquad \textbf{(Ec. 1.7)}$$

Dado que ahora los puntos se transformaron en vectores de $n + 1$ elementos es necesario encontrar una matriz de translación tal que cumpla con la ecuación 1.8:

$$P' = P * T \quad \textbf{(Ec. 1.8)}$$

Para un espacio bidimensional la matriz de translación está dada la matriz 1.1:

$$T = \begin{vmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ T_x & T_y & 1 \end{vmatrix}$$

Matriz 1.1

Sustituyendo las ecuaciones 1.4 y 1.5 y la matriz 1.1. en la ecuación 1.8, se tiene:

$$[\, x,\ y,\ 1 \,] * T = [\, x + T_x,\ y + T_y,\ 1 \,] = [\, x',\ y',\ 1 \,] \quad \textbf{(Ec. 1.9)}$$

De esta manera, cada punto que conforma dicho objeto geométrico es desplazado a otra posición de acuerdo a la cantidad especificada por el vector T. Para regresar a la posición original es necesario reemplazar los valores de T_x y T_y por sus contrapartes negativas, por ejemplo, los valores $-T_x$ y $-T_y$, respectivamente. En el caso de espacios tridimensionales, la matriz 1.1 se escribe como:

$$T = \begin{vmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ T_x & T_y & T_z & 1 \end{vmatrix}$$

Matriz 1.2

Análogamente, sustituyendo las ecuaciones 1.6 y 1.7 y la matriz 1.2. en la ecuación 1.8, se tiene:

$$[x,\ y,\ z,\ 1] * T = [x + T_x,\ y + T_y,\ z + T_z,\ 1] = [x',\ y',\ z',\ 1]$$
(Ec. 1.10)

La translación de cuerpos geométricos es la transformación de coordenadas más simple y sencilla de realizar, debido a su propia naturaleza.

1.3.2. Rotación.

La transformación de rotación es una de las más delicadas, debido a que hay una serie de reglas que considerar al momento de aplicar esta operación, así pues, las rotaciones de los objetos geométricos deben considerar que las distancias y los ángulos entre los puntos e inclinaciones deben conservarse aún después de esta transformación. Se debe tomar en cuenta la regla de la mano derecha[3] para obtener los signos de los ángulos de rotación, la cual establece que:

- Si el eje de rotación es X, entonces la dirección positiva de rotación es del eje Y al eje Z.
- Si el eje de rotación es Y, entonces la dirección positiva de rotación es del eje Z al eje X.
- Si el eje de rotación es Z, entonces la dirección positiva de rotación es del eje X al eje Y.

Las rotaciones alrededor del origen tienen tres componentes, a decir, α, β y θ donde α representa el ángulo de rotación sobre el eje X, β sobre el eje Y, y θ sobre el eje Z, siempre y cuando se opere sobre un espacio tridimensional. El orden en como se aplican las rotaciones al objeto geométrico por los tres componentes es muy importante. La regla general indica que:

[3] La regla de la mano derecha, en el campo de estudio de la Electricidad y el Magnetismo, indica que los dedos, excepto el pulgar, apuntan en el mismo sentido que el campo magnético, mientras que el pulgar apunta en el sentido de la corriente y, por último, la palma indica el sentido de la fuerza.

- Primero se rota alrededor del eje Z si θ es diferente de cero.
- Después se rota sobre el eje Y si β es diferente de cero.
- Por último, se rota sobre le eje X si α es diferente de cero.

En el caso particular de que uno o dos de estos ángulos sea igual a cero, entonces se aplica la regla con mayor precedencia, únicamente si el ángulo correspondiente es diferente de cero. Por ejemplo, si θ=0, entonces se debe rotar primero sobre β y después sobre α, de esta manera, no se violentan las reglas anteriores. La ecuación de rotación de un punto P se define como:

$$P' = P * R \qquad \textbf{(Ec. 1.11)}$$

Ahora, si convertimos al punto P a su respectivas coordenadas homogéneas, dentro de un espacio tridimensional, obtenemos que:

$$[x',\ y',\ z',\ 1] = [x,\ y,\ z,\ 1] * R \qquad \textbf{(Ec. 1.12)}$$

Donde R es la matriz de rotación que se desea aplicar y está compuesta por tres matrices de rotación con respecto a cada eje del espacio tridimensional, es decir, existe una matriz para la rotación sobre el eje X, otra matriz para la rotación sobre eje Y, y otra más para la rotación sobre el eje Z. La conjunción de estas tres matrices nos dará como resultado la matriz de rotación general R. Así pues, la matriz de rotación sobre el eje X en un ángulo α está definida por la matriz 1.3:

$$R_x(\alpha) = \begin{vmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\alpha & \text{sen}\alpha & 0 \\ 0 & -\text{sen}\alpha & \cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix}$$

Matriz 1.3

La matriz de rotación sobre el eje Y está dada por el ángulo β y es definida por la matriz 1.4:

$$R_y(\beta) = \begin{vmatrix} \cos\beta & 0 & -\text{sen}\beta & 0 \\ 0 & 1 & 0 & 0 \\ \text{sen}\beta & 0 & \cos\beta & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix}$$

Matriz 1.4

La matriz de rotación sobre el eje Z sobre un ángulo θ, es definida por la matriz 1.5:

$$R_z(\theta) = \begin{vmatrix} \cos & \text{sen} & 0 & 0 \\ -\text{sen}\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix}$$

Matriz 1.5

Se tiene pues que la matriz de rotación general está dada por el producto de las matrices 1.3, 1.4 y 1.5, tal como lo muestra la ecuación 1.13:

$$R = R_x(\alpha) * R_y(\beta) * R_z(\theta) \qquad \textbf{(Ec. 1.13)}$$

Expresada en forma matricial:

$$R(\alpha, \beta, \theta) = \begin{vmatrix} \cos\theta\cos\beta & -\text{sen}\theta & \cos\theta\,\text{sen}\beta \\ \text{sen}\theta\cos\beta\cos\alpha - \text{sen}\beta\,\text{sen}\alpha & \cos\theta\cos\alpha & \text{sen}\theta\,\text{sen}\beta\cos\alpha + \cos\beta\,\text{sen}\alpha \\ -\text{sen}\theta\cos\beta\,\text{sen}\alpha - \text{sen}\beta\cos\alpha & -\cos\theta\,\text{sen}\alpha & -\text{sen}\theta\,\text{sen}\beta\,\text{sen}\alpha + \cos\beta\cos\alpha \end{vmatrix}$$

Matriz 1.6

Las inversas de las matrices de rotación 1.3, 1.4 y 1.5, se obtiene sustituyendo el ángulo de rotación por su contraparte negativa, de igual manera, la matriz identidad de cada una de ella se obtiene cuando los ángulos de rotación respectivos son iguales a cero. Ahora sabemos los pasos necesarios para llevar a cabo la rotación de un cuerpo geométrico, pero ¿estas ecuaciones sirven cuando se desea rotar un cuerpo geométrico alrededor de un eje arbitrario? Si, pero es necesario realizar algunas operaciones antes. Primero se debe trasladar y rotar el eje arbitrario de tal forma que coincida con algún eje

del sistema coordenado, después se rota un ángulo γ para después rotar y trasladar el eje arbitrario a su posición original, sin afectar su orientación. Este proceso consta de cinco pasos:

1. Se debe trasladar el eje arbitrario y el objeto geométrico de tal forma que el eje arbitrario coincida con algún eje del sistema cartesiano, y se cumpla con la expresión; $P' = P * T_1$.

2. Se debe rotar el objeto geométrico y el eje arbitrario de tal manera que esté último sea colineal al eje X. Para ello es necesario rotar $-\theta$ en el eje Z y $-\beta$ en el eje Y. De acuerdo a la siguiente figura 1.1, se deduce que $\theta = tan^{-1}(y_2 - y_1 / x_2 - x_1)$ y $\beta = sin^{-1}(z_2 - z_1 / T_2 - T_1)$.

3. Entonces, tenemos que $P' = P * T_1 * R_{-\theta} * R_{-\beta}$.

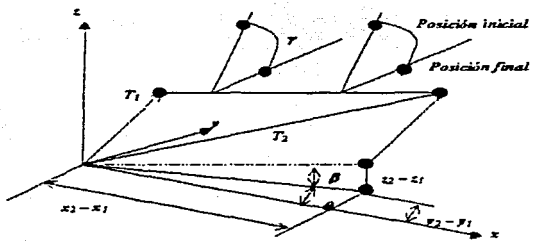


Figura 1.1. Proceso de rotación con eje arbitrario

4. Realizar la rotación γ grados haciendo que: $\alpha = \gamma$, por convención, entonces $P' = P * T_1 * R_{-\theta} * R_{-\beta} * R_{\alpha}$.

5. Repetir el paso 2, pero esta vez invirtiendo el signo de los ángulos θ y β, resultando en la siguiente expresión: $P' = P * T_1 * R_{-\theta} * R_{-\beta} * R_{\alpha} * R_{\theta} * R_{\beta}$.

6. Finalmente, aplicar la primer translación, pero ahora con los signos opuestos con la finalidad de regresar al eje arbitrario y el objeto geométrico a su posición original. Así, la expresión completa queda como sigue: $P' = P * T_1 * R_{-\theta} * R_{-\beta} * R_{\alpha} * R_{\theta} * R_{\beta} * T_{-1}$.

Es importante hacer notar que el orden de ejecución de cada una de estas operaciones es de vital importancia, ya que con sólo una modificación, los resultados pueden ser absurdos e inesperados. Aclarado este punto, es necesario definir las matrices de rotación del espacio bidimensional. La matriz de rotación general para el espacio de dos dimensiones está dada por la matriz 1.8.

$$R = \begin{vmatrix} cos\theta & sen\theta & 0 \\ -sen\theta & cos\theta & 0 \\ 0 & 0 & 1 \end{vmatrix}$$

Matriz 1.7

Entonces, para cada punto del espacio bidimensional, en coordenadas homogéneas, $P = [x, y, 1]$, podemos obtener su rotación, $P' = [x', y', 1]$, a través de la ecuación 1.14:

$$P' = P * R \qquad \textbf{(Ec. 1.14)}$$

Donde R representa la matriz de rotación siempre y cuando el centro de rotación coincida con el origen del sistema coordenado, pero ¿qué pasa si se elige un punto arbitrario como centro de rotación?. Para ello es necesario realizar esencialmente los mismos pasos que cuando la rotación se realiza usando un eje arbitrario, hablando en el espacio tridimensional, así, tenemos que trasladar el punto de rotación al origen del sistema cartesiano, aplicar la rotación y regresar el punto de rotación a su sistema original. Basado en lo anterior, podemos afirmar que:

$$P' = P * T^{-1} * R * T \qquad \textbf{(Ec. 1.15)}$$

Donde $T = [T_x, T_y, 1]$ representa las coordenadas homogéneas del punto de rotación. A este tipo de rotaciones se le conoce como rotación con punto pivote. Matricialmente, la formulación anterior quedaría como sigue:

$$P' = P * \begin{vmatrix} cos\theta & sen\theta & 0 \\ -sen\theta & cos\theta & 0 \\ T_x(1-cos\theta) - T_y sen\theta & T_y(1-cos\theta) - T_x sen\theta & 1 \end{vmatrix}$$

Matriz 1.8

Como hemos visto, la rotación de cuerpos geométricos implica una serie de pasos que no permiten la conmutatividad entre sus miembros, no obstante, estas matrices de rotación poseen la propiedad de la ortogonalidad, es decir, representan vectores unitarios mutuamente perpendiculares y la determinante es la unidad. Cualquier objeto geométrico que sea sometido a estas matrices preserva sus distancias y ángulos, por esta razón, se denomina transformaciones de cuerpos rígidos, ya que cualquier objeto geométrico rotado no se deforma.

1.3.3. Escalamiento.

La transformación de escalamiento consiste en aplicar un mismo factor de escala a cada uno de los coeficientes geométricos del objeto a fin de modificar el tamaño del mismo de manera uniforme sin afectar la forma del mismo. No obstante, lo anterior no es una regla general, ya que es posible aplicar diferentes tipos de factores de escala a fin de afectar el tamaño y forma del objeto.

Cuando esta transformación es aplicada a un objeto geométrico, el punto de referencia puede ser el centro del objeto o algún otro punto del espacio cartesiano. En el primer caso, el centro del cuerpo geométrico permanece invariable, mientras que los demás puntos del objeto pierden su posición original, a menos que el factor de escala sea la matriz identidad *I*. En el segundo caso, todos los puntos del cuerpo geométrico pierden su posición original incluyendo el centro del mismo. El escalamiento se define en la ecuación 1.16:

$$P' = P * S \qquad \textbf{(Ec. 1.16)}$$

El factor de escala siempre debe ser mayor a 0, de lo contrario obtendremos la reflexión del objeto en vez de su escalamiento. Nuevamente tendrá que

recurrir a la homogenización de las coordenadas de cada punto del objeto para obtener una matriz de escalamiento que satisfaga, para un espacio bidimensional, la ecuación 1.17 satisface la representación:

$$[x',\ y',\ 1] = [x,\ y,\ 1] * S \qquad \textbf{(Ec. 1.17)}$$

Así pues, la matriz de escalamiento está dada por:

$$S = \begin{vmatrix} S_x & 0 & 0 \\ 0 & S_y & 0 \\ 0 & 0 & 1 \end{vmatrix}$$

Matriz 1.9

Sustituyendo la matriz 1.9 en la ecuación 1.17 se tiene que:

$$[x',\ y',\ 1] = [S_x x,\ S_y y,\ 1] = [x,\ y,\ 1] * S \qquad \textbf{(Ec. 1.18)}$$

Para llevar a cabo la inversión del escalamiento, es necesario sustituir a S_x y S_y en la ecuación 1.18 por $1/S_x$ y $1/S_y$, respectivamente. De igual manera, la matriz identidad se obtiene haciendo $S_x = S_y = 1$. La anterior ecuación sirve para el caso en que el centro del escalamiento coincide con el origen del objeto. Para el caso en que el punto de referencia del escalamiento sea arbitrario es necesario llevar a cabo algunas modificaciones a la matriz de escalamiento.

Dado un punto de referencia $P_r = [x_r,\ y_r,\ 1]$ es necesario trasladar este punto al centro del cuerpo geométrico, aplicar el escalamiento y después restaurar el punto P_r a su posición original, esto equivale a aplicar una translación, escalamiento y finalmente una translación inversa a la primera, tal como lo describen las siguientes matrices:

$$P' = P * \begin{vmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -x_r & -y_r & 1 \end{vmatrix} * \begin{vmatrix} S_x & 0 & 0 \\ 0 & S_y & 0 \\ 0 & 0 & 1 \end{vmatrix} * \begin{vmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ x_r & y_r & 1 \end{vmatrix}$$

$$P' = P * \begin{vmatrix} S_x & 0 & 0 \\ 0 & S_y & 0 \\ x_r(1-S_x) & y_r(1-S_y) & 1 \end{vmatrix}$$

Matriz 1.10

Como podrá haberse dado cuenta si el punto de referencia es *[0, 0, 1]*, entonces las anteriores ecuaciones se reducen a la matriz original de escalamiento. Análogamente, para el espacio tridimensional, la matriz de escalamiento 1.11, está dada por:

$$S = \begin{vmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix}$$

Matriz 1.11

La matriz 1.12 sólo es válida cuando el centro del escalamiento coincide con el centro del objeto. Cuando el punto de referencia del escalamiento es diferente al origen, $P_r = [x_r, y_r, z_r, 1]$, tenemos que:

$$P' = P * \begin{vmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -x_r & -y_r & -z_r & 1 \end{vmatrix} * \begin{vmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix} * \begin{vmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ x_r & y_r & z_r & 1 \end{vmatrix}$$

$$P' = P * \begin{vmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ x_r(1-S_x) & y_r(1-S_y) & z_r(1-S_z) & 1 \end{vmatrix}$$

Matriz 1.12

Cuando un objeto geométrico es transformado por alguna de las matrices vistas anteriormente, se dice que el objeto ha sido transformado por un conjunto de transformaciones *afines*, es decir, que el objeto conserva el paralelismo entre sus líneas, no así los valores originales de sus distancias y ángulos, como en el caso de la ortogonalidad.

1.4. Primitivas de Dibujo.

1.4.1. Puntos y Líneas.

La mayoría de los autores[4] concuerdan con la versión de que un punto es una posición en el espacio y cuya característica principal es que no tiene forma, dimensión y no es algo tangible, y por tal razón es infinitesimal, así pues, no es posible distinguir un punto de otro, la única diferencia, si así se desea ver, es la distancia que existe entre dos puntos. Basado en la anterior definición, puedo afirmar que la representación gráfica de un punto corresponde a un *pixel*, que es la unidad mínima de la graficación. Dentro de este contexto, un *pixel* no tiene forma, pero si tiene tamaño.

La siguiente primitiva básica de dibujo es la recta o línea, cuya definición corresponde a una sucesión de puntos localizados dentro de un punto inicial y uno final y estos definen la dirección de la misma. En un monitor de CRT, una línea es una sucesión de *píxeles* calculados usando la ecuación de la recta, empero, la resolución del monitor influye mucho en el trazado de esta primitiva, debido a que se traza sobre espacios rectangulares y puede originar efectos de *aliasing*[5]. Existen diversos métodos para graficar una recta sobre un dispositivo de despliegue, entre ellos destaca el algoritmo de *Bresenham* y el algoritmo diferencial. Ambos algoritmos parten de la ecuación fundamental de la recta:

$$y = m\,x + b \qquad \textbf{(Ec. 1.19)}$$

Donde m representa la pendiente de la recta y b la intersección sobre el eje Y. Los extremos de una recta figuran, en un plano bidimensional, con los valores (x_0, y_0) y (x_1, y_1), así pues, se puede calcular los valores correspondientes a m y b a través de las siguientes fórmulas:

$$m = (y_1 - y_0) \cdot (x_1 - x_0) \qquad \textbf{(Ec. 1.20)}$$

---

[4] BRODLY, K.W. y EARNSHAW, Ray

[5] En la sección 1.6 de este capítulo se profundiza sobre este tema.

$$b = y_0 - m\,x_0 \qquad \textbf{(Ec. 1.21)}$$

1.4.1.1. Algoritmo de Bresenham.

El algoritmo de *Bresenham* precisa la generación de líneas de rastreo mediante la utilización de cálculos increméntales con enteros. Para ilustrar el planteamiento de *Bresenham*, se debe considerar el proceso para líneas con pendiente positiva menor que 1. Las posiciones de cada *pixel* a lo largo de la trayectoria de la línea se determinan con el uso de parámetros de decisión. Si se inicia desde el extremo izquierdo (x_0, y_0) los cálculos de Δx y de Δy se determinan con las ecuaciones 1.22, 1.23 y 1.24:

$$\Delta x = (x_1 - x_0) \qquad \textbf{(Ec. 1.22)}$$
$$\Delta y = (y_1 - y_0) \qquad \textbf{(Ec. 1.23)}$$
$$\Delta y = m\,\Delta x \quad \text{ó} \quad \Delta x = m\,\Delta y \qquad \textbf{(Ec. 1.23)}$$

Para calcular la siguiente coordenada de *pixel* se necesita un parámetro de decisión p_0 dado en la ecuación 1.25.

$$p_0 = 2\Delta y - \Delta x \qquad \textbf{(Ec. 1.25)}$$

Si $p_0 > 0$, entonces la siguiente coordenada de *pixel* será, $(x+1, y+1)$ y el siguiente parámetro de decisión está dado por la ecuación 1.26.

$$p_{k+1} = p_k + 2\Delta y - \Delta x \qquad \textbf{(Ec. 1.26)}$$

Donde $k = 0$ en este primer paso. En caso de que $p_0 < 0$, entonces la siguiente coordenada de *pixel* será, $(x+1, y)$ y el siguiente parámetro de decisión está dado por la ecuación 1.27.

$$p_{k+1} = p_k + 2\Delta y \qquad \textbf{(Ec. 1.27)}$$

Para las siguientes coordenadas de *pixel*, los parámetros de decisión se calculan con las ecuaciones 1.26 y 1.27, dependiendo del caso. El algoritmo continua hasta que se alcanza el punto (x_1, y_1). La otra situación que puede ocurrir es cuando la pendiente es mayor a 1. En este caso, se realizan los mismo cálculos con las ecuaciones 1.22 y 1.23, sin embargo, el primer

parámetro de decisión, p_0, difiere en el cálculo. La ecuación 1.28 muestra la fórmula para calcular este parámetro.

$$p_0 = 2\Delta x - \Delta y \qquad \textbf{(Ec. 1.28)}$$

Ahora, aplicamos las mismas reglas que en el primer caso, es decir, si $p_0 > 0$, entonces la siguiente coordenada de *pixel* será, $(x+1, y+1)$ y el siguiente parámetro de decisión está dado por la ecuación 1.29.

$$p_{k+1} = p_k + 2\Delta x - \Delta y \qquad \textbf{(Ec. 1.29)}$$

En caso contrario, la siguiente coordenada de *pixel* será, $(x+1, y)$ y el siguiente parámetro de decisión está dado por la ecuación 1.30.

$$p_{k+1} = p_k + 2\Delta x \qquad \textbf{(Ec. 1.30)}$$

Aunque el algoritmo de *Bresenham* es muy fácil de implementar se pueden dar algunos casos que pueden inducir en confusiones, por tal razón, en las siguientes dos tablas muestro todos los posibles casos que pueden ocurrir cuando los incrementos de Δx y de Δy son positivos o negativos.

Líneas con pendiente menor a 1						
			$P_k > 0$		$P_k < 0$	
Δx	Δy	P_0	*Pixel*	P_{k+1}	*pixel*	P_{k+1}
+	+	$2\Delta y - \Delta x$	$(x+1, y+1)$	$p_k + 2\Delta y - 2\Delta x$	$(x+1, y)$	$p_k + 2\Delta y$
+	-	$2\Delta y - \Delta x$	$(x+1, y-1)$	$p_k + 2\Delta y - 2\Delta x$	$(x+1, y)$	$p_k + 2\Delta y$
-	+	$2\Delta y - \Delta x$	$(x-1, y+1)$	$p_k + 2\Delta y - 2\Delta x$	$(x-1, y)$	$p_k + 2\Delta y$
-	-	$2\Delta y - \Delta x$	$(x-1, y-1)$	$p_k + 2\Delta y - 2\Delta x$	$(x-1, y)$	$p_k + 2\Delta y$
Líneas con pendiente mayor a 1						
			$P_k > 0$		$P_k < 0$	
Δx	Δy	P_0	*Pixel*	P_{k+1}	*pixel*	P_{k+1}
+	+	$2\Delta x - \Delta y$	$(x+1, y+1)$	$p_k + 2\Delta x - 2\Delta y$	$(x+1, y)$	$p_k + 2\Delta x$

-	-	$2\Delta x - \Delta y$	$(x-1, y-1)$	$p_k + 2\Delta x - 2\Delta y$	$(x-1, y)$	$p_k + 2\Delta x$
-	-	$2\Delta x - \Delta y$	$(x-1, y-1)$	$p_k + 2\Delta x - 2\Delta y$	$(x-1, y)$	$p_k + 2\Delta x$
-	-	$2\Delta x - \Delta y$	$(x-1, y-1)$	$p_k + 2\Delta x - 2\Delta y$	$(x-1, y)$	$p_k + 2\Delta x$

Tabla 1.1.

Como puede verse en la tabla 1.1 se contemplan todos los posibles casos de trazo de lineas, aun cuando la pendiente sea negativa.

1.4.1.2. Algoritmo Analizador Digital Diferencial.

El Analizador Diferencial Digital (*DDA, Digital Differential Analyzer*) es un algoritmo de línea de conversión de rastreo que se basa en el cálculo para Δy o Δx por medio de las ecuaciones siguientes:

$$\Delta y = m * \Delta x \qquad \textbf{(Ec. 1.31)}$$
$$\Delta x = \Delta y / m \qquad \textbf{(Ec. 1.32)}$$

En este caso, se efectúa un muestreo de línea en intervalos unitarios en una coordenada y determinamos los valores enteros correspondientes más próximos a la trayectoria de la línea para la otra coordenada. Si la pendiente es menor o igual que 1, se lleva a cabo un muestreo de x en intervalos unitarios ($\Delta x = 1$) y se calcula cada valor sucesivo de y como:

$$y_{k+1} = y_k + m \qquad \textbf{(Ec. 1.33)}$$

El subindice k toma valores enteros a partir de la unidad y aumenta a razón de 1 hasta que se alcanza el valor final. Ya que m puede ser cualquier número real entre 0 y 1, los valores calculados de y deben redondearse al entero más cercano.

Para líneas como una pendiente positiva mayor a 1 se realiza un muestreo de y en intervalos unitarios, $\Delta y = 1$, y se calcula cada valor sucesivo de x como:

$$x_{k+1} = x_k + 1/m \qquad \textbf{(Ec. 1.34)}$$

Las anteriores ecuaciones se basan en la suposición de que las líneas deban procesarse del extremo izquierdo al derecho. Si este procesamiento se revierte, de manera que sea el extremo derecho donde se inicia, entonces tenemos que $\Delta x = -1$ y

$$y_{k+1} = y_k - m \qquad \textbf{(Ec. 1.35)}$$

o, cuando la pendiente es mayor a 1, $\Delta y = -1$ con:

$$x_{k+1} = x_k - 1/m \qquad \textbf{(Ec. 1.36)}$$

Dichas ecuaciones también se pueden utilizar para calcular posiciones de *píxel* a lo largo de la línea con una pendiente negativa. Si el valor absoluto de la pendiente es menor a 1 y el extremo en que comienza es el izquierdo, se determina que $\Delta x=1$ y se calculan los valores de y con la ecuación 1.33. Cuando se empieza en el extremo derecho, para la misma pendiente, $\Delta x=-1$ y se obtienen los valores de y mediante la ecuación 1.35. De modo similar, cuando el valor absoluto de la pendiente negativa es mayor a 1, entonces se utiliza $\Delta y=-1$ y la ecuación 1.36 o bien, $\Delta y=1$ y la ecuación 1.34.

1.4.1.3. Atributos.

Las líneas tienen tres atributos que las caracterizan, y que determinarán la forma en que serán desplegadas, y, por tanto, afectarán a la imagen correspondiente. Estos atributos son el color, la anchura y el estilo. El color de una línea se puede indicar mediante instrucciones de software, solamente si esté tiene al menos una tabla de colores. La anchura de la línea que, también, se puede especificar mediante software, se refiere al grosor de la misma. Por último, el estilo de la línea se refiere a la forma en como será dibujada; habitualmente, es una recta continua, pero algunos paquetes de dibujo, permiten al usuario elegir entre varios modelos de líneas, por ejemplo, líneas punteadas, líneas entrecortadas, etc.

El uso de algoritmos eficientes para el trazo de primitivas, la línea en este caso, es muy importante ya que las imágenes vectoriales dependen de ella y de la información que puedan proporcionar a través de sus atributos.

1.4.2. Curvas.

1.4.2.1. Circunferencia.

Una circunferencia se define como un conjunto de puntos que se encuentran, en su totalidad, a una distancia determinada de una posición central (x_c, y_c), esta relación de distancia se expresa por medio del teorema de Pitágoras como:

$$(x - x_c)^2 + (y - y_c)^2 = r^2 \qquad \textbf{(Ec. 1.37)}$$

Se puede utilizar esta ecuación para calcular los puntos a lo largo del eje X en pasos unitarios de $x_c - r$ a $x_c + r$ y calcular los valores correspondientes del eje Y en cada posición como:

$$y = y_c \pm [r^2 - (x_c - x)^2]^{1/2} \qquad \textbf{(Ec. 1.38)}$$

No obstante, no es el mejor método para calcular una circunferencia, ya que implica cálculos considerables en cada paso, además, el espacio entre las posiciones del *píxel* trazadas no es uniforme. Para eliminar el espacio irregular, es necesario calcular los puntos a lo largo de la frontera circular utilizando las coordenadas polares θ y r, dadas en las ecuaciones:

$$x = x_c + r\cos\theta \qquad \textbf{(Ec. 1.39)}$$
$$y = y_c + r\,sen\theta \qquad \textbf{(Ec. 1.40)}$$

Cuando un despliegue se realiza con estas ecuaciones se utiliza un tamaño angular fijo, una circunferencia se traza con puntos equidistantes a través de la misma. El tamaño de paso de θ depende de la aplicación y del dispositivo de despliegue. En la tabla 1.2 se muestra un ejemplo usando las coordenadas polares. En este caso se considera que el círculo esta en el centro y que el radio es de 5 unidades.

θ	X	Y
0	5	0
1	4.99	0.08

2	4.99	0.17
10	4.92	0.86
30	4.33	2.5
45	3.53	3.53
60	2.5	4.33
80	0.86	4.92
90	0	5

Tabla 1.2

En el ejemplo de la tabla 1.2, los paso de θ se consideraron en radianes y los incrementos fueron unitarios. En este caso, sólo se calculó el cuadrante 1 del plano cartesiano, no obstante es posible calcular los demás cuadrantes con el uso de simetrías, así el cuadrante dos será *(-x, y)*; el cuadrante tres será *(-x, -y)*; y el cuadrante cuarto es *(x, -y)*.

Sin embargo, en ambos casos las operaciones son demasiado costosas en cuanto al procesamiento que realizan, por ello, fue modificado el algoritmo de *Bresenham* para adecuarlo al cálculo de circunferencias, de esta forma, el trazo de una circunferencia no implica demasiado cálculos. El algoritmo parte del supuesto de que la circunferencia posee un radio R con un centro ubicado en el origen del sistema, *(0, 0)*. El primer paso de este algoritmo consiste en calcular el primer parámetro de decisión, tal que:

$$p_1 = 3 - 2R$$

Los valores correspondientes a x_1 y y_1 son los mismos que el centro de la circunferencia, mientras que los valores x_2 y y_2 se calculan con las siguientes ecuaciones:

$$x_2 = x_1 + 1$$
$$y_2 = R$$

Inmediatamente después, entramos a un ciclo hasta que se cumpla que $x > y$, así calculamos el siguiente paso de x con la siguiente ecuación:

$$x_{k+1} = x_k + 1$$

Dentro del ciclo, calculamos el siguiente parámetro de decisión siguiendo las siguientes reglas:

- Si $p_i \geq 0$, entonces $y_{k-1} = y_k - 1$ y el parámetro de decisión es: $p_{k-1} = 4(x_k - y_k) + p_k + 10$.
- En caso contrario, $y_{k-1} = y_k$ y el parámetro de decisión es: $p_{k-1} = 4x_k + p_k - 6$.

Como habrá notado este algoritmo sólo calcula una parte de la circunferencia ya que el resto se obtiene por simetría, es decir, por cada paso de x_k y de y_k podemos calcular el resto con las siguientes coordenadas: $(-x_k, y_k)$, $(x_k, -y_k)$ y $(-x_k, -y_k)$.

1.4.2.2. Elipse.

La elipse se define, matemáticamente, como un conjunto de puntos cuya suma de las distancias desde dos posiciones fijas (focos) sea la misma para todos los puntos. La ecuación de la elipse está dada por la ecuación 1.41.

$$[(x - x_c) / r_x]^2 + [(y - y_c) / r_y]^2 = 1 \qquad \textbf{(Ec. 1.41)}$$

Para fines prácticos la elipse se puede ver como una circunferencia alargada, y modificando el primer algoritmo descrito para la circunferencia, pero en esta ocasión para la elipse. Partiendo de la ecuación 1.41 podemos afirmar que:

$$y = y_c \pm r_y * [1 - ((x - x_c) / r_x)]^{1/2} \qquad \textbf{(Ec. 1.42)}$$

Donde r_x y r_y, representan el radio mayor y radio menor, respectivamente. Las coordenadas x_c y y_c pertenecen al centro de la elipse. En caso de que el centro de la elipse coincida con el origen del sistema cartesiano, entonces las ecuaciones 1.41 y 1.42 se reducen a las siguientes expresiones:

$$[x / r_x]^2 + [y / r_y]^2 = 1 \qquad \textbf{(Ec. 1.43)}$$
$$y = \pm r_y * [1 - (x / r_x)]^{1/2} \qquad \textbf{(Ec. 1.44)}$$

Con la ecuación 1.44 es posible determinar los valores correspondientes a y con incrementos unitarios de x, tal que $\Delta x = 1$, iniciando desde $x_c - r_x$ hasta finalizar en $x_c - r_x$. Sin embargo, este método trae consigo los mismos problemas que con las circunferencias, esto es, los espacios entre cada coordenada y son muy espaciados y el trazo de la elipse se ve demasiado

afectada. Para resolver este problema podemos utilizar las coordenadas polares para la elipse, definidas en las siguientes expresiones:

$$x_k = x_c + r_x \cos\theta_k \qquad \textbf{(Ec. 1.45)}$$
$$y_k = y_c + r_y \operatorname{sen}\theta_k \qquad \textbf{(Ec. 1.46)}$$

Los incrementos de θ_k pueden ser dados en radianes o en grados. En cualquier caso el cálculo de la coordenada y tiene una apariencia uniforme.

1.5. Superficies.

El modelado de superficies resulta de vital importancia para la graficación por computadora, ya que la mayoría de los cuerpos geométricos se pueden ver como superficies o formas geométricas menos complejas, ello posibilita el despliegue gráfico en tiempo real. Empero, hay que considerar que las superficies de los cuerpos geométricos sean totalmente bidimensionales, es decir, que no existan puntos singulares donde la superficie tenga una intersección consigo misma o se abra en varias hojas. Existen varias técnicas para representar cuerpos geométricos a través de superficies, que analizaremos a continuación.

1.5.1. Superficies con Mallas Poligonales.

Las superficies compuestas por mallas poligonales son un conjunto de vértices, aristas y polígonos conectados de tal manera que un vértice puede estar compartido por más de una arista, y a su vez, una arista puede formar parte de dos o más polígonos. Esta propiedad les otorga la facilidad de representar superficies planas con gran exactitud, sin embargo, si la superficie presenta algunas curvas, este método sólo puede dar una aproximación presentando efectos de *aliasing*. Son cinco las operaciones algoritmicas que se deben realizar para implementar este método:

1. Encontrar las aristas incidentes a un vértice.
2. Encontrar los polígonos que comparten una arista o un vértice.
3. Encontrar los vértices conectados por una arista (deben ser dos únicamente).
4. Encontrar las aristas de un polígono.
5. Identificar los errores de representación.

De igual manera, se deben tomar ciertas reglas al momento de realizar el despliegue de la superficie, esto es a fin de evitar incoherencias u hoyos en la misma. Estas normas son:

1. Verificar que todos los polígonos de la superficie estén cerrados.
2. Verificar que todas las aristas sean utilizada al menos una vez, pero menos que el máximo número de aristas que componen a un polígono individual.
3. Verificar que todos los vértices sean referenciados por, al menos, dos aristas.

Existe varias formas de implementar este tipo de superficies, cada una de ellas con sus correspondientes ventajas y desventajas. El más eficiente consiste en crear una lista de vértices y una lista de aristas, esta última se compone por dos apuntadores hacia la lista de vértices y dos punteros más hacia los polígonos que la comparten, si se diera el caso. En la figura 1.2 se ilustra esta implementación:

$$V = [V_1, V_2, V_3, V_4] = [(x_1, y_1, z_1), (x_2, y_2, z_2), (x_3, y_3, z_3), (x_4, y_4, z_4)]$$
$$A_1 = [V_1, V_2, P_1, NULO]; \quad A_2 = [V_2, V_3, P_2, NULO]$$
$$A_3 = [V_3, V_4, P_2, NULO]; \quad A_4 = [V_1, V_4, P_1, NULO]$$
$$A_5 = [V_3, V_4, P_1, P_2]$$
$$P_1 = [A_1, A_4, A_5]; \quad P_2 = [A_2, A_3, A_4]$$

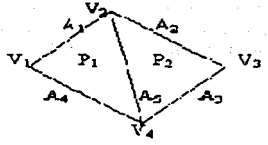


Figura 1.2. Mallas poligonales

1.5.2. *Superficies Descritas Mediante Polígonos Cúbicos.*

Para representar superficies con curvas es necesario recurrir a los polinomios de grado superior, ya que estos hacen aproximaciones muy reales. Estos polinomios tienen diferentes representaciones analíticas y de acuerdo a la que se utilice tienen varios métodos para evaluar la superficie. En general existen tres representaciones analíticas: explícita, implícita y paramétrica.

- La primera consiste en representar a los valores del eje Y y del eje Z en función del eje X, de tal forma que $Y = f(X)$ y $Z = g(X)$ o más propiamente dicho $Y = h(X, Z)$. La dificultad de este tipo de representación, es que son relaciones uno a uno, es decir, no se pueden obtener múltiples valores para Y con una sola X, así pues, es imposible representar circunferencias;
- La segunda consiste en definir un campo escalar en el espacio y luego tomar todos los puntos $P=[x, y, z]$ en los que la magnitud toma un valor constante, tal como $f(x, y, z) = 0$. A este tipo de superficies se les conoce como Isosuperficies o superficies equipotenciales;
- Y la última corresponde a uno de los métodos más empleados para representar curvas, ya que consiste en definir funciones para x, y y z en función de un parámetro independiente t, de tal forma que $x = x(t)$, $y = y(t)$ y $z = z(t)$. Aunque las dos anteriores representaciones pueden determinar cuales puntos tienden hacia la curva, esta representación sustituye el uso de pendientes por derivadas en los puntos. Esta característica le otorga una flexibilidad casi nula a la forma de la curva impidiendo deformaciones.

Realizando algunas operaciones matemáticas es posible convertir la representación explícita en una representación paramétrica. No así, para la forma implícita, ya que se requieren de otros cálculos diferentes. Así pues, cada segmento de la curva está representado por la ecuación 1.47:

$$Q(t) = [x(t), y(t), z(t)] \qquad \textbf{(Ec. 1.47)}$$

Donde $x(t)$, $y(t)$ y $z(t)$ tiene su representación en las ecuaciones 1.48, 1.49 y 1.50, respectivamente:

$$x(t) = a_x t^3 + b_x t^2 + c_x t + d_x \qquad \textbf{(Ec. 1.48)}$$
$$y(t) = a_y t^3 + b_y t^2 + c_y t + d_y \qquad \textbf{(Ec. 1.49)}$$
$$z(t) = a_z t^3 + b_x t^2 + c_z t + d_z \qquad \textbf{(Ec. 1.50)}$$

Si descompónemos las anteriores ecuaciones de dos matrices agrupando el término t en una de ellas y los coeficientes en otra, tenemos las matrices 1.13 y 1.14:

$$T = [\, t^3 \quad t^2 \quad t \quad 1 \,] \qquad \textbf{(Matriz 1.13)}$$

$$C = \begin{vmatrix} a_x & a_y & a_z \\ b_x & b_y & b_z \\ c_x & c_y & c_z \\ d_x & d_y & d_z \end{vmatrix}$$

Matriz 1.14

Rescribiendo la ecuación 1.47 en términos matriciales:

$$Q(t) = T * C \qquad \textbf{(Ec. 1.51)}$$

La respectiva derivada de la ecuación 1.51 es:

$$d/dt\, Q(T) = Q'(t) = [d/dt\, x(t) \quad d/dt\, y(t) \quad d/dt\, z(t)]$$
$$d/dt\, Q(T) = d/dt\, T * C = [3t^2 \quad 2t \quad 1 \quad 0] * C$$
$$d/dt\, Q(T) = [3a_x t^2 + 2b_x t + c_x \quad 3a_y t^2 + 2b_y t + c_y \quad 3a_z t^2 + 2b_z t + c_z]$$

Una curva puede dividirse en varios segmentos pequeños unidos por uno o varios puntos de empalme o coyuntura (*join point*), esto a fin de hacer una mejor descripción de la curva, de esta manera, si la dirección de los vectores tangenciales, en dos segmentos de la curva, es igual en el punto de empalme, entonces se dice que la curva es geométricamente continua, esto significa que las pendientes de ambos segmentos son iguales en el punto de empalme. En cambio si la dirección y magnitud de los vectores tangenciales son iguales en el punto de empalme se dice que existe una continuidad paramétrica en la variable independiente *t*, si la dirección y magnitud se mantiene a lo largo de toda la curva se dice que la curva es paramétricamente continua.

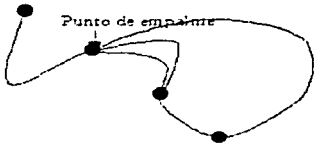


Figura 1.3. Punto de empalme

Regresando a la representación polinómica, estas están conformados por cuatro coeficientes con sus respectivas restricciones que permiten formular un sistema de ecuaciones para determinar los puntos que conforman la curva. Para ello es necesario realizar algunas modificaciones a la ecuación 1.51 haciendo que la variable C sea igual a $M * G$, así:

$$Q(t) = T * M * G \qquad \textbf{(Ec. 1.52)}$$

Donde M es una matriz base y G un vector de fuerzas llamado vector geométrico, expresado en forma matricial, la ecuación 1.52 es definida como:

$$Q(t) = [x(t)\ y(t)\ z(t)] = [t^3\ t^2\ t\ 1] * \begin{vmatrix} m_{11} & m_{12} & m_{13} & m_{14} \\ m_{21} & m_{22} & m_{23} & m_{24} \\ m_{31} & m_{32} & m_{33} & m_{34} \\ m_{41} & m_{42} & m_{43} & m_{44} \end{vmatrix} * \begin{vmatrix} G_1 \\ G_2 \\ G_3 \\ G_4 \end{vmatrix}$$

(Ec. 1.53)

De esta manera, las ecuaciones 1.48, 1.49 y 1.50 se pueden rescribir como:

$$x(t) = (t^3 m_{11} + t^2 m_{21} + t m_{31} + m_{41})g_{1x} + (t^3 m_{12} + t^2 m_{22} + t m_{32} + m_{42})g_{2x} + (t^3 m_{13} + t^2 m_{23} + t m_{33} + m_{43})g_{3x} + (t^3 m_{14} + t^2 m_{24} + t m_{34} + m_{44})g_{4x}$$

$$y(t) = (t^3 m_{11} + t^2 m_{21} + t m_{31} + m_{41})g_{1y} + (t^3 m_{12} + t^2 m_{22} + t m_{32} + m_{42})g_{2y} + (t^3 m_{13} + t^2 m_{23} + t m_{33} + m_{43})g_{3y} + (t^3 m_{14} + t^2 m_{24} + t m_{34} + m_{44})g_{4y}$$

$$z(t) = (t^3 m_{11} + t^2 m_{21} + t m_{31} + m_{41})g_{1z} + (t^3 m_{12} + t^2 m_{22} + t m_{32} + m_{42})g_{2z} + (t^3 m_{13} + t^2 m_{23} + t m_{33} + m_{43})g_{3z} + (t^3 m_{14} + t^2 m_{24} + t m_{34} + m_{44})g_{4z}$$

Así, estas ecuaciones enfatizan los pesos (*weights*) de cada elemento de la matriz geométrica, por lo que reciben el nombre de funciones *blending*. La ecuación 1.52 representa la fórmula general de una curva paramétrica, no obstante, es necesario recurrir a otros cálculos para determinar las dos incógnitas M y G, que veremos más adelante.

1.5.2.1. Curvas de Bezier.

Las diversas técnicas que existen para calcular los puntos por donde pasa una curva se pueden dividir en dos grandes grupos: los que se basan en cálculos de interpolación dado un conjunto de puntos, y aquellos que utilizan cálculos de aproximación. Los primeros obligan a los cálculos, de tal manera, que la curva debe pasar por todos los puntos de control especificados, mientras que los segundos sólo calculan aproximaciones a dichos puntos de control, por donde la curva ni siquiera los toca. En este último caso caen los cálculos de *Bezier*, quien parte del principio de que un punto en la curva puede ser descrito como una función paramétrica. La ecuación 1.54 representa este concepto:

$$p(u) = \sum_{i=0}^{n} p_i \; f_i(u) \qquad u \in [0, 1] \qquad \textbf{(Ec. 1.54)}$$

Donde la variable vector p_i representa $n + 1$ vértices de un polígono característico que define la curva, estos vértices también se conocen como puntos de control. Del mismo modo, define cuatro propiedades que las funciones *blending*, $f_i(u)$, deben poseer:

- Las funciones deben interpolar del primer punto de control hasta el último, a decir, el segmento de curva debe comenzar desde p_0 hasta p_n.
- La tangente en el punto de control p_0 debe estar dada por $p_1 - p_0$, de esta manera, se tendrá el control de la curva en cada punto.
- La segunda derivada en el punto p_0 debe estar definida por los puntos p_0, p_1 y p_2. Así, la *n-ésima* derivada, en cada punto de control, está determinada por sus n puntos vecinos.
- Las funciones $f_i(u)$ deben ser simétricas con respecto a u y a $(1-u)$.

Bezier se auxilia de los polinomios *Bernstein* para redefinir la ecuación 1.54.

$$p(u) = \sum_{i=0}^{n} p_i \, C(n, i) \, u^i (1-u)^{n-i} \qquad u \in [0, 1] \qquad \textbf{(Ec. 1.55)}$$

Y como sabemos, $C(n, i)$ es el coeficiente binomial, dado por:

$$C(n, i) = n! / i! (n-i)!$$

De esta manera, *Bezier* desarrolla la ecuación 1.55 para calcular 3, 4, 5 y 6 puntos de control, tal como se muestra a continuación.

1. Para tres puntos de control, $n = 2$.
$$p(u) = (1-u)^2 p_0 + 2u(1-u)p_1 + u^2 p_2.$$

2. Para cuatro puntos de control, $n = 3$.
$$p(u) = (1-u)^3 p_0 + 3u(1-u)^2 p_1 + 3u^2(1-u)p_2 + u^3 p_3.$$

3. Para cinco puntos de control, $n = 4$.
$$p(u) = (1-u)^4 p_0 + 4u(1-u)^3 p_1 + 6u^2 (1-u)^2 p_2 + 4u^3 (1-u)p_3 + u^4 p_4.$$

4. Para seis puntos de control, $n = 5$.
$$p(u) = (1-u)^5 p_0 + 5u(1-u)^4 p_1 + 10u^2(1-u)^3 p_2 + 10u^3(1-u)^2 p_3 + 5u^4 (1-u)p_4 + u^5 p_5$$

Todas las anteriores ecuaciones pueden formar matrices bases con sus respectivos vectores geométricos, sin embargo, la que más se utiliza es la que describe cuatro puntos de control. Esa ecuación se puede reordenar como:

$$p(u) = [(1 - 3u + 3u^2 - u^3) \;\; (3u - 6u^2 + 3u^3) \;\; (3u^2 - 3u^3) \;\; u^3] * [p_0 \; p_1 \; p_2 \; p_3]^T$$
(Ec. 1.56)

Aún más, podemos hacer que la ecuación 1.56 cumpla con la fórmula de la ecuación 1.53:

$$p(u) = [\, u^3 \; u^2 \; u \; 1 \,] * \begin{vmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{vmatrix} * \begin{vmatrix} p_0 \\ p_1 \\ p_2 \\ p_3 \end{vmatrix}$$
(Ec. 1.57)

Con la ecuación 1.57 se ha establecido matriz base para calcular curvas paramétricas usando las ecuaciones de *Bezier*, ya que se cumple con la ecuación 1.52. Una propiedad destacable de este tipo de funciones *blending* es

que cada elemento del vector p tiene un peso asociado dentro de la curva, así, el peso más bajo le corresponde al vértice p_0 incrementando sucesivamente hasta que se alcanza el máximo peso cuando $u = i = n$. Llegado a este lugar, comienza a decaer hasta cero en el punto p_n cuando $u = 1$.

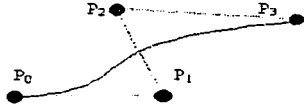


Figura 1.4. Curva de Bezier

1.5.2.2. Splines.

En el diseño de aeronaves son muy útiles los *splines* pues representan pedazos de metal o plástico flexible para modelar usando puntos de diseño. Los pesos o *ducks* mantienen el *spline* en su lugar mientras el diseñador traza una curva suave guiándose en la forma del *spline*. El comportamiento matemático de este tipo de curvas es equiparable con la forma en que se comportan las vigas. El objetivo de usar *splines* es encontrar una relación lineal entre la fuerza y tensión con respecto a un rango de elasticidad. Esta relación puede expresarse como un polinomio cúbico y, es por ello, que estos *splines* son muy recurridos para desplegar curvas o superficies curveadas en dispositivos gráficos. Existen varios dos tipos de *splines*, los cúbicos naturales (*natural cubic splines)* y los *B-splines*. Los primeros utilizan n puntos de control para poder manipular las curvas, además, si un punto de control es afectado por alguna transformación la curva completa se ve afectada por este cambio, ello implica recalcular una matriz de $n - 1 \times n - 1$. Estas propiedades hacen que los *splines* cúbicos naturales sea imposibles de representar en un dispositivo gráfico.

Los *B-splines*, en cambio, evitan el uso de puntos de control globales utilizando un conjunto de funciones tipo *blending* que le otorgan la facilidad de modificar cualquier punto de control de manera local, sin que esté afecte a toda la curva, dichos puntos reciben el nombre de controles locales. Además, los *B-splines* evitan el uso de interpolaciones para determinar los puntos de control de la curva. En la figura 1.5 se precisa la explicación anterior.

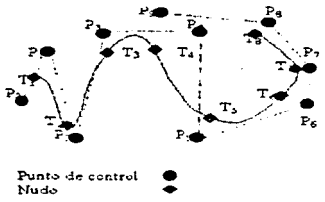


Figura 1.5. Spline

1.5.2.3. B-Splines.

Los *B-splines* cúbicos aproximan una serie de $n+1$ puntos de control $[p_0, p_1, \ldots, p_n]$ con una curva compuesta por $n-2$ polinomios de grado 3 que pasan por los segmentos de curvas $Q_1, Q_2, \ldots, Q_n$. Matemáticamente se expresa como sigue:

$$p(u) = \sum_{i=0}^{n} p_i \, N_{i,k}(u) \qquad \textbf{(Ec. 1.58)}$$

La ecuación 1.58 hace uso de las funciones *blending*, $N_{i,k}(u)$, cuyo grado depende de un parámetro k que es independiente al número de puntos de control, he aquí la diferencia con las curvas de *Bezier*. Estas funciones se define como:

$$N_{i,1}(u) = \begin{cases} 1, & si \quad t_i \le u < t_{i+1} \\ 0, & en\ cualquier\ otro\ caso \end{cases}$$

y

$$N_{i,k}(u) = \frac{(u - t_i)\, N_{i,k-1}(u)}{t_{i+k-1} - t_i} + \frac{(t_{i+k} - u)\, N_{i+1,k-1}(u)}{t_{i+k} - t_{i+1}}$$

Donde k controla el grado de los polinomios y la continuidad de la curva. La variable t_i representa los valores nodo (*knot values*), los cuales se encargan de relacionar los puntos de control, p_i, con la variable paramétrica u, a decir, estos

valores son los puntos de empalme o coyuntura; y se definen en la siguiente relación:

$$t_i = \begin{cases} 0, & si \quad i < k \\ i - k + 1, & si \quad k \le i \le n \\ n - k + 2, & si \quad i > n \end{cases}$$

Con las restricciones:

$$0 \le i \le n + k \quad \text{y} \quad 0 \le u \le n - k + 2$$

Donde n representa el número de puntos de control. De esta manera, podemos calcular las funciones *blending* para $N_{i,1}(u)$, $N_{i,2}(u)$, $N_{i,3}(u)$ y $N_{i,4}(u)$ para un polinomio cúbico ($k = 4$) a fin de obtener la matriz base y el vector de geometría, especificados en la ecuación 1.59:

$$p_i(u) = \frac{1}{6}\begin{bmatrix} u^3 & u^2 & u & 1 \end{bmatrix} * \begin{vmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 0 & 3 & 0 \\ 1 & 4 & 1 & 0 \end{vmatrix} * \begin{vmatrix} p_{i-1} \\ p_i \\ p_{i+1} \\ p_{i-2} \end{vmatrix}$$

(Ec. 1.59)

La ecuación 1.59 se basan en la suposición que los nudos estén espaciados en intervalos iguales a lo largo de la variable paramétrica t. Es por ello que a este tipo de *splines* se les conoce como *splines* cúbicos uniformes. asimismo se caracterizan por la racionalidad de los polinomios, es decir, un *spline* es racional por que define a $x(t)$, $y(t)$ y $z(t)$ como el radio de dos polinomios cúbicos.

El término B-spline se acuña por que este tipo de curvas define la suma de pesos (*weigths*) de sus polinomios bases, es por ello que la *B* quiere decir base (*basis*). Basado en estas propiedades se han subdividido los *B-splines* para dar origen a otro tipo de curvas más especializadas, que analizaremos a continuación.

1.5.2.3.1. B-splines No Uniformes y No Racionales.

Su nombre lo dice todo, este tipo de *splines* no tiene sus valores nudo de manera espaciada, ello significa que las funciones *blending* no son del mismo tamaño para cada intervalo, no obstante, esta propiedad le otorga la

característica de poder eliminar el uso de la segunda derivada en el punto de empalme para determinar el grado de continuidad de la curva, además, pueden interpolarse los puntos de control y calcular el punto inicial y final.

A diferencia de las anteriores curvas, se pueden agregar puntos de control para tener una mejor manipulación de la curva. El número mínimo de puntos de control que definen a una curva de este tipo, es de cuatro, así, el vector de nudos debe tener al menos $n+4$ elementos para definir la curva o segmento de la misma.

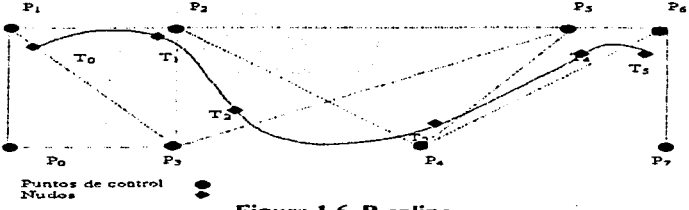


Figura 1.6. B-spline

El vector de nudos puede tener valores repetidos entre sus elementos, a este fenómeno se le conoce como multiplicidad. Por ejemplo, si se tiene el siguiente vector de nudos $t = [0, 1, 2, 2, 3, 4, 4, 5, 5, 5]$ se dice que el valor cero tiene una multiplicidad de uno, ya que se encuentra sólo una vez en el vector t. En cambio el valor de 5 tiene una multiplicidad de tres, ya que se repite tres veces el mismo valor dentro del vector t. Cuando existe repetición de valores indica que el segmento de curva que hay entre los valores t_i y t_{i-1} no es un segmento en realidad, sino un punto de empalme. Si la multiplicidad de un valor dentro del vector de nudos es igual a cuatro, es decir, $t_i = t_{i-1} = t_{i-2} = t_{i-3}$, entonces se ha encontrado el punto final que define a la curva.

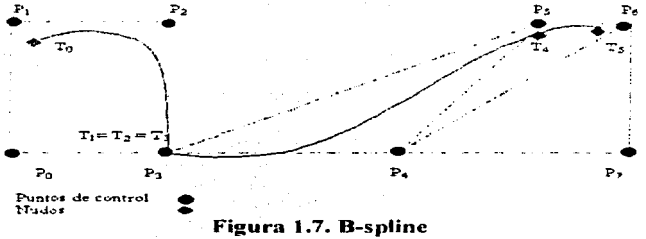


Figura 1.7. B-spline

La representación matemática de una curva *B-spline* no uniforme es:

$$Q(t) = P_{i-3}\,B_{i-3,4}(t) + P_{i-2}\,B_{i-2,4}(t) + P_{i-1}\,B_{i-1,4}(t) + P_i\,B_{i,4}(t) \qquad \textbf{(Ec. 1.60)}$$
$$3 \le i \le n \quad y \quad t_i \le t < t_{i+1}$$

La funciones *blending*, $B_{i,k}(t)$, están definidas por los intervalos entre los valores de los nudos y se definen recursivamente como en el caso de los *B-splines* uniformes. El cálculo de estas funciones puede llegar a ser muy tardado y poco eficiente, por lo que algunos investigadores sugieren calcular todos las posibles casos y almacenarlos en algún lugar para poder realizar todo tipo de operaciones en los diferentes casos que se presenten con los segmentos de la curva.

1.5.2.3.2. B-splines No Uniformes y Racionales.

Los NURBS (*No-rational, No-uniform B-splines*) son curvas muy utilizadas para dibujar secciones cónicas o cilíndricas. Para ello se auxilian de la representación homogénea de la ecuación 1.47:

$$Q(t) = [X(t) \quad Y(t) \quad Z(t) \quad W(t)] \qquad \textbf{(Ec. 1.61)}$$

Donde $X(t)$, $Y(t)$, $Z(t)$ y $W(t)$ son polinomios cúbicos que expresan a los puntos de control con coordenadas homogéneas, de esta forma, las ecuaciones 1.48, 1.48 y 1.50 son definidas como:

$$x(t) = X(t) / W(t)$$
$$y(t) = Y(t) / W(t)$$

$z(t) = Z(t) \quad W(t)$

Aplicando funciones *blending*, la ecuación 1.61 queda como sigue:

$$q(u) = \Sigma^{n}_{i=0} \; p_i w_i N_{i,k}(u) \qquad \Sigma^{n}_{i=0} \; w_i N_{i,k}(u) \qquad \textbf{(Ec. 1.62)}$$

Donde w_i representa el peso en cada punto de control p_i, y $N_{i,k}(u)$ son las funciones *blending* definidas para los *B-splines* uniformes. El número de elementos del vector de nudos es de $n+k+2$, donde k representa el grado del polinomio ($k=4$, si se desea una representación cúbica). Esta ecuación 1.62 otorga la propiedad de invariabilidad de los puntos de control ante cualquier transformación que se les aplique, estas son, translación, rotación, escalamiento y perspectiva.

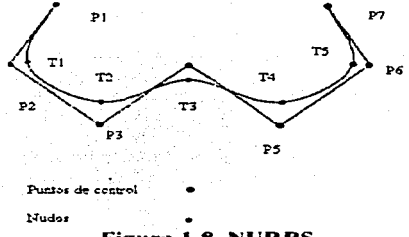


Figura 1.8. NURBS

1.5.2.4. Spline de Catmull-Rom.

Entre la familia de los splines nos encontramos con las ecuaciones *Catmull-Rom*, que basa sus cálculos en interpolaciones de un conjunto de puntos de control dados por $p_1, \ldots, p_n$ para encontrar otra serie de puntos de control, $p_0, \ldots, p_{n-1}$. La ecuación de *Catmull-Rom* es:

$$C(u) = \Sigma^{n}_{i=0} \; p_i \; G_i(u) \qquad \textbf{(Ec. 1.63)}$$

Donde $G_i(u)$ representan las funciones *blending*, cuya formulación nos arroja un factor de tensión δ entre los puntos de empalme, cuyo valor es de *0.5*. La ecuación 1.63 puede representarse como la ecuación 1.53:

$$C_i(u) = [\,u^3 \;\; u^2 \;\; u \;\; 1\,] * \begin{vmatrix} \delta & 2-\delta & \delta-2 & \delta \\ 2\delta & \delta-3 & 3-2\delta & -\delta \\ -\delta & 0 & \delta & 0 \\ 0 & 1 & 0 & 0 \end{vmatrix} * \begin{vmatrix} p_{i-1} \\ p_i \\ p_{i+1} \\ p_{i+2} \end{vmatrix}$$

(Ec.1.64)

1.5.2.5. Beta Spline.

Por último tenemos los *Beta Spline*, muy utilizado para tener un mayor control sobre la curva adicionando dos parámetros: la polaridad y la tensión, representados con β_1 y β_2, respectivamente. La matriz base está dada por:

$$M = \frac{1}{\delta} \begin{vmatrix} -2\beta^3_1 & 2(\beta_2 + \beta^3_1 + \beta^2_1 + \beta_1) & -2(\beta_2 + \beta^2_1 + \beta_1 + 1) \\ 6\beta^3_1 & -3(\beta_2 + 2\beta^3_1 + 2\beta^2_1) & -3(\beta_2 + 2\beta^2_1) \\ -6\beta^3_1 & 6(\beta^3_1 - \beta_1) & 6\beta_1 \\ 2\beta^3_1 & \beta_2 + 4(\beta^2_1 + \beta_1) & 2 \end{vmatrix}$$

(Matriz 1.15)

Donde:

$$\delta = \beta_2 + 2\beta^3_1 + 4\beta^2_1 + 4\beta_1 + 2$$

Conocer la forma en como los paquetes gráficos, sobretodo los de diseño, calculan y determinan la forma y manipulación de curvas o superficies es muy importante ya que con ello se aprende y se entiende mejor la utilización de estas herramientas que proporciona dichos paquetes. Existe una cantidad considerable de bibliografía que trata más a fondo el uso de las formulaciones de *Bezier* y de los *Splines*.

1.5.3. Superficies Descritas Mediante Polígonos Cuadráticos.

Como se había mencionado antes, existe una forma de representar una superficie mediante una representación implícita, dicha formulación tiene la siguiente forma general:

$$f(x, y, z) = ax^2 + by^2 + cz^2 + 2dxy + 2eyz + 2fxz + 2gx + 2hy + 2jz + k = 0$$
(Ec. 1.65)

Con la ecuación 1.65 se debe manejar un campo escalar donde los valores correspondientes a x, y y z toman un valor constante. A este tipo de superficies se les denomina isosuperficies. Por otro lado, si el campo escalar tiene variaciones suaves, entonces se llaman superficies equipotenciales. La analogía que se puede hacer para entender este tipo de superficies es usando el ejemplo de los campos eléctricos que rodean a un cuerpo, de esta manera, el cuerpo genera un potencial a su alrededor en cada uno de los puntos del espacio y se tiene que la función de la anterior fórmula es sumar las contribuciones del potencial en cada punto para realizar una aproximación de la superficie. La mayoría de los autores evocan el uso de funciones *gaussianas* para determinar el potencial en cada punto. La figura 1.9 representa esta función.

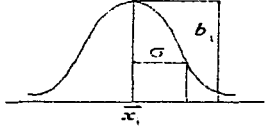


Figura 1.9. Función Guassiana

Donde x_i representa la posición, b_i la fuerza y $r_i(\sigma)$ es la desviación estándar. Así se puede definir el potencial en cada punto como:

$$F(x, b, r) = b_i e^{u}; \quad u = (1/r_i^2)(x - x_i)^2$$

Pasando este sistema a las coordenadas cartesianas, tenemos que:

$$F(x, y, z) = \Sigma_i b_i e^{a}$$

Esta generalización es muy utilizada por los paquetes de animación para producir objetos globulares tales como *blobs*, *metaballs* y *soft objects*, ya que las aproximaciones que hacen en tercera dimensión son excelentes.

1.6. Antialiasing.

El término de *aliasing* se refiere al fenómeno producido por el muestreo de una señal con un rango inadecuado de intervalos, en el caso del procesamiento de imágenes, este intervalo causa que los bordes de las imágenes digitalizadas presenten pequeños errores, que en conjunto pueden producir un efecto de bordes dentados o en forma de escalera que hacen aún más evidente este fenómeno. Los efectos del *aliasing* en el procesamiento de imágenes son causados cuando se intenta forzar las posiciones de la imagen para que coincida exactamente con las posiciones individuales de los *píxeles* dentro de la pantalla de despliegue; aún cuando los bordes de la imagen no se alineen a la perfección, es posible disfrazar los bordes dentados usando una mezcla de colores apropiada.

El truco más recurrido para que el observador no detecte a simple vista el efecto del *aliasing* consiste en la manipulación de la resolución del monitor, a decir, entre mayor sea la resolución del monitor gráfico menos perceptibles serán los efectos de este fenómeno. Matemáticamente, el *aliasing* no puede ser eliminado totalmente, no obstante, estos paquetes (CAD, PDI, etc.) han desarrollado técnicas para disminuirlo al mínimo, que, básicamente, consisten en aumentar el rango de muestreo de la imagen. Estas técnicas combinadas con trucos hacen que el despliegue de la imagen sea casi perfecto. A continuación enlistaremos las principales técnicas utilizadas por estos paquetes:

1. La técnica más directa es incrementar la resolución del dispositivo donde será desplegada la imagen, sin embargo, esta técnica resulta mucho más costosa en términos de recursos y procesamiento en comparación a los beneficios que otorga.

2. Otro método consiste en aplicar ruido o pequeñas manchas (*blurring*) cuando los efectos del *aliasing* son mínimos.
3. La combinación de métodos de detección de orillas y de alisamiento de bordes[6] pueden dar resultados bastantes satisfactorios para imágenes con alto contraste. Este tipo de combinación requiere de dos procesos fundamentales: el primero consiste en realizar comparaciones entre vecindades de cada área de la imagen. El siguiente paso consiste en procesar los bordes mezclando colores para encontrar el más adecuado, el costo de este proceso depende del contraste de la imagen.

1.7. Herramientas de Despliegue.

1.7.1. Tubo de Rayos Catódicos "Repasado".

El primero de todos los dispositivos de despliegue y el más exitoso hasta hoy es el tubo de rayos catódicos repasado; su funcionamiento se basa en un haz de electrones dirigido hacia posiciones específicas de una pantalla recubierta con una película de fósforo a través de un sistema de enfoque y de reflexión. El fósforo emite una pequeña mancha de luz en cada posición donde hace contacto el haz de electrones. Las distintas clases de fósforos empleadas proporcionan ciertas características al CRT como el color y la persistencia, esta última se refiere al tiempo que el fósforo sigue emitiendo luz después de que se retira el haz de electrones.

Los principales componentes de este cañón son el cátodo de metal y una rejilla de control. El haz de electrones se produce con el suministro de calor al cátodo a través de una bobina de alambre llamada filamento; esta acción hace que los electrones se desprendan de la superficie del cátodo y viajen en el vacío hacia el recubrimiento de fósforo cargado con voltaje positivo. Como la cantidad de luz emitida por el recubrimiento de fósforo depende del número de

[6] En la sección 2.2 del capítulo 2 se expone este tema.

electrones que hacen contacto con la pantalla, es posible controlar la rapidez del despliegue al variar el voltaje de la rejilla de control; y el nivel de intensidad para las posiciones individuales en la pantalla se especifica con comandos de software. Asimismo, para lograr que el haz de electrones converja en una pequeña mancha de luz conforme hace contacto con el fósforo es necesario un sistema de enfoque, de lo contrario el haz se expandiría conforme se aproximara a la pantalla; este sistema puede utilizar campos eléctricos o magnéticos, de acuerdo a las necesidades requeridas.

Otra particularidad del CRT es la resolución, es decir, el número máximo de puntos que se pueden desplegar sin que se traslapen. La resolución más común es *1280 x 1024*, aunque existen otras más altas. Esta propiedad no determina el tamaño físico del área de despliegue. Por último, la razón de aspecto del CRT se refiere a la proporción de los puntos verticales con respecto de los puntos horizontales necesaria para producir líneas con una longitud igual en ambas direcciones de la pantalla, a decir, una razón de aspecto de tres cuartos implica que una línea vertical trazada con tres puntos tiene la misma longitud que una línea horizontal que se traza con cuatro puntos.

1.7.2. Despliegue de Barridos con Rastreador.

Esta tecnología fue introducida por la televisión y, actualmente se aplica a los monitores gráficos. Esta técnica utiliza un haz de electrones para recorrer la pantalla, renglón por renglón de arriba hacia abajo, con el objeto de crear un patrón de manchas iluminadas. Consta de tres dispositivos:

- Un *frame buffer*, llamado también memoria de imagen, donde la imagen será almacenada como una matriz de *píxeles*. Si se desea desplegar variaciones de color e intensidad, se requieren bits adicionales, por lo regular, se incluyen hasta 24 bits por *píxeles*, y dependiendo de la resolución, se requieren varios *megabytes* de almacenamiento para el búfer de imagen.
- Un monitor, que será el dispositivo de despliegue.
- Un controlador de video (*display controller*), que es una interface de transferencia entre el dispositivo de despliegue y la memoria de imagen; la comunicación digital analógica y viceversa entre estos dispositivos debe ser de 60 a 80 cuadros por segundo para realizar una

actualización de la imagen desplegada, reduciendo al mínimo el *flickering* o proceso de desvanecimiento.

1.7.3. Monitores de TRC de Color.

Un monitor con un tubo de rayos catódicos despliega imágenes a color utilizando una combinación de fósforos que emiten luz con colores distintos, cada *pixel* en el monitor está recubierto por tres tipos de fósforo que producen los colores del sistema RGB (*Red, Green, Blue*) y a cada color se le asocia un cañón de electrones. Entre el dispositivo de despliegue y el haz de electrones se encuentra una barrera de metal, denominada máscara de sombra (*shadow mask*), cuya función es atraer los tres haces de electrones sobre la serie de orificios alineados para producir una mancha triangular o cuadricular de color sobre el dispositivo de despliegue. Los puntos de fósforo se ordenan para que cada haz de electrones pueda activar sólo un punto de color correspondiente a su máscara de sombra, además, las correspondientes variaciones del haz de electrones afecta directamente el brillo del fósforo, obteniendo diferentes tipos de colores.

1.7.4. Tubos de Almacenamiento con Vista Directa.

Los tubos de almacenamiento con vista directa (*Direct View Storage Tubes*) o CRT de memoria, son unos tubos especiales que se comportan como una placa de fósforo de alta persistencia, en la cual una imagen trazada sobre esa placa puede mantenerse por más de una hora sin deformidades o sin presentar *aliasing*. La construcción de este tipo de dispositivo consiste en tubo con propiedades de fósforo normales, un cátodo lanzador de electrones y una máscara montada sobre la superficie del fósforo. Los electrones son lanzados uniformemente por toda la superficie de la pantalla y la máscara funciona como filtro que selecciona los puntos y / o áreas de la superficie que serán iluminados por los electrones. La máscara está construida con un material dieléctrico, de esta manera, las áreas cargadas positivamente se atraen permitiendo que pasen

a través de la máscara. Las ventajas de este tipo de sistema recaen en su alta resolución y la ausencia de *flickering* o desvanecimiento, empero, esta tecnología ha sido desplazada debido al proceso de borrado de la imagen, pues produce un destello que desgasta el fósforo y las imágenes desplegadas pierden definición.

1.7.5. Despliegues de Plano.

Los sistemas de despliegue plano se caracterizan por su poco volumen y peso, además, son ampliamente usados en relojes de mano, calculadoras, videojuegos de bolsillo, monitores de computadoras portátiles, y actualmente, en televisores, monitores de computadoras personales y *display* de teléfono celulares. Por su construcción, esta tecnología se divide en dos tipos: emisivo y no emisivo.

Los dispositivos planos emisivos transforman la energía eléctrica en luz utilizando paneles de plasma, diodos de luz o paneles electroluminiscentes. Los paneles de plasma se construyen con dos placas de cristal y el vacío entre ambas se llena con una mezcla de gases. En los dispositivos de despliegues electroluminiscentes, la zona entre las placas de cristal es rellenada con fósforo. Los diodos de emisión de luz (*LED*) se ordenan en un arreglo matricial para formar las posiciones de *píxel* en la pantalla.

Los dispositivos de despliegue plano no emisivo utilizan la luz de cualquier tipo para convertirla en patrones gráficos, esto es, producen una imagen al pasar luz polarizada de su alrededor o de una fuente de luz interna a través de un material de cristal líquido que puede alinearse con cualquier bloque o transmitir la luz. Por lo general, las imágenes se almacenan en un buffer de repaso y la pantalla se refresca con un índice de 60Hz.

1.7.6. Dispositivos de Vista Tridimensional, Sistemas Estereoscópicos y Realidad Virtual.

En la actualidad los dispositivos gráficos pueden presentar imágenes en tres dimensiones, o realizar algunos trucos para que las imágenes aparezcan con profundidad. Estos sistemas se clasifican en sistemas tridimensionales, estereoscópicos y de realidad virtual. Los sistemas tridimensionales utilizan un CRT normal acompañado de un espejo vibrador cóncavo que cuando el haz de electrones atraviesa la pantalla de fósforo, la vibración del espejo hace que los *píxeles* se reflejen en ciertas posiciones del espejo, otorgándoles a los objetos desplegados un aspecto de tridimensionalidad sin importar la posición del observador.

Los sistemas estereoscópicos utilizan un fenómeno llamado estereósis para engañar la visión del observador. Este fenómeno se origina debido a la disparidad binocular que presentan los ojos humanos, esto es, la visión de cada ojo difiere por centímetros uno del otro, no obstante, el cerebro junta ambas imágenes y las combina en una sola para otorgar la apariencia de tridimensionalidad. Esta particularidad en la visión es aprovechada para dar a los objetos gráficos planos una apariencia de profundidad.

Por último, se tiene a la realidad virtual, la cual considerara una serie de aspectos para llevar a los objetos desplegados a una forma de realidad virtual:

- Eliminación las superficies ocultas.
- Sombreado de las superficies visibles.
- Iluminación.

1.7.7. Dispositivos de Entrada.

Es importante considerar que los sistemas de despliegue muchas veces necesitan interactividad con el usuario, por lo que es necesario incluir dispositivos de entrada para manipular las escenas que serán desplegadas. Entre los dispositivos de entrada más comunes se encuentran los siguientes:

- Teclado. Empleado como un dispositivo para capturar cadenas de texto que se deseen asociar a la escena.
- Ratón. Se utiliza para una manipulación más directa y precisa de los movimientos del cursor en el área de despliegue, su función principal es la de registrar la cantidad y dirección del movimiento. El diseño ofrece seis grados de libertad para seleccionar posiciones, rotaciones y otros parámetros especiales.
- Palanca de control. Esta construida para detectar movimientos hacia delante, hacia atrás, a la izquierda y a la derecha.
- Esfera palmar. Una pequeña esfera de control que calcula la cantidad y dirección de los movimientos efectuados y los traduce a coordenadas especificas de *píxel*.
- Guante de Datos. Consiste en una serie de sensores colocados sobre un guante, que interpretan los movimientos de la mano y la orientación que presenta. Los sensores transmisores y receptores están construidos con un conjunto de reticulas mutuamente perpendiculares que forman un sistema de coordenadas cartesianas denominado *headtrack*.
- Plumas emisoras de luz. Este dispositivo consta de un mecanismo fotoeléctrico en la punta y de un interruptor de presión en el otro extremo. Su funcionamiento consiste en emitir luz sobre la pantalla de fósforo, la cual dibuja los modelos de movimiento.
- Paneles de tacto. Los paneles de tacto o *touch screen* son dispositivos gráficos sensibles al tacto, las cuales, al ser tocadas responden con una serie de eventos previamente programados. Existen tres tipos de paneles de tacto, los ópticos, acústicos y eléctricos.
- Rastreador de Imágenes (*Scanner*).
- Digitalizadores. Esta tabla tiene una serie de sensores colocados de tal manera que forman una matriz de coordenadas, cuando se hace contacto con la tabla los sensores envían las señales correspondientes y el software las traduce en coordenadas dentro del monitor.

1. 8. Aplicaciones de la Graficación.

Son varias las ciencias y artes que han encontrado en la graficación por computadora las herramientas necesarias para llevar a cabo sus tareas. Entre ellas destacan:

- CAD. El diseño asistido por computadora (CAD, *Computer Assisted Design*) es una de las aplicaciones que explota al máximo las bondades de la graficación por computadora. Encuentra su máxima utilidad en tareas de ingeniería y de arquitectura donde el diseño y la simulación de sistemas físicos. Este tipo de software explota al máximo las interfaces gráficas para presentar un elaborado sistema de ventanas para comunicarse con el usuario.
- Arte por Computadora. El arte por computadora tiene fines más comerciales e, irónicamente, es esta finalidad la que le ha dado un gran auge y amplio desarrollo en la perfección de herramientas dedicadas a este propósito. Tanto así, que incluso se ha desarrollado hardware especializado para ayudar a los artistas a pintar sobre los monitores de sus computadoras, y en algunos casos se combinan software de CAD (*Computer Assisted Design*), programas de dibujo, programas de modelado y diagramación de texturas para reconstruir, de manera virtual, obras de artes que han sido dañadas por el paso del tiempo. Básicamente, el arte por computadora se enfoca a la creación de paisajes fractales, creación de imágenes foto-realistas y técnicas de *Morphing*[7].
- Visualización. Una de las herramientas más aplicadas en todos lo ámbitos es la visualización, cuya tarea principal es la de convertir datos planos en una forma visual más comprensible. En el ámbito de la graficación por computadora, la visualización se divide en dos: la visualización científica y la visualización empresarial. La primera se encarga de representar datos originados de simulaciones matemáticas en una forma visual, mientras que la segunda se encarga de representar datos financieros y económicos.
- Procesamiento de Imágenes. Aunque el procesamiento de imágenes y la graficación por computadora realizan tareas bien diferenciadas, en la

[7] El *Morphing* es una técnica que cambia la forma de un agente para adaptarlo a una plataforma especifica.

práctica, ambas llegan a confundirse y se toma al procesamiento de imágenes como parte de la graficación por computadora. Por ello, la literatura especializada hace énfasis en la diferencia entre ambas técnicas, de este modo, se afirma que la graficación por computadora se encarga de crear imágenes originadas a partir de datos externos, mientras que el procesamiento de imágenes manipula y / o interpreta imágenes ya existentes originadas por alguna fuente externa.

- Interfaces Gráficas de Usuario. En la actualidad son muy pocos los paquetes de software que no tiene una interfaz gráfica, la gran mayoría se auxilia de un GUI (*Graphical User Interface*) para facilitar al usuario la realización de tareas complejas. Las características sobresalientes de este tipo de paquetes es la administración de ventanas (*MDI, Multiple Developed Interface*) que le permiten al usuario tener múltiples ventanas donde pueden usar elementos gráficos o textuales. Del mismo modo, destaca la interacción con el ratón y otros elementos interactivos que hacen más agradable el ambiente de desarrollo. Otra característica es el uso de menús e iconos que ayudan a asociar información o tareas usando elementos gráficos.
- Educación, Capacitación y Mantenimiento. Como es sabido, el aprendizaje se facilita más cuando se presentan imágenes representativas del tema, es por esta razón que han surgido una serie de paquetes gráficos que ayudan al aprendizaje de personas adultas o de niños denominados CAL (*Computer Aided Learning*), cuya tarea principal es facilitar la comprensión de conceptos abstractos o difíciles de entender, utilizando imágenes relacionadas al mismo. Otra aplicación de este tipo de software son aquellos que están destinados a capacitar personal en tareas específicas, esto con el fin de que el estudiante vaya asimilando el adiestramiento antes de tener la oportunidad de manipular un equipo real.

Capítulo II.

Volume Rendering

2.1. Visualización Científica.

En el capítulo anterior se mencionaron conceptos introductorios de la Graficación por Computadora y de las herramientas que utiliza, así como de las áreas de aplicación más comunes. También se hizo mención sobre la necesidad de interpretar datos abstractos de la mejor manera posible, en este sentido, una de las áreas de estudio que se encarga de satisfacer esta necesidad es la visualización.

La Visualización se enfrenta al problema de transformar datos crudos, por decirlo de alguna manera, en forma representativa entendibles para cualquier individuo, ya sea a través de gráficas, imágenes, símbolos, e inclusive texto. Sin embargo, esto conlleva a otro problema, la visualización debe encontrar una forma de representación adecuada para los datos, es decir, no se pueden utilizar imágenes para representar datos estadísticos, para ello es mejor utilizar gráficas de pastel o de barras, ni tampoco se deben utilizar gráficas para representar datos de una tomografía, por ejemplo. Aunque estas reglas no están escritas, el personal dedicado a esta área sabe que tipo de visualización corresponde para cierto tipo de datos.

Existen dos tipos de Visualización, la Empresarial y la Científica; ambas persiguen el mismo objetivo, pero la finalidad de su producto es muy distinta. En este trabajo de investigación, profundizaremos en la Visualización Científica, pues es la que aportado más herramientas y técnicas a la Graficación por Computadora. Primeramente, es necesario definir qué es la visualización científica, y para ello existen varias definiciones:

1. ".... es la representación visual de los datos ...."
2. ".... aplicación de métodos gráficos para apoyar la interpretación y significado de datos científicos ...."
3. ".... el uso de imágenes generadas por computadora para ganar información y entendimiento de los datos (geometrías) y sus relaciones (topologías), para comunicarlos a otros (científicos, estudiantes, sociedad, etc.)."

4. "... es la generación de imágenes a partir de datos, con el objeto de transformarlos en información y ganar entendimiento."[1]

Todas estas cuatro definiciones destacan la tarea de la visualización científica, transformar datos en alguna forma visual. La definición propuesta por el autor de este trabajo de investigación es la siguiente: la Visualización Científica es un conjunto de herramientas, técnicas y teoría aplicadas a encontrar la mejor manera de transformar una gran cantidad de datos numéricos, en representaciones geométricas coherentes, a fin de ayudar a los sentidos a entender la información de manera más inteligible. La razón de proponer una nueva definición para la VC es por que considero que las existentes no rescatan el espíritu de esta disciplina, ya que no se trata de generar imágenes, sino de entender toda una metodología a fin de lograr un mayor entendimiento.

Por otro lado, el adjetivo de científico que se le agrega a esta disciplina es por que los datos con los que se trabaja provienen de simulaciones científicas, experimentación, censos, etc., es decir, datos que representan fenómenos físicos. Los objetivos de la VC son:

- Definir el problema y establecer un plan de acción.
- Analizar y explorar los datos, en su estado original, para clasificarlos o agruparlos, con el fin de entender su comportamiento.
- Comunicar ideas a través del análisis de datos.
- Construir representaciones visuales que se apeguen al modelo establecido previamente.

De acuerdo con lo anterior, el proceso de la VC puede interpretarse como en el siguiente modelo de la figura 2.1.[2]

[1] La última definición fue obtenida del Departamento de Visualización en la Dirección General de Servicios de Computo Académico (DGSCA) de la UNAM.

[2] Tomado de HyperVis

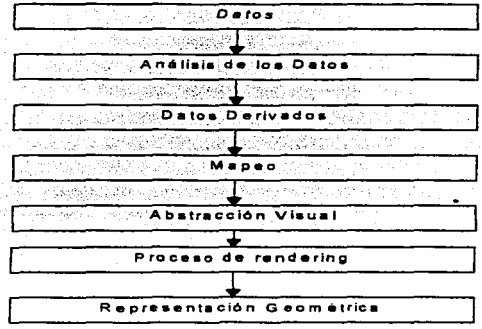


Figura 2.1. Proceso de la Visualización Científica.

Como se puede observar en el anterior modelo, gran parte del proceso de visualización se dedica al análisis de los datos. Es aquí donde hay que prestar especial atención al origen de los datos que pueden ser instrumentos de percepción remota, observaciones, experimentos, simulaciones matemáticas, etc. y de acuerdo a ello, es necesario clasificar los datos. Básicamente existen tres tipos de datos: los datos escalares, los cuales poseen magnitudes y normalmente representan cantidades unitarias; los datos vectoriales, quienes poseen magnitud y dirección y están formados por un conjunto de valores escalares de igual dimensión; y, por último, los datos tensoriales.

Existe una segunda clasificación de datos de acuerdo al sistema de escala utilizados para generarlos:

1. Datos en la escala de proporción. Normalmente este tipo de datos está ligado a un punto de origen, o punto cero, con un significado físico; los intervalos de medición son constantes y se puede realizar cualquier tipo de operaciones con ellos.

2. Datos originados a partir de una escala de intervalo. Son parecidos a los anteriores, pero estos no están ligados a un punto de cero, que se determina de manera arbitraria.
3. Datos producidos en una escala ordinal. Son datos ordenados de acuerdo a una escala arbitraria, no existen diferencias cuantitativas entre los mismos.
4. Datos creados a partir de una escala nominal. Son datos que se clasifican de acuerdo a las cualidades que posee el objeto mesurado.

Dentro del proceso de análisis de los datos es necesario clarificar las posibles imperfecciones que estos pueden presentar, tales como corrupciones, incoherencias, incertidumbre; ante esta situación, la tarea de la VC es presentar estos errores cuando se realiza la representación visual de los datos, haciendo hincapié en ese aspecto. El modelo de análisis de datos es mostrado en la figura 2.2.

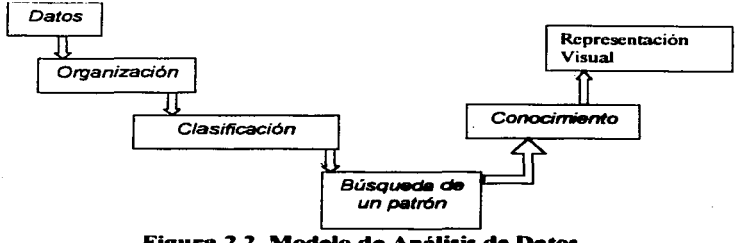


Figura 2.2. Modelo de Análisis de Datos.

Como se puede observar en la figura 2.2, es básicamente la misma que la figura 2 1, la única diferencia estriba en que este diagrama está orientado al análisis de los datos. Así, se definen una serie de técnicas de visualización especificas para cada tipo de datos. Entre las más recurridas son:

1. Isocontornos. Esta técnica está orientada a la graficación de superficies de valor constante, ideal para determinar la estructura general de datos escalares en una escala de tiempo.
2. *Volume Rendering*. Es una técnica basada en la transformación de datos tridimensionales a datos bidimensionales, sin que estos pierdan

su apariencia de tridimensionalidad. Esta técnica es aplicable a datos de tipo vectorial y tensorial.
3. Isosuperficies. Ideal para representar campos vectoriales, dentro de espacios tridimensionales donde la magnitud de los vectores es igual.
4. Convolución Integral de Línea. Se utiliza para definir un filtro para determinar la dirección de los campos vectoriales.
5. Técnicas de advección. Consiste en el cálculo de trayectorias de objetos sin masa dentro de un campo vectorial.
6. Grafos. Esta técnica busca las cualidades de los datos y los representa de manera simbólica, normalmente se utilizan íconos.
7. Topologías. Esta técnica se basa en el análisis y clasificación de los puntos críticos y en el cálculo de sus relaciones.
8. *Data Mining*. Se basa en la búsqueda de algoritmos de agrupamiento para descubrir tendencias y comportamientos de los datos.

Después de toda esta información sólo queda describir el *pipeline* o metodología de la Visualización Científica, expuesto por *Robertson* y *DeFerrari*, quienes exponen seis componentes de la Visualización, a decir, el modelo de datos, especificaciones de Visualización, representación de la Visualización, aplicación de requerimientos (*matching procedures*), despliegue de la visualización e interacción.

1. El modelo de datos se refiere al conjunto de normas y procedimientos que nos ayudan a mantener, manipular y comprender un conjunto de datos, los requisitos que debe cubrir son: soporte para múltiples conjuntos de datos, descripción de las relaciones entre ellos, información sobre los datos.
2. Las especificaciones de la visualización son las reglas o restricciones que el usuario define, o los fines que se persigue con la interpretación de los datos.
3. La representación de la visualización es la exposición de posibles técnicas elegibles para representar los datos.
4. Aplicación de los requerimientos. Se refiere a encontrar una visualización realizable tomando en cuanta los tres puntos anteriores.
5. Despliegue. Es la representación visual de los datos a través de un dispositivo de despliegue.
6. Interacción. Se refiere a la manipulación de los parámetros de la representación visual sin tener que modificar el sistema de visualización, establecido en los anteriores pasos.

Los pasos anteriores se pueden resumir en el diagrama de la figura 2.3:

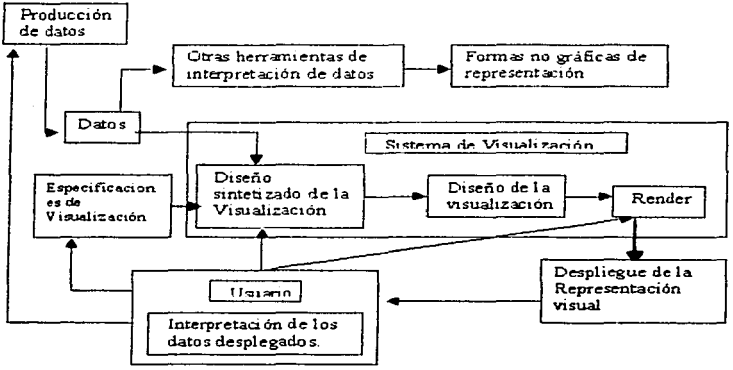


Figura 2.3. Pipeline de la Visualización Científica.

Después de esta breve introducción a VC, la técnica que más interesa en este capítulo es el *VR*, sin embargo, es necesario explicar algunos conceptos breves sobre el procesamiento de imágenes, ya que mucha de la teoría que ahí se utiliza puede extenderse al espacio tridimensional. La próxima sección está enteramente dedicada a ello.

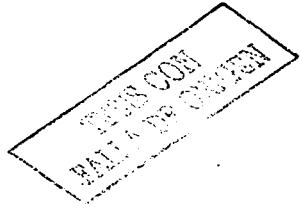
2.2. Introducción al Procesamiento Digital de Imágenes.

2.2.1. Definición.

Desde principios del siglo pasado, los investigadores sintieron la necesidad de procesar imágenes para reducir el ruido inherente adquirido por el dispositivo de adquisición, así como de manipular sus propiedades para una mejor interpretación. Algunos investigadores propusieron utilizar la teoría del procesamiento de señales ya que proporcionaba las herramientas matemáticas para llevar un completo análisis de señales, y en este contexto, una imagen puede ser considerada como una señal bidimensional o tridimensional, así pues, el análisis matemático de una dimensión podría extenderse a otros dominios *n* dimensionales con relativa facilidad. De esta manera, surge una nueva teoría con características propias, a la que se le denominó Procesamiento Digital de Imágenes, PDI.

Con el avance de las computadoras y su poder de procesamiento, la implementación de todos los conceptos de la nueva teoría del procesamiento digital de imágenes se hizo bastante accesible incluso para computadoras personales, así pues, la herramienta principal del PDI es la computadora, es por ello que varios autores define al PDI como un conjunto de procedimientos y técnicas aplicadas a procesar una imagen utilizando una computadora.

Las áreas de estudio se enfocan principalmente a tres tareas fundamentales, a decir, el procesamiento de la imagen, el análisis de la imagen y la interpretación de la misma. Todas esta tareas están bien diferenciadas entre sí y cada una posee fundamentos matemáticos propios, por ejemplo. El procesamiento de la imagen consiste en reconstruir y restaurar imágenes afectadas por una señal de ruido, para ello se tiene varias herramientas de convolución y filtros. El reconocimiento de patrones está más enfocado a identificar las propiedades de la imagen. Y, por último, la comprensión de la imagen (*image understanding*), se refiere a encontrar un sentido físico a las propiedades de la imagen.



2.2.2. Procesamiento de la Imagen.

Una imagen está definida como una función de dos variables independientes *a(x, y)*, donde *a* representa la amplitud de la señal en la coordenada *(x, y)*. Para iniciar el procesamiento de la misma es necesario cuantizarla en un rango de valores determinado, es decir, discretizarla con el objetivo de obtener una imagen en escala de grises. La imagen cuantizada es del mismo tamaño, en renglones y columnas, que la imagen original. El proceso de discretización incluye una serie de formulaciones matemáticas demasiado abstractas como para describirlas en esta sección, por el momento veremos ese proceso como una caja negra, como lo muestra la figura 2.4.

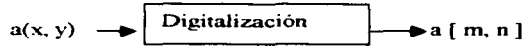


Figura 2.4. Digitalización de una Imagen.

La imagen digitalizada, *a[m, n]*, es el resultado de un proceso de cuantización donde cada coordenada *(x, y)* fue discretizada en un rango de valores para obtener la coordenada *[m, n]*. Asimismo, existen tres tipos de operaciones que se pueden aplicar a la imagen digitalizada: operaciones puntuales, locales y globales. Las operaciones puntuales son aquellas que operan a nivel *pixel*, es decir, por cada *pixel* de entrada tenemos un *pixel* de salida. Las operaciones locales hacen uso de vecindades (*neighbors*) para procesar la imagen, es decir, dado un conjunto de *pixeles* agrupados en una vecindad se obtiene sólo un *pixel* de salida. Las vecindades por lo regular son regiones rectangulares o hexagonales de dimensiones variadas. Por último, las operaciones globales operan sobre toda la imagen con el objeto de obtener un sólo *pixel* de salida.

Las operaciones descritas pueden caer en el domino de la frecuencia o del espacio. Una imagen como tal, definida en escala de grises, se dice que está en el dominio del espacio (*spatial domain*), bajo este dominio sólo es posible aplicar ciertos operadores para procesarla mientras que en el dominio de la frecuencia (*frecuency domain*) se pueden aplicar filtros derivativos y operaciones de alisado (*smoothing*). La transformación de la imagen del dominio del espacio al dominio de la frecuencia se lleva acabo con alguna de las transformaciones de *Fourier*, *Walsh*, o *Hadamard*.

2.2.2.1. Dominio del Espacio.

El procesamiento de una imagen definida en el dominio espacial se reduce a la siguiente ecuación $g(x, y) = T[f(x, y)]$ donde $f(x, y)$ es la imagen original en escala de grises, $g(x, y)$ es la imagen resultante del procesamiento y T es un operador cualquiera que procesa la imagen; el operador T, normalmente, es una matriz cuadrada o rectangular centrada en el *pixel* (x, y) y cuyas vecindades determinan su nuevo valor. A este operador también se le conoce como máscara (*masks*). El operador más simple es una máscara de $1x1$, utilizada para el negativo de la imagen.

2.2.2.1.1. Negativo de la Imagen.

Obtener el negativo de la imagen es relativamente fácil. Sabiendo el máximo nivel de grises solamente es necesario restarle el valor del color en cada coordenada o *pixel* (x, y), por ejemplo, si el máximo nivel de grises es de $256 = 2^8$, entonces el color inverso en la localidad (x, y) será $256 - (x, y)$. La figura 2.5 muestra la típica transformación para obtener el negativo de una imagen; donde L representa el máximo nivel de gris:

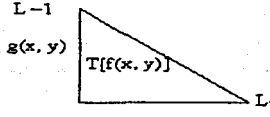


Figura 2.5. Negativo de una Imagen.

2.2.2.1.2. Estiramiento del Contraste.

No siempre es posible realizar una representación exacta de la brillantez de una imagen dentro de un rango determinado en la escala de grises, por eso, es necesario alisar la imagen para incrementar el rango dinámico de la imagen. Para ello se pueden elegir dos valores de *pixel*, digamos $a = f(x_1, y_1)$ y $b = f(x_2, y_2)$ que representarán el rango mínimo y máximo de alisamiento, de esta manera, la fórmula a aplicar es:

$$g(x,y) \begin{cases} 0 & si\ f(x,y) \le a \\ ((L-1) * f(x,y) - a)\ (b-a)\ si\ a \le f(x,y) \le b \\ L-1 & si\ f(x,y) \ge b \end{cases}$$

Donde $L = 2^n$ es la escala de grises; $a, b \in [0, L-1]$ y $a < b$. La gráfica típica de una imagen con alisamiento la muestra la figura 2.6.

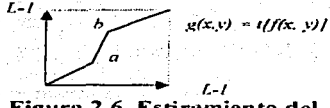


Figura 2.6. Estiramiento del Contraste.

2.2.2.1.3. Corte a Nivel de Grises.

El corte en escala de gris se refiere al proceso de asociar todos los valores de grises más altos a un valor constante, dentro de un rango determinado por los *pixeles a* y *b*. Mientras, los demás valores conservan su nivel de gris. Las figuras 2.7 y 2.8 representan dos ejemplos de este proceso.

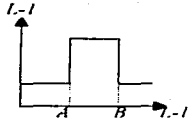


Figura 2.7. Corte a Nivel de Grises.

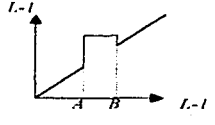


Figura 2.8. Corte a Nivel de Grises.

2.2.2.1.4. Corte a Nivel de Bits.

La imagen *f(x, y)* se compone de contribuciones de cada uno de los bits que componen el nivel de gris en el *pixel (x, y)*; así, el plano cero contiene las contribuciones más bajas y el plano siete las más altas. La figura 2.9 ilustra este concepto de corte a nivel de bits (*Bit plane Slicing*).

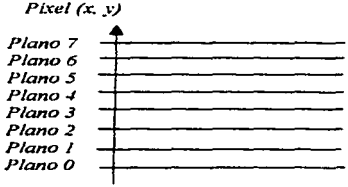


Figura 2.9. Bit Plane Slicing.

2.2.2.1.5. Histograma.

El histograma de una imagen está dado por la función discreta $p(r_k) = m_k / m$; donde r_k es el k-ésimo nivel de gris, m_k es el número de *píxeles* en la imagen con ese nivel de gris, y m es el número total de *píxeles* en la imagen. El parámetro K está dentro del rango $[0, L-1]$, donde $L = 2^n$ es la escala de grises; así entonces, la función $p(r_k)$ da un estimado de la ocurrencia del nivel de gris r_k. Las figura 2.10, 2.11, 2.12 y 2.13 muestran 4 casos típicos de histogramas:

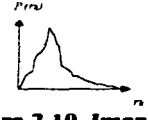


Figura 2.10. Imagen Oscura.

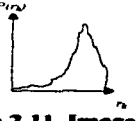


Figura 2.11. Imagen Brillante.

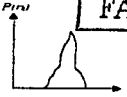


Figura 2.12. Imagen con Contraste Bajo.

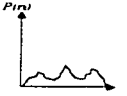


Figura 2.13. Imagen con Contraste Alto.

La brillantez y el contraste de la imagen, de tamaño $M \times N$, se calculan con las ecuaciones 2.1 y 2.2, respectivamente:

$$Brillantez(B) = (1/MN) \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} f(x,y) \quad \textbf{(Ec. 2.1)}$$

$$Contraste(C) = [(1/MN) \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} [f(x,y) - B]^2]^{1/2} \quad \textbf{(Ec. 2.2)}$$

2.2.2.1.5.1. Ecualización del Histograma.

Como se vio en las figuras anteriores, el histograma de una imagen no siempre es uniforme. Un gran porcentaje de *píxeles* se concentra en un rango relativamente pequeño en comparación con la escala de grises utilizada, es por esta razón que surgió la ecualización del histograma, que no es más que una forma de distribuir los niveles de grises sobre todo el rango dado por $[0, L-1]$ de manera uniforme. Así, la fórmula para la ecualización de un histograma es la ecuación 2.3:

$$S_k = T(r_k) = \sum_{j=0}^{k} (n_j/n) = \sum_{j=0}^{k} p_r(r_k) \quad \textbf{(Ec. 2.3)}$$

Donde $0 \leq r_k \leq 1$ y $k \in [0, L-1]$

2.2.2.1.6. Procesamiento a través del Color.

Muchas aplicaciones del PDI permiten procesar las bandas de color de una imagen, dichas bandas siempre cumplen con el modelo RGB, descrito en el capítulo anterior. El procesamiento de la imagen a través del color se divide en dos categorías: la primera corresponde al color verdadero (*full-color*), que es una representación real de los colores de la imagen. La segunda corresponde al procesamiento con un color falso (*pseudo-color*), que son una mezcla de intensidades del modelo RGB para dar una aproximación al color original de la imagen. Es en esta última donde enfocaremos el análisis.

Como se mencionó en el capítulo anterior, el estándar RGB es un modelo de aproximación a la forma en que la retina del ojo humano capta las tres longitudes de onda básicas y de cómo mezcla esas longitudes para formar un color. Básicamente el modelo RGB está representado en la figura 2.14.

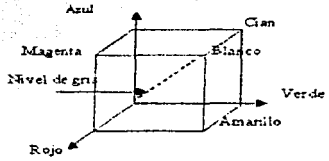


Figura 2.14. Modelo RGB.

Después de la digitalización de la imagen, es posible mantener por separado las bandas de color bajo el modelo RGB, con las cuales podemos hacer una serie de operaciones para transformarlas a otros modelos de color, así pues, tenemos el modelo HSI, cuya representación gráfica la muestra la figura 2.15:

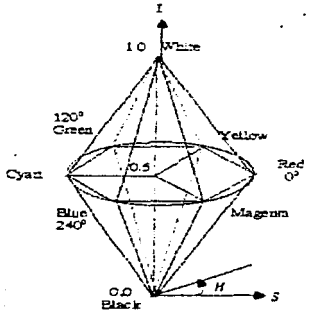


Figura 2.15. Modelo HSI.

De acuerdo con la anterior ilustración, las fórmulas para convertir del modelo RBG al modelo HSI están dada por las ecuaciones 2.4, 2.5 y 2.6:

$$I = (R + G + B) / 3 \qquad \textbf{(Ec. 2.4)}$$

$$S = 1 - [3 * mínimo(R, G, B) / (R + G + B)] \qquad \textbf{(Ec. 2.5)}$$

$$H = \cos^{-1}\{[1/2 * [(R-G) + (R-B)]] / [[(R-G)^2 + (R-B)(G-B)]^{1/2}]\} \qquad \textbf{(Ec. 2.6)}$$

Para regresar al modelo RGB, son las siguientes ecuaciones:

$$b = (1 - S) / 3 \qquad \textbf{(Ec. 2.7)}$$

$$r = 1/3 * [1 + (S * \cos(H) / \cos(60 - H))] \qquad \textbf{(Ec. 2.8)}$$

$$g = 1 - (r + b) \qquad \textbf{(Ec. 2.9)}$$

$$R = 3 * I * r \qquad \textbf{(Ec. 2.10)}$$

$$G = 3 * I * g \qquad \textbf{(Ec. 2.11)}$$

$$B = 3 * I * b \qquad \textbf{(Ec. 2.12)}$$

Donde *r, g, b* representan las coordenadas de cromacidad. Existen otros dos modelos muy utilizados que es necesario mencionar. El primero es el modelo CMY, utilizado para impresiones digitales, y cuya conversión a RGB es muy sencilla ya que ambos modelos son complementarios entre sí, por tanto, sólo es necesario sustraer la unidad del valor en cuestión para obtener su complemento, tal como se muestra en la figura 2.16.

$$\begin{vmatrix} C \\ M \\ Y \end{vmatrix} = \begin{vmatrix} 1 \\ 1 \\ 1 \end{vmatrix} - \begin{vmatrix} R \\ B \\ G \end{vmatrix}$$

Figura 2.16. Matriz de Conversión RGB a CMY.

El segundo es el modelo YIQ, utilizado para la transmisión de señales de televisión. La matriz correspondiente de valores YIQ a partir de un modelo RGB se muestra a continuación.

$$\begin{vmatrix} Y \\ I \\ Q \end{vmatrix} = \begin{vmatrix} 0.299 & 0.587 & 0.114 \\ 0.596 & -0.275 & -0.321 \\ 0.212 & -0.523 & 0.311 \end{vmatrix} \begin{vmatrix} R \\ B \\ G \end{vmatrix}$$

Figura 2.17. Matriz de Conversión RGB a YIQ.

2.2.2.1.7. Convolución.

La convolución es una operación local más recurrida en el análisis de imágenes. Básicamente consiste en escoger una máscara de tamaño finito *n x n* y mapearla sobre toda la imagen. El *pixel* de salida es el resultado de la sumatoria de los valores de la máscara multiplicado por las vecindades del *pixel (x, y)*, donde la máscara ha hecho coincidir su centro. Matemáticamente, la convolución se describe en la ecuación 2.13:

$$g(x, y) = f(x, y) \otimes h(a, b) = \sum_{a=0}^{j-1} \sum_{b=0}^{k-1} h(a, b) * f(x-a, y-b)$$

(Ec. 2.13)

Donde *h(a, b)* representan la máscara con dimensiones *j x k*. Bajo estos conceptos, a la máscara *H(a, b)* se le conoce como *kernel* de convolución; para ilustrar mejor estos conceptos considere el siguiente *kernel* de convolución de *3 x 3*:

$$H = \begin{vmatrix} H_1 & H_2 & H_3 \\ H_4 & H_5 & H_6 \\ H_7 & H_8 & H_9 \end{vmatrix}$$

Que multiplica a una región, en la imagen *f(x, y)* dada por

$$f(x, y) = \begin{vmatrix} f_1 & f_2 & f_3 \\ f_4 & f_5 & f_6 \\ f_7 & f_8 & f_9 \end{vmatrix}$$

Ahora, los nuevos valores para la imagen *g(x, y)* están dados por las siguientes formulaciones:

$$\begin{aligned}
G_1 &= H_5 f_1 + H_6 f_2 + H_8 f_4 + H_9 f_5 \\
G_2 &= H_4 f_1 + H_5 f_2 + H_6 f_3 + H_7 f_4 + H_8 f_5 + H_9 f_6 \\
G_3 &= H_4 f_2 + H_5 f_3 + H_7 f_5 + H_8 f_6 \\
G_4 &= H_2 f_1 + H_5 f_4 + H_8 f_7 + H_3 f_2 + H_6 f_5 + H_9 f_8 \\
G_5 &= H_1 f_1 + H_2 f_2 + H_3 f_3 + H_4 f_4 + H_5 f_5 + H_6 f_6 + H_7 f_7 + H_8 f_8 + H_9 f_9 \\
G_6 &= H_1 f_2 + H_4 f_3 + H_7 f_8 + H_2 f_3 + H_5 f_6 + H_8 f_9 \\
G_7 &= H_2 f_4 + H_5 f_7 + H_3 f_5 + H_6 f_8 \\
G_8 &= H_1 f_4 + H_2 f_5 + H_3 f_6 + H_4 f_7 + H_5 f_8 + H_6 f_9
\end{aligned}$$

$$G_0 = H_1 f_5 - H_2 f_6 - H_3 f_8 + H_5 f_0$$

2.2.2.1.8. Filtrado.

La inherente aparición de ruido en las imágenes es un problema que puede llegar a causar serios obstáculos al momento de hacer el análisis de los datos contenidos en la imagen, es por ello que se ha desarrollado una teoría matemática para eliminar el ruido. El ruido como tal corrompe las altas frecuencias en datos cuyas frecuencias son bajas, es por ello que se han desarrollado varios tipos de filtros, que operan de manera local sobre la imagen, es decir, convolucionan sobre ciertas regiones. A continuación se describe los principales filtros.

2.2.2.1.8.1. Filtros de Alisamiento.

El objetivo principal de los filtros de alisamiento (*Smoothing Filters*) es eliminar manchas o reducir el ruido dentro de la imagen, para ello se cuenta con varios máscaras o *kernels* de convolución. Entre ellos tenemos los siguientes.

2.2.2.1.8.1.1. Filtro Paso-Bajas.

Este filtro está compuesto por la respuesta impulso que le da la facilidad de manipular todos los coeficientes. Los *kernels* más utilizados son máscaras de *3x3*, *5x5* y *7x7*, con un factor de normalización de *1/9*, *1/25* y *1/49* respectivamente. Entre mayor sea el *kernel* a utilizar mejores serán los resultados. La figura 2.18 muestra una máscara clásica de *5 x 5*.

$$1/25 \begin{vmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{vmatrix} .$$

Figura 2.18. Kernel Paso-Bajas.

Básicamente, el *kernel* de este filtro es centrado sobre el *píxel* de la imagen *(x, y)* y se sustituye su valor actual por el promedio de sus vecindades. Es por ello que se le atribuye el nombre de filtro de promedios.

2.2.2.1.8.1.2. Filtro de Medianas.

El filtro de promedios es muy efectivo en varios tipos de imágenes, no obstante se vuelve ineficiente cuando se trata de eliminar ruido en bordes de objetos contenidos en la imagen, por ellos el filtro de medianas es el más utilizado ya que en lugar de reemplazar al *píxel (x, y)* con el promedio de sus vecindades, utiliza la mediana de las vecindades (*Median Filter*), así, el *kernel* se compone con la mitad de los valores menores a la mediana del *píxel (x, y)*, y la otra mitad con valores mayores al mismo.

2.2.2.1.8.2. Filtros de Afinamiento.

El objetivo de los filtros de afinamiento (*Sharpening Filtres*) es resaltar líneas o bordes de objetos dentro de la imagen, independientemente que posean ruido o no. Los filtros más comunes para este propósito son los filtros paso-altas y los filtros derivativos, los cuales se utilizan para la detección de bordes.

2.2.2.1.8.2.1. Filtro Paso-Altas.

Para la construcción del filtro de paso-altas es necesario recurrir a la señal impulso y adecuarla para que sólo existan coeficientes positivos en el centro del *kernel* y coeficientes negativos en sus vecindades, por ejemplo, un *kernel* paso-altas de *3x3* sería como en la figura 2.19.

$$\frac{1}{9}\begin{vmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{vmatrix}$$

Figura 2.19. Kernel Paso-Altas.

Calcular este tipo de filtros a veces conlleva una serie de operaciones matemáticas muy extensas, por esta razón, se ha buscado una alternativa de implementación conocido como filtro de empuje (*high-pass boosts filter*), que se compone de la resta de la imagen original con la misma imagen pero filtrada con pasa-bajas y multiplicada por un factor de amplificación, tal como lo describe la ecuación 2.14:

$$Filtro\ de\ empuje = (A-1)f(x, y) - f_{low}(x, y) \qquad \textbf{(Ec. 2.14)}$$

Donde A representa el factor de amplificación o ganancia, *f(x, y)* la imagen original y $f_{low}(x, y)$ es la imagen filtrada por un paso-bajas. La recomendación para el factor de ganancia es que oscile entre valores de 0 a 2, así, el anterior *kernel* queda como se muestra en la figura 2.20.

$$A/9 \begin{vmatrix} -1 & -1 & -1 \\ -1 & 8+9/A & -1 \\ -1 & -1 & -1 \end{vmatrix}$$

Figura 2.20. Filtro de Empuje.

La máscara más utilizada para el filtro de empuje es una matriz de *7x7* con un factor de normalización de *1/49*, y cuyo valor central es de *48*, rodeado de coeficientes negativos unitarios.

2.2.2.1.8.2.2. Filtros Derivativos.

Este tipo de filtro se usa principalmente para detectar orillas o bordes dentro de una imagen utilizando las cualidades que presenta el gradiente, así, es posible encontrar la dirección de los vectores. En la sección siguiente[3] profundizaremos un poco más sobre los aspectos matemáticos y los *kernels* más utilizados para este tipo de operaciones.

2.2.2.1.9. Otras Operaciones.

Existe otro tipo de operaciones que se pueden aplicar a una imagen tales como operaciones aritméticas u operaciones binarias. Cada una de estas operaciones se realizan de *píxel* a *píxel*, por lo que son operaciones puntuales, como todas las anteriores. La siguiente tabla resume todas las operaciones binarias y aritméticas posibles a efectuar en una imagen. Considere a *f(x, y)* y *h(x, y)* como imágenes de entrada y a *g(x, y)* como la imagen de salida. La tabla 2.1 muestra las operaciones existentes.

[3] En la sección 2.2.3

Tipo	Operación	Definición	Explicación
Binaria	NOT	g(x, y) = NOT[f(x, y)]	Obtiene el Negativo de la Imagen
Binaria	AND	g(x, y) = f(x, y) AND h(x, y)	Calcula la operación lógica AND entre dos imágenes.
Binaria	OR	G(x, y) = f(x, y) OR h(x, y)	Calcula la operación lógica OR entre dos imágenes.
Binaria	XOR	g(x, y) = f(x, y) XOR h(x, y)]	Calcula la operación lógica XOR entre dos imágenes.
Binaria	SUB	g(x, y) = f(x, y) AND NOT[h(x, y)]	Calcula la operación lógica de resta entre dos imágenes.
Aritmética	Suma	g(x, y) = f(x, y) + h(x, y)	Calcula la suma de dos imágenes.
Aritmética	Resta	g(x, y) = f(x, y) - h(x, y)	Calcula la resta de dos imágenes.
Aritmética	Multiplicación	g(x, y) = f(x, y) * h(x, y)	Calcula la multiplicación de dos imágenes.
Aritmética	División	g(x, y) = f(x, y) / h(x, y)	Calcula la división de dos imágenes.
Aritmética	Logaritmo	g(x, y) = Log(f(x, y))	Calcula el logaritmo de una imagen.
Aritmética	Raiz	g(x, y) = [f(x, y)]$^{1/2}$	Calcula la raiz de una imagen.
Aritmética	Trigonométrica	g(x, y) = sen /cos /tan[f(x, y)]	Aplica una función trigonométrica a una imagen para obtener otra.
Aritmética	Negativo	g(x, y) = L-1- f(x, y)	Calcula el negativo de una imagen.

Tabla 2.1

Note que las operaciones aritméticas se realizan en imágenes cuyos valores de intensidad son cantidades enteras o reales, esto significa que tiene sus

respectivas bandas de color, así pues, las operaciones aritméticas se realizan a nivel color, mientras que en una imagen digitalizada se eliminan las bandas de color y se traduce a una escala de grises, así, es posible aplicarle operaciones binarias. Nótese, además, que las operaciones de *NOT* y *Negativo* obtienen el mismo resultado, únicamente difieren por el tipo de valor que contienen sus *píxeles*.

2.2.2.2. Dominio de la Frecuencia.

Las imágenes se encuentran, por omisión, en el dominio del espacio, para llevarlas al dominio de la frecuencia es necesario aplicarles una serie de transformaciones análogas a la transformación de señales. Una vez localizadas en el dominio de la frecuencia se debe aplicar una operación y al resultado debe aplicársele una transformación inversa para obtener su correspondiente en el dominio del espacio. Es importante recalcar que una imagen localizada en el dominio de la frecuencia no puede ser visualizada ya que no contiene ninguna información:

2.2.2.2.1. La Transformada Discreta de Fourier.

Para explicar la Transformada de *Fourier* es necesario explicar algunos conceptos matemáticos. Suponga una función $f(x)$ continua sobre el espacio x, la transformada de *Fourier* se define como:

$$\Im\{f(x)\} = F(u) = \int f(x)\ exp^{-j2\pi ux}\, dx \qquad \textbf{(Ec. 2.15)}$$

Donde $exp^{-j2\pi ux} = cos(2\pi ux) - j\ sen(2\pi ux)$, de acuerdo con el teorema de *Euler*. Podemos destacar algunas propiedades de la transformada de *Fourier* si aplicamos el teorema de *Euler* como sigue: $F(u) = R(u) - j\ I(u)$, donde $R(u)$ representa la parte real e $I(u)$ la parte imaginaria, así pues, tenemos que el espectro de *Fourier* está dado por:

$$|F(u)| = \sqrt{R^2(u) + I^2(u)} \qquad \textbf{(Ec. 2.16)}$$

Además, el ángulo de fase, $\Phi(u)$, y el poder del espectro, $P(u)$, se calculan con las ecuaciones 2.17 y 2.18, respectivamente:

$$\Phi(u) = \tan^{-1}\left[\frac{I(u)}{R(u)}\right] \qquad \textbf{(Ec. 2.17)}$$
$$P(u) = F(u) \qquad \textbf{(Ec. 2.18)}$$

La respectiva transformación inversa de *Fourier* tiene la siguiente ecuación:

$$\mathcal{F}^{-1}\{F(u)\} = f(x) = \int F(u)\ exp^{j2\pi ux}\, du \qquad \textbf{(Ec. 2.19)}$$

Hablando de una señal bidimensional, $f(x, y)$, la Transformada de *Fourier* y su respectiva inversa están definidas por las ecuaciones 2.20 y 2.21, respectivamente.

$$\mathcal{F}\{f(x, y)\} = F(u, v) = \iint f(x, y)\ exp^{-j2\pi(ux + vy)}\, dx\, dy \qquad \textbf{(Ec. 2.20)}$$
$$\mathcal{F}^{-1}\{F(u, v)\} = f(x, y) = \iint F(u, v)\ exp^{j2\pi(ux + vy)}\, du\, dv \qquad \textbf{(Ec. 2.21)}$$

Todas estas ecuaciones se realizan en espacios continuos, sin embargo, necesitamos aplicar estas mismas fórmulas a señales discretizadas, de esta manera si discretizamos la señal $f(x)$ en una secuencia de intervalos iguales, Δx, y tomamos N muestras, entonces la función $f(x)$ se define como:

$$f(x) = \{f(x_0 + x\Delta x) = \{f(x_0), f(x_0 + \Delta x), f(x_0 + 2\Delta x), \ldots, f(x_0 + [N-1]\Delta x)\}$$

Para $x = 0, 1, 2, 3, 4, 5, \ldots\ldots\ldots N-1$ muestras. Ahora podemos aplicar la ecuación 2.15 a una señal discretizada como lo indica la ecuación 2.22:

$$\mathcal{F}\{f(x)\} = F(u) = \sum_{x=0}^{N-1} f(x)\ exp^{-j2\pi ux/N} \qquad \textbf{(Ec. 2.22)}$$
$$\text{para } x = 0, 1, 2, \ldots\ldots, N-1$$

Aplicando lo mismo a la ecuación 2.21, obtenemos la inversa para una señal discreta:

$$\mathcal{F}^{-1}\{F(u)\} = f(x) = \sum_{x=0}^{N-1} F(u)\ exp^{j2\pi ux/N} \qquad \textbf{(Ec. 2.23)}$$
$$\text{para } u = 0, 1, 2, \ldots\ldots, N-1$$

Para una imagen, o señal bidimensional, discretizada en MxN muestras, la transformada de *Fourier*, y su inversa, dada por las ecuaciones 2.24 y 2.25, respectivamente, están definidas como:

$\Im\{f(x, y)\}$ $F(u, v) = 1/MN \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} f(x, y) \exp^{-j2\pi ux/M + vy/N)}$

(Ec. 2.24)

para $x = 0, 1, 2, \ldots, M-1$; $y = 0, 1, 2, \ldots, N-1$

$\Im^{-1}\{F(u, v)\} = f(x, y) = 1/MN \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} F(u, v) \exp^{j2\pi ux/M + vy/N)}$

(Ec. 2.25)

para $u = 0, 1, 2, \ldots, M-1$; $v = 0, 1, 2, \ldots, N-1$

Con estas ecuaciones es posible crear un algoritmo denominado Transformación Rápida de *Fourier* (*FFT*, *Fast Fourier Transform*), y que la mayoría de las aplicaciones de PDI ya la traen implementada.

2.2.2.2.2. La Transformada de Walsh.

Otra transformada muy recurrida es la transformada de *Walsh*; la cual define los *kernels* a utilizar, con dimensiones de $n \times n$ aplicada a una señal de N muestras, de esta manera, la transformada de *Walsh*, y su inversa, se definen en las ecuaciones 2.26 y 2.27:

$W(u) = 1/N \sum_{x=0}^{N-1} f(x) g(x, u)$ **(Ec. 2.26)**

$f(x) = \sum_{u=0}^{N-1} W(u) h(x, u)$ **(Ec. 2.27)**

Donde $g(x, u)$ y $h(x, u)$ son *kernels* de $n \times n$ dimensiones, y tiene la forma:

$g(x, u) = \Pi_{i=0}^{n-1} (-1)^{a}$; $a = b_i(x) b_{n-1-i}(u)$

$h(x, u) = \Pi_{i=0}^{n-1} (-1)^{a}$; $a = b_i(x) b_{n-1-i}(u)$

En dos dimensiones tenemos que la transformada de *Walsh* y su inversa están dadsa por las ecuaciones 2.28 y 2.29, respectivamente:

$W(u, v) = 1/MN \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} f(x, y) g(x, y, u, v)$ **(Ec. 2.28)**

$f(x, y) = 1/MN \sum_{u=0}^{M-1} \sum_{v=0}^{N-1} W(u, v) h(x, y, u, v)$ **(Ec. 2.29)**

Con sus respectivos *kernels*:

$g(x, y, u, v) = \Pi_{i=0}^{N-1} (-1)^{a}$; $a = b_i(x) b_{n-1-i}(u) + b_i(y) b_{n-1-i}(v)$

$h(x, y, u, v) = \Pi_{i=0}^{N-1} (-1)^{a}$; $a = b_i(x) b_{n-1-i}(u) + b_i(y) b_{n-1-i}(v)$

2.2.2.2.3. La Transformada del Coseno.

La Transformación del coseno y su inversa, están definidas, para una señal, como:

$$C(u) = \alpha(u) \sum_{x=0}^{N-1} f(x) \cos[\pi u(2x+1) / 2N] \quad \textbf{(Ec. 2.30)}$$
$$f(x) = \sum_{u=0}^{N-1} \alpha(u) C(u) \cos[\pi u(2x+1) / 2N] \quad \textbf{(Ec. 2.31)}$$

Donde u = 0, 1, 2, ..., N-1 y $x = 0, 1, 2, ..., N-1$. Para una imagen:

$$C(u, v) = \alpha(u)\alpha(v) \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} f(x, y) \cos[\pi u(2x+1) / 2N] \cos[\pi v(2y+1) / 2N] \quad \textbf{(Ec. 2.32)}$$

$$f(x, y) = \sum_{u=0}^{M-1} \sum_{v=0}^{N-1} \alpha(u)\alpha(v) C(u, v) \cos[\pi u(2x+1) / 2N] \cos[\pi v(2y+1) / 2N] \quad \textbf{(Ec. 2.33)}$$

Con $u = 0, 1, 2, ..., N-1$, $v = 0, 1, 2, ..., N-1$, $x = 0, 1, 2, ..., M-1$ y $y = 0, 1, 2, ..., N-1$. Además,

$$\alpha(u) = \begin{cases} \sqrt{1/M} & si\ u = 0 \\ \sqrt{2/M} & si\ u > 0 \end{cases} \quad y \quad \alpha(v) = \begin{cases} \sqrt{1/N} & si\ v = 0 \\ \sqrt{2/N} & si\ v > 0 \end{cases}$$

2.2.2.2.4. La Transformada de Hadamard.

Al igual que la transformada de *Walsh*, *Hadamard* define sus propios *kernels* para aplicarlos en sus transformaciones. Los *kernels* para señales están dados por las ecuaciones 2.34 y 2.35, mientras que los kernel para imágenes están dados por las ecuaciones 2.36 y 2.37:

$$g(x, u) = 1/N (-1)^a \quad ; \quad a = \sum_{i=0}^{n-1} b_i(x) b_i(u) \quad \textbf{(Ec. 2.34)}$$
$$h(x, u) = (-1)^a \quad ; \quad a = \sum_{i=0}^{n-1} b_i(x) b_i(u) \quad \textbf{(Ec. 2.35)}$$
$$g(x, y, u, v) = 1/N (-1)^a \quad ; \quad a = \sum_{i=0}^{n-1} b_i(x) b_i(u) + b_i(y) b_i(v) \quad \textbf{(Ec. 2.36)}$$
$$h(x, y, u, v) = 1/N (-1)^a \quad ; \quad a = \sum_{i=0}^{n-1} b_i(x) b_i(u) + b_i(y) b_i(v) \quad \textbf{(Ec. 2.37)}$$

Aplicando los *kernels* de las ecuaciones 2.34 y 2.35 a sus respectivas transformadas tenemos que, para una dimensión, la transformada de *Hadamard*, con su inversa, están dadas por las ecuaciones 2.38 y 2.39, respectivamente:

$$H(u) = \sum_{x=0}^{N-1} f(x)\, g(x, u) \qquad \textbf{(Ec. 2.38)}$$
$$f(x) = \sum_{u=0}^{N-1} H(u)\, h(x, u) \qquad \textbf{(Ec. 2.39)}$$

En el caso de una imagen, aplicamos los *kernels* de las ecuaciones 2.36 y 2.37 para obtener la transformada de *Hadamard*, con su inversa.

$$H(u, v) = \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} f(x, y)\, g(x, y, u, v) \qquad \textbf{(Ec. 2.40)}$$
$$f(x, y) = \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} H(u, v)\, h(x, y, u, v) \qquad \textbf{(Ec. 2.41)}$$

2.2.2.2.5. Filtrado.

Una vez que la imagen ya se encuentra en el domino de la frecuencia es posible aplicarles algunos filtros, como el filtro de *Butterworth*, tanto para filtrar frecuencias altas y bajas, esto es, $G(u, v) = H(u, v)F(u, v)$. Donde. $G(u, v)$ es la imagen resultante. $F(u, v)$ es la imagen original en el dominio de la frecuencia, y $H(u, v)$ es el filtro a aplicar. Para filtrar frecuencias bajas, el filtro $H(u, v)$ tiene la siguiente forma:

$$H(u, v) = \begin{cases} 1 & si \quad D(u, v) \le D_0 \\ 0 & si \quad D(u, v) > D_0 \end{cases}$$

Donde $D(u, v)$ representa la distancia del punto (u, v) al origen del sistema coordenado, es decir. $D(u, v) = [u^2 + v^2]^{1/2}$ y D_0 es un coeficiente no negativo. llamado frecuencia de corte. En el caso de un filtro paso-altas, la definición de $H(u, v)$ es:

$$H(u, v) = \begin{cases} 0 & si \quad D(u, v) \le D_0 \\ 1 & si \quad D(u, v) > D_0 \end{cases}$$

En ambas definiciones es posible aplicar las formulaciones de *Butterworth* a $H(u, v)$. Así, tenemos que las ecuaciones 2.42 y 2.43 representan, respectivamente, las fórmulas para un filtro paso-bajas y para un filtro paso-altas.

$$H(u, v) = 1 / (1 + [D(u, v) / D_0]^{2n}) \qquad \textbf{(Ec. 2.42)}$$
$$H(u, v) = 1 / (1 + [D_0 / D(u, v)]^{2n}) \qquad \textbf{(Ec. 2.43)}$$

Como se ha visto a lo largo de esta sección, existen una gran cantidad de herramientas disponibles para el procesamiento de una imagen, donde cada una de ellas proporciona los elementos suficientes para cubrir con las tareas que se requieren.

2.2.3. Segmentación de Imágenes.

Al principio del tema se había comentado acerca de las áreas de estudio del PDI, una de ellas es el reconocimiento de patrones. Esta área se auxilia de varias herramientas para detectar patrones dentro de una imagen usando máscaras que recorren la imagen. Las máscaras pueden tener distintas finalidades, por ejemplo, existen máscara para detectar discontinuidades en los *píxeles*, un segundo tipo corresponde a la detección de líneas, y, finalmente, un tercer tipo para la detección de fronteras. Estas máscaras son de *3 x 3* y la respuesta sobre la imagen se puede calcular como:

$$R = \sum_{i=0}^{9} w_i z_i$$

Donde w_i es el *i-ésimo* coeficiente dentro de la máscara, z_i es el *i-ésimo píxel* superpuesto en la máscara. El centro de la máscara debe estar localizado en el *píxel (x, y)* en cuestión. La máscara más utilizada para la detección de puntos es la que muestra la figura 2.21.

$$M = \begin{vmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{vmatrix}$$

Figura 2.21. Kernel para la Detección de Puntos.

Donde la relación $|R| > T$, donde T es un umbral no negativo. En el caso de la detección de líneas, tenemos cuatro máscaras, representadas en las figuras 2.22, 2.23, 2.24 y 2.25.

$$M = \begin{vmatrix} -1 & -1 & -1 \\ 2 & 2 & 2 \\ -1 & -1 & -1 \end{vmatrix}$$

Figura 2.22. Detección de Líneas Horizontales.

$$M = \begin{vmatrix} -1 & -1 & 2 \\ -1 & 2 & -1 \\ 2 & -1 & -1 \end{vmatrix}$$

Figura 2.23. Detección de Líneas Diagonales.

$$M = \begin{vmatrix} -1 & 2 & -1 \\ -1 & 2 & -1 \\ -1 & 2 & -1 \end{vmatrix}$$

Figura 2.24. Detección de Líneas Verticales.

$$M = \begin{vmatrix} 2 & -1 & -1 \\ -1 & 2 & -1 \\ -1 & -1 & 2 \end{vmatrix}$$

Figura 2.25. Detección de Líneas Diagonales.

2.2.3.1. Detección de Bordes.

Existe un sin número de herramientas que se especializan en detectar bordes. las que analizaremos a continuación utilizan operadores derivativos para determinar que regiones existen dentro de una imagen. Primeramente, un borde se define como una frontera entre dos regiones con características muy diferentes entre sí. Suponga una imagen con un recuadro negro centrado sobre un fondo blanco, si nosotros trazamos una línea horizontal de extremo a extremo y graficamos su perfil de grises, encontraremos que existe un cambio abrupto cuando la línea hace sus respectivas transiciones de blanco a negro y de negro a blanco. Además, si calculamos la primera y la segunda derivada de este perfil, encontraremos los máximos y mínimos dentro de esa imagen. Las figuras 2.26, 2.27, 2.28 y 2.29 representan este concepto.

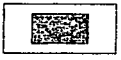


Figura 2.26. Imagen con Alto Contraste.

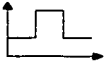


Figura 2.27. Perfil de Grises.

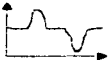


Figura 2.28. Primera Derivada.

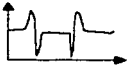


Figura 2.29. Segunda Derivada.

2.2.3.1.1. Operador Gradiente.

El gradiente de una imagen está definido por la ecuación 2.44.

$$\nabla f = [G_x \quad G_y]^T = [\delta f / \delta x \quad \delta f / \delta y]^T \qquad \textbf{(Ec. 2.44)}$$

Esta ecuación indica la dirección del cambio en el *pixel (x, y)*. Otros dos valores de importancia es la magnitud del gradiente y el ángulo de dirección, que se calculan como:

$$\nabla f = mag\ (\nabla f) = [G^2_x + G^2_y]^{1/2} \qquad \textbf{(Ec. 2.45)}$$
$$\alpha(x, y) = \tan^{-1} [G_y / G_x] \qquad \textbf{(Ec. 2.46)}$$

Donde $G_x = (z_7 + 2z_8 + z_9) - (z_1 + 2z_2 + z_3)$ y $G_y = (z_3 + 2z_6 + z_9) - (z_1 + 2z_4 + z_7)$. Transformando las ecuaciones 2.45 y 2.46 a un modelo matricial tenemos a los operadores de *Sobel*, cuyas máscaras están representadas en las figuras 2.30 y 2.31.

$$S_1 = \begin{vmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{vmatrix} \qquad S_2 = \begin{vmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{vmatrix}$$

Figura 2.30. Kernel de Sobel. **Figura 2.31. Kernel de Sobel.**

2.2.2.3.1.2. Laplaciano.

El Laplaciano de una función dimensional es una derivada de segundo orden y se define como en la ecuación 2.47.

$$\nabla^2 f = (\delta f^2 / \delta x^2) + (\delta f^2 / \delta y^2) \qquad \textbf{(Ec. 2.47)}$$

La ecuación 2.47 se puede implementar de varias maneras, la más común es una máscara de *3 x 3*, con la ecuación 2.48:

$$\nabla^2 f = 4z_5 - (z_2 + z_4 + z_6 + z_8) \qquad \textbf{(Ec. 2.48)}$$

El Laplaciano es muy sensitivo al ruido por lo que es muy poco utilizada, sin embargo, se utiliza una propiedad llamara *zero-crossings* para realizar la

detección de bordes. Este concepto consiste en convolver una imagen con el Laplaciano de una función *gaussiana*, definida por la ecuación 2.49.

$$h(x, y) = exp^{\alpha} \qquad \textbf{(Ec. 2.49)}$$

Donde $\alpha = -x^2 + y^2 \quad 2\sigma^2$ y σ es la desviación estándar. Si hacemos a $r^2 = x^2 + y^2$, entonces el Laplaciano de $h(x, y)$ estará dado por la segunda derivada con respecto a r:

$$\nabla^2 h = (\delta h^2 \quad \delta r^2) + [r^2 + \sigma^2 \cdot \sigma^4] / exp^{\alpha}$$

Donde $\alpha = -r^2 \quad 2\sigma^2$. La técnica del *zero-crossings* es muy recurrida debido a las ventajas que muestra cuando las orillas o bordes son borrosos o existe una gran cantidad de ruido, no obstante, los cálculos que involucra son demasiado complejos. La máscara clásica de esta técnica está representada por la figura 2.132.

$$L = \begin{vmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{vmatrix}$$

Figura 2.32. Kernel del Laplaciano.

2.2.2.3.1.3. Umbral.

La manera más fácil de separar objetos contenidos en un fondo constante es elegir un umbral T, tal que separe esas regiones, así podemos sacar una relación sencilla para cada *píxel* (x, y) en la imagen:

$$g(x, y) \begin{cases} 1 & si\ f(x, y) > T \\ 0 & si\ f(x, y) \leq T \end{cases}$$

También es posible utilizar mas de un umbral para llevar acabo la operación de segmentación. Por ejemplo, si se eligen dos umbrales, T_1 y T_2, tal que $T_1 \neq T_2$, entonces podemos sacar una relación como la anterior, de tal suerte que se cumpla $T_1 < f(x, y) < T_2$. La forma más eficiente se logra a través de

formar una relación entre el umbral, el gradiente y el Laplaciano, tal como se muestra:

$$g(x, y) = \begin{cases} 0 & si \quad \nabla f < T \\ - & si \quad \nabla f \geq T \quad y \quad \nabla^2 f \geq 0 \\ - & si \quad \nabla f \geq T \quad y \quad \nabla^2 f < 0 \end{cases}$$

2.2.3.2. Segmentación Orientada a Regiones.

Considérese una imagen R, la cual se puede dividir en *n* subregiones, tal que cumpla con las siguientes condiciones:

1. $\bigcup_{i=0}^{n} R_i = R$.
2. R_i es una región conectada.
3. $R_i \cap R_j = Q$ para toda *i* y *j*, cuando $i \neq j$.
4. $P(R_i)$ =Verdadero para toda $i = 1, 2, 3, \ldots, n$.
5. $P(R_i \cup R_j)$ = Falso para $i \neq j$.

Donde $P(R_i)$ es un predicado lógico sobre los puntos R_i en el conjunto, y Q es el conjunto nulo. Existen dos técnicas básicas para implementar estas instrucciones: la agregación de *píxeles* (*pixel aggregation*), y el método de *splitting and merging*.

2.3. *Volume Rendering.*

2.3.1. *Definición.*

Primero es necesario definir que significa la palabra inglesa *render*. De acuerdo con la organización internacional del *Siggraph*[4], esta palabra se usa para describir al proceso de convertir polígonos, u otras primitivas, en una imagen tomando en cuenta su color y opacidad. Mientras que la palabra *Volume* se refiere al conjunto de datos en arreglos escalares o vectoriales enclavados en un espacio de *n* dimensiones, siendo la configuración más simple y más utilizada el espacio tridimensional.

De acuerdo con lo anterior, podemos definir al *VR* como el proceso de transformar los datos de *n* dimensiones en una imagen, formalmente, el VR consiste en proyectar datos tridimensionales en una imagen bidimensional, sin que esta pierda su apariencia de tridimensionalidad; a decir del *Siggraph*, el VR es una técnica que no necesita representaciones intermedias para visualizar datos, es decir, no se auxilia de geometrías ni de otras primitivas, sino que hace una visualización directa de los datos.

Los datos pueden estar distribuidos en una malla, que puede ser rectangular o de cualquier otro tipo. En el caso de las mallas rectangulares, los datos están repartidos de manera uniforme por lo que es posible dividir al volumen en regiones, así, encontramos que la región mínima de un conjunto de datos volumétricos es el *voxel* (*Volume Element*). Esta unidad se puede interpretar como un punto dentro del espacio tridimensional con coordenadas (*x, y, z*) con un valor asociado a él, *f(x, y, z)*. Aunque algunos autores prefieren ut el término *voxel* un promedio de valores de una región dada. En este sentido, otros autores prefieren utilizar el término célula computacional definir la región mínima dentro de los datos volumétricos; en este cas célula está definida en un espacio dentro de un número determinado de vecindades, por ejemplo, en una representación con malla rectangular, una célula estaría formada por 8 vecindades. Para este trabajo tomaremos la

[4] La dirección en Internet es www siggraph.org

definición *voxel*, ya que es la que más se adapta a los conceptos con los que se trabajarán. Las figuras 2.33 y 2.34 ilustran los conceptos de *voxel* y de célula, respectivamente:

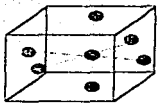


Figura 2.33. Célula.

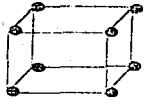


Figura 2.34. Voxel.

La técnica del VR se enfrenta a cinco dilemas, a saber, la gran cantidad demanda para el almacenamiento de datos volumétricos, la cantidad de memoria que se necesita para su procesamiento, el mapeo de múltiples valores del eje z sobre una sola posición (x, y) dentro del área de despliegue, el manejo de números reales y la posibilidad de crear hoyos en la representación final cuando la correspondencia entre *voxeles* y *píxeles* no es la adecuada. Para resolver el problema del almacenamiento, la Visualización Científica se auxilia del PDI para desarrollar algoritmos de compresión y aplicarlos en el almacenamiento de datos en general. En el segundo problema, la teoría de la computación ha desarrollado algoritmos más eficientes para la administración de la memoria, tales como paginación o el uso de estructuras jerárquicas. En el sentido estricto estos dos problemas no conciernen al VR, sin embargo, es necesario considerarlos.

El problema de proyectar múltiples valores del eje Z sobre un área de despliegue se resuelve a través de la función de composición, la cual colapsa todos los valores de Z y determina cual valor debe ser asignado a dicha área de despliegue. La función de composición es muy importante en el VR y más adelante, en la sección 2.5.4, se hablará de este tema.

En el primer capítulo se habló de las matrices de transformación existente, es evidente que todas ellas operan en el dominio de los números reales, bueno, ¿qué pasa con la parte fraccionaria cuando se desea proyectar el volumen, después de haber sufrido una transformación?. La solución de este problema está en la aplicación de filtros de tal manera que se tome en cuenta la suma de todas las contribuciones de los *voxeles* al momento de ser mapeadas. Como se

vio en la sección de PDI, existen varias máscaras de filtros que permiten sumar los pesos de los coeficientes evitando la pérdida de datos.

Finalmente, a veces la proyección de *voxeles* sobre *píxeles* puede producir pérdidas de datos y, por tanto, huecos en la imagen final, sobretodo cuando se ha aplicado una transformación al volumen. Para evitar esta situación es necesario determinar la proporción que cada *voxel* tiene sobre un *píxel* determinado.

Una vez finalizado el tema de los datos volumétricos y sus problemas, es el momento de explicar, de manera sencilla, en que consiste el VR. Sabemos que se trata de proyectar coordenadas de tres dimensiones en dos dimensiones, pero ¿cómo?. La literatura pone como ejemplo clásico una radiografía de rayos X. Como todos saben, una radiografía es una imagen donde se muestran los órganos internos de un cuerpo, algunos con mayor claridad que otros, esto se debe a que la absorción de rayos X varía de acuerdo al "material" o textura de los órganos. Cada órgano emite, reflecta o absorbe cierta cantidad de rayos X, es decir, contribuye para determinar la intensidad final dentro de la radiografía. Si se dividiera la radiografía en cuadrículas diminutas, y por cada una de ellas se lanzará un rayo X a través del cuerpo, el resultado final sobre cada cuadrícula será la suma de todas las contribuciones hechas por cada órgano a lo largo del rayo. Así es como funciona el *VR*, un determinado número de rayos atraviesa los datos volumétricos con el fin de obtener la contribución de cada *voxel*, y por ende, la proyección sobre un *píxel*.

Sin embargo, el proceso de *VR* no es tan simple, por ello es necesario definir un flujo de tareas a realizar para obtener una buena imagen a partir de datos volumétricos. A este flujo de tareas se le conoce como *pipeline* del *VR*.

2.3.2. Pipeline.

Básicamente existen dos modelos de flujo que representan el *pipeline* el VR: el primero, es un modelo general que puede aplicarse a la VC y es mostrado en la figura 2.35.

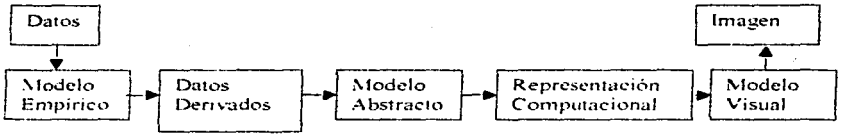


Figura 2.35. Pipeline del Volume Rendering.

Como se puede ver en el este *pipeline*, la generalización es evidente y casi describe el proceso de la Visualización Científica, pero si se aplica al *VR*, cada modelo tiene tareas específicas que se describen a continuación. El modelo empírico se encarga de hacer las tareas de interpolación y normalización de los datos en su forma original, asimismo, está encomendado de aplicar las transformaciones correspondientes. El modelo abstracto se encarga de determinar las funciones de transferencia aplicables a los datos volumétricos, así como de la asignación de color y opacidad a cada *voxel* y la identificación de superficies dentro del volumen. Por último, el modelo visual se encarga de aplicar el proceso de *render* final, dar la perspectiva adecuada y sombrear de acuerdo al modelo de iluminación.

El siguiente *pipeline* es un modelo propuesto por *Barthold Lichtenbelt* y es más particular para el *VR*. La figura 2.36 muestra este modelo.

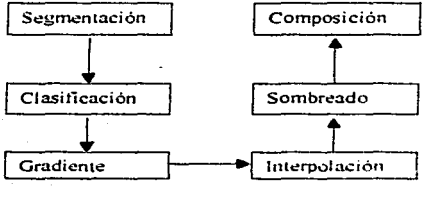


Figura 2.36. Pipeline del Volume Rendering.

Este modelo de *pipeline* consiste de seis etapas, las cuales pueden variar en el orden de ejecución. La segmentación se encarga de separar los datos

volumétricos en unidades estructurales a fin de obtener entendimiento sobre las características propias de los datos. La clasificación se utiliza para etiquetar los voxeles asignándoles un cierto color y opacidad de acuerdo a sus propiedades. El gradiente se utiliza para detectar orillas o fronteras, tal como en el PDI. La interpolación está encargada de generar nuevos valores entre cada *voxel* de acuerdo con la trayectoria del rayo que los atraviesa. En el sombreado (*shading*) se aplican las técnicas del modelo de iluminación para identificar partes del volumen. La composición se refiere a la etapa de colapsamiento de las contribuciones de cada *voxel* usando una función de composición.

2.3.3. Interpolación.

Ambos *pipeline* muestran las etapas básicas para conseguir un buen proceso de *VR*, no obstante, se debe hacer notar que las diferentes implementaciones de esta técnica, dentro de los paquetes de VC, difieren en el orden de los pasos, por lo que no nos basaremos en los anteriores modelos de flujo para describir cada etapa del *VR*. En vez de ello haremos una descripción de cada etapa, de acuerdo al orden de importancia, y empezaremos por la interpolación.

Cuando los datos volumétricos están contenidos en una malla rectangular, esta se puede dividir en regiones uniformes, así, un *voxel* puede estar compuesto por una región formada por ocho vértices; a veces es necesario calcular el valor contenido dentro de esa región cúbica, tomando en cuenta las vecindades, o vértices, que rodean ese punto; a este proceso se le conoce como interpolación. La interpolación provee *kernels* con coeficientes de contribución que ayuda a calcular el nuevo valor. Estos *kernels* funcionan de la misma manera que en el PDI, es decir, el centro del *kernel* coincide con el punto donde se va a calcular el nuevo valor. Los *kernels* de interpolación pueden ser definidos para *n* dimensiones, en el caso de un espacio tridimensional, el proceso de interpolación se describe a continuación. Sea una región definida por ocho vecindades y se desea calcular el punto X dentro de esa región, tal como se muestra en la figura 2.37.

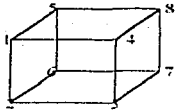


Figura 2.37. Región de 8 vecindades.

De acuerdo con las siguientes ilustraciones, primero se debe calcular las interpolaciones en dirección del eje *X*, es decir, se debe aplicar el *kernel* para obtener el punto A de los vértices 1 y 4; el punto B de los vértices 2 y 3; el punto C de los vértices 5 y 8; y el punto D de los vértices 6 y 7. Como segundo paso, se debe aplicar el *kernel* de interpolación entre los puntos A y B, y los puntos C y D para obtener los puntos E y F, respectivamente. En este caso, el *kernel* es aplicado en dirección del eje *Y*. El último paso consiste en aplicar el *kernel* entre los puntos E y F, en dirección del eje *Z*, a fin de calcular el valor del punto X. Las figura 2.38, 2.39 y 2.40 ilustran este concepto.

Figura 2.38. Paso Uno.

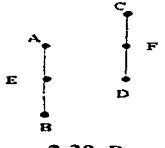


Figura 2.39. Paso Dos.

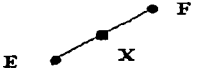


Figura 2.40. Paso Tres.

2.3.3.1. La Vecindad Más Próxima.

El *kernel* de la vecindad más próxima es el más sencillo de todos y más fácil de implementar y, por tanto, el costo de procesamiento es muy bajo, no obstante, no sirve para hacer transformaciones de escalamiento debido a que llega a producir efectos de *aliasing*. Básicamente consiste en elegir el vecino *(x, y, z)* más cercano al punto de interés, dentro de la región. El *kernel* de interpolación para una señal está definido por la función de la figura 2.41. En ella, el centro del *kernel* coincide con la posición *x* en cuestión, adquiriendo el valor de 1, mientras sus vecindades adquieren el valor de 0.

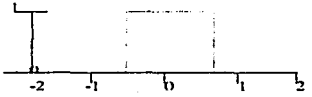


Figura 2.41. Kernel Unidimensional para la Vecindad más Próxima.

2.3.3.2. Interpolación Lineal.

La interpolación lineal está definida dentro de la teoría del procesamiento de señales, aunque se puede extrapolar a un espacio de tres dimensiones. La aplicación de este *kernel* produce una mejor calidad de muestreo que la anterior máscara, no obstante, no elimina el problema del escalamiento, ya que llega a producir puntos negros o blancos en forma de cruces que pueden afectar el resultado final. En un espacio unidimensional, se define en la ecuación 2.50.

$$f(d) = [d - f(x_0)] * [(f(x_1) - f(x_0)) \quad (x_1 - x_0)] \qquad \textbf{(Ec. 2.50)}$$

Gráficamente, el *kernel* está definido en la figura 2.42. Ahí se muestra que el centro del kernel coincide con la posición *x* en cuestión, adquiriendo el valor de 1. Este valor disminuye en las vecindades de esta posición. Obviamente, la figura 2.42 muestra el *kernel* de interpolación para una dimensión.

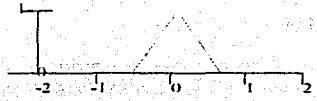


Figura 2.42. Kernel Unidimensional para la Interpolación.

2.3.3.3. Convolución Cúbica.

Con un nivel alto de consumo de procesamiento, la interpolación cúbica produce una mejor calidad y resolución en las imágenes. La implementación de este *kernel* conlleva a utilizar optimizaciones en el cálculo de los coeficientes, además de que esos coeficientes pueden ser negativos, por lo que es necesario aplicar reglas de re-escalamiento de los valores, o simplemente, hacer cero todo los coeficientes negativos. En un espacio unidimensional se define como:

$$f(x) = \begin{cases} a+2|x|^3 - a+3|x|^2 + 1 & si\ 0 \le |x| < 1 \\ a|x|^3 - 5a|x|^2 + 8a|x| - 4a & si\ 1 \le |x| < 2 \\ 0 & si\ 2 \le |x| \end{cases}$$

Donde a es un punto de control y está definido en el intervalo $-3 \le a \le 0$. La literatura recomienda como valor ideal de $a = -0.5$, ya que un valor de -3 produce efectos de ruido, y un valor de 0 puede llegar a producir efectos de alisamiento. El *kernel* está definido en la figura 2.43. En esa figura, la convolución cúbica se aplica a una espacio unidimensional. Los valores del *kernel* oscilan entre -1 y 1; tomando el máximo valor positivo, el centro del *kernel*, mientras que las vecindades adquieren valores menores a 1.

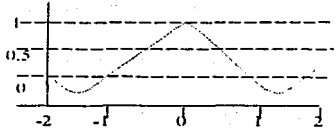


Figura 2.43. Kernel Unidimensional para la Convolución Cúbica.

2.3.3.4. B-spline.

A diferencia del anterior, este *kernel* no produce valores negativos, por ello, es utilizado como filtro paso-bajas; pero sigue siendo muy costoso en cuanto al procesamiento requerido. Una de las posibles optimizaciones es eliminar las operaciones multiplicativas para calcular el cuarto coeficiente, esto es, calcular los primeros 3 coeficientes, y el cuarto haciendo una substracción de los 3 coeficientes a la unidad. En la teoría del procesamiento de señales se define como:

$$f(x) = \begin{cases} 1.2|x|^3 - |x|^2 - 2.3 & si\ 0 \le |x| < 1 \\ -1.6|x|^3 - |x|^2 - 2|x| - 4.3 & si\ 1 \le |x| < 2 \\ 0 & si\ 2 \le |x| \end{cases}$$

Gráficamente el *kernel* de *B-spline*, para una señal, está definido en la figura 2.44. En esta ocasión, los valores del *kernel* oscilan entre 0 y 0.8, y ningún elemento del *kernel* adquiere el máximo valor.

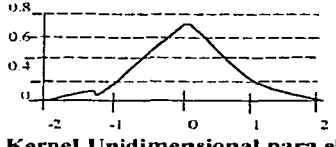


Figura 2.44. Kernel Unidimensional para el B-spline.

La decisión de que *kernel* de interpolación utilizar se desea aplicar depende de los requerimientos iniciales del VR que se desea obtener, obviamente, si se desea una interacción directa pues lo más recomendable es usar el *kernel* de la vecindad más próxima o el de interpolación lineal, en cambio, si desea una mayor calidad de *render* y menos interactividad, sin duda, es recomendable el uso de los *B-Splines*.

2.3.4. Modelos de Iluminación.

El modelo de iluminación se encarga de describir la manera en que un color es asignado en un punto del espacio *n* dimensional, basado en la fuente de luz.

el material, la orientación, la posición de ese punto en el espacio, y el ángulo entre el observador y la fuente de luz. Existen dos tipos de iluminación: local y global. La iluminación local sirve para definir la forma de los objetos contenidos en la escena y casi no tiene interacción con ellos. En el caso de la iluminación global, los objetos tienen una mayor interacción con la fuente de luz ya que pueden absorber, emitir y reflejar cierta cantidad de luz. Además, una fuente de iluminación global otorga una apariencia de tridimensionalidad a la escena. Existen varios modelos de iluminación global, pero los más importantes son el trazado de rayos (*ray tracing*) y la radiosidad.

2.3.4.1. Trazado de Rayos.

El trazo de rayos (*Ray Tracing*) es una técnica que permite calcular la iluminación de una escena en base en un modelo de iluminación global. La característica principal de esta técnica es que se aproxima bastante a las leyes físicas que definen el comportamiento entre los objetos y la luz. El objetivo de esta técnica es determinar el nivel de intensidad para cada *pixel (x, y)* contenido en la imagen. Para ello se traza un rayo en dirección inversa del punto del observador hacia el *pixel* en cuestión. Si el rayo intercepta un objeto, los cálculos locales determinarán el color de ese *pixel*, basándose en la iluminación directa que recibe. La trayectoria del rayo finaliza cuando ya no hay más objetos que interceptar o cuando las contribuciones son muy cercanas a cero.

La implementación más simple de este algoritmo consiste en detener el rayo cuando ocurre la primera intersección con algún objeto, de esta manera, se eliminan las superficies ocultas y agiliza el proceso de *render*, sin embargo, esta implementación puede producir efectos de *aliasing* en la imagen final.

El cálculo de la intersección de los rayos con los objetos puede llegar a ser muy complejo, debido a la geometría de los mismos, por ello, el *ray tracing* utiliza un *bounding volume* para encerrar al objeto y hacer que los cálculos de intersección sean más fáciles. Dependiendo de la geometría del objeto, se define su *boundig volume* correspondiente, que puede ser una forma geométrica básica tal como esferas, cubos, cilindros, etc. Es importante hacer una correcta elección del *boundig volume* a utilizar, porque esto evita realizar cálculos innecesarios, por ejemplo, si se tiene un objeto largo y estrecho en la escena, y se elige una esfera como *boundig volume* es evidente que un rayo

puede interceptar a la esfera, pero ello no significa que se ha interceptado al objeto, tal como lo muestra la figura 2.45.

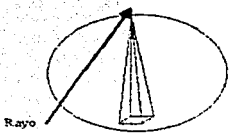


Figura 2.45. Bounding Volume.

2.3.4.1.1. Esferas.

La intersección de un rayo con un *boundig volume* esférico es fácilmente calculable. Si la trayectoria del rayo está definida entre los puntos $R_1(x_1, y_1, z_1)$ y $R_2(x_2, y_2, z_2)$, y la ecuación de la esfera, cuyo centro es (l, m, n) y tiene un radio r, se define como:

$$(x-l)^2 + (y-m)^2 - (z-n)^2 - r^2 = 0 \qquad \textbf{(Ec. 2.51)}$$

El primer paso es parametrizar la trayectoria del rayo con las siguientes ecuaciones:

$$x = x_1 + (x_2 - x_1)\,t = x_1 + it \qquad \textbf{(Ec. 2.52)}$$
$$y = y_1 + (y_2 - y_1)\,t = y_1 + it \qquad \textbf{(Ec. 2.53)}$$
$$z = z_1 + (z_2 - z_1)\,t = z_1 + it \qquad \textbf{(Ec. 2.54)}$$

Sustituyendo los valores de x, y, z parametrizados en la ecuación 2.51, desarrollando los binomios y agrupando en términos de la variable t, tenemos que:

$$(i^2+j^2+k^2)t^2 + [2i(x_1-l)+2j(y_1-m)+2i(x_1-l)]t + [l^2+m^2+n^2-x_1^2-y_1^2+z_1^2+2(-lx_1-my_1-nz_1)-r^2]$$

Si el determinante de esta ecuación es menor a cero, entonces el rayo no intercepta a la esfera; en cambio, si el determinante es igual a cero entonces el rayo es tangencial a la esfera. Por último, si el determinante es mayor a cero, entonces el rayo intercepta la esfera, en este caso, se sustituye los valores t por

los originales valores paramétricos con el fin de calcular las coordenada exactas donde ocurre la intersección. En algunos caso es importante conocer la normal[5] en el punto de intersección (x_i, y_i, z_i), para la cual se utiliza la fórmula:

$$N = [x_i - l / r,\ y_i - m / r,\ z_i - n / r] \qquad \textbf{(Ec. 2.55)}$$

2.3.4.1.2. Poliedros Convexos.

Cuando un objeto es representado por una serie de polígonos convexos unidos entre sí, es posible determinar si el rayo a traviesa a cada uno de los polígonos, por ejemplo, si un plano contiene a un polígono y está definido como $ax + by + cz + d = 0$ y si la trayectoria del rayo fue parametrizada, como en el anterior caso, la ecuación de la intersección está dada por la ecuación 2.56.

$$t = -(ax_1 + by_1 + cz_1 + d) \quad (ai + bj + ck) \qquad \textbf{(Ec. 2.56)}$$

Los parámetros de decisión son los siguientes: si el numerador es positivo, entonces el rayo no intercepta al polígono; si el denominador es cero, entonces el rayo viaja en forma paralela al polígono.

2.3.4.1.3. Cubos.

Es una generalización del caso anterior, y consiste en el cálculo y comparación de distancias entre dos planos. Por ejemplo, se calcula la distancia a lo largo del rayo con respecto a la primer polígono, y después con el segundo polígono. Si el valor de distancia del primer polígono es mayor al valor de distancia del segundo polígono entonces el rayo no intercepta al cubo.

2.3.4.1.4. Superficies Cuadráticas.

La ecuación general de una superficie cuadrática, incluyendo a la esfera, se define como:

$$Ax^2 + Ey^2 + Hz^2 + 2Bxy + 2Fyz + 2Cxz + 2dx + 2Gy + 2Iz + J = 0$$
(Ec. 2.57)

[5] Un vector normal es un vector que apunta en dirección perpendicular a la posición de la superficie u objeto en cuestión.

Aplicando las mismas operaciones de normalización que en la ecuación 2.51 y agrupando la ecuación 2.57 en términos de t, tenemos que la ecuación de intersección está dada por:

$$at^2 + bt + c = 0 \qquad \textbf{(Ec. 2.58)}$$

Donde:

$$a = Ai + Ej + Hk + 2Bij + 2Cik + 2Fjk$$

$$b = 2[Ax_1 i + B(x_1 j + y_1 i) + C(x_1 k + z_1 i) + Di + Ey_1 j + F(y_1 k + z_1 j) + Gj + Hz_1 k + Ik]$$

$$c = Ax_1 + Ey_1 + Hz_1 + 2Bx_1 y + 2Cx_1 z_1 + 2Dx_1 + 2Fy_1 z_1 + 2Gy_1 + 2Iz_1 + J$$

2.3.4.2. Radiosidad.

La técnica de la radiosidad se aplica a escenas con ambientes cerrados. En este caso, la radiosidad basa sus cálculos a través de las interacciones entre la luz con la forma, textura y color de los objetos contenidos en la escena, es decir, incorpora técnicas para el cálculo de la transferencia de calor radiado que se emplea en termodinámica. Estas características hacen a esta técnica ideal para recrear ambientes muy realistas, sin embargo, el nivel de procesamiento requerido es muy alto.

La idea fundamental es buscar el equilibrio de la energía que es emitida por la fuente de luz y la energía que es absorbida por los objetos dentro de la escena. Bajo este concepto un objeto puede absorber cierta cantidad de energía proveniente de otro objeto qué es iluminado, así, todos los objetos emiten su propia energía y, al mismo tiempo, absorben la energía proveniente de los demás objetos. La fórmula de la radiosidad es:

$$B_i = E_i + \rho_i \sum_{j=1}^{n} B_j F_{ij} \qquad \textbf{(Ec. 2.59)}$$

La forma más común de resolver esta ecuación es convirtiéndola en un sistema de ecuaciones con n número de incógnitas, y si tomamos en cuenta que una escena puede contener miles de objetos, entonces el cálculo de ese sistema de ecuaciones se vuelve muy complicado. Las ventajas que la radiosidad presenta sobre el *ray tracing* son tres

1. Es independiente del observador.
2. Los objetos contenidos en la escena son tratados como elementos finitos.
3. Realiza un tratamiento correcto de las superficies difusas.

2.3.5. Sombreado y Clasificación.

El sombreado (*Shading*) es un proceso que determina los parámetros que debe aplicarse al momento de utilizar el modelo de iluminación. A decir, el proceso de sombreado se encarga de encontrar la mezcla (*blending*) de colores que pertenecen a cierto punto *(x, y, z)*, bajo el modelo de RGBα. Primero, es necesario hacer un mapeo de las intensidades de los datos volumétricos sobre el modelo RGB usando funciones de transferencia y calcular la opacidad α (alfa) de acuerdo a la intensidad. Al primer paso se le conoce como procesamiento de color (*color processing*), y al segundo como clasificación.

2.3.5.1. Procesamiento del Color.

El procesamiento de color consiste en aplicar funciones de transferencia por cada una de las bandas del modelo RGB. El modelo más utilizado para obtener el color de cada banda es el modelo de iluminación *Phong*, que describe los efectos de la fuente de luz sobre cada punto *f(x, y, z)*, de acuerdo a la luz ambiental, la reflexión especular y difusa sobre ese mismo punto.

Conforme a este modelo, el color final es la suma de las contribuciones de la luz ambiental, la reflexión difusa y la reflexión especular. Siendo que la luz ambiental es una fuente de luz con la misma intensidad en cualquier punto de la escena, así pues, la contribución de la luz ambiental sobre un punto está dada por la ecuación 2.60:

$$I_a = C_a K_a O_d \qquad \textbf{(Ec. 2.60)}$$

Donde C_a es el color de la luz ambiental; K_a es el coeficiente de reflexión de ese punto y su restricción es $0 \leq K_a \leq 1$, siendo 0 una aproximación al color negro; y, O_d es el color difuso en ese punto. La contribución de la reflexión

difusa depende de la distancia entre el punto *f(x, y, z)* y la fuente de luz, así como de la orientación que tenga el punto *f(x, y, z)*, así, la contribución de la reflexión difusa es:

$$D = C_p K_d O_d \cos\theta = C_p K_d O_d (N \cdot L) \qquad \textbf{(Ec. 2.61)}$$

Donde C_p es el color de la fuente; K_d es el coeficiente de reflexión difusa; O_d es el color difuso en ese punto; θ es el ángulo entre la normal de la superficie sobre el punto *f(x, y, z)* y la dirección de la fuente de luz, *N* es el vector normal sobre el punto *f(x, y, z)*; y *L* es el vector que indica la dirección del punto *f(x, y, z)* hacia la fuente de luz. Por último, tenemos la contribución de la reflexión especular, que es la cantidad de brillantez en el punto *f(x, y, z)*, definida por la ecuación:

$$S = C_p K_s O_s (R \cdot V)^n \qquad \textbf{(Ec. 2.62)}$$

Donde C_p es el color de la fuente; K_s es el coeficiente de reflexión especular; O_s es el color especular en el punto *f(x, y, z)*; *R* es el vector de normal de la reflexión difusa, es decir, es el vector *L* rotado θ grados sobre el vector *N*; *V* es el vector de dirección del punto *f(x, y, z)* hacia el punto de vista; y, *n* es el exponente de la reflexión especular. Todas las ecuaciones 2.60, 2.61 y 2.62 crean un nuevo color, C_0, del punto *f(x, y, z)*, dada por la expresión final:

$$C_0 = L + D + S = C_a K_a O_d + C_p [K_d O_d (N \cdot L) + K_s O_s (R \cdot V)^n] \qquad \textbf{(Ec. 2.63)}$$

Si consideramos a $A(A_x, A_y, A_z)$ como la posición de la fuente de luz dentro de la escena, y la posición del observador como $O(O_x, O_y, O_z)$, entonces las ecuaciones para calcular los vectores *L*, *R*, y *V*, de la ecuación 2.63, son:

$$L = [A_x - x / d,\ A_y - y / d,\ A_z - z / d]; \quad d = \sqrt{(A_x - x)^2 + (A_y - y)^2 + (A_z - z)^2}$$
$$V = [O_x - x / d,\ O_y - y / d,\ O_z - z / d]; \quad d = \sqrt{(O_x - x)^2 + (O_y - y)^2 + (O_z - z)^2}$$
$$R = 2(N \cdot L)N - L$$

Este proceso es bastante costoso en cuanto al procesamiento que requiere, debido a que las ecuaciones deben aplicarse por cada una de las bandas del modelo RGB, por cada punto *f(x, y, z)* y por cada fuente de luz que este presente en la escena. No obstante, existe otro modelo de sombreado capaz de realizar el mismo proceso más rápidamente pero con una menor calidad, este proceso se llama modelo de iluminación *Gourand*, que consiste en interpolar los colores pertenecientes a dos vértices, es decir, calcula los vectores normales

de cada vértice y les asigna el color correspondiente, luego, hace una interpolación del color en el vértice A hacia el color del vértice B.

2.3.5.2. Clasificación.

El objetivo de clasificar los *voxeles* consiste en la búsqueda de objetos o superficies dentro de la escena, de esta manera, se determina si es necesario una visualización o no. El proceso de clasificación se lleva a cabo con la asignación de opacidad a cada uno de los *voxeles* que conforman los datos volumétricos. El uso de funciones de transferencia ayuda a detectar objetos dentro de un volumen utilizando el histograma o el gradiente.

El histograma es bastante útil para observar la distribución de *voxeles* dentro de los datos volumétricos y determinar el tipo de función de transferencia que se desea aplicar. En cuanto al gradiente, su gran utilidad es la detección de orillas, tal como se vio en la sección anterior, puede asignarse distintos valores de opacidad de acuerdo a la información que proporcione el gradiente. A continuación se presenta un ejemplo donde se muestra el uso de una función de transferencia para calcular la opacidad de un *voxel* determinado.

$$\alpha_i(r, f_v) = \begin{cases} 1 - \left| f_v - I_i \right| & r \left| \nabla_i \right| \quad si \quad \left| \nabla_i \right| > 0 \ y \ I_i - r \left| \nabla_i \right| \le f_v \le I_i + r \left| \nabla_i \right| \\ 1 & si \quad \left| \nabla_i \right| = 0 \ \acute{o} \ f_v = I_i \\ 0 & en \ cualquier \ otro \ caso \end{cases}$$

Donde α_i es la opacidad de i-ésimo *voxel*; r es el máximo nivel de intensidad presente en el volumen; f_v es un umbral de intensidad elegido; I_i es la intensidad de i-ésimo *voxel*; y, ∇_i es el gradiente aplicado en el i-ésimo *voxel*. En la sección del PDI se discutió acerca de las propiedades del operador gradiente y de los *kernels* más utilizados para la detección de orillas. En el caso de datos volumétricos, el *kernel* está representado en la figura 2.46:

x = -1

$$\begin{vmatrix} -2 & 0 & 2 \\ -3 & 0 & 3 \\ -2 & 0 & 2 \end{vmatrix}$$

x = 0

$$\begin{vmatrix} -3 & 0 & 3 \\ -6 & 0 & 6 \\ -3 & 0 & 3 \end{vmatrix}$$

x = 1

$$\begin{vmatrix} -2 & 0 & 2 \\ -3 & 0 & 3 \\ -3 & 0 & 3 \end{vmatrix}$$

Figura 2.46. Kernel de Sobel para 3D.

Como habrá notado, estos *kernels* obedecen a las reglas del operador *Sobel*, pero, en esta ocasión, se aplica a un espacio tridimensional. Tal como se muestra, el *kernel* puede detectar orillas en dirección del eje *X*; si desea detectar las orillas en el eje *Y* se necesita rotar la máscara y alinearla sobre el eje en cuestión. El mismo procedimiento se aplica para detectar orillas en dirección del eje *Z*.

2.3.6. Composición.

La composición es el método de aproximar la intensidad de luz a medida que atraviesa una serie de objetos semitransparentes, partiendo del supuesto de que cada objeto puede emitir luz y que la intensidad es atenuada en forma proporcional a la intensidad de energía que recibe. En pocas palabras, la composición es el proceso de proyectar todos los valores acumulados en la trayectoria del rayo sobre un dispositivo de despliegue. Existen tres técnicas de *render* que se pueden aplicar para visualizar los datos volumétricos sobre un dispositivo de despliegue, estos son: *render* interactivo, *render* de superficies y *render* directo.

2.3.6.1. Render Interactivo.

Este tipo de *render* no se especializa en la compresión de datos y no necesita de interpolaciones complejas, principalmente trabaja en mallas de tipo cubertillas para realizar representaciones sencillas sin utilizar algún tipo de superficies. Los algoritmos principales de esta técnica son:

1. Contornos. Construye el volumen como una lista de isovalores conectados a través de líneas, por lo que da la apariencia de una malla de alambre. La principal ventaja de este modelo es su rápida respuesta a los procesos interactivos con el usuario, sin embargo, la información que pueda proporcionar es nula.
2. *Tiny Cubes*. Representa el volumen a través de pequeños cubos sin valor de opacidad.
3. *Vanishing Cubes*. Es el mismo caso que el anterior, pero esta vez toma encuentra la opacidad de cada cubo en el volumen.

4. *Slicing*. Esta técnica consiste en cortar el volumen en secciones de igual tamaño sobre algún eje del espacio tridimensional. La ventaja de esta técnica es que permite explorar y clasificar las diferentes secciones que componen al volumen, pero al igual que la primera técnica no proporciona demasiada información.
5. Reproyección adaptativa. Utiliza el promedio de todos los valores acumulados a lo largo de la trayectoria del rayo para calcular la intensidad del *pixel* correspondiente, ello le permite una gran interactividad con el usuario, sin embargo, la proyección obtenida está definida en la escala de grises y no permite modelos de sombreado.
6. *Maximum Voxel*. Esta técnica utiliza el valor con mayor peso del conjunto de valores acumulados en la trayectoria del rayo para determinar el valor del *pixel* correspondiente. Utiliza el *kernel* de interpolación del vecino más próximo (*nearest neighbor*). Posee las mismas ventajas y desventajas que la técnica anterior.

2.3.6.2. Render de Superficies.

Este tipo de técnica utiliza interpolaciones de mediana calidad, sobre todo aquellas que se basan en umbrales con el objeto de detectar superficies. Fundamentalmente consiste de tres pasos: el primero detecta superficies utilizando umbrales; el segundo consiste en la elección del modelo de iluminación; y, el tercero consiste en la aplicación del *render*. Los algoritmos que se utilizan para realizar esta técnica se describen a continuación.

2.3.6.2.1. Cubos Opacos.

Esta técnica atribuye valores opacos a aquellas celdas cuyo valor esté por debajo del umbral y después, poligoniza sólo aquellas celdas que tienen un valor igual o mayor al del umbral y aplica el proceso de *render* a los polígonos obtenidos uno por uno hasta obtener la superficie deseada. Las ventajas de este algoritmo son su facilidad de implementación y la habilidad para la clasificación. Su desventaja principal es que llega a producir superficies dentadas o con efectos de aliasing. La figura 2.47 ilustra este concepto.

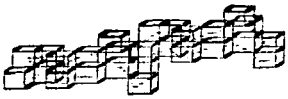


Figura 2.47. Cubos Opacos.

2.3.6.2.2. Dividing Cubes.

Cuando una celda es interceptada por una isosuperficie, de acuerdo a un umbral específico, se calcula el valor de *pixel* correspondiente sólo si esa celda contribuye a ese *pixel*. En caso contrario, la celda se subdivide en regiones muy pequeñas. Este proceso continúa hasta que las celdas se reducen a puntos, y con ello obtener una superficie definida por puntos en vez de polígonos.

2.3.6.2.3. Tracking de Contornos.

Conecta isocontornos en cortes transversales produciendo un conjunto de triangulaciones, tal y como se muestra en las figuras 2.48 y 2.49.

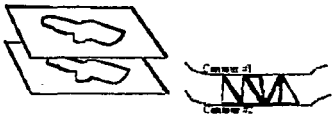


Figura 2.48. Contornos.

Figura 2.49. Contornos.

2.3.6.2.4. Tracking de Isosuperficies.

Básicamente, busca una superficie umbral clasificando las celdas según los valores de sus vértices respecto al valor del umbral. Una celda es parte de la superficie, si al menos, uno de sus vértices está por encima del valor del umbral y por lo menos otro de sus vértices está por abajo del umbral especificado.

Existen sólo 15 configuraciones posibles para determinar si una celda pertenece a la superficie tal como se muestra en la figura 2.50.

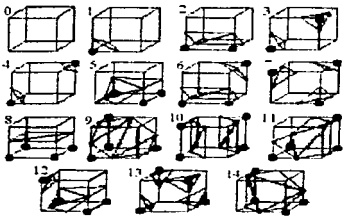


Figura 2.50. Marching Cubes.

Este algoritmo es eficiente, sin embargo, puede producir agujeros en la superficie umbral debido a un fenómeno llamado "caras ambiguas". Cuando una celda tiene una cara tal que dos de sus vértices tiene valores por encima del umbral y están separados diagonalmente por otros dos vértices que no poseen valores por arriba del umbral, el algoritmo no puede determinar si la superficie umbral pasa por dentro o por fuera de la celda.

De acuerdo con la opinión de Andrea Silvetti[6], del Departamento de Ciencias de la Computación de la Universidad del Sur, existen dos algoritmos para eliminar el problema de caras ambiguas. El primero es propuesto por los investigadores *Nelson* y *Hamman*, que consiste en hacer una distinción entre las caras ambiguas que separan los vértices con valor arriba del umbral, y en una cara ambigua que separa dichos vértices. De esta manera, se toma la intensidad de cada uno de los vértices de una cara para obtener su promedio y si el valor calculado está por arriba del umbral, entonces es una cara unida, por el contrario, se considera como una cara separada. La otra técnica es propuesta por *Dennis Bouvier*, quien aconseja encontrar una aproximación de la superficie considerando los valores de los vértices con el fin de evitar ambigüedades.

[6] Publicó un artículo denominado "Mejoras al algoritmo de Marching Cubes" donde expuso esta situación.

2.3.6.3. Render Directo.

El proceso de *render* directo es la técnica más utilizada por la mayoría de las implementaciones de paquetes de Visualización Científica, e incluye todos los pasos descritos anteriormente, por lo que el nivel de procesamiento es muy alto, sin embargo, los resultados finales superan en mucho a la calidad que otorgan las anteriores técnicas. Los algoritmos que existen se dividen en dos clases, los de mapeo hacia delante (*forward mapping*) y los de mapeo inverso (*backward mapping*). En el primer caso los datos son mapeados sobre la imagen, mientras que en el segundo caso, la imagen es mapeada sobre los datos.

2.3.6.3.1. Lanzamiento de Rayos.

El lanzamiento de rayos (*Ray Casting*) es una de las técnicas más utilizadas para la generación de imágenes de alta calidad. Como su nombre lo indica, consiste en lanzar un rayo por cada *píxel* en el área de despliegue, que atraviesa el volumen. Los colores y opacidades son sumadas a lo largo de la trayectoria del rayo, y el resultado se utiliza para determinar el nivel de intensidad que le corresponde al *pixel* que originó el rayo. Matemáticamente se define como:

$$I(i, j) = \int_i^j g(s)\, e^{-a}\, ds \qquad \textbf{(Ec. 2.64)}$$
$$a = \int_i^s T(x)\, dx$$

Donde $I(i, j)$ es la intensidad del *pixel* (i, j) dentro del área de despliegue; $g(s)$ es el modelo de iluminación utilizado en el volumen; $T(x)$ es el coeficiente de extinción, que indica el rango de oclusión de la luz por unidad de longitud y ds es la dirección del rayo. Existen dos modalidades para implementar la ecuación del *ray casting* en un espacio discretizado. La primera corresponde al tipo *front-to-back*, definida por la ecuación 2.65.

$$I(i, j) = \sum_{n=0}^{N} I_n \prod_{m=0}^{n-1} T_j = I_0 + I_1 T_0 + I_2 T_0 T_1 + \ldots + I_N T_0 \ldots T_{N-1} \qquad \textbf{(Ec. 2.65)}$$

La intensidad final del *pixel* (i, j) es la sumatoria de las intensidades I_n del *n-ésimo voxel* con una transparencia acumulada T_n a lo largo de la trayectoria del rayo sobre n números de *voxeles*, así, la transparencia está definida como $1 - \alpha$. En este caso, se utiliza la transparencia en lugar de la opacidad pues el costo de

procesamiento es ligeramente menor. Habitualmente, el RC en su modalidad *front-to-back* es muy poco utilizado dado su alto requerimiento de procesamiento, es más utilizada la versión *back-to-front*, o mapeo inverso (*backward mapping*), que es más rápido y más fácil de implementar. La ecuación que la define es:

$$I(i,j) = \sum_{n=0}^{N} I_n \prod_{m=n-1}^{N} (1-\alpha_j) \qquad \textbf{(Ec. 2.66)}$$

Las ventajas de utilizar el lanzamiento de rayos como la técnica de *render* es que se obtienen resultados de alta calidad, además de que, se puede paralelizar el algoritmo para reducir el tiempo de procesamiento requerido.

2.3.6.3.2. Composición de Planos.

Es una técnica basada en el mapeo directo (*forward mapping*). El proceso es relativamente sencillo: la contribución de cada *voxel*, a lo largo de la trayectoria del rayo, estará determinada por la sumatoria del color y opacidad de sus vecinos. El tamaño del *kernel* es determinado por las necesidades de la imagen final, normalmente se utiliza la máscara *Guassiana*. La proyección final de todos los *voxeles* acumulados en el rayo se le denomina huella del pie (*footprint*). Cada contribución hecha por el *voxel* y sus vecindades es almacenada por *frame buffer*, donde se realizan las operaciones de la función de composición.

Las ventajas que presenta sobre el *ray casting* son varias, principalmente, porque es más fácil de implementar y el tiempo requerido de procesamiento es mucho menor, además de que es más exacto al momento de hacer las interpolaciones y reconstrucciones requeridas, además, el algoritmo puede ser paralelizado para agilizar la rapidez de los cálculos. Sin embargo, la desventaja más considerable es su alto requerimiento de memoria. La figura 2.51 ilustra este concepto.

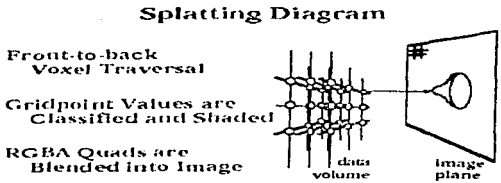


Figura 2.51. Composición de Planos.

2.3.6.3.3. Proyección de Máxima Intensidad.

En ocasiones, los datos volumétricos poseen característi[cas] de ruido o son incoherentes entre sí, sobretodo aquellos que proceden de to[mo]grafías hechas a través de resonancia magnética. Bajo este concepto, la téc[nica] del RC y las técnicas de superficie umbral resultan ineficientes para lograr un buen VR, requerido por las aplicaciones médicas. El problema principal al que se enfrentan las aplicaciones de imágenes médicas (*medical imaging*) es lograr que el VR muestre sólo los valores más altos contenidos en el conjunto de datos, es así como surge la técnica de la proyección basada en la máxima intensidad, *MIP*, la cual consiste en lanzar un rayo, utilizando el mapeo directo, a través del volumen y determinar el máximo valor de todos los *voxeles* acumulados a lo largo de la trayectoria del rayo.

Existen 3 métodos para calcular el máximo valor. El primero, se realiza a través del cálculo de las funciones de transferencia de color para cada *voxel* en el rayo, y por cada una de las bandas del modelo RGB; este método es extremadamente costoso en cuanto al nivel de procesamiento que requiere. El segundo, método utilizado es el uso de *kernels* de interpolación lineal. El tercer método, y el más utilizado, corresponde a la interpolación basada en el vecino más próximo, donde el máximo valor de intensidad está dado por el *voxel* que tenga dicho valor.

Aunque los resultados son bastantes buenos, las imágenes están definidas en nivel de grises, aunado a que no poseen un modelo de sombreado, ya que la técnica no lo permite, debido a ello, las imágenes proporcionan información muy pobre, por lo que, los paquetes que utilizan el *MIP* como técnica de *render*

deben proporcionar herramientas de transformación de coordenadas para aumentar la comprensibilidad sobre los datos desplegados. Sin embargo, han surgido algunas variantes de esta técnica para tratar de resolver este problema. Las imágenes finales tienen una apariencia de profundidad, propiedad que puede ser utilizada otorgándole un modelado de sombreado básico y con ello, obtener un mejor VR. A esta variante se le conoce como *DMIP*. Cuando los valores máximos de los datos volumétricos están muy concentrados en determinada región, es imposible utilizar el *DMIP* para calcular sus niveles de sombreado.

Una segunda variante es el *LMIP* que consiste en lanzar un rayo a través del volumen y detener su trayectoria cuando encuentre un valor por encima de un valor umbral dado por el usuario. En caso de que el algoritmo no encuentre ningún valor, entonces utilizará alguno de los métodos mencionados con anterioridad para calcular el valor de intensidad máximo. En esta técnica es de vital importancia elegir correctamente el valor del umbral, ya que el proceso depende enteramente de esta base, a decir, si el valor de umbral es muy grande, los resultados obtenidos serán los mismos que si se aplicara la técnica del *MIP*, en cambio, si el valor del umbral es pequeño, los resultados serán de mejor calidad, e inclusive, se mostrarán algunos efectos de sombreado.

2.3.6.3.4. Volume Rendering a través de Fourier.

Como se ha mencionado a lo largo del capítulo, los datos volumétricos suelen ser de arreglos muy grandes, y aún aplicando las técnicas de *render* más eficiente y rápidas pueden requerir de niveles altos de procesamiento, sin contar con el hecho de que los resultados no tendrán la mejor calidad pretendida. Algunos investigadores han propuesto una técnica que sólo permita aplicar VR a ciertas regiones de los datos volumétricos, en lugar de procesar todo el conjunto. Esta técnica es a través del análisis de *Fourier*, descrito anteriormente.

El análisis de *Fourier* propuesto en el PDI puede aplicarse a un espacio tridimensional. Partiendo del supuesto de que se desea hacer una proyección de rayos X con el conjunto de datos, la función de densidad que define a tal conjunto es:

$$d = \int F(x(t), y(t), z(t))\, dt$$

Donde f es la función que define al volumen X, Y y Z es la representación paramétrica de la trayectoria del rayo. Aplicando el teorema del corte de *Fourier* (*Slicing Fourier Teorem*), introducido por *T. Malzbender*[7], podemos obtener proyecciones cortando el volumen en pedazos de tamaño uniforme aplicando el análisis de *Fourier*. Sin embargo, existe un problema, el análisis de *Fourier* involucra el uso de números complejos, lo que hace difícil su implementación y compresión del algoritmo, así pues, el mismo autor de este teorema sugiere utilizar un método alternativo proporcionado por la transformada de *Hartley*, cuyas propiedades son muy similares a las de *Fourier*, con la ventaja de que esta transformada no maneja números complejos.

2.3.7. *Información Adicional.*

2.3.7.1. *Perspectiva y Proyección Ortogonal.*

En el espacio tridimensional existen dos herramientas de transformación muy útiles al momento de hacer VR en algún dispositivo de despliegue: la perspectiva y la proyección paralela u ortográfica. Ambas poseen su propia matriz de transformación.

La perspectiva otorga la particularidad de que los objetos de la escena de menor tamaño son los que están más lejos, mientras que los objetos de mayor tamaño son los que están más cerca. En este modelo, el observador se encuentra el centro de proyección, que es el punto donde convergen las líneas de proyección. Asimismo, sólo algunas cara de los objetos contenidos en este modelo son visibles al observador, conforme el observador se mueve, las caras de los objetos se hacen, paulatinamente, visibles. El modelo de la perspectiva está definido en la figura 2.52.

[7] Consulte la siguiente dirección de Internet para más información: http://wwweecg.toronto.edu/~fender/fvr/references.html

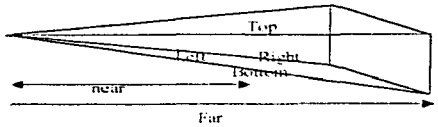


Figura 2.52. Modelo de Perpectiva.

Cuando se utiliza alguna técnica de VR, como por ejemplo el RC, es necesario tomar en cuenta el *frustrum* de la escena- La matriz de la perspectiva está definida en la matriz 2.1. Cuando los objetos, contenidos en una escena 3D, ya han sido modificados por alguna de las transformaciones afines, es necesario, transformar las coordenadas 3D a 2D y proyectarlas en un dispositivo de despliegue, es por ello, que es útil la matriz 2.1.

$$\begin{vmatrix} \frac{2\,Near}{Right - Left} & 0 & \frac{Right + Left}{Right - Left} & 0 \\ 0 & \frac{2\,Near}{Top - Bottom} & \frac{Top + Bottom}{Top - Bottom} & 0 \\ 0 & 0 & \frac{-Far + Near}{Far - Near} & \frac{2\,Far\,Near}{Far - Near} \\ 0 & 0 & -1 & 0 \end{vmatrix}$$

Matriz 2.1

En cuanto a la proyección paralela, se caracteriza porque las líneas de proyección, que atraviesan la escena, corren de manera paralela; esta característica supone que el observador está, infinitamente, alejado de la escena. La literatura concuerda de que este tipo de proyección es muy deficiente, debido a que es improbable de que el observador esté lo bastante lejos para reproducir los efectos de este modelo. El modelo es definido en la figura 2.53.

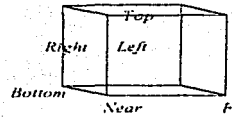


Figura 2.53. Modelo de Proyección Ortográfica.

La matriz de proyección ortográfica está definida por la matriz 2.2. la funcionalidad de esta matriz es básicamente la misma que la 2.1, pues traduce las coordenadas 3D de los objetos a coordenadas 2D, y sea posible el despliegue en un dispositivo bidimensional.

$$\begin{bmatrix} \frac{2}{Right - Left} & 0 & 0 & \frac{Right + Left}{Right - Left} \\ 0 & \frac{2}{Top - Bottom} & 0 & \frac{Top + Bottom}{Top - Bottom} \\ 0 & 0 & \frac{2}{Far - Near} & \frac{-Far + Near}{Far - Near} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Matriz 2.2

2.3.8. Aplicaciones del Volume Rendering.

En años recientes varias áreas de investigación han necesitado del VR, como técnica de análisis de sus datos. Las ciencias que más utilizan esta técnica de visualización de volúmenes son:

- *Medical Imaging*. Los datos volumétricos son obtenidos a partir de resonancias magnéticas, ultrasonidos, TAC, etc. y básicamente se espera que las aplicaciones permitan la manipulación de los datos con transformaciones de coordenadas, la manipulación del color, y la simulación preoperatoria, es decir, el planteamiento de una estrategia de cirugía sin el paciente.

- Modelado. Es usado principalmente por climatólogos, con el fin de visualizar el comportamiento de los fenómenos naturales y, por supuesto, el comportamiento del clima.
- Educación. A principios de la década de los noventa, la Biblioteca Nacional de medicina de los Estados Unidos dio pie a un proyecto llamado *Visible Human*, que consistió en diseccionar los cuerpos de un hombre y una mujer en pequeños cortes para digitalizarlos y construir cuerpos de sus respectivos sexos totalmente tridimensionales.
- Paleontología y Exploración de mantos petroleros y acuíferos. Con la información obtenida de las capas de la tierra, tales como presión, porosidad, temperatura, permeabilidad, etc. es posible construir un modelo tridimensional donde se muestren posibles puntos de localización de fósiles, mantos acuíferos, o pozos petroleros.
- Dinámica de fluidos computacionales. La visualización de modelos tridimensionales que muestren el comportamiento de los fluidos en diferentes superficies es posible gracias al análisis de *Navier-Stokes*, que proveen datos volumétricos sobre la vorticidad[8] y velocidad de los fluidos.

En conclusión, la técnica del VR no se ha desarrollado plenamente y aún falta mucho por estudia de esta técnica, sin embargo, los visualizadores hacen grandes esfuerzos y aportaciones para que el VR tenga un mayor desarrollo y, por tanto, un campo de estudio mayor.

[8] La vorticidad es una medida vectorial que caracteriza a la rotación que experimenta y a la que está sometido el fluido.

Capítulo III.

Introducción a IDL

IDL (*Interactive Data Language*) fue desarrollado, en 1981, por *David Stern*, quien trabajaba para RSI (*Research Systems Incorporated*). Aunque, originalmente, se enfocó a plataformas VMS, en años recientes, las nuevas versiones de IDL han incursionado en otros sistemas operativos para otorgar a los usuarios una mayor portabilidad de sus aplicaciones entre las diversas plataformas. En este aspecto, IDL es compatible con Microsoft Windows, VMS, MacOS y toda la familia UNIX; ello ha facilitado a que IDL sea ampliamente aceptado por la comunidad de científicos e investigadores.

La principal característica de IDL es que es un lenguaje orientado a arreglos, es decir, todas las operaciones y funciones que posee están diseñadas para operar con este tipo de datos; asimismo, posee un gran número de funciones, herramientas interactivas y un ambiente de programación agradable. Además, en Internet existen una serie de recursos y librerías, hechas por usuarios expertos, que ayudan a mejorar las capacidades de IDL. Las ventajas que IDL posee son las siguientes:

1. Todas las funciones de IDL están orientadas al manejo de arreglos, por lo que elimina la necesidad de utilizar ciclos para procesar un arreglo, a diferencia de lenguajes como C, Java o Fortran.
2. Las variables son dinámicas, es decir, adquieren su tipo en el momento en que se les asigna un valor.
3. Posee un gran número de funciones especializadas al análisis de datos y estadístico.
4. Posee una gran capacidad de visualización de datos gracias al uso de las librerías de OpenGL, otorgando una mayor rapidez de procesamiento.
5. Soporta una gran cantidad de formatos para realizar operaciones de entrada y salida de datos.
6. El código realizado en IDL garantiza portabilidad en cualquiera de las plataformas donde se encuentre instalado este lenguaje.
7. Provee de herramientas que permiten a los usuarios construir interfaces gráficas para combinarlas con sus aplicaciones. Además, recientemente, fue introducida la metodología orientada a objetos para acelerar el despliegue gráfico de los datos.
8. Por último, IDL tiene funciones que permiten hacer enlaces con lenguajes como C y Fortran con el fin de añadir una funcionalidad especializada.

3. 1. Conceptos Básicos.

IDL es un lenguaje procedural esencialmente, pero permite la construcción de unidades estructurales. IDL se compone por dos módulos: el primero se refiere al modo interactivo, donde las instrucciones se reciben en línea de comando. El segundo es el IDLDE (*IDL Developed Enviroment*), que incluye herramientas de edición y construcción de interfaces, un editor para escribir funciones o aplicaciones, herramientas de depuración y análisis de código, etc. La ventana principal de esta interfaz se compone de una barra de menú; una barra de herramientas; una ventana de proyectos activos; un panel del tipo MDI, donde se permiten tener varias ventanas con *scripts*; una ventana de salida, donde se muestra el resultado de las operaciones o errores de ejecución y compilación; una ventana donde se muestra información acerca de las variables utilizadas durante el procesamiento; una ventana donde se pueden escribir directamente comando de IDL para una rápida interacción; y, una barra de estado que indica la etapa de procesamiento. La interfaz IDLDE aparece por omisión en plataformas Windows y MacOS, mientras que las plataformas UNIX y VMS, sólo aparece la línea de comando y es necesario llamarlo con el comando IDLDE. Asimismo, IDL se instala con un manual de ayuda que puede ser invocado con el comando *HELP* o '*?*'.

3.1.1. Variables.

Las variables son utilizadas para almacenar algún tipo de dato o un conjunto de ellos. De acuerdo con la sintaxis de IDL, una variable está compuesta de dos atributos: la estructura y el tipo. El primer atributo hace referencia al valor asociado a esa variable, mientras que el segundo atributo hace referencia al tipo de dato que posee esa variable. En este aspecto, IDL maneja 12 tipos de datos resumidos en la tabla 3.1.

Tipo	Representación	Bits	Rango
Byte	*Byte*	8	[0, 255]
Entero con signo	*Int*	16	[-32 768, 32767]
Enteros sin signo	*Uint*	16	[0, 65 535]

Entero Largo con signo	*Long*	32	$[-2^{31}, 2^{31}-1]$
Entero Largo sin signo	*Ulong*	32	$[0, 2^{32}-1]$
Entero largo con signo	*long64*	64	$[-2^{63}, 2^{63}-1]$
Entero Largo sin signo	*ulong64*	64	$[0, 2^{64}-1]$
Flotante	*float*	32	$[-10^{38}, 10^{38}]$
Doble Precisión	*double*	64	$[-10^{308}, 10^{308}]$
Número complejo	*complex*	64	$[-10^{38}, 10^{38}]$
Número Complejo Doble	*dcomplex*	128	$[-10^{308}, 10^{308}]$
Cadena	*string*	8	[0, 32 767]

Tabla 3.1

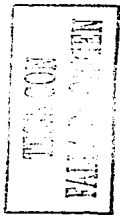
El nombre de una variable tiene una longitud mínima de un carácter y una longitud máxima de 128 caracteres, si se excede esta cifra los caracteres sobrantes son ignorados. El primer carácter debe ser alfabético seguido de cualquier otro carácter imprimible, exceptuando el punto, la arroba (@), y el espacio en blanco. Además, IDL es un lenguaje no sensitivo, es decir, no hace diferencia de caracteres alfabéticos en mayúsculas o minúsculas. Como IDL no es restrictivo al uso de variables es posible declarar una variable en cualquier momento que se necesite, únicamente se le asigna un valor o se define a partir de otra como en el ejemplo siguiente:

```
IDL > Variable_uno = 1.0
IDL > Variable_dos = Variable_uno
```

También IDL provee una forma para conocer el tipo de datos que corresponde a dicha variable, y esto se realiza a través del comando *HELP*, como se muestra a continuación:

```
IDL > HELP, Variable_uno
VARIABLE_UNO    FLOAT  =  1.0
```

Algunas veces es necesario convertir un tipo de dato a otro, para tal propósito, IDL posee una serie de funciones que realizan este trabajo, no



obstante, el uso de estas funciones implica una escisión, es decir, IDL trunca los valores de las variables sin enviar algún aviso previo.

1. *STRING*(). Convierte un dato en cadena.
2. *BYTE*(). Cambia un dato a byte.
3. *FIX*(). Transforma un dato a un entero de 16 bits
4. *UINT*(). Convierte un dato a un entero sin signo de 16 bits.
5. *LONG*(). Cambia un dato a un entero de 32 bits
6. *ULONG*(). Transforma un dato a un entero sin signo de 32 bits.
7. *LONG64*(). Muda un dato a un entero de 64 bits.
8. *ULONG64*(). Transforma un dato a un entero sin signo de 64 bits
9. *FLOAT*(). Cambia un dato en un flotante.
10. *DOUBLE*(). Convierte un dato en una cifra numérica de doble precisión.
11. *COMPLEX*(). Muda un dato en una cifra compleja, con parte real e imaginaria.
12. *DCOMPLEX*(). Cambia un dato en una cifra compleja de doble precisión, con su parte real y su parte imaginaria.

Existe una serie de variables internas que maneja IDL, las cuales almacenan información acerca del ambiente de desarrollo; estas variables de sistema no pueden ser modificadas de algún modo y están disponibles para todas las unidades de programa o bloques de ejecución. IDL clasifica las variables de sistema en cuatro rubros, a decir, variables, constantes, variables de gráficos, variables de ambiente y variables de manejo de errores. El rasgo distintivo de estas variables es que comienzan con el carácter de admiración, y pueden contener cualquier tipo de datos, incluyendo estructuras. Un programador puede definir una variable de sistema a través del procedimiento *DEFSYSV*, cuya sintaxis es:
DEFSYSV, Name, Value [, Read_Only] [, EXISTS = status].

Donde *Name* es el nombre de la variable; *Value* es el valor que contendrá la variable y puede ser de cualquier tipo; *Read_only* es una bandera binaria que indica si el valor de la variable puede ser modificado o no, por omisión la variable es de sólo lectura; y, *Status* es una variable donde se almacena el resultado de la operación. Las variables de sistema más utilizadas son:

1. *!VERSIÓN.* Es una estructura donde se almacena información acerca del sistema operativo en uso.
2. *!DIR.* Esta variable de sistema contiene la ruta absoluta de los directorios principales de IDL.
3. *!PATH.* Contiene la ruta de acceso a las diferentes librerías que utiliza IDL.
4. *!ERROR_STATE.* Es una estructura que guarda el estado del último error ocurrido.
5. *!PI.* Es un dato flotante que contiene el valor de pi (π).

El alcance y ciclo de vida de las variables se limita al bloque donde son declaradas, ello significa que IDL no maneja variables globales, excepto las variables del sistema. No obstante, en ocasiones es necesario utilizar variables globales, para que estén disponibles a lo largo de todas las unidades estructurales definidas. Para solucionar este problema IDL define la instrucción *COMMON*, la cual agrupa una serie de variables de cualquier tipo con el fin de otorgarles un alcance global. Existen dos manera de utilizar esta instrucción, la primera se conoce como definición del bloque, y la segunda como referencia al bloque.

La definición del bloque, como su nombre lo dice, consiste en establecer las variables que pertenecerán al bloque y cualquier unidad de programa podrá hacer referencia a esas variables, sólo cuando haga una referencia al bloque. La sintaxis de declaración de un bloque de variables es:
COMMON block_name, variable1, variable2,..., variableN.

Donde *block_name* es el nombre asociado al conjunto de variables, y *variableN* es la n-*ésima* variable contenida dentro del bloque. Cada variable puede adquirir un tipo de dato indistinto de acuerdo a las necesidades de programación. Cuando una unidad de programa desea utilizar las variables de un bloque, debe hacer referencia al mismo utilizando la instrucción *COMMON* y el nombre asociado a ese bloque, de esta manera, IDL replicará el bloque previamente definido, para que sea utilizado por la unidad de programa. Las variables de bloque conservarán el valor de la última operación realizada.

3.1.2. Operadores.

IDL maneja ocho tipos de operadores enfocados a la manipulación de arreglos y matrices. Estos operadores son:

3.1.2.1. Paréntesis.

Los paréntesis se utilizan para agrupar expresiones o encerrar los argumentos que recibirán las funciones. También son utilizados para alterar el orden de evaluación de una expresión. Ejemplo de uso de paréntesis son los siguientes:

```
IDL > Variable = SIN(ángulo * !Pi / 180)
IDL > Variable = ((A + 20)/300)-67
```

3.1.2.2. Corchetes Cuadrados.

En las primeras versiones los paréntesis eran utilizados para indexar arreglos, pero esto era demasiado confuso, ya que en ocasiones no se sabía si una variable era un arreglo o una función. Para solucionar este problema se utilizaron los corchetes cuadrados ([]), quienes se utilizan para definir un arreglo o subscribir[1] el índice del mismo, por ejemplo:

```
IDL > ARREGLO = [1, 2, 3, 4, 5] & PRINT, ARREGLO[2]
        1       2       3       4       5
```

3.1.2.3. Operadores Matemáticos.

Los operadores matemáticos son los clásicos que posee la mayoría de los lenguajes. Estos operadores son el operador de asignación (=), que sirve para asignar valores a una variable; operador de adición (+); operador de substracción (-), que sirve, también, para indicar el signo negativo de alguna variable; operador de multiplicación (*); el operador de división (/); el operador de potencia (^); y, el operador de módulo (*MOD*).

[1] Suscribir

```
IDL >A = 13.
IDL >A = 12 + 56.
IDL >A = 12.45 – 7.958
IDL >A = -13
IDL >A = 2 * 78
IDL >A = 12.98 / 4.56
IDL >A = 2 ^ 6
IDL >A = 35 MOD 6
```

3.1.2.4. Operadores Relacionales.

IDL maneja seis operadores relacionales que sirven para verificar la relación existente entre dos variables. estos operadores se pueden utilizar como condicional dentro de una instrucción para determinar si es verdadera o falsa; IDL maneja sus propias definiciones de verdadero y falso. Todos los valores impares enteros, cadenas no nulas y valores flotantes diferentes de cero, son tomados como verdaderos; en cualquier otro caso, se consideran como falsos. Los operadores relacionales son mostrados en la tabla 3.2.

Operador	Significado
EQ	Igual a
NE	Diferente a
GE	Mayor o igual
GT	Mayor que
LE	Menor o Igual
LT	Menor que

Tabla 3.2

3.1.2.5. Operadores Boléanos.

Estos operadores sólo son válidos cuando se utilizan valores enteros o binarios. en cualquier otro caso IDL lo considera un error. Básicamente existen 4 operadores boléanos. los cuales son: el operador *AND*, el cual regresa verdadero siempre que sus dos operandos sean verdaderos; el operador *OR*, que regresa verdadero cuando alguno de sus dos operandos es verdadero; el

operador *XOR*, que regresa verdadero cuando sus dos operandos tiene valores opuestos; y el operador *NOT*, que regresa el valor opuesto del operando.

3.1.2.6. Operadores de Multiplicación de Matrices.

En las primeras secciones de este capítulo se mencionaba que IDL es un lenguaje orientado a arreglos, por ello, se introdujeron dos nuevos operadores para facilitar las operaciones entre estos, sin utilizar ciclos. El primer operado corresponde al operador de multiplicación de vectores, representado por el carácter #, que calcula la multiplicación entre los elementos de un vector, multiplicando las columnas del primer vector por los renglones del segundo vector. El segundo vector debe tener el mismo número de columnas que el primer vector.

El segundo operador es el operador de multiplicación de matrices, representado por el carácter ##, que calcula la multiplicación de dos matrices multiplicando los elementos del primer renglón de la primera matriz con los elementos de la primera columna de la segunda matriz. El resultado es una matriz del mismo número de renglones que la primer matriz y el mismo número de columnas de la segunda matriz.

3.1.2.7. Operadores de Concatenación.

Los corchetes cuadrados ([]) pueden utilizarse para concatenar arreglos a partir de valores escalares, vectores, o los dos. Cada uno de estos valores debe estar separado por comas y encerrado por los operadores de concatenación.

```
IDL >A = [1, 2, 3,4, 5]  &  B = [[6, 7, 8, 9, 0], [11, 12, 13, 14, 15]]
IDL >C = [A, B]
```

3.1.2.8. Operadores de Máximo y Mínimo Valor.

Los operadores de mayor (>) y menor (<) regresan el valor mayor o el menor de sus dos operandos, respectivamente. IDL exige que los valores negativos deben estar encerrados entre paréntesis para que haya una interpretación correcta al momento de realizar las operaciones pertinentes.

Al igual que la mayoría de los lenguajes de alto nivel, IDL maneja un valor de prioridad para cada uno de los operadores. La tabla 3.3 muestra la prioridad asociada a cada uno de los operadores.

Prioridad	Operador
Primera	Paréntesis ()
Segunda	* apuntador
	^ potencia
Tercera	* multiplicación
	Operador # y ##
	/ División
	Módulo (MOD)
Cuarta	+ Suma
	- Substracción
	> Mayor que
	< Menor que
	NOT
Quinta	EQ
	NE
	GE
	LE
	GT
	LT
Sexto	AND
	OR
	XOR
Séptima	Operador Ternario (?:)

Tabla 3.3

3.1.3. Arreglos.

En la mayoría de los lenguajes de alto nivel[2], un arreglo es un conjunto de valores direccionados en una sola dirección de memoria, y mediante un índice se puede hacer referencia a cada uno de los elementos en el arreglo; este mismo concepto es manejado por IDL, pero a diferencia de lenguajes como C o Java, IDL almacena los arreglos en forma de columnas, es decir, los elementos son almacenados consecutivamente en memoria de acuerdo a su posición dentro de una columna.

IDL permite que los arreglos sean de cualquier tipo de dato y pueden manejar arreglos de hasta 8 dimensiones. Los arreglos pueden ser creados mediante el uso de los corchetes cuadrados, encerrando a los valores y separándolos con comas, tal como sigue:

```
IDL > Vector = [1, 2, 3, 4, 5, 6, 7]
```

Un arreglo de una sola dimensión se le conoce como vector, mientras que un arreglo de dos dimensiones se le conoce como matriz, y puede ser declarado como:

```
IDL > Matriz = [[1.0,2.1,3.2,4.3,5.4], [6.5,7.6,8.7,9.8,0]]
```

Si se desea obtener información acerca de las variables *Vector* y *Matriz* es necesario utilizar el comando HELP como sigue:

```
IDL > HELP, Vector, Matriz
VECTOR          INT             =Array  [5]
MATRIZ          FLOAT           =Array[5, 2]
```

Además, IDL posee algunas funciones para crear arreglos de algún tipo e inicializar los elementos del arreglo en ceros o de acuerdo al índice que le corresponde, por ejemplo, las funciones *INTARR()* y *INDGEN()* crean un arreglo de *N* dimensiones, la primera inicializa todos los elementos a cero, mientras que la segunda les asigna como valor el índice que le corresponde. Es importante recalcar que IDL, al igual que C, inicia el índice de su arreglo en

[2] Un lenguaje de alto nivel es aquel que utiliza instrucciones más semejantes la lenguaje humano, por tal razón, es más fácil de enterder, no obstante, es necesario compilarlo para traducirlo a nivel de bits y sean entendible para la computadora.

cero, a decir, si se tiene un arreglo de *N* elementos. IDL maneja este arreglo de *0* a *N-1* elementos.

```
IDL>ceros = INTARR(6) & indices = INDGEN(6) & PRINT, ceros, indices
      0       0       0       0       0       0
      0       1       2       3       4       5
```

Ambas funciones reciben como argumento el número de dimensiones que se desea, de esta forma, si queremos declarar una matriz de *12 x 15*, basta con usar el siguiente ejemplo, *matriz = INDGEN(12, 15)*. Para cada uno de los tipos de datos que maneja IDL, existen sus respectivas funciones y todas reciben como argumentos el número de dimensiones deseado. En la tabla 3.4 se enlistan tales funciones.

Tipo de Dato	Iniciación en ceros	Iniciación por índices
byte	*BYTARR()*	*BINDGEN()*
int	*INTARR()*	*INDGEN()*
uint	*UINTARR()*	*UINDGEN()*
long	*LONARR()*	*LINDGEN()*
ulong	*ULONARR()*	*ULINDGEN()*
long64	*LON64ARR()*	*L64INDGEN()*
ulong64	*ULON64ARR()*	*UL64INDGEN()*
float	*FLTARR()*	*FINDGEN()*
double	*DBLARR()*	*DINDGEN()*
complex	*COMPLEXARR()*	*CINDGEN()*
dcomplex	*DCOMPLEXARR()*	*DCINDGEN()*
string	*STRARR()*	*SINDGEN()*

Tabla 3.4

Existen otras dos funciones que pueden crear arreglos e inicializarlos en algún valor arbitrario, de acuerdo a las necesidades del usuario; estas funciones son *REPLICATE()* y *MAKE_ARRAY()*, cuya sintaxis es la siguiente:

Result = REPLICATE (Value, D1 [,..., D8])

Result = MAKE_ARRAY ([D1, ..., D8] [, /BYTE | , /COMPLEX | , /DCOMPLEX | , /DOUBLE | , /FLOAT | , /INT | , /L64 | , /LONG | , /OBJ, | , /PTR | , /STRING | , /UINT | , /UL64 | , /ULONG] [, DIMENSION=vector] [, /INDEX] [, /NOZERO] [, SIZE=vector] [, TYPE=type_code] [, VALUE=value])

Donde D_n es la dimensión del arreglo, *Value* es el valor que contendrá cada uno de los elementos del arreglo, y en el caso de *MAKE_ARRAY()*, se utilizan palabras claves para indicar el tipo de datos que contendrá el arreglo. Por ejemplo,

```
IDL > arreglo = REPLICATE( 4.345,  2,  3)
IDL > arreglo_dos = MAKE_ARRAY(2, 3, /FLOAT, VALUE = 4.345)
IDL > arreglo_tres = MAKE_ARRAY(SIZE = [2, 3], TYPE = 4, /NOZERO, /INDEX)
```

En los ejemplos anteriores, *arreglo* es una matriz de *2 x 3* y cuyos elementos tiene el valor de 4.345, asimismo, *arreglo_dos* posee las mismas características que *arreglo*, mientras *arreglo_tres* es una matriz con las mismas dimensiones que las anteriores del tipo flotante, pero sus elementos tiene el valor del índice que les corresponde. En el caso de la palabra reservada *TYPE*, IDL maneja un valor entero asociado para definir cada tipo de dato, tal como lo muestra la tabla 3.5.

Código asignado	*Tipo de Dato*
0	*Indefinido*
1	*Byte*
2	*Entero*
3	*Entero de 32 bits*
4	*Flotante*
5	*Doble precisión*
6	*Complejo Flotante*
7	*Cadena*
8	*Estructura*
9	*Complejo de Doble precisión*
10	*Apuntador*
11	*Apuntador a un Objeto*
12	*Entero sin signo*
13	*Entero de 32 bits sin signo*
14	*Entero de 64 bits*
15	*Entero de 64 bits sin signo*

Tabla 3.5

IDL permite acceder a los elementos de un arreglo usando índices, a través de alguna de las siguientes formas *arreglo[índice]* o *(expresión)[índice]*, donde *índice* es un tipo de dato *LONG*. Por ejemplo:

```
IDL > arreglo = INDGEN(10) *3  &  indice = 5  &  PRINT, arreglo[indice]
      15
IDL > PRINT, (arreglo * 10)[indice]
      150
```

Si el índice sobrepasa el rango de las dimensiones, IDL produce una excepción del tipo "fuera de rango" y se detiene la ejecución de la unidad de programa. En ocasiones es necesario acceder a un rango de elemento dentro del mismo arreglo, para ello IDL permite utilizar índices que indiquen un rango de la forma *[índice inicial: índice final]*, donde *índice final* debe ser mayor o igual a *índice inicial*, y ninguno de los dos índices debe superar el número de dimensiones del arreglo. Por ejemplo, si la variable denominada *arreglo* es un vector de 200 elementos, la instrucción: *arreglo[50:125]* hará referencia desde el elemento número 50 hasta el elemento número 125. Existe otra forma de hacer referencia a cierto rango de elementos, y es a través del carácter asterisco (*), el cual indica todos los elementos, si se desea imprimir todos los elementos de la variable *arreglo* a partir del centésimo elemento, basta con escribir: *PRINT, arreglo[100: *]*, además, si el asterisco se especifica como índice, entonces hará referencia a todos los elementos del arreglo. Las anteriores reglas se pueden resumir en la tabla 3.6:

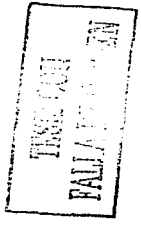
Índice	*Resultado*
i	*Hace referencia sólo al i-ésimo elemento*
$i_m : i_n$	*Hace referencia del m-ésimo elemento hasta el n-ésimo elemento*
i_n : *	*Hace referencia del n-ésimo elemento hasta el final del arreglo*
*	*Hace referencia a todos los elementos del arreglo*

Tabla 3.6

En el caso de arreglos con dimensiones mayores a uno, las reglas anteriores son válidas y pueden combinarse en cada una de las dimensiones del arreglo de acuerdo a las necesidades del usuario, por ejemplo:

```
IDL > matriz = FINGEN(10, 10, 10) * 0.5  &  variable = matriz[ 5,  1: 3, *]  &
HELP, matriz, variable
MATRIZ      FLOAT    = Array[10, 10, 10]
VARIABLE    FLOAT    = Array[1, 3, 10]
```

IDL tiene varias funciones que permiten manipular los arreglos de manera rápida y sencilla, útiles para aquellas situaciones donde se desea conocer el valor máximo o mínimo del arreglo, la sumatoria de todos los elementos, la desviación estándar, etc. A continuación se mencionan las funciones más útiles con una breve descripción de su funcionalidad.



1. *N_ELEMENTS()*. Devuelve un valor escalar indicando el número de elementos contenidos en un arreglo.
2. *SIZE()*. Regresa una vector con información a cerca del arreglo.
3. *MIN()*. Devuelve un valor escalar indicando el mínimo valor dentro del arreglo.
4. *MAX()*.Regresa un valor escalar indicando el máximo valor dentro del arreglo.
5. *MEAN()*. Regresa el promedio de valores contenidos en el arreglo.
6. *VARIANCE()*. Obtiene la varianza del arreglo.
7. *STDDEV()*.Obtiene la desviación estándar del arreglo.
8. *MOMENT()*.Obtiene el promedio, la desviación estándar, la varianza y población.
9. *TOTAL()*. Calcula la sumatoria de todos los elementos contenidos en el arreglo.
10. *REFORM()*. Cambia las dimensiones de un arreglo sin cambiar su contenido.
11. *REVERSE()*. Ordena de manera inversa los elementos de un arreglo.
12. *ROTATE()*. Rota *N* grados los elementos de un arreglo.
13. *TRANSPOSE()*. Calcula la transpuesta de un arreglo.
14. *SHIFT()*. Desplaza los elementos de un arreglo *N* posiciones.
15. *SORT()*. Ordena los elementos de un arreglo.
16. *UNIQ()*. Crea un arreglo con valores únicos a partir de otro arreglo previamente ordenado.
17. *REBIN()*. Cambia las dimensiones de un arreglo.
18. *CONGRID()*. Expande o acorta las dimensiones de un arreglo.
19. *INTERPOLATE()*. Calcula la interpolación lineal, bilineal o trilineal.
20. *WHERE()*. Encuentra uno o más valores dentro de un arreglo de acuerdo a un criterio de búsqueda. Este criterio puede utilizar los operadores relaciónales.

3.1.4. Estructuras.

Una estructura es una colección de variables escalares, vectoriales, o de cualquier otro tipo. Básicamente existen dos tipos de estructuras: las nombradas y las anónimas. Las estructuras nombradas (*Named Structures*) poseen un nombre único que las identifica, y cualquier instancia de estas estructuras comparte la misma definición y no se permite cambios en las misma, incluyendo la redefinición de sus miembros. Las estructura anónimas son todo lo contrario pues no tiene un nombre que las identifica por lo que IDL les asigna un identificador interno. Asimismo, su definición puede variar de acuerdo a las necesidades de cada instancia. Existen dos formas para crear una estructura, la primera es a través del uso de la función *CREATE_STRUCT()*, y la segunda mediante la definición de las llaves ({}), por ejemplo:

```
IDL > variable = CREATE_STRUCT (NAME = "lista", ['A', 'B', 'C'], 1, 2, 3)
```

Donde *variable* es una estructura con el nombre de *lista*, cuyos campos son *A*, *B* y *C*, con los valores *1*, *2* y *3*, respectivamente. Si se omite el nombre se crea una estructura anónima, como en el siguiente ejemplo:

```
IDL> variable = CREATE_STRUCT ('Nombre', 'Luis', 'Edad', 25)
```

En este caso, se crea una estructura anónima con los campos *Nombre* y *Edad* y cuyos valores son *Luis* y *25*, respectivamente. La segunda forma para definir una estructura es mucho más intuitiva:

```
IDL > variable = {lista, A: 1, B: 2, C: 3}
IDL > variable2 = {Nombre: "Luis", Edad: 25}
```

En ambos casos es posible obtener información sobre las estructuras utilizando el comando *HELP*, adicionándole la palabra reservada *STRUCTURE*, por ejemplo:

```
IDL > HELP, variable2, /STRUCTURE
**      STRUCTURE <1052178>, 2 tags, length = 262160, refs = 1:
        NOMBRE          STRING                  'Luis'
        EDAD            INT             25
```

En este caso, IDL nos muestra el identificador asociado a la estructura anónima, el número de campos, la longitud y el número de instancias de la estructura, también muestra los nombres de los campos, su tipo de dato y el valor actual de los mismos. Adicionalmente, es posible conocer los nombres de

los campos y el número de ellos utilizando las funciones *TAG_NAMES()* y *N_TAGS()*, respectivamente:

```
IDL > PRINT, TAG_NAMES(variable2),  N_TAGS(variable2)
        NOMBRE          EDAD            2
```


La sintaxis básica para referirse a un campo de la estructura es *variable.campo*. Es posible que las estructuras tengan arreglos como miembros, en ese caso es posible aplicar las reglas de indexado de arreglos. También se puede dar el caso de que una estructura contenga otras estructuras como miembros, en tal situación la referencia a los campos puede ser con la siguiente sintaxis *variable.campo.campo*. Ejemplo:

```
IDL> PRINT, variable2.Nombre,  variable2.Edad
        Luis      25
```

Para crear instancias de estructuras ya definidas sólo se necesita hacer una asignación entre variables, o en el caso de estructuras nombradas, llamarlas a través de llaves, por ejemplo:

```
IDL > variable3 = variable2  &  variable4 = {lista}
```

Otra manera de crear múltiples instancias de estructuras y almacenarlas en arreglos es a través de la función *REPLICATE*, la cual crea *N* instancias de un tipo de dato, por ejemplo:

```
IDL >  variable5 = REPLICATE (variable2, 100)
```

Donde *variable5* es un arreglo de 100 elementos y cada uno de ellos es una instancia de la estructura *variable2*. En el caso de estructuras nombradas se utiliza el nombre de la estructura, como se muestra a continuación:

```
IDL >  variable6 = REPLICATE ({lista}, 100)
```

Tanto en arreglos de estructuras, como arreglos miembros de estructuras, es posible aplicar todas las reglas de indexado mencionadas en la sección anterior. De esta manera, IDL sólo define cuatro tipos de acceso a los miembros de una estructura:

- *Estructura.Campo.*
- *Estructura.Campo[índice].*
- *Estructura[índice].Campo.*
- *Estructura[índice].Campo[índice].*

Por último, una propiedad más que poseen las estructuras es que pueden heredarse, es decir, puede definirse una nueva estructura heredando los campos de otra estructura y adicionando sus propios campos, de esta manera, podemos crear una estructura que contenga los campos *teléfono* y *código postal*, y que herede de la estructura *variable2*. La palabra reservada *INHERITE* hace que la estructura *herencia* adquiera los mismos campos que contiene la estructura *variable2*, así:

```
IDL > herencia = {Datos, INHERITE variable2, Teléfono: "12345678", CP: "123456"}
```

3.1.5. Apuntadores.

Los apuntadores son referencias a variables ya definidas previamente y son ampliamente utilizados para crear estructuras de datos. A partir de la versión 5.0 de IDL, se introdujo el manejo de punteros utilizando variables de tipo *heap*, que otorgan una gran ventaja sobre las variables de tipo bloque en cuanto al ciclo de vida y alcance. Las variables *heap* son variables dinámicamente direccionadas por IDL y sus características permiten el uso de la metodología orientada a objetos, además, proveen de funcionalidades para la restauración y almacenamiento de las mismas, usando las funciones *SAVE*() y *RESTORE*(). Para crear un apuntador a una variable *heap* se utiliza la función *PTR_NEW*():

```
IDL > puntero = PTR_NEW(2.34)
IDL > HELP, puntero
PUNTERO          POINTER      =      <PtrHeapVar1>
```

En el anterior ejemplo se creó una variable llamada *puntero* que apunta a una variable tipo *heap* cuyo valor es 2.34. Igualmente, la función *PTR_NEW*() permite crear punteros a variables ya existentes, por ejemplo:

```
IDL >  array = INDGEN (10, * 5 &L pointer = PTR_NEW (array) &L HELP, pointer
<PtrHeapVar2>  INT      =      ARRAY [10]
```

La explicación lógica del anterior ejemplo es que la variable *pointer* es un apuntador a la variable *array*, sin embargo, esto es erróneo. IDL crea una variable de tipo *heap* que contiene los mismos valores que la variable *array* y, entonces, *pointer* es una variable que apunta a la variable *heap* y no a la variable *array*. Opcionalmente, la función *PTR_NEW*() admite el argumento *NO_COPY*, que evitaría la creación de la variable *heap* y, entonces, la variable *pointer*

apuntaría a la variable *array*. Para de-referenciar un apuntador sólo se necesita anteponer el carácter asterisco (*) al nombre de la variable puntero, tal como se muestra:

```
IDL >  PRINT, *pointer
       0       5       10      15      20      25      30      35      40
       45
```

Otro argumento opcional de la función *PRT_NEW()* es *ALLOCATE_HEAP*, que permite reservar memoria para un tipo de dato no especificado, es decir, crea una variable *heap* no definida a la cual podrá asignársele un tipo de dato después de la declaración del apuntador:

```
IDL > puntero2 = PTR_NEW (/ALLOCATE_HEAP) &L HELP, *puntero2
      <PtrHeapVar3>  UNDEFINED              = <Undefined>
IDL > puntero2 = COS(!PI) &L HELP, *puntero2
      <PtrHeapVar3>  FLOAT          = 1.0
```

Por ultimo, la función *PTR_NEW()* sin ningún argumento crea punteros nulos, necesarios para definir punteros que aún no tienen valor de inicio. Intentar de-referenciar un apuntador nulo causa un error.

```
IDL > puntero3 = PTR_NEW() &L HELP, puntero3 &L PRINT, *puntero3
      A               POINTER = <NullPointer>
Unable to dereference NULL pointer: A.
Execution halted at:  $MAIN$
```

Para destruir una variable *heap* o liberar la memoria de un apuntador, IDL tiene el procedimiento *PTR_FREE*. Es importante que al finalizar el uso de un apuntador sea inmediatamente liberada la memoria, de esta manera, tendremos un código más eficiente y limpio. La sintaxis del procedimiento *PTR_FREE* es:

```
IDL > PTR_FREE, puntero3, puntero2, puntero, pointer
```

La función *PTR_VALID()* verifica la validez de un puntero regresando un valor binario que indica si el apuntador es válido o no; en caso de que no se le pase ningún argumento, esta función regresa un vector indicando cuantos punteros válidos existen en la sesión IDLDE hasta ese momento.

```
IDL > PRINT, PTR_VALID (puntero3) &L puntero = PTR_NEW("Hola")
      0
IDL > PRINT, PTR_VALID (puntero)
      1
```

```
IDL > puntero = PTR_NEW("Hola") && puntero2 = PTR_NEW("Que") && puntero3 =
PTR_NEW("onda?")
IDL > Vector = PTR_VALID()
IDL > PRINT, vector
<PtrHeapVar6><PtrHeapVar7><PtrHeapVar8>
IDL > HELP, vector
VECTOR      POINTER  = Array[3]
IDL > PTR_FREE, vector
```

La flexibilidad de IDL permite crear arreglos de punteros mediante la función *PTRARR*(), la cual recibe las dimensiones del arreglo y regresa un apuntador al arreglo de *N* dimensiones, donde todos sus elementos son punteros nulos, si se desea evitar esta inicialización es necesario especificar la palabra reservada */ALLOCATE_HEAP*.

```
IDL> puntero2 = PTRARR(5, 5) && HELP, puntero2, puntero2[0,0]
PUNTERO2              POINTER  = Array(5, 5)
<Expression>          POINTER  = < NullPointer >
IDL > puntero3 = PTRARR(5, /ALLOCATE_HEAP) && HELP, puntero3, puntero3[0]
PUNTERO3    POINTER  = Array[5]
<Expression>  POINTER  = <PtrHeapVar2>
```


3.1.5.1. Operaciones con Apuntadores.

Existen dos operaciones básicas que se realizan con punteros: asignación y desreferencia. En el primer caso, los punteros pueden asignarse a otras variables, tal como se muestra en el siguiente ejemplo, ambas variables apuntan a la misma variable *heap*:

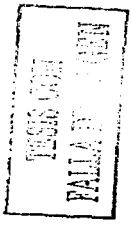
```
IDL > a = PTR_NEW(FINDGEN(10)) && b = a  && HELP, a, b
A           POINTER  = <PtrHeapVar1>
B           POINTER  = <PtrHeapVar1>
```

Como se mencionó en párrafos anteriores, la manera de desreferenciar un apuntador es mediante el uso del asterisco (*), sin embargo, IDL contempla varios casos comunes donde las operaciones de desreferencia pueden causar errores.

a) Valores Escalares.

Cuando un apuntador apunta, valga la redundancia, a un valor escalar, la manera de dereferenciarlo es antecediendo el asterisco (*) al nombre de la variable, como se muestra a continuación:

```
IDL > puntero = PTR_NEW(15) & PRINT, *puntero
        15
```



b) Arreglo de Punteros.

En el caso de arreglos de punteros, cada elemento debe ser de-referenciado de manera independiente, intentar de-referenciar al arreglo completo sería un error:

```
IDL > ptarr = PTRARR(3, /ALLOCATE_HEAP)
IDL > FOR i = 0,2 DO *ptarr[i] = i
IDL > PRINT, *ptarr
% Expression must be a scalar in this context: PTARR.
% Execution halted at:  $MAIN$
```

La mejor manera de acceder a cada uno de los elementos de la variable *ptarr* es mediante el uso de un ciclo:

```
IDL > FOR i = 0, N_ELEMENTS (ptarr)-1 DO PRINT, *ptarr[i]
```

c) Matrices.

Cuando un apuntador apunta a una matriz, es posible de-referenciarla como en el primer caso:

```
IDL > puntero = PTR_NEW(INDGEN(2, 2) & PRINT, *puntero
        1       2       3
```

No obstante, en ocasiones sólo se desea acceder a un sólo elemento del arreglo o a un rango dentro del arreglo. El razonamiento lógico indica que la siguiente instrucción es correcta:

```
IDL > PRINT, *puntero[*, 0]
```

Pero IDL marca un error de sintaxis, ya que la función *PRINT* espera un escalar. El método correcto es encerrar la variable *puntero* en paréntesis y después indicar el índice del elemento al cual se desea acceder:

```
IDL > PRINT, (*puntero)[*,0]
```

De esta manera, IDL primero de-referenciará a la variable *puntero*, después aplicará una búsqueda para imprimir los elementos requeridos.

d) Estructuras.

En el caso de las estructuras y punteros se pueden llegar a manejar varias situaciones. Primero, cuando un campo de alguna estructura es un apuntador, la manera de de-referenciarlo es antecediendo el nombre de la variable que contiene la estructura con el carácter asterisco, por ejemplo:

```
IDL > estructura = {Campo1: "Campo1", Campo2: PTR_NEW ("Campo2")}
IDL > PRINT, *estructura.Campo2
      Campo2
```

Una segunda situación ocurre cuando se crea un apuntador a una estructura y se desea acceder a los campos correspondientes mediante el apuntador. Para resolver esta condición se aplica la misma regla utilizadas en las matrices, es decir, la variable a de-referenciar se encierra entre paréntesis y después se indica el campo requerido, por ejemplo:

```
IDL > puntero = PTR_NEW (estructura)
IDL > PRINT, (*puntero).Campo1
      Campo1
```

La tercera situación es una conjunción de las dos anteriores. Si la variable *puntero* es utilizada para referenciar a la variable *estructura* que contiene un apuntador como campo, entonces, es necesario de-referenciar a la variable *puntero* mediante el uso de los paréntesis y antecederlo con el carácter asterisco, tal como se muestra a continuación:

```
IDL > PRINT, *(*puntero).Campo2
```

e) Punteros.

El último caso ocurre cuando se desea de-referenciar un apuntador que apunta a otro apuntador. La manera correcta es antecediendo dos veces el asterisco:

```
IDL > puntero1 = PTR_NEW(1000) &IDL puntero2 = PTR_NEW(puntero1)
IDL > PRINT, **puntero2
      1000
```

3.1.5.2. Problemas con el Uso de Apuntadores.

En IDL se manejan dos tipos de situaciones más recurrentes en errores con punteros. La primera se le conoce como referencias pendientes (*Dangling References*), la cual ocurre cuando se intenta referenciar a un apuntador nulo o

un apuntador que no existe, situación que se puede evitar con una llamada previa a *PTR_VALID()* para verificar la validez del apuntador:

```
IDL> puntero = PTR_NEW(2.5) & PTR_FREE, puntero & PRINT, *puntero
% Invalid pointer: PUNTERO.
% Execution halted at: $MAIN$
IDL> HELP, puntero
PUNTERO      POINTER  = <PtrHeapVar1>
```

La segunda situación se le conoce como filtración de memoria (*Leakage Memory*), la cual consiste en utilizar el mismo nombre de variable para crear *n* número de punteros:

```
IDL> puntero = PTR_NEW(BINDGEN(7)) & HELP, puntero
PUNTERO      POINTER  = <PtrHeapVar2>
IDL> puntero = PTR_NEW("cadena") & HELP, puntero
PUNTERO      POINTER  = <PtrHeapVar3>
IDL> puntero = PTR_NEW(2.5) & HELP, puntero
PUNTERO      POINTER  = <PtrHeapVar4>
IDL> HELP, /HEAP_VARIABLES
Heap Variables:
   # Pointer: 3
   # Object : 0
<PtrHeapVar2>  BYTE    = Array[7]
<PtrHeapVar3>  STRING  = 'cadena'
<PtrHeapVar4>  FLOAT   =      2.50000
```

En este caso, la memoria previa de la variable denominada *puntero* se pierde, ya no puede ser referenciada, aunque IDL la mantenga con vida. Para eliminar esas variables es necesario utilizar el colector de basura que IDL posee. Este colector se encarga de buscar todas aquellas variables, de cualquier tipo, que no son referenciadas y eliminarlas, algunos autores se niegan a utilizar este procedimiento ya que IDL aún no ha perfeccionado esta herramienta y pueden ocurrir errores cuando se utiliza, no obstante, en este ejemplo, se utiliza esta herramienta para eliminar las variables <*PtrHeapVar2*> y <*PtrHeapVar3*>.

```
IDL> HEAP_GC, /VERBOSE
<PtrHeapVar2>  BYTE    = Array[7]
<PtrHeapVar3>  STRING  = 'cadena'
```

3.1.6. Cadenas.

Las cadenas, bajo la concepción de IDL, son una secuencia de caracteres con una longitud máxima de 32,767 caracteres. El tamaño de una cadena es dinámico, es decir, la cadena puede crecer o acortase de acuerdo a las necesidades del usuario sin la necesidad de hacer declaraciones previas sobre el tamaño inicial de la cadena. IDL permite crear arreglos de cadenas de *N* dimensiones, donde cada elemento del arreglo es de tamaño arbitrario e independiente de los demás elementos.

Los procedimientos *PRINT, PRINTF* y la función *STRING*() convierten cualquier expresión en una cadena; las dos últimas se auxilian de un formato para realizar una mejor conversión. Además, posee varias herramientas para operar con cadenas, como la concatenación entre ellas, formateo de cadenas, la extracción de cadenas a partir de otras, etc. Para cada una de estas operaciones IDL tiene una función especifica.

1. *STRLEN*(). Regresa la longitud de una cadena.
2. *STRLOWCASE*(). Convierte a letras minúsculas.
3. *STRUPCASE*(). Convierte a letras mayúsculas.
4. *STRCOMPRESS*(). Reduce a uno solo todos los espacios en blanco.
5. *STRTRIM*(). Elimina todos los espacios en blanco a la derecha e izquierda.
6. *STRPOS*(). Regresa la posición de la primera ocurrencia de una cadena.
7. *STRMID*(). Extrae *N* número de caracteres, a partir de una posición arbitraria.
8. *STRPUT*(). Inserta una cadena en otra a partir de una posición especificada
9. *STRSPLIT*(). Esta función rompe una cadena en *tokens*.
10. *STRJOIN*(). Junta varias cadenas en una sola.
11. *STRCMP*(). Compara dos cadenas.
12. *STRMATCH*(). Busca una cadena dentro de otra.
13. *STRREGEX*(). Busca todos los patrones posibles dentro de una cadena base.

3.1.6.1. Expresiones Regulares.

Las expresiones regulares son una poderosa herramienta para buscar cadenas que cumplan con cierto patrón. En el sistema operativo UNIX, las expresiones regulares son ampliamente utilizadas por todos los comandos de administración. IDL implementa esta misma lógica para facilitar al usuario la búsqueda de cadenas, además de que exporta esta herramienta a otras plataformas. Las expresiones regulares se componen a partir de meta caracteres, los cuales son caracteres con significados especiales. Los meta caracteres que define IDL están definidos en la tabla 3.7.

Carácter	Descripción
.	El punto representa cualquier carácter.
[ ]	Los corchetes cuadrados tienen diferentes simbolismos. En ocasiones pueden indicar un rango de caracteres, en otras pueden indicar un patrón de compatibilidad.
\	La diagonal inversa suprime el significado especial al carácter que le precede otorgándole el significado de un carácter ordinario.
( )	Los paréntesis indican la repetición de uno o más caracteres, por ejemplo, si se tiene la expresión "(sol){3}", entonces será compatible con la cadena "solsolsol", mientras que la expresión "sol{3}" será compatible con la cadena "solll"
*	Indica cero o más caracteres de una expresión, por ejemplo, la expresión "a*" es compatible con las cadenas "a", "aa", "aaaa", etc.
+	Igual que la expresión anterior, pero esta indica uno o más caracteres
?	Indica cero o un carácter.
{ }	Indica un número de repeticiones posibles, o un rango, para un carácter determinado.
|	Indica que puede ocurrir una u otra selección de acuerdo a una lista de patrones. Por ejemplo la expresión "(a | b | c)z" será compatible con alguna de las cadenas 'az', 'bz', o 'cz'.
^	El apóstrofe puede tener varios significados de acuerdo a la posición que tome dentro de una expresión regular. Si se pone al principio de la expresión regular, indica que todas las cadenas compatibles deberán iniciar con el carácter que le

	precede, por ejemplo, si se tiene la expresión regular "^a", serán compatibles todas aquellas cadenas que comiencen con la letra a. En cambio, si la ubicación del apóstrofe está en otra parte de la expresión regular, ello indica todos las cadenas que no contengan el carácter que le precede.
$	Indica el final de una cadena. Por ejemplo, si se tiene la siguiente expresión regular: "*e$", entonces serán compatibles todas aquellas cadenas que finalicen con la letra e.

Tabla 3.7

3.1.6.2. Formateo de Cadenas.

IDL provee de una función que convierte cualquier expresión o variable en una cadena, dicha función es *STRING()*, que recibe como parámetros las expresiones a convertir en cadenas. Como argumento opcional recibe una serie de caracteres que indican el formato de salida que se debe utilizar al momento de realizar la conversión. La sintaxis básica de la función *STRING()* es:
IDL > PRINT, STRING (expr1, expr2, ..., exprN, FORMAT = '(q1f1 s1 f2 s2 ... fn qn)')

Cada una de las expresiones a convertir debe tener su propio secuencia de formato, compuesto por un terminador de cadena, un código de formato y, un separador. El terminador de cadena, representado por *q*, indica que la salida debe ser movida a la siguiente línea. El código de formato *f*, especifica la manera de transferencia de los datos. Los separadores de campo *s*, consisten en una o más comas que funcionan como separadores.

3.2. Programación.

IDL maneja tres tipos de programación: la programación básica, la programación secuencial y la programación estructurada. La programación básica consiste en la ejecución de instrucciones IDL directamente desde la línea de comando, este método es muy utilizado cuando son pocas instrucciones que se desean ejecutar, y cuando se requiere una mayor velocidad

para obtener los resultados esperados. Los ejemplos manejados en la sección anterior ejemplifican el uso de este tipo de programación.

La programación secuencial consiste en un conjunto de instrucciones IDL almacenadas en un archivo de texto, y que son ejecutados de manera secuencial hasta encontrar la instrucción *END*. El ambiente de desarrollo de IDL posee un editor cuyas características facilitan la construcción de *scripts* o programas secuenciales, ya que maneja colores para diferenciar las palabras reservadas de IDL, los nombres de variables, los nombres de procedimientos o funciones. El archivo de texto también puede ser editado en cualquier otro editor disponible en la plataforma de uso, únicamente se debe guardar con la extensión PRO.

Los programas secuenciales deben ser compilados antes de ser ejecutados. En el caso de plataformas UNIX y VMS, es necesario especificar la ruta completa del archivo junto con el comando *COMPILE* desde la línea de comandos:
IDL > .COMPILE, Archivo.pro

Nótese que el comando *COMPILE* va antecedido por un punto (.). Si el programa secuencial o *script* tiene errores de sintaxis, IDL indicará la línea donde se encuentra el posible error junto con una breve descripción del mismo. Si no tiene errores, entonces puede ejecutarse utilizando el comando *RUN*:
IDL > .RUN, Archivo.pro

Antes de ejecutar el *script*, el comando *RUN* realiza una compilación previa y después ejecuta el *script*. Si durante la ejecución ocurre un error de lógica, IDL envía los mensajes de error correspondiente y detiene la ejecución. En este aspecto, es necesario recalcar que IDL no detecta algunos errores de lógica que pueden manifestarse cuando se ejecuta un programa secuencial, originando que IDL se detenga totalmente y sea necesario reiniciarlo.

En el caso de las plataformas Microsoft Windows y MacOS, el ambiente de desarrollo de IDL, evita el uso de los comandos *COMPILE* y *RUN*. En vez de ello, el *script* a compilar se abre en editor, y presionando las secuencia de teclas *ctrl. - F5*, IDLDE compila automáticamente el programa secuencial. Si existen errores en el programa, IDL marcará las líneas del posible error con un punto rojo. Si no existen errores, la ejecución del programa se realiza oprimiendo la

tecla *F5*. En el ambiente IDLDE puede ocurrir la misma situación que se explicó anteriormente con los errores de lógica.

Por último, IDL maneja la programación estructural, la cual consiste en definir procedimientos y funciones para realizar tareas específicas, la diferencia que existe entre los procedimientos y funciones estriba en que estas últimas regresan un valor. Ambas pueden recibir argumentos y / o *keywords*. Los argumentos son parámetros que se utilizan para la correcta ejecución, mientras que los *kerwords* son parámetros opcionales que no son necesarios para la ejecución.

Los programas estructurales pueden estar compuestos por cualquier cantidad de procedimientos o funciones denominadas rutinas. Todas las rutinas deben estar contenidas en el mismo archivo, que debe tener la extensión PRO, para que sea reconocible por IDL, además, el archivo debe contener una rutina principal que controle el flujo de ejecución de las demás. El nombre de la rutina principal debe ser igual al nombre del archivo que las contiene, si este requisito no se cumple IDL enviará un error durante la ejecución, indicando que no es posible encontrar la rutina principal.

Una excelente practica de programación es el comentar los programas. IDL permite la inclusión de comentarios dentro de los *scripts* o programas estructurados, antecediendo el punto y coma en la línea de comentarios, además, reconoce la ejecución de múltiples instrucciones en una sola línea utilizando el carácter *ampersand* (&) para separarlas. Otro carácter muy útil es el carácter signo de dólar ($), que indica que una instrucción continua en la línea siguiente.

3.2.1. Sentencias de Control.

Como se vio anteriormente, IDL soporta una programación de alto nivel, y como muchos otros lenguajes, otorga una serie de instrucciones de control que ayudan a los programadores a especificar el flujo de ejecución del programa. IDL divide estas sentencias o instrucciones de control en cuatro tipos: instrucciones compuestas, instrucciones condicionales, instrucciones de salto e instrucciones de ciclo.

3.2.1.1. Instrucciones Compuestas.

Las instrucciones compuestas son un grupo de instrucciones apiladas en un bloque de sentencias con la finalidad de ser tratadas como una sola. Los bloques de sentencias son útiles cuando se necesita ejecutar más de una instrucción contenidas en un ciclo o en una instrucción de condición. Para crear un bloque se sentencias IDL provee las instrucciones *BEGIN* y *END*, la primera indica que se iniciará un bloque de sentencias, mientras que la segunda el fin del bloque. La sintaxis de estas dos sentencias es definida como:

```
BEGIN
        Sentencia Uno
            ...
        Sentencia N
END
```

La definición de un bloque debe estar acompañada por alguna de las siguientes instrucciones: *IF, FOR, ELSE, WHILE, REPEAT.* Por ejemplo, para definir un bloque sujeto a un condicional de tipo IF es necesario utilizar la siguiente sintaxis:

```
IF expresión de condición THEN $
BEGIN
        Sentencia Uno
            ...
        Sentencia N
END
```

Para facilitar la tarea del programador, IDL permite añadir sufijos a la instrucción *END* para identificar a que tipo de sentencia de control pertenece ese bloque, de esta manera, el *END* del ejemplo anterior puede ser sustituido por un *ENDIF*. La tabla 3.8 muestra los sufijos válidos para cada uno de las sentencias de control, que se verán más adelante:

Sentencia de Control	Sufijo Valido
ELSE BEGIN	*ENDELSE*
FOR ... DO BEGIN	*ENDFOR*
IF ... THEN BEGIN	*ENDIF*
REPEAT BEGIN	*ENDREP*
WHILE ... DO BEGIN	*ENDWHILE*

Etiqueta: BEGIN	*END*
Expression Case: BEGIN	*END*
Expression Switch: BEGIN	*END*
CASE ... OF BEGIN	*ENDCASE*
SWITCH	*ENDSWITCH*

Tabla 3.8

3.2.1.2. Instrucciones de Condición.

Las instrucciones de condición son muy útiles para controlar las diferentes acciones que pueden ocurrir ante diferentes condiciones, IDL define tres instrucciones de este tipo: *IF*, *CASE* y *SWITCH*.

3.2.1.2.1. IF.

La instrucción *IF* es utilizada cuando sé desea ejecutar una o varias instrucciones si la condición a evaluar es verdadera. La sintaxis básica es:
IF expresión THEN sentencia [ELSE sentencia]

O, utilizando bloques de sentencias:
IF expresión THEN BEGIN
Bloque de sentencias
ENDIF [$
ELSE BEGIN
Bloque de sentencias
ENDELSE]

En ambos casos, cuando la condición principal no es verdadera se ejecuta las instrucciones definidas en el *ELSE*, siempre que se haya incluido, en caso contrario, IDL continúa con la siguiente instrucción definida fuera del bloque *IF*.

3.2.1.2.2. CASE.

La instrucción *CASE* es utilizada cuando se desea ejecutar una sola selección de un conjunto de valores que pueden adquirir una expresión. La sintaxis de esta instrucción es:

```
CASE expresión OF
Valor 1: sentencia
        Valor 2: sentencia
        Valor 3: BEGIN
                Bloque de sentencias
        END
        [ELSE : sentencia]
ENDCASE
```


IDL evalúa cada valor en el orden en que fueron escritos hasta que encuentra el valor correspondiente de la expresión, en caso contrario, es ejecutada la sentencia definida en bloque *ELSE*; si este bloque no fue definido y IDL no encuentra el valor correspondiente a la expresión se produce un error lógico y se detiene la ejecución del programa.

3.2.1.2.3. SWITCH.

La instrucción *SWITCH* tiene la misma lógica que la instrucción *CASE*, dada una expresión, IDL evalúa cada uno de los casos contenidos en esta instrucción hasta encontrar un valor que sea compatible con el valor actual de la expresión. La sintaxis del condicional *SWITCH* es:

```
SWITCH expresión OF
        Valor 1: sentencia
        Valor 2: BEGIN
                Bloque de sentencias
        END
        [ELSE : sentencia]
ENDSWITCH
```

Existen dos diferencias fundamentales entre los condicionales *CASE* y *SWITCH*. La primera diferencia es el bloque *ELSE*, en el caso del condicional *SWITCH*, si ninguno de los casos es compatible con el valor de la expresión y este bloque no está definido no se produce ningún error, a diferencia del condicional *CASE*. La segunda diferencia radica en la forma de ejecución de estas dos condicionales, por ejemplo, el condicional *CASE* sólo ejecuta las sentencias correspondientes al valor actual de la expresión, mientras que el condicional *SWITCH* no solo ejecuta todas las sentencias adjudicada al caso

correspondiente, sino también, aquellas de los casos subsiguientes, el siguiente ejemplo, clarifica esta situación:

x = 2 *CASE x OF* *1: PRINT, x* *2: PRINT, x* *3: PRINT, x* *ELSE: PRINT, x* *ENDCASE* *IDL imprime: 2*	*x = 2* *SWITCH x OF* *1: PRINT, x* *2: PRINT, x* *3: PRINT, x* *ELSE: PRINT, x* *ENDSWITCH* *IDL imprime: 2 3 2*

3.2.1.3. Instrucciones de Ciclo.

Los ciclos o bucles permite ejecutar un determinado número de veces un bloque de sentencias, y aunque IDL está optimizado para manejar arreglos sin la necesidad de utilizar ciclos de repetición, existen situaciones que necesitan manejar este tipo de sentencia de control. Para ello IDL provee de tres formas de crear un ciclo: *FOR*, *REPEAT* y *WHILE*.

La sentencia *FOR* ejecuta una o varias sentencias repetidamente hasta que la condición es cumplida. En cada ciclo de repetición se maneja el incremento o decremento de una variable utilizada para cumplir con la condición, de acuerdo a un monto especificado. La sintaxis de esta instrucción es:
FOR variable = inicio, fin [, incremento] DO sentencia

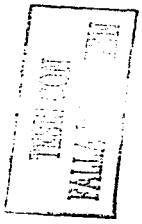
Si se desea incluir un bloque de sentencias, entonces las sintaxis será:
FOR variable = inicio, fin [, incremento] DO BEGIN
Bloque de sentencias
ENDFOR

Si el *incremento* no se especifica, IDL asume que serán incrementos unitarios, además, las variables *inicio* y *fin* deben ser del mismo tipo de dato, de lo contrario, IDL convierte una de las dos variables en el tipo de dato que tenga menor precedencia originando que los resultados no sean los esperados. También es necesario cuidar que el incremento sea distinto de cero, ya que esto originaría un ciclo infinito.

Otro ciclo de repetición lo facilita la sentencia *REPEAT ... UNTIL*, la cual ejecuta una o más sentencias mientras la condición sea falsa. A diferencia de la sentencia *FOR*, esta ejecuta primero las sentencias y después verifica la condición, así, el bloque de sentencias es ejecutado al menos una vez. La sintaxis de esta instrucción es:

REPEAT sentencia UNTIL condición ó, para más de una sentencia:

REPEAT BEGIN
Bloque de sentencias
ENDREP UNTIL condición



Finalmente tenemos a la instrucción *WHILE*, la cual ejecuta una serie de instrucciones repetidamente hasta que la condición deja de ser verdadera. En este caso, *WHILE* verifica la condición antes de ejecutar las instrucciones que le suceden. La sintaxis para una sola sentencia o para un bloque de ellas, es:

WHILE condición DO sentencia ó, para más de una sentencia:

WHILE condición DO BEGIN
Bloque de sentencias
ENDWHILE

3.2.1.4. Instrucciones de Salto.

Este tipo de instrucción puede llegar modificar el comportamiento de un programa, ya que transfieren el control de la ejecución a cualquier parte del mismo. Algunos programadores expertos aconsejan no utilizar este tipo de sentencias bajo la programación estructurada, ya que rompe el espíritu mismo de esta metodología, sin embargo, la mayoría de los lenguajes de alto nivel ofrecen este tipo de herramientas e IDL es uno de ellos; queda, pues, a criterio del programador, el uso de este tipo de instrucciones.

La primera instrucción de salto que ofrece IDL es la sentencia *GOTO*, la cual transfiere el control de la ejecución a otro punto del programa identificado por una etiqueta, dicha etiqueta es un nombre cualquiera compuesto por no más de 15 caracteres alfanuméricos, sucedido por el carácter dos puntos (:). La sintaxis de la instrucción *GOTO* es:

GOTO etiqueta

Otra sentencia disponible para alterar la ejecución de un programa es la instrucción *BREAK*, la cual rompe con los ciclos de repetición o con los bloques de condición y transfiere el control de la ejecución a la línea siguiente. *BREAK* es muy utilizado para resolver el problema que presenta el condicional *SWITCH*.

Por último, se tiene la instrucción *CONTINUE*, la cual obliga a continuar un ciclo de repetición sin importar si se ejecuta o no el bloque de sentencias correspondientes al ciclo.

3.2.2. Procedimientos y Funciones.

Los procedimientos son una secuencia de sentencias IDL agrupadas bajo un nombre, las cuales pueden ser compiladas y guardadas para usos futuros. La definición de un procedimiento tiene la siguiente sintaxis:

```
PRO Nombre_del_procedimiento [, lista de argumentos]
        Bloque de sentencias IDL
        [RETURN]
END
```

La instrucción PRO indica a IDL que se iniciará la definición de un procedimiento y terminará hasta encontrar la correspondiente sentencia *END*. Los procedimientos pueden o no recibir argumentos, en caso de que reciba más de uno, estos deben estar separados por comas. Asimismo, el procedimiento puede llamar, opcionalmente, a la sentencia *RETURN* para regresar el control de la ejecución a la rutina que la llamó. La sintaxis utilizada para invocar un procedimiento es:

```
procedimiento [, parámetro1, parámetro2, ..., parámetroN]
```

Al igual que los procedimientos, las funciones también son un conjunto de instrucciones IDL compiladas y guardadas en algún lugar, pero a diferencia de los procedimientos, las funciones regresan un valor a la rutina que las invocó, y debe ser invocado con al menos un sólo argumento como parámetro. La definición de una función sigue la siguiente sintaxis:

```
FUNCTION Nombre_ de_la_Función, parámetro1 [, parámetro2, ...., parámetroN]
        Bloque de Sentencias
```

RETURN, expresión
END

La sentencia *FUNCTION* indica a IDL que se inicia la definición de una función, finalizada hasta encontrar su respectivo *END*. Los argumentos de las funciones deben ir separados por comas, asimismo, es obligatorio que las funciones regresen un valor de cualquier tipo acompañados por la sentencia *RETURN*. La llamada a una función tiene la siguiente sintaxis:
Expresión = Nombre_de_la_Función (parámetro1 [, parámetro2, ..., parámetroN])

Los parámetros y los *keyword* se dividen dos clases: de entrada y de salida. Los parámetros y *keyword* de entrada son aquellos que tiene un valor inicial que servirá para las operaciones de la rutina. Los parámetros y *keyword* de salida son argumentos que no contienen ningún valor significativo para la ejecución de la rutina, más bien, sirven para almacenar resultados que serán útiles para la rutina que invocó al procedimiento o función.

El número de argumentos que puede recibir un procedimiento o función puede ser variable. Ante tal situación, ¿cómo saber cuántos y cuáles argumentos se han recibido?. Para resolver este problema IDL provee de funciones que otorgan información acerca del número de argumentos y el tipo de los mismos. Estas funciones son: *N_ELEMENTS*(), *SIZE*(), *N_PARAMS*(), *ARG_PRESENT*() y *KEYWORD_SET*(). Las dos primeras funciones ya se explicaron anteriormente.

La función *N_PARAMS*() regresa el número de argumentos que tiene una rutina, este número no incluye argumentos de tipo *keyword*. La función *ARG_PRESENT*() determina si un argumento está presente o no; la función *KEYWORD_SET*() verifica que un argumento de tipo *keyword* haya sido definido como argumento de una rutina.

Aparte, IDL tiene un mecanismo muy particular que utiliza cuando pasa parámetros a una rutina. A diferencia de otros lenguajes, donde los programadores deciden si pasan argumentos por valor o por referencia, IDL ya supone esta situación y por sí solo interpreta el deseo del programador tomando en cuenta el tipo de dato de los argumentos. Antes de entrar en detalles, es necesario dar una breve explicación de lo que significa el paso por valor y el paso por referencia. El paso por valor implica que la rutina hará una copia de

todos los argumentos que reciben y realizarán sus tareas pertinentes con esas copias, cuando la rutina regresa, los valores originales de los argumentos permanecen inalterados. El paso por referencia implica que la rutina trabajará con los valores originales de los argumentos que recibe, y al finalizar su ejecución, tales valores ya han sufrido modificaciones.

Así pues, IDL utiliza cualquiera de los dos métodos anteriores para pasar argumentos a rutinas. Por ejemplo, variables, arreglos, estructuras y escalares, son pasados por referencia, mientras que constantes, índices de arreglos, elementos de estructuras, expresiones y variables de sistema son pasados por valor.

3.2.3. Operaciones de Entrada y Salida.

Existen un gran número de funciones y procedimientos, en IDL, especializados en operaciones de entrada y salida. Estas rutinas están divididas en dos tipos: aquellas que trabajan con archivos y aquellas que trabajan con la entrada y salida estándar.

IDL tiene cuatro procedimientos que se especializan en operaciones de entrada y salida estándar, las cuales son, *PRINT*, *READ*, *READS* y *STRING*(). El procedimiento *PRINT* escribe datos ASCII a la salida estándar de acuerdo a un formato. El procedimiento *READ* lee, de la estrada estándar, datos ASCII formateados, mientras que *READS*, lee una cadena de la entrada estándar. La sintaxis de cada uno de estos procedimientos y funciones se describe a continuación:

PRINT, argumento1 [, argumento2, argumento3, ..., argumentoN, FORMAT = expresión]
READ, argumento1 [, argumento2, argumento3, ..., argumentoN, FORMAT = expresión]
READS, argumento1 [, argumento2, argumento3, ..., argumentoN, FORMAT = expresión]
Variable = STRING(expresión [, FORMAT = expresión])

El argumento *keyword FORMAT* especifica un formato que debe seguir el procedimiento o función para llevar a cabo sus operaciones de entrada o salida. El formato está compuesto por una serie de códigos que indican la forma con la que se debe leer o imprimir los datos. La tabla 3.9 muestra los códigos más utilizados.

Código	Significado
iN.M	Imprime o lee un valor entero con N dígitos y con M espacios en blanco a la derecha del último dígito.
fN.M	Imprime o Lee un valor decimal con N dígitos para la parte entera y M digito para la parte decimal.
dN.M	Imprime o lee un valor decimal de doble precisión con N dígitos para la parte entera y M dígitos para la parte racional.
eN.M	Imprime o Lee un valor decimal con formato exponencial con N dígitos para la parte entera y M dígitos especificando la parte decimal.
aN	Imprime o lee una cadena con N caracteres
Nx	Indica un salto de N posiciones
/	Comienza una nueva línea
S	Suprime una nueva línea aplicable sólo a formatos de salida
:	Termina la salida si no hay más argumentos.

Tabla 3.9

Existen algunas reglas que IDL aplica cuando se utilizan códigos de formato, a decir: los códigos de formato son leídos de izquierda a derecha, además, debe haber un código de formato por cada argumento. Si existen más códigos de formato que valores a imprimir o leer, sólo son utilizados los códigos correspondientes al número de argumentos. En caso contrario, si hay más valores a imprimir o leer que códigos de formatos, se utiliza el último código especificado para dar formatos a los valores o argumentos sobrantes.

En cuanto al manejo de archivos, existen otros tipos de funciones y procedimientos encargados de trabajar con ellos, así pues. IDL maneja dos tipos de archivos, los archivos formateados, que contienen datos ASCII distribuidos en columnas y renglones; y los archivos no formateados, que contienen datos binarios. Cuando un archivo es abierto, IDL le asocia un identificador único llamado LUN (*logical unit number*) con el cual todas las operaciones de E / S deberán referirse a él utilizando este identificador. El LUN es un valor escalar entero que va del rango de –2 a 128, no obstante, no todos los valores pueden ser utilizados, ya que IDL reserva algunos para los archivos *stdout* (salida estándar), *stdin* (entrada estándar) y *stderr* (salida estándar de

error), cuyos LUN asociados son el –2, -1 y 0 respectivamente. Los LUN del 1 al 99 son identificadores que pueden ser usados de manera arbitraria por los usuarios, mientras que, el rango de identificadores del 100 al 128 son administrados por los procedimientos *GET_LUN* y *FREE_LUN*, los cuales reservan y liberan el uso de estos identificadores. Es conveniente trabajar con estos procedimientos en lugar de utilizar un LUN de manera arbitraria, ya que de lo contrario pueden originarse errores. La sintaxis de estos dos procedimientos es:
GET_LUN, identificador
FREE_LUN, identificador

Donde *identificador* es el nombre de la variable donde es almacenado el LUN. Una vez que tenemos un identificador, podemos invocar algunos de los procedimientos de apertura, tales como *OPENR*, que abre un archivo para lectura; *OPENW*, que abre un archivo para escritura; y, *OPENU*, que abre un archivo para lectura y escritura. La sintaxis de estos tres procedimientos es:
OPENR, identificador, ruta del archivo
OPENW, identificador, ruta del archivo
OPENU, identificador, ruta del archivo

Para cerrar un archivo sólo se tiene que utilizar el procedimiento *CLOSE*, como sigue
CLOSE, identificador

Existen algunas funciones útiles que puede ayudar a obtener información acerca de los archivos con los que se trabaja. *FINDFILE*() es una función que encuentra todos los archivos de un directorio cuyo nombre cumpla con algún patrón de búsqueda, si no se especifica dicho patrón, entonces regresa la lista de archivos contenidos en el directorio actual, además, con el *keyword COUNT* podemos saber el número total de archivos encontrados: La sintaxis es:
Resultado = FINDFILE([patrón de búqueda, COUNT = variable])

Otra función que realiza algo similar es *FINDPATH*(), la cual regresa la ruta absoluta de un archivo especificado. La sintaxis es:
Resultado = FINDPATH(archivo)

Por otro lado, existe la función *DIALOG_PICKFILE*() que provee de una pequeña interfaz gráfica que permite navegar entre el sistema de archivos para

buscar y seleccionar un archivo en particular. La función *FSTAT*() regresa una estructura con información acerca de un archivo que está abierto. Por último, la función *EOF*() verifica si el desplazamiento del archivo ha llegado al final del mismo. Las sintaxis correspondiente a estas funciones son:
Resultado = DIALOG_PICKFILE ([FILTER = expresión])
Resultado = FSTAT(identificador)
Resultado = EOF(identificador).

Para escribir o leer datos de un archivo ASCII se tienen los procedimientos *PRINTF* y *READF*, respectivamente. Estos procedimientos tienen las mismas características que los procedimientos *PRINT* y *READ*, pero con la diferencia de que a estos se les especifica el LUN. La sintaxis de estas funciones son:
PRINTF, identificador, argumento1[, argumento2, ..., argumentoN, FORMAT = expresión]
READF, identificador, argumento1[, argumento2, ..., argumentoN, FORMAT = expresión]

Mientras que para la escritura y lectura de archivos binarios tenemos los procedimientos *WRITEU* y *READU*, cuya sintaxis es:
WRITEU, identificador, argumento1[, argumento2, ..., argumentoN]
READU, identificador, argumento1[, argumento2, ..., argumentoN]

Otros procedimientos y funciones útiles para las operaciones con archivos binarios son: *POINT_LUN*, la cual posiciona el desplazamiento (*offset*) del archivo a una posición especificada; *ASSOC*() es una función que asocia un LUN a una variable; *SAVE* es un procedimiento que guarda el estado de las variables especificadas en un archivo con formato XDR; y, *RESTORE*, es un procedimiento que restaura las variables contenidas en un archivo.

3.2.4. Manejo de Errores.

Cuando ocurre un error de lógica durante la ejecución de un procedimiento o función, IDL detiene la ejecución completa del programa y envía mensajes de error correspondientes, sin embargo, existe una rutina que permite cambiar el comportamiento de IDL cuando ocurre un error de lógica. Este procedimiento es *ON_ERROR*, que recibe un valor escalar que le indica el comportamiento a seguir durante situaciones de error. Los valores escalares permitidos para la

función *ON_ERROR* son el 0, que indica la detención total del programa: 1, que indica que regrese al programa principal; 2, que indica que regrese a la rutina que invocó al procedimiento o función que causó el error; y 3, que indica que continúe con la ejecución

Existe otra forma más sofisticada para detectar posibles errores y es interceptándolos en tiempo de ejecución, para ello se tiene la rutina *CATCH*, la cual establece una variable de error y un manejador del error. La sintaxis de esta rutina es:

```
CATCH, variable
IF variable THEN BEGIN
        Bloque de sentencias que manejan el error
ENDIF
```

La instrucción *CATCH* define una variable donde se almacena un código de error, en caso de ocurrir alguno, la secuencia de instrucciones que le suceden son consideradas como el manejador del error. Si éste último no se especifica, IDL tomará como manejador de error a la siguiente línea que le sucede a *CATCH*. Para cancelar un manejador de error sólo se utiliza la instrucción:

```
CATCH, variable, /CANCEL.
```

Otra posible fuente de errores durante la ejecución de un programa ocurre con las operaciones de entrada y salida. En este caso, IDL tiene la función *ON_IOERROR*, la cual establece una etiqueta donde se transferirá el control de la ejecución cuando ocurra un error de E / S. La sintaxis de esta herramienta es:

```
ON_IOERROR, etiqueta
```

La definición de esta etiqueta debe establecerse antes de efectuar cualquier operación de entrada o salida. Dentro de las posibles fuentes de error no se descartan las operaciones matemáticas, donde puede ocurrir desbordamiento de memoria o una división entre cero; cuando ocurre, IDL imprime un mensaje de advertencia y continua con la ejecución. Para evitar situaciones de error de este tipo, se tiene la función *FINITE()*, que busca valores *NaN* (*not a number*) e *Inf* (*infinity*). La función regresa verdadero cuando la expresión que se le pasa como argumento contiene valores de ese tipo. La sintaxis de esta función es:

```
Variable = FINITE (expresión)
```

3.3. Construcción de Interfaces Gráficas.

Otra de las características sobresalientes de IDL es que permite construir interfaces gráficas de usuario a través de *widgets*. Es tan flexible el uso de estos elementos que un programador puede construir GUI con rapidez y una alta funcionalidad. Existen dos maneras de crear una interfaz gráfica de usuario con IDL, la primera es programándola completamente, desde la declaración de los *widgets*, el manejo de sus atributos, así como la orientación que tendrán. La segunda forma es a través del IDL *GUIBuilder*, provisto en el IDLDE, que es una herramienta interactiva al estilo Microsoft Visual Basic que permite crear de interfaces gráficas a partir de una forma básica y arrastrando iconos que representan los controles sobre la forma desde una barra de herramientas. Cada icono representa un *widget*, que tiene propiedades que se pueden modificar a través de las diferentes ventanas de diálogo diseñadas para tal propósito.

El diseño de las interfaces gráficas también puede llevar consigo el manejo de objetos gráficos para un despliegue más real. Es por ello que también se mencionarán algunos de los objetos gráficos más utilizados en la construcción de GUI's.

3.3.1. Widgets.

Los *widgets* o controles son objetos gráficos simples, unidades básicas que componen a una interfaz gráfica, tal como, botones, barras de menú, cajas de texto, listas de selección, etc. IDL maneja tres tipos de *widgets*, estos son los *widgets* primitivos o elementales; los *widgets* compuestos; y las ventanas de diálogo.

3.3.1.1. *Widgets* Elementales.

Cada *widget* tiene asociado una función de creación, la cual puede recibir una serie de argumentos *keywords* y el identificador del *widget* padre. La siguiente tabla muestra los nueve *widgets* elementales, así como la sintaxis de creación de cada uno de ellos.

1. *Widget Base*. Los *widgets* base son los únicos *widgets* que pueden contener otros *widgets*, incluyendo *widgets* del mismo tipo. Todos los *widgets* deben tener un padre o un contenedor, excepto los *widgets base*, que pueden prescindir de esté, en tal caso, se les conoce como *widgets top-level*. Cualquier evento ocurrido en un *widget top-level* afecta a todos los *widgets* contenidos en él. Los *widgets* que contiene otros *widgets* se les conoce como *widgets* padre, mientras que los *widgets* que están contenidos en otro *widget* se les conoce como *widgets* hijos. La relación padre e hijo que se construye se le conoce como jerarquía de *widgets*. La construcción de un *widget* tipo base se hace a través de la llamada a la función *WIDGET_BASE()*, la cual recibe una serie de parámetros keyword que modifican la apariencia y atributos de este *widget*, a cambio regresa un identificador único asociado a este *widget*, que posteriormente puede ser utilizado para cambiar su apariencia usando los procedimientos *WIDGET_CONTROL*, *WIDGET_INFO*, *WIDGET_EVENT*, *XMANAGER* o *XREGISTERED*. La sintaxis de creación es:
   Identificador = WIDGTE_BASE([identificador del padre], [KEYWORDS])

2. *Widget Button*. Los botones pueden ser creados a partir de la llamada a la función *WIDGET_BUTTON()*, la cual regresa un identificador único para este *widget*. Esta función también recibe argumentos keyword para modificar la apariencia de este *widgets*. La sintaxis de creación es:
   Identificador = WIDGET_BUTTON(identificador del padre, [KEYWORDS])

3. *Widget Draw*. La ventana de gráficos puede ser creada por un *widget* de dibujo con el fin de desplegará imágenes, gráficas volúmenes, etc. La función que crea un *widget* de este tipo es *WIDGET_DRAW()*. Regresa un identificador único y recibe argumentos keyword que modifican el aspecto y nivel de detalle de los gráficos desplegados. La sintaxis de creación es:

Identificador = WIDGET_BUTTON(identificador del padre, [KEYWORDS])

4. *Widget Droplist.* Este *widgets* crea una lista de valores desplegable donde el usuario puede elegir uno de ellos. La sintaxis de creación es:
Identificador = WIDGET_DROPLIST(identificador del padre, [KEYWORDS])

5. *Widget Label.* Crea una etiqueta no editable con información. La sintaxis de creación es:
Identificador = WIDGET_LABEL(identificador del padre, [KEYWORDS])

6. *Widget List.* Crea una lista de valores donde el usuario puede elegir elementos de texto posicionando el cursor del ratón sobre el ítem deseado y presionando el botón correspondiente. La acción genera un evento que indica el índice del ítem seleccionado. La sintaxis de creación es:
Identificador = WIDGET_LIST(identificador del padre, [KEYWORDS])

7. *Widget Slider.* Crea una barra de medición con un rango de valores posibles. Básicamente es una barra de desplazamiento donde cada posición indica un valor y puede ser manipulable a través del ratón. La sintaxis de creación es:
Identificador = WIDGET_SLIDER (identificador del padre [, KEYWORDS])

8. *Widget Table.* Este *widget* despliega una tabla matricial de datos y permite la edición de cada uno de sus registros, asimismo, puede tener una o más columnas y renglones que pueden ser manipuladas a través de una barra de desplazamiento. La sintaxis de creación es:
Identificador = WIDGET_TABLE(identificador del padre [, KEYWORDS])

9. *Widget Text.* La creación de cajas de texto es a través de *widgets* de tipo texto, los cuales pueden desplegar texto o leer texto introducido por el usuario. Habitualmente tiene una sola línea, pero modificando sus atributos puede alojar múltiples líneas de texto y, opcionalmente, puede contener barras de desplazamiento. La sintaxis de creación es:
Identificador = WIDGET_TEXT(identificador del padre [, KEYWORDS])

Los argumentos *kewywords* más comunes se enlistan en la tabla 3.10.

Argumento	Definición	Aplicable
EVENT_FUNC, EVENT_PRO	Son argumentos donde se almacena el nombre de la función y del procedimiento que servirán para atender los eventos generados por este *widget*.	Todos los *widgets*
FRAME	Este atributo dibuja un borde alrededor del *widget*, con un ancho especificado.	Todos los *widgets*
FONT	Este atributo es una expresión que indica el tipo, estilo y tamaño de la fuente a utilizar.	Todos los *widgets*
SENSITIVE	Indica si el *widget* permanece activo o inactivo	Todos los *widgets*
UVALUE	Es un valor que el usuario puede asignar arbitrariamente al *widget*, normalmente se le asigna un alias para poder identificarlo después dentro de la función o procedimiento que manejará sus eventos.	Todos los *widgets*
XSIZE, YZISE	Indican el tamaño del *widget* en *pixeles*.	Todos los *widgets*
GROUP_LEADER	Este atributo guarda un identificador de algún *widget* que indica al líder del grupo de *widgets*.	Todos los *widgets*
VALUE	Asigna un valor escalar o un arreglo que contendrá el *widget*.	Todos los *widgets*, excepto el *widget base*
TITLE	Escribe una etiqueta de título que identifica a los *widgets*.	*Base*, *Slider*, *DropList*
MAP	Desoculta un *widget*.	*base*
ALIGN_CENTER, ALIGN_RIGHT, ALIGN_LEFT	Alinea el valor de un *widget* al centro, a la derecha o a la izquierda	Todos los *widgets*, excepto, *draw*.

Tabla 3.10

3.3.1.2. *Widgets* Compuestos.

Los *widgets* compuestos siguen la misma sintaxis que los *widgets* elementales, igualmente poseen los mismo atributos *keyword*, para modificar su apariencia. Los *widgets* compuestos más utilizados son:

1. *CW_ANIMATE*(). Crea un *widget* para manipular secuencias de imágenes.
2. *CW_CLR_INDEX*(). Crea una barra de desplazamiento donde se puede seleccionar un color.
3. *CW_COLORSEL*(). Crea una ventana donde se muestra los mapas de colores disponibles y permite hacer una selección.
4. *CW_RGBSLIDER*(). Crea una serie de tres barras de desplazamiento para modificar cada una de las bandas de color en el modelo RGB, HIS o HSV.
5. *CW_FIELD*(). Permite crear cajas de texto asociadas a etiquetas.
6. *CW_FROM*() Permite la creación de diferentes *widgets* con valores indistintos.
7. *CW_DEFROI*(). Permite realizar una selección de una región en particular de alguna imagen.
8. *CW_ZOOM*(). Crea un porcentaje de escalamiento sobre las imágenes.
9. *CW_ARCBALL*(). Permite la interacción de transformación de coordenadas.
10. *CW_ORIENT*(). Permite modificar las coordenadas de un sistema de manera interactiva.
11. *CW_BGROUP*(). Crea un grupo de botones asociados con características particulares.
12. *CW_FSLIDER*_ () Crea una barra de desplazamiento con valores flotantes.
13. *CW_PDMENU*(). Crea menús del tipo *pulldown*.

3.3.1.3. *Widgets* de Diálogo.

Los *widgets* de diálogo provee de una pequeña interfaz gráfica destinada a realizar pequeñas tareas especificas. Los *widgets* de diálogo son:

1. *DIALOG_PICKFILE*(). Crea una ventana gráfica para navegar entre el sistema de archivos.
2. *DIALOG_MESSAGE*(). Crea una ventana de diálogo para enviar mensajes.
3. *DIALOG_PRINTJOB*(). Es una ventana de diálogo que permite modificar los parámetros de impresión.
4. *DIALOG_PRINTERSETUP*(). Es una ventana de diálogo que permite modificar las propiedades actuales de la impresora.

3.3.2. Procedimientos para la Manipulación de Widgets.

IDL tiene varios procedimientos que permiten modificar el estado de un *widget* u obtener información acerca de ellos. Básicamente se utilizan tres, *WIDGET_CONTROL*, utilizado para realizar, administrar, modificar o destruir los *widgets*; *WIDGET_INFO*(), se utiliza para obtener información a cerca del estado actual de algún *widget*; y, *XMANAGER*, que se encarga de registrar los eventos de los *widgets*.

El procedimiento *WIDGET_CONTROL* puede modificar casi todos los atributos de todos los *widgets* elementales, únicamente presentando el identificador del *widget* en cuestión y la propiedad que se desea modificar, la sintaxis general es:
WIDGET_CONTROL, [identificador del widget], KEYWORDS

Algunos argumentos *keywords* no necesitan de un identificador de *widget* para realizar una acción, tal es el caso de *DEFAULT_FONT*, que especifica la fuente a utilizar por omisión; *HOURGLASS*, que convierte el cursor del ratón en un reloj de arena; *RESET*, que reinicia todos los parámetros del sistema. En cambio otros *keywords* si necesitan de un identificador para poder realizar su tarea encomendada. Alguno de los argumentos *keyword* más utilizados son:

1. *GET_VALUE*, SET_*VALUE*. Obtiene o asigna el valor correspondiente del *widget*, sólo es válido para aquellos *widgets* que tiene el atributo *VALUE*.

2. *GET_UVALUE, SET_UVALUE.* Obtiene o asigna el valor arbitrario asignado por el usuario.
3. *DESTROY.* Destruye un *widget*.
4. *REALIZE.* La definición de *widget* no implica que se despliegue, para ello hay que llamar a la función *WIDGET_CONTROL* con este argumento.
5. *SENSITIVE.* Sensibiliza o insensibiliza el *widget*.
6. *UPDATE.* Actualiza al *widget*.

El segundo procedimiento más importante es *WIDGET_INFO*, ya que nos ayuda a obtener información a cerca del estado actual de los *widgets*. Al igual que el procedimiento anterior, *WIDGET_INFO* tiene argumentos *keyword* para cada uno de los diferentes tipos de *widgets*. La sintaxis de este procedimiento es:
resultado = WIDGET_INFO(identificador del widget, KEYWORD)

Finalmente, existe el procedimiento *XMANAGER*, el cual registra todos los *widgets* con sus funciones o procedimientos que administrarán sus eventos, y crea un ciclo de eventos que mantiene hasta que los *widgets* registrados sean destruidos. Después de haber realizado un *widget* con *WIDGET_CONTROL*, el paso siguiente es registrarlo con *XMANAGER*, cuya sintaxis es:
XMANAGER, [, Nombre, identificador], [KEYWORDS]

Donde el *Nombre* es una cadena que contiene el nombre del procedimiento o función que crea los *widgets*. *Identificador* es el número ID del *widget*. Sin embargo, no es necesario registrar cada *widget* que se crea con *XMANAGER*, sólo basta con registrar el *widget top-level* y con ello, automáticamente, quedarán registrados todos las jerarquías de *widgets*. Los *keywords* más utilizados por *XMANAGER* son:

1. *CLEANUP.* Es una cadena que contiene el nombre del procedimiento que se llamará inmediatamente después de la destrucción del *widget top-level*.
2. *EVENT_HANDLER.* Es una cadena que contiene el nombre del procedimiento que administrará, por omisión, a todos los *widgets* que no se les haya asignado un manejador de eventos al momento de declararlos.

3. *GROUP_LEADER*, Contiene el identificador de un *widget* líder dentro de la jerarquía.

3.4. Librería de Gráficos.

A partir de la versión 5.0 de IDL se añadió la capacidad de la programación orientada a objetos aumentando las potencialidades de este lenguaje. Un objeto en IDL es una clase especial de variable *heap* que tiene métodos y datos con los cuales el usuario tiene un perfecto control sobre éstos. Las clases que IDL ha implementado como parte de su librería de objetos obedece a las propiedades de la metodología orientada a objetos, es decir, se basan en las características de encapsulamiento de datos, polimorfismo, herencia y persistencia. Todos los objetos deben ser manipulados a través de referencias, y aunque el usuario puede definir sus propias clases; IDL cuenta con una serie de clases básicas que pueden ser utilizadas. Es este último aspecto, IDL ha implementado una serie de clases que permiten la creación y manipulación de gráficas.

Para crear un objeto, IDL proporciona la función *OBJ_NEW*(), la cual regresa una referencia de una nueva instancia de la clase que se le especifica. La sintaxis válida para la creación de una referencia a objeto es:
Objeto = OBJ_NEW([nombre de la clase [, KEYWORDS]])

Si no se especifica el nombre de una clase, IDL crea un objeto nulo que debe ser redefinido después, a través del método *INIT*(). Cada clase tiene argumentos *keyword* que modifican el comportamiento y encapsulado de los objetos. Los argumentos *kerword* se pueden utilizar cuando se define el objeto con la función *OBJ_NEW*(), o cuando se redefine un objeto nulo a través el método *INIT*(). Existen otros métodos muy utilizados para modificar u obtener el estado actual del objeto, éstos métodos son *GETPROPERTY* y *SETPROPERTY*

Para destruir un objeto se utiliza el procedimiento *OBJ_DESTROY*, que llama al método *CLEANUP*, inherente a la clase, donde se especifica las últimas instrucciones a seguir para liberar la memoria utilizada por el objeto. La sintaxis de este procedimiento es:

OBJ_DESTROY, objeto [,KEYWORDS]

También existe una función que permite crear un arreglo de objetos nulos, con un máximo de ocho dimensiones. Esta función es *OBJ_ARR*(), donde cada elemento del arreglo es una referencia nula que deberá ser redefinida después para evitar complicaciones. Otras funciones que son útiles para el manejo de objetos son: *OBJ_ISA*(), que es una función que verifica si una referencia de objeto es una instancia o una subclase de otro; *OBJ_VALID*(), que es una función que verifica si la referencia de objeto es válida o es un objeto nulo; y *OBJ_CLASS*(), es una función que obtiene el nombre de la clase a la que el objeto hace referencia.

IDL tiene un operador especial para que los objetos definidos por el usuario puedan hacer referencia a los métodos de sus respectivas clases, este operador es ->, cuya sintaxis es:
Objeto->Método

En algunas ocasiones, las clases heredan de otras clases y es necesario utilizar los métodos de las superclases o clases padre, para ello se utiliza la siguiente sintaxis:
Objeto->Nombre de la clase::Método

Capítulo IV.

Volume Rendering con IDL (MainVolume)

En los capítulos anteriores se ha discutido la técnica del *Volume Rendering*, además, di una breve descripción de IDL como lenguaje de visualización para volúmenes. Bajo este concepto se construyó la aplicación *MainVolume*, la cual es una colección de funciones y procedimientos enfocados a la manipulación de datos volumétricos utilizando las herramientas que proporciona IDL para este propósito. De igual manera, *MV* proporciona una interfaz gráfica para tener una mayor y mejor comunicación con el usuario, además, *MV* puede ser ejecutado en cualquier plataforma que soporte IDL, ya que está desarrollado en lenguaje IDL puro.

Para poder ejecutar la aplicación *MV* necesita tener instalado IDL versión 5.5 o superior en su computadora personal o estación de trabajo. Tomando en cuenta la plataforma donde se encuentre instalado IDL, es necesario ejecutar las siguientes instrucciones.

1. Microsoft Windows. Para ejecutar IDL es necesario presionar el botón de *Inicio* de Windows y seleccionar Programas, después elegir *Research System IDL 5.5* y, por último, elegir IDL. Esta secuencia de comandos abrirá la interfaz de IDLDE. Finalizado el paso anterior, elegir la opción **Open** del menú **File** de la interfaz IDLDE y buscar en el sistema de archivos la ubicación del documento *"MainVolume.pro"*. Una vez abierto en el panel MDI de IDLDE, es necesario compilar el documento con el comando *Ctrl + F5*, o si lo prefiere, elija la opción **Compile** del menú **Run**. Finalmente, para ejecutar la aplicación *MainVolume* sólo debe presionar la tecla *F5* o elegir la opción **Run MainVolume** del menú **Run**.
2. Familia UNIX. Para la ejecución de la aplicación de *MainVolume* es indispensable seguir la siguiente secuencia de comando:
   HOME % idl
   IDL >.compile MainVolume.pro
   IDL > MainVolume
3. MacOS. Para iniciar IDL es necesario presionar dos veces en el icono de esta aplicación, e inmediatamente después aparecerá la interfaz IDLDE; después deberá seguir los mismos pasos descritos para la plataforma Windows.

4.1. MainVolume.

Cuando inicia por primera vez, la aplicación *MainVolume* detecta la versión de IDL que está instalada. En caso de que esta sea menor a la versión requerida, *MV* enviará un mensaje de advertencia sobre esta situación. La razón de este comportamiento es por que *MV* utiliza algunas herramientas que no se encuentran en versiones anteriores a la 5.5 y, por tanto, no funcionará adecuadamente. De igual manera, *MV* busca la ruta absoluta del navegador de Internet, si no la encuentra entonces aparecerá un mensaje de advertencia y se le pedirá que configure manualmente la ubicación del navegador. La interfaz y funcionalidad se detalla en las siguientes secciones.

4.1.1. Barra de Menú.

La barra de menú contiene un acceso directo a varias herramientas de manipulación para los datos volumétricos. Inicialmente varios de estos controles aparecen inhabilitados debido a que no existe un volumen a manipular en la ventana de *render*. Cuando lo haya, todos los controles serán accesibles al usuario. A continuación se explican a detalle cada uno de los controles contenidos en la barra de menú.

4.1.1.1. Menú Archivo.

Como tradicionalmente se utiliza, el menú de archivo tiene opciones para abrir archivos o para salir de la aplicación, y *MV* no es la excepción. El menú de archivo tiene las opciones de abrir, reconstruir y salir. La opción de abrir proporciona una interfaz para que el usuario especifique la ruta de un archivo que contenga un volumen y algunos datos importantes sobre el mismo. La opción de reconstrucción proporciona una interfaz donde el usuario puede especificar la ruta de un archivo con algún número de imágenes con el objetivo de reconstruir el volumen, asimismo, tiene la opción de que *MV* genere los datos a partir de la especificación de algunos parámetros. Por último, la opción de salida proporciona una pequeña ventana de diálogo donde se pide al usuario

la confirmación de salida. A continuación describimos más a fondo los anteriores tópicos.

4.1.1.1.1. Menú Abrir.

Como se mencionó anteriormente, esta opción proporciona una ventana de diálogo donde se especifica la ruta de un archivo con datos volumétricos. La ventana de diálogo está dividida en tres paneles, cada uno de los cuales se habilitan en secuencia. En el primer panel aparece habilitado por omisión y es aquí donde el usuario debe especificar la ruta del archivo presionando el botón que esta al lado de la caja de texto. El evento que produce el botón es una llamada a la función *DIALOG_PICKFILE()*, que es una interfaz que permite navegar en el sistema de archivos para elegir un archivo.

Cuando haya elegido el archivo se habilitará el segundo panel de la ventana de diálogo; en este panel se le pide el formato del archivo seleccionado. Como IDL maneja diferentes funciones especializadas para leer los diferentes formatos, es importante que el usuario haga la elección correcta sobre el formato de su archivo. En este caso *MV* maneja cuatro tipos de formato, que se enlistan a continuación.

1. Binario. Este formato es especialmente para datos volumétricos que no están formateados dentro del archivo. Si el usuario elige esta opción el tercer panel se habilitará con tres cajas de texto para que introduzca las dimensiones del volumen. Si las dimensiones especificadas no corresponden al tipo de dato *LONG*, entonces *MV* enviará un mensaje advirtiendo de tal situación y cancelará todas las operaciones hechas hasta el momento y tendrá que repetirse el procedimiento de apertura desde el principio.

2. XDR (*External Data Representation*). Este es un tipo de formato para datos binarios no formateados que portabiliza dichos datos entre las diversas plataformas, a decir, el manejo de datos binarios es nativo al sistema operativo que lo maneja y en ocasiones estos datos no son compatibles entre sistemas operativos. Los procedimientos de entrada y salida que IDL maneja aceptan el *keyword* XDR para indicar que el archivo tiene un formato binario de acuerdo a ese estándar. Cuando se ha seleccionado este formato, el tercer panel de la ventana de diálogo se habilita y es necesario introducir las dimensiones del arreglo

volumétrico. Al igual que en el formato anterior, *MV* sólo acepta tipo de dato *LONG* para especificar las dimensiones, en cualquier otro caso se envía un error y tendrá que repetirse el procedimiento desde el principio.

3. ASCII. Este formato está reservado a archivos formateados, en este caso *MV* utiliza el *keyword FORMAT* en los procedimientos de entrada para especificar la manera de leer los datos volumétricos contenidos en el archivo; en este caso, por omisión, *MV* lee datos ASCII de tipo entero con un máximo de cuatro dígitos separados entre sí por 4 espacios. Al igual que los anteriores incisos, al elegir esta opción, se habilita el tercer panel de la ventana de diálogo donde tendrá que especificarse las dimensiones de volumen.

4. HDF[1] (*Hierarchical Data Format*). IDL posee un amplio conjunto de funciones y procedimientos dedicados a operaciones de entrada y salida de datos con este formato. Los archivos HDF pueden contener cualquier cantidad de arreglos, incluyendo tridimensionales, dentro de su contenido y cada uno de ellos puede representar cualquier tipo de información. Cuando se elige el formato HDF del segundo panel, *MV* verifica que el archivo seleccionado tenga el formato HDF a través de la función *HDF_ISHDF()*, la cual informa si un archivo tiene tal formato. Si *MV* detecta que el archivo especificado no posee el formato HDF entonces envía un mensaje de error y cancela todas las operaciones realizadas hasta ese momento. En caso contrario, *MV* hace una exploración rápida en el archivo para determinar cuantos datos tiene y el número de dimensiones de cada uno de ellos. Si no encuentra ningún dato o si no encuentra al menos un arreglo volumétrico entonces envía un mensaje de advertencia avisando al usuario sobre esta situación y cancela todas las operaciones realizadas hasta ese momento. Si por el contrario, encuentra uno o más arreglos volumétricos, habilita el tercer panel de la ventana de diálogo y en una lista de selección pone todos los nombres de datos volumétricos encontrados para que el usuario elija uno de ellos.

Una vez que haya completado este proceso y la información sea correcta, el botón **Aceptar** se habilitará. En el momento en que el usuario presione este

[1] Véase el apéndice B para mayor información a cerca de este formato

botón, *MV* iniciará la lectura del archivo aplicando toda la información recabada, no obstante, es probable que ocurran errores de lectura ocasionados por que no se especificaron correctamente las dimensiones del arreglo volumétrico, o por que el archivo no corresponde al formato especificado. En cualquier caso, *MV* enviará un mensaje de error indicando el tipo de error ocurrido. Si *MV* lee correctamente los datos, entonces se crea un objeto gráfico del tipo *IDLGRVOLUME* y se añade al árbol gráfico de la ventana de *render*, habilitando, así, todos los demás controles de la interfaz y de la barra de menú.

4.1.1.1.2. Menú Reconstruir.

El objetivo de este menú es proporcionar al usuario una herramienta para reconstruir un volumen a partir de un conjunto de imágenes o secciones del mismo. Para este propósito IDL tiene la función *RECON3()* la cual genera un volumen de tamaño arbitrario a partir de un conjunto de imágenes y de cierta información adicional que debe proporcionársele. La llamada a este menú genera la aparición de una ventana de diálogo que no es más que una interfaz gráfica para manipular los diferentes parámetros que recibe la función *RECON3()*.

Esta ventana de diálogo está compuesta por tres paneles. Los dos primeros están enfocados para determinar las dimensiones de las imágenes y el número total de ellas. En este sentido, el usuario puede elegir entre leer las imágenes de un archivo o permitir que *MV* genere tales datos. En el primer caso, el usuario deberá especificar la ruta del archivo, las dimensiones de las imágenes y el número total de ellas, si *MV* detecta que los datos son incorrectos se enviarán los mensajes de error correspondientes.

En caso de que el usuario elija que *MV* genere los datos, solamente deberá elegir las dimensiones de las imágenes y el número de las mismas de una lista de selección ya establecida. El tercer panel se compone por una serie de controles *slider* o barras de desplazamiento ordenados en forma de columnas. Los primeros tres controles *slider* de la primera columna sirven para indicar la posición del lente de la cámara con respecto al origen del objeto, siendo que la cámara se encuentra en el origen del sistema coordenado y apunta en dirección a la parte negativa del eje Z, es por ello que el control *slider* que controla esta propiedad sólo acepta valores negativos. Los siguientes tres controles agrupados en la segunda columna indican el monto de rotación que se debe

aplicar al objeto para que sea coherente a la posición de las imágenes. El primer control *slider* de la tercera columna indica el factor de magnificación, es decir, la longitud total en la que el objeto debe aparecer dentro de la imagen. El segundo control indica la distancia del foco del lente de la cámara con respecto al objeto, si este valor se coloca en cero indica que las imágenes son proyecciones paralelas del objeto. Por último, se tiene una lista de selección desplegable: *droplist*, para indicar las coordenadas opuestas de un cubo imaginario donde se localiza el objeto.

Con esta información *MainVolume* genera un volumen con ayuda de la función *RECON3()* añadiéndolo al objeto gráfico *IDLGRVOLUME* y habilitando todos los demás controles de la interfaz principal. Si ocurre un error de lectura de datos o la función *RECON3()* no logra crear el volumen, *MV* enviará los mensajes de error pertinentes.

4.1.1.2. Menú Proyecciones.

Este menú se localiza bajo la opción de **Herramientas** y ofrece una serie de opciones que están enfocadas a proyectar o extraer el volumen sobre una imagen. Todos los submenús asociados a este menú crean ventanas de diálogo para manipular los parámetros de varios procedimientos y funciones que IDL proporciona para este fin.

La primera herramienta de proyección es la de **contorno**, la cual utiliza el procedimiento *SHADE_VOLUME* para producir una lista de vértices y polígonos que describen una superficie de contorno a partir de un volumen y un valor de contorno. Después utiliza la función *POLYSHADE()* para obtener una imagen sombreada de la superficie descrita con uno o más sólidos. La ventana de diálogo de esta herramienta está compuesta por un par de controles *slider* los cuales representan el valor del IsoContorno y de Contorno, los cuales serán tomados en cuenta para generar la lista de vértices y polígonos; la imagen resultante es desplegada en escala de grises en la ventana de *render* que está al lado de estos controles. Además, esta ventana ofrece la posibilidad de guardar la imagen en formato JPEG presionando el botón **Guardar como...**, que invoca a la función *DIALOG_PICKFILE()* para leer la ruta de un archivo. Para salir de esta ventana de diálogo sólo de debe presionar el botón **Salir**.

La segunda herramienta proporcionada por *MainVolume* es la proyección del volumen sobre una imagen utilizando la técnica de *ray casting* a través de la función *VOXEL_PROJ()*. La interfaz de esta función no necesita manipulación de parámetros ya que todos son internos y los calcula *MV* por omisión, únicamente se ofrece la posibilidad de guardar la imagen desplegada en la ventana de *render* con un formato JPEG. Para salir de esta ventana de diálogo sólo se debe presionar el botón **Salir**.

La tercera herramienta es similar a la anterior pero con la diferencia de que esta ofrece una mayor interactividad con el manejo de sus parámetros y utiliza la técnica de los rayos X para realizar la proyección del volumen sobre un plano aunque resulte un poco más lenta. La ventana de diálogo de esta herramienta es una interfaz sencilla que permite manipular los valores de la función *PROJECT_VOL()*, la cual regresa una imagen translúcida del volumen de acuerdo al número de rayos que atraviesan al volumen. La ventana de diálogo cuenta con tres controles *slider* para manipular el número de rayos que atraviesan al volumen en cada uno de los ejes del sistema coordenado, asimismo, permite guardar la imagen en formato JPG.

Por último, se tiene una herramienta muy útil para extraer cortes del volumen en diferentes ángulos, para ello, *MV* se auxilia de la función *EXTRACT_SLICER()* y construye una interfaz que funge como ventana de diálogo en donde se pueden manipular el monto de los ángulos en cada uno de los ejes del sistema coordenado, a través de una conjunto de tres controles *slider*, para calcular el corte sobre el volumen y extraer una imagen en escala de grises y desplegarla sobre la ventana de *render*. Al igual que las anteriores herramientas, esta ofrece la posibilidad de guardar las imágenes en formato JPEG.

MainVolume integra a este menú dos herramientas que IDL ya tiene definidas para la manipulación de volúmenes, las cuales son *XVOLUME* y *SLICER3*. El procedimiento *XVOLUME* crea una interfaz sencilla para la manipulación de volúmenes con más o menos las mismas herramientas que proporciona *MV*, mientras que el procedimiento *SLICER3* es una interfaz gráfica que permite manipular el volumen a través de cortes (*slicers*) de diferentes ángulos y posiciones.

4.1.1.3. Menú Color.

La asignación del color y de la opacidad al volumen es un aspecto importante para el *Volume Rendering*, por ello *MV* implementa varias funciones y herramientas para la manipulación del color. Primeramente, *MV* utiliza el procedimiento *XLOADCT* que provee de una interfaz gráfica para obtener una tabla de color (*look-up table*) a partir de una lista de tablas disponibles para la plataforma donde se halla instalado IDL. Además, permite manipular la función de transferencia de cada una de ellas, la corrección gama, y demás parámetros. Para aplicar la tabla de color seleccionada sobre el volumen sólo hay que presionar el botón **Done** de la interfaz y *MV* calculará las tres bandas del modelo RGB, a partir de la tabla de color elegida, para asignárselas como propiedad al objeto *IDLGRVOLUME* que contiene el árbol de gráficos de la ventana de *render*.

MainVolume también implementa algunas funciones matemáticas continuas para aplicar color al volumen por cada una de las bandas del modelo RGB, así por ejemplo, se tienen varias funciones seno con diferentes frecuencias que se utilizan para determinar el color del volumen. Por omisión, *MV* presenta al modelo volumétrico en una escala de grises. Para aplicar cualquier señal sobre las tres bandas del volumen sólo debe elegir alguna de las opciones que presenta la caja de diálogo del menú de **Otros**. Esta ventana de diálogo está compuesta por tres listas de selección desplegables que representan a cada una de las bandas del modelo RGB y el usuario podrá elegir arbitrariamente alguna de las funciones que ahí se presentan para aplicársela al objeto volumétrico de la ventana de *render*.

4.1.1.4. Menú Opacidad.

Otra característica importante de la manipulación de volúmenes es el manejo de la opacidad, siendo que la opacidad es el nivel de transparencia que tiene asignado cada *voxel*, de esta manera, el objeto gráfico *IDLGRVOLUME*, que proporciona IDL, posee la propiedad *OPACITY_TABLE*, la cual proporciona la posibilidad de asignar diferentes tablas de opacidad al volumen. *MainVolume* explota esta capacidad e implementa, al igual que el menú color, una serie de funciones matemáticas continuas a diferentes frecuencias que se pueden asignar al volumen con resultados muy diferentes entre sí. Por omisión,

la tabla de opacidad del volumen es la rampa lineal, por ello aparece un carácter asterisco antecediendo a esta opción. Este menú ofrece varias funciones para calcular la tabla de opacidad y la elección de alguna de ellas genera automáticamente un proceso de *render* sobre la ventana que contiene al volumen.

En este mismo menú, MV ofrece la posibilidad de desplegar las gráficas de color y opacidad que están actualmente en el volumen. Esta acción se lleva acabo con el menú **Gráficas**.

4.1.1.5. Menú Histograma.

Dentro del menú de **Herramientas** se cuenta con la opción **Histograma...**, que como su nombre los dice, calcula el histograma del volumen. En este caso, *MV* construye una ventana de diálogo con una gráfica, que está dedicada a graficar el histograma. Esta interfaz también ofrece la posibilidad de calcular y aplicar la ecualización del histograma, la cual reemplaza el volumen original por un volumen ecualizado. La siguiente vez que se invoque esta interfaz de histograma, esta opción de ecualización aparecerá deshabilitada y a cambio se habilita la opción para regresar al volumen original. Para realizar todos estos procesos, *MV* se auxilia de las funciones *HISTOGRAM()* e *HISTO_EQUAL()*, las cuales calculan el histograma y su respectiva ecualización de un arreglo mayor a dos dimensiones.

4.1.1.6. Menú Composición.

En el capítulo dos de este trabajo se analizó una de las propiedades del *Volume Rendering* llamada función de composición, la cual es un método que determina el valor de cada *píxel* de la proyección de un volumen sobre una imagen tomando en cuenta los valores de los *voxeles* del volumen acumulados a lo largo de la trayectoria de un rayo imaginario. IDL toma en cuenta esta característica y con su objeto gráfico *IDLGRVOLUME* es posible manipular o asignar esta función a través de la propiedad *COMPOSITE_FUNCTION*. *MainVolume* aprovecha esta facilidad de IDL e implementa en este menú las cuatro funciones disponibles para esta propiedad, las cuales son:

1. Mezcla de Opacidades (*Alpha Blending*). Es una función recursiva que calcula el color del *píxel* de acuerdo a las opacidades de cada *voxel*.
2. MIP (*Maximum Intensity Projection*). En este caso, el color de cada *píxel* de la imagen está determinado por el *voxel* más brillante acumulado en la trayectoria del rayo. Los *voxeles* más opacos son reflectados en la proyección final.
3. Suma de Opacidades (*Alpha Sum*). Esta función es semejante a la mezcla de opacidades pero con la diferencia de que esta no toma en cuenta la tabla de color definida actualmente en el volumen, mientras que la mezcla de opacidades sí lo hace.
4. AIP (*Average Intensity Projection*). Determina el promedio de la brillantez de cada uno de los *voxeles* acumulados por el rayo y determina el color de la imagen final de acuerdo a este valor.

4.1.1.7. Menú Opciones.

El menú de opciones aprovecha la propiedad *HIDE*[2] de cada uno de los objetos gráficos que componen el árbol gráfico de la ventana de *render* para proporcionar al usuario la facilidad de ocultar alguno de estos objetos. Dentro de este mismo menú se encuentra la opción de configuración que sirve para establecer la ruta del navegador de Internet y la ruta del directorio de trabajo actual. Es importante que estas rutas sean correctas, ya que *MV*, a través del procedimiento *SPAWN*, genera un proceso donde invoca al navegador de Internet para ejecutar la página de ayuda Temas.html que debe encontrarse en el mismo directorio donde se ejecuta la aplicación *MainVolume*.

Al principio se mencionó que *MainVolume* busca la ruta absoluta del navegador de Internet, si no la encuentra envía un mensaje de advertencia. Ahora bien, es en esta opción de este menú donde se debe configurar la ruta del navegador, y en su caso, la del directorio actual de trabajo. La interfaz que implementa *MV* consta de dos cajas de texto editables que muestran las rutas por omisión; para cambiar alguna de ellas sólo debe presionar el botón que se encuentra al lado derecho de estas cajas de texto y aparecerá una nueva ventana de diálogo. Esta ventana de diálogo es una especie de explorador del sistema de

[2] El atributo *HIDE* es una propiedad que poseen algunos objetos gráficos que les permite ocultarse.

archivos donde muestra, en dos listas separadas los directorios y archivos contenidos en una determinada ruta.

En esta ventana de explorador podrá navegar a través del sistema de archivos de su computadora o estación de trabajo utilizando los botones **Subir** o **Bajar**, según sea el caso, que se habilitarán cuando haya una selección de algún directorio de la lista correspondiente. Para habilitar el botón **Aceptar** es necesario que exista una selección de algún archivo de la lista que muestra éstos elementos. Cuando haya realizado las configuraciones pertinentes y aceptado los cambios, la ventana de configuración mostrará los cambio hechos a las rutas respectivas. Para finalizar la sesión de esta ventana sólo es necesario presionar el botón **Aceptar** o el botón **Cancelar**; en cualquiera de los dos casos, *MV* mantendrá las rutas de configuración internamente durante toda la sesión, siendo necesario configurar, nuevamente, si se abre otra sesión[3].

4.1.1.8. Menú Ayuda.

En la sección anterior se mencionó que existe el procedimiento *SPAWN*, cuya función es generar un proceso hijo que ejecutará una serie de comandos. La ejecución del proceso padre se ve detenida hasta que el proceso hijo finaliza su ejecución, no obstante, es posible modificar este comportamiento y hacer que ambos procesos se ejecuten de manera paralela usando el *keyword NOWAIT*.

Con esta poderosa herramienta, *MainVolume* engendra un proceso para abrir un archivo HTML que fungirá como archivo de ayuda de esta aplicación. Es importante recalcar que la sintaxis de este procedimiento cambia de acuerdo al sistema operativo de la computadora por ello es de vital importancia que las rutas de configuración del navegador de Internet y del directorio de trabajo sean correctas, de lo contrario, *SPAWN* no podrá generar el proceso hijo, y *MV* enviará un mensaje de error.

[3] Sólo funciona para plataformas Microsoft Windows.

4.1.2. Área de Render.

El área de *render* está compuesto por 3 ventanas: la ventana a la izquierda corresponde a la barra de colores, la ventana central corresponde a la ventana de *render* y la ventana a la derecha corresponde a la barra de opacidad. Estas tres ventanas están compuestas por sus respectivos árboles de objetos gráficos. Estos objetos gráficos son proporcionados por IDL, junto con sus correspondientes propiedades o métodos, para poder ser manipulados para la creación del área de *render*.

En el capítulo anterior, se habló, brevemente, de los *widgets* y de las clases gráficas que IDL ya tiene implementadas. *MainVolume* utiliza estas herramientas para construir un árbol de instancias de estas clases para asignárselas a los *widgets draw*. Estos *widgets* son los únicos que pueden contener, o mejor dicho, dibujar una colección de instancias de clases gráficas ligadas entre si en forma de un árbol jerárquico, modificando los argumentos *keyword RETAIN, RENDERER* y *GRAPHICS_LEVEL*. Asimismo, IDL clasifica sus objetos gráficos en nueve tipos, todos ellos son creados a través de la llamada a la función *OBJ_NEW()*, y la mayoría de los atributos pueden ser modificables con los métodos *SETPROPERTY* y *GETPROPERTY*.

1. Contenedores (*IDL_CONTAINER*). Este tipo de clase permite almacenar una colección completa de objetos gráficos como si fuera un contenedor, sin embargo, las instancias de esta clase no se consideran como parte del árbol gráfico y sólo pueden contener objetos de tipos escena, vista o grupo de vistas. La sintaxis básica de declaración es:
   Objeto = OBJ_NEW ("IDL_CONTAINER")

2. Escenas (*IDLGRSCENE*). Esta clase sirve como contenedor para las clases de tipo vista o grupo de vistas y es considerada como la clase gráfica con mayor jerarquía entre las demás, por tal razón es la clase raíz que puede tener un árbol gráfico. Los métodos *ADD* y *REMOVE* añaden o remueven objetos gráficos menores. La sintaxis es:
   Objeto = OBJ_NEW ("IDLgrSCENE" [, COLOR = índice o vector RGB] [, /HIDE] [, NAME = cadena] [, /TRANSPARENT] [,UVALUE = valor])

3. Grupo de Vistas (*IDLGRVIEWGROUP*). Esta clase sólo puede contener objetos de la clase vista y también pueden fungir como raiz dentro de un árbol de gráficos. La diferencia entre esta clase y la anterior es que *IDLGRSCENE* no puede contener objetos no dibujables, además de que la clase grupo de vistas no limpia su área de dibujo cada vez que se llama el método *DRAW* de los objetos destino. Los métodos más recurrentes de esta clase son *ADD*, *REMOVE*, *SETPROPERTY* y *GETPROPERTY*. La sintaxis básica es:

   Objeto = OBJ_NEW ("IDLgrVIEWGROUP" [, /HIDE] [, NAME = cadena] [, UVALUE = valor])

4. Vistas (*IDLGRVIEW*). Las vistas son los objetos gráficos más utilizados y pueden ser utilizados para contener objetos de tipo modelo. Esta clase también puede fungir como raíz de un árbol gráfico sencillo ya que crea un área rectangular donde serán dibujados todos los objetos modelos que contenga. La sintaxis básica es:

   Objeto = OBJ_NEW ("IDLgrVIEW" [, COLOR = índice o vector RGB] [, DEPTH_CUE = [zbright, zdim]] [, DIMENSIONS = [ancho, alto]] [, /DOUBLE] [, EYE = distancia] [, LOCATION = [x, y]] [, PROJECTION = [1 | 2]] [, /TRANSPARENT] [, UNITS = [0 | 1 | 2 | 3]] [, UVALUE = valor] [, VIEWPLANE_RECT=[x, y, ancho, alto]] [, ZCLIP = [near, far]]).

5. Modelos (*IDLGRMODEL*). Es la única clase que puede contener objetos atómicos y objetos de su mismo tipo. Esta clase incorpora el manejo de transformación de coordenadas a través de matrices que afectan la posición de cada uno de los objetos contenidos en él. Los métodos más utilizados son *ADD*, *DRAW*, *SETPROPERTY* y *GETPROPERTY*. El primero sirve para añadir objeto gráficos atómicos, el segundo método es para dibujar los objetos gráficos contenidos en él; y, el tercero y cuarto, sirven para modificar los atributos de esta clase. La sintaxis básica es:

   Objeto = OBJ_NEW ("IDLgrMODEL" [, /HIDE] [, LIGHTING=[0 | 1 | 2]] [, NAME = cadena] [, /SELECT_TARGET] [, TRANSFORM = matriz de 4x4] [, UVALUE = valor]).

6. Objetos Atómicos. Esta clase de objetos ocupa el último nivel dentro del árbol gráfico, por tanto, no pueden contener a otros objetos. Los objetos atómicos son:

a) *IDLGRAXIS.* **Ayuda a la creación de ejes individuales. La sintaxis básica de creación es:** *Objeto = OBJ_NEW('IDLgrAXIS' [, Dirección] [, COLOR = índice o vector RGB] [, DIRECTION = entero] [, /EXACT] [, /EXTEND] [,/HIDE] [, LOCATION = [x, y] o [x, y, z]] [, /LOG] [, MAJOR = entero] [, MINOR = entero] [, PALETTE = IDLGRPALETTE] [, RANGE = [min, max]] [, TEXTALIGNMENTS = [horiz[0.0 to 1.0], vert[0.0 to 1.0]]] [, TEXTPOS = [0 | 1]] [TITLE = IDLGRFONT] [, XCOORD_CONV = vector] [, YCOORD_CONV = vector] [, ZCOORD_CONV = vector]).*

b) *IDLGRCONTOUR.* **Crea un conjunto de líneas para representar contornos. La sintaxis de creación es:** *Objeto = OBJ_NEW ('IDLgrCONTOUR' [, Valores] [, ANISOTROPY = [x, y, z]] [, C_COLOR = vector] [, C_VALUE = escalar o vector] [, COLOR = índice o vector RGB] [, DATA_VALUES= vector o matriz] [, /DOUBLE_DATA] [, /DOUBLE_GEOM] [, /DOWNHILL] [, /FILL] [, GEOMX = vector o matriz] [, GEOMY = vector o matriz] [, GEOMZ = escalar, vector, o matriz] [, /HIDE] [, MAX_VALUE = valor] [, MIN_VALUE = valor] [, N_LEVELS = valor] [, /PLANAR] [, SHADE_RANGE = [min, max]] [, SHADING=[0 | 1]] [, XCOORD_CONV = vector] [, YCOORD_CONV = vector] [, ZCOORD_CONV = vector]).*

c) *IDLGRIMAGE.* **Esta clase se especializa en graficar arreglos bidimensionales dentro de una vista, básicamente, estos arreglos corresponden a imágenes. La sintaxis de creación es:** *Objeto = OBJ_NEW('IDLgrImage' [, matriz] [, BLEND_FUNCTION = vector] [, CHANNEL = bitmask] [, DIMENSIONS = [ancho, alto]] [, /GREYSCALE] [, /HIDE] [, INTERLEAVE = [0 | 1 | 2]] [, /INTERPOLATE] [LOCATION = [x, y] o [x, y, z]] [, /ORDER] [, XCOORD_CONV = vector] [YCOORD_CONV = vector] [, ZCOORD_CONV = vector]).*

d) *IDLGRLIGHT.* **Esta clase crea una fuente de luz posicionada dentro del sistema coordenado con el objeto de iluminar los demás objetos contenidos en la vista o escena. Las fuentes de luz añadidas en una escena no son visibles por lo que el**

proceso de *render* no las toma en cuenta al momento de realizarse, no obstante, los efectos que causa sobre los demás objetos sí son visibles. La sintaxis de creación es: *Objeto = OBJ_NEW('IDLgrLIGHT' [, COLOR = [R, G, B]] [, CONEANGLE = grados] [, DIRECTION = vector] [, FOCUS = valor] [, /HIDE] [, INTENSITY = valor] [, LOCATION = [x, y, z]] [, TYPE = [0 | 1 | 2 | 3]][, XCOORD_CONV = vector] [, YCOORD_CONV = vector] [, ZCOORD_CONV = vector])*

e) IDLGRPLOT. Esta clase se especializa en mostrar gráficas de valores almacenadas en vectores, opcionalmente, es posible indicarle los valores de cada conjunto en vectores separados. Los ejes desplegados por esta clase no son los mismos de la clase *IDLGRAXIS*. La sintaxis de creación es: *Objeto = OBJ_NEW ("IDLgrPLOT" [, [X,] Y] [, COLOR = índice o vector RGB | , VERT_COLORS = vector] [, DATAS = vector] [, DATAY = vector] [, /DOUBLE] [, /HIDE] [, /HISTOGRAM] [, MAX_VALUE = valor] [, MIN_VALUE = valor] [, XCOORD_CONV = vector] [, XRANGE = [xmin, xmax]] [, YCOORD_CONV = vector] [, YRANGE = [ymin, ymax]] [, ZCOORD_CONV = vector] [, ZVALUE = valor])*

f) IDLGRPOLYGON. Crea polígonos a partir de un conjunto de vértices almacenados en un vector de valores por cada uno de los ejes del sistema coordenado. La sintaxis de creación es: *Objeto = OBJ_NEW ("IDLgrPOLYGON" [, X [, Y[, Z]]] [, COLOR = índice o vector RGB | , VERT_COLORS = vector] [, DATA = arreglo] [, /HIDDEN_LINES] [, /HIDE] [, LINESTYLE = valor] [, POLYGONS = arreglo decribiendo los polígonos] [, SHADING=[0|1]] [, XCOORD_CONV = vector] [, YCOORD_CONV = vector] [, ZCOORD_CONV = vector] [, ZERO_OPACITY_SKIP = [0 | 1]])*

g) IDLGRPOLYLINE. Crea líneas simples a partir de un conjunto de vértices almacenados en un vector de valores por cada uno de los ejes del sistema coordenado. La sintaxis de creación es: *Objeto = OBJ_NEW ("IDLgrPOLYLINE" [, X [, Y[, Z]]] [, COLOR = índice o vector RGB | , VERT_COLORS = vector] [, DATA = arreglo] [, /HIDE] [, LINESTYLE = valor] [, POLYLINES = arreglo*

describiendo las líneas] [, SHADING = {0 | 1}] [, XCOORD_CONV = vector] [, YCOORD_CONV = vector] [, ZCOORD_CONV = vector]).

h) IDLGRSURFACE. **Crea superficies tridimensionales. La sintaxis de creación es:** *Objeto = OBJ_NEW ("IDLgrSURFACE" [, Z [, X, Y]] [, COLOR = índice o vector RGB] [, DATA5 = vector o matriz] [, DATA4 = vector o matriz] [, DATA3 = matriz] [, /DOUBLE{Get, Set}] [, /EXTENDED_LEGO] [, /HIDDEN_LINES] [, /HIDE] [, LINESTYLE = valor] [, MAX_VALUE = valor] [, MIN_VALUE = valor] [, SHADING={0 | 1}] [, STYLE = {0 | 1 | 2 | 3 | 4 | 5 | 6}] [, /USE_TRIANGLES] [, VERT_COLORS = vector] [, XCOORD_CONV = vector] [, YCOORD_CONV = vector] [, ZCOORD_CONV = vector] [, ZERO_OPACITY_SKIP = {0 | 1}]).*

i) IDLGRTEXT. **Otorga la facilidad de crear cadenas de texto y posicionarlas en laguna parte de la escena o vista para aplicarles el proceso de *render*. La sintaxis de creación es:** *Objeto = OBJ_NEW ("IDLgrTEXT" [, cadean o vector de Cadenas] [, ALIGNMENT = valor] [, BASELINE = vector] [, CHAR_DIMENSIONS = [ancho, alto]] [,COLOR = índice o vector RGB] [, FONT = IDLGRFONT] [, /HIDE] [,RECOMPUTE_DIMENSIONS = {0 | 1 | 2}] [, STRINGS = cadena o vector de cadenas] [,XCOORD_CONV = vector] [, YCOORD_CONV = vector] [, ZCOORD_CONV = vector]).*

j) IDLGRVOLUME. **Crea una proyección de un volumen, en una escena o vista, a partir de un conjunto de datos volumétricos. La sintaxis de creación es:** *Objeto = OBJ_NEW ("IDLgrVOLUME" [, volumen0 [, volumen1 [, volumen2 [, volumen3]]]] [, AMBIENT = vector RGB] [, BOUNDS = [xmin, ymin, zmin, xmax, ymax, zmax]] [, COMPOSITE_FUNCTION = {0 | 1 | 2 | 3}] [, CUTTING_PLANES = arreglo] [, DATA0 = [dx, dy, dz]] [, DEPTH_CUE = [zbright, zdim]] [, /HIDE] [, HINTS = {0 | 1 | 2 | 3}] [, /INTERPOLATE] [, /LIGHTING_MODEL] [, OPACITY_TABLE0 = vector] [,RGB_TABLE0 = matriz] [, VOLUME_SELECT={0 | 1 | 2}] [, XCOORD_CONV = vector] [,YCOORD_CONV = vector] [, /ZBUFFER] [, ZCOORD_CONV = vector] [, ZERO_OPACITY_SKIP = {0 | 1}]).*

7. Objetos Ayudantes y de Atributos. Esta clase de objetos no forma parte de ningún nivel dentro de la jerarquía de objetos gráficos, su función más bien, es ayudar al despliegue gráfico de los objetos atómicos ya sea modificando sus atributos o apariencia de éstos; no son dibujables y, por tanto, no son visibles al momento de efectuar un proceso de *render*. Los objetos que pertenecen a esta clase son:

   a) *IDLGRFONT*. Sirve para definir el tipo de fuente a utilizar con objetos de tipo texto. La sintaxis básica es: *Objeto = OBJ_NEW ("IDLgrFONT" [, nombre de la fuente] [, SIZE = puntos] [, SUBSTITUTE = ['Helvetica' | 'Courier' | 'Times' | 'Symbol' | 'Hersliey']] [, THICK = puntos])*.

   b) *IDLGRPALETTE*. Ayuda a definir la tabla de colores a utilizar por algunas de las clases atómicas. La sintaxis básica de declaración es: *Objeto = OBJ_NEW ("IDLgrPALETTE', Rojo, Verde, Azul [, BLUE_VALUES = vector] [, BOTTOM_STRETCH = valor [, GAMMA = valor] [, GREEN_VALUES = vector] [, RED_VALUES = vector] [, TOP_STRETCH = valor])*.

   c) *IDLGRPATTERN*. Define el patrón de llenado para las clases atómicas que utilizan la propiedad *SHADING*. La sintaxis básica es: *Objeto = OBJ_NEW ("IDLgrPATTERN" [, Estilo] [, ORIENTATION = vector] [, SPACING= pixeles] [, STYLE = [0 | 1 | 2]] [, THICK = pixeles])*

   d) *IDLGRSYMBOL*. Define una clase para dibujar cualquier tipo de símbolos. La sintaxis básica es: *Objeto = OBJ_NEW ("IDLgrSYMBOL" [, Datos] [, COLOR = índice o vector RGB] [, DATA = entero o referencia a un objeto] [, SIZE = vector] [, THICK = puntos])*.

   e) *IDLGRTESSELLATOR*. Esta clase descompone un polígono cóncavo en una serie de polígonos convexos usando

triángulos. La sintaxis básica es: *Objeto = OBJ_NEW ("IDLgrTESSELATOR")*.

f) TRACKBALL. Esta clase interpreta los eventos generados por el movimiento del ratón en un *widget draw*, para emular un *trackball* virtual y poder manipular las rotaciones de objetos en escenas o vistas tridimensionales. La sintaxis básica es: *Objeto = OBJ_NEW ("TRACKBALL", Centro, Radio [, AXIS=[0 | 1 | 2]] [, /CONSTRAIN] [, MOUSE = máscara de bits])*.

8. Objetos Destino. Estos objetos no forman parte del árbol de gráficos, sin embargo, son los únicos que contiene el método *DRAW*[4] que permite dibujar todos los objetos atómicos en algún dispositivo de despliegue. IDL tiene clases especializadas para diferentes tipo de despliegue.

   a) IDLGRBUFFER. Esta clase mantiene el árbol gráfico completo en memoria, sin embargo, todos los objetos añadidos, por así decirlo, a esta clase no pueden ser modificados o salvados. La sintaxis de creación es: *Objeto = OBJ_NEW ("IDLgrBUFFER" [, COLOR_MODEL = [0 | 1]] [, DIMENSIONS = [ancho, alto]] [, GRAPHICS_TREE = referencia a un objeto] [, N_COLORS = entero] [, QUALITY = [0 | 1 | 2]] [, RESOLUTION = [resolución x, resolución y]] [, UNITS = [0 | 1 | 2 | 3]])*.

   b) IDLGRCLIPBOARD. Esta clase copia la imagen generada por el árbol gráfico y la transforma a una imagen *raster* nativa a la plataforma, para ser pegada en algún software de imágenes. La sintaxis de creación es: *Objeto = OBJ_NEW ("IDLgrClipboard" [, COLOR_MODEL = [0 | 1]] [, DIMENSIONS = [ancho, alto]] [, GRAPHICS_TREE = referencia a un objeto] [, N_COLORS = entero] [, QUALITY = [0 | 1 | 2]] [, RESOLUTION = [resolución x, resolución y]] [, UNITS = [0 | 1 | 2 | 3]])*.

[4] El método *DRAW* de la clase *IDLgrWINDOW* tiene la función de dibujar todos aquellos objetos atómicos contenidos en la ventana de *render*.

c) *IDLGRPRINTER*. Esta clase envía la imagen generada por el árbol gráfico a la impresora. Los parámetros de impresión pueden ser modificados a través de las funciones *DIALOG_PRINTJOB()* y *DIALOG_PRINTSETUP()*. La sintaxis de creación es: *Objeto = OBJ_NEW ("IDLgrPRINTER" [,COLOR_MODEL = [0 | 1]] [,GRAPHICS_TREE = referencia a un objeto] [,/LANDSCAPE] [, N_COLORS = entero][,N_COPIES = entero] [,PRINT_QUALITY = [0 | 1 | 2]])*.

d) *IDLGRVRML*. Esta clase permite salvar el árbol gráfico en código VRML. La sintaxis de creación es: *Objeto = OBJ_NEW ("IDLgrVRML" [, COLOR_MODEL = [0 | 1]] [, DIMENSIONS = [ancho, alto]] [, FILENAME = ruta del archivo] [, GRAPHICS_TREE = referencia a un objeto] [, N_COLORS = entero] [, QUALITY = [0 | 1 | 2]] [,RESOLUTION = [resolución x, resolución y]] [, WORLDINFO = arreglo de cadenas] [, WOLRDTITLE = cadena])*.

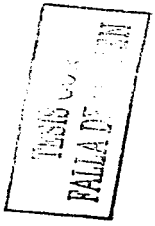
e) *IDLGRWINDOW*. Esta clase permite hacer un despliegue del árbol gráfico sobre una ventana, no obstante, los *widgets draw* pueden simular las propiedades de esta clase y desplegar dicho árbol en su área de dibujo, como se observa a continuación, algunas de los atributos de esta clase son argumentos *keyword* para este tipo de *widget*. La sintaxis básica es: *Objeto = OBJ_NEW ("IDLgrWindow" [, COLOR_MODEL = [0 | 1]] [, DIMENSIONS = [ancho, alto]] [, GRAPHICS_TREE = referencia a un objeto] [, LOCATION = [x, y]] [, N_COLORS = entero] [, QUALITY = [0 | 1 | 2]] [, RENDERER = [0 | 1]] [, RETAIN =[0 | 1 | 2]])*.

9. Objetos para animación. Esta clasificación está constituida por una sola clase, dedicada a la creación de animaciones a partir de *frames* o un conjunto de imágenes. La sintaxis básica es:

Objeto = OBJ_NEW ("IDLgrMPEG" [, BITRATE = valor] [, DIMENSIONS = matriz] [, FILENAME = ruta del archivo] [, FORMAT = [0 | 1]] [, FRAME_RATE = [1 | 2 | 3 | 4 | 5 | 6 | 7 |

S]] [, /INTERLACED] [, MOTION_VEC_LENGTH = (1 | 2 | 3]] [QUALITY = valor] [, SCALE = [escalax, escalay]])

10. Objetos Compuestos. Estas heredan los atributos y propiedades de otras con el fin de facilitar aún más el despliegue gráfico. Los objetos pertenecientes a esta clase son:



a) IDLGRCOLORBAR. Crea un área rectangular para representar una tabla de color, opcionalmente se puede especificar la aparición de ejes o leyendas para este objeto. La sintaxis básica es: *Objeto = OBJ_NEW ("IDLgrCOLORBAR" [, vector, vector, vector] [, BLUE_VALUES = vector] [, COLOR = índice o vector RGB] [, DIMENSIONS = [x, y]] [, GREEN_VALUES = vector] [, /HIDE] [, MAJOR = entero [, MINOR[Get, Set]= entero] [, RED_VALUES = vector] [, SHOW_AXIS = (0 | 1 | 2]] [, TICK = puntos] [, TICKFORMAT = cadena] [, XCOORD_CONV = vector] [, YCOORD_CONV = vector] [, ZCOORD_CONV = vector]).*

b) IDLGRLEGEND. Despliega una leyenda que provee información acerca de los objetos desplegados en una escena. La sintaxis básica es: *Objeto = OBJ_NEW ("IDLgrLegend" [, Items] [, COLUMNS = entero] [, FILL_COLOR = índice o vector RGB] [, FONT = IDLGRFONT] [, GLYPH_WIDTH = valor] [, /HIDE] [, ITEM_COLOR = arreglo de colores] [, ITEM_NAME = arreglo de cadenas] [, /SHOW_FILL] [, TEXT_COLOR = índice o vector RGB] [, XCOORD_CONV = vector] [, YCOORD_CONV = vector] [, ZCOORD_CONV = vector])*

4.1.2.1. Barra de Color y Opacidad.

El árbol gráfico utilizado para la construcción de una barra de color que *MV* implementó en su interfaz es muy sencillo. Como objeto de despliegue se utiliza un *widget draw* con algunas de sus propiedades modificadas para este fin. Después, como contenedor, fue empleado un objeto *IDLGRVIEW* sin proyección hacia el eje Z debido a que el despliegue de la barra de color no

necesita profundidad. Enseguida, se añadió un objeto *IDLGRMODEL* sin modificar sus atributos. Finalmente, la creación de un objeto *IDLGRCOLORBAR* finaliza el árbol de gráficos. Los valores iniciales de este objeto son tres vectores de 256 elementos que representan las tres bandas del modelo RGB con un valor inicial de cero, no obstante, no es visible sino hasta que existe un objeto volumétrico en la ventana de *render*. La configuración del árbol gráfico para la barra de color se ilustra en la figura 4.1.

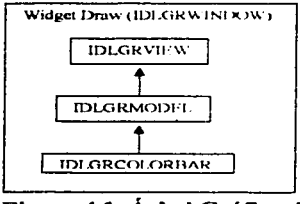


Figura 4.1. Árbol Gráfico de la Barra de Opacidad y Color.

Cuando se crea por primera vez un volumen u ocurre una actualización en la tabla de color, *MV* obtiene las bandas de dicha tabla y las almacena en vectores para asignárselas al objeto *IDLGRCOLORBAR* usando el método *SETPROPERTY*. Además, *MV* determina los valores máximo y mínimo del arreglo volumétrico y se los asigna como atributos *MAJOR* y *MINOR*, respectivamente, del objeto *IDLGRCOLORBAR* y, de acuerdo a ello, la barra de color despliega las divisiones necesarias en la escala correspondiente.

La barra de opacidad, ubicada en el extremo derecho del área de *render*, tiene las mismas particularidades que la barra de color. Cada vez que el usuario hace alguna actualización en la opacidad del volumen, esta barra muestra los cambios de manera inmediata. Asimismo, el árbol jerárquico de objetos gráficos tiene el mismo diseño de la figura 4.1.

4.1.2.2. Ventana de Render.

La configuración de este árbol es un poco más complicada. Nuevamente se utiliza un *widget draw* en vez de un objeto *IDLGRWINDOW* debido a sus propiedades similares, pero en esta ocasión se utiliza un objeto *IDL_CONTAINER* con un doble propósito: primero, para almacenar el árbol gráfico; y, segundo, para contener un objeto *TRACKBALL* cuya función es emular los eventos del ratón e interpretarlos como rotaciones que afectan a todo el árbol. Las dimensiones de acción de este objeto son las mismas que se utilizaron para definir el tamaño del *widget draw*, mientras que el radio de acción es definido como la mitad de la dimensión horizontal.

En ese mismo nivel, se define un objeto *IDLGRVIEW*, que en esta ocasión, tiene asignado el atributo de *EYE* y *PROJECTION* para simular una apariencia de profundidad a todos los objetos atómicos que se añadirán después, asimismo, el color por omisión de la vista es negro. El siguiente nivel del árbol lo ocupa un objeto *IDLGRMODEL*, cuyos atributos no fueron modificados, ni serán modificados por alguno de los eventos ocurridos en la interfaz principal.

El cuarto nivel lo ocupan tres objetos, el primero de ellos corresponde a la clase *IDLGRPOLYLINE*, cuya función es la de crear los ejes globales *X*, *Y* y *Z* del sistema coordenado. Cada uno con un color diferente, así pues, el color rojo representa el eje *X*, el color verde representa el eje *Y*, y el color azul corresponde al eje *Z*. El segundo objeto de este mismo nivel pertenece a la clase *IDLGRMODEL*, cuya matriz de transformación será afectada cada vez que ocurra un evento en el objeto *TRACKBALL* y todos los demás objetos atómicos añadidos a él serán afectados. En este sentido, es importante hacer notar que todas las transformaciones realizadas en este objeto no afectarán de ningún modo a los objetos que están en un nivel superior ni a los objetos que están en su mismo nivel. La última clase, de ese nivel, es la *IDLGRTEXT*, con la cual se crearon tres instancias con el fin de colocar etiquetas a cada uno de los ejes globales.

El nivel más básico lo ocupan aquellos objetos que serán afectados en todas sus propiedades. Para la construcción de este nivel se emplearon dos instancias de la clase *IDLGRLIGHT* con el fin de crear fuentes de luz de color blanca y de tipo *omnilight* para que iluminen la escena ubicadas en los extremos opuestos del eje *Z* global. Para la construcción del cubo, en él cual estará contenido el

volumen se emplearon dos instancias de la clase *IDLGRPOLYLINE*: la primera instancia fue utilizada para construir ejes relativos a la posición del volumen, mientras que la segunda instancia se utilizó para completar la parte restante del cubo. La creación de estos ejes inscritos al cubo obedece a la necesidad de representar, de alguna manera, los índices del arreglo volumétrico, y, en este sentido, fueron creadas cuatro instancias de la clase *IDLGRTEXT* con el fin de poner etiquetas en los extremos de estos ejes inscritos. Al inicio de la aplicación, *MV* pone estas etiquetas con la leyenda "Sin Dimensión", pues no existe un rango tridimensional que defina sus valores. Cuando un volumen es abierto, el valor de estas etiquetas cambia tomando el valor que le corresponde de acuerdo a su posición dentro del arreglo volumétrico.

Finalmente, el objeto más importante para esta aplicación es el de la clase *IDLGRVOLUME*, la cual puede contener hasta 4 volúmenes, obviamente todos deben ser de las mismas dimensiones y del mismo tipo de datos. Esta clase posee una gran cantidad de atributos manipulables que aprovecha *MV* al máximo para otorgar al usuario la flexibilidad necesaria para la manipulación de volúmenes. El árbol gráfico de esta ventana de *render* está representado por la figura 4.2.

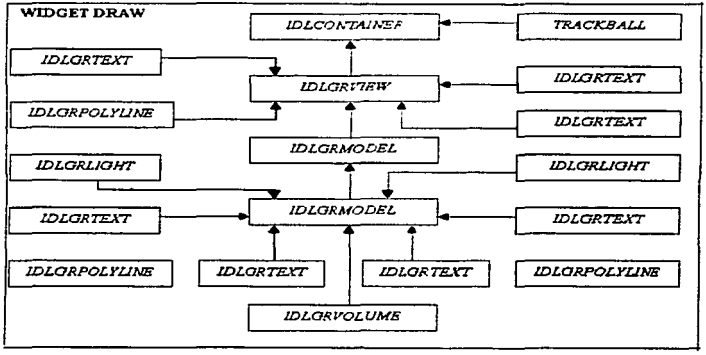


Figura 4.2. Árbol Gráfico del Área de Render.

4.1.2.3. Barra de Estado.

La barra de estado está constituida por un *widget label*, cuya función es informar al usuario, de manera breve, de los procesos internos que realiza *MV* en cada uno de los eventos generados. Asimismo, informa sobre el valor del *voxel* cuando el cursor del ratón de posiciona en una coordenada del área de despliegue principal usando la función *PICK_VOXEL()*, proporcionado como método de la clase *IDLGRVOLUME*.

4.1.3. Área de Control.

El área de control se compone por una serie de *widgets* que permiten la manipulación del volumen. Básicamente se divide en dos módulos: los *widgets* dedicados a la manipulación de planos de corte y los *widgets* dedicados a la transformación de coordenadas.

4.1.3.1. Planos de Corte.

El objeto *IDLGRVOLUME* tiene un atributo llamado *CUTTING_PLANES* cuya finalidad es cortar el volumen en un determinado porcentaje sobre alguno de los ejes del sistema coordenado. Aprovechando esta herramienta, *MV* implementa tres *widgets slider* para otorgar al usuario la facilidad de controlar el porcentaje de corte en cada uno de los ejes. El rango de estos *widgets* fluctúa entre el 0% y el 100%, por omisión, los valores están colocados en la posición más baja. Cada vez que ocurra un evento sobre alguno de estos controles, el botón con la etiqueta **Aplicar**, se habilitará. Si el usuario decide oprimirlo, *MainVolume* obtendrá los valores actuales de cada uno de estos controles y calculará de manera interna el porcentaje de corte que debe aplicársele al volumen usando una matriz de 4x3 para almacenar los valores de corte por cada uno de los ejes.

Posteriormente, usando el método de *SETPROPERTY* de la clase *IDLGRVOLUME* se asigna la matriz al atributo *CUTTING_PLANES* y el

proceso de *render* se encargará de realizar los demás cálculos para realizar una proyección adecuada del volumen con el porcentaje de corte deseado.

4.1.3.2. Transformación de Coordenadas.

Una de las partes esenciales de *MV* es el manejo que otorga al usuario sobre las transformaciones que puede realizar a la escena que, básicamente, son rotación, escalamiento y translación. En la parte central de este módulo hay un panel con tres botones exclusivos, etiquetado con alguna de las transformaciones de coordenadas, por omisión, el botón que corresponde al escalamiento aparece seleccionado. En este mismo panel hay un cuarto botón deshabilitado cuya función es recalcular la posición y la forma inicial del volumen. Cada vez que ocurre un evento sobre cualquiera de los botones exclusivos, *MV* desoculta un panel que contiene más herramientas para la mejor manipulación de las coordenadas.

En el caso del escalamiento, el panel oculto tiene un control de selección desplegable con una lista de porcentajes que van desde el 25% hasta el 200%; estos valores indican el factor de escalamiento que se debe aplicar al volumen. Inmediatamente después de que ocurre un evento sobre esté, *MV* aplica un proceso de *render* sobre la escena para actualizar los cambios.

El panel correspondiente a las operaciones de translación está compuesto por tres *widgets slider*, cada uno de los cuales representan a los tres ejes del sistema coordenado y cuyos valores representan el número de unidades en los que se han divido tales ejes. El rango de translación va desde –10 hasta 10 unidades; por omisión, los valores de estos *widgets* están en ceros, pues el volumen está en el origen, en este aspecto, es necesario hacer hincapié que los cálculos de transformación se realizan tomando como base los ejes principales de la escena y no los ejes inscritos en el cubo. Por otro lado, cualquier evento que ocurra sobre estos *widgets* habilitarán el botón con la etiqueta **Aplicar** que deberá ser presionado para que *MV* realice los cálculos necesarios de translación y actualice la escena mediante un proceso de *render;* cuando haya finalizado lo anterior, el botón será deshabilitado en espera, nuevamente, de que ocurra un evento sobre los *widgets* de translación.

Por último, el panel de rotación está compuesto por un botón de selección etiquetado con la leyenda **Usar Ratón**, seguido por tres *widgets slider* y un botón deshabilitado. Cuando el botón de selección es activado, los *widgets slider*, que le suceden, son deshabilitados y el usuario puede utilizar el cursor del ratón, presionando el botón izquierdo, para rotar libremente el volumen en la escena con ayuda del objeto *TRACKBALL*; para agilizar el proceso de *render*, el volumen se oculta y *MV* dibuja el movimiento de rotación que se efectúa sobre la escena. Una vez que se libera el botón del ratón, la aplicación calcula el monto de rotación que deberá aplicarse al volumen y después aplica el proceso de *render*. A pesar de que este método de rotaciones es más visible y fácil de interpretar, los porcentajes de rotación deseados son difíciles de lograr con exactitud, por tal razón, los *widgets slider* de este mismo panel presenta un modo más confiable de aplicar el monto deseado de rotación con exactitud. Cada uno de estos controles representa a uno de los tres ejes principales y sus valores van desde 0 grados hasta los 360 grados y cualquier evento sobre alguno de estos origina que le botón etiquetado con la leyenda **Visualizar** se active para que *MV* aplique la rotación deseada.

Cualquier evento que ocurra sobre alguno de los controles de los paneles descritos anteriormente origina que el botón con la leyenda **Reinicia Posición** se habilite y cuya función es descartar todos los cambios aplicados en las coordenadas para regresar a la posición inicial de la escena.

Conclusiones

Cuando me interesé por desarrollar esta investigación, yo tenía antecedentes en el campo de la Visualización Científica pues pertenecía al Plan de Becarios de Supercomputo que auspicia la DGSCA, Dirección General de Servicios de Computo Académico, de la UNAM. En este sentido, el Departamento de Visualización de esta dependencia me abrió las puertas con el fin de adquirir conocimientos sobre esta área de estudio, y al mismo tiempo, poner en practicar estas ideas desarrollando sistemas y herramientas para ayudar a los investigadores a resolver sus problemas de despliegue de datos. Básicamente, el desarrollo de estos sistemas se hacían en el entorno de programación de los lenguajes de C o Java, auxiliándonos de algunas librerías gráficas, como OpenGL, para un mejor despliegue de la información; por otro lado, estos lenguajes proporcionan la robustez necesaria para el tipo de desarrollo que se lleva a cabo en ese Departamento.

Paralelamente, en el Instituto de Astronomía de la UNAM, el Doctor Alfredo Santillán tenía la necesidad de contar con una herramienta que le ayudara a manipular datos volumétricos originados de sus investigaciones sobre las galaxias. Asimismo, requería que dicha herramienta estuviera desarrollada totalmente en IDL, ya que para la mayoría de los investigadores de este instituto, este lenguaje de programación es de uso corriente. La experiencia que este investigador tiene sobre IDL la ha adquirido porque ya había desarrollado aplicaciones sencillas para visualizar sus datos en formato HDF, y por otro lado, ya estaba familiarizado con el uso del SLICER3, que es una herramienta estándar que IDL proporciona para el manejo de volúmenes.

Bajo este contexto, la presente tesis se fundamentó en dos objetivos principales. El primero, fue desarrollar una aplicación que permitiera manejar un *Volume Rendering* con calidad satisfactoria proporcionando las herramientas necesarias para lograr esta finalidad; y, el segundo objetivo, es mostrar las potencialidades de IDL como un lenguaje funcional para la Visualización Científica. El día de hoy, que finalizo este tema de estudio, puedo afirmar, categóricamente, que ambos objetivos han sido alcanzados. En cada uno de los capítulos de esta tesis se tocaron conceptos que ayudaron a la construcción de la aplicación final.

Los temas expuestos en el primer capítulo son importantes debido a que todas las aplicaciones construidas para o por la Graficación por Computadora llevan, intrínsecamente, la utilización de estos conceptos. Aún cuando *MainVolume* no utilice de manera explícita alguno de estos conceptos, IDL si

lo hace porque la mayoría de los objetos gráficos son primitivas de dibujo o superficies, de hecho, los objetos *IDLgrPOLYLINE* deben utilizar algún algoritmo, ya sea el de *Bresenham* o el DDA, para el trazo de líneas, mientras que el objeto *IDLgrCONTOUR* debe utilizar algún método para calcular las superficies y construir los contornos.

Quizá el tema con mayor utilidad para el desarrollo de *MainVolume* es el manejo de transformación de coordenadas, pues utiliza los conceptos expuestos en este tema para proporcionar al usuario una herramienta fácil de manejar para manipular rotaciones, escalamientos y translaciones, siendo este módulo parte integral de *MainVolume*. Este módulo de transformación de coordenadas otorga al usuario una interactividad con los datos volumétricos casi en tiempo real. La razón de esta situación es que el proceso de *render* es, por definición, tardado y cada evento generado en la escena principal obliga a recalcular este proceso y dibujar los cambios en la escena.

El segundo capítulo, sin embargo, intenta responder a la pregunta: ¿por qué es necesario un *Volume Rendering*?. Para este propósito es necesario recordar que la Visualización Científica busca la técnica más adecuada para representar, visualmente, un conjunto de datos, siendo el *Volume Rendering* la mejor técnica para representaciones tridimensionales en sistemas de despliegue bidimensionales. Es importante el entendimiento de la forma y metodología de la Visualización Científica, porque con ello sabremos responder a la pregunta inicial de este párrafo. Ante esta situación, cada uno de los temas tratados en este capítulo es de suma importancia ya que son la base para entender e implementar un *Volume Rendering*, independientemente del lenguaje de programación que se utilice.

Por otro lado, al finalizar el tercer capítulo me di cuenta que IDL aún no tiene la robustez necesaria en comparación a Java o C, sin embargo, durante el período de elaboración de este proyecto, *Research Systems* lanzó al mercado la versión 5.6 de IDL, con lo que se demuestra la constante evolución de este lenguaje de programación en el ámbito de la Visualización Científica. Además, como se mencionó en este capítulo, IDL tiene ventajas muy superiores, por ejemplo, C es un lenguaje de programación es de uso general y, por tanto, no está especializado en un área específica como lo es IDL.

Las mejoras que presenta la versión 5.6 son la introducción del manejo de memoria compartida y mapeada, la implementación de nuevas rutinas para la

administración de operaciones de entrada y salida, la posibilidad de un procesamiento a través de hilos (*multi-threads*), mayores opciones para la creación de archivos DLL y compilación de unidades funcionales de programas. En cuanto al manejo de archivos HDF, esta nueva versión proporciona herramientas para "navegar" en archivos con este formato, además, soporta la versión HDF5 y los nuevos formatos ITIFF y XML. Finalmente, las herramientas de visualización mejoraron en cuanto a la introducción de nuevos *widgets* para la creación de interfaces gráficas, mayores opciones para manipular los atributos de los objetos gráficos y de los *widgets* ya existentes. En conclusión, IDL evoluciona rápidamente hacia una especialización, con el fin de resolver los problemas de optimización que presentan algunos otros lenguajes de alto nivel cuando se dedican a implementar aplicaciones para la Visualización Científica.

Considero que el cuarto capítulo es una conclusión en sí, pues en él se describe el funcionamiento de *MainVolume*, asimismo, me siento satisfecho de esta aplicación porque es una herramienta con la capacidad necesaria para realizar un buen *Volume Rendering* proporcionando los instrumentos necesarios para manipular propiedades y otras bondades que facilitan el proceso de esta técnica. No obstante, debo aceptar que *MainVolume* puede mejorarse en varios aspectos, sobre todo en el módulo dedicado al análisis de archivos con formato HDF, pues la versión 5.5 de IDL proporciona una gama de funciones y procedimientos especializados en el manejo de archivos con este formato. En su estado actual, la aplicación puede leer archivos con formato XDR, ASCII, HDF y binario, siendo este último formato, el más especializado en la aplicación final, pues todos los instrumentos que proporciona *MainVolume* están enfocados en el manejo de archivos binarios, empero, esta característica no le resta importancia al manejo de los demás formatos.

Otro de los de los aspectos que pueden introducirse en *MainVolume* es el manejo de múltiples vistas; la posibilidad de manejar uno o más volúmenes en la misma escena de *render*, así como la introducción de una herramienta de *slicer* (corte) para complementar el manejo de los planos de corte. Por último, sería agradable desarrollar a *MainVolume* bajo la metodología de la programación orientada a objetos (POO), ya que IDL, en su versión 5.5, permite este paradigma de programación.

Lo único que resta decir es que estoy satisfecho con los logros de esta tesis y me gustaría seguir con este proyecto para implementarle los requerimientos mencionados anteriormente, sin embargo, el tiempo es el principal adversario y ello obliga a finalizar este tema de tesis con el mayor avance posible.

Apéndice

APÉNDICE A. Código Fuente de MainVolume.

En este apéndice debería ir el código fuente de *MainVolume*, sin embargo, considero que es innecesario imprimir o publicar un código de tal tamaño y magnitud, por ello se anexa un CD con el código fuente de *MainVolume*, además, el CD contiene otras bondades, entre ellas, una página de ayuda en formato HTML. Todos estos archivo contenidos en la carpeta **MainVolume**. En esta misma carpeta existen dos subcarpetas etiquetadas como **Datos** y **Imágenes**. En la primer carpeta el usuario encontrará tres volúmenes binarios, almacenados en diferentes archivos, y 24 volúmenes en formato HDF, almacenados en diferentes archivos. Los volúmenes binarios están almacenados en la ruta **MainVolume/Datos/Binario**.

Para leer los datos binarios contenidos en este directorio es necesario especificar las dimensiones de los arreglos volumétricos, que a continuación se enlistan:

1. Head.dat. Este archivo contiene un volumen binario que representa una cabeza humana. Las dimensiones de este arreglo volumétrico son: *80x100x57*.
2. Hipiph.dat. Este archivo contiene un volumen binario que representa una masa de electrones. Las dimensiones de este arreglo volumétrico son: *64x64x64*.
3. Mri.dat. Este archivo contiene un volumen binario que representa un cerebro humano. Las dimensiones de este arreglo volumétrico son: *138x174x119*.

Para los volúmenes en formato HDF, podrán ser encontrados en el directorio **MainVolume/Datos/HDF**. Además, el CD incluye una serie de imágenes donde se muestra las funcionalidades de *MainVolume*. Este directorio de imágenes tiene la siguiente ruta: **MainVolume/Imágenes**.

Por último en la carpeta **MainVolume/IDL_INSTALLN** se encuentra el *setup* de instalación de IDLDE 5.5 para plataformas WINDOWS y Linux.

APÉNDICE B. Formatos HDF.

Las siglas HDF provienen de las palabras inglesas *Hierarchical Data Format* y de acuerdo con el NCSA (*National Center for Supercomputing Applications*) es un conjunto de librerías y objetos como tipos de datos cuya finalidad es permitir la transferencia de gráficos y conjuntos numéricos entre las diferentes plataformas, a decir, el objetivo principal de este formato es la estandarización. En general, algunas de las características de este formato son:

1. Versatilidad. El formato HDF soporta diferentes modelos de datos y cada uno de ellos define su tipo de datos, así como el conjunto de rutinas para lectura y / o escritura, con ello se evita la necesidad de recurrir a otras aplicaciones para obtener información a cerca de la estructura de los datos. Asimismo, cada modelo de datos puede soportar imágenes tipo *raster*, tablas y arreglos multidimensionales.
2. Portabilidad. Los archivos HDF son portables entre las diversas plataformas existentes, sin importar en cual de ellas se ha creado el archivo, todas las demás podrán manipularlo sin ningún problema.
3. Flexibilidad. La búsqueda de modelos de datos es verdaderamente fácil ya que es posible buscar objetos como elementos individuales o como parte de un grupo.
4. Extensibilidad. Un archivo HDF puede crecer o decrecer de manera dinámica si necesidad de utilizar declaraciones previas.

El formato HDF soporta varios tipos de estructuras de datos, entre ellos están los arreglos multidimensionales (SDS, *Scientific Data Standard*), tablas binarias (*Vdata*), imágenes, texto (*annotations*), paletas de color y grupos (*Vgroups*). Asimismo, cada archivo contiene una librería base, que consiste una interface general con funciones y procedimientos para la administración de memoria, el manejo de E / S, el manejo de errores y el almacenamiento físico; una librería SDS, que implementa el modelo netCDF para soportar múltiples accesos a archivos y modelos de datos; una librería JPEG y una librería GZIP para soportar las diferentes compresiones que manejan las imágenes.

Todas estas características pertenecen al formato HDF en su versión 4.x y anterior. A pesar de los beneficios que aporta esta versión, se ha hecho ineficiente para algunas aplicaciones debido a ciertas características inherentes

a esta versión, por ejemplo, el tamaño máximo de un archivo HDF esta limitado a 2 GB con un máximo de 20 modelos de datos contenidos en él, asimismo estos modelos son rígidos y es necesario transformar cualquier tipo de datos a la estructura que siguen estos modelos.

Ante esta situación varias organizaciones se han dado a la tarea de superar los problemas mencionados anteriormente y lanzar al mercado la versión 5 de HDF. Esta nueva versión incluye una manera más eficiente de manejar las operaciones de entrada y salida para soportar el acceso paralelo a los archivos, permite el manejo de hilos, el crecimiento dinámico del archivo no esta limitado y el modelo de datos se simplifico a solo dos clases de objetos, en lugar de los seis que usaba la anterior versión. El modelo de datos de HDF5 se divide en dos objetos básicos, a decir, los arreglos multidimensionales y los grupos de estructuras. Los primeros se les conoce como *datasets* y los segundos como *groups*. Ambos tienen asociado una lista de atributos, las cuales son definidas por el usuario y tiene la finalidad de proveer información extra sobre el modelo de objeto al que hacen referencia. Los atributos constan de dos partes: el nombre y el valor. En el caso del valor este tiene referencias a varios registros del mismo tipo de dato.

El tipo de dato HDF5 *Group* es una estructura que puede almacenar cero o más objetos de datos en él, y se compone de dos partes: una cabecera que contiene el nombre del grupo y la lista de atributos que hacen referencia a cada uno de los objetos de datos almacenados. La segunda parte es una tabla de símbolos que es una lista de todos los objetos de datos que pertenecen al grupo.

El tipo de dato HDF5 *Dataset* se compone de dos partes: la cabecera y los datos multidimensionales. La cabecera contiene información necesaria para interpretar el arreglo como un metadato o apuntador, asimismo, la cabecera incluye el nombre del arreglo, las dimensiones, el tipo de dato, etc. Toda esta información es almacenada en sus cuatro campos básicos, los cuales son:

1. Nombre (*Name*). Es una secuencia de caracteres alfanuméricos que describen el nombre del arreglo.
2. Tipo de Dato (*DataType*). HDF5 permite definir dos clases de tipos de datos, los datos atómicos y los datos compuestos. El tipo de dato atómico es aquel dato definido en forma nativa por la plataforma, un ejemplo son los números enteros o flotantes, los cuales poseen características propias que varían entre las diversas plataformas. El tipo de datos compuesto es aquel que se define a partir de uno o varios tipos

de datos atómicos, a decir, como si fuera una estructura. Los tipos de datos compuestos pueden tener varios miembros, inclusive otros tipos de datos compuestos.

3. Tamaño del dato (*DataSpace*). Este campo describe las dimensiones del arreglo.
4. Método de almacenamiento (*Storage Layout*). HDF5 tiene varias formas de almacenar los arreglos multidimensionales, aunque en esta versión sólo se cuenta con tres métodos, se espera que próximamente haya más métodos. El primer método es el más sencillo y consiste en el almacenar los datos de manera consecutiva, por eso denomina lineal. El segundo método se denomina compacto y consiste en almacenar los datos en la cabecera siempre y cuando estos no sean muy grandes. Por último, el tercer método consiste en partir el arreglo en trozos de igual tamaño para ser almacenados separadamente, de ahí proviene su nombre. (*Chunk*.

APÉNDICE C. Muestrario de Imágenes.

Las siguientes 14 páginas muestran al usuario una serie de ilustraciones que dan cuenta de las funcionalidades de *MainVolume*. Entre ellas destaca, la asignación de color y opacidad, la aplicación de kernels, el cálculo del histograma, etc.

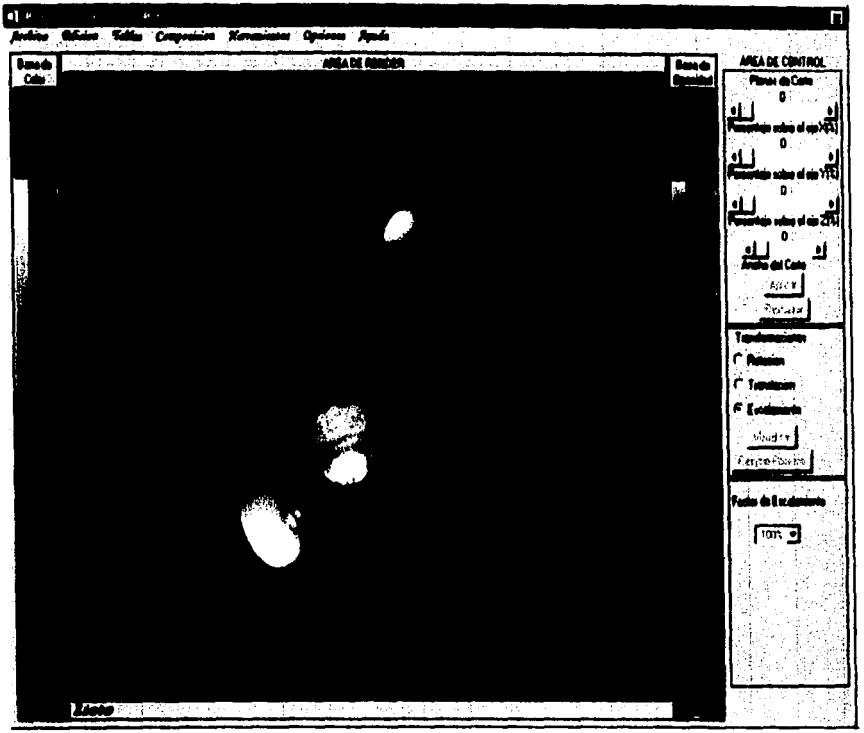


Ilustración 1. Esta ilustración muestra un volumen representando una masa de electrones con una función de opacidad tangencial y una tabla de color que resaltan algunas de sus características.

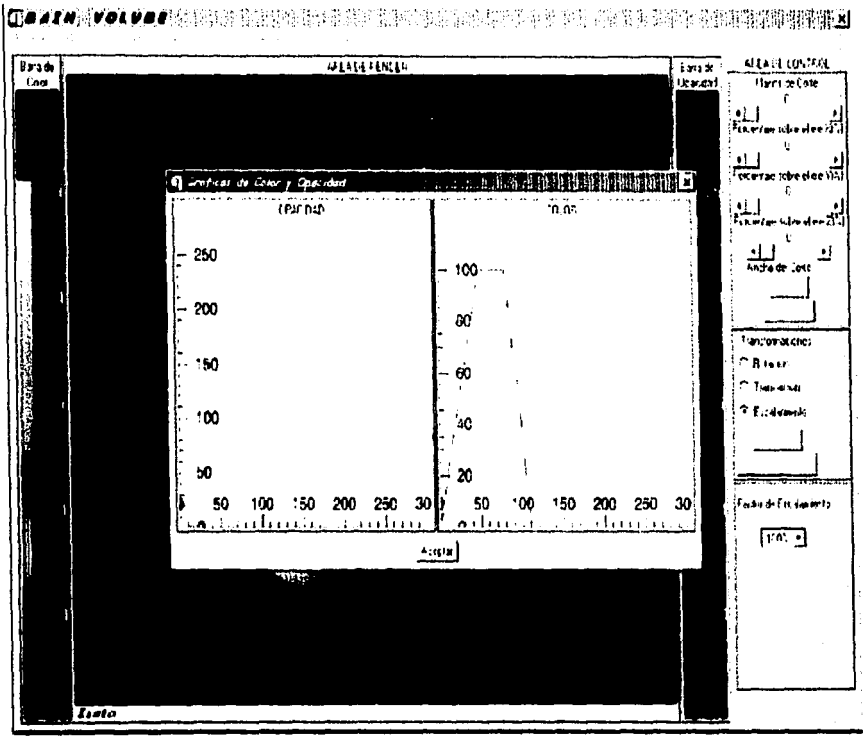


Ilustración 2. Esta imagen muestra las gráficas de las funciones de opacidad y color que fueron aplicadas al volumen del campo de electrones de la ilustración 1.

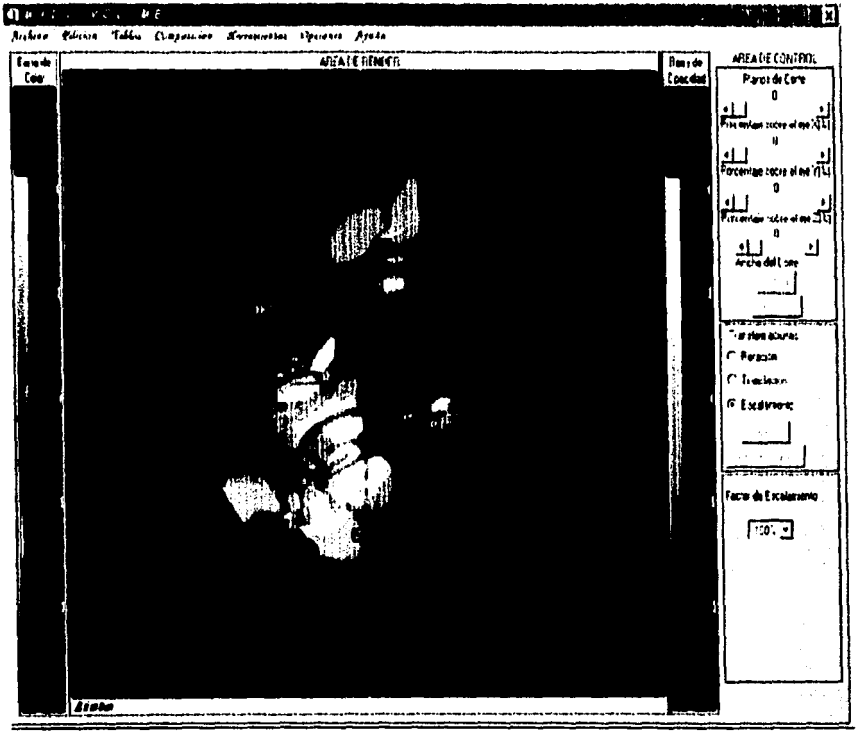


Ilustración 3. Esta ilustración muestra el mismo volumen de masa de electrones pero convolucionada con un kernel de Sobel para tres dimensiones y determinar patrones sobre la dirección X.

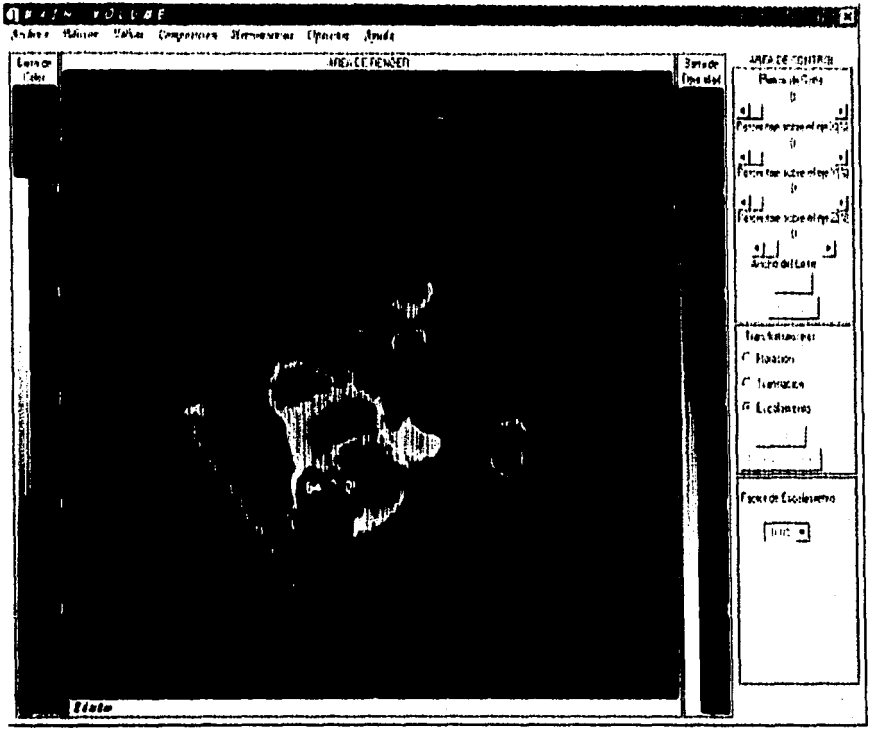


Ilustración 4. Esta ilustración muestra el volumen de la masa de electrones convolucionada con el kernel del Laplaciano y con una opacidad inversa a la rampa lineal, y sin ninguna tabla de color.

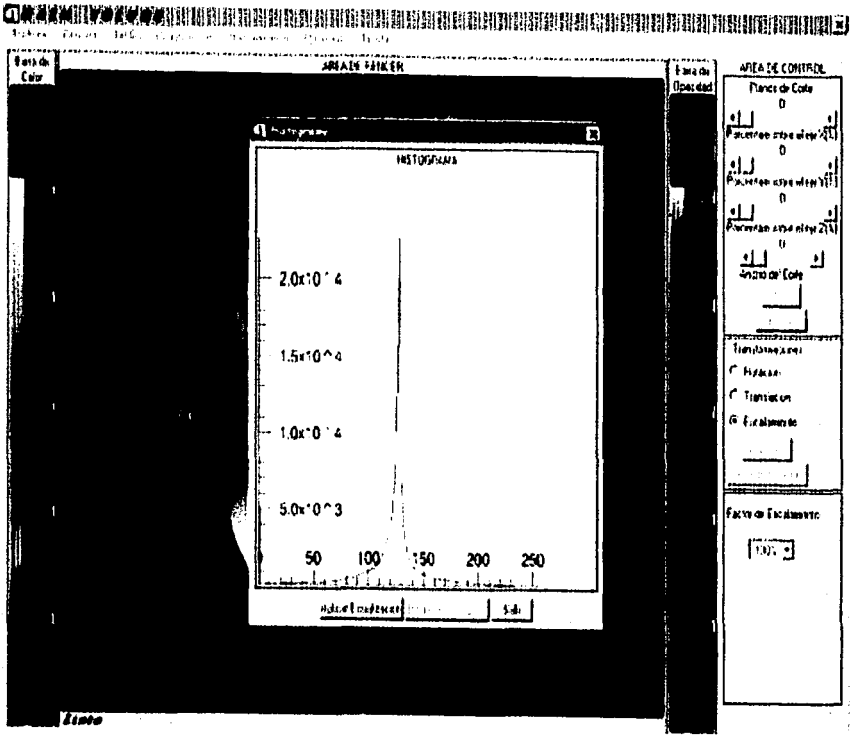


Ilustración 5. Esta ilustración muestra el histograma del volumen de la masa de electrones.

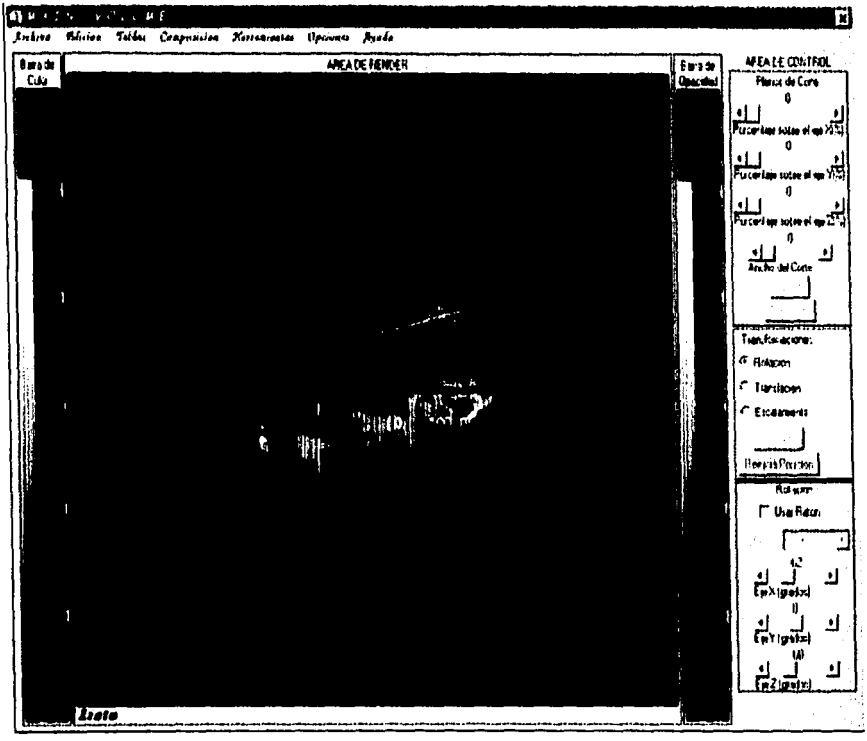


Ilustración 6. Este volumen representa una cabeza humana. En este caso no tiene asignado una tabla de color y la función de opacidad es la rampa lineal.

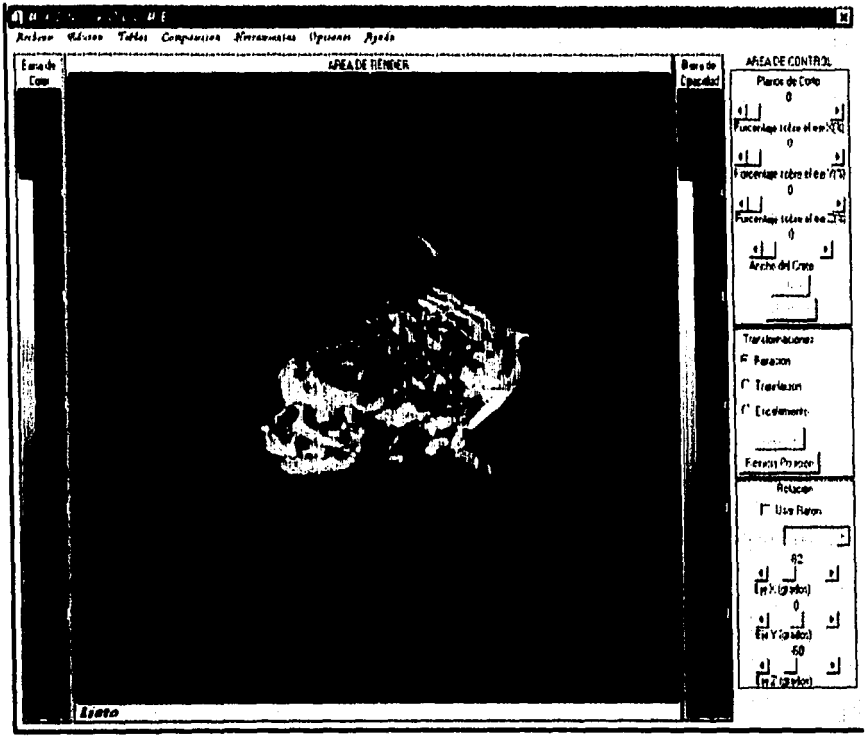


Ilustración 7. Ahora el volumen de la cabeza humana ha sido convolucionado con un kernel de Sobel para tres dimensiones y asiganado una tabla de color. La función de opacidad es la rampa lineal.

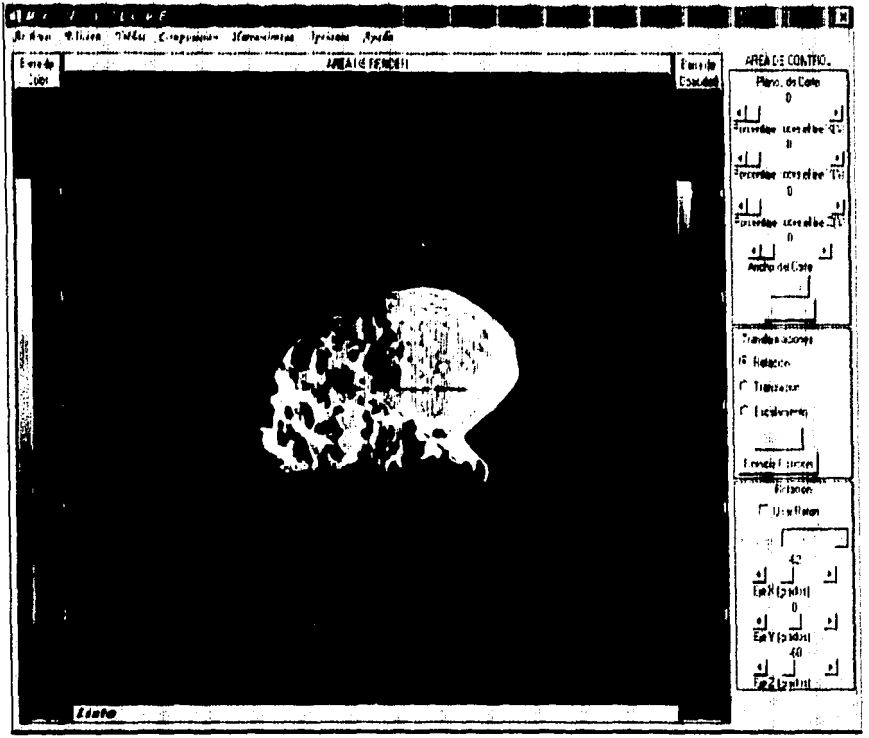


Ilustración 8. Esta ilustración es la misma que la ilustración 7, pero en esta ocasión el volumen de la cabeza humana tiene una función de opacidad tangencial.

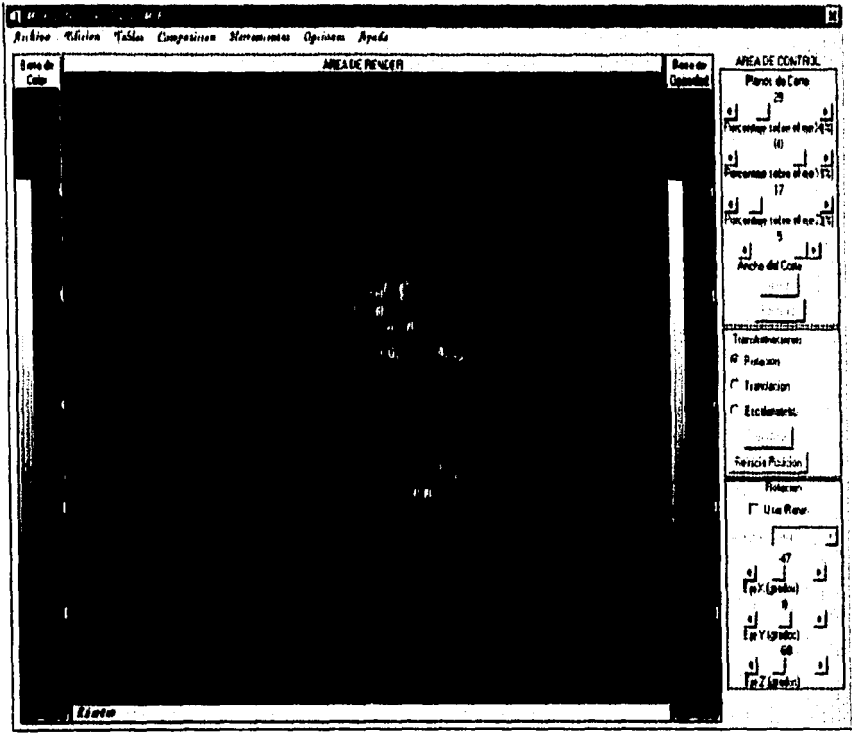


Ilustración 9. Esta ilustración representa un volumen cuyos datos asemejan la forma de un cerebro humano. En este caso, el volumen no tiene asignado ninguna tabla de color y función de opacidad.

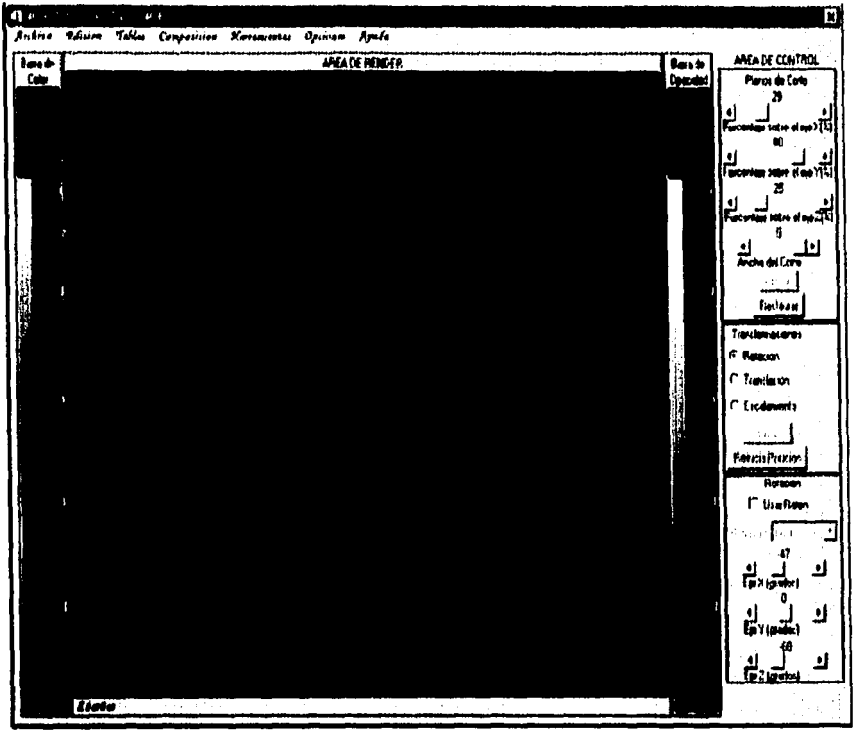


Ilustración 10. Esta ilustración muestra un porcentaje de corte sobre cada uno de los ejes del sistema. Además, el volumen ya tiene asignado una tabla de color que hace más evidente esta situación.

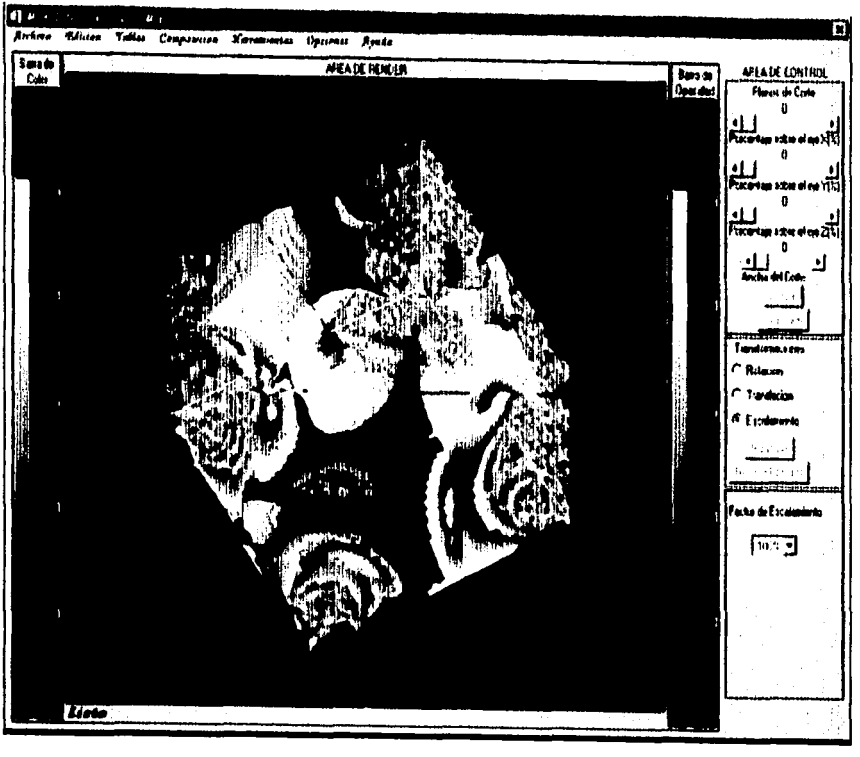


Ilustración 11. Esta ilustración muestra un volumen tomado de un archivo HDF, cuyos valores representan un campo de velocidades de un modelo galáctico cuando ha sido perturbado por diferentes eventos astronómicos.

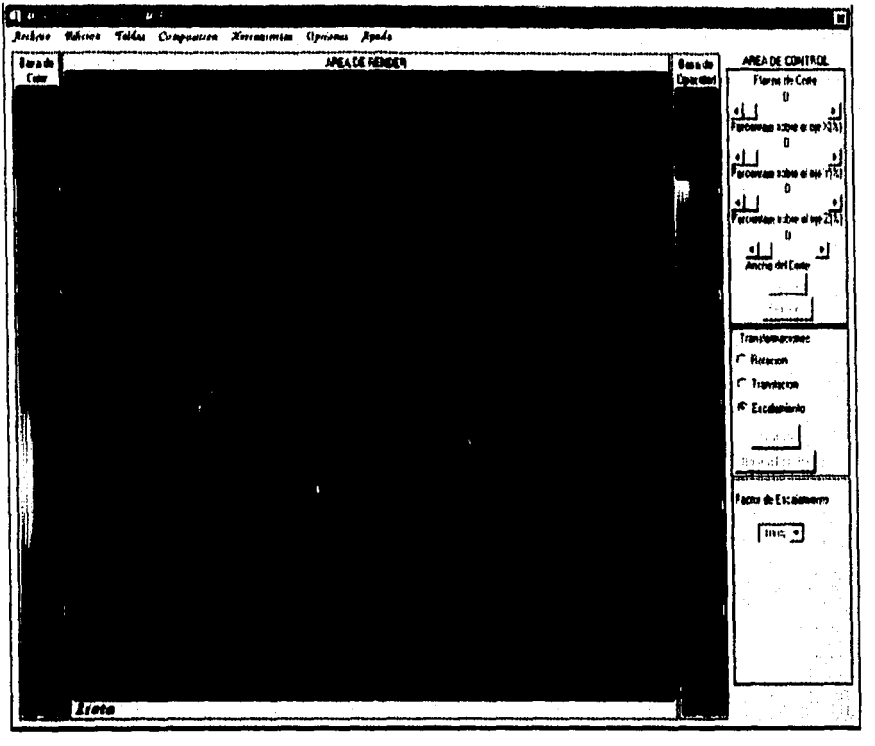


Ilustración 12. Esta ilustración es la misma que la ilustración 11, pero en esta ocasión se ha asignado una tabla de color y una función de opacidad tangencial que hacen más evidente algunas características.

Bibliografía

Articulos Consultados

"Fundamental of Image Processing"
Autor: Ian T. Young, Jan J. Gerbarnds y Lucas J. Vanvliet

"Introducción al Procesamiento Digital de Imágenes"
Autor: Universidad Politécnica de Madrid

"Introducción al Procesamiento de Imágenes Digitales"
Autor: Beatriz Elena Alzate A.

"Mejoras al Algoritmo de Marching Cubes"
Autor: Andrea Silvetti, Claudio Delrieux y Silvia Castro

"Modeling and Animation"
Autor: Tom Ellman

"Ruido en Imágenes"
Autor Rafael Molina

"Visulización"
Autor: Universidad Nacional del Sur

"X-Ray Casting:Fast Volume Visualization using 2D Texture Mapping Techniques"
Autor : Youngser Park, Rpbert W. Lindenman y James K. Hahn

Direcciones de Internet

http://academic.mu.edu/phys/matthysd/web226/index.htm
http://fciencias.unam.mx/graf/graf.htm
http://gbdi.icmc.sc.usp.br/documentacao/apostilas/cg/index.html
http://iwia.sis.epn.edu.ec/~elascano/sistemasmultimediales/cg
http://java.sun.com/people/jag/SimulaHistory.html
http://journal.info.unlp.edu.ar/cacic2000/Visualizacion.html
http://kaka.cosc.cantebury.ac.nz/~wolfgang/cosc25/smalltalk1.html
http://mailweb.udlap.mx/~tesis/lis/cardona_a_jf/capiyulo4.html
http://tigre.aragon.unam.mx/graficas/glosario
http://wgpi.tsc.avigo.es/libro/tecnolog/tecnolog.html
http://www.buap.mx/~yakelin/graf1.html
http://www.buap.mx/~yakelin/graf2.html
http://www.cc.gatech.edu/scivis
http://www.cegs.itesm.mx/ventana/ligas/nlaboratoriom23.htm
http://www.cs.buap.mx/~yakelin/
http://www.cs.mtu.edu/~shene/COURSES/cs3621/NOTES
http://www.dc.uba.ar/people/materias/ec/VC
http://www.etereaestudio.com.docs/studio_index.htm
http://www.etsimo.uniovi.es/mieres/egi/doo
http://www.filosofia.org/enc/ros/espa.htm
http://www.geocities.com/cidepaz/cores/default.htm
http://www.iescuravalera.org/grimaldos/imagendigital/node4.html
http://www.ii.uam.es/~pedro/graficos/teoria/index.html
http://www.iwc.uni-heidelberg.de/groups/ngg/VolumeRendering
http://www.labvis.unam.mx
http://www.nakl.t.u-tokyo.ac.jp/~furu/NURBS
http://www.siggraph.org/education/materials/concepts/concepts.htm
http://www.siggraph.co education/materials/HyperVis/vistoc.htm
http://www.siggraph.or lucation/materiasl/HyperVis/vistech/volume
http://www.sun.com/960 0/feature3
http://www.txemaweb.com/xol/siglas/siglas-r.htm
http://www.vrvis.at/vis/resources
http://www.wpi.edu

Libros consultados

BARTELS, Richard H.
An Introduction to Splines for Use in Computer Graphics and Geometric Modelling.
Los altos California, E.U.A., 1987.
Ed. M. Kaufmann, 476 páginas.

BLACKLEDGE, Jonathan M.
Quantitative Coherent Imaging.

BRODLY, K.W.
Scientific Visualization.

COLIN, Ware.
Information Visualization: Perception for Design.
San Diego, E.U.A., 2000.
Ed. Academic Press, 438 páginas.

CROW, Frankiln C.
Computer Graphics. Techniques, Theory and Practice.

EARNSHAW, Ray.
An Introductory Guide to Scientific Visualization.
Berlin, Alemania, 1992.
Ed. Springer, 150 páginas.

FARIN, Gerald E.
Curves and Surfaces for Computer Aided Geometric Design.
Boston, E.U.A., 1988.
Ed. Academic Press, 334 páginas.

FOLEY, James D.
Fundamentals of Interactive Computer Graphics.
Massacussets, E.U.A., 1982.
Ed. Addison-Wesley, 664 páginas.

GALLAGHER, R.
Computer Visualization: Graphics Techniques for Sc. And Eng. Analysis.

GONZALES, Rafael C.
Digital Image Processing.
Massachussets, E.U.A., 1991.
Ed. Addison-Wesley, 716 páginas.

HAGEN, Hans.
Focus in Scientific Visualization.

HEARN, Donald.
Gráficas por Computadora.
México, D.F., 1989.
Ed. Prentice Hall, 300 páginas.

HILL, Francis S.
Computer Graphics using OpenGL.
New York, E.U.A., 1990.
Ed. McMillan, 754 páginas.

HOWARD, Eves.
Estudio de la Geometrías.

JACOBS R., Harold.
Geometry.

LICHTENBELT, Barthold.
Introduction to Volume Rendering.
New Jersey U.S.A., 1998.
Ed. Prentice Hall, 236 páginas.

LINDLEY, Craig A.
Practical Image Processing in C.
New York U.S.A., 1991.
Ed. J. Wiley, 548 páginas.

MOTENSOR, Michael E.
Computer Graphics. An Introduction to the mathematics and Geometry.
New York U.S.A., 1989.
Ed. Industrial, 373 páginas.

MOTENSOR, Michael E.
Geometry Modeling.

MORRISON, Mike.
The magic of Computer Graphics.
Indianapolis U.S.A., 1995.
Ed. SAMS, 474 páginas.

NIELSON, Gregory M.
Visualization in Scientific Computing.
California U.S.A., 1997.
Ed. IEEE, 577 páginas.

NEWMAN, William.
Principles of Interactive Computer Graphics.
México D.F., 1976.
Ed. McGraw Hill, 607 páginas.

SALMON, Rod.
Computer Graphics, Systems and Concepts.
Workingham, Inglaterra, 1987.
Ed. Addison-Wesley, 702 páginas.

STAR J.
Introduction to Image Processing.

WATT, Alan.
3D Computer Graphics.
Harlow, Inglaterra, 2000.
Ed. Addison-Wesley, 570 páginas.