VNIVERSIDAD NACIONAL AVFNOMA DE MEXICO

UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE ESTUDIOS SUPERIORES ARAGÓN

"ANÁLISIS E IMPLEMENTACIÓN DE ALGORITMOS PARA MODELADO TRIDIMENSIONAL DE OBJETOS ORGANICOS"

T E S I S

QUE PARA OBTENER EL TÍTULO DE:

INGENIERO EN COMPUTACIÓN

P R E S E N T A:

JOSÉ ARTURO HERNÁNDEZ SÁNCHEZ

ASESOR DE TESIS:

M.EN C. MARCELO PEREZ MEDEL

MEXICO, 2005

Agradecimientos:

A mis padres:

por su paciencia infinita y apoyo incondicional.

A Marcelo Pérez Medel:

por creer en mí desde el principio, y hacerme participe de éste proyecto. Gracias.

Ing. Jesús Hernández Cabrera:

por sus consejos, amistad y por compartir retos. Gracias.

A mis sinodales:

M. en I. Liliana Hernández Cervantes.
Mat. Luis Ramírez Flores.
M. en C. Martín Solís Pérez.

*A la **FES Aragón** por ser el instrumento de mi superación.*

*A la **Universidad Nacional Autónoma de México** por alimentar mi alma y fortalecer mi espíritu.*

Dedicatorias:

A mis padres ***Marcela Sánchez Vásquez*** *e* ***Irineo Hernández Mendoza*** *por estar a mi lado en momentos difíciles, y enseñarme a enfrentar la vida con valentía. De ustedes obtengo la fuerza para superarme y crecer como persona todos los días. Gracias por darme la vida.*

A María del Carmen, Víctor Manuel y Marco Antonio, por estar a mi lado y tener fe en mí. Los quiero mucho.

A mis sobrinos que son la semilla del futuro. Sus sonrisas me hacen tener esperanza.

A Arcelia Bernal Díaz por ayudarme en todo y hacerme ver la vida de una manera diferente. Gracias por hacerme feliz.

A Marcelo Pérez Medel por todo el apoyo otorgado durante el desarrollo de este proyecto. El tiempo que hemos trabajado juntos ha valido la pena.

A todos mis amigos que directa o indirectamente me ayudaron a hacer este trabajo.

INDICE

INTRODUCCIÓN

Alrededor del año de 2001 se inicio un proyecto orientado a desarrollar software de modelado tridimensional y animación denominado ***Material 3D*** desarrollado por M. en C. Marcelo Pérez Medel. El prototipo cuenta con herramientas de modelado a base de primitivas por lo que se requería enriquecer este proyecto con herramientas adicionales que permitieran incrementar su potencial. Así es como se comenzaron a integrar algunos proyectos de desarrollo como fueron un editor de texturas, editor de trayectorias, etc.

Sin embargo el diseño de formas a base de primitivas produce necesariamente modelos muy geométricos, por lo cual era necesario analizar que altemativas nos proporcionarían una mayor libertad para modelar objetos con una alta complejidad de forma.

La necesidad de modelar objetos gráficos con alta diversidad de forma nos llevo a contemplar a los objetos orgánicos como una alternativa de diseño. Sin embargo las herramientas y métodos que permitían crear dichos modelos eran muy diversos entre sí. Por lo cual se eligió, como una primera aproximación, a dos de las herramientas que se pueden considerar como representativas de este campo del diseño tridimensional: las superficies paramétricas basadas en ***splines*** y las ***isosuperficies*** basadas en campos escalares.

Para abordar estas dos herramientas se necesitaba en primer lugar entender los algoritmos que le dan origen y forma. Por lo cual el primer paso de esta tesis es analizar cada método y entender como se desarrollan dichos algoritmos.

Una vez analizados y entendidos es necesario encontrar un camino viable el cual nos llevara a plasmar tales algoritmos en software accesible y de fácil manipulación.

Sin embargo nos encontramos con la impresión que desarrollar tales herramientas sólo es tarea destinada a grandes grupos de desarrollo, con acceso a grades recursos de computo y económicos, por lo tanto estos esfuerzos de desarrollo terminan en software muy robusto y caro. Sin embargo a partir del inicio del movimiento del desarrollo software libre, que ha ido casi de la mano del desarrollo y expansión del sistema operativo GNU Linux, ha surgido una corriente, cada vez más creciente, de desarrollar

utilerías de libre uso (y algunas de libre distribución de código) para una amplia gama de tareas, desde herramientas de administración del Sistema Operativo, herramientas para uso de Bases de Datos, juegos, hasta herramientas de gráficos y software de diseño. Parte de la expansión de esta corriente ha sido posible gracias al desarrollo de bibliotecas de programación de libre uso y distribución como lo es ***GTK+***.

Utilizando esta plataforma, Linux con Bibliotecas de uso libre, es posible desarrollar software funcional para el modelado tridimensional de formas orgánicas. El presente trabajo describe los pasos para el desarrollo de las citada herramientas.

Se hace una descripción de los conceptos básicos de lo que es la graficación por computadora y el modelado en el primer capítulo.

En el segundo capítulo se hace una comparativa entre el modelado geométrico con primitivas y el modelado orgánico, así como un listado de los métodos para modelar objetos orgánicos.

En el tercer capítulo se analizan los algoritmos de las herramientas seleccionadas para su desarrollo.

Después, en el cuarto capítulo, se describe como se reflejaron dichos algoritmos en lenguaje de programación, así como la forma de estructurar y desarrollar la interfaz de usuario de la aplicación. Por último en el quinto capítulo se hace una prueba funcional del prototipo para ambas herramientas de modelado.

— ∞ —

Introducción a los gráficos por computadora.

I.1. Inicio de los Gráficos por Computadora.

A pesar que desde principios de los 50's ya comenzaban a desarrollarse tecnología de hardware que permitiera la visualización de gráficos en forma primitiva, tanto en dispositivos basados en CRT (Tubos de Rayos Catódicos) como en dispositivos de impresión, no fue sino hasta principios de los 80's cuando su uso se extendió a casi todos los sistemas de cómputo, gracias en parte a que estos comenzaron a ser más accesibles para la mayoría de los usuarios.

A partir del desarrollo de los sistemas de cómputo a principios de los 60's, la necesidad de expresar los procesos, así como sus salidas de una forma más clara y comprensible a los usuarios de estos sistemas, ha dado pie al desarrollo de implementaciones que permitan, tanto en hardware como en software, una visualización gráfica de estos procesos.

Tanto en el desarrollo en hardware, como dispositivos de salida como el de software, algoritmos y rutinas, han permitido que los gráficos por computadora se conviertan, a lo largo del tiempo, en un elemento cada vez más imprescindible en los sistemas de cómputo actuales, tanto en las áreas educativas y de investigación hasta la industrial y comercial.

El constante avance de los sistemas de hardware ha permitido cada vez más un mayor poder de cómputo capaz de soportar eficientemente la carga de trabajo que significa implementar gráficos. Añadiendo que constantemente se desarrollan e implementan algoritmos y métodos destinados a mejorar la calidad de las imágenes obtenidas a un nivel fotorrealista, hacen posible el acceso a un trabajo grafico alta calidad.

Pero para que éste sea posible, hace falta el intermediario que plasme las ideas de los usuarios, es decir, las herramientas de software de diseño.

Desde principios de los 70's se han desarrollado herramientas para el diseño y modelado de gráficos, sin embargo su funcionamiento se basaba en la implementación de modelos geométricos primitivos, tales como la esfera, el cubo, etc., y la implementación de operaciones de conjunto sobre estos modelos (CSG, Geometría Sólida Constructiva); sin embargo, la capacidad de formar escenas complejas con estas herramientas era un trabajo difícil y muy laborioso, además que el nivel de detalle alcanzado por este tipo modelos era de muy bajo nivel.

Ante la necesidad, tanto del campo comercial como del científico, de obtener realismo, no solo en la calidad de imagen, sino también en el modelo, se comienzan a desarrollar complejas herramientas que permitan modelar objetos de la realidad basados en algoritmos matemáticos más elaborados.

I.2 Conceptos Básicos de Graficación.

Los primeros tipos de representación visual que se dieron para simular procesos, o representar un conjunto de datos, se hicieron en gráficos bidimensionales. Esto es porque las estructuras capaces de almacenar y generar gráficos apenas se comenzaban a desarrollar. Al presentarse una de los primeros esquemas de uso y programación de gráficos, como lo fue GKS (Graphical Kernel System) presentado por ISO (International Standards Organization), permitió unificar los esfuerzos de programación de software capaz de implementar gráficos en computadora. Sin embargo ésta primera versión de la librería GKS, en síntesis, sólo permitía manipular el despliegue de puntos sobre la pantalla, y cualquier objeto más complejo originado a partir de ellos.

Paulatinamente las librerías de software para tratamiento y creación de gráficos ha ido evolucionando de modo que permita no sólo manipular objetos primitivos, sino también curvas y objetos más complejos.

Sin embargo hasta estos objetos gráficos complejos tienen su fundamento en principios básicos, a partir de los cuales se puede definir cualquier gráfico que se quiera visualizar.

Los conceptos básicos para la representación en pantalla son píxel, resolución y modelos de color.

I.2.1 Píxel

Es la unidad mínima gráfica. Píxel es un término que deriva de "picture element" y se define como la unidad básica o mínima que compone un gráfico o imagen. Posee propiedades propias tales como posición, magnitud y color.

Cuando hablamos de una imagen o gráfico representado en pantalla, hablamos esencialmente de una representación bidimensional compuesta por un conjunto

ordenado de puntos de color, estos 'puntos de color' son píxeles, por lo tanto podemos decir que una imagen o gráfico en es una ordenación bidimensional de píxeles, en donde la posición del píxel es su ubicación dentro del área de visualización.

Fig. 1.1 Representación de los píxeles de un gráfico de 20x20.

A diferencia del concepto de línea en geometría definida como una sucesión infinita de puntos, en donde el concepto de 'punto' posee posición pero no posee área, el píxel si posee área, por lo tanto cualquier representación grafica se puede definir como un agrupamiento finito de píxeles.

Por último, la propiedad fundamental del píxel, desde el punto de vista de la representación en pantalla, es que posee tonalidad o color. Tomemos por ejemplo un monitor monocromático, en donde sólo visualiza 2 tonos: brillante (blanco) y obscuro (negro). Es gracias a esta última propiedad mediante la cual se logra la visualización de los objetos gráficos, ya que la representación ordenada de un conjunto de píxeles dará como resultado la visualización de una imagen o gráfico.

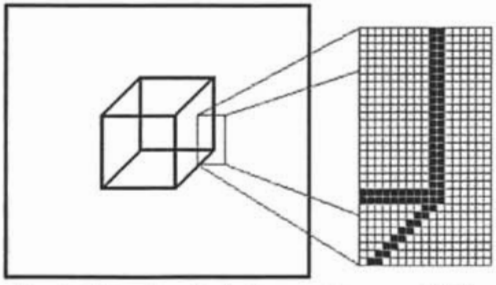


Fig. 1.2 Muestra de píxeles en una mapa de bits.

Actualmente la tecnología permite la representación de una amplia gama de tonalidades, basados en modelos de color más elaborados, lo que permite un despliegue gráfico con un mayor grado de detalle y nitidez.

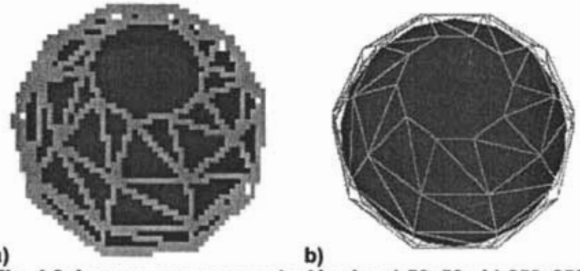
I.2.2 Resolución

El término "Resolución" en los gráficos por computadora hace referencia a dos conceptos básicos, uno asociado directamente con la imagen y el otro con la tecnología de dispositivos de salida.

En ambos sentidos se refiere a la cantidad o densidad de información gráfica que existe en un área determinada de un objeto visual.

En el primer caso se referirá a la cantidad de píxeles que conforman una imagen para su visualización, y estará dado por la cantidad de píxeles de ancho por la cantidad de píxeles de alto que contenga la imagen. Tomemos por ejemplo una imagen que posee 10 píxeles de ancho por 10 píxeles de alto su resolución será de 10x10 píxeles.

Bajo esta perspectiva, se deduce que a mayor resolución en una imagen, mayor nivel de detalle contendrá.



a) b)

Fig. 1.3 Imagen con una resolución de: a) 50x50; b) 250x250

La resolución es la cantidad de píxeles en un área determinada, sin embargo cuando se trata de resolución de imágenes es común referirnos a ella con la indicación de las cantidades de ancho y alto de ésta. Es decir, la resolución del ejemplo anterior debería indicarse como 100 píxeles, sin embargo definimos la resolución como 10x10 píxeles.

En el segundo caso, resolución se define como el número de píxeles o puntos por unidad de área. Este concepto se utiliza primordialmente en dispositivos de salida como impresoras o monitores, en los cuales es común hacer referencia en sus

especificaciones a la resolución como la cantidad de píxeles o puntos por pulgada (DPI, Dot Per Inch).

1.2.3 Relación de Aspecto

La relación de aspecto es un concepto importante en los gráficos por computadora, ya que se refiere a la relación entre ancho y alto en una imagen o gráfico.

Por ejemplo los televisores y los monitores tienen una relación de aspecto de 4:3 o 1 : 1.333, 4 unidades de ancho por 3 unidades de alto. Sin embargo existen otros formatos para visualizar imágenes como son el de 16:9[1] son aproximadamente 1:1.85. Cuando una imagen es redimensionada de una relación de aspecto de 4:3 a una de 16:9, o viceversa se genera una deformación en ésta por lo cual es necesario aplicarle algunas transformaciones a la imagen.

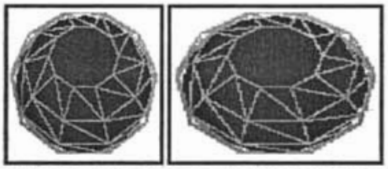


Fig. 1.4 Imagen ajustada a dos áreas con diferente relación de aspecto.

Éste concepto ha de tenerse en cuenta cuando se ha de pasar una imagen o gráfico de un área de visualización a otra con relaciones de aspecto diferentes. Para evitar deformaciones se pueden utilizar varias técnicas como las de desperdiciar áreas no muy importantes del gráfico o redimensionar la imagen o gráfico de forma en que no se ocupen algunas porciones del área destino

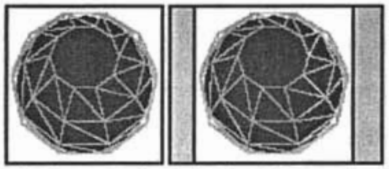


Fig. 1.5 Imagen ajustada sin deformaciones a un área con diferente relación de aspecto.

[1] Como el usado en la televisión de alta resolución.

I.2.4 Modelos de Color.

Un modelo de color es un método para explicar las propiedades o el comportamiento del color en algún contexto particular. Es una representación de un subconjunto visible de colores con el propósito de permitir la especificación práctica de colores dentro de una gama. Una gama de colores es un subconjunto de todas las cromaticidades visibles.

Lo que el ojo humano percibe como color es luz de diferentes tonalidades, y esta luz es una banda de frecuencias en el espectro electromagnético. Cada frecuencia en la banda visible del espectro electromagnético corresponde a un color distinto. En el extremo de banda visible se encuentra un tono o color rojo ($4.3x10^{14}$ Hertz) y la frecuencia más alta que podemos percibir es un color violeta ($7.5x10^{14}$ Hertz). El intervalo de colores espectrales va desde los rojos, hasta el anaranjado y el amarillo en el extremo de baja frecuencia a los verdes, azules y violeta en el intervalo de alta frecuencia.

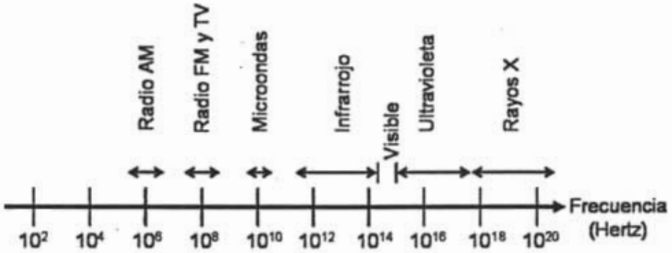


Fig. 1.6 Espectro Electromagnético

"Según las investigaciones de Young y la teoría de los tres estímulos de Helmholtz, los ojos perciben los colores mediante el estímulo de tres pigmentos visuales en los conos de la retina. Estos pigmentos visuales tienen una sensibilidad pico en longitudes de onda de alrededor de 630 nm (rojo), 530 nm (verde) y 450 nm (azul), esta teoría nos permite describir los colores como la mezcla estos tres (colores primarios)."[2] La figura 1.7 nos muestra la cantidad de luz de cada color (rojo r_λ, verde g_λ, azul b_λ) que se necesitan para igualar un color sobre el rango del espectro visible.

[2] Introducción a la graficación por computador. James D. Foley. Pag. 365.

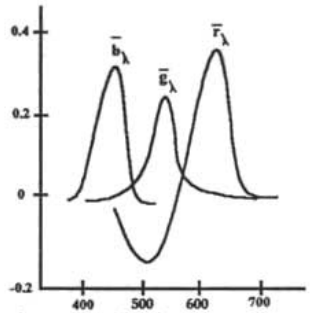


Fig. 1.7 Cantidades de cada primario necesarias para desplegar color.

Sin embargo la figura 1.7 nos describe la curva del rojo (r_λ) como una función que toma tanto magnitudes positivas como negativas. Y es precisamente entre los 445 nm y los 450 nm en los cuales la curva toma valores negativos. Por lo tanto, de acuerdo a este modelo, no podemos representar color alguno en este rango como una combinación de los tres colores. Por lo tanto cualquier modelo de color basado en el esquema de curvas anterior es un modelo parcial o incompleto. Sin embargo el ojo humano posee una baja capacidad para reconocer variaciones pequeñas entre colores muy parecidos, por lo tanto aunque los modelos de color sean incompletos, son suficientes para expresar los colores visibles para el usuario.

Existen varios modelos de color, pero generalmente se organizan en dos categorías básicas. Éstas son los modelos basados en la percepción y los modelos basados en el despliegue. Como pudiera pensarse de estos nombres, el primero es organizado similarmente a la manera como percibimos el color y el segundo esta basado en las características de un dispositivo de despliegue.

Los modelos que se basan en el primer concepto son: **HSV** (Hue, Saturation, Value, Matiz, Saturación y Valor) y el **HLS** (Hue, Light, Luz y Saturation). En estos modelos para dar una especificación de color, se selecciona un color espectral y se le agregan cantidades de blanco para aclararlo o negro para obscurecerlo.

Los modelos más comunes basados en despliegues, o en dispositivos de hardware, son el **YIQ**, el **RGB**, y el **CMY**. El **YIQ** es usado para transmisión de televisión, y posee componentes que definen la luminancia (brillantez), y su tonalidad. Sin embargo dentro de los modelos de color más usados en Graficación por computadora están los modelos **RGB** y **CMY**, los cuales por la simplicidad de sus estructuras facilita su uso para la

expresión del color de modo que facilite su manipulación para el despliegue y su almacenamiento.

El primero, el modelo **RGB**, se refiere a un modelo de color basado en síntesis aditiva, es decir, para definir un color específico este se obtiene de la suma de los colores primarios rojo, verde y azul, de ahí sus siglas en ingles (Red, Green, Blue). El modelo toma las características de la luz la cual indica que la suma de todas las tonalidades dará como resultado luz blanca. Por lo tanto la suma de los colores primarios dará como resultado blanco, y su ausencia resultará en negro. Una representación cómoda de éste modelo es su visualización tridimensional, que consiste en un cubo ubicado en el origen. Sobre cada eje se colocan los colores primarios. En el origen se encuentra el negro (0, 0, 0) y en el vértice opuesto (1, 1, 1) se encuentra el blanco. En los vértices restantes estarán los colores complementarios de acuerdo a los colores primarios que lo conforman. El segmento de recta de (0, 0, 0) a (1, 1, 1) contendrá la escala de grises del modelo.

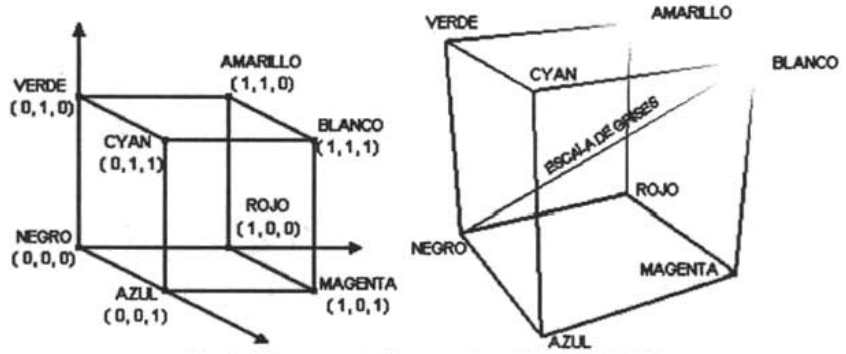


Fig. 1.8 Representación en cubo del modelo RGB

Este modelo está orientado a dispositivos basados en despliegue de luz, como por ejemplo los monitores y proyectores. Por lo cual es un modelo muy práctico para representar color en gráficos y por lo tanto es muy usado para almacenar imágenes sobre archivos, teniendo en cuenta que el modelo maneja 3 canales (rojo, verde y azul), se almacena cada píxel de color sobre 3 bytes, uno para cada canal, lo cual nos proporciona 256 niveles (0-255) para cada color primario, y proporciona aproximadamente 16.7 millones de tonalidades, suficiente para representar una imagen

a color real, ya que el ojo humano casi no distingue diferencias de tonalidad de 1/256 o menos.

El modelo **CMY** se refiere a un modelo que, igual que el modelo RGB, se basa en el manejo de tres canales de color y toma como colores primarios el Cian, el Magenta y el Amarillo, de ahí sus siglas en ingles (Cyan, Magenta, Yellow). Sin embargo es un modelo que se basa en síntesis sustractiva, es decir, resta tonalidades para obtener un tono o color específico, Por lo tanto la suma de sus colores primarios dará como resultado negro, y la ausencia de los mismos dará como resultado blanco. Su representación tridimensional consistirá en un cubo ubicado en el origen. Sobre cada eje se colocan los colores primarios. En el origen se encuentra el blanco (0, 0, 0) y en el vértice opuesto (1, 1, 1) se encuentra el negro. En los vértices restantes están los colores complementarios de acuerdo a los primarios que lo conforman. El segmento de recta de (0, 0, 0) a (1, 1, 1) contiene la escala de grises del modelo.

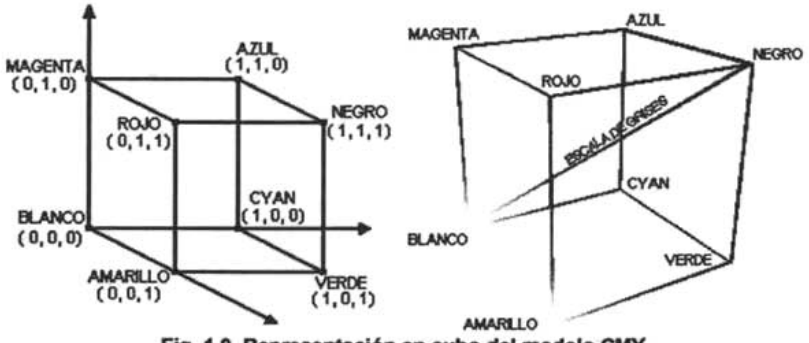


Fig. 1.9 Representación en cubo del modelo CMY

Es un modelo orientado a dispositivos de impresión, que se basan en la combinación de tintes para obtener una tonalidad específica de un color, por lo cual es muy usado para la descripción de color para impresión sobre papel.

I.3 Conceptos básicos de Graficación Tridimensional.

A partir del fortalecimiento de los gráficos por computadora, en su tratamiento bidimensional, y del desarrollo de las herramientas de hardware, como por ejemplo tecnologías de salida a color y mejores tarjetas de video, y herramientas de software, como por ejemplo la conformación de un estándar para uso y programación de gráficos

bidimensionales como lo fue el GKS (Graphical Kernel System) anunciado por ISO (International Standards Organization), la proliferación del uso de gráficos como herramienta se incremento en casi todas las áreas de desarrollo.

Sin embargo aún existía dentro del área del desarrollo de los gráficos un objetivo por satisfacer y era la representación de objetos o fenómenos del entorno de una manera que se acercara a la realidad.

Éste objetivo implicaba la capacidad para representar objetos o fenómenos que se encontraban o se desarrollaban dentro de un entorno con más de 2 dimensiones. Por ejemplo: la representación de un objeto cotidiano como lo es un vaso requiere necesariamente de un entorno tridimensional, es decir, expresar la magnitud de cada una de las características de visualización: alto, ancho y profundidad. Por otro lado, la representación visual de un fenómeno físico, como por ejemplo el de la aceleración de un objeto en su caída, requiere además de las características de objeto, la ubicación de la posición de éste, dentro de un entorno tridimensional, a lo largo de una variable de tiempo. Esto si bien podía ser simulado en gráficos bidimensionales, lo cierto es que obteníamos una representación incompleta y poco similar al objeto original.

I.3.1 Sistemas Coordenados.

Para localizar un punto en el espacio se debe de saber a que sistema de coordenadas se hace referencia.

El sistema de coordenadas en dos dimensiones, se define como un plano conformado por dos referencias: un eje o recta coordenado, que corta el plano a la mitad y un segundo eje o recta coordenado que será perpendicular al primero, de forma que entre ambos, dividen al plano en cuatro partes (o cuadrantes). Cada punto a lo largo del eje 'x' se le denomina abscisa, mientras que a los puntos sobre el eje 'y' se le denomina ordenada. La intersección de ambos se definirá como el origen de cada eje, teniendo cada uno de estos una parte positiva y una negativa. De esta forma, los cuadrantes quedan conformados de la siguiente forma:

I : abscisa positiva, ordenada positiva.
II : abscisa negativa, ordenada positiva.
III : abscisa negativa, ordenada negativa.
IV : abscisa positiva, ordenada negativa.

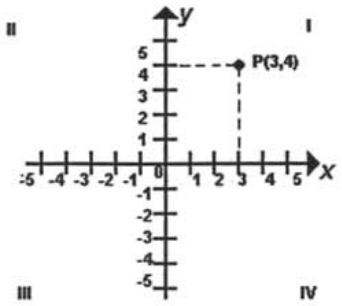


Fig. 1.10 Referencia cartesiana bidimensional

De esta forma, cada punto sobre el plano se identificara mediante dos referencias: su distancia con el eje 'x'(abscisa) y su distancia con el eje 'y' (ordenada).

Sin embargo el sistema anterior sólo contempla el conjunto de puntos situados en un plano. Pero para ubicar un punto en el espacio es necesario aumentar al sistema anterior una referencia más, eso se consigue teniendo tres ejes o rectas coordenadas mutuamente perpendiculares (x, y, z), dividiendo al espacio en octetos y agregando al sistema la característica de profundidad, mediante el cual ahora cada punto esta definido por tres referencias: su distancia con el eje 'x', su distancia con el eje 'y', y su distancia con el eje 'z'.

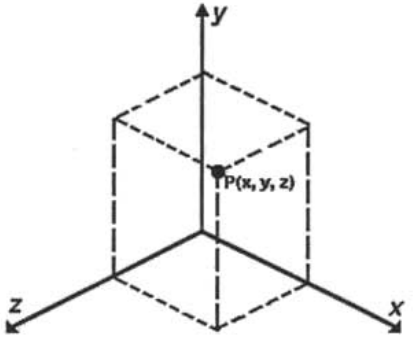


Fig. 1.11 Referencia Cartesiana Tridimensional.

I.3.2 Proyección.

Aunque la representación que estemos haciendo de algún objeto sea de manera tridimensional, cualquier representación de algún objeto que se muestre en pantalla es una visualización bidimensional. Esto se resuelve mostrando en pantalla la proyección de los puntos del objeto de acuerdo al ángulo de observación y la posición del objeto observado. Es decir, podemos ver al área de visualización como una porción de plano, al cual llamaremos plano de visión. Este plano tiene asociado un vector perpendicular a él que es su vector normal, en donde este vector define a que región del espacio se proyectará sobre el plano de visión.

Existen dos tipos básicos de proyección: en perspectiva y paralela.

En la proyección paralela una vez definido el plano de proyección y la dirección de visión, cualquier punto en el espacio se proyecta sobre el plano de visión en forma paralela al vector de dirección de vista. Hay dos categorías generales en la proyección paralela que son proyecciones ortogonales y las oblicuas, las primeras son aquellas donde el vector de dirección es perpendicular al plano de visión, y en las oblicuas no lo es.

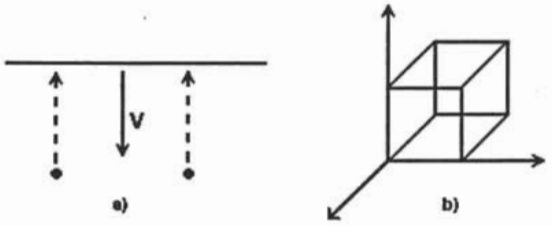


Fig. 1.12 a) Manera en que se proyectan los puntos sobre el plano de visión; b) cubo proyectado en forma paralela.

Este tipo de proyección se utiliza mucho en el área de diseño y modelado, en dibujo arquitectónico y de ingeniería comúnmente utilizan esta proyección porque los dibujos o trazos hechos conservan su forma y escala, es decir no existen deformaciones en el gráfico. En modelado de objetos tridimensionales este tipo de proyección se utiliza para presentar una vista previa del modelo mientras se esta diseñando, usándose comúnmente en la presentación de las vistas frontal, lateral, superior.

La proyección en perspectiva es el tipo de proyección más usada para obtener visualizaciones más realistas, ya que representa a los objetos de forma similar a como los ve el ojo humano.

Generalmente para generar este tipo de proyección se debe de aplicar una transformación al sistema de coordenadas de referencia para generar un sistema de coordenadas solidario al usuario, que es un sistema coordenado con origen en la posición del observador y orientado en el eje 'z' hacia la dirección de visión.

Este tipo de proyección se caracteriza por tener un acortamiento perspectivo, es decir que los objetos sufren una deformación en la escala, la cual disminuye conforme se alejan del centro de proyección, y por poseer puntos de fuga, anomalía que consiste en crear la ilusión de que las líneas que no son paralelas al plano de proyección convergen en un punto.

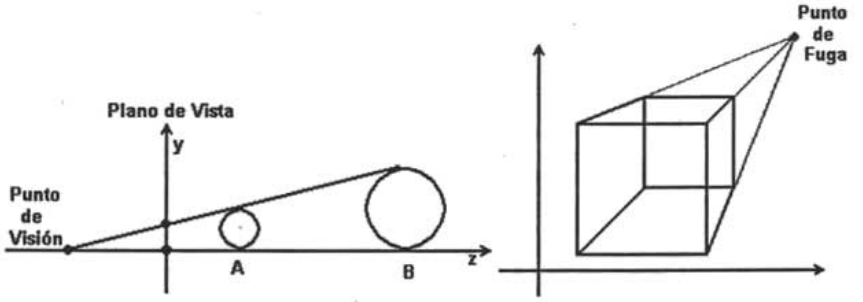


Fig. 1.13 Acortamiento perspectivo y puntos de fuga.

Aunque estas deformaciones alteran el modelo o los objetos originales, permiten a este tipo de proyección agregar a la visualización un grado de profundidad que aumenta el realismo de ésta.

I.4 Modelado de Objetos.

Para la representación de modelos que sean semejantes a los de la realidad comúnmente se generan objetos en tres dimensiones, y los más usuales son aquellos objetos, simples o compuestos, que basan su conformación en superficies. Por superficie se entiende un conjunto de puntos que cumplen con ciertas condiciones

geométricas. Para modelar un objeto en la computadora, necesitamos calcular y proyectar el conjunto de puntos que forman su superficie, para visualizarla en pantalla.
Sin embargo existe más de una forma de representar la superficie de un objeto en gráficos por computadora. Aunque en ambos se calcula los puntos de tal objeto, varia la forma en representar su superficie. Se pueden calcular los puntos de la superficie mediante ecuaciones matemáticas, o mediante la fragmentación de la superficie en trozos de plano y calcular los puntos sobre tales planos.
Un objeto se puede representar generando un modelo que contenga la totalidad de su superficie, o como el resultado de la combinación de un conjunto de modelos.

I.4.1 Modelado por Ecuaciones.

Podemos definir la superficie de un objeto como un conjunto de puntos obtenidos a partir de una ecuación matemática o de un conjunto de éstas. Con respecto del modelado de un objeto tridimensional, tales ecuaciones tendrán que arrojarnos como resultado un punto de la superficie con sus tres parámetros de posición (x, y, z). Por ejemplo, para visualizar un objeto tal como lo es una esfera podemos representarla mediante su ecuación cuadrática:

$$x^2 + y^2 + z^2 = r^2,$$

o mediante las ecuaciones paramétricas:

$x = r_x \cdot \cos(u) \cdot \cos(v)$ $\qquad -\pi / 2 \leq u \leq \pi / 2$

$y = r_y \cdot \cos(u) \cdot \text{sen}(v)$ $\qquad -\pi \leq v \leq \pi$

$z = r_z \cdot \text{sen}(u)$

Usando estas ecuaciones, cualquier punto obtenido de ellas estará situado sobre la superficie de la esfera.

Fig. 1.14 Imagen creada mediante la ecuación cuadrática $x^2 + y^2 + z^2 = r^2$

Mediante ecuaciones cuadráticas se puede definir un limitado número de superficies tales como la esfera, el cono o el cilindro.

Estos elementos se consideran objetos primitivos de los objetos gráficos tridimensionales.

Para estructurar modelos más complejos a partir de primitivas, se pueden emplear técnicas como lo es la Geometría Sólida Constructiva (CSG), que consiste en implementar operaciones de conjuntos (tales como unión, intersección y diferencia, o sus combinaciones) para elaborar modelos más complicados, sin embrago la capacidad de generar modelos muy complicados sigue siendo limitada, por lo cual se emplea otro tipo de modelado basado en aproximaciones a superficies específicas.

I.4.2 Modelado por Polígonos.

Cuando nos enfrentamos al problema que implica modelar un objeto que no es tan fácilmente expresada por un conjunto de ecuaciones, nos vemos en la necesidad de utilizar herramientas alternas que nos permitan expresar la superficie a modelar.

Una de las técnicas más usadas para generar una superficie sin usar ecuaciones es la de representación de superficies mediante polígonos, que se basa en hacer un muestreo sobre la superficie de forma discreta, para de ello obtener un conjunto de puntos sobre ésta. Una vez realizado esto, se generan un conjunto de polígonos tomando como vértices los puntos obtenidos, de modo que la unión de éstos reconstruye en forma aproximada la superficie original.

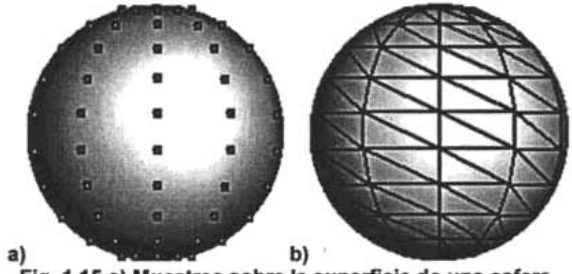


Fig. 1.15 a) Muestreo sobre la superficie de una esfera. b) Muestra de los polígonos resultantes.

Este tipo de modelado es más utilizado que el de por ecuaciones, ya que trabajar sobre polígonos y vértices es mucho más sencillo el editar un modelo para formar una forma

mas complicada, ya que sólo tendríamos que modificar algunos vértices para que la forma de la superficie resultante se aproximará al modelo deseado. A diferencia del modelado por ecuaciones en donde se tendría que obtener la modificación matemática apropiada para obtener la deformación deseada en el modelo.

Sin embargo en este tipo de modelado existen algunas desventajas. Ya que finalmente lo que se hace es aproximar una superficie mediante un poliedro, el modelado de superficies curvas resulta en una malla poliédrica y con facetas muy visibles, por lo cual sólo podemos aproximar curvas con este método. Sin embargo para obtener una mejor visualización final comúnmente lo que se implementa es aumentar el número de polígonos para aumentar el nivel de detalle de la superficie aproximada. De lo cual podemos deducir que a mayor número de polígonos tendremos un mayor nivel de detalle en la superficie.

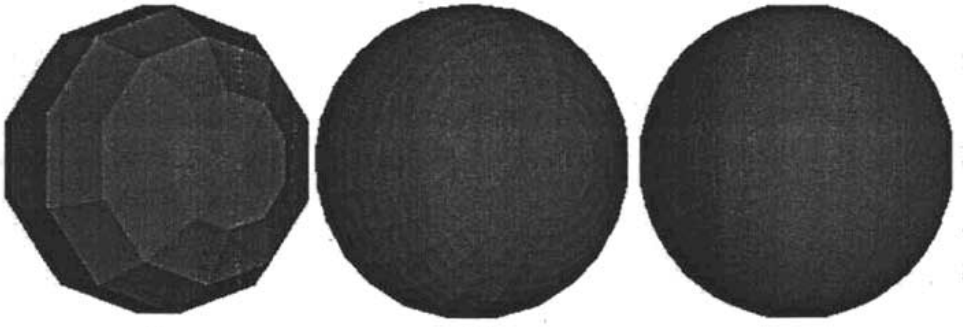


Fig. 1.16 Muestra de una esfera aproximada por: a) 120 polígonos, b) 1250 polígonos, c)5000 polígonos

Cuando se implementa por polígonos un modelo, se generan un conjunto de polígonos que se ajustan a la superficie del objeto a modelar, pero finalmente estos polígonos son objetos planos que se encuentran en el espacio, por lo tanto se deben de tratar como fragmentos de plano. Para efectos de visualización se utiliza esta propiedad para definirles textura, y darles una presentación final a estos modelos, mediante métodos de sombreado.

1.4.5 Sombreado de Intensidad Constante.

Es el método de sombreado más rápido y fácil de implementar ya que requiere de pocos cálculos. Sólo se requiere calcular una sola intensidad de color por polígono. Esto se realiza calculándola respecto al vector que forma la fuente de luz y el centro del polígono y al vector normal del mismo polígono. La intensidad de color obtenida es asignada a todo el polígono.

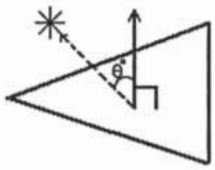


Fig. 1.17 La iluminación se calcula mediante el ángulo que forman la normal del polígono y el vector de iluminación.

El modelo genera visualizaciones muy pobres o una representación poliédrica, ya que polígonos adyacentes pueden tener cambios drásticos de intensidad de color. El modelo es correcto solo sobre ciertas condiciones:

a) El objeto representa un poliedro y no una aproximación a una superficie curva.
b) Se suponen las fuentes de luz y el punto de observación están en infinito, para que la atenuación de la apariencia sea constante en todo el polígono.

Fig. 1.18 Esfera formada por 120 polígonos con iluminación plana

Por su sencillez de cálculo es método de sombreado muy útil para la representación rápida y general de una superficie curva.

I.4.6 Sombreado de Gouraud.

El modelo de Gouraud se basa en la interpolación de intensidades de color. Es un método más complejo que el del Intensidad constante, ya que requiere significativamente más cálculos, pero permite una mejor visualización de la aproximación de una superficie curva mediante polígonos, ya que disminuye las diferencias de color entre polígonos adyacentes.

Para implementar el método primero se debe de encontrar el vector Normal asociado a cada vértice del polígono, es decir, cada polígono tiene un vector normal asociado, y para cada vértice ese vector será el promedio de los vectores normales de todos los polígonos adyacentes a él. Una vez obtenido el vector normal de cada vértice del polígono se obtiene la intensidad de color en cada uno. Por último, ya obtenidos todas la intensidades de color de cada uno de los vértices del polígono, se interpolan los colores de manera lineal sobre la superficie del polígono.

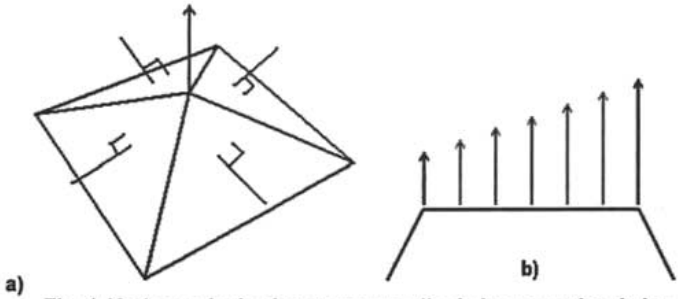


Fig. 1.19 a) se calcula el vector promedio de las normales de los polígonos adyacentes, y se calcula la iluminación, b) se interpolan los colores entre los obtenidos de sus vértices.

El resultado final es una aproximación visual a una superficie curva, pero tiene deficiencias. Los toques de luz en la superficie se visualizan de forma anormal, y la interpolación lineal de la intensidad de color puede provocar que aparezcan líneas de intensidad oscura o brillante, que se denominan bandas de Mach.

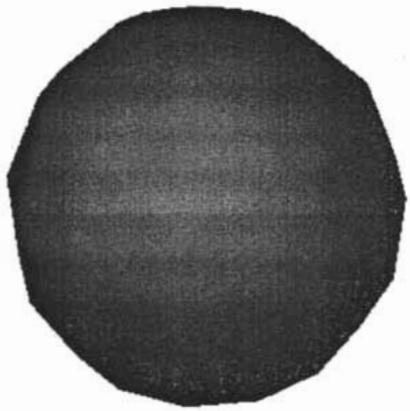


Fig. 1.20 Esfera con sombreado Gouraud.

I.4.7 Sombreado de Phong.

El modelo de Phong es un método que se basa en la interpolación de normales. Consigue mejor definición de imagen pero es más complejo que el de Gouraud y requiere una mayor cantidad de cálculos que éste, por lo tanto es un modelo que consume un mayor tiempo de implementación.

Para implementar el método primero se debe de encontrar el vector Normal promedio asociado a cada vértice del polígono. Una vez obtenido el vector normal de cada vértice del polígono se interpolan de manera lineal las normales de los vértices para cada punto sobre la superficie del polígono, y con las normales resultantes se calcula la intensidad de color correspondiente.

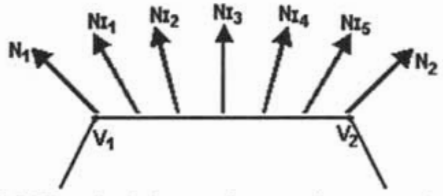


Fig. 1.21 Se calcula la normal promedio para cada vértice y se interpolan a lo largo del polígono.

Con este modelo se eliminan las deficiencias que se tenían en el método de Gouraud, ya que para cada punto visible de la superficie se obtiene su intensidad, por lo tanto si un toque de luz queda lejos de los vértices este no se omitirá de la imagen de la

superficie modelada. Con este método se pueden lograr captar características extras de los objetos modelados como sus reflejos especulares

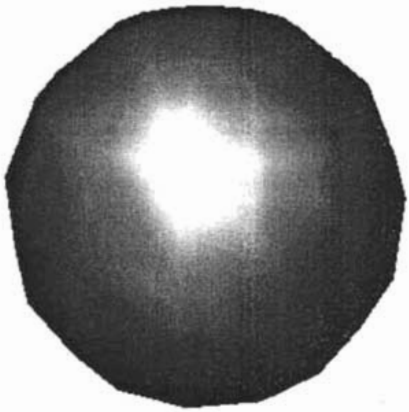


Fig. 1.22 Esfera con sombreado Phong.

CAPÍTULO II

Modelado de objetos orgánicos.

II.1 Objetos Orgánicos.

Una de las metas del modelado tridimensional es la creación de modelos que representen a los objetos de la realidad. El uso de primitivas en modelado nos permite crear objetos sencillos, pero simples, que poseen una estructura rígida, con pocos cambios sobre su superficie o cambios drásticos sobre ella.

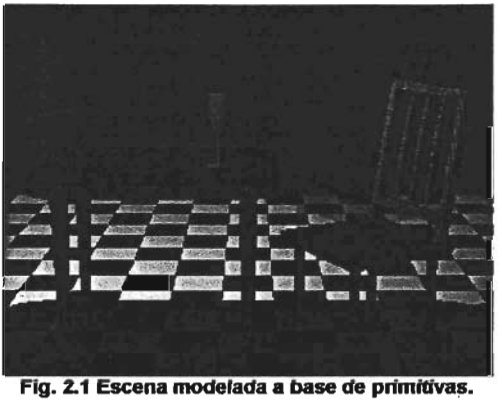


Fig. 2.1 Escena modelada a base de primitivas.

Algunos de los objetos tridimensionales modelados por computadora se realizan usando primitivas tales como cajas, esferas, cilindros, conos, etc., sin embargo cuando se desea modelar objetos con formas más complicadas este tipo de objetos resultan un tanto inútiles ya que proporcionan poco margen de flexibilidad para generar detalles específicos en dicho modelos.

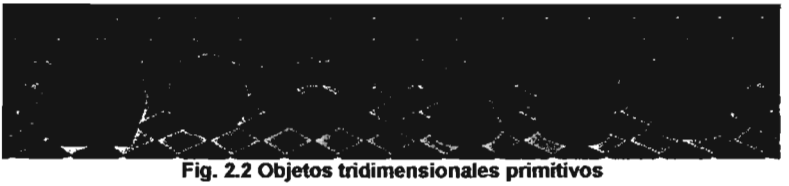


Fig. 2.2 Objetos tridimensionales primitivos

El uso de primitivas se implementa generalmente para modelar objetos geométricos o para la representación de objetos artificiales del entorno real, como pueden ser estructuras físicas de tipo arquitectónico.

Sin embargo en el entorno real abundan los objetos que poseen una forma poco poliédrica, con una superficie compuesta por curvas de pendiente suave, difíciles de modelar a base de primitivas.

En la naturaleza los objetos generalmente se caracterizan por no tener una estructura o forma rígida, sino más bien por poseer una estructura formada a partir de superficies curvas suavizadas y poco predecibles.

Tomemos como ejemplo la superficie de un rostro o el cuerpo de un animal, la cual es prácticamente imposible modelar con el uso de primitivas si se desea un moderado grado de realismo, ya que es claro que a través de esta superficie el cambio de pendiente no es constante, y varia de forma un poco predecible pero no parametrizable.

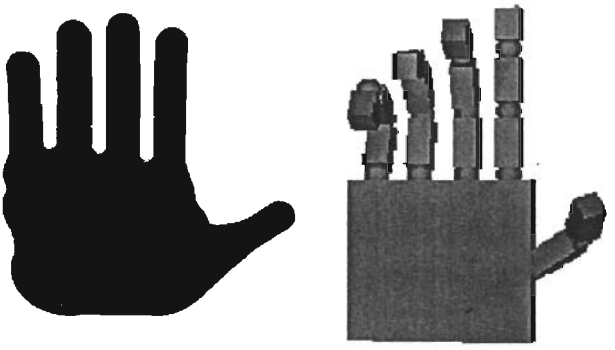


Fig. 2.3 Comparación de objeto organico y uno poliedrico.

Un objeto orgánico es entonces generalmente un objeto no poliédrico, sino compuesto por superficies curvas. Pero a la vez tales curvas no son fácilmente parametrizables o es difícil encontrar una función específica que la genere y a la vez permita una manipulación sencilla de ella.

En general la composición del cuerpo de la mayoría de los seres vivos, animal o vegetal, es una superficie de curvas suavizadas por la naturaleza orgánica de su composición. Pero también la estructura de elementos que interactúan con fluidos, o los fluidos mismos, adopta formas orgánicas. Por ejemplo las resinas o las formaciones rocosas desgastadas por erosión. Todos estos presentan las mismas características como son las de una superficie no fácilmente parametrizable y compuesta de áreas curvas.

A este tipo de objetos se les denomina Orgánicos, por ser estos tipos de elementos de la naturaleza los que precisamente presentan estas características.

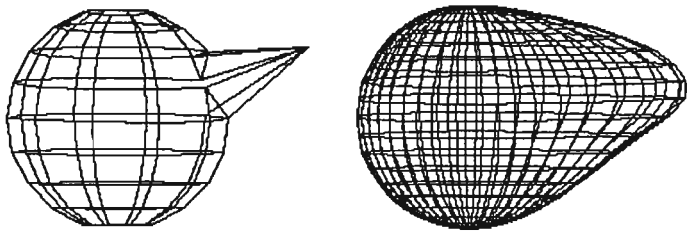
II.2 Herramientas de Modelado de Objetos Orgánicos.

Como ya se ha explicado, la representación de un objeto orgánico a base de ecuaciones especificas o implícitas es una tarea demasiado difícil. Por este motivo las herramientas de modelado orgánico generalmente fundamentan su funcionamiento en el uso de métodos matemáticas que se basan en la implementación de funciones paramétricas o la combinación de ellas.

Dentro de las herramientas de modelado orgánico podemos mencionar las siguientes basadas en: polígonos, splines y campo escalar.

II.2.1 Polígonos

Uno de los métodos para modelar superficies muy complicadas, sin el uso de primitivas es el uso de mallas de polígonos, la cual consiste en generar un conjunto de polígonos embaldosados sobre la superficie de modo que permitan la reconstrucción aproximada de la superficie original.
Sin embargo el sólo uso de polígonos para formar una superficie implica un conjunto de desventajas al momento de la edición sobre modelo, ya que al trabajar con curvas suaves necesariamente se necesita modificar más de un vértice para que la suavidad de la curva no desaparezca.



a) b)

Fig. 2.4 a) Modificación de un solo vértice a la malla de una esfera; b) para no perder la curvatura se deben de modificar mas de uno.

Además del inconveniente que si se desea más resolución sobre el objeto, al incrementar el número de polígonos que conforman la superficie, también aumenta el número de vértices a editar, volviéndose un trabajo tedioso y extremadamente difícil el editar eficientemente el modelo.

El uso de mallas de polígonos como herramienta de modelado generalmente es utilizado para objetos poliédricos, sin embargo, por la facilidad que permite esta forma de modelado para la edición por separado de cada vértice, permite la manipulación individual de los polígonos que forman el objeto. Por esta característica, a veces se recomienda utilizarla para generar objetos orgánicos que posean una alta complejidad en su forma. Sin embargo el uso de una malla de polígonos editados individualmente solo es recomendable para objetos que no presenten movilidad, es decir en una composición estática.

Algunas de las ventajas que ofrece este tipo de modelado están relacionadas con la manipulación individual de cada polígono.

Se puede variar la densidad de la malla de polígonos, al ser tan fáciles de implementar. Esto permite aumentar la cantidad de ellos sobre un área específica, y una vez hecho esto agregar detalles sobre la superficie elegida y no sobre toda la superficie del modelo. Podemos variar la densidad de la malla dependiendo del grado de detalle que deseamos.

Se puede utilizar el modelado mediante polígonos para la creación preeliminar de un objeto orgánico, modelándolo en baja resolución, y una vez obtenido la forma base aplicar sobre el objeto preliminar técnicas de subdivisión que permita incrementar la densidad de malla sobre el modelo.

II.2.2 Herramientas Basadas en Splines.

Las herramientas basadas en polígonos poseen obvias limitaciones al momento de modelar un objeto orgánico. Por esa razón se han desarrollado un conjunto de herramientas basadas en otro tipo de elementos.

A diferencia del uso de polígonos que se forman de puntos y la interconexión lineal de ellos, el uso de curvas basadas en ecuaciones matemáticas para definir su forma aumenta las posibilidades de generar objetos más complicados y con mejor resolución

ya que al generar los puntos de la curva mediante un cálculo matemático ya no es tan necesario almacenar todos los puntos que la conforman mientras se esta editando, y se puede alterar el número de puntos sobre la curva sin necesidad de alterar su forma.

Sin embargo la implementación de ecuaciones matemáticas para generar curvas o superficies debe proporcionar un método sencillo de modelado y edición para el objeto a tratar, si no se caería de nuevamente en el uso de primitivas. Para evitar este inconveniente se ha desarrollado un conjunto de herramientas de modelado basadas en curvas paramétricas, que permitan generar una curva o superficie mediante el uso de unos pocos puntos que permitan intuir la forma final de la curva.

Para lograr esto se utilizan métodos de interpolación o aproximación que proporciona mediante algunos pocos vértices guía una curva que se distribuye a lo largo del polígono que éstos forman. Este tipo de curvas son denominadas ***spline*** que es un término industrial que se le daba a unas tiras flexibles de metal o madera para generar curvas suaves a través de un conjunto de puntos designados previamente y se solía mantener la forma de la curva mediante contrapesos distribuidos a lo largo de la tira de tal forma que si se movían tales contrapesos se modificaba la forma de la curva. Actualmente en ambientes CAD (Diseño Asistido por Computadora) se le conoce generalmente como ***spline*** a cualquier curva compuesta que se forma con secciones polinómicas que pasa por un conjunto de puntos de manera que se ajuste a ellos y mediante los cuales en combinación con los polinomios generadores es posible modificar su forma.

Estas curvas son utilizadas para generar superficies basadas en la superposición de conjuntos ortogonales de curvas, para así generar una superficie que herede las propiedades de manipulación que poseen las curvas por si solas, y así poder generar superficies lo suficientemente complicadas y que ofrezcan poca dificultad al momento de editarla.

Una de las herramientas más comunes basada en ***splines*** son las curvas de Bézier. Fue una de las primeras en usarse en ambientes CAD, implementadas por Pierre Bézier a principios de los años 60 para la Renault. Es una de las herramientas más usadas para generar curvas y superficies suaves por su fácil implementación y propiedades propias de la curva que facilitan el diseño y edición.

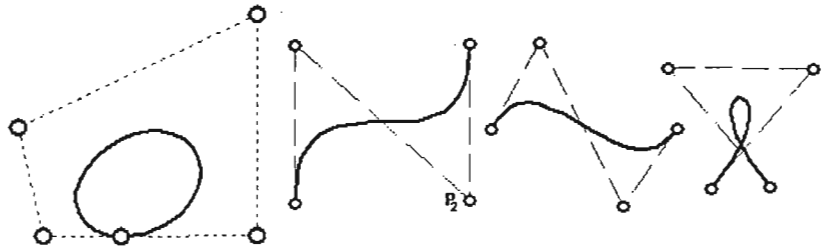


Fig. 2.5 Curvas de Bézier.

Esta definida por **n+1** puntos de control que generan un polígono abierto o cerrado al cual se ajusta la curva calculada. Los puntos de la curva paramétrica se calculan en base a la combinación de los puntos de control y la base de Bernstein, el cual genera para cada punto o vértice de control un polinomio de grado **n** que asociara a cada vértice un grado de influencia sobre la curva resultante. Por lo tanto para modificar la curva solo es necesario modificar los puntos de control. Como la curva sigue el polinomio formado por los vértices de control el diseño de la curva se vuelve intuitivo. Sin embargo por la naturaleza de sus ecuaciones paramétricas que dan origen a la curva Bézier, la curva no puede poseer una gran cantidad de puntos de control y a la vez ofrecer facilidad en el manejo de la curva, ya que los puntos o vértices de control ejercen una influencia global sobre esta. De modo que al modificar un sólo punto se altera toda la curva. Por tal motivo se recomienda usar curvas Bézier cúbicas, es decir de sólo cuatro vértices de control. Tanto en el diseño de curvas como en el diseño de superficies.

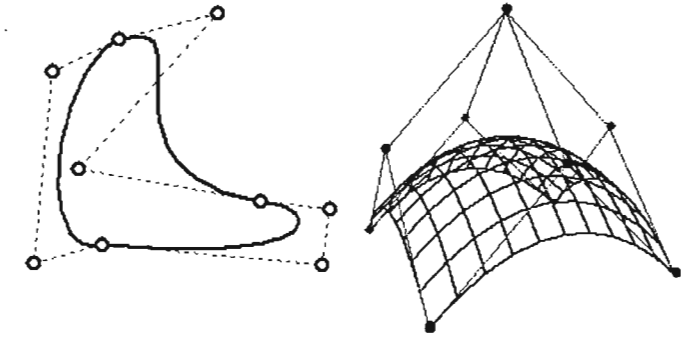


Fig. 2.6 Diseño utilizando curvas y superficies de Bézier.

Sin embargo este número de vértices de control no permiten generar superficies más sofisticadas, por lo cual generalmente se modelan superficies complejas a partir de más de una superficie de Bézier, lo que nos da como resultado una superficie total a base de pequeños parches. Por tal motivo se le denomina Parches de Bézier a este tipo de herramientas para modelar superficies.

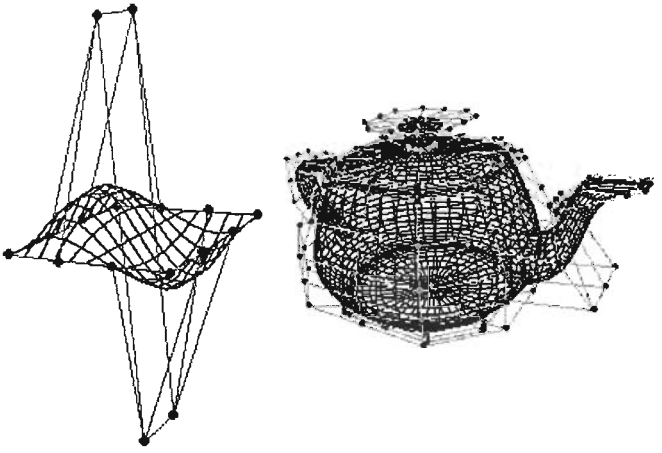


Fig. 2.7 Modelo a base de parches Bézier.

Una de las ventajas que tiene el uso de las curvas de Bézier es que se puede modelar la superficie solo con pocos vértices de control pero por su naturaleza parametrica de su método matemático es posible variar la densidad de la malla de polígonos de la forma que nos convenga con sólo variar un parámetro.

El uso de Parches de Bézier se recomienda para modelos que contengan una superficie curva muy suavizada. También se recomienda para modelar de forma rápida y sencilla un modelo con mínimos detalles en su superficie.

Pero no se recomienda el uso de curvas y Parches Bézier cuando se intenta modelar un objeto con agudos y complicados detalles sobre la superficie o cambios drásticos sobre ésta.

II.2.3 NURBS.

Existe otra herramienta basada en otro tipo de ***spline*** que es muy usada en el modelado de superficies. Esta basada de curvas NURBS o B-Spline Racionales No Uniformes (Non Uniform Rational B-Splines). Por la naturaleza de sus ecuaciones

paramétricas que dan origen a las curvas es posible tener una mayor manipulación de la superficie basada en NURBS que la que ofrecen las curvas Bézier, por lo cual es una de las herramientas más usadas para modelado orgánico.

Las NURBS usan puntos o vértices de control para definir una curva, como lo hacen las curvas Bézier, pero estos pueden ser cargados de forma que entre mayor sea su carga asociada, más se acerca la curva al vértice.

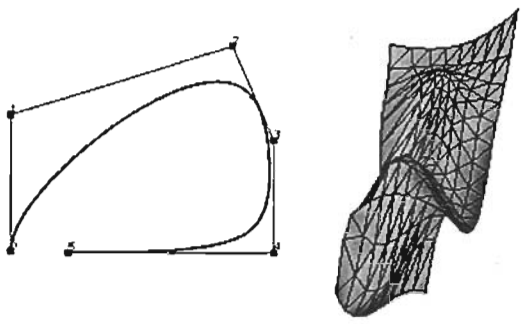


Fig. 2.8 Diseño usando curvas y superficies NURBS.

El uso de NURBS permite el manejo eficiente de líneas y superficies curvas.

Por su naturaleza matemática permite generar una malla de polígonos a cualquier resolución, es decir, permite variar la densidad de la malla de polígonos.

Permite una mayor facilidad de modelado, es decir, permite una mejor manipulación de la superficie que se modela.

Sin embargo, a pesar de ser una herramienta con una gran maniobrabilidad de edición, no permite crear fácilmente pequeños detalles sobre la superficie.

Puede requerir mucho tiempo de procesamiento por cada acción de edición de la herramienta sobre el objeto modelado.

II.2.4 Superficies en Revolución.

Otra herramienta que se puede utilizar para generar algunos modelos orgánicos es utilizar la técnica de superficies en revolución. Que consiste básicamente en tomar un polígono, ya sea abierto o cerrado, y girarlo a lo largo de un eje determinado, y con los puntos que resulten del giro generar la superficie final.

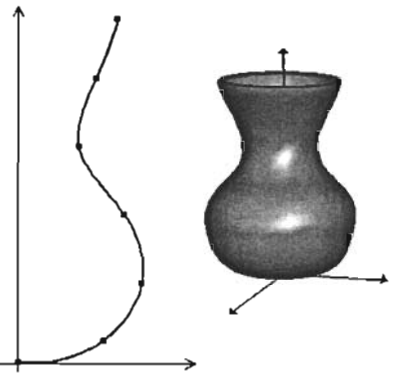


Fig. 2.9 a) Curva en dos dimensiones; b) superficie en revolución resultante

Se puede generar una superficie en revolución basándose en la rotación de una figura primitiva bidimensional, como es el caso del Toro, que se puede interpretar como una circunferencia girando alrededor de un eje. Como el ejemplo anterior podemos mencionar otros casos, sin embargo el uso de primitivas como polígono base para la superficie en revolución nos proporciona muy poca flexibilidad con respecto al modelado. Por lo cual las herramientas de software generalmente proporcionan herramientas que permitan que el propio usuario genere el polígono base. Lo pueden generar de manera lineal, es decir manejar el polígono como una unión de segmentos de recta, o pueden utilizar herramientas de curvas paramétricas, mediante las cuales pueden definir una curva con unos cuantos vértices de control, como son las curvas Bezier y las algunas otras curvas Spline.

Aunque los modelos derivados de esta técnica son objetos claramente geométricos, y la finalidad de un objeto orgánico es obviamente la representación de un objeto poco geométrico, sí es posible usar ésta herramienta para generar modelos orgánicos. Sin embargo implica agregarle un conjunto de transformaciones extras que permitieran generar un modelo más complicado, pero teniendo como fundamento básico el giro sobre un eje una figura.

A continuación se muestra un objeto basado en la rotación de una semicircunferencia pero modificando su distancia al eje de rotación y escalando la figura de rotación.

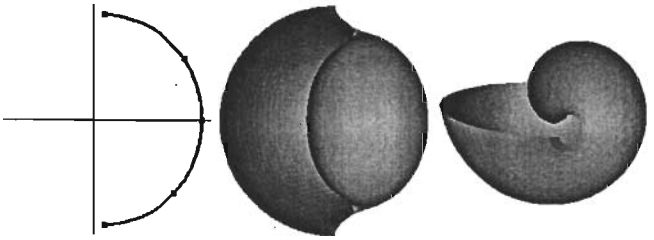


Fig. 2.10 Superficie en revolución con una escalación lineal respecto al ángulo de giro.

II.2.5 Blobs o Metaballs.

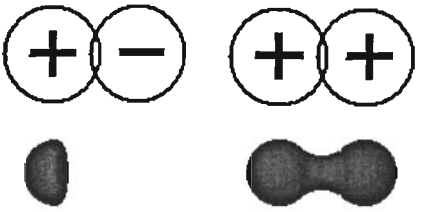
Los blobs o metaballs son una de las herramientas más usadas para modelar objetos orgánicos. Consiste en la generación de una superficie a partir de una serie de subprimitivas llamadas componentes las cuales tienen forma esférica. Cada componente posee tamaño propio y tienen asociado un campo escalar de fuerza de atracción o de repulsión.

La forma final de un blob o metaball es caiculada basándose en la posición, tamaño y principalmente del campo de fuerza de cada componente.

El campo escalar de la fuerza de atracción o de repulsión asociada a cada componente esta dada por un campo de densidad, que puede ser positivo o negativo, que se distribuye desde el centro del componente y disminuye de forma radial hacia fuera del componente aproximadamente en forma exponencial.

Un blob o metaball es una isosuperficie que se forma a partir de la combinación o suma de todos los campos de densidad que forman al blob o metaball. Es decir que la superficie final esta conformada por todos aquellos puntos en donde la suma o combinación de todos los campos de densidad tienen un mismo valor especifico o de umbral.

Esto permite que la interacción de componentes con campo de densidad positivo cause un efecto de atracción entre ellos, y componentes con campos de densidad positivos combinados con componentes con campo de densidad negativo muestra un efecto de repulsión entre aquellos con campo positivo con respecto a los últimos.



a) b)

Fig. 2.11 a)Efecto de repulsión con blobs de campo negativo; b)efecto de atracción entre blobs con campo positivo

Una de las ventajas de modelar con blobs es que se pueden generar modelos orgánicos con cierto grado de complejidad sin demasiado esfuerzo. La simple combinación de blobs puede generar algunos detalles orgánicos complejos que no son posibles de lograr con splines o son demasiado difíciles de lograr mediante el uso de polígonos.

Permite un rápido modelado de objetos orgánicos con detalle. Además de que proporciona cierta facilidad para manipular el objeto resultante y agregarle movimiento para poder crear una animación a partir de él.

Sin embargo la dificultad inherente al uso de blobs es que el modelado y la edición del objeto se vuelven intuitivos, es decir, la posición de cada blob en la escena influye en el objeto final, por lo tanto no hay reglas precisas que indiquen a que distancia se debe colocar cada elemento de los demás para lograr un resultado específico sobre la superficie.

La implementación de pequeños detalles o detalles finos sobre el modelo usando blobs es muy difícil, ya que los blobs tienden a suavizar los detalles pequeños.

Un modelo generado mediante blobs a veces contiene a través de su superficie bultos y depresiones tenues que son resultado de la combinación de sus elementos usados en el objeto. Este tipo de defecto se puede corregir aumentando el número de blobs en el modelo, pero esto complica más aún la edición y animación de éste.

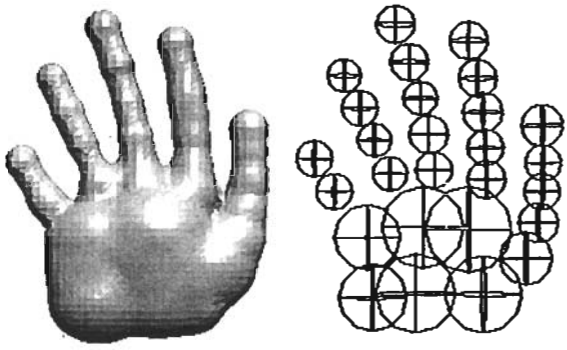


Fig. 2.12 Modelo basado en blobs y la distribución de sus componentes.

El uso de blobs o metaballs se recomienda cuando se intenta modelar objetos orgánicos naturales como árboles, algunos terrenos, rocas y efectos de líquidos, es decir para objetos orgánicos grandes sin finos detalles en su superficie.

El modelado mediante blobs permite un rápido diseño y edición de un modelo, así como también permite agregarle rápidamente y sencillamente pequeños detalles al modelo.

No es una herramienta recomendable para el modelado de objetos que posean partes lineales o áreas planas sobre su superficie.

El modelado con blobs o metaballs no es un modelado de precisión, si se desea un objeto con curvas precisas o exactas, se recomienda usar otro tipo de herramienta, como las splines.

II.2.6 Metashapes.

Los Metashapes son una herramienta que se comporta básicamente igual que los blobs o Metaballs, pero la diferencia radica en que con los Metashapes no sólo se modela con objetos esféricos, si no que se utilizan otras formas de objetos como cilindros, cajas, tiroides, etc., además de esferas.

Como en los Metaballs o blobs el campo de densidad dependía de una esfera, y por lo tanto el campo resultante terminaba siendo una esfera, en los Metashapes el campo depende de la forma del objeto utilizado y el campo de densidad adopta la forma del objeto o Metashape.

El hecho de que esta herramienta nos permita modelar con algo más que esferas, amplia las posibilidades de modelar un objeto orgánico muy complejo con menor

esfuerzo que con los blobs o metaballs, y es posible generar mayor grado de detalle en la superficie que con estos últimos.

II.2.7 Campo de Alturas.

Un campo de alturas o modelos de elevación digital es básicamente una malla bidimensional de polígonos cuyos vértices se encuentran asociados a un conjunto de valores, de los cuales se obtiene la magnitud para la dimensión restante. Generalmente se generan este tipo de objetos a partir de la asociación con un mapa de bits, tomando como valor de magnitud el color del píxel de la imagen.

El funcionamiento básico de esta herramienta consiste en el mapeo de la información contenida en el archivo, de acuerdo con el ancho y alto establecido, y asignar a cada vértice de la malla de polígonos la altura contenida en el archivo, ya sea color o dato crudo. Por lo tanto es obvio que la herramienta esta fundamentada en el uso de polígonos, que a diferencia del modelado por polígonos, en esta sólo podemos interactuar o modificar un parámetro del modelo, la altura o magnitud por vértice. A simple vista esta herramienta sólo produciría modelos poliédricos y un poco simples, y parecería que como herramienta de modelado orgánico no sería de gran utilidad. Sin embargo, existe la posibilidad de tomar como elemento orgánico el modelo resultante de un campo de altura.

Habíamos mencionado que un objeto orgánico se puede definir como un elemento tridimensional con una estructura no rígida, que posee una superficie con curvas suavizadas a lo largo de ésta, y es poco parametrizable. Es precisamente esta última propiedad la que nos permitiría englobar a los campos de alturas como una herramienta más de modelado orgánico.

A pesar de que el modelo es creado a base de una malla de polígonos de características lineales, lo cierto es que el modelo adopta la forma que se encuentra definida en el archivo de datos del cual extrae la información de altura. Y es precisamente en el archivo en donde radica el enlace que nos permitiría obtener como resultado un objeto orgánico, dependiendo de cómo se distribuyan los cambios de colores sobre el mapa de bits.

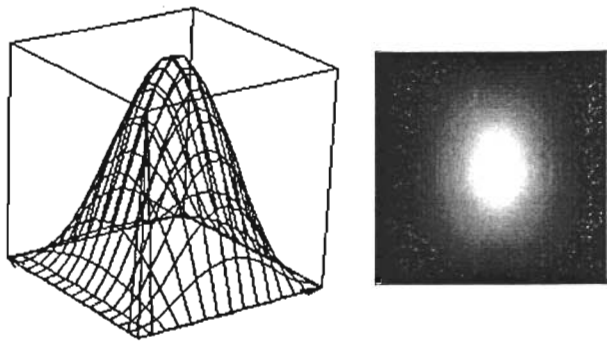


Fig. 2.13 Campo de alturas con su correspondiente mapa de bits.

Sin embargo es de mucha utilidad el uso de archivos de datos de información externa que definan la forma de la superficie resultante. Por ejemplo, el uso de esta herramienta se encuentra también el modelado de superficies topográficas en donde el archivo contiene información referente a la altitud de una superficie lo cual permite reconstruir áreas geográficas.

El uso de los mapas de alturas o modelos de elevación digital proporcionan muy poca capacidad de modelado, sin embargo suelen ser herramientas útiles al momento de representar superficies geográficas de manera rápida y eficaz.

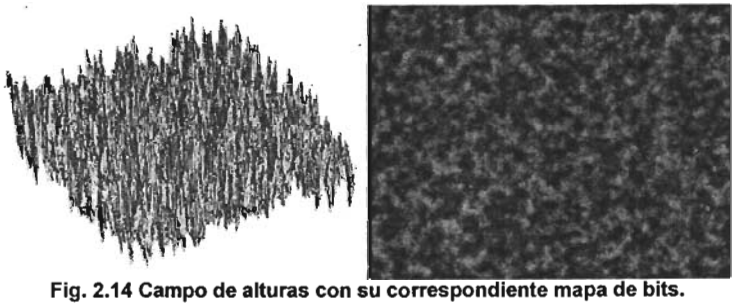


Fig. 2.14 Campo de alturas con su correspondiente mapa de bits.

— ∞ —

CAPÍTULO III

Análisis de las herramientas a desarrollar.

III.1 Herramientas a desarrollar.

De las herramientas de modelado mencionadas anteriormente se ha notado que existen una variedad de métodos para modelar objetos orgánicos. Sin embargo podemos notar que existen dos tipos de herramientas que se distinguen de entre el grupo mencionado y son las herramientas basadas en ***splines*** y el modelado basado en campos escalares (blobs).

De los métodos basados en campos escalares se pueden mencionar el modelado mediante ***Blobs*** y ***Metashapes***, sin embargo se ha elegido para su implementación los ***Blobs*** por su simplicidad y facilidad de uso.

De las herramientas basadas en ***splines*** también se encuentra un variado conjunto de métodos, pero las curvas Bézier son consideradas como la primera herramienta basada en ***splines*** usada en ambientes de diseño.

Las dos herramientas mencionadas se pueden considerar como representativas del modelado de objetos orgánicos, además de que se consideran las primeras herramientas implementadas en diseño y modelado en su categoría.

III.2 Curvas Bézier.

Tomaremos en primer lugar las curvas de Bézier por ser una de las primeras herramientas de modelado en aplicarse en diseño industrial.

La herramienta deriva del trabajo de unos ingenieros de las fábricas de automóviles Citroën y Renault. Se trata de P. de Casteljan y P. Bézier, quienes a finales de los años 50 y principios de los 60 desarrollaron sistemas de modelado muy similares para su aplicación en diseño de carrocerías.

Pierre Etienne Bézier fue un dedicado matemático con el fabricante de autos francesa Renault. A principios de los años 60's, alentado por sus patrones, comenzó a buscar una manera para automatizar el proceso de diseño de carros. Su método sería la base del moderno campo del CAGD (Computer Aided Geometric Design, diseño geométrico asistido por computadora), un campo de aplicaciones prácticas en muchas áreas.

Es interesante notar que Paul de Faget de Casteljau, un matemático de Citroën, fue el primero, en 1959, en desarrollar varios métodos de Curvas Bézier pero, por la manera

tan discreta de manejar su investigación, nunca fueron publicados sus resultados, excepto por dos documentos internos que fueron descubiertos en 1975. Es por esto que éste método creación y modelado de curvas lleva el nombre de Bézier, la segunda persona que lo desarrolló.

Aunque el trabajo del primero de ellos quedó como documento interno, el sistema UNISURF de Renault implementado por Bézier fue ampliamente publicado.

De aquí que el tipo de curvas, y superficies, que ambos estudiaron e implementaron lleven hoy el nombre de Bézier.

P. Bézier comenzó a trabajar en un método matemático para darle al diseñador mayor flexibilidad que las técnicas de interpolación usuales.

III.2.1 Definición de curva Bezier

Una curva Bezier es una curva paramétrica que se basa en los polinomios de Bernstein y que permite interpolar valores de una función entre un conjunto de puntos dados.

El proceso de construir una función f(x) verificando que en valores predeterminados de la variable independiente $x_0, x_1,\ldots, x_n$ tome valores dados $y_0 = f(x_0), y_1 = f(x_1),\ldots, y_n = f(x_n)$ es conocido como interpolación y es uno de los procedimientos clásicos de aproximación a funciones o a conjuntos de valores.

"En el contexto del modelado computacional, el diseñador que desea dibujar una curva (que posiblemente será el contorno de un objeto), generalmente comienza especificando puntos del plano $(x_0, y_0), (x_1, y_1),\ldots, (x_n, y_n)$ que intuitivamente siguen la forma que conceptualmente quiere obtener, o bien que procede de datos medidos experimentalmente y el sistema tiene que construir una curva suave que pase por dichos puntos. En esto consiste básicamente la interpolación."[1]

[1] Curvas y Superficies parta modelado geométrico. Juan Manuel CorderoValle. Pag. 27.

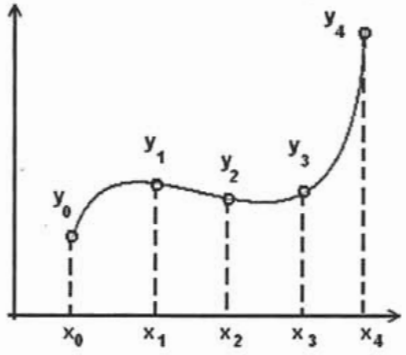


Fig. 3.1 Interpolación de los puntos de una curva.

Es obvio que una de las razones por las cuales se desea utilizar un método de interpolación sobre un conjunto finito de datos o puntos, es precisamente la perspectiva de obtener mediante estos métodos el resto de la información de la cual no contamos, es decir la información intermedia que se encuentra entre puntos y que permitiría la creación completa de la forma de un objeto grafico. Y es precisamente el método el que definirá la forma de esa información obtenida y la forma en como se manipula. Por ejemplo, dentro de los métodos conocidos de interpolación el más trivial es la interpolación lineal, en la cual solamente se tienen que encontrar la ecuación del segmento de recta que pasa por un punto y el siguiente. De esta manera se obtendría un polígono, ya sea abierto o cerrado que comunica la secuencia de puntos del conjunto de datos.

Fig. 3.2 a) interpolación lineal de un conjunto de datos; b) aumento de la cantidad de datos para aumentar la resolución.

Sin embargo es totalmente obvio que el resultado es un polígono y ante la necesidad de obtener una resolución de curva mayor implicaría el aumento significativo del número de puntos o vértices en la figura.

Una solución puede ser una interpolación basada en polinomios de grado n, que implicaría encontrar el polinomio que pasa por todos los puntos. Existen varios métodos

que permiten encontrar tal polinomio, tengamos por ejemplo 4 puntos (x_0, y_0), (x_1, y_1), (x_2, y_2), (x_3, y_3). Se necesitaría encontrar un polinomio de grado 3 que defina la función que contenga a los puntos antes mencionados

$$F(x) = ax^3 + bx^2 + cx + d$$

y que tenga las siguientes restricciones:

- En $x = x_0 \Rightarrow F(x) = ax_0^3 + bx_0^2 + cx_0 + d = y_0$
- En $x = x_1 \Rightarrow F(x) = ax_1^3 + bx_1^2 + cx_1 + d = y_1$
- En $x = x_2 \Rightarrow F(x) = ax_2^3 + bx_2^2 + cx_2 + d = y_2$
- En $x = x_1 \Rightarrow F(x) = ax_3^3 + bx_3^2 + cx_3 + d = y_3$

Lo que nos arroja el problema de encontrar las tres incógnitas que son a, b, c y d.

Supongamos que los puntos a interpolar son: (0, 2), (1, 1), (2, 0), (3, 5). El sistema de ecuaciones que proporciona la solución es:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 1 & 2 & 4 & 8 \\ 1 & 3 & 9 & 27 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix} = \begin{bmatrix} 2 \\ 1 \\ 0 \\ 5 \end{bmatrix}$$

Resolviendo obtenemos $[\, a_0, a_1, a_2, a_3 \,] = [\, 2, 1, -3, 1 \,]$ por lo que el polinomio de interpolación es:

$$F(x) = x^3 - 3x^2 + x + 2$$

Y cumple las condiciones de interpolación:

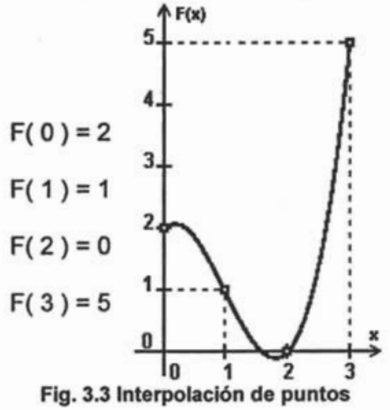


Fig. 3.3 Interpolación de puntos

Es obvio que si aumentamos el número de puntos o datos a interpolar a ***n*** el número de ecuaciones y de incógnitas se eleva igualmente a ***n***. Además de que el cambio de uno sólo de los puntos o vértices implicaría un cambio no intuitivo en el usuario y por lo tanto implica el recalculo de la solución.

Otro problema significativo desde el punto de vista de una herramienta CAD es que el uso de este tipo de interpolación nos ofrece muy poco margen de manipulación de la curva con respecto a las necesidades del usuario. Por ejemplo, supongamos que los puntos son el resultado de un muestreo sobre un objeto determinado cuya forma sigue una cierta trayectoria, marcada en gris en la figura 3.4, la curva resultante del método de interpolación presenta ciertas diferencias sobre la curva original, que son oscilaciones propias del método matemático que le da origen.

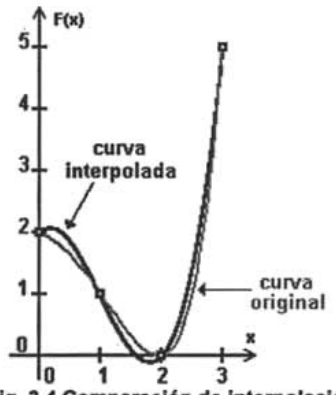


Fig. 3.4 Comparación de interpolación.

El problema es que tales oscilaciones son muy difíciles de eliminar y por lo tanto dificultan el proceso de edición y manipulación sobre la curva. Posiblemente existan un conjunto de ecuaciones que permitan ajustar de mejor manera la curva interpolada a la forma original del objeto pero entonces se necesitaría encontrar mediante un método externo al de interpolación y le quita generalidad al modelo de interpolación, haciéndolo no óptimo para un sistema CAD (Diseño Asistido por Computadora).

Es por eso que los métodos de aproximación por polinomios para ambientes CAD se basan principalmente en herramientas basadas en curvas ***Splines***.

Una ***Spline*** es un término industrial que se le daba a unas delgadas tiras flexibles de metal o madera para generar curvas suaves a través de un conjunto de puntos designados previamente y se solía mantener la forma de la curva mediante contrapesos distribuidos a lo largo de la tira de tal forma que si se movían tales contrapesos se

modificaba la forma de la curva. La curva resultante era la más suave posible, ya que la lamina tiende a minimizar su energía de flexión para no romperse.

Fig. 3.5 Representación de una tira spline.

Los contrapesos se pueden situar en cualquier lugar a lo largo de la tira doblándola en formas predecibles, lo que hace muy fácil e intuitiva su manipulación y edición.

Actualmente en ambientes CAD el término curva de ***spline*** "se refiere a cualquier curva compuesta que se forma con secciones polinómicas"[2], que pasa por un conjunto de puntos de manera que se ajuste a ellos y mediante los cuales en combinación con los polinomios generadores es posible modificar su forma.

Para especificar una curva de ***spline*** al proporcionar un conjunto de posiciones de coordenadas, que se conocen como *puntos de control*, que indican la forma general de la curva. Estos puntos de control se ajustan después con funciones polinómicas paramétricas continuas en el sentido de la pieza en una de dos maneras. Cuando las secciones polinómicas se ajustan de modo que la curva pasa a través de cada punto de control se dice que la curva que resulta realiza la ***interpolación*** del conjunto de puntos de control. Cuando los polinomios se ajustan a la trayectoria general del punto de control sin pasar necesariamente a través de ningún punto de control, se dice que la curva que resulta se ***aproxima*** al conjunto de puntos de control.

Una curva de ***spline*** se define, modifica y manipula con operaciones en los puntos de control. Mediante una posición inicial de los puntos de control se puede establecer una curva la cual puede ser editada mediante el cambio de posición de todos o algunos de los puntos de control para reestructurar su forma.

Las Curvas de Bézier son un tipo especial de ***splines*** que tienen varias propiedades que hacen que sean muy útiles y convenientes para el diseño de curvas y superficies, además de que son fáciles de implementar.

[2] Gráficas por computadora. Donald Hearn. Pag. 331.

III.2.2 Polinomios de Bernstein.

El método básico se basa en la implementación de la Base de Bernstein, que fue descrita en 1912 por S. Bernstein, quien la empleó en el contexto de la aproximación uniforme a funciones continuas.

La base de Bernstein puede ser descrita de varias maneras. Una construcción simple es considerar la expansión binomial de la expresión:

$$((1-x)+x)^n$$

La expansión binomial consiste de ***n+1*** términos de grado ***n*** en ***x***. Por ejemplo:

$$((1-x)+x)^3 = (1-x)^3 + 3(1-x)^2x + 3(1-x)x^2 + x^3$$

Los términos resultantes de la expansión son las funciones base de Bernstein de grado n. Aplicando el teorema del binomio produce una expresión explicita para la ***k-ésima*** función base de Bernstein de grado ***n***:

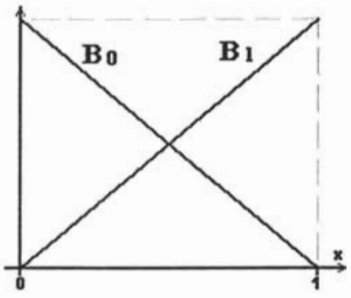
$$B_k(x) = \binom{n}{k} (1-x)^{n-k} x^k \qquad \text{con} \quad 0 \le k \le n$$

Donde $\binom{n}{k} = \frac{n!}{k!(n-k)!}$ son los coeficientes binómicos.

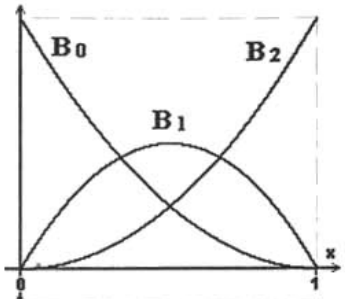
Generalmente sólo se definen para $x \in [0, 1]$, que es donde se manifiestan sus propiedades importantes.

Desarrollando los polinomios de Bernstein para n = 1, 2, 3, 4, tenemos las siguientes ecuaciones con sus respectivas visualizaciones:

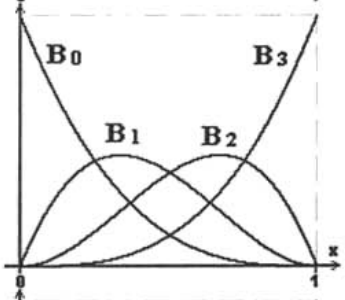
- n = 1
  $B_0(x) = 1 - x$
  $B_1(x) = x$



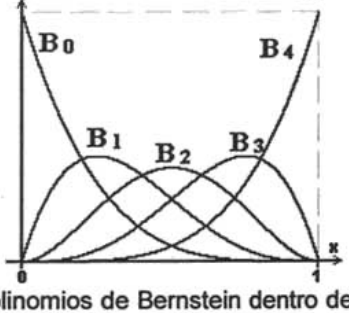
- n = 2
  $B_0(x) = (1 - x)^2$
  $B_1(x) = 2\,x\,(1 - x)$
  $B_2(x) = x^2$



- n = 3
  $B_0(x) = (1 - x)^3$
  $B_1(x) = 3\,x\,(1 - x)^2$
  $B_2(x) = 3\,x^2\,(1 - x)$
  $B_3(x) = x^3$



- n = 4
  $B_0(x) = (1 - x)^4$
  $B_1(x) = 4\,x\,(1 - x)^3$
  $B_2(x) = 6\,x^2\,(1 - x)^2$
  $B_3(x) = 4\,x^3\,(1 - x)$
  $B_4(x) = x^4$



Dentro de las propiedades que poseen los polinomios de Bernstein dentro del intervalo [0, 1] se pueden mencionar las siguientes:

- Las funciones base $B_k(x)$ son polinomios ***no negativos*** de grado ***n*** en el intervalo [0,1].

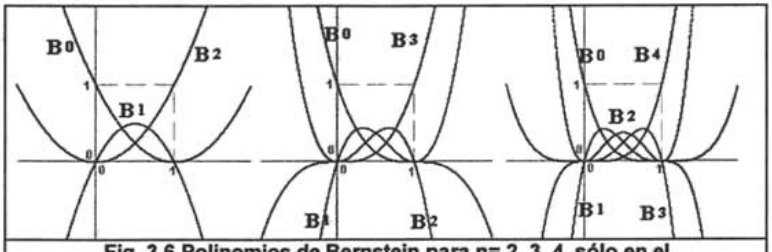


Fig. 3.6 Polinomios de Bernstein para n= 2, 3, 4, sólo en el intervalo de [0,1] se asegura la no negatividad

- $\sum_{k=0}^{n} B_k(x) = 1 \quad \forall x$. Esto se sigue inmediatamente del desarrollo del binomio de Newton;

$$\sum_{k=0}^{n} B_k(x) = \binom{n}{k} x^k (1-x)^{n-k} = [x + (1-x)]^n = 1 \qquad (3)$$

- $B_k(x)$ tiene exactamente un máximo en $x = \frac{k}{n}$. Esta propiedad puede ser verificada sencillamente comprobando que la derivada de $B_k(x)$ es cero en $x=\frac{k}{n}$.

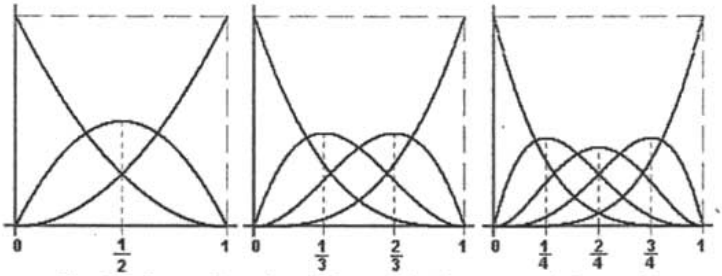


Fig. 3.7 Los polinomios de Bernstein tienen un máximo en k/n.

- $B_0(0) = 1, \quad B_k(0) = 0, \quad$ para $1 \leq k \leq n$.
- $B_n(1) = 1, \quad B_k(1) = 0, \quad$ para $0 \leq k < n$.

III.2.3 Curvas de Bézier

Una curva de Bézier es una aproximación a una curva mediante la combinación lineal de los polinomios de Bernstein. Teniendo ***n+1*** puntos de control, que serán los que formen el polígono de control, definidos de la forma $P_0=(x_0, y_0, z_0)$, $P_1=(x_1, y_1, z_1)$, ..., $P_n=(x_n, y_n, z_n)$, una curva Bézier es una función paramétrica de grado n y se expresa como:

$$C(u) = \sum_{k=0}^{n} P_k B_{k,n}(u)$$

[3] Curvas y Superficies parta modelado geométrico. Juan Manuel CorderoValle. Pag. 100.

Donde P_k son los puntos o vértices de control denotados en la forma (x_k, y_k), si la curva es de R^2, o (x_k, y_k, z_k), si la curva es de R^3, y $B_{k,n}(u)$ el k-ésimo polinomio de Bernstein de grado n, con $u \in [0, 1]$.

La función C(u) es una función vectorial que describe la trayectoria de la curva de Bézier aproximada entre P_0 y P_n, ya que los puntos de control P_k son vectores punto o vectores posición, entonces C(u) no proporcionara un vector punto o posición de una determinada coordenada de la curva. Esta se puede descomponer en tres ecuaciones paramétricas, con puntos en R^3, para las coordenadas individuales de la curva de Bézier:

$$x(u) = \sum_{k=0}^{n} x_k B_{k,n}(u)$$

$$y(u) = \sum_{k=0}^{n} y_k B_{k,n}(u)$$

$$z(u) = \sum_{k=0}^{n} z_k B_{k,n}(u)$$

Desarrollando la función vectorial C(u) para n = 1 tenemos:

$$C(u) = \sum_{k=0}^{1} P_k B_k(u)$$
$$= P_0B_0(u) + P_1B_1(u)$$
$$= (1 - u)P_0 + uP_1$$

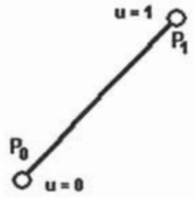


Fig. 3.8 Curva Bezier de grado 1.

Es notorio que el polinomio resultante del desarrollo de la ecuación C(u) es de primer grado, por lo tanto, la curva de Bézier resultante es una simple línea recta. Aunque el diseño de recta no es el fin de este método, este ejemplo sirve para verificar que la ecuación de Bézier sigue la poligonal formada por los puntos o vértices de control. Además que se verifican las siguientes características:

$$C(0) = P_0$$
$$C(1) = P_1$$

Recordando la forma de las ecuaciones de los polinomios de Bernstein para n = 1, es natural que la suma ponderada de los puntos de control, en donde el peso asociado a cada punto de control es el respectivo polinomio de Bernstein. Esto se vera más claramente para n>1.

Desarrollando la función vectorial C(u) para n = 2 tenemos:

$$C(u) = \sum_{k=0}^{2} P_k B_k(u)$$
$$= P_0B_0(u) + P_1B_1(u) + P_2B_2(u)$$
$$= (1-u)^2P_0 + 2u(1-u)P_1 + u^2P_2$$

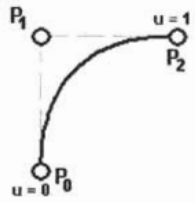


Fig. 3.9 Curva Bezier de grado 2.

Podemos apreciar de la figura 3.9 que las mismas características que se encontraron en la curva con n=1, también se encuentran en la curva con n=2:

$$C(0) = P_0$$
$$C(1) = P_2$$

Pero otra característica importante se encuentra derivando la función:

$$C'(u) = -2(1-u)P_0 + 2(1-2u)P_1 + 2uP_2$$

de donde obtenemos sustituyendo:

$$C'(0) = 2(P_1 - P_0)$$
$$C'(1) = 2(P_2 - P_1)$$

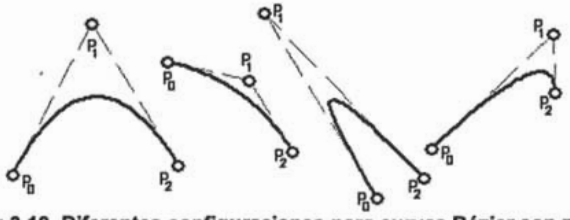


Fig 3.10 Diferentes configuraciones para curvas Bézier con n=2.

Es decir la curva pasa por el primer punto (P_0) y el último (P_n). Lo que significa que una curva de Bézier interpola el primer y último punto o vértice de control, pero sólo aproxima la curva al resto de ellos.

Desarrollando la función para n = 3 tenemos:

$$C(u) = \sum_{k=0}^{3} P_k B_k(u)$$
$$= P_0B_0(u) + P_1B_1(u) + P_2B_2(u) + P_3B_3(u)$$
$$= (1-u)^3P_0 + 3u(1-u)^2P_1 + 3u^2(1-u)P_2 + u^3P_3$$

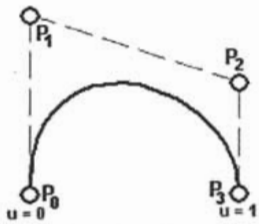


Fig. 3.11 Curva Bezier de grado 3.

Se aprecia de la figura 3.11 las mismas características que se encontraron en la curva con n=1 y n=2:

$$C(0) = P_0$$
$$C(1) = P_3$$

Y derivando tenemos:

$$C'(u) = -3(1-u)^2P_0 + 3(1-u)(1-3u)P_1 + 3u(2-3u)P_2 + 3u^2P_3$$

$$C'(0) = 3(P_1 - P_0)$$
$$C'(1) = 3(P_3 - P_2)$$

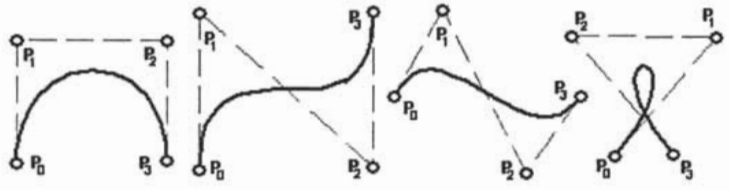


Fig. 3.12 Diferentes configuraciones para curvas Bézier con n=3.

Se puede observar que como regla, una curva Bézier es un polinomio de grado uno menos a el número de puntos o vértices de control que posee la curva: dos generan una recta, tres una parábola, cuatro puntos una curva cúbica y así en sucesivamente. Sin embargo conforme el grado de la curva se incrementa, el grado de la curva

aumenta y la magnitud de los cálculos también, lo cual dificulta su implementación en ambientes CAD.

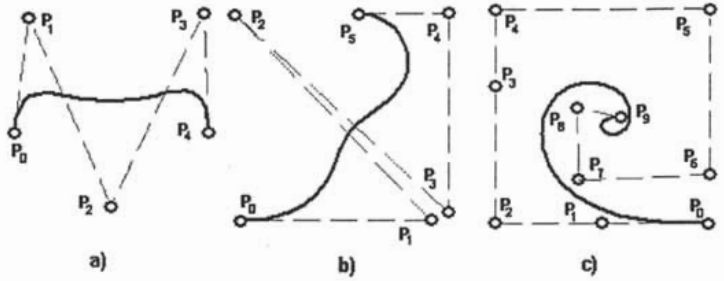


Fig. 3.13 Curvas de Bézier para: a) n=4, b) n=5, c) n=9

Además existe otra desventaja al aumentar el numero de vértices de control, y por lo tanto el grado de la curva, y que consiste en que la curva Bézier se vuelve muy difícil de manipular, ya que, de acuerdo con los polinomios de Bernstein, todos los vértices de control influyen en la curva resultante, a excepción de C(0) y C(1) en donde sólo influyen el primer y último vértice respectivamente. Por lo tanto al aumentar los vértices de control cada trozo de la curva esta influenciado en mayor o menor medida por el resto de los vértices de control complicando su edición y manipulación.

Generalmente se usan curvas cúbicas de Bézier, con 4 vértices de control, en ambientes CAD lo cual simplifica su implementación y manipulación.

III.2.4 Propiedades de una Curva de Bézier.

Algunas propiedades elementales de las curvas Bezier son:

- Interpolación de los puntos de control extremos.

  $$C(0) = P_0$$
  $$C(1) = P_n$$

  Una curva Bézier siempre pasa por el primero y el último punto de control. Esta propiedad se hereda de las características de los polinomios de Bernstein,

  $$B_k(x) = \binom{n}{k} (1-x)^{n-k} x^k$$

  ya que sólo el primer polinomio y el último alcanzan como valor máximo el valor de 1 en x=0 y x=1.

$$B_0(0) = \binom{n}{0} (1-0)^{n-0} x^0 = (1)^n (0)^0 = 1$$

$$B_n(1) = \binom{n}{n} (1-1)^{n-n} x^n = (0)^0 (1)^n = 1$$

- Control global o seudo local. Dado que los polinomios de Bernstein $B_k(u)$ alcanzan un máximo y éste ocurre para $u = \frac{k}{n}$, por lo tanto la máxima influencia de ese polinomio, y por lo tanto del k-ésimo vértice de control, sobre la curva Bézier se da precisamente en ese valor y decrece conforme se aleja de él hasta llegar a 0 para u=0 y u=1.

  Sin embargo si tenemos en cuenta nuevamente la forma de los polinomios de Bernstein, solo el primer polinomio y el último llegan a tener el valor máximo igual a 1, mientras que el resto no, por lo tanto, la modificación de sólo un vértice de control no afecta de manera sustantiva la curva resultante, por eso se dice que una curva Bézier posee control global. Sin embargo la modificación de un punto de control, aunque afecta la forma de toda la curva, es mas notable en las cercanías del punto editado.

- Restricción a la envolvente convexa, esto quiere decir que una curva de Bézier con puntos de control P_0, P_1,..., P_n es siempre interior a la envolvente convexa de dichos puntos. La envolvente convexa de un polígono de vértices P_0, P_1, ..., P_n es la intersección de todos los conjuntos convexos que contienen al polígono. Es decir, el menor de los polígonos convexos que lo contienen.

  Esto obliga a la curva de Bézier ***C(u)*** a permanecer cercana a lo puntos de control, lo que implica un control global a distancia de los puntos o vértices de control.

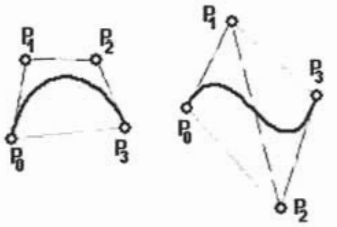


Fig. 3.14 Restricción a la envolvente convexa

Esto se desprende de las propiedades de los polinomios de Bernstein:

$$B_k(u) \geq 0$$

$$\sum_{k=0}^{n} B_k(u) = 1$$

y que la curva es una suma ponderada de las posiciones de los puntos de control P_k. Esta propiedad de la curva de Bézier garantiza que el polinomio siga con suavidad los puntos de control sin oscilaciones erráticas.

- Invarianza Afin. Quiere decir que la curva no varía si es sometida a transformaciones. Considerando una curva Bézier C(u) con puntos de control P_0, P_1, ..., P_n. Si aplicamos una afinidad a estos puntos:

  $$P' = A\,P + H$$

  donde A es una matriz no singular y H un vector columna representando una traslación, y a continuación construimos una curva de Bézier C'(u) a partir de los puntos transformados, ésta es precisamente la transformada afín (punto a punto) de la curva original C(u).

  Esto permite que, por ejemplo si deseamos rotar una curva de Bézier, es suficiente con aplicar la trasformación de rotación a los puntos de control y a partir de estos ya trasformados reconstruir la curva de Bézier.

  Esta propiedad se cumple solamente por el hecho de que la suma de los polinomios de Bernstein es igual a 1.

III.2.5 Diseño de formas con curvas de Bézier.

Al recomendar el uso generalizado de curvas de Bézier de 3er grado, 4 vértices de control, se limita el rango de manipulación de la curva y por lo tanto la cantidad de formas que puede tomar una curva de Bézier al modelar algún objeto. Es por eso que se explotan algunas de sus características para facilitar el modelado.

Un primer caso, quizás el más trivial, es el uso de curvas cerradas de Bézier. Esto es posible gracias a la propiedad que consiste en que "una curva de Bézier siempre pasa por el primero y el último vértice de control".

Por lo tanto, para generar una curva cerrada de Bézier tan sólo es necesario hacer coincidir el primero y el último punto de control.

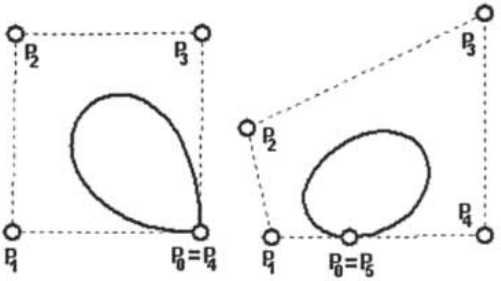


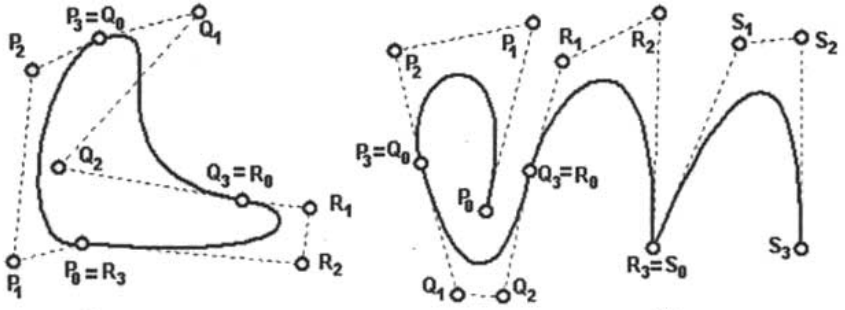
Fig. 3.15 Curvas cerradas de Bezier.

Sin embargo al posicionar más de un vértice de control en una misma posición se obtiene más peso para esa coordenada específica. Esto dará como resultado que la curva se jale más hacia esa posición repetida de los puntos de control.

Es claro que esta configuración del polígono de control otorga muy poco rango de manipulación de la curva y por lo tanto ofrece un pobre diseño de formas.

Este caso se repite también para curvas de Bézier abiertas, ya que el uso de curvas cúbicas otorga poca maniobrabilidad de las forma de éstas.

Por eso, cuando se necesita generar curvas algo más complicadas, es común utilizar un método de diseño a base de la unión de varias secciones de curva de Bézier de 3er grado o inferior.



a) b)

Fig. 3.16 a) curva Bézier cerrada modelada con 3 secciones (P, Q y R);
b) curva Bézier abierta modelada con 4 secciones (P, Q, R y S)

Al unir secciones más pequeñas permite un mejor control sobre la forma de la curva en regiones más pequeñas.

Uno de los problemas de usar este tipo de diseño es el de la continuidad de su la tangente de cada curva entre secciones. Si se observan las figuras anteriores, se notara que la figura 3.16.a casi no existen discontinuidades en las uniones de las

curvas Bézier, sin embargo en la figura 3.16.b se observa una notoria discontinuidad en las pendientes de la unión de las curvas R y S (R_3=S_0) y esto da un efecto negativo desde el punto de vista de diseño si lo que se intenta modelar son figuras orgánicas, ya que para ello se necesita modelar curvas suaves con pocos cambios de pendiente. En tal caso la figura 3.16.b no reúne los requisitos para tal meta.

Para ello se necesitan definir algunos parámetros para asegurar la continuidad de pendiente en las uniones de secciones de curva Bézier.

La primera condición de continuidad para una unión de dos curvas de Bézier P y Q es, como ya habíamos mencionado, que se cumpla lo siguiente:

$$P_n = Q_0$$

Cuando sólo se cumple este requisito de continuidad se dice que la curva tiene ***continuidad geométrica o visual de orden cero*** G^0.

Anteriormente habíamos hecho notar que al derivar la ecuación de una curva Bézier de 3er. grado se obtenía las siguientes ecuaciones de sus extremos:

$$P'(0) = 3(P_1 - P_0)$$
$$P'(1) = 3(P_3 - P_2)$$
$$Q'(0) = 3(Q_1 - Q_0)$$
$$Q'(1) = 3(Q_3 - Q_2)$$

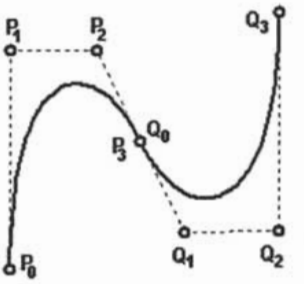


Fig. 3.17 Unión de la curvas Bezier P y Q.

Estas ecuaciones nos proporcionan el vector tangente a la curva de Bézier en sus puntos extremos. Podemos generalizar estas ecuaciones de la siguiente forma:

$$C'(0) = n(P_1 - P_0)$$
$$C'(1) = n(P_n - P_{n-1})$$

Desde el punto de vista geométrico se puede observar que el vector tangente a la curva P en P_n tiene la dirección de $P_n - P_{n-1}$, mientras que el vector tangente a la curva Q en Q_0 tiene la dirección $Q_1 - Q_0 = Q_1 - P_n$. Por lo tanto una condición necesaria y suficiente para que la pendiente a la primera curva en Pn tenga la misma dirección que la tangente a la segunda en Q0 es que los vectores $P_n - P_{n-1}$ y $Q_1 - Q_0$ sean proporcionales, y para ello el punto Q_1 debe ser colineal con P_{n-1} y $P_n = Q_0$. Es decir, el primer segmento del polígono de control de la curva Q debe ser una prolongación del

último segmento del polígono de control de la curva P, entonces se dice que la curva tiene ***continuidad geométrica o visual de orden uno*** G^1.

Pero si los vectores tangentes, primera derivada, son iguales tanto en magnitud como en dirección, entonces se dice que la curva tiene ***continuidad de orden uno*** C^1. Es decir, los puntos P_{n-1}, $P_n=Q_0$ y Q_1 deben de ser colineales y deben de tener el mismo espaciado.

Si los vectores resultado de la derivada ***n*** de ambas curvas en el punto de control donde se unen son iguales, entonces se dice que la curva tiene ***continuidad de orden n***, C^n.

La continuidad de orden mayor a 1 es posible de implementar, sin embargo conlleva restricciones que dejan poco margen de manipulación a la curva de Bézier y por lo tanto no es común su uso. Por ejemplo, la continuidad de segundo orden en curvas cúbicas de Bézier, con 4 vértices de control, requiere fijar 3 puntos de control en forma colineal con respecto al punto de unión entre curvas, lo que solo deja 1 punto de control libre para su manipulación. Pero si una curva de Bézier esta unida por ambos extremos, esta configuración resulta absurda e inútil de implementar, ya que en el extremo C(0) deberían de estar alineados los puntos de control P_0, P_1, y P_2 para asegurar la continuidad, y por el otro lado en C(1) la alineación debería ser con P_1, P_2, y P_3 para asegurar la continuidad en ese extremo, lo que nos deja una configuración colineal de P_0, P_1, P_2, y P_3, esto resulta absurdo ya que esa configuración nos genera como curva de Bézier una línea recta de P_0 a P_3, lo cual resulta una implementaión inútil.

Generalmente una continuidad geométrica G^1 es suficiente para proporcionar una buena herramienta de diseño y modelado.

III.2.6 Superficies de Bézier.

Para modelar curvas en el espacio es necesario obtener la ecuación paramétrica para cada componente del vector posición o vector punto Pk(x_k, y_k, z_k):

$$x(u) = \sum_{k=0}^{n} x_k \, B_{k,n}(u)$$

$$y(u) = \sum_{k=0}^{n} y_k \, B_{k,n}(u)$$

$$z(u) = \sum_{k=0}^{n} z_k \, B_{k,n}(u)$$

en donde las ecuaciones dependen de una sola variable independiente ***u*** y generalmente definen una trayectoria en el espacio.

Sin embargo, en principio una superficie se representa mediante una ecuación en la que generalmente una de las coordenadas esta expresada explícitamente en función de las otras dos:

$$z = f(x, y)$$

donde la pareja (x, y) pertenecen a un determinado dominio en R^2.

En el caso de la expresión de una superficie mediante su forma paramétrica esta dada por:

$$S(u, v) = \begin{cases} x = f(u,v) \\ y = g(u,v) \\ z = h(u,v) \end{cases}$$

donde *f*, *g* y *h* serán en general funciones de un espacio vectorial con una base ***producto tensorial***.

Ya habíamos visto que para el problema de interpolación se necesita un conjunto de puntos o vértices de control, y una base de interpolación, que en el caso de las curvas de Bézier es la base de Bernstein de grado n: $B_0(u)$, $B_1(u)$, ..., $B_n(u)$, de donde obteníamos las funciones base para una sola variable (***u***). Sin embargo, ahora la base tiene que quedar en función de ***u*** y ***v***. Una generalización para este fin es lo que se denomina ***producto tensorial*** de dos bases de potencias de grado n y m:

$$\begin{aligned} \text{Base} &= [\, B_0(u), B_1(u), \ldots, B_n(u) \,] \otimes [\, B_0(v), B_1(v), \ldots, B_m(v) \,] \\ &= \{\, B_0(u), B_1(u), \ldots, B_n(u), \\ &\quad B_0(u)B_1(v), B_1(u)B_1(v), \ldots, B_n(u)B_1(v), \\ &\quad B_0(u)B_2(v), B_1(u)B_2(v), \ldots, B_n(u)B_2(v), \\ &\quad \ldots, \\ &\quad B_0(u)B_m(v), B_1(u)B_m(v), \ldots, B_n(u)B_m(v) \,\} \end{aligned}$$

donde las funciones son todos los productos posibles entre un polinomio de la primera base y otro polinomio de la segunda base.

Los elementos de la base obtenida se pueden agrupar en una estructura rectangular de la siguiente manera:

$$\begin{array}{ccccc} B_0(u) & B_1(u) & B_2(u) & \cdots & B_n(u) \\ B_0(u)B_1(v) & B_1(u)B_1(v) & B_2(u)B_1(v) & \cdots & B_n(u)B_1(v) \\ B_0(u)B_2(v) & B_1(u)B_2(v) & B_2(u)B_2(v) & \cdots & B_n(u)B_2(v) \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ B_0(u)B_m(v) & B_1(u)B_m(v) & B_2(u)B_m(v) & \cdots & B_n(u)B_m(v) \end{array}$$

y esta constituida por (n+1)(m+1) funciones.

Una función expresada como combinación lineal de sus elementos será entonces de la forma:

$$S(u, v) = \sum_{i=0}^{n}\sum_{j=0}^{m} P_{i,j}\, B_i^n(u)\, B_j^m(v)$$

se denomina superficie ***producto tensorial*** y podemos describirla como el conjunto de puntos generado cuando una curva barre el espacio a la vez que se deforma. Es decir, agrupando se tiene:

$$S(u, v) = \sum_{i=0}^{n} \left(\sum_{j=0}^{m} P_{i,j}\, B_j^m(v)\right) B_i^n(u) = \sum_{i=0}^{n} C(v) B_i^n(u)$$

Entonces la superficie se puede considerar como una curva polinomial en la variable ***u***, donde los coeficientes están controlados por otra curva polinomial, esta vez en la variable ***v***. Expresando la ecuación anterior en forma matricial se tiene:

$$S(u, v) = \begin{bmatrix} B_0(u) & B_1(u) & \cdots & B_n(u) \end{bmatrix} \begin{bmatrix} P_{00} & P_{01} & \cdots & P_{0m} \\ P_{10} & P_{11} & \cdots & P_{1m} \\ \cdots & \cdots & \cdots & \cdots \\ P_{n0} & P_{n1} & \cdots & P_{nm} \end{bmatrix} \begin{bmatrix} B_0(v) \\ B_1(v) \\ \cdots \\ B_m(v) \end{bmatrix}$$

Tomando la definición de la ecuación de la superficie paramétrica, entonces denominaremos superficies de Bézier o Parches de Bézier a una ***superficie paramétrica producto tensorial*** que es la combinación lineal de funciones que son el producto tensorial de dos bases univariadas, las cuales son los polinomios de Bernstein definidos en [0, 1].

$$B_i^n(u) = \binom{n}{i}(1-u)^{n-i}u^i$$

$$B_j^m(v) = \binom{m}{j}(1-v)^{n-i}v^j$$

Una superficie de Bézier en general se expresa:

$$S(u, v) = \sum_{i=0}^{n}\sum_{j=0}^{m} P_{i,j}\, B_i^n(u)\, B_j^m(v)$$

donde los coeficientes P_{ij} son los puntos o vértices de control y estos constituyen una malla tridimensional que denominaremos ***red de control*** que asume la misma relación funcional que el polígono de control en las curvas de Bézier.

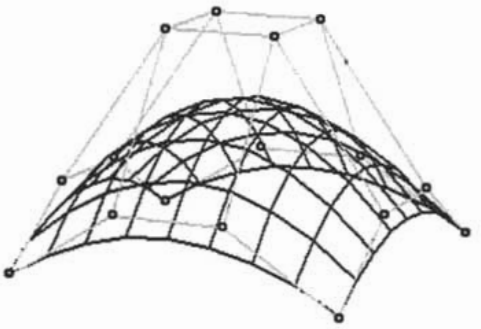


Fig. 3.18 Parche de Bézier.

Desarrollando una superficie de Bézier para n = m =1 tenemos:

$$S(u, v) = \sum_{i=0}^{1}\sum_{j=0}^{1} P_{i,j}\, B_i^1(u)\, B_j^1(v)$$

$$= \sum_{i=0}^{1} B_i^1(u)\,(\, P_{i,0}\, B_0^1(v) + P_{i,1}\, B_1^1(v)\,)$$

$$= P_{0,0}B_0^1(u)B_0^1(v) + P_{0,1}B_0^1(u)B_1^1(v) + P_{1,0}B_1^1(u)B_0^1(v) + P_{1,1}B_1^1(u)B_1^1(v)$$

desarrollando los polinomios de Bernstein:

$$B_0^1(u) = (1 - u) \qquad B_1^1(u) = u$$

$$B_0^1(v) = (1 - v) \qquad B_1^1(v) = v$$

Sustituyendo se tiene:

$$S(u, v) = P_{0,0}(1 - u)(1 - v) + P_{0,1}(1 - u)v + P_{1,0}u(1 - v) + P_{1,1}uv$$

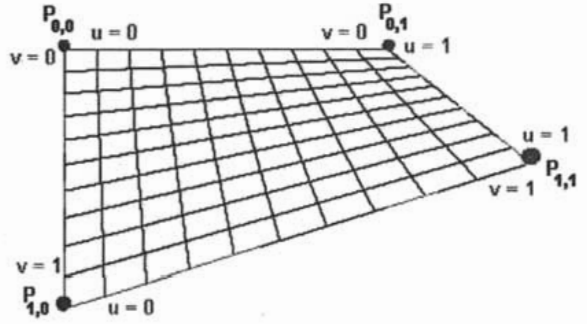


Fig. 3.19 Superficie de Bézier con cuatro puntos de control, n=1 y m=1.

Se observa que la superficie obtenida interpola los cuatro puntos de control y, análogamente a las curvas de Bézier, se puede notar las siguientes características:

$$S(0,0) = P_{0,0}$$
$$S(0,1) = P_{0,1}$$
$$S(1,0) = P_{1,0}$$
$$S(1,1) = P_{1,1}$$

Desarrollando para n=2 y m=2 se tiene:

$$S(u, v) = \sum_{i=0}^{2}\sum_{j=0}^{2} P_{i,j}\, B_i^2(u)\, B_j^2(v)$$

$$= \sum_{i=0}^{2} B_i^2(u)\ (\ P_{i,0}\, B_0^2(v) + P_{i,1}\, B_1^2(v) + P_{i,2}\, B_2^2(v)\ )$$

$$= P_{0,0}B_0^2(u)B_0^2(v) + P_{0,1}B_0^2(u)B_1^2(v) + P_{0,2}B_0^2(u)B_2^2(v) +$$
$$P_{1,0}B_1^2(u)B_0^2(v) + P_{1,1}B_1^2(u)\,B_1^2(v) + P_{1,2}B_1^2(u)B_2^2(v) +$$
$$P_{2,0}B_2^2(u)B_0^2(v) + P_{2,1}B_2^2(u)B_1^2(v) + P_{2,2}B_2^2(u)B_2^2(v)$$

desarrollando los polinomios de bezier:

$$B_0^2(u) = (1-u)^2 \qquad B_1^2(u) = 2u(1-u) \qquad B_2^2(u) = u^2$$
$$B_0^2(v) = (1-v)^2 \qquad B_1^2(v) = 2v(1-v) \qquad B_2^2(v) = v^2$$

Sustituyendo se tiene:

$$S(u, v) = P_{0,0}(1-u)^2(1-v)^2 + P_{0,1}(1-u)^2 2v(1-v) + P_{0,2}(1-u)^2 v^2 +$$
$$P_{1,0}\,2u(1-u)\,(1-v)^2 + P_{1,1}2u(1-u)\,2v(1-v) + P_{1,2}2u(1-u)\,v^2 +$$
$$P_{2,0}\,u^2(1-v)^2 + P_{2,1}\,u^2 2v(1-v) + P_{2,2}\,u^2 v^2$$

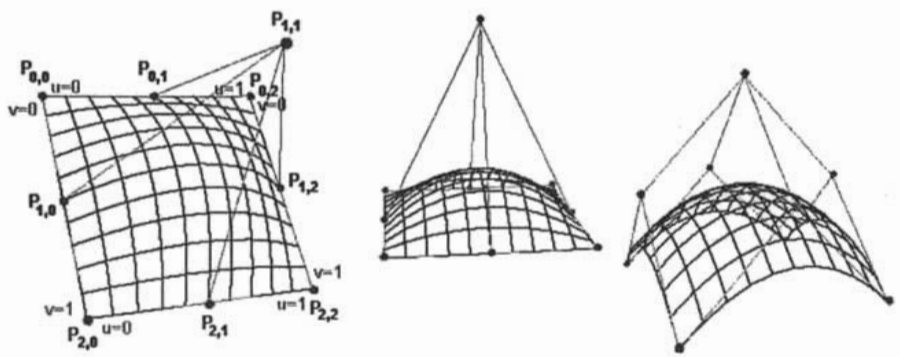


Fig. 3.20 Superficies de Bézier con cuatro puntos de control, n=2 y m=2.

Se observa que la superficie obtenida interpola las cuatro puntas de la malla de control y solo aproxima al resto de vértices, y presenta las mismas características:

$$S(0,0) = P_{0,0}$$
$$S(0,1) = P_{0,2}$$
$$S(1,0) = P_{2,0}$$
$$S(1,1) = P_{2,2}$$

Finalmente desarrollando para n=3 y m=3 se tiene:

$$S(u, v) = \sum_{i=0}^{3}\sum_{j=0}^{3} P_{i,j}\, B_i^3(u)\, B_j^3(v)$$

$$= \sum_{i=0}^{3} B_i^3(u)\,(\, P_{i,0}\, B_0^3(v) + P_{i,1}\, B_1^3(v) + P_{i,2}\, B_2^3(v) + P_{i,3}\, B_3^3(v)\,)$$

$$= P_{0,0}B_0^3(u)B_0^3(v) + P_{0,1}B_0^3(u)B_1^3(v) + P_{0,2}B_0^3(u)B_2^3(v) + P_{0,3}B_0^3(u)B_3^3(v) +$$
$$P_{1,0}B_1^3(u)B_0^3(v) + P_{1,1}B_1^3(u)B_1^3(v) + P_{1,2}B_1^3(u)B_2^3(v) + P_{1,3}B_1^3(u)B_3^3(v) +$$
$$P_{2,0}B_2^3(u)B_0^3(v) + P_{2,1}B_2^3(u)B_1^3(v) + P_{2,2}B_2^3(u)B_2^3(v) + P_{2,3}B_2^3(u)B_3^3(v) +$$
$$P_{3,0}B_3^3(u)B_0^3(v) + P_{3,1}B_3^3(u)B_1^3(v) + P_{3,2}B_3^3(u)B_2^3(v) + P_{3,3}B_3^3(u)B_3^3(v)$$

desarrollando los polinomios de bezier:

$B_0^3(u) = (1-u)^3$ $\quad B_1^3(u) = 3u(1-u)^2$ $\quad B_2^3(u) = 3u^2(1-u)$ $\quad B_3^3(u) = u^3$

$B_0^3(v) = (1-v)^3$ $\quad B_1^3(v) = 3v(1-v)^2$ $\quad B_2^3(v) = 3v^2(1-v)$ $\quad B_3^3(v) = v^3$

Sustituyendo se tiene:

$S(u, v) =$

$$P_{0,0}(1-u)^3(1-v)^3 + P_{0,1}(1-u)^3 3v(1-v)^2 + P_{0,2}(1-u)^3 3v^2(1-v) + P_{0,3}(1-u)^3 v^3 +$$
$$P_{1,0}3u(1-u)^2(1-v)^3 + P_{1,1}3u(1-u)^2 3v(1-v)^2 + P_{1,2}3u(1-u)^2 3v^2(1-v) + P_{1,3}3u(1-u)^2 v^3 +$$
$$P_{2,0}3u^2(1-u)(1-v)^3 + P_{2,1}3u^2(1-u)3v(1-v)^2 + P_{2,2}3u^2(1-u)\,3v^2(1-v) + P_{2,3}3u^2(1-u)v^3 +$$
$$P_{3,0}u^3(1-v)^3 + P_{2,1}u^3 3v(1-v)^2 + P_{3,2}u^3 3v^2(1-v) + P_{3,3}u^3 v^3$$

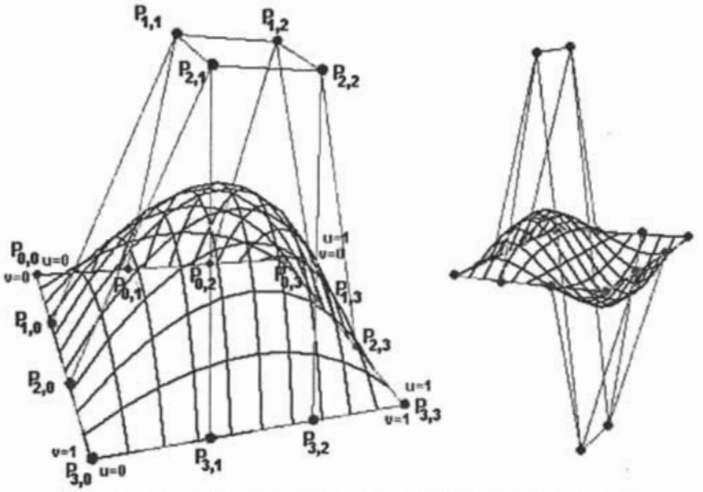


Fig. 3.21 Superficies de Bézier con cuatro puntos de control, n=3 y m=3.

Se observa que:

$$S(0,0) = P_{0,0}$$
$$S(0,1) = P_{0,3}$$
$$S(1,0) = P_{3,0}$$
$$S(1,1) = P_{3,3}$$

Aunque se ha desarrollado solamente Superficies de Bézier cuadradas, donde n = m, en la práctica el modelo permite implementar superficies con valores de n y m arbitrarios ($n \neq m$), sin embargo por cuestiones de diseño y facilidad de manejo se prefieren usar con n = m y además optar por n=m=3, ya que el cálculo de n>3 implica mayores requerimientos de recursos de memoria, así como mayor tiempo de procesador al calcular los puntos sobre la superficie.

III.2.7 Propiedades Básicas de una Superficie de Bézier.

Algunas de las propiedades de las superficies de Bézier son análogas a las propiedades de las curvas de Bézier, y son:

- Interpolación de los vértices de la malla de control:

$$S(0, 0) = P_{00}$$
$$S(0, 1) = P_{0m}$$
$$S(1, 0) = P_{n0}$$
$$S(1, 1) = P_{nm}$$

- Restricción a la envolvente convexa. Dado que :

$$\sum_{i=0}^{n}\sum_{j=0}^{m} B_i^n(u)\, B_j^m(v)$$

$$= \sum_{i=0}^{n} \left(\sum_{j=0}^{m} B_j^m(v)\right) B_i^n(u)$$

$$= \sum_{i=0}^{n} (1)\, B_i^n(u) = 1$$

y

$$B_i^n(u)\, B_j^m(v) \geq 0$$

por encontrarse en el intervalo [0,1], cualquier punto de la superficie es una combinación lineal convexa de los puntos de control $P_{i,j}$. ([4])

- Invarianza afín. A igual de las curvas, las superficies de Bézier admiten transformaciones afines de sus vértices de control sin que la curva desarrollada a partir de estos cambie. Es decir, si se desea aplicar una transformación a la superficie basta con aplicar esta transformación (translación, rotación, escalación, etc.) a los vértices de la malla de control y a partir de estos transformados reconstruir la superficie.
- Control Seudolocal. De la misma forma que ocurre en las curvas de Bézier, la modificación de un vértice de control, aunque afecta a toda la superficie, es más notoria su influencia en las proximidades del punto modificado.

  Sin embargo existe una característica que disminuye la influencia de cada vértice de control, en comparación con el comportamiento que ocurría en la curva de Bézier.

  Si recordamos el comportamiento de los polinomios de Bernstein veremos que existen dos características importantes:

  1. Los polinomios $B_i^n(u)$ solo se desarrollan en el intervalo [0, 1].
  2. Los únicos polinomios que tienen valores iguales a 1 son:

     $B_0^n(u)$ en $u = 0$

     y

     $B_n^n(u)$ en $u = 1$

     el resto de los polinomios tienen valores menores a 1.

[4] Curvas y Superficies parta modelado geométrico. Juan Manuel CorderoValle. Pag. 397.

Esto implica la primera propiedad, que la superficie pase por los 4 vértices extremos de la malla de control. Pero el resto de la superficie es el resultado del producto de las funciones base de Bernstein las cuales habíamos hecho notar que tienen valores menores a 1. La implicación de que el producto de dos escalares menores a 1 tiende a obtener un valor menor al escalar con valor máximo del cual se obtuvo, nos indica que cualquier vértice de la malla de control tendrá una influencia menor que la que tenia los vértices en el polígono de control de la curva, y por lo tanto se requerirá de un desplazamiento mayor para hacer más notorio la influencia sobre la superficie.

En el uso de superficies de Bézier, se vuelve complicado la implementación de continuidad C^1, por lo que suele ser suficiente el uso de ***continuidad geométrica o visual de orden uno*** G^1, el cual ya hemos visto que es suficiente con hacer colineales los vértices involucrados en la unión.

III.2.8 Diseño con Superficies de Bézier.

La última propiedad mencionada puede resultar un inconveniente a la hora de modelar y presenta limitantes en el diseño. Ya que no permitiría realizar modelos muy complicados con sólo una superficie de Bézier.

Es por esto que la solución es un método de modelado que se basa en la construcción de una superficie a trozos, es decir compuesta por varios superficies básicas o simples de Bézier (con n=m=3). Aprovechando una de sus características básicas que indica que una superficie de Bézier siempre pasa por sus puntos extremos de la malla de control.

Mejor aún, si hacemos coincidir dos superficies de Bézier en sus vértices de todo un lado de la malla de control, $P_{0,0} = Q_{3,0}$, $P_{0,1} = Q_{3,1}$, $P_{0,2} = Q_{3,2}$, $P_{0,3} = Q_{3,3}$, podemos afirmar que la curva que pasa por el conjunto de puntos $P_{0,0}$, $P_{0,1}$, $P_{0,2}$, $P_{0,3}$, es la misma curva que pasa por los puntos $Q_{3,0}$, $Q_{3,1}$, $Q_{3,2}$, $Q_{3,3}$, con lo cual quedaría asegurado que existe continuidad en la superficie sin "hoyos" en ella, es decir si aberturas entre las uniones de los lados de las superficies.

Por eso el uso común de estas superficies es en base al *embaldosamiento* de pequeñas superficies para generar un modelo más complicado, por eso se le denominan a estas superficies "***Parches de Bézier***". Por que se asemeja a "remendar" con pequeños ***parches*** un objeto o modelo.

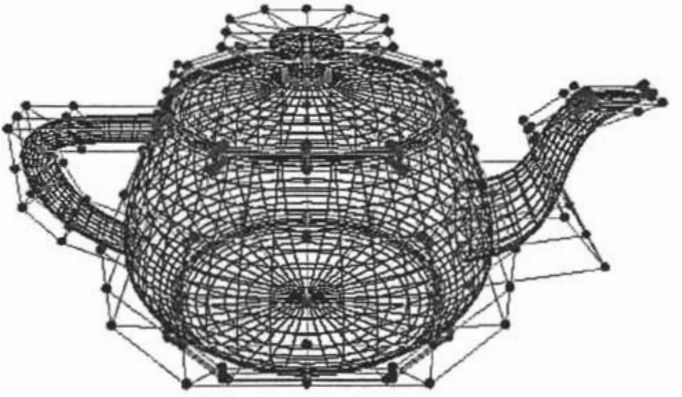


Fig. 3.22 Modelo a base de parches de bezier.

III.3 BLOBS

Los ***blobs*** son una de las herramientas más usadas para modelar objetos orgánicos.
Consiste en la generación de una superficie a partir de un conjunto de componentes. Cada componente posee características propias, un campo escalar asociado entre ellas, y la superficie es formada a partir de la combinación de todos los componentes.
La forma final de un objeto modelado por ***blobs*** es calculada basándose en la combinación y suma de los campos escalares de cada componente.
La primera mención de la incorporación del modelo de ***blobs*** en modelado se le atribuye a James F. Blinn en 1981, el cual para crear una animación de una duplicación de una secuencia de ADN para el programa de televisión COSMOS de Carl Sagan desarrolló un nuevo algoritmo para poder representar simulaciones de campos de densidad de electrones de estructuras moleculares, a las cuales Blinn se refiere como unas "gotas pegándose y despegándose". A este modelo le denomino "***blobby model***".
Sin embargo en forma casi simultánea se estaba creando un modelo similar por Koichi Omura de la Universidad de Osaka. Desarrollo un modelo el cual llamó "***metaballs***". Sin embargo el modelo usado sólo era una aproximación a un campo Gausiano. El modelo fue implementado en el sistema de cómputo paralelo LINKS-1, en el cual Yoichiro Kawaguchi generó algunas animaciones que fueron presentadas en muestras SIGGRAPH, la cual le otorgo renombre al modelo.

III.3.1 Definición de blob o metaball.

Un modelo de ***Blobs*** o ***metaballs*** se puede definir como una isosuperficie formada mediante la combinación de un conjunto de cargas puntuales llamadas ***componentes***.
Un ***componente***, que tiene asociado un campo escalar y que en unión con otros ***componentes*** forman un objeto grafico, puede ser bidimensional o tridimensional.
No se define en si como un objeto grafico, como lo es una esfera o un cubo, ya que estos conllevan dentro de su definición características propias que permiten generar su forma, o mejor dicho, obtener su superficie a partir de ellos mismos. Más bien un ***componente*** es un objeto básico, que no posee forma o definición grafica desde el

punto de vista del modelado, pero que forma parte de una estructura más compleja que dará como resultado un objeto tridimensional o una superficie dada.

El ***componente*** solo contiene información característica del ***blob*** y a partir de ésta se genera mediante un algoritmo de combinación, el modelo final.

Se puede considerar al ***componente*** como una primitiva puntual que posee asociado una magnitud de un campo de densidad especifico para todas las primitivas. Tal campo de densidad esta dado mediante una función matemática que da como resultado un valor escalar para el algoritmo de combinación.

Por lo dicho anteriormente podemos decir que las propiedades que poseen los ***componentes***, como objetos básicos, son:

- Posición de cada componente básico, es decir, cada ***componente*** posee un centro desde el cual irradia su campo de densidad.
- Una magnitud o radio, el cual afectara la intensidad del campo de densidad asociado.

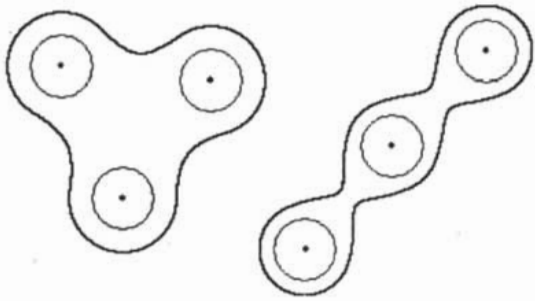


Fig.3.23 Composición de blobs y su isosuperficie.

El modelo final, resultado de la combinación de cada componente estará formada por una isosuperficie.

III.3.2 Definición de Isosuperficie.

Podemos definir a una ***isosuperficie*** como aquella superficie en la que todos sus puntos tienen el mismo valor. "Iso" significa "igual", sin embargo es obvio que en una superficie ubicada en el espacio, cada punto que la forma tiene diferentes valores para

sus coordenadas, por lo tanto resultaría absurda la definición anterior. Sin embargo se puede explicar de la siguiente forma: una isosuperficie esta formada por cada punto en el espacio que al ser procesado por una fórmula matemática genere un mismo valor específico para todos ellos.

Pongamos como ejemplo el campo eléctrico y potencial de una carga puntual. El concepto de ***superficie equipotencial*** se entiende como "cualquier superficie que contiene un distribución continua de puntos que tienen el mismo potencial". [5]

El potencial de una carga puntual Q en un punto P a una distancia r de la carga esta dada por:

$$V = k\frac{q}{r}$$

"De aquí se puede observar que V es constante sobre una superficie esférica de radio *r*. Por lo que podemos concluir que las ***superficies equipotenciales*** (superficies en las cuales V permanece constante) para una carga aislada, constan en una familia de esferas concéntricas con la carga".[6]

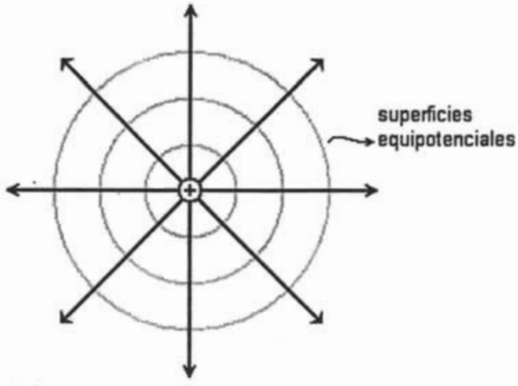


Fig. 3.24 Imagen de las líneas de fuerza de una carga puntual. Las superficies equipotenciales son superficies esféricas concéntricas.

"El potencial eléctrico de dos o mas cargas puntuales se obtiene aplicando el principio de superposición. Es decir, el potencial total en un punto dado P debido a varias cargas puntuales es la suma de los potenciales debidos a cada carga individual. Para un conjunto de cargas, se puede escribir el potencial total en P en la forma:

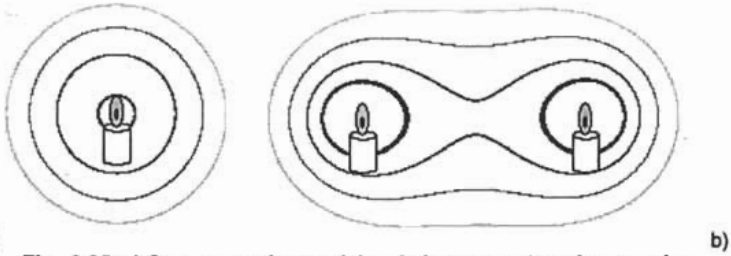
[5] Fisica -incluye fisica moderna-.Tomo II. Raymond A. Serway, McGRAW-HILL Pág. 698.
[6] Fisica -incluye fisica moderna-.Tomo II. Raymond A. Serway, McGRAW-HILL Pág. 700-701.

$$V = k \sum_i \frac{q_i}{r_i}$$

donde r_i es la distancia del punto P a la carga q_i. Nótese que es una suma algebraica de escalares". [7]

Otro ejemplo de cómo interpretar una ***isosuperficie***, quizá un poco más empírico, es considerar una habitación vacía donde tenemos colocada un foco o una vela, la cual emite calor en forma radial, hacia todas partes de la estancia. Considerando el objeto como un punto que irradia calor se puede considerar que el centro del foco o la vela es el punto en donde el calor es mas intenso y que conforme se aleja del centro del punto de calor, este disminuye.

Si tomáramos un termómetro y midiéramos la temperatura En cada punto de la estancia obtendríamos una lectura de la temperatura, una ***isosuperficie*** seria todos aquellos puntos en el espacio en donde el termómetro marcara la misma temperatura.



a) b)

Fig. 3.25 a) Campos equipotenciales de la temperatura de una vela; b) temperatura combinada de dos puntos de calor.

Si colocamos dentro de la misma habitación otro punto de calor, foco o vela, entonces la temperatura resultante en el espacio que las rodea sería la combinación de la irradiación de ambos puntos de calor, es decir, la suma de los campos de densidad de temperatura de ambos puntos de calor. Por lo tanto la ***isosuperficie*** estará formada por aquellos puntos de espacio que, sumados los campos de ambos objetos, tengan una misma temperatura.

Se puede observar que las cargas puntuales que hemos descrito emiten su campo de densidad de manera radial, por lo tanto las superficies equipotenciales que produce son generalmente esferas concéntricas al centro de la carga. Por lo tanto si eligiéramos un valor específico para generar la ***isosuperficie*** obtendríamos una superficie esférica.

[7] Idem.

Por eso dentro del modelado con ***blobs*** y ***metaballs*** se usan como elementos primitivas o componentes las esferas como medio de edición y manipulación del modelo, ya que facilita el entendimiento intuitivo de el objeto a modelar.
Una vez que ya hemos visto en que consiste una ***isosuperficie*** entonces definiremos en que se basa el algoritmo de modelado con ***blobs*** o ***metaballs***.

III.3.3 Modelo de Blob o Metaballs.

Un modelo con ***blobs*** o ***metaballs*** es una ***isosuperficie*** resultante de la combinación de pequeñas primitivas llamadas componentes que tienen asociado un campo escalar las cuales mediante una función de combinación generan una superficie que da origen a un modelo con características orgánicas.
Un modelo de blobs o metaballs posee las siguientes propiedades para poder generar un objeto:

1. Un conjunto de primitivas o componentes que darán forma al objeto a modelar, y que a su vez contienen las siguientes características:
    - Posición, es decir el centro donde se encontrara ubicada la primitiva.
    - Un escalar que definirá la intensidad de su campo escalar.
2. Propiedades globales que se definen el objeto a modelar, y son:
    - Una función global la cual define la manera en que se distribuye el potencial o campo de densidad. La cual se aplica a todas las primitivas o componentes del modelo.
    - Un valor específico o de umbral que permitirá generar la isosuperficie.

El uso de varias primitivas o componentes implica que sus respectivos campos de densidad se sumen para formar la ***isosuperficie***.

$$D(x, y, z) = \sum_{i}^{n} D_i(x, y, z)$$

Donde D_i es valor de densidad para el componente i-ésimo en la posición (x, y, z), y D es la densidad total calculada. Sin embargo, como ya se ha mencionado, la finalidad de este modelo es obtener una ***isosuperficie*** con un valor específico de umbral o de densidad T>0, por lo cual la ecuación se puede expresar como una función implícita:

$$D(x, y, z) = T$$

Sólo quedaría por definir la ecuación del campo escalar.

Básicamente existen tres métodos de modelado basados en campos escalares, los cuales generalmente difieren sólo en la forma en que describen la función de densidad. Y son:

1. Blobby Model
2. Soft Objects u objetos suves.
3. Metaballs.

III.3.4 Blobby Model

Es considerado el primer modelo de ***blobs***, fue diseñado e implementado por James F. Blinn y lo que pretendía modelar eran orbitales moleculares. "Por ejemplo, una forma molecular se puede describir como esférica en aislamiento, pero esta forma cambia cuando la molécula se acerca a otra molécula. Esta distorsión de la nube de densidad de electrones se debe al *enlace* que ocurre entre dos moléculas".[8] Es decir, ocurren "efectos de estiramiento, división y contracción en las formas moleculares cuando dos moléculas se separan".[9] Este comportamiento se puede modelar mediante la combinación de funciones gaussianas de densidad o "bultos". Utilizó como función de campo la de la densidad de un átomo de hidrogeno, que tiene una distribución gaussiana y describe el potencial en función del radio.

$$D_i(x, y, z) = b_i \exp(-a_i r_i^2)$$

En donde a_i (> 0) afecta a la caída exponencial del campo, se identifica como la desviación estándar de la curva, b_i es la magnitud de la curva o intensidad del campo escalar, y r_i es la distancia de el i-ésimo componente al punto (x, y, z).

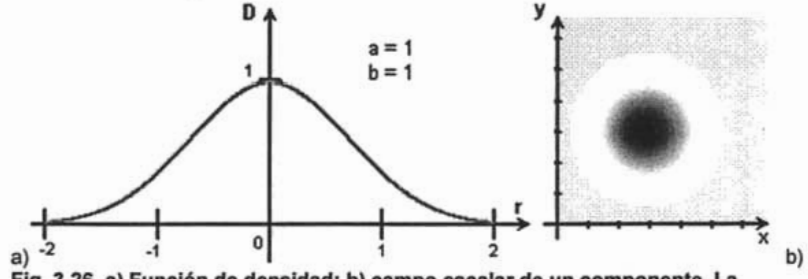
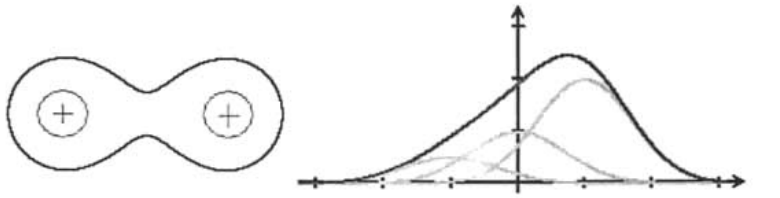


Fig. 3.26 a) Función de densidad; b) campo escalar de un componente. La parte más obscura es la mas intensa.

[8] Gráficas por computadora. Donald Hearn. Pag. 330.
[9] Idem.

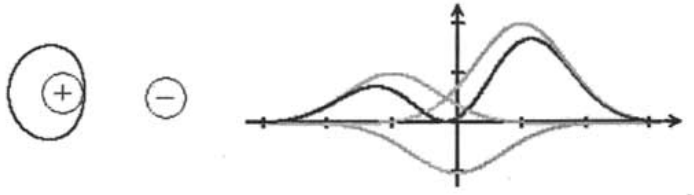
Tanto a_i como b_i se utilizan para ajustar la cantidad de abultamiento de los objetos individuales.



a) b)

Fig. 3.27 a) Isosuperficie creada a partir de 2 componentes; b) Suma de 3 campos de densidad.

Se puede notar que al ser sumados los campos escalares de todas las primitivas o componentes involucrados en la generación del objeto se generan superficies suaves y hay un abultamiento en las depresiones entre funciones atenuando la discontinuidad entre componentes. Si se emplearan valores negativos de intensidad del campo escalar, b_i, se restaría con el resto de las primitivas en la función de combinación, para producir hendiduras o erosiones en lugar de abultamientos sobre el campo escalar de los componentes cercanos.



a) b)

Fig. 3.28 a) Isosuperficie con un componente con campo negativo; b) Suma de 3 campos de densidad, uno de ellos negativo.

Uno de los problemas inherentes a este método es que la función generadora del campo de densidad, a pesar de que la caída de potencial es muy rápida, conforme se incrementa la distancia, el campo de densidad es exponencial, es decir, llega a ser cero en infinito, por lo tanto su influencia se extiende por prácticamente todo el espacio, aunque con una intensidad muy pequeña. Lo que implica añadir condiciones de distancia para dejar de tomar en cuenta la influencia de una primitiva. Otra desventaja es que el campo, al decaer muy rápido, pierde influencia significativa en una distancia pequeña para con las demás primitivas o componentes. Lo cual puede en algunos casos dificultar el diseño y la manipulación sobre un modelo de ***blobs***.

III.3.5 Metaballs.

Es otro método para generar objetos abultados utilizando funciones de densidad cuyo valor mínimo es cero en un intervalo finito, en lugar de uno exponencial que se extiende al infinito, lo que hace computacionalmente costoso si se hace un cálculo completo para cada primitiva.

El método de ***metaballs*** describe objetos compuestos como combinaciones de funciones cuadráticas de densidad que son:

$$D(r) = \begin{cases} b\left(1-3\frac{r^2}{d^2}\right), & \text{si } 0 < r \le \frac{d}{3} \\ \frac{3}{2}b\left(1-\frac{r}{d}\right)^2, & \text{si } \frac{d}{3} < r \le d \\ 0, & \text{si } r > d \end{cases}$$

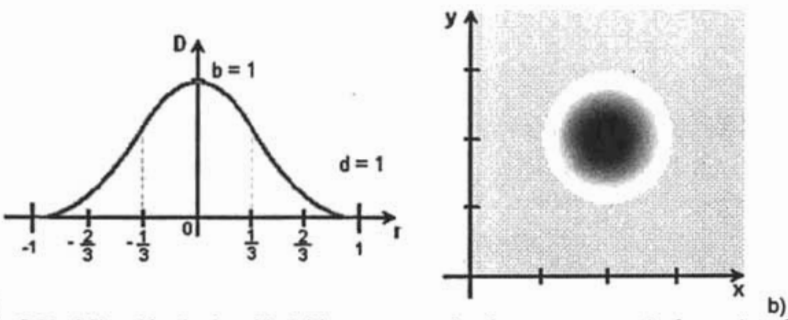


Fig. 3.29 a) Función de densidad; b) campo escalar de un componente. La parte más obscura es la mas intensa.

Donde *b* es el factor de escala para el campo escalar, y *d* es un control para un máximo de distancia en donde una primitiva o componente contribuye con su campo de densidad a otros componentes.

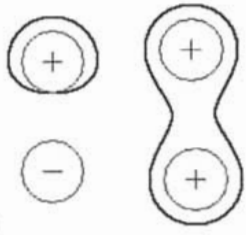


Fig. 3.30 a) Metaballs con campo negativo; b) Metaballs con campo positivo.

III.3.6 Objetos Suaves.

Este método es atribuido al trabajo de los hermanos Wyvill, la cual propone un campo de densidad modificado del método de Blinn, haciendo que la función generadora del campo dependiese sólo de la distancia del punto a la primitiva.

Su función fue construida para que presentara las siguientes propiedades:

- El alcance del campo se hace a una distancia finita de la primitiva o componente, es decir, un radio de influencia R.
- La caída del campo dentro de su zona de influencia es más suave que la de Blinn.

$$D(r) = \begin{cases} 1 - \frac{22r^2}{9R^2} + \frac{17r^4}{9R^4} - \frac{4r^6}{9R^6}, & \text{si } 0 < r \le R \\ 0, & \text{si } r < R \end{cases}$$

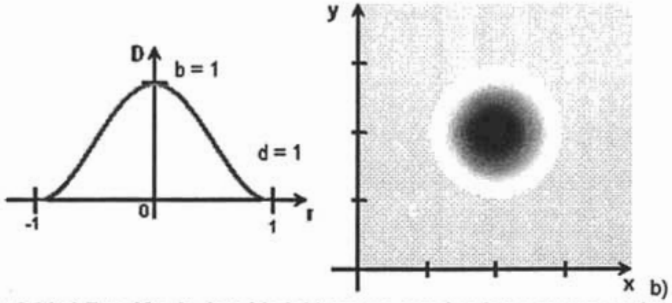


Fig. 3.31 a) Función de densidad; b) campo escalar de un componente. La parte más obscura es la más intensa.

Es notable que esta función de densidad genera una curva más suave. Por lo tanto puede generar una transición menos brusca entre componentes.

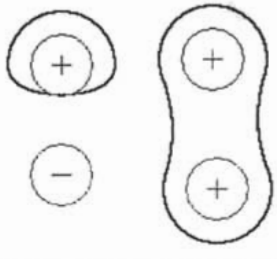


Fig. 3.32 a) Objetos suaves con campo negativo; b) Objetos suaves con campo positivo.

Donde ***r*** es la distancia de algún punto en el espacia al componente. Es posible agregar una característica más que permita escalar la intensidad de su campo de densidad, lo cual es posible solamente multiplicando el valor de densidad por un escalar ***b***.

Básicamente el modelado por ***blobs*** es el mismo método para todos los casos anteriores vistos, solamente cambia la forma en que cada uno describe la función de densidad. Lo cual posibilita usar cualquier función personal que se comporte de manera similar a las anteriores vistas. Pero de los casos anteriores, se puede utilizar dependiendo el tipo de comportamiento que se desee de los campos de densidad.

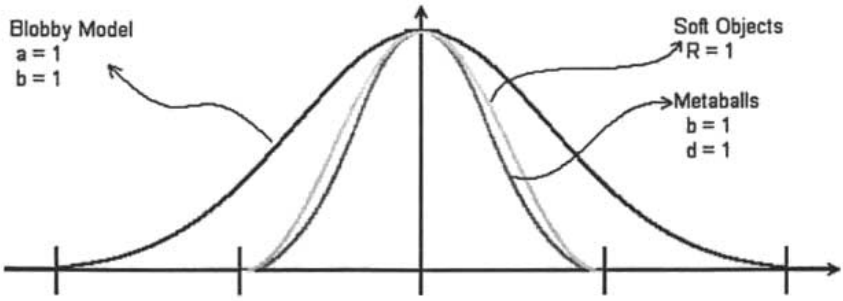


Fig. 3.33 Comparación de los campos de densidad vistos.

Queda claro entonces que los parámetros para modificar la función de densidad y por lo tanto la isosuperficie generada son, la intensidad o magnitud del campo escalar (***b***) y la amplitud con que se distribuye, que en el método de Blinn es la desviación estándar, por ser una distribución normal, y en las demás modificando el parámetro de radio de influencia (***R***, ***d***) del campo escalar. Además que es obvio què el parámetro T o el valor de umbral de la isosuperficie también se define como un parámetro más, ya que las funciones escalares tienen su valor máximo de densidad en el centro puntual del componente y disminuye en forma radial conforme se aleja de este punto, por lo tanto el valor de T generalmente variará en el rango de [0,1] donde un valor cercano a 1 proporcionara puntos de la isosuperficie más cercanos a los componentes, y por lo tanto el tamaño de objeto a modelar será pequeño. De forma análoga, si elegimos un valor de umbral T de un valor pequeño, entonces obtendremos objetos más grandes

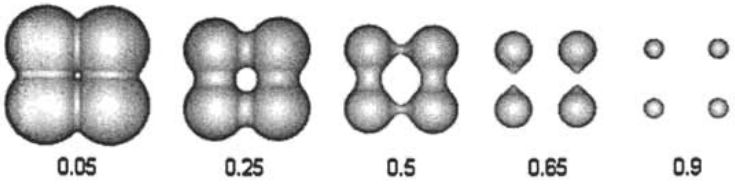


Fig. 3.34 Composición de blobs variando el valor de umbral.

III.3.7 Problemas de Representación.

Una vez definido el algoritmo general de los modelos de ***blobs*** se observa que el método permite encontrar un valor de densidad para cada punto en el espacio, aunque para formar la isosuperficie sólo se tomen en cuanta los valores con densidad igual a un cierto valor de umbral.

Esto da pie a encontrar una manera de representar visualmente tal isosuperficie, de manera de poder representar el modelo diseñado.

Un problema intrínseco a este tipo de modelos es que no sólo se forman de la función de densidad, la cual es una ecuación la cual se puede resolver para el valor de umbral, sin embargo ello implicaría utilizar algo muy cercano a una técnica de barrido, la cual tendría que "barrer" el área de visualización censando cuales puntos del área de visión son parte de la isosuperficie, y graficarlas. Lo cual no siempre es apto para las necesidades de modelado, por el consiguiente gasto en recursos de cómputo.

Generalmente se utilizan dos técnicas para visualizar isosuperficies, y son dos métodos totalmente opuestos desde el punto de vista del enfoque, y son generación por ***trazado de rayos*** y por generación de polígonos ***Marching Cubes***.

La generación por ***trazado de rayos*** nos ofrece una mejor definición de forma e imagen, ya que por la naturaleza de su funcionamiento que ofrece imágenes fotorrealistas. El trabajo consiste entonces en resolver la intersección de cada rayo proyectado en la escena con las fórmulas de densidad del modelo. Sin embargo este tipo de método no es rápido, ya que también basan su funcionamiento en un barrido del área de proyección obre la escena. Se podría optar como un método de imagen final, por su gran calidad de definición de imagen.

El otro método de presentación es quizás el más apropiado para generar una vista interactiva del modelo, visto desde el enfoque de manipulación y edición interactiva. Ya

que el método de ***Marching Cubes*** es un método de aproximación de la isosuperficie mediante la generación de polígonos que se ajusten a su forma.

III.3.8 Algoritmo Marching Cube.

Es un algoritmo presentado por W.E. Lorensen y H.E. Cline en 1987 y describe la aproximación de la isosuperficie por polígonos que se ajusten a ella. El resultado del algoritmo es una malla, generalmente no uniforme, de polígonos el cual permitirá modelar un sólido mediante técnicas de representación poligonal.

El algoritmo clásico comienza realizando una descomposición del espacio que contiene a la isosuperficie en cubos de un tamaño predeterminado (geometrías base), a estos cubos se suelen denominar ***voxel***. "Los elementos individuales de un espacio tridimensional se llaman **elementos de volumen** o ***voxels***"[10], algo análogo a los píxeles bidimensionales.

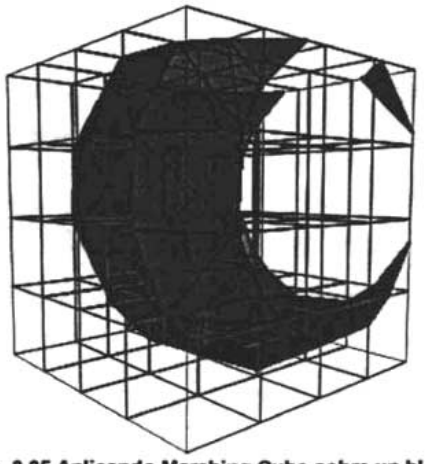


Fig. 3.35 Aplicando Marching Cube sobre un blob.

La situación de cada ***voxel*** puede tener tres casos:

1. Que el ***voxel*** este completamente fuera de la superficie.
2. Completamente dentro de la superficie
3. Parcialmente dentro, o sea intersectado por la superficie.

[10] Gráficas por computadora. Donald Hearn. Pag. 379.

Para saber esto se estudia el valor de densidad en los ocho vértices de cada ***voxel***. Si el valor de potencial en un vértice es menor al valor umbral predefinido para formar la isosuperficie, entonces podemos afirmar que ese vértice del ***voxel*** se encuentra afuera del sólido definido por la ***isosuperfcie***, o que esta por encima de la superficie. Si es mayor, entonces se entiende que el vértice se encuentra dentro de el sólido o que esta por debajo de la superficie a aproximar.

La isosuperficie corta aquellos ***voxels*** donde al menos un vértice este afuera de ella o al menos uno se encuentra dentro.

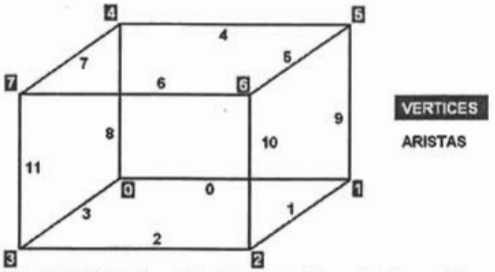


Fig. 3.36 Distribución de los índices de los vértices.

Se utiliza una ordenación de vértices y aristas que permitan identificar su situación con respecto a la ***isosuperficie***. Los ocho vértices que componen el cubo, V_0, ..., V_7, pueden ser representados como datos binarios, ya que para el primer paso del algoritmo sólo se necesita saber si el vértice esta dentro o fuera de la ***isosuperficie***, por lo tanto se puede asignar un '1' a aquellos vértices que se encuentren dentro de la superficie y un '0' a aquellos que se encuentren fuera de ella. Reacomodando estos valores, se puede tener la siguiente configuración:

	V_7	V_6	V_5	V_4	V_3	V_2	V_1	V_0
Con todos fuera:	0	0	0	0	0	0	0	0
Con todos dentro:	1	1	1	1	1	1	1	1
Con sólo V_5 fuera:	1	1	0	1	1	1	1	1

Esta configuración de valores nos da una palabra binaria de 8 bits, la cual traducida a número decimal se tiene los numero 0, 255 y 223 respectivamente. Es decir que existen 256 posibles casos de intersección de la isosuperficie y el ***voxel***.

Si un vértice se encuentra fuera de la superficie y sus vértices adyacentes se encuentran dentro, el cubo es cortado por la superficie. Eso quiere decir que la superficie debe de cortar a las aristas adyacentes al vértice que se encuentra fuera de ella, por lo tanto se puede encontrar un polígono que se aproxime a la superficie y que a su vez tenga como vértices los puntos donde ésta corta a dichas aristas.

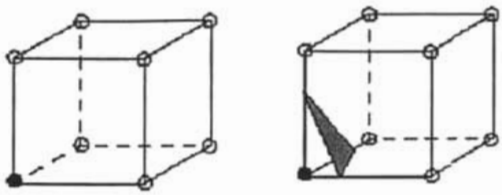


Fig. 3.37 Intersección de blob con el cubo.

Estas intersecciones se prefieren implementar triangulares para simplificar su manipulación.

De las 256 posibles combinaciones se puede reducir a 15 casos diferentes, ya que algunas combinaciones se repiten para cada vértice.

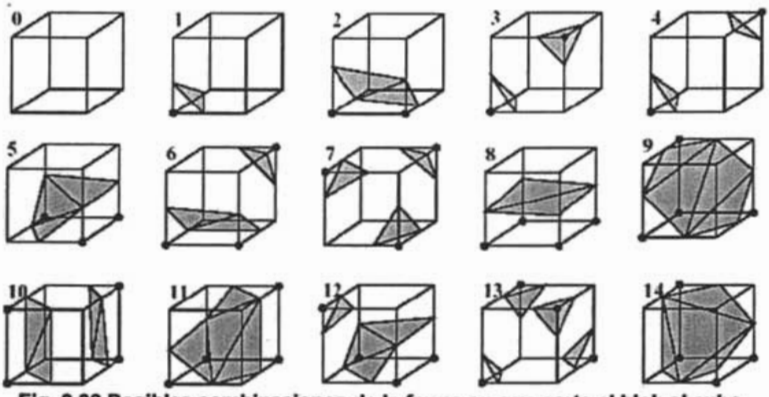


Fig. 3.38 Posibles combinaciones de la forma en que corta el blob al cubo.

Una vez que se ha identificado los vértices, y se ha formado la palabra binaria vista antes, se puede identificar, mediante una tabla predefinida, las aristas con las cuales se interfecta la superficie, para a continuación generar los triángulos correspondientes, para lo cual también es conveniente tener una tabla para generarlos de acuerdo a las aristas que se cortan.

$V_7V_6V_5V_4V_3V_2V_1V_0$	Aristas que cortan	Triángulos a formar
0 0 0 0 0 0 0 0	Ninguna	Ninguno
0 0 0 0 0 0 0 1	0, 3, 8	<0,8,3>
...	...	...
1 0 1 0 1 0 1 0	0, 1, 2, 3, 4, 5, 6, 7	<3, 6, 2>, <3, 7, 6>, <1, 5, 0>, <5, 4, 0>
...	...	...
1 1 1 1 1 1 1 0	0, 3, 8	<0,3,8>
1 1 1 1 1 1 1 1	Ninguna	Ninguna

Pero queda un problema por solucionar, que es precisamente encontrar el punto en que la superficie corta a la arista, para poder generar los vértices del triangulo.

Un método eficaz y rápido consiste en interpolar linealmente los valores de densidad que tienen los vértices que componen la arista.

Teniendo la ecuación paramétrica de la recta :

$$P = P_1 + (P_2 - P_1)\,u \quad ; \quad u \in [0, 1]$$

Variando el parámetro u en el rango de [0,1] se interpola el punto entre los vértices del segmento de recta, por lo tanto para cada valor de u en [0,1] se obtiene un punto dentro del segmento de recta. Si tomamos la arista a interpolar como el segmento de recta a interpolar, entonces solo tenemos que encontrar el valor de u para el cual se obtiene el punto donde la superficie corta a la arista.

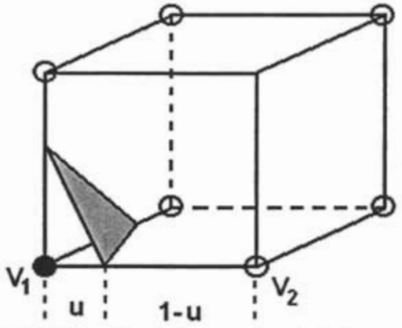


Fig. 3.39 Obtención de los puntos de intersección.

La obtenemos calculando la proporción de los valores de densidad de los vértices (V_1 y V_2) y el valor de umbral ***T*** que se esta buscando.

$$u = \frac{T - V_1}{V_2 - V_1}$$

Por lo tanto la ecuación de interpolación para encontrar el punto donde es cortada la arista es:

$$P = P_1 \frac{(V_1 - T)(P_2 - P_1)}{(V_1 - V_2)}$$

Otro problema de este método de visualización es el de elegir adecuadamente el tamaño del cubo o ***voxel***, ya que al representar la isosuperficie a partir de las muestras tomadas en los vértices de los ***voxels***, pueden aparecer problemas de aliasing, o perdida de información en la forma obtenida.

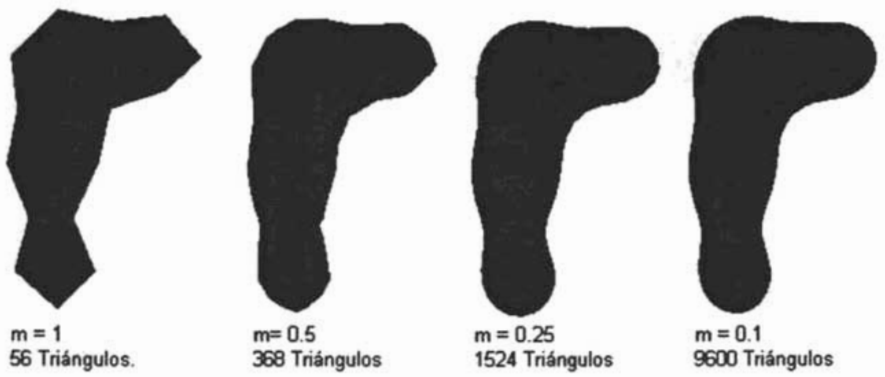


Fig. 3.40 Composición de blobs a diferentes resoluciones, variando la magnitud (m) del cubo.

Este método de visualización tiene ventajas como son la simplicidad del algoritmo y la facilidad de la implementación, además de que es posible aumentar la resolución de la malla de polígonos, pudiendo ajustar el grado del acabado de la superficie poliédrica.
Sin embargo algunas de las desventajas que tiene este método es que genera una malla no uniforme, y pudieran encontrarse problemas de desconexión en la malla poligonal, ocasionando hoyos en la superficie.
Aún así es un buen método de visualización, y muy conveniente, ya que el manejo de objetos tridimensionales tiende al uso de ambientes poligonizados, por lo tanto se pueden usar técnicas de presentación de polígonos, ocultación de partes ocultas, técnicas de sombreado (Gouraud, Pong), que permiten una buena presentación final del modelo.

III.3.9 Modelado usando Blobs

De acuerdo al algoritmo que se encuentra detrás del método de modelado, podemos entonces concluir que el diseño de objetos mediante el uso de ***blobs*** se basa en el posicionamiento de componentes en el espacio, y que mediante una la función de combinación se genera la superficie. Por lo tanto en único medio con el que puede interactuar el diseñador, son con los componentes, ubicándolos en el espacio de trabajo, y regulando los parámetros que regularan el campo de densidad del los métodos y el valor de umbral de la isosuperficie.

Uno de los aspectos de trabajar con ***blobs*** es que el diseño del objeto se vuelven intuitivos, es decir, la posición de cada ***blob*** en la escena influye en el objeto final, por lo tanto no hay reglas precisas que indiquen a que distancia se debe colocar cada componente de los demás para lograr un resultado específico sobre la superficie. Por lo cual el diseño se vuelva casi empírico, sin embargo las formas que se pueden obtener con respecto a otras técnicas de modelado, son de mejor calidad y mayor complejidad. Añadiendo que la modificación de un modelo se facilita y tiende a ser mas rápido.

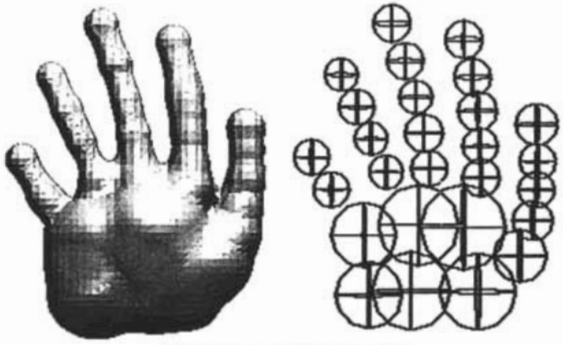


Fig. 3.41 Modelo de blobs.

CAPÍTULO IV

Desarrollo de las herramientas.

IV.1 Herramientas de Desarrollo.

Se eligió el lenguaje de programación C por ser prácticamente el lenguaje nativo del desarrollo en Linux y proporcionar una estabilidad lo suficientemente robusta para la generación y liberación de proyectos en la comunidad GNU.

La problemática de desarrollar una aplicación que permitiera de manera visual manipular de forma interactiva los elementos de modelado gráfico requería forzosamente trabajar sobre la interfaz de gráfica usuario (***GUI***) de alguno de los escritorios de Linux (X Window, KDE, Gnome). La existencia de librerías para el manejo de ventanas para cada uno de los escritorios facilitaba el desarrollo de manera decisiva. Uno de los escritorios que ha cobrado recientemente una importancia creciente es GNOME con su conjunto de librerías ***GTK+***.

Alrededor de esta biblioteca se ha desarrollado todo un conjunto de utilerías orientadas a impulsar el desarrollo en Linux. Un ejemplo es la herramienta ***GLADE***, que es una herramienta ***RAD*** que facilita y disminuye el tiempo y esfuerzo de desarrollo en un proyecto de programación.

Finalmente se utilizaron las funciones de la biblioteca ***OpenGL*** para Linux por ser un estándar para la programación y visualización de gráficos en 3D.

IV.1.1 GTK+

GTK significa ***GIMP Toolkit*** (Conjunto de herramientas GIMP) y es una biblioteca utilizada para desarrollar aplicaciones con interfaz gráfica de usuario (GUI, Graphical User Interface) en ambiente Linux, aunque recientemente se ha expandido para volverse en una librería multiplataforma[11].

GTK+ fue escrito originalmente por Peter Mattis, Spencer Kimball y Josh MacDonald, y tiene su origen con la aplicación GIMP (GNU Image Manipulation Program, Programa de Manipulación de Imágenes GNU), proyecto iniciado por los estudiantes de Berkeley Spencer Kimball y Peter Mattis, la cual en sus orígenes utilizaba la librería MOTIF para

[11] http://www.gtk.org/faq

generar su interfaz gráfica de usuario, sin embargo por problemas de licencia y distribución decidieron escribir sus propias librerías a las que llamaron ***GTK*** (Gimp Tool Kit) y ***GDK*** (Gimp Drawing Kit)[12]. Al principio solo intentaban generar librerías auxiliares que les facilitara el uso de GIMP, sin embargo con el paso del tiempo estas librerías se han convertido en un conjunto de herramientas de propósito general.

Actualmente **GTK+** esta disponible bajo la ***GNU LGPL*** (*GNU Library General Public License*, Licencia Pública General de Biblioteca GNU) y se compone básicamente de tres bibliotecas:

GLib La Biblioteca G. Una biblioteca con utilerías de propósito general, que no esta especificada para interfaces graficas de usuario. Contiene tipos de datos básicos, estructuras de datos útiles, macros, funciones de conversión de tipos, funciones de manejo de cadenas, manejo de archivos, etc.

GDK El conjunto de dibujo de ***GTK+***. Contiene funcionalidades para el manejo de ***widgets***[13] y funciones de dibujo se hacen a través de esta biblioteca.

GTK El conjunto de herramientas GIMP. Un conjunto de ***widgets*** para implementar la interfaz gráfica de usuario.

GTK+ esta escrita en lenguaje C e implementa características del estilo de orientación a objetos. Al igual que la mayor parte de las herramientas modernas para interfaces gráficas de usuario, ***GTK+*** es una herramienta conducida por sucesos. La interfaz de usuario en pantalla se construye mediante ***widgets*** (ventanas, botones, cuadros de texto, etiquetas, etc.) y se establecen una serie de *retrollamadas* para esos ***widgets*** con el fin de realizar el procesamiento basado en señales, que usualmente son eventos de ratón o teclado. ***GTK+*** invoca a ***GDK*** para todo lo que se relacione con la visualización de los ***widgets***. ***GDK*** es una API[14] dependiente de la plataforma y que se sitúa por encima de la API gráfica nativa (Xlib, Win32). "*Puesto que **GDK** esta situado como un nivel entre Xlib y **GTK+**, puede portarse **GTK+** a otros sistemas operativos con sólo cambiar la biblioteca **GDK***"[15]. GTK como GDK hacen uso de las funcionalidades que

[12] http://www.gimp.org/about/ancient_history.html
[13] Un ***widget*** es un control o componente en una interfaz grafica de usuario. *http://www.gtk.org/glossary.html*
[14] Application Programming Interface, Interfaz de programación de Aplicaciones.
[15] Desarrollo de aplicaciones Linux con GTK+ y GDK. Eric Harlow, pag. 28.

contiene la biblioteca GLib, y es posible utilizar GLib y GDK directamente sin pasar por la biblioteca GTK+.

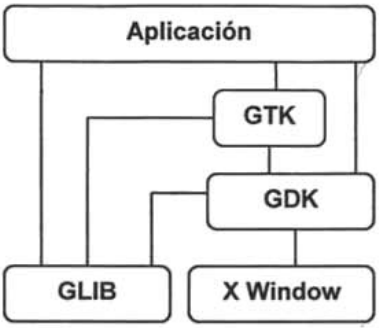


Fig. 4.1 Niveles de Aplicación de GTK

Para construir una aplicación ***GTK+***, cada archivo de la aplicación que utilice sus funciones o definiciones debe incluir el archivo "*gtk/gtk.h*" que es el archivo cabecera principal de ***GTK+***. Antes de invocar cualquier función de la biblioteca de ***GTK+*** es necesario inicializarla con una llamada a la función de **gtk_init** con los argumentos de la aplicación (argc, argv).

```
gtk_init( &argc, &argv )
```

Una vez que se ha inicializado ***GTK*** se puede proceder a llamar y crear la interfaz de usuario, mediante la declaración y configuración de los componentes o ***widgets***, así como la definición de las funciones de *retrollamada* que procesaran las señales emitidas. Las funciones de *retrollamada* se pueden declarar mediante la función **gtk_signal_connect**:

```
gtk_signal_connect (objeto, nombre, función, datos);
```

Donde *objeto* es el ***widget*** asociada a la señal, *nombre* es el nombre de la señal que se va a procesar, *función* es el nombre de la rutina que atenderá la señal emitida y *datos* es un puntero a un dato el cual se le enviara a la rutina.

Una vez que se ha inicializado ***GTK+*** y se han construido la interfaz de usuario se debe ceder el control de la ejecución a ***GTK+***, para que puedan procesar los eventos de ratón, del teclado, etc. Esto se hace mediante la llamada a la función **gtk_main**() la cual

genera un proceso que atiende de forma constante los eventos y señales de la aplicación ***GTK+***. La función **gtk_main()** no finaliza hasta que la aplicación haga una llamada a la función **gtk_main_quit()**.

A continuación se muestra un ejemplo de una aplicación creada con ***GTK+***.

```
#include <gtk/gtk.h>

void Cerrar(GtkObject *object, gpointer user_data);

int  main ( int argc, char *argv[] )
{
 GtkWidget *window;
 GtkWidget *etiqueta;


  gtk_init(&argc, &argv);

  window = gtk_window_new(GTK_WINDOW_TOPLEVEL);

  gtk_window_set_title (GTK_WINDOW (window), "Ejemplo de GIMP Tool Kit");

  gtk_signal_connect (GTK_OBJECT(window), "destroy", Cerrar,  NULL);

  etiqueta = gtk_label_new ("Hola Mundo !");

  gtk_container_add (GTK_CONTAINER (window), etiqueta);

  gtk_widget_show_all(window);

  gtk_main();

 return 0;
}

void Cerrar (GtkObject *object, gpointer user_data)
{
  gtk_main_quit();
}
```

El ejemplo simplemente crea una ventana con el titulo "*Ejemplo de GIMP Tool Kit*" y asigna a el contenedor de la ventana una *widget* etiqueta con el texto "*Hola Mundo!*", y agrega a la ventana una manejador para la señal "*destroy*" que indica la destrucción del objeto mediante la función de *retrollamada* "*Cerrar*" en la cual implementamos la salida de la aplicación mediante la función **gtk_quit_main()**.

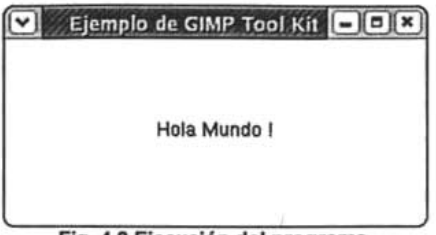


Fig. 4.2 Ejecución del programa.

De esta forma se pueden agregar controles más elaborados y asignarles a sus funciones de *retrollamada* funcionalidad más compleja. De esta forma se delega a ***GTK+*** la responsabilidad de la presentación de la interfaz grafica de usuario, y es posible generar aplicaciones de forma rápida.

IV.1.1 OpenGL y MESA.

La programación de gráficos en 3D implica no sólo implementar los algoritmos correspondientes a los modelos a presentar, sino también implementar los algoritmos propios de la visualización de estos, que implica la representación de la textura de estos, su la proyección en pantalla, etc. El uso de una biblioteca gráfica que implemente todas estas cuestiones permite orientar el esfuerzo de desarrollo a otros aspectos.

La API ***OpenGL*** se desarrollo en ***Silicon Graphics Inc***. Desde su introducción en 1992 se ha convertido prácticamente en un estándar de la industria para gráficos en 3D. Existen implementaciones para la casi todos los sistemas operativos (UNIX, BeOs, MacOs, Windows, Linux,...) y es posible implementarla desde diferentes lenguajes (Ada, C, C++, Fortran, Python, Perl, Java).

Aunque existen implementaciones comerciales de ***OpenGL*** para Linux, la implementación más popular y más utilizada es ***Mesa***. Mesa es una implementación de código abierto y gratuita de una API estilo ***OpenGL***. No la podemos llamar ***OpenGL*** por que no tiene licencia de ***Silicon Graphics***. Sin embargo ***Mesa*** implementa casi todo lo que tiene la API ***OpenGL***.

Sin embargo como la responsabilidad del gráfico de ventanas se delega a la biblioteca ***GTK+*** se necesita una implementación de ***OpenGL*** que fuera compatible con ésta. Se decidió utilizar la biblioteca ***GtkGLExt*** que es una extensión de ***OpenGL*** para la librería ***GTK+*** 2.0 en adelante. Este proporciona objetos adicionales a la librería ***GDK*** para que soporte visualizaciones ***OpenGL*** en componentes (***widgets***) estándar[16].

IV.1.1 GLADE

Glade[17] es un constructor de interfaces de usuario para GTK+. Es gratuito y es distribuido bajo GPL, y contiene soporte para ***GTK+*** y ***GNOME***. Es una herramienta RAD (Diseño Rápido de Aplicaciones) que permite de forma visual construir interfaces de usuario a partir de un conjunto de ***widgets*** de ***GTK+***.

Fig. 4.3 Apariencia de una sesión de GLADE

Glade produce código fuente para C, C++, Ada95, Python y Perl. Entre las ventajas que proporciona Glade es que desde su interfaz de usuario se pueden generar la estructura visual de la aplicación a desarrollar, y se pueden asignar ciertos parámetros de configuración para cada componente o ***widget***, y las señales de estos con las cuales se

[16] http://gtkglext.sourceforge.net
[17] http://glade.gnome.org/

desean trabajar. Al generar el código desde la funcionalidad que ofrece Glade la aplicación construye un conjunto de archivos básicos a partir de los cuales se compilara la aplicación, y son:

Fig. 4.4 Opciones de C en GLADE

main.c	Archivo que contiene a la función main(). Aquí se configuran los parámetros iniciales e inicia la aplicación.
support.c	Contiene algunas funciones de ayuda para la construcción de la interfaz hecha en Glade. Contiene funcionalidades de manejo de cadenas, búsqueda de componentes, búsqueda de componentes en memoria, etc. Genera también su correspondiente archivo cabecera: "***support.h***".
interface.c	Contiene funciones con el código correspondiente a la creación y configuración de todos los componentes de la interfaz (ventanas, menús, etc.). Generalmente Glade escribe una función para cada contenedor principal. Estas funciones son llamadas en el momento de la aplicación en que se quiera crear en memoria el componente a través de su correspondiente archivo cabecera: "***interface.h***".
callbacks.c	Contiene todas las funciones de retrollamada para cada ***widget***. Es en este archivo donde se incluye el código correspondiente para procesar cada señal generada por la aplicación. Se genera también su correspondiente archivo cabecera: "***callbacks.h***".

Otra ventaja de Glade también construye un conjunto de archivos que facilita la compilación de los archivos que son generados y permite la creación de la aplicación final, entre los más importantes se encuentran:

autogen.sh	Es un script o guión shell que permite verificar las dependencias de las bibliotecas que se utilizaran (GTK, etc.) y las utilerías a implementar y configura un archivo ***Makefile*** mediante el cual se construirá la aplicación con ayuda de la herramienta ***make*** de Linux.
Makefile	Guión shell que contiene todos los pasos por los cuales debe pasar la compilación a partir del cual se construirá la aplicación.
Makefile.am	Guión shell que contiene los archivo y paquetes que se utilizaran en la compilación. Este archivo es posible editarlo para personalizar la aplicación.

Sin embargo una de las desventajas de esta herramienta es que aún no implementa todos los ***widgets*** de ***GTK+***, y de los que utiliza no configura todos los posibles parámetros de para inicializarlos. Otra desventaja es que construye el código correspondiente a la interfaz de usuario, sin embargo para editar estas instrucciones o agregarle funcionalidad a estas es necesario utilizar un editor de textos externo a la herramienta. Lo cual dificulta el desarrollo del diseño o la modificación del prototipo.

Sin embargo se puede considerar a ***Glade*** como una excelente opción para un diseño rápido de interfaces. Al momento de escribir esta tesis, existe ya en marcha un proyecto en la comunidad de software libre que integra a ***Glade*** dentro de un IDE para desarrollo de aplicaciones ***GTK+*** para C/C++ de nombre ***Anjuta***[18].

[18] http://anjuta.org

IV.2 Estructuras Básicas.

Para poder implementar las herramientas, de manejo de parches Bézier y de modelado de Blobs, primero debemos definir las estructuras de datos básicas que nos permitan manipular el modelo a nivel de aplicación.

La estructura más básica necesaria es el punto en 3D, que es la que nos permitirá almacenar en memoria una determinada posición en el espacio, y consta de tres valores reales:

```
typedef struct{
    gdouble x, y, z;
}point3D;
```

Para poder referirnos a los planos de proyección, usamos un enumerado que permitirá hacer el código fuente más legible.

```
enum enum_planos{ PLANO_XY = 0, PLANO_XZ, PLANO_YZ, PERSPECTIVA };
```

Uno de los primeros problemas a resolver es que los lienzos sobre los que se visualiza y edita son en realidad mapas de bits y una posición dentro de ellos es en realidad coordenadas de píxeles, sin embargo los objetos gráficos que se manipularan son objetos ubicados en el espacio tridimensional, y son valores numéricos reales.
Para resolver éste problema se decidió implementar los métodos referentes a las *transformaciones de coordenadas de ventana a puerto de vista*[19] que son un conjunto de operaciones que preemitieran realizar un mapeo eficiente entre un área delimitada de nuestro plano de proyección (ventana, referidas como coordenadas mundiales) a un área en el dispositivo de salida (puerto de vista, referidas como coordenadas de dispositivo).

[19] Graficas por computadora. Donald Hearn. Cap 6, pag 231.

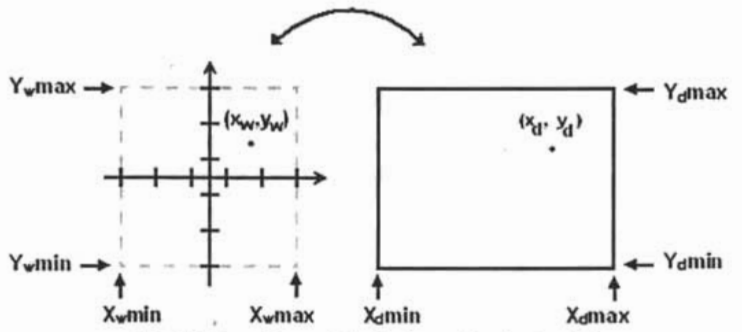


Fig. 4.5 Transformación Ventana-Puerto de vista

Esto se logra usando las siguientes proporciones entre las *coordenadas mundiales* (x_w, y_w) y las *coordenadas de dispositivo* (x_d,y_d):

$$\frac{x_d - x_dmín}{x_dmáx - x_dmín} = \frac{x_w - x_wmín}{x_wmáx - x_wmín}$$

$$\frac{y_d - y_dmín}{y_dmáx - y_dmín} = \frac{y_w - y_wmín}{y_wmáx - y_wmín}$$

Despejando para obtener las incógnitas tenemos:

$$x_d = x_dmín + (x_w - x_wmín)\, s_dx \qquad s_dW = \frac{x_dmáx - x_dmín}{x_wmáx - x_wmín}$$

$$y_d = y_dmín + (y_w - y_wmín)\, s_dy \qquad s_dy = \frac{y_dmáx - y_dmín}{y_wmáx - y_wmín}$$

$$x_w = x_wmín + (x_d - x_dmín)\, s_wx \qquad s_wW = \frac{x_wmáx - x_wmín}{x_dmáx - x_dmín}$$

$$y_w = y_wmín + (y_d - y_dmín)\, s_wy \qquad s_wy = \frac{y_wmáx - y_wmín}{y_dmáx - y_dmín}$$

Se implemento la estructura **coor_W_D** que contiene todos los valores necesarios para generar el cálculo de la transformación.

```
typedef struct{
    gdouble dminX, dmaxX;
    gdouble dminY, dmaxY;
    gdouble wminX, wmaxX;
    gdouble wminY, wmaxY;
    gdouble dsx, dsy;
    gdouble wsx, wsy;
}coor_W_D;
```

La función de configuración del la estructura es **Ini_CoorWD** que recibe como argumentos el ancho y alto del mapa de píxeles y los valores mínimo y máximo de los ejes del plano de edición a los que se desea ajustar el lienzo, y regresa una estructura

configurada con las escalas y los valores límites para cada tipo de coordenadas. Se debe notar que la función resuelve los problemas de relación de aspecto que se deriven de la modificación del tamaño del lienzo de edición, ajustando los límites del eje horizontal.

```
coor_W_D Ini_CoorWD ( gint      Width,  gint Height,
                      gdouble x_min,  gdouble y_min,
                      gdouble x_max, gdouble y_max )
{
  coor_W_D coorWD;
  gdouble wancho;

  coorWD.dminX = 0;
  coorWD.dmaxX = Width-1;
  coorWD.dminY = Height-1;
  coorWD.dmaxY = 0;
  coorWD.wminY = y_min;
  coorWD.wmaxY = y_max;
  wancho = (y_max-y_min) *  Width / Height;
  coorWD.wminX = -wancho/2;
  coorWD.wmaxX = wancho/2;

  coorWD.dsx = (coorWD.dmaxX - coorWD.dminX)/(coorWD.wmaxX - coorWD.wminX);
  coorWD.dsy = (coorWD.dmaxY - coorWD.dminY)/(coorWD.wmaxY - coorWD.wminY);
  coorWD.wsx = (coorWD.wmaxX - coorWD.wminX)/(coorWD.dmaxX - coorWD.dminX);
  coorWD.wsy = (coorWD.wmaxY - coorWD.wminY)/(coorWD.dmaxY - coorWD.dminY);

  return coorWD;
}
```

Una vez configurada esta estructura disponemos de funciones que transformen de un tipo de coordenadas a otro, esto es importante ya que todos los cálculos y las posiciones de los modelos a editar están en relación al espacio tridimensional y es necesario realizar un mapeo de esas posiciones en relación a los píxeles del lienzo de dibujo y viceversa. Las funciones son para pasar de un sistema de referencia a otro para cada eje del plano en edición.

```
gdouble T_Coord_D_W_X       ( coor_W_D coorWD,  gint x_d )
{    return (coorWD.wminX + ( x_d - coorWD.dminX ) * coorWD.wsx);  }

gdouble T_Coord_D_W_Y       ( coor_W_D coorWD,  gint y_d )
{    return (coorWD.wminY + ( y_d - coorWD.dminY ) * coorWD.wsy);  }

gint T_Coord_W_D_X  ( coor_W_D coorWD,  gdouble x_w )
{    return (coorWD.dminX + ( x_w - coorWD.wminX ) * coorWD.dsx);  }

gint T_Coord_W_D_Y  ( coor_W_D coorWD,  gdouble y_w )
{    return (coorWD.dminY + ( y_w - coorWD.wminY ) * coorWD.dsy);  }
```

IV.3 Parches de Bézier.

Para crear un parche Bézier se parte del criterio de generar sólo superficies cúbicas de Bézier, lo que quiere decir que se genera una malla de control de 4x4 la cual definimos mediante la siguiente declaración:

```
# define MAX_BEZIER 4
```

La cual utilizaremos para definir el tamaño de la malla en el código fuente.

Para almacenar la estructura de los parches de Bézier se implemento la siguiente estructura de datos que contiene las propiedades básicas necesarias para construir una superficie de Bézier:

```
typedef struct{
    gchar nombre[20];
    point3D v_c[MAX_BEZIER][MAX_BEZIER];
    gint n_u, n_v;
    point3D *p;
}parche_bezier;
```

gchar nombre	Cadena de Identificación de la superficie.
point3D v_c[MAX_BEZIER][MAX_BEZIER]	Arreglo para almacenar los vértices de la malla de control, que es de 4x4.
gint n_u	Numero de segmentos en que se divide el parámetro '***u***' para desarrollar la superficie.
gint n_v	Numero de segmentos en que se divide el parámetro '***v***' para desarrollar la superficie.
point3D *p	Puntero a un arreglo de puntos. Se derivan del calculo de la superficie en base a los parámetros '***u***' y '***v***'. Como desde la interfaz el numero de segmentos de estos es configurable, se maneja como puntero ya el arreglo puede crecer o disminuir.

Para generar la estructura de un parche Bézier se implemento la función **Parche_Bezier_new**, que crea un parche con parámetros iniciales y con un identificador predeterminado.

```
parche_bezier*  Parche_Bezier_new    ( char *nombre )
{
 parche_bezier *parche;
 gint u, v;

  parche = g_malloc( sizeof(parche_bezier) );
  sprintf(parche->nombre, "%s", nombre);

  for(u=0; u <= MAX_BEZIER-1; u++ )
     for(v=0; v <= MAX_BEZIER-1; v++ )
     {
       parche->v_c[u][v].x = -0.5 + 0.25*u;
       parche->v_c[u][v].y = 0;
       parche->v_c[u][v].z = -0.5 + 0.25*v;
     }

  parche->n_u = 4;
  parche->n_v = 5;

  parche->p = calloc( (size_t)(parche->n_u+1)*(parche->n_v+1), sizeof(point3D) );

  Sup_Pnt_Bezier( parche->p, parche->v_c, parche->n_u, parche->n_v );

 return parche;
}
```

La función asigna memoria para una estructura de tipo ***parche_bezier***, a cuyo miembro *nombre* es asignado la cadena nombre pasada a la función por parámetro. Se calculan los vértices de control de forma lineal, se asignan el número de segmentos para los parámetros '***u***' y '***v***', y se calcula el tamaño del arreglo de los puntos de la superficie y se asigna mediante la función de manejo de memoria *calloc* la dirección de memoria asignada, y se calculan los puntos de la superficie mediante la función ***Sup_Pnt_Bezier***, y se regresa el puntero de la estructura del parche Bézier para ser tratado o modificado.

La rutina para calcular los puntos de la superficie de Bézier esta dividida en dos funciones ***Sup_Pnt_Bezier*** y ***p_sup_bezier***. La función ***Sup_Pnt_Bezier*** recibe como parámetros los valores de los cuales va a calcular todos los puntos de una superficie de Bézier.

```
void Sup_Pnt_Bezier(   point3D *p,  point3D c_v[][MAX_BEZIER],  gint seg_u,  gint seg_v )
{
  gint u, v;
  for( u=0; u<= seg_u; u++ )
    for( v=0; v<= seg_v; v++ )
      p[ u*(seg_v+1)+v ] = p_sup_bezier ( (gdouble)u/seg_u, (gdouble)v/seg_v, c_v );
}
```

Crea dos ciclos anidados, uno para cada parámetro ('*u*' y '*v*'), y para cada valor manda llamar a la función **p_sup_bezier** a la cual se le pasa por parámetro los valores específicos de ***u*** y ***v*** y el conjunto de vértices de control con los cuales generara el cálculo devolviendo un punto específico en la superficie de Bézier. La función **p_sup_bezier** hace uso de las siguientes macros que son la definición de los polinomios de Bernstein para n=3.

```
#define b0(x)   pow(1-x, 3)
#define b1(x)   3*x*pow(1-x, 2)
#define b2(x)   3*pow(x, 2)*(1-x)
#define b3(x)   pow(x, 3)

point3D p_sup_bezier   ( gdouble u, gdouble v, point3D c_v[][MAX_BEZIER] )
{
  point3D p;
  gdouble b0_u, b1_u, b2_u, b3_u, b0_v, b1_v, b2_v, b3_v;

  b0_u = b0(u);
  b1_u = b1(u);
  b2_u = b2(u);
  b3_u = b3(u);
  b0_v = b0(v);
  b1_v = b1(v);
  b2_v = b2(v);
  b3_v = b3(v);

  p.x =  c_v[0][0].x*b0_u*b0_v + c_v[0][1].x*b0_u*b1_v +
         c_v[0][2].x*b0_u*b2_v + c_v[0][3].x*b0_u*b3_v +
         c_v[1][0].x*b1_u*b0_v + c_v[1][1].x*b1_u*b1_v +
         c_v[1][2].x*b1_u*b2_v + c_v[1][3].x*b1_u*b3_v +
         c_v[2][0].x*b2_u*b0_v + c_v[2][1].x*b2_u*b1_v +
         c_v[2][2].x*b2_u*b2_v + c_v[2][3].x*b2_u*b3_v +
         c_v[3][0].x*b3_u*b0_v + c_v[3][1].x*b3_u*b1_v +
         c_v[3][2].x*b3_u*b2_v + c_v[3][3].x*b3_u*b3_v;
  p.y =  c_v[0][0].y*b0_u*b0_v + c_v[0][1].y*b0_u*b1_v +
         c_v[0][2].y*b0_u*b2_v + c_v[0][3].y*b0_u*b3_v +
         c_v[1][0].y*b1_u*b0_v + c_v[1][1].y*b1_u*b1_v +
         c_v[1][2].y*b1_u*b2_v + c_v[1][3].y*b1_u*b3_v +
         c_v[2][0].y*b2_u*b0_v + c_v[2][1].y*b2_u*b1_v +
         c_v[2][2].y*b2_u*b2_v + c_v[2][3].y*b2_u*b3_v +
         c_v[3][0].y*b3_u*b0_v + c_v[3][1].y*b3_u*b1_v +
         c_v[3][2].y*b3_u*b2_v + c_v[3][3].y*b3_u*b3_v;
  p.z =  c_v[0][0].z*b0_u*b0_v + c_v[0][1].z*b0_u*b1_v +
         c_v[0][2].z*b0_u*b2_v + c_v[0][3].z*b0_u*b3_v +
         c_v[1][0].z*b1_u*b0_v + c_v[1][1].z*b1_u*b1_v +
         c_v[1][2].z*b1_u*b2_v + c_v[1][3].z*b1_u*b3_v +
         c_v[2][0].z*b2_u*b0_v + c_v[2][1].z*b2_u*b1_v +
         c_v[2][2].z*b2_u*b2_v + c_v[2][3].z*b2_u*b3_v +
         c_v[3][0].z*b3_u*b0_v + c_v[3][1].z*b3_u*b1_v +
         c_v[3][2].z*b3_u*b2_v + c_v[3][3].z*b3_u*b3_v;

 return (p);
}
```

IV.3.1 Diseño de la Aplicación.

La Interfaz de usuario se compone de una ventana principal de Nivel Superior (GTK_WINDOW_TOPLEVEL) que se divide básicamente en tres secciones:

- El área de Menús
- Un área de Edición y visualización gráfica.
- Un área destinada a contener las herramientas de modelado y edición.

Fig. 4.6 Diseño de la Interfaz de usuario.

Esta distribución de la interfaz se generó mediante los widgets contenedores GtkHBox (contenedor de 1 renglón por n columnas), GtkVBox (contenedor de una columna por n renglones) y GtkTable (contenedor de n columnas por m renglones), que son *widgets* para empaquetar a otros *widgets* en una determinada celda.

Para las vistas múltiples[20] y la de perspectiva se utilizó un marco (GtkFrame) con etiqueta y un área de dibujo (GtkDrawingArea) para cada vista, empaquetados en una tabla de 2x2 configurada para una distribución homogénea de sus celdas.

[20] Proyecciones ortográficas de los planos XY, XZ y YZ.

Se utilizó una barra de menús (GtkMenuBar) conteniendo cada uno de ellos. Para las herramientas de edición y modelado una carpeta (GtkNotebook) con dos pestañas, una de ellas conteniendo una tabla con las herramientas de edición y manipulación, y otra con un árbol (GtkTreeView) con el árbol de objetos en el editor.

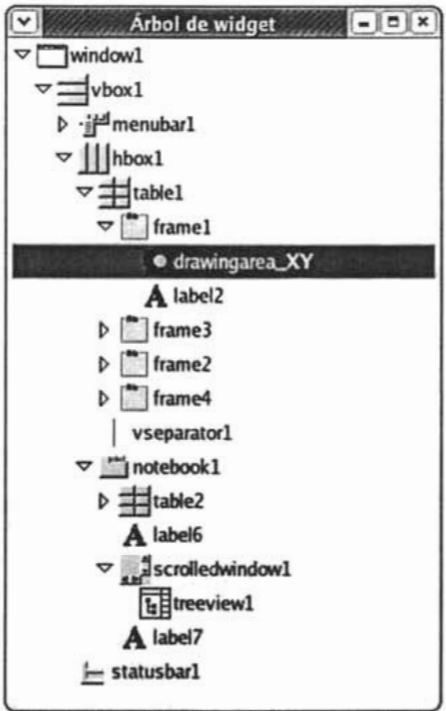


Fig. 4.7 Árbol de widgets con la jerarquía de objetos.

Para el área de herramientas se diseño de forma que se pudieran manipular las propiedades configurables de una superficie de Bézier, que son el número de segmentos para los parámetros '*u*' y '*v*'. En ese sentido se implementó un *widget* de escala horizontal (GtkHScale) que nos proporciona una barra desplazable para ajustar un valor numérico, sus valores se limitaron entre 1 y 10, es decir, que se puede configurar el número de segmentos en u o v entre estos límites. El otro elemento configurable en un parche Bézier son sus vértices de control los cuales, para efecto de esta tesis, solo se implementara la modificación de la posición de ellos, para lo cual se implemento un botón de dos estados (GtkToggleButton) el cual si esta presionado habilitará la translación de los vértices de control seleccionados, para lo cual se

agregaron botones para generar rutinas de selección, como son "Seleccionar todos", "Vaciar Selección" e "Invertir Selección".

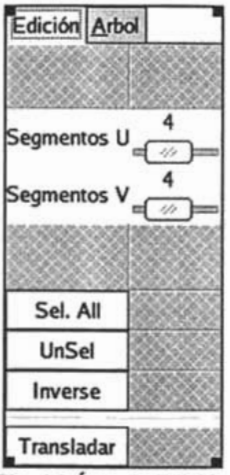


Fig. 4.8 Área de Edición

Para la selección de una superficie de Bézier específica se implementó un árbol de objetos (GtkTreeView) en donde se van agregando los parches que se van creando en la escena y el usuario puede seleccionarlos de manera interactiva.

Fig. 4.9 Árbol de Objetos.

La parte de la visualización y manipulación gráfica de la superficie Bézier se realizó directamente sobre las áreas de dibujo (***GtkDrawingArea***). El área de dibujo es un *widget* que nos permite aplicar funciones del API de ***GDK*** para dibujo de primitivas o elementos gráficos especiales del usuario. Sobre esta se realizará la visualización y la edición de las superficies de Bézier, pero para esto es necesario que el *widget*

responda a una determinada acción del usuario a través de eventos de ratón y teclado. Para esto es necesario que después de crear el *widget* se configuren los eventos a los que responderá el componente y se definan las funciones de *retrollamada* que manejaran el suceso.

```
drawingarea_XY = gtk_drawing_area_new ();
...
gtk_widget_set_events (drawingarea_XY,
        GDK_EXPOSURE_MASK |
        GDK_BUTTON1_MOTION_MASK |
        GDK_BUTTON_PRESS_MASK |
        GDK_BUTTON_RELEASE_MASK);
...
 g_signal_connect ((gpointer) drawingarea_XY, "button_press_event",
            G_CALLBACK (on_drawingarea_XY_button_press_event),
            NULL);
 g_signal_connect ((gpointer) drawingarea_XY, "button_release_event",
            G_CALLBACK (on_drawingarea_XY_button_release_event),
            NULL);
 g_signal_connect ((gpointer) drawingarea_XY, "expose_event",
            G_CALLBACK (on_drawingarea_XY_expose_event),
            NULL);
 g_signal_connect ((gpointer) drawingarea_XY, "motion_notify_event",
            G_CALLBACK (on_drawingarea_XY_motion_notify_event),
            NULL);
 g_signal_connect ((gpointer) drawingarea_XY, "configure_event",
            G_CALLBACK (on_drawingarea_XY_configure_event),
            NULL);
...
```

Con este código se esta configurando al área de dibujo que responda a los eventos de:

GDK_EXPOSURE_MASK	Evento de exposición parcial o total de la ventana, para redibujarla.
GDK_BUTTON1_MOTION_MASK	Movimiento con el botón 1 del ratón pulsado
GDK_BUTTON_PRESS_MASK	Cuando se presiona cualquier botón del ratón.
GDK_BUTTON_RELEASE_MASK	Cuando se libera cualquier botón del ratón.

Cada uno de estos sucesos son manejados por su correspondiente función de *retrollamada*. Se agrega la función para la señal "***configure_event***", la cual ocurre cuando el tamaño del componente cambia, por lo cual es necesario ajustar las rutinas de dibujo.

IV.3.2 Visualización y Edición de las Superficies.

Para la construir la funcionalidad del programa se utilizaron un conjunto de variables que almacenarán el estado actual del sistema o estructuras de datos importantes que permitieran facilitar el manejo de la información dentro de la aplicación.

```
static parche_bezier *curr_p_b;
static coor_W_D PLcoord[4];
```

El puntero a una estructura **parche_bezier** *curr_p_b* sirve para hacer referencia al la superficie de Bézier que se esta editando en ese momento en la aplicación. Esta referencia se actualiza cada vez que se crea un nuevo parche Bézier para edición o se selecciona otro parche en árbol de objetos. Y un arreglo del tipo **coor_W_D** para almacenar la configuración de a trasformación "*ventana-puerto de vista*" para cada plano edición.

Las variables *lista* y *selec* son objetos de tipo **GSList** que son un tipo de datos que ofrece la biblioteca **GLib** para manejar listas enlazadas.

```
typedef struct {
    gpointer data;
    GSList *next;
} GSList;
```

En el campo de datos (*data*) es un puntero a cualquier tipo de datos y el campo *next* es un puntero al siguiente nodo de la lista. De esta forma cada vez que se crea un parche para manipularlo es agregado a *lista* que contiene todos los parches Bézier en la escena para edición. Del mismo modo, cada vez que se efectúa una acción de selección de vértices en el parche actual, la referencia a los vértices de control seleccionados son agregados a la lista *selec* para su manipulación.

```
static GSList *lista = NULL;
static GSList *selec = NULL;
```

Las variables *xini*, *yini*, *xfin*, *yfin* son variables que almacenan la posición inicial y final de cualquier acción de manipulación en los planos de edición, como son translación de los vértices de control o la selección de los vértices dentro de una área determinada.

```
static gdouble xini, yini, xfin, yfin;
static gboolean flg_sel = FALSE, flg_Trans= FALSE;
```

Las variables *flg_sel* y *flg_Trans* son usadas como banderas para señalizar las acciones de selección o translación respectivamente.

Para dibujar sobre las áreas de dibujo correspondientes a las vistas múltiples se utilizó una técnica llamada de "doble Buffer"; que consiste en minimizar la perdida de información gráfica en pantalla al momento de redibujarla. En esta técnica se utiliza un buffer como lienzo para el dibujo. La imagen en pantalla no es modificada por las rutinas de dibujo sino que estas tienen lugar en buffer. Una vez que se ha terminado de dibujar se copia el buffer con el gráfico terminando a la ventana de visualización.
Para los buffers se utilizó el objeto GdkPixmap:

```
static GdkPixmap *pixmap_XY = NULL;
static GdkPixmap *pixmap_XZ = NULL;
static GdkPixmap *pixmap_YZ = NULL;
```

Cuando ocurre una modificación en el tamaño del área de dibujo (sucede al iniciar la aplicación) se genera la señal de configuración del widget de dibujo, la cual se utiliza para reinicializar tanto el buffer y en caso de que existieran superficies en edición, dibujarlas sobre él.

```
gboolean
on_drawingarea_XY_configure_event     (GtkWidget       *widget,
                                       GdkEventConfigure *event,
                                       gpointer         user_data)
{
  if( pixmap_XY )  gdk_drawable_unref( pixmap_XY );
  pixmap_XY = gdk_pixmap_new( widget->window,
                  widget->allocation.width, widget->allocation.height, -1 );
  gdk_draw_rectangle( pixmap_XY, widget->style->white_gc, TRUE,
                  0, 0,  widget->allocation.width, widget->allocation.height );

  PLcoord[PLANO_XY] = Ini_CoorWD( widget->allocation.width, widget->allocation.height,
                                  -1,  -1, 1, 1);
  if(lista) Repaint( pixmap_XY, PLANO_XY, PLcoord[PLANO_XY], lista, widget);

  return FALSE;
}
```

La señal de exposición es muy simple y no sirve para redibujar la parte que quedo expuesta del área de dibujo, por ejemplo que haya pasado una ventana sobre esta ocultándola y la ha dejado de de nuevo expuesta. En este caso solo tenemos que dibujar la parte que ha sido expuesta desde el buffer correspondiente. Esto lo

realizamos mediante un puntero a una estructura ***GdkEventExpose*** que ***GTK+*** pasa como argumento a la función de retrollamada. Esta estructura contiene información sobre la región que necesita ser redibujada en el *widget*.

```
struct GdkEventExpose {
    GdkEventType type;
    GdkWindow *window;
    gint8 send_event;
    GdkRectangle area;
    GdkRegion *region;
    gint count;
};
```

En el miembro "*area*" de la estructura contiene la coordenada inicial (*x, y*) y las dimensiones (ancho y alto) de la región que es afectada en el área de dibujo.

```
gboolean
on_drawingarea_XY_expose_event      (GtkWidget      *widget,
                                    GdkEventExpose  *event,
                                    gpointer        user_data)
{
  gdk_draw_drawable( widget->window, widget->style->white_gc, pixmap_XY,
                                    event->area.x, event->area.y,
                                    event->area.x, event->area.y,
                                    event->area.width, event->area.height );
  return FALSE;
}
```

Los eventos principales para la manipulación de la superficie son los relacionados con el manejo del ratón sobre el lienzo de dibujo.

El suceso generado es por la pulsación de un botón del ratón sobre el área de dibujo es captada por la función **on_drawingarea_XY_button_press_event** que recibe como argumento un puntero a una estructura ***GdkEventButton***.

```
struct GdkEventButton {
    GdkEventType type;
    GdkWindow *window;
    gint8 send_event;
    guint32 time;
    gdouble x;
    gdouble y;
    gdouble *axes;
    guint state;
    guint button;
    GdkDevice *device;
    gdouble x_root, y_root;
};
```

Esta estructura contiene información sobre el tipo de evento ('*type*', si es pulsación del botón, doble clic, o si fue liberado el botón del ratón), la coordenada *(x,y)* relativa al widget donde ocurrió el evento, el botón que fue pulsado ('*button*', derecho, izquierdo, medio) y la mascara de los modificadores que acompañan a el evento ('*state*', las teclas "Shift", "Control", "Alt").

```
gboolean
on_drawingarea_XY_button_press_event  (GtkWidget      *widget,
                                       GdkEventButton  *event,
                                       gpointer        user_data)
{
  xini = event->x;
  xfin = event->x;
  yini = event->y;
  yfin = event->y;
  if(event->state & GDK_SHIFT_MASK && event->button == 1)
    if(!flg_sel)
          flg_sel = TRUE;
  return FALSE;
}
```

Al recibir la señal de que se presiono el botón del ratón guarda las coordenadas del suceso. Si es el botón principal y la mascara es la tecla "SHIFT" entonces asigna al la variable '*flg_sel*' el valor de verdadero, esta variable nos servirá como bandera para identificar cuando el usuario ha requerido una acción de selección sobre el lienzo de dibujo.

La función **on_drawingarea_XY_button_release_event** es relativamente sencilla, solo verifica si el usuario esta terminando una acción de selección sobre el lienzo, y si es así pone la variable '*flg_sel*' a falso, y asigna a la variable '*selec*' la lista de los vértices que la función **Buscar_Punto** encuentre en el parche que se esta editando dentro de los límites que definió el usuario.

```
gboolean
on_drawingarea_XY_button_release_event (GtkWidget      *widget,
                                        GdkEventButton  *event,
                                        gpointer        user_data)
{
  if( flg_sel )
  {
    Dib_Recuadro( widget, pixmap_XY, widget->style->white_gc,
                  xini, yini, event->x, event->y, TRUE);
    flg_sel = FALSE;
    selec = Buscar_Punto(curr_p_b, xini, yini, xfin, yfin, PLANO_XY, PLcoord[PLANO_XY]);
  }

  return FALSE;
}
```

La función **on_drawingarea_XY_motion_notify_event** es un poco más complicada. Recibe como argumento un puntero a una estructura ***GdkEventMotion***.

```
struct GdkEventMotion {
      GdkEventType type;
      GdkWindow *window;
      gint8 send_event;
      guint32 time;
      gdouble x;
      gdouble y;
      gdouble *axes;
      guint state;
      gint16 is_hint;
      GdkDevice *device;
      gdouble x_root, y_root;
};
```

Esta estructura contiene información parecida a la ***GdkEventButton***, con la diferencia que para conocer el botón pulsado se obtiene de la máscara contenida en '*state*'.

```
gboolean
on_drawingarea_XY_motion_notify_event  (GtkWidget      *widget,
                                        GdkEventMotion  *event,
                                        gpointer        user_data)
{
  if( event->state &  GDK_BUTTON1_MASK )
  {
    if( (event->state & GDK_SHIFT_MASK) && flg_sel )
    {
      Dib_Recuadro( widget, pixmap_XY, widget->style->white_gc,  xini, yini, xfin, yfin, TRUE);
      xfin = event->x;
      yfin = event->y;
      Dib_Recuadro( widget, pixmap_XY, widget->style->black_gc, xini, yini, xfin, yfin, FALSE);
    }
    else
    {
      if( lista && flg_Trans )
      {
          GSList *sel = selec;
          point3D *p;
        while( sel )
          {
            p = (point3D *)sel->data;
            p->x = p->x -
                    T_Coord_D_W_X( PLcoord[PLANO_XY], xfin-xini) +
                    T_Coord_D_W_X( PLcoord[PLANO_XY], event->x-xini);
            p->y = p->y -
                    T_Coord_D_W_Y( PLcoord[PLANO_XY], yfin-yini) +
                    T_Coord_D_W_Y( PLcoord[PLANO_XY], event->y-yini);
            sel = g_slist_next(sel);
        }
        Calc_Surfc(curr_p_b);
          Redibuja();
        xfin = event->x;
```

```
      yfin = event->y;
    }
   }
  }
  return FALSE;
}
```

Si el botón principal esta pulsado puede ocurrir dos acciones, el usuario esta realizando una acción de selección sobre el lienzo de dibujo, o esta realizando una translación a los vértices de control de la superficie ya seleccionados.

Si es el primer caso, debe estar activada la tecla "SHIFT" y la bandera de selección '*flg_sel*' debe estar en estado **verdadero**, y lo que realiza la rutina es almacenar la coordenada a la cual se ha movido el puntero del ratón y dibuja sobre el lienzo un rectángulo mediante la llamada a la función **Dib_Recuadro** señalizando el área que se esta delimitando.

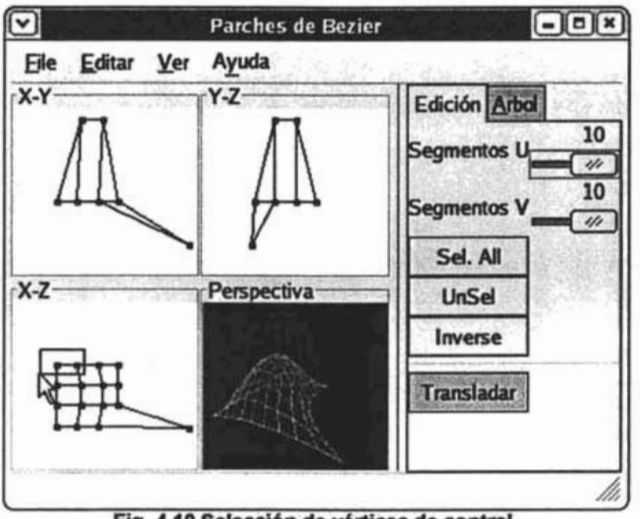


Fig. 4.10 Selección de vértices de control.

Si no se cumplen esas condiciones, se revisa el valor de la variable '*flg_Trans*', cuyo valor es asignado cuando se presiona el botón de dos estados del área de herramientas "*Transladar*" desde su correspondiente manejador de la señal.

```
void
on_togglebutton_Transladar_toggled     (GtkToggleButton *togglebutton,
                                        gpointer         user_data)
{
  flg_Trans = gtk_toggle_button_get_active(togglebutton);
}
```

Si el valor de *'flg_Trans'* es verdadero y existen parches a editar, entonces el usuario intenta modificar la posición de los vértices de control. Para cada uno de los vértices seleccionados y contenidos en la lista *'selec'* se calcula la magnitud del desplazamiento y se le asigna a la coordenada actual del vértice de control y una vez modificados todos los vértices, se vuelve a calcular los puntos de la superficie y se dibuja.

Las funciones de dibujo son simples y se dividen básicamente en dos tipos: la visualización en los planos de edición (XY, XZ, YZ) y la visualización en perspectiva.
Las rutinas de dibujo en las áreas de edición son simples ya que sólo tienen que utilizar dos de las tres coordenadas y descartar la otra sin agregarle un proceso extra (proyección ortográfica de vista múltiple) y se delega a la responsabilidad de dibujar los parches Bézier a la función **Repaint**.

```
void Repaint(   GdkPixmap    *pixmap_pl,
                guint        plano,
                coor_W_D     coord_pl,
                GSList       *lista,
                GSList       *selec,
                parche_bezier *curr_pb,
                GtkWidget    *widget )
{
 GSList *tmplist;
 parche_bezier *parche;
 gboolean activo = FALSE;

 if(plano>=PLANO_XY && plano<=PLANO_YZ)
 {
  gdk_draw_rectangle( pixmap_pl, widget->style->white_gc, TRUE,
            0, 0,  widget->allocation.width, widget->allocation.height );
  tmplist = lista;
  do
  {
   parche = tmplist->data;
   if( parche == curr_pb ) activo = TRUE;
   dibuja_Malla_ctrl( pixmap_pl, plano, coord_pl, parche, selec, activo );
   tmplist = (GSList *)tmplist->next;
   activo = FALSE;
  }while(tmplist);
 }

 else
 {
  gdk_draw_rectangle( pixmap_pl, widget->style->black_gc, TRUE,
            0, 0,  widget->allocation.width, widget->allocation.height );
  tmplist = lista;
  do
  {
```

```
    parche = tmplist->data;
    dibuja_Perspectiva( pixmap_pl, coord_pl, parche );
    tmplist = (GSList *)tmplist->next;
   }while(tmplist);
 }

 gdk_draw_drawable( widget->window, widget->style->white_gc, pixmap_pl,
                    0, 0, 0, 0,
                    widget->allocation.width, widget->allocation.height );

}
```

La función **Repaint** recibe como parámetros un identificador del plano sobre el que se va a dibujar (*plano*), la estructura **coor_W_D** que asignada al plano (*coord_pl*), el buffer sobre el cual se va a dibujar (*pixmap_pl*) y la referencia al componente del área de dibujo (*widget*) al que se va a asignar la imagen una vez terminado el proceso de dibujo. También se pasan a la función la lista de parches (*lista*) para dibujar todos los contenidos en la escena, el parche actual en edición (*curr_pb*) y la lista de vértices seleccionados (*selec*) para hacer una visualización diferenciada. Sin embargo esta función no es la encargada de dibujar directamente en el buffer, sino más bien recorre la lista de parches y los pasa a la función **dibuja_Malla_ctrl** que es la encargada de dibujarla en el buffer.

```
void dibuja_Malla_ctrl ( GdkPixmap  *pixmap_pl,   guint          plano,
                         coor_W_D   coord_plano,  parche_bezier *parche,
                         GSList     *selec,       gboolean      activo)
{
 GdkGC *penPatch, *penVert;
 gint u, v;

  if( activo )  penPatch = penM_S;
  else          penPatch = penM_C;

 switch(plano)
 {
  case PLANO_XY:
   for(u=0; u <= parche->n_u; u++)
    for(v=0; v < parche->n_v; v++)
     gdk_draw_line(pixmap_pl, penSurfc,
                   T_Coord_W_D_X ( coord_plano,  parche->p[u*(parche->n_v+1)+v].x),
                   T_Coord_W_D_Y ( coord_plano,  parche->p[u*(parche->n_v+1)+v].y),
                   T_Coord_W_D_X ( coord_plano,  parche->p[u*(parche->n_v+1)+v+1].x),
                   T_Coord_W_D_Y ( coord_plano,  parche->p[u*(parche->n_v+1)+v+1].y));
   for(v=0; v <= parche->n_v; v++)
    for(u=0; u < parche->n_u; u++)
     gdk_draw_line(pixmap_pl, penSurfc,
                   T_Coord_W_D_X ( coord_plano,  parche->p[u*(parche->n_v+1)+v].x),
                   T_Coord_W_D_Y ( coord_plano,  parche->p[u*(parche->n_v+1)+v].y),
                   T_Coord_W_D_X ( coord_plano,  parche->p[(u+1)*(parche->n_v+1)+v].x),
```

```
                T_Coord_W_D_Y ( coord_plano,  parche->p[(u+1)*(parche->n_v+1)+v].y));
    for(u=0; u < MAX_BEZIER; u++)
      for(v=1; v < MAX_BEZIER; v++)
        {
        gdk_draw_line(pixmap_pl, penPatch,
                        T_Coord_W_D_X ( coord_plano,  parche->v_c[u][v-1].x),
                        T_Coord_W_D_Y ( coord_plano,  parche->v_c[u][v-1].y),
                        T_Coord_W_D_X ( coord_plano,  parche->v_c[u][v].x),
                        T_Coord_W_D_Y ( coord_plano,  parche->v_c[u][v].y));
        gdk_draw_line(pixmap_pl, penPatch,
                        T_Coord_W_D_X ( coord_plano,  parche->v_c[v-1][u].x),
                        T_Coord_W_D_Y ( coord_plano,  parche->v_c[v-1][u].y),
                        T_Coord_W_D_X ( coord_plano,  parche->v_c[v][u].x),
                        T_Coord_W_D_Y ( coord_plano,  parche->v_c[v][u].y));
        }
    for(u=0; u < MAX_BEZIER; u++)
      for(v=0; v < MAX_BEZIER; v++)
        {
          penVert = Pen_Vertice ( &parche->v_c[u][v], selec);
          gdk_draw_arc( pixmap_pl,  penVert,  TRUE,
                        T_Coord_W_D_X ( coord_plano,  parche->v_c[u][v].x)-2,
                        T_Coord_W_D_Y ( coord_plano,  parche->v_c[u][v].y)-2,
                        5, 5, 0, (360*64));
        }
    break;
   case PLANO_XZ:
     ...
   case PLANO_YZ:
     ...
  }
}
```

La función **dibuja_Malla_ctrl** obtiene como argumentos un parche Bézier específico y una bandera (*activo*) si en ese momento esta activo en el área de edición. Dibuja la malla de la superficie de Bezier resultado del algoritmo, que se encuentra en el puntero a un arreglo dinámico en la estructura del parche '*parche->p*'. Después dibuja las líneas y los vértices de la malla de control. Haciendo un dibujo diferenciado de los vértices de control si están o no en la lista de selección (*selec*).

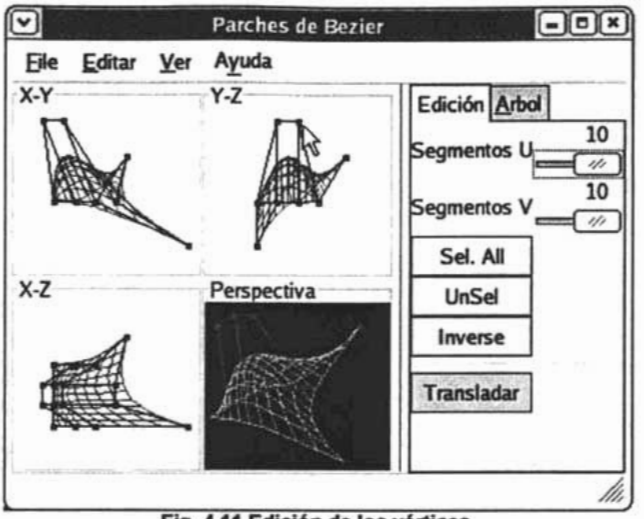


Fig. 4.11 Edición de los vértices.

IV.4 BLOBS.

La implementación de la herramienta de modelado para los Blobs fue de mayor complejidad. Ya que la superficie resultante del modelo no se forma a partir de un componente individual como los es en los parches de Bézier, sino que se crea a partir de la combinación de varios componentes primitivos. Por lo cual se construyeron las siguientes estructuras de datos que permitieran plasmar esa jerarquía de objetos.

La estructura **blob_componente** se diseño para contener a los componentes primitivos a partir de los cuales calcular la *isosuperficie*.

```
typedef struct{
    gchar nombre[10];
    point3D p;
    gdouble radio;
}blob_componente;
```

gchar nombre	Cadena de identificación del componente.
point3D p	Posición del centro del componente.
gdouble radio	Magnitud del radio del componente. Es utilizado para calcular la magnitud del campo escalar.

La estructura que contiene los parámetros globales del modelo, así como los componentes primitivos que lo forman es **blob_modelo**.

```
typedef struct{
  gchar nombre[20];
  gdouble umbral;
  gint num_componentes;
  blob_componente *blobs;
}blob_modelo;
```

gchar nombre	Cadena de identificación del modelo.
gdouble umbral	Valor de umbral del modelo. A partir de este se calcula la *isosuperficie*.
gint num_componentes	Cantidad de componentes a partir de los cuales se va a calcular el modelo.
blob_componente *blobs	Puntero a un arreglo de componentes. El tamaño del arreglo es igual a *num_componentes*.

Además se utilizaron las siguientes declaraciones para el manejo de los componentes (definición de un radio por omisión), así como la definición del modelo de ecuaciones que se utiliza para el cálculo de la *isosuperficie*.

```
#define BlbRadioDflt        1.0
#define BLBEC_BLOB          1
#define BLBEC_METABALL      2
#define BLBEC_SOFT          3
```

Para comenzar a editar un modelo es necesario tener un objeto del tipo **blob_modelo** para sobre el agregarle los componentes necesarios para su manipulación. La función **Blob_Modelo_new** recibe como argumento una cadena la cual asignara como identificador al modelo. Asigna memoria al modelo, y regresa la referencia a una estructura del tipo **blob_modelo** con parámetros iniciales (umbral = 0.6 y sin componentes).

```
blob_modelo*  Blob_Modelo_new       ( char *nombre )
{
  blob_modelo *modelo;

   modelo = g_malloc( sizeof(blob_modelo) );
   sprintf(modelo->nombre, "%s", nombre);
   modelo->umbral = 0.6;
   modelo->num_componentes = 0;
   modelo->blobs = NULL;
 return modelo;
}
```

Una vez que tenemos una variable de este tipo podemos agregarle componentes según convenga, esto se hace mediante la llamada a la función **Blob_Componente_new** que recibe como argumento la referencia al modelo y una cadena como identificador al componente.

```
blob_componente*  Blob_Componente_new     ( blob_modelo *modelo,  char *nombre )
{
  blob_componente *componente;
  blob_componente *blobs_modelo;
  int iCount;

   blobs_modelo = (blob_componente*) calloc( (size_t)(modelo->num_componentes+1),
sizeof(blob_componente) );
   for( iCount = 0; iCount < modelo->num_componentes; iCount ++)
   {
    sprintf(blobs_modelo[iCount].nombre, "%s", modelo->blobs[iCount].nombre);
    blobs_modelo[iCount].p.x = modelo->blobs[iCount].p.x;
    blobs_modelo[iCount].p.y = modelo->blobs[iCount].p.y;
    blobs_modelo[iCount].p.z = modelo->blobs[iCount].p.z;
    blobs_modelo[iCount].radio = modelo->blobs[iCount].radio;
   }
   sprintf(blobs_modelo[iCount].nombre, "%s", nombre);
   blobs_modelo[iCount].p.x = 0.0;
   blobs_modelo[iCount].p.y = 0.0;
   blobs_modelo[iCount].p.z = 0.0;
   blobs_modelo[iCount].radio = BlbRadioDflt;
   free( modelo->blobs );
   modelo->blobs = blobs_modelo;
   modelo->num_componentes++;
 return (blob_componente*) &modelo->blobs[iCount];
}
```

La función aumenta la memoria asignada al arreglo de componentes y los copia al nuevo espacio de memoria, y al agregar un nuevo componente con parámetros iniciales, incrementa la cantidad de componentes en *modelo->num_componentes* y después de liberar el anterior espacio de memoria referenciado por *modelo->blobs* asigna el nuevo arreglo de componentes. La función regresa la referencia al nuevo componente para ser utilizado con herramientas de edición.

IV.4.1 Diseño de la aplicación.

La Interfaz de usuario que se desarrollo para modelar Blobs es muy parecida a que se utilizo para los parches de Bézier, tres áreas principales sobre una ventana de Nivel Superior (GTK_WINDOW_TOPLEVEL): el área de menús, el área de visualización y edición, y el área de herramientas de modelado.

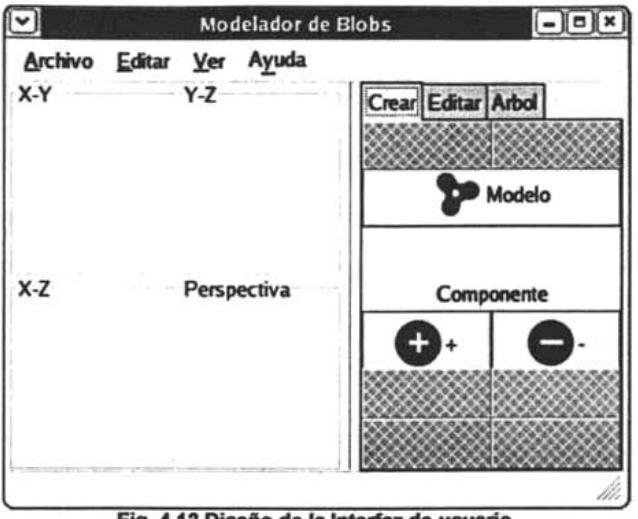


Fig. 4.12 Diseño de la interfaz de usuario.

Sin embargo la forma en como se estructura el área de herramientas de edición requiere mayor elaboración y complejidad que la de los parches Bézier. Se necesitaba tener procedimientos que permitieran crear en forma sencilla un modelo de Blobs y sus componentes. Así que para el diseño del área de herramientas se utilizó una carpeta (GtkNotebook) con tres pestañas.

La pestaña "*Crear*" contiene un botón (GtkButton) con la rutina para generar un nuevo modelo de Blobs y dos botones de dos estados (GtkToggleButton) que habilitan la creación de un componente al modelo en edición mediante una pulsación sobre los planos de edición. Son botones mutuamente excluyentes, es decir que sólo se pueden agregar Blobs con potencial positivo o sólo con potencial negativo, se pueden agregar cuantos componentes se desee mientras estos botones estén habilitados.

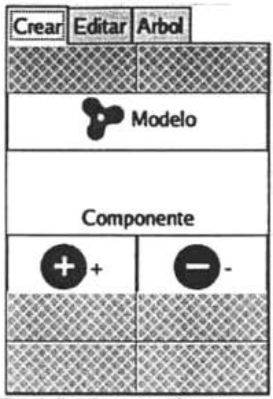


Fig. 4.13 Pestaña de creación de componentes

La pestaña "*Arbol*" contiene una árbol de objetos donde se localizan los modelos que se van agregando a la escena y el usuario puede seleccionarlos de manera interactiva tanto el modelo como los componentes que lo forman para fines de edición.

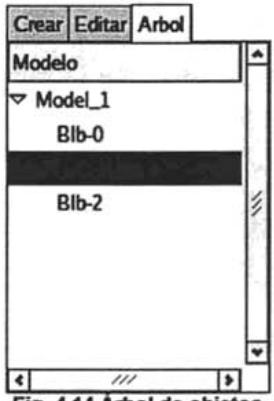


Fig. 4.14 Árbol de objetos

Por último la parte de las herramientas de edición y manipulación requirió mayor complejidad. Esta pestaña con nombre "*Editar*" esta dividida básicamente en tres partes.

El área correspondiente al modelo en edición, la cual contiene el identificador del modelo, y el valor de umbral de la *isosuperficie*, con un *widget* para desplazar un valor numérico en un rango determinado (GtkSpinButton).

El área correspondiente al componente la cual se divide el dos a su vez. Una que contiene las propiedades básicas del componente, como es su cadena de identificación, la magnitud de su radio, y su localización en el espacio. La otra sección contiene las

operaciones de transformación que se pueden ejercer sobre los componentes, como son la modificación de su posición (translación) y la modificación del radio de la primitiva (escalación).

Fig. 4.15 Herramientas de edición.

La otra sección básica tiene que ver con la visualización del modelo en pantalla, la cual contiene otro *GtkSpinButton* para modificar la resolución de la malla con que se presentará la superficie calculada del modelo de Blobs. Un botón que permite actualizar la visualización de esa malla en la vista de visualización, y botón con una rutina de reinicio de la vista de presentación del modelo.

IV.4.2 Edición de los modelos.

Para construir a funcionalidad del programa se utilizaron las siguientes variables para almacenar el estado actual de la aplicación.

Un arreglo del tipo **coor_W_D** para almacenar la configuración de la trasformación "*ventana-puerto de vista*" para cada plano de edición, las referencias del tipo ***GdkPixmap*** para los buffers de dibujo.

```
static coor_W_D      PLcoord[4];
static GdkPixmap     *pixmap_XY = NULL;
static GdkPixmap     *pixmap_XZ = NULL;
static GdkPixmap     *pixmap_YZ = NULL;
```

Punteros a estructuras del tipo **blob_modelo** y **blob_componente** para almacenar la referencia al modelo y al componente activo en la aplicación.

```
static blob_modelo     *curr_BM = NULL;
static blob_componente *curr_BC = NULL;
```

Una lista enlazada (*GSList*) donde se almacenan los modelos en edición en al aplicación.

```
static GSList        *Lista = NULL;
```

Las variables *xini*, *yini*, *xfin*, *yfin* son variables que almacenan la posición inicial y final de cualquier acción de manipulación en los planos de edición. La variable *radio_ini* es para almacenar el radio del componente mientras se realizan operaciones de escalación.

```
static gdouble xini, yini, xfin, yfin, radio_ini;
static guint8 flg_gui = 0, flg_view = V_WIREFRAME;
```

La variable '*flg_gui*' del tipo ***guint8*** (entero sin signo de 8 bits, 0 a 255) contiene el estado de las herramientas de edición de la aplicación, activando un *bit* por cada acción efectuada en la aplicación, genera una mascara que puede ser comparada con el código de las declaraciónes de cada acción.

```
# define CREAR_BLOB     1
# define UTIL_MODELO    2
# define UTIL_COMPNT    4
# define UTIL_TRANSFM   8
# define UTIL_VISUAL    16
# define T_TRASLADAR    32
# define T_ESCALAR      64
# define T_ROTAR        128
```

De igual manera '*flg_view*' contiene la máscara de estado del tipo de visualización que se esta llevando a cabo en la aplicación.

```
# define V_WIREFRAME    15      // 00001111
# define V_FLAT         60      // 00111100
# define V_SMOOTH       240     // 11110000
```

Con respecto a las rutinas de configuración y exposición (*expose_event*) de las áreas de edición son casi iguales a las de los parches de Bézier. Pero las de manejo de eventos de ratón difieren un poco. Los eventos del ratón ya no están solo asociados a acciones de manipulación y edición de los objetos gráficos, sino que ahora tendrán también asociado la creación de estos.

La función **on_drawingareaX_Y_button_press_event** al recibir la señal que se ha pulsado un botón del ratón, verifica si existe un modelo activo en edición, si es así verifica que el botón oprimido sea el botón principal.

```
gboolean
on_drawingareaX_Y_button_press_event  (GtkWidget      *widget,
                                       GdkEventButton  *event,
                                       gpointer        user_data)
{
  gchar name[20];
  if( curr_BM )
    if( event->button == 1 )
    {
      if( flg_gui & CREAR_BLOB )
      {
        sprintf( name, "Blb-%i", curr_BM->num_componentes);
        if(curr_BC) dibuja_blob_Planos(widget, FALSE);
        curr_BC = (blob_componente*) Blob_Componente_new( curr_BM, name );
        if( curr_BC )
        {
          curr_BC->p.x = T_Coord_D_W_X( PLcoord[PLANO_XY], event->x);
          curr_BC->p.y = T_Coord_D_W_Y( PLcoord[PLANO_XY], event->y);
          GtkToggleButton *tgbtnBlobN = (GtkToggleButton*)lookup_widget (widget, "tgbtnBlobN");
          if(gtk_toggle_button_get_active(tgbtnBlobN)) curr_BC->radio = -BlbRadioDflt;
          radio_ini = curr_BC->radio;
          dibuja_blob_Planos(widget, TRUE);
          agrega_Blob_Arbol((GtkTreeView*)lookup_widget(widget, "trvwArbol"),
                            curr_BM->nombre,  name );
        }
      }
      if( flg_gui & ( T_TRASLADAR | T_ESCALAR ) )
      {
        xini = event->x;
        xfin = event->x;
        yini = event->y;
        yfin = event->y;
        if(curr_BC && flg_gui & T_ESCALAR )
          radio_ini = curr_BC->radio;
      }
    }
  return FALSE;
}
```

Si se cumplen estas condiciones entonces verifica el status de la aplicación mediante la variable '*flg_gui*'. Si la mascara de bits indica que se esta creando un componente

(*CREAR_BLOB*) entonces manda llamar a la función **Blob_Componente_new** y lo asigna al puntero '*curr_BC*', asigna las coordenadas donde fue creado, asigna al radio el valor por omisión, dependiendo que tipo de componente se esta creando (con campo escalar positivo o negativo). Por último agrega el componente al árbol de objetos mediante el llamado a la función **agrega_Blob_Arbol**.

Si la mascara de bits indica que lo que se esta realizando es una acción de transformación (translación o escalado) entonces almacena las coordenadas del puntero del ratón, así como el radio actual del componente que se esta editando. Notese que no se consideraron las dos acciones anteriores, creación de los componentes y transformaciones, como excluyentes sino que es posible crear un componente e inmediatamente después, es decir si soltar el botón del ratón, modificarlo.

La función **on_drawingareaX_Y_motion_notify_event** maneja la señal de movimiento del puntero dentro del área de dibujo es semejante a la función de movimiento de los parches de Bézier, sólo se agrego el código que procesa una acción de escalación. La transformación de escalado modifica la magnitud del radio del componente, y vuelve a graficar los componentes en los planos de edición.

```
gboolean
on_drawingareaX_Y_motion_notify_event (GtkWidget      *widget,
                                       GdkEventMotion *event,
                                       gpointer       user_data)
{
  if( flg_gui & T_TRASLADAR && curr_BC )
  {
    curr_BC->p.x = curr_BC->p.x –
                   T_Coord_D_W_X( PLcoord[PLANO_XY], xfin-xini) +
                   T_Coord_D_W_X( PLcoord[PLANO_XY], event->x-xini);
    curr_BC->p.y = curr_BC->p.y –
                   T_Coord_D_W_Y( PLcoord[PLANO_XY], yfin-yini) +
                   T_Coord_D_W_Y( PLcoord[PLANO_XY], event->y-yini);
    actualizar_Componente(widget, curr_BC);
    dibuja_blob_Planos( widget, TRUE);
    xfin = event->x;
    yfin = event->y;
  }
  if( flg_gui & T_ESCALAR && curr_BC )
  {
    gdouble DeltaX = T_Coord_D_W_X( PLcoord[PLANO_XY], xfin) –
                     T_Coord_D_W_X( PLcoord[PLANO_XY], xini);
    if (curr_BC->radio < 0) curr_BC->radio = -fabs(radio_ini - DeltaX);
    else curr_BC->radio = fabs(radio_ini + DeltaX);
    actualizar_Componente(widget, curr_BC);
    dibuja_blob_Planos( widget, TRUE);
```

```
    xfin = event->x;
    yfin = event->y;
  }
 return FALSE;
}
```

Fig. 4.16 Escalación del radio de un componente.

La función **on_drawingareaX_Y_button_release_event** procesa la señal que ocurre cuando se libera cualquier botón del ratón. En éste caso en la rutina se considera que se liberó el botón después de cualquiera de dos posibles acciones: la creación de un componente o la transformación de él.

```
gboolean
on_drawingareaX_Y_button_release_event (GtkWidget       *widget,
                                        GdkEventButton  *event,
                                        gpointer         user_data)
{
 GtkWidget      *wPerspectiva = lookup_widget(widget, "drawingareaPerspectiva");
 if( curr_BM )
 {
 Triangulos = Marching_Cube( curr_BM, iResolucion, Triangulos );
 gtk_widget_queue_draw_area( wPerspectiva, 0, 0,
                             wPerspectiva ->allocation.width, wPerspectiva ->allocation.height);
 }
 return FALSE;
}
```

Ello implica que el modelo ha cambiado sus propiedades por lo tanto si existe un modelo activo en edición, se procede a calcular la superficie del modelo mediante la

llamada a la función **Marching_Cube**, y después se forza el evento de exposición (*expose_event*) del área de dibujo *wPerspectiva*, que es el widget donde se presenta la visualización de la superficie del modelo, invalidando el área de dibujo mediante la función **gtk_widget_queue_draw_area**.

IV.4.3 Visualización de los Modelos.

La configuración del área de dibujo para la visualización tridimensional se realizó implementando la biblioteca ***GtkGLExt*** sobre el widget de área de dibujo *wPerspectiva*. Se utilizó la función **gtk_widget_set_gl_capability** para establecer la capacidad de uso de instrucciones ***OpenGL*** sobre el *widget* de área de dibujo *drawingareaPerspectiva*.

```
gtk_widget_set_gl_capability (drawingareaPerspectiva,
                              glconfig,
                              NULL,
                              TRUE,
                              GDK_GL_RGBA_TYPE);
```

Una vez que se ha establecido la capacidad del *widget* para el uso de OpenGL, se utiliza la señal *'realize'* del área de dibujo para configurar los parámetros de visualización. La señal '*realize*' se emite cuando se crean en memoria los recursos gráficos del componente o *widget*. Para configurar el entorno de visualización de OpenGl se mandan llamar a la funciones **gdk_gl_drawable_gl_begin** y **gdk_gl_drawable_gl_end**, y entre estas llamadas se puede utilizar funciones del API de OpenGL.

```
void
on_drawingareaPerspectiva_realize    (GtkWidget      *widget,
                                      gpointer       user_data)
{
  GdkGLContext *glcontext = gtk_widget_get_gl_context (widget);
  GdkGLDrawable *gldrawable = gtk_widget_get_gl_drawable (widget);
  GdkGLProc proc = NULL;

  if (!gdk_gl_drawable_gl_begin (gldrawable, glcontext))
    return;

   ...

  glEnable (GL_DEPTH_TEST);
  glDepthFunc (GL_LEQUAL);
  glClearColor (0.0, 0.0, 0.5, 0.0);
  gdk_gl_glPolygonOffsetEXT (proc, 1.0, 1.0);
```

```
  glEnable (GL_CULL_FACE);
  glHint (GL_LINE_SMOOTH_HINT, GL_NICEST);
  glHint (GL_POLYGON_SMOOTH_HINT, GL_NICEST);
  glHint (GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST);
  glEnable (GL_COLOR_MATERIAL);
  glColorMaterial (GL_FRONT, GL_DIFFUSE);
  glLightfv (GL_LIGHT0, GL_POSITION, lightPosition);
  glEnable (GL_LIGHT0);
  glShadeModel (GL_FLAT);

  gdk_gl_drawable_gl_end (gldrawable);
  /*** OpenGL END ***/
}
```

El evento de configuración manejado por **on_drawingareaPerspectiva_configure_event** ocurre cuando existe una modificación a las dimensiones del área de dibujo, por lo cual se utiliza para reconfigurar los parámetros de la puerto de visualización con respecto a la relación de aspecto.

```
gboolean on_drawingareaPerspectiva_configure_event (GtkWidget      *widget,
                                                    GdkEventConfigure *event,
                                                    gpointer       user_data)
{
  GdkGLContext *glcontext = gtk_widget_get_gl_context (widget);
  GdkGLDrawable *gldrawable = gtk_widget_get_gl_drawable (widget);

  GLfloat w = widget->allocation.width;
  GLfloat h = widget->allocation.height;

  /*** OpenGL BEGIN ***/
  if( !gdk_gl_drawable_gl_begin(gldrawable, glcontext) )
    return FALSE;

  aspect = (float)w/(float)h;
  glViewport (0, 0, w, h);

  gdk_gl_drawable_gl_end (gldrawable);
  /*** OpenGL END ***/
  return FALSE;
}
```

El evento de exposición se maneja con la función de *retrollamada* **on_drawingareaPerspectiva_expose_event** en la cual se manejara la rutina de dibujo de la superficie del modelo en edición.

```
gboolean on_drawingareaPerspectiva_expose_event (GtkWidget      *widget,
                                                 GdkEventExpose  *event,
                                                 gpointer       user_data)
{
  GdkGLContext *glcontext = gtk_widget_get_gl_context(widget);
  GdkGLDrawable *gldrawable = gtk_widget_get_gl_drawable(widget);

  /*** OpenGL BEGIN ***/
```

```
if (!gdk_gl_drawable_gl_begin(gldrawable, glcontext))
  return FALSE;

glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
glMatrixMode (GL_PROJECTION);
...
glColor3f(1.0, 1.0, 1.0);
if( curr_BM )
{
  triangulo *tri;
  GSList *tmpTri = Triangulos;
  while(tmpTri)
  {
    tri = tmpTri->data;
    glBegin(iGlModoDibujo);
      glVertex3f(tri->v[2].x, tri->v[2].y, tri->v[2].z );
      glVertex3f(tri->v[1].x, tri->v[1].y, tri->v[1].z );
      glVertex3f(tri->v[0].x, tri->v[0].y, tri->v[0].z );
    glEnd();
    tmpTri = tmpTri->next;
  }
}
 ...
gdk_gl_drawable_gl_end (gldrawable);
/*** OpenGL END ***/
return TRUE;
}
```

La llamada a la función **Marching_Cube** en la rutina **on_drawingareaX_Y_button_release_event** devuelve una lista enlazada del con variables del tipo ***triangulo*** que es la estructura del polígono a representar en la pantalla de visualización tridimensional. Dentro de la función de exposición se recorre la lista enlazada '*Triangulos*' para graficar los polígonos dentro de las funciones de ***OpenGL*** *glBegin(...)* y *glEnd()*.

La Función **Marching_Cube** genera la superficie del modelo mediante el algoritmo de poligonización ***marching cube*** explicado en el capitulo anterior. Recibe como argumento el modelo actual en edición y un valor numérico entero para la resolución de la malla de las superficie a calcular y la referencia a la lista enlazada '*Triangulos*'. Utiliza una estructura **cube** que almacena las posiciones del los vértices y el valor del campo escalar en cada vértice.

```
typedef struct{
  point3D p[8];
  double val[8];
}cube;
```

Se buscan los límites del paralelepípedo que contiene a los componentes del modelo y se descompone en una cantidad de cubos por cada eje igual a la resolución, y se calcula el método de poligonización para cada cubo en el volumen del paralelepípedo.

```
GSList* Marching_Cube( blob_modelo *BlbM, gint iResolucion, GSList *Triangulos)
{
 GSList *Trian = NULL;
 cube cubo;
 gdouble xi, yi, zi, xf, yf, zf;
 gdouble xc, yc, zc;
 gdouble isovalue, fSize;
 gint i;
 g_slist_free(Triangulos);
  fSize = Limites_Modelo( BlbM, iResolucion, &xi, &yi, &zi, &xf, &yf, &zf );
  zc = zi;
  do{       yc = yi;
    do{     xc = xi;
      do {
        cubo.p[0].x = xc;
        cubo.p[0].y = yc;
        cubo.p[0].z = zc;
        ...
        cubo.p[7].x = xc;
        cubo.p[7].y = yc + fSize;
        cubo.p[7].z = zc + fSize;
        for (i=0;i<8;i++)
          cubo.val[i] = (gdouble) isoValor( cubo.p[i], BlbM );
        Trian = g_slist_concat ( Trian, Poligonizar( cubo, BlbM->umbral ) );
        xc += (gdouble)fSize;
      }while( xc < xf );
      yc += (gdouble)fSize;
    }while( yc < yf );
    zc += (gdouble)fSize;
  }while( zc < zf );
 return Trian;
}
```

La rutina **Poligonizar** se encarga de obtener los triángulos del cubo de acuerda al algoritmo presentado en el capítulo III. En el puntero a una lista enlazada '*Trian*' se obtienen la lista de polígonos resultantes y se concatenan con los ya existentes y se regresa la referencia a esta lista enlazada para ser visualizada.

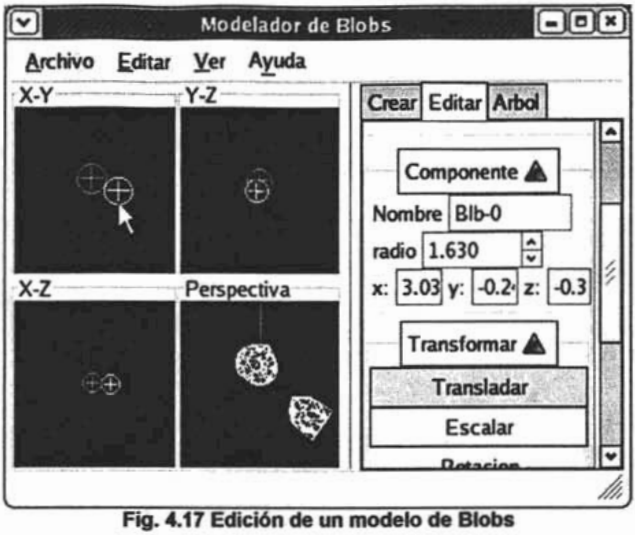


Fig. 4.17 Edición de un modelo de Blobs

CAPÍTULO V

Uso de las herramientas desarrolladas.

V.1. Parches Bézier.

Para probar la operatividad del prototipo se utilizó en el caso de los parches de Bézier un modelo ya creado previamente.

Se utilizo la tetera de Utah de Martin Newell, que viene como ejemplo en los archivos de ejemplo de **POVRAY**[21] en la ruta de ".../povray/scenes/advanced/teapot"[22].

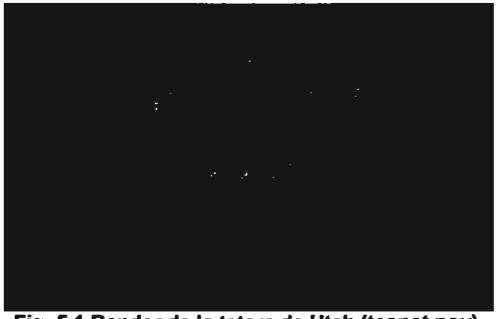


Fig. 5.1 Render de la tetera de Utah (teapot.pov).

El modelo viene definido en el lenguaje de ***povray*** en un archivo cabecera de nombre "teapot.inc" en el cual están definidos los 32 parches de Bezier que componen el modelo completo.

La adaptación de la definición de los parches de Bezier en povray a un formato entendible para el prototipo se realizo mediante la obtención de los puntos de control.

```
...
bicubic_patch { ...
    <1.40000, 0.00000, 2.40000> <1.40000, -0.78400, 2.40000>
     <0.78400, -1.40000, 2.40000> <0.00000, -1.40000, 2.40000>
    <1.33750, 0.00000, 2.53125> <1.33750, -0.74900, 2.53125>
     <0.74900, -1.33750, 2.53125> <0.00000, -1.33750, 2.53125>
    <1.43750, 0.00000, 2.53125> <1.43750, -0.80500, 2.53125>
     <0.80500, -1.43750, 2.53125> <0.00000, -1.43750, 2.53125>
    <1.50000, 0.00000, 2.40000> <1.50000, -0.84000, 2.40000>
     <0.84000, -1.50000, 2.40000> <0.00000, -1.50000, 2.40000>
  ...
}

bicubic_patch { ...
    <0.00000, -1.40000, 2.40000> <-0.78400, -1.40000, 2.40000>
     <-1.40000, -0.78400, 2.40000> <-1.40000, 0.00000, 2.40000>
    <0.00000, -1.33750, 2.53125> <-0.74900, -1.33750, 2.53125>
     <-1.33750, -0.74900, 2.53125> <-1.33750, 0.00000, 2.53125>
    <0.00000, -1.43750, 2.53125> <-0.80500, -1.43750, 2.53125>
```

[21] POVRAY (Persistence of Vision Ray-Tracer). Programa Freeware para producir graficas tridimensionales. http://www.povray.org.

[22] Obtenido de los archivos resultado de la instalación del programa POVRAY.

```
    <-1.43750, -0.80500, 2.53125> <-1.43750, 0.00000, 2.53125>
   <0.00000, -1.50000, 2.40000> <-0.84000, -1.50000, 2.40000>
    <-1.50000, -0.84000, 2.40000> <-1.50000, 0.00000, 2.40000>
  ...
}
...
```

Se tomaron los puntos de control de cada parche bézier y se guardaron en un formato que pudiera interpretar la aplicación de manera correcta.

Para lo cual se agrego al proyecto las funciones de ***guarda_BEZ()*** y ***abre_BEZ()*** las cuales guardan y abren respectivamente un archivo con un formato especifico y con extensión 'BEZ'.

El formato aunque temporal[23] es funcional y simple y se estructura de la siguiente manera.

1. Los primeros cuatro bytes definen un entero que indica cuantos parches bézier contiene el archivo.
2. Una serie del tamaño de número anterior de bloques de bytes que contienen la información básica de los parches bézier:
   a. Un registro de 20 bytes que contiene la cadena de identificación del parche bézier.
   b. Cuatro bytes para almacenar un entero que indica en cuantos segmentos se dividirá el parámetro 'u'.
   c. Cuatro bytes para almacenar un entero que indica en cuantos segmentos se dividirá el parámetro 'v'.
   d. 16 registros de tipo 'point3D' que representan los 16 vértices de control de la superficie de bézier.

Estas funciones están implementadas en el archivo cabecera exportaPov.c.

```
int guarda_BEZ( GSList *Lista,   char *strNombre )
{
 GSList *tmpLista = NULL;
 FILE *fpBez;
 char strNomBez[200];
 parche_bezier *rPb;
 int n, u, v;
```

[23] Se considera utilizar a futuro un formato basado en XML el cual no se incluye su análisis por quedar fuera del alcance de este trabajo.

```
  sprintf(strNomBez, "%s.bez", strNombre);    g_print("Guardando %s.bez \n", strNombre);
   if( (fpBez = fopen( strNomBez, "w" )) != NULL )
   {
    n = g_slist_length(Lista);
    fwrite( &n, sizeof(n), 1, fpBez);
    tmpLista = Lista;
    while(tmpLista)
    {
     rPb = tmpLista->data;
     fwrite( &rPb->nombre, sizeof(rPb->nombre), 1, fpBez);
     fwrite( &rPb->n_u, sizeof(rPb->n_u), 1, fpBez);
     fwrite( &rPb->n_v, sizeof(rPb->n_v), 1, fpBez);
     for(u=0; u<MAX_BEZIER; u++)
      for(v=0; v<MAX_BEZIER; v++)
       fwrite( &rPb->v_c[u][v], sizeof(point3D), 1, fpBez);
     tmpLista = tmpLista->next;
    }

    fclose(fpBez);
    return 1;
   }
  return 0;
}


GSList* abre_BEZ( GSList *Lista,    char *strNombre )
{
  GSList *tmpLista = NULL;
  FILE *fpBez;
  parche_bezier *rPb;
  int n, u, v, i;
   if(Lista)
   {
    tmpLista = Lista;
    while(tmpLista)
    {
     rPb = tmpLista->data;
     free(rPb->p);
    }
    g_slist_free(Lista);
    tmpLista = NULL;
   }

   if( (fpBez = fopen( strNombre, "r" )) != NULL )
   {
    fread( &n, sizeof(n), 1, fpBez);
    for(i=0; i<n; i++)
    {
     rPb = g_malloc( sizeof(parche_bezier) );
        rPb->p = NULL;
     fread( &rPb->nombre, sizeof(rPb->nombre), 1, fpBez);
     fread( &rPb->n_u, sizeof(rPb->n_u), 1, fpBez);
     fread( &rPb->n_v, sizeof(rPb->n_v), 1, fpBez);
     for(u=0; u<MAX_BEZIER; u++)
      for(v=0; v<MAX_BEZIER; v++)
       fread( &rPb->v_c[u][v], sizeof(point3D), 1, fpBez);
     Parche_Bezier_set(rPb);
     tmpLista = g_slist_append(tmpLista, rPb);
```

```
    }

    fclose(fpBez);
    return tmpLista;
   }
  return NULL;
}
```

Además se implementaron las funciones del archivo callbacks.c con el control ***gtk_file_chooser***.

```
void on_abrir1_activate  (GtkMenuItem     *menuitem,   gpointer       user_data)
{
   ...
   dialog = gtk_file_chooser_dialog_new ("Abrir modelo...",
                                         NULL,
                                         GTK_FILE_CHOOSER_ACTION_OPEN,
                                         GTK_STOCK_OPEN, GTK_RESPONSE_ACCEPT,
                                         GTK_STOCK_CANCEL, GTK_RESPONSE_CANCEL,
                                         NULL);
   if (gtk_dialog_run (GTK_DIALOG (dialog)) == GTK_RESPONSE_ACCEPT)
   {
    char *filename;
    filename = gtk_file_chooser_get_filename (GTK_FILE_CHOOSER (dialog));

    lista = abre_BEZ(lista, filename);
    ...
   }
   ...
}


void on_guardar1_activate  (GtkMenuItem     *menuitem,   gpointer       user_data)
{
   ...
   dialog = gtk_file_chooser_dialog_new ("Guardar modelo...",
                                         NULL,
                                         GTK_FILE_CHOOSER_ACTION_SAVE,
                                         GTK_STOCK_OPEN, GTK_RESPONSE_ACCEPT,
                                         GTK_STOCK_CANCEL, GTK_RESPONSE_CANCEL,
                                         NULL);
   if (gtk_dialog_run (GTK_DIALOG (dialog)) == GTK_RESPONSE_ACCEPT)
   {
    char *filename;
    filename = gtk_file_chooser_get_filename (GTK_FILE_CHOOSER (dialog));

    guarda_BEZ(lista, filename );
    ...
   }
   ...
}
```

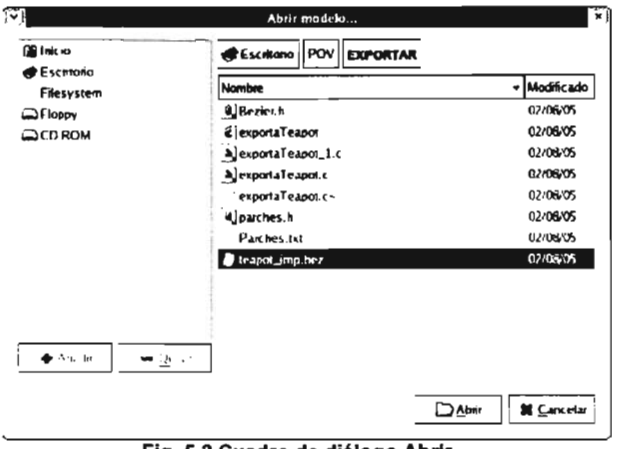


Fig. 5.2 Cuadro de diálogo Abrir...

Una vez exportado el modelo de la tetera al archivo con extensión '.BEZ', lo cargamos a la aplicación para su visualización y obtenemos la lista de parches que lo componen en el árbol de objetos para su manipulación.

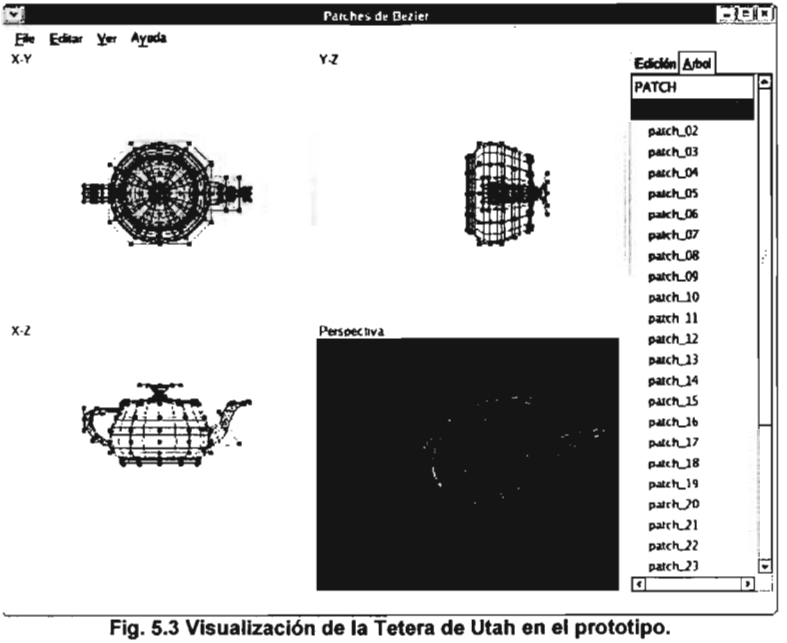


Fig. 5.3 Visualización de la Tetera de Utah en el prototipo.

A continuación se procedió a modificar el modelo cargado al modelador. Modelando a partir de la tetera algo muy parecido a un jarrón. Esto implica modificar los parches de la parte inferior del modelo para aumentarlo en su altura.

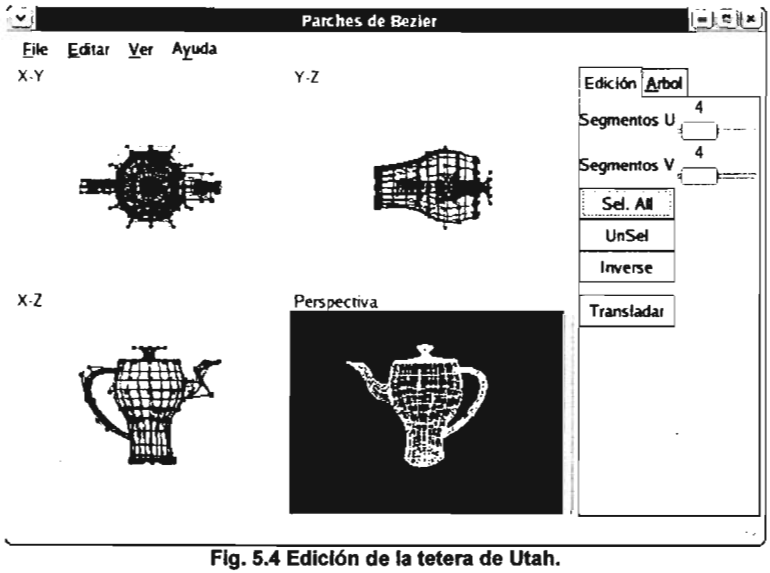


Fig. 5.4 Edición de la tetera de Utah.

También es necesario modificar los parches correspondientes al pico vertedor para modelar el asa opuesta.

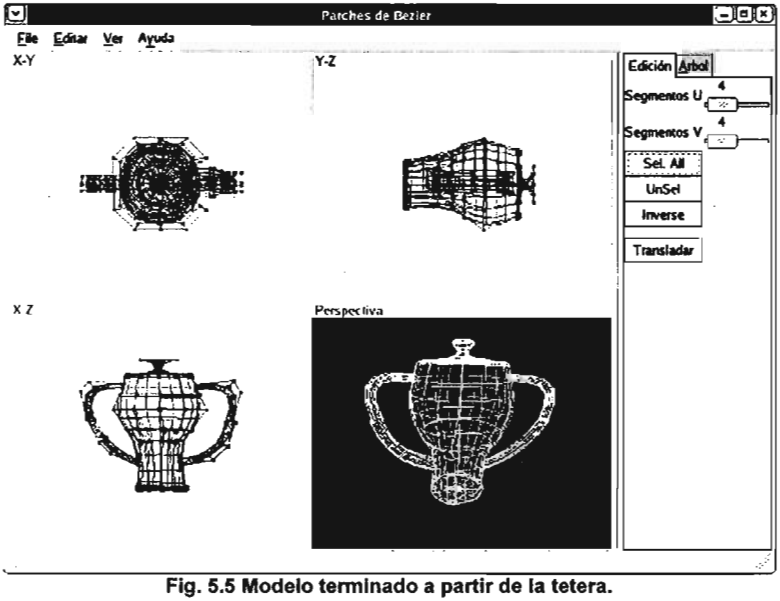


Fig. 5.5 Modelo terminado a partir de la tetera.

Por último se agrego una rutina que sea capaz de exportar la estructura de datos del modelo a formato del lenguaje de POVRAY. Para eso se uso el siguiente esquema:

- Un archivo con extensión 'INI' donde se especifican los valores de configuración para la composición de la escena, como por ejemplo el ancho y alto de la imagen, nombre del archivo 'POV' que contiene la escena.
- Un archivo con extensión 'POV' donde se definen los objetos de la escena, como por ejemplo la cámara, los puntos de luz, etc.
- Un archivo con extensión 'INC' que se utiliza como archivo cabecera donde se definen objetos auxiliares en la escena. Es aquí donde definiremos el objeto creado en la aplicación. Y se agregará la referencia en el archivo con extensión 'POV'.

```
int crear_INC( GSList *Lista,    char *strNombre )
{
 GSList *tmpLista = NULL;
 FILE *fpINC;
 char strNomINC[200];
 parche_bezier *rPb;
 int u, v;
  sprintf(strNomINC, "%s.inc", strNombre);
  if( (fpINC = fopen( strNomINC, "w" )) != NULL )
  {
   tmpLista = Lista;
   while(tmpLista)
   {
    rPb = tmpLista->data;          fprintf(fpINC, "mesh{ // %s \n", rPb->nombre);
    for(v=0; v<rPb->n_v; v++)
    for(u=0; u<rPb->n_u; u++)
    {
     fprintf(fpINC, "   triangle{ <%f, %f, %f>, <%f, %f, %f>, <%f, %f, %f> } \n",
          rPb->p[(rPb->n_v+1)*u+v].x,  rPb->p[(rPb->n_v+1)*u+v].y, rPb->p[(rPb->n_v+1)*u+v].z,
          rPb->p[(rPb->n_v+1)*u+v+1].x,  rPb->p[(rPb->n_v+1)*u+v+1].y,  rPb->p[(rPb->n_v+1)*u+v+1].z,
          rPb->p[(rPb->n_v+1)*(u+1)+v+1].x,   rPb->p[(rPb->n_v+1)*(u+1)+v+1].y,
          rPb->p[(rPb->n_v+1)*(u+1)+v+1].z );
     fprintf(fpINC, "   triangle{ <%f, %f, %f>, <%f, %f, %f>, <%f, %f, %f> } \n",
          rPb->p[(rPb->n_v+1)*(u+1)+v+1].x, rPb->p[(rPb->n_v+1)*(u+1)+v+1].y,
          rPb->p[(rPb->n_v+1)*(u+1)+v+1].z, rPb->p[(rPb->n_v+1)*(u+1)+v].x,
          rPb->p[(rPb->n_v+1)*(u+1)+v].y, rPb->p[(rPb->n_v+1)*(u+1)+v].z,
          rPb->p[(rPb->n_v+1)*u+v].x, rPb->p[(rPb->n_v+1)*u+v].y, rPb->p[(rPb->n_v+1)*u+v].z  );
    }
    fprintf(fpINC, "  } // %s \n", rPb->nombre);
    tmpLista = tmpLista->next;
   }
   fclose(fpINC);        return 1;
  }
 return 0;
}
```

Fig. 5.6 Comando del menú para exportar generar el render del modelo.

Esta rutina se ejecuta desde el menú Archivo -> Exportar POV. El cual pide el nombre del archivo a guardar mediante un ***gtk_file_chooser*** y guarda la estructura de archivos para generar la imagen en **POVRAY** y ejecuta el comando de povray para visualizar la composición de la escena.

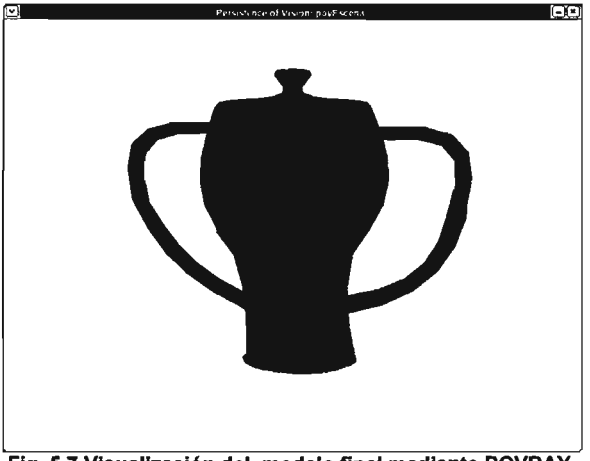


Fig. 5.7 Visualización del modelo final mediante POVRAY.

V.2. Blobs.

Para probar la operatividad del prototipo en el caso del modelado de Blobs se implemento el diseño de un modelo de un ave, como algo representativo del modelado orgánico.

De manera análoga al prototipo de los parches de Bézier, se agregaron las funciones ***guarda_Blob()*** y ***abre_Blob()*** los cuales abren y guardan los archivos que contienen los modelos y que se estructuran de la siguiente manera.

1. Los primeros cuatro bytes definen un entero que indica cuantos modelos contiene el archivo.
2. Una serie del tamaño de número anterior de bloques de bytes que contienen la información básica de los modelos de Blobs:
   a. Un registro de 20 bytes que contiene la cadena de identificación del modelo d Blobs.
   b. Ocho bytes para almacenar un flotante de doble precisión que indica el valor de umbral para definir la isosuperficie.
   c. Cuatro bytes para almacenar un entero que indica en cuantos componentes posee el modelo.
   d. Una serie del tamaño del número de componentes de bloques de información de los componentes del modelo. Estos bloques se estructura de la siguiente forma.
      I. Un registro de 10 bytes que contiene la cadena de identificación del componente.
      II. Un registro de tipo 'point3D' que representa la posición del componente en el espacio.
      III. Un registro de ocho bytes que almacenan un flotante de doble precisión que contiene el radio de influencia del componente.

```
...
  if( (fpBlb = fopen( strNombre, "r" )) != NULL )
  {
    fread( &n, sizeof(n), 1, fpBlb);
    for(i=0; i<n; i++)
    {
      blbM = g_malloc( sizeof(blob_modelo) );
      blbM->blobs = NULL;
      fread( &blbM->nombre, sizeof(blbM->nombre), 1, fpBlb);
```

```
    fread( &blbM->umbral, sizeof(blbM->umbral), 1, fpBlb);
    fread( &blbM->num_componentes, sizeof(blbM->num_componentes), 1, fpBlb);
    free( blbM->blobs );
    blbM->blobs = (blob_componente*) calloc( (size_t)(blbM->num_componentes),
                                             sizeof(blob_componente) );
    for(b=0; b<blbM->num_componentes; b++)
    {
     fread( &blbM->blobs[b].nombre, sizeof(blbM->blobs[b].nombre), 1, fpBlb);
     fread( &blbM->blobs[b].p, sizeof(point3D), 1, fpBlb);
     fread( &blbM->blobs[b].radio, sizeof(blbM->blobs[b].radio), 1, fpBlb);
    }
    tmpLista = g_slist_append(tmpLista, blbM);
   }
   fclose(fpBlb);
   return tmpLista;
...
```

Para comenzar a crear el modelo se iniciara generando el cuerpo principal de manera que a partir de el se puedan derivar las demás extremidades.

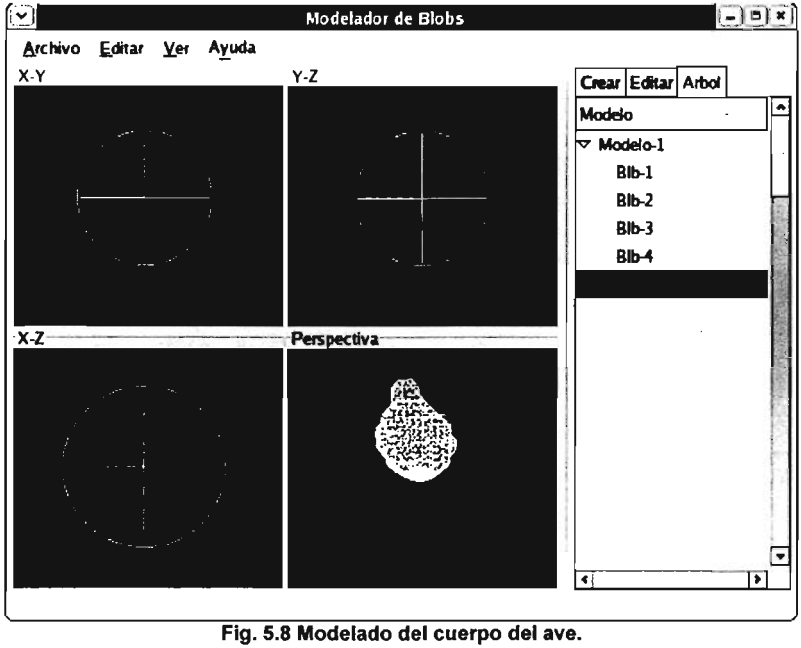


Fig. 5.8 Modelado del cuerpo del ave.

Una vez que se obtuvo el cuerpo podemos comenzar a derivar las partes más difíciles de modelar, la cabeza y las patas.

Modelar las patas del animal es la parte más laboriosa, ya que se caracterizan por ser extremidades muy delgadas y largas, lo cual complica su diseño con blobs, ya que se tiene que implementar un conjunto de componentes muy pequeños para no ensanchar

esa parte del modelo. Pero ello implica que los componentes tengan muy poco radio de influencia, por lo cual se tiene que colocar un gran número de estos componentes a lo largo de una trayectoria, para definir las patas del ave.

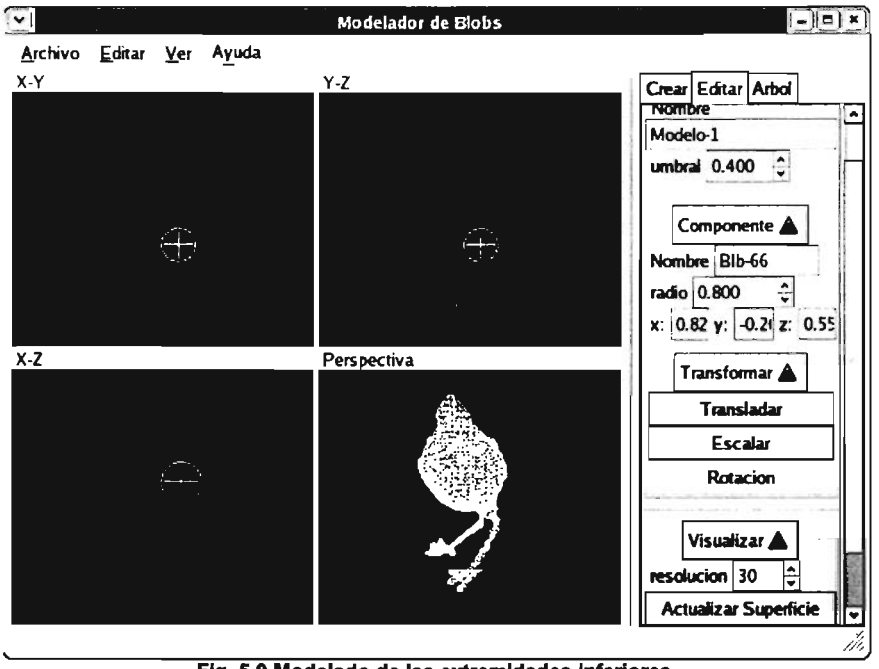


Fig. 5.9 Modelado de las extremidades inferiores.

Por último se modela la cabeza del animal. Esta se compone de la parte del cuello que lo une con el cuerpo principal, la cabeza con los ojos sobresaliendo de ella y el pico.

El cuello solo es una composición simple de componentes, pero la cabeza del animal debe presentar por ojos protuberancias que sobresalgan de ella, ello solo implica posicionar el par de componentes muy cerca de la orilla del blob que forma la cabeza, para que estos contribuyan al campo escalar en esa zona y se produzca el abultamiento deseado.

El pico del animal tiene las mismas singularidades que presentaron las patas, es una extremidad delgada y larga, pero con la particularidad de que disminuye su grosor conforme se aleja de la cabeza, por lo tanto se agruparan lo componentes de igual forma en que se hizo con las patas pero decreciendo su radio de influencia.

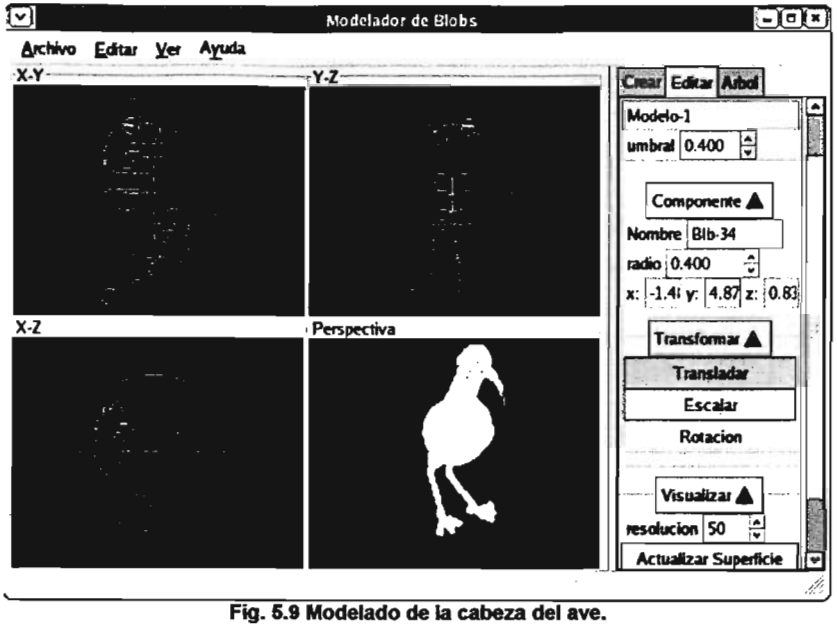


Fig. 5.9 Modelado de la cabeza del ave.

De ésta manera tenemos finalizado el modelo, por lo cual sólo resta exportarlo a formato de POVRAY para su visualización. Se utilizaron el mismo esquema y las mismas funciones que se definieron para los parches de Bézier, solamente cambiando la forma es que se obtienen los vértices de los polígonos en la función ***crear_INC()***.

```
...
  sprintf(strNomINC, "%s.inc", strNombre);
  if( (fpINC = fopen( strNomINC, "w" )) != NULL )
   {
    fprintf(fpINC, "mesh{ // %s \n", blbM->nombre);
    triangulo *tri;
    GSList *tmpTri = Triangulos;
    while(tmpTri)
    {
     tri = tmpTri->data;
     fprintf(fpINC, "    triangle{ <%f, %f, %f>, <%f, %f, %f>, <%f, %f, %f> } \n",
              tri->v[2].x, tri->v[2].y, tri->v[2].z,
              tri->v[1].x, tri->v[1].y, tri->v[1].z,
              tri->v[0].x, tri->v[0].y, tri->v[0].z );
     tmpTri = tmpTri->next;
    }
    fprintf(fpINC, "   } // %s \n", blbM->nombre);
    fclose(fpINC);
...
```

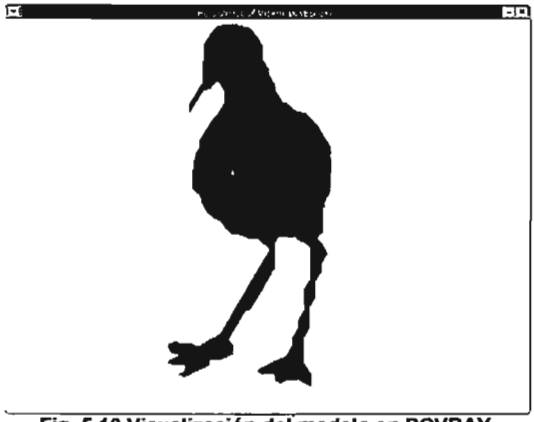


Fig. 5.10 Visualización del modelo en POVRAY.

— ∞ —

CONCLUSIONES

Podemos concluir que los prototipos desarrollados realmente alcanzaron el objetivo inicial, implementar los algoritmos elegidos para el modelado orgánico en diseño de objetos tridimensionales. Con ellos es posible generar de una manera más intuitiva modelos de naturaleza no predecibles en sus formas, que de otra manera no podrían ser diseñadas a base de primitivas.

La construcción de este prototipo contribuyó al desarrollo de aplicaciones graficas en Linux ya que casi no existen herramientas de modelado grafico para esta plataforma, lo cual continúa con la corriente de enriquecer este sistema operativo de uso y distribución gratuita.

El uso de la Biblioteca ***GTK+*** facilita el desarrollo de aplicaciones para ambientes gráficos, siendo este la base del escritorio ***GNOME*** de Linux. El uso de esta biblioteca se ha popularizado por su cada vez más fácil migración a otros sistemas operativos por lo cual se concluye que es una buena opción de desarrollo.

El constructor de interfaces de usuario GLADE es una buena opción cómo utilería RAD, ya que disminuye considerablemente los tiempos de desarrolló, y aunque no ha implementado totalmente las capacidades de GTK+, permite generar aplicaciones con una muy buena calidad.

Parte de la tarea de generar aplicaciones para manejo de objetos tridimensionales es generar la correspondiente visualización de ellos, por ello el uso de las bibliotecas para manejo de gráficos en 3D es indispensable, ya que ahorran tiempo de desarrollo. Es por eso que el uso de las bibliotecas estándar de ***OpenGL***, y su implementación de software libre ***MESA***, mejoraron el desempeño y la calidad de la visualización de los modelos tridimensionales que se manejan en el prototipo.

La elección del lenguaje de programación C estándar se puede considerar una buena opción, ya al estar desarrollándose las versiones de las bibliotecas de GTK+ a otras plataformas, Windows por ejemplo, facilita la migración. Además las versiones de GTK+ para C++, aunque ya existen, aún se encuentra en etapa de desarrollo. También permite la fácil asimilación del prototipo para un futuro enriquecimiento del mismo.

Además al desarrollar estas herramientas de modelado tridimensional se demostró que teniendo el algoritmo correcto y haciendo un análisis eficiente del mismo es posible implementar herramientas de modelado que de otra forma sólo se tendría acceso mediante programas propietario, lo cual implica el pago por derechos de uso. Y es por eso que esta tesis ha demostrado que el desarrollo de software libre y de código abierto ayudan a enriquecer, no sólo proyectos como Material 3D, sino a toda la comunidad de desarrollo de software, ya sea propietario o de libre distribución, ya que puede llegar a aportar ideas de desarrollo para futuros proyectos o enriquecer los ya existentes.

—∞—

BIBLIOGRAFIA

- **Programación de Gráficos en 3D.** Manuel Escribano Cauqui. Edit. Addison-Wesley Iberoamericana. 1ª ed., E.U.A. 1995, 226 pp.
- **Introducción a la graficación por computador.** James D. Foley, et. al. Edit. Addison-Wesley Iberoamericana. 1ª ed., E.U.A. 1996, 650 pp.
- **Graficas por computadora.** Donald Hern. M. Pauline Baker. Edit. Prentice Hall Hispanoamericana. 2ª ed., México, 1995, 686 pp.
- **3D Modeling & Surfacing.** Bill Fleming. Edit. Morgan Kaufman. 1ª ed., E.U.A., 1999, 340 pp.
- **Digital 3D Design.** Simon Danaher. Edit. Watson-Guptill Publications. 1ª ed., E.U.A., 2001, 192 pp.
- **3D Computer Graphics.** Alan Watt. Edit. Addison-Wesley. 3ª ed., E.U.A., 2000, 569 pp.
- **Curvas y Superficies para Modelado Geométrico.** Juan Manuel Cordero Valle, José Cortés Parejo. Edit. Alfaomega Grupo Editor. 1ª ed., México, 2003, 441 pp.
- **Fisica. Tomo II.** Raymond A. Serway. Edit. McGraw Hill. 3ª ed., México, 1994, 1463 pp.
- **Programación en Linux.** Kurt Wall, et al. Edit. Prentice Hall. 2ª ed., Madrid, 2001, 846 pp.
- **Desarrollo de Aplicaciones Linux con GTK+ y GDK.** Eric Harlow. Edit. Prentice Hall. 1ª ed., Madrid, 1999, 482 pp.

- http://www.acm.org/crossroads/espanol/xrds3-3/color.html Técnicas y Herramientas para Usar Color en el Diseño de la Interfaz de una Computadora. Peggy Wright, Diane Mosser-Wooley, y Bruce Wooley
- http://astronomy.swin.edu.au/~pbourke/modelling/implicitsurf/ Superficies Implicitas. Descripción de las superficies implicitas mesdiante campos escalares. Paul Bourke, Junio 1997
- http://astronomy.swin.edu.au/~pbourke/modelling/polygonise/ Poligonizar un campo escalar. Descripción del algoritmo Marching Cube. Paul Bourke, Mayo 1997.

- http://www.gnu.org/
  Sistema Operativo GNU. Sistemas GNU/Linux.

- http://www.gnome.org/
  GNOME: El proyecto de Escritorio de softwarelLibre. Escritorio y plataforma de desarrollo de linux.

- http://www.gimp.org/
  GIMP. GNU Image Manipulation Program. Programa de manipulación de imágenes.

- http://gtk.org/
  EL conjunto de Utilerias GIMP. Conjunto de Utilerias multplataforma para crear interfaces graficas de usuario.

- http://glade.gnome.org/
  Glade – Un Constructor de Interfaces de Usuario para GTK+ y GNOME.

- http://anjuta.org/
  Anjuta DevStudio. Proyecto de IDE para C y C++ para GNU/Linux.

- http://www.opengl.org/
  Estandar Industrial multiplataforma para el desarrollo de graficas en 2D y 3D.

- http://www.mesa3d.org/
  Blibioteca Grafica MESA 3D. Implementación libre de la API de OpenGL.

- http://gtkglext.sourceforge.net/
  GtkGLExt. GtkGLExt es una extensión de OpenGL para GTK+ 2.0 o superior.

- http://povray.org/
  Persistente of Vision Ray-Tracer. Apliacación para generación de graficas tridimensionales.

- http://tigre.aragon.unam.mx/m3d/
  Sitio del proyecto Material 3D.

— ∞ —