20

UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

ESCUELA NACIONAL DE ESTUDIOS PROFESIONALES
CAMPUS ARAGÓN

"DISEÑO DE UN EDITOR DE TRAYECTORIAS DE OBJETOS TRIDIMENSIONALES"

T E S I S

QUE PARA OBTENER EL TÍTULO DE:
INGENIERO EN COMPUTACIÓN

P R E S E N T A :

JESÚS HERNÁNDEZ CABRERA

ASESOR DE TESIS:
M. EN C. MARCELO PÉREZ MEDEL

MÉXICO, 2002.

Dedicatoria.

A mis padres Cleotilde Cabrera Ortega
y Ángel Hernández Cruz , por la grandeza, sacrificio
y tolerancia de todos los días,
por ser sol del amanecer y luna entre las estrellas.

Doy gracias a dios por permitirme ser en este espacio y permitirme llagar a este momento, un paso más en mi vida.

Doy gracias a mis padres por su sacrificio y sepan que estoy muy orgulloso de sus logros, esta tesis uno de ellos.

Doy gracias a mis hermanas y hermanos quienes me han apoyado incondicionalmente aún en tiempos adversos.

Doy gracias a mis cuñadas y cuñados por ser como son.

Doy gracias al M. en C. Marcelo Pérez quien además de ser mi asesor y profesor, ha sabido ser buen amigo, que con su experiencia y apoyo me supo guiar e impulsar a un constante mejoramiento.

Doy gracias a mis sobrinos, luz de mis días que diluye todo mal.

Doy gracias mis amigos, por contribuir con granitos de arena para mejorar como persona.

Doy gracias a mis revisores de tesis: Mat. Luis Flores, Ing. Arturo Ocampo, Ing. Liliana Hernández e Ing. Martín Solís Pérez por tomarse el tiempo.

Doy gracias a la Universidad Nacional Autónoma de México, mi alma mater.

ÍNDICE

INTRODUCCIÓN

Algunas de las aplicaciones típicas de la animación generada por computadora son el entretenimiento (películas y dibujos animados), la publicidad, visualización de estudios científicos y de ingeniería, capacitación y educación. Muchas de estas aplicaciones requieren que los despliegues en los movimientos y trayectorias de objetos sean realistas, lo cual no siempre se puede lograr por limitaciones en las capacidades del software existente. Una de estas limitaciones es el cálculo de trayectorias y determinación de la velocidad de objetos, ya que actualmente en algunos programas estos se tienen que hacer manualmente, en otros como 3D Studio y Maya si contienen un editor de trayectorias a través de una interfaz gráfica de usuario, pero aún tienen deficiencias como la limitación para definir diferentes tipos de trayectorias; además de ser caros y funcionar bajo un solo sistema operativo.

La experiencia que se adquiere en la creación de animaciones digitales utilizando diferente software nos ha dado un panorama general de las necesidades existentes, y nos han permitido determinar cuales de estas necesidades es preciso satisfacer. Para ilustrar estas deficiencias mencionaremos los problemas a los cuales nos enfrentamos en un proyecto de animación desarrollado en el Centro Tecnológico Aragón. El primer gran inconveniente fue la decisión de que software utilizar, debido a que no contábamos con licencias de Maya o 3D Studio optamos por la utilización de Software Libre, tal es el caso de Pov-Ray y Moray. Con esta elección solucionábamos el costo de la licencia pero agregaba otros inconvenientes como a continuación se platica.

Gran parte del tiempo fue empleado en el cálculo de trayectorias ya que desgraciadamente para la animación fue necesario editar un guión(script) manualmente, haciendo cálculos en papel lo cual hace que el movimiento de elementos de la escena no sea del todo realista, además para determinar la velocidad del movimiento de los objetos es también necesario calcularlos manual e individualmente, de esta manera difícilmente se logra la sincronización de movimientos y esto hace que la calidad de la animación no sea satisfactoria. Otro de los inconvenientes de este software es que solo se puede hacer uso de una sola variable de entorno, la variable clock de Pov-Ray [R2] lo cual dificulta un poco más el cálculo de trayectorias, además de no ofrecer flexibilidad en la forma de introducir las trayectorias.

Dentro del equipo de trabajo de esta animación consideramos necesaria la programación de un nuevo módulo para facilitar la edición de trayectorias para la creación de animaciones por computadora, ya que con base a la experiencia antes mencionada, se ha detectado una enorme perdida de tiempo en el cálculo de las trayectorias de cada uno de los objetos, además de la necesidad de tener que hacerlo por separado de la fase de modelado, cuando debería tener la capacidad de editar las trayectorias en la etapa de modelado y obtener como salida los datos que se utilizarán en la generación de las imágenes digitales, para así obtener mejor realismo en la calidad de las animaciones.

Consideramos también importante disminuir el costo de software y brindar flexibilidad al sistema al proporcionar el código fuente, es por eso que se propone hacer que dicho software funcione bajo el concepto de licencia GPL[1], formando parte del proyecto Material 3D[2], que entre otras características ofrece gran portabilidad a diferentes plataformas que funcionan con diferentes sistemas operativos.

Delimitación del problema

Como se menciono en la justificación existe una gran cantidad de perdida de tiempo en el cálculo de trayectorias, lo cual representa un problema para un grupo de personas que se dediquen a la animación por computadora, ya que el tiempo utilizado para la generación digital de imágenes es elevada, debido a la cantidad de cálculos que debe de hacer el procesador para cada píxel de la imagen y esta complejidad crece proporcionalmente con respecto a la cantidad de objetos que contenga dicha escena. Debido a esto es necesario buscar alternativas para economizar tiempo y para solucionarlo se puede recurrir a la utilización de software propietario que permitan la edición de trayectorias, tal es el caso de 3d Studio y Maya[3]. Pero desgraciadamente el precio de la licencia para el uso de dicho software es muy alto y además están limitados a funcionar en muy pocas plataformas (Windows y Silicon graphics principalmente) y esto presenta una limitante para la transportabilidad de los modelos, que a menudo es necesaria para permitir el uso de computadoras más rápidas o económicas.

[1] GPL. General Public License [R8]
[2] Material 3D. Proyecto de software libre para desarrollar un programa de modelado y animación 3D. [R9]
[3] Software propietario de Alias Wavefront. [RX]

Así podemos decir que además de la necesidad existente de economizar el tiempo de creación de una animación, es necesario disminuir costos e integrar la característica de portabilidad.

La presente tesis busca solucionar los tres problemas enfocándose en la solución de uno de ellos, aunque parezca contradictorio.

¿Cómo lograr esto?

Es claro darse cuenta que la solución al costo de licencia se soluciona utilizando software del proyecto GNU, afortunadamente existe un proyecto bajo esta filosofía que se ajusta a estas necesidades, el cual nació en el ámbito universitario concentrándose principalmente en el Centro Tecnológico Aragón. Ya que además de ser un software de licencia libre esta proyectado para poder ser recompilado en diferentes plataformas, gracias a las bibliotecas glibc. Este proyecto se llama Material 3D busca crear un software para el modelado en 3D con librerías Mesa[4] y GTK+ (Ambas de licencia libre)

Otra característica importante es que contará con tecnología de módulos insertables. Con esta característica nuestro problema se limita a solamente programar el módulo editor de trayectorias para que funcione como parte del proyecto Material 3D, ya que el resto lo soluciona Material 3D por sí solo. A continuación se dará una breve descripción de lo que abordaremos en cada capítulo.

En el primer capítulo se repasan los conceptos básicos necesarios para el desarrollo de nuestra aplicación. En la primera parte definimos el espacio tridimensional de manera formal, así como las fórmulas matemáticas que utilizaremos en capítulos posteriores, además definimos el método utilizado para la representación de objetos tridimensionales en el monitor. En la segunda parte hacemos un repaso breve a las diferentes etapas para la creación de animaciones por computadora. Y en la tercera parte hacemos un análisis a las características del software de modelado en 3D existente, para determinar los aspectos a solucionar.

En el segundo capítulo explicamos la forma en que editaremos las trayectorias, para posteriormente codificarlas en lenguaje "C" y librerías GTK+.

En el tercer capítulo definimos las estructuras de datos a utilizar, así como la estructura de los archivos que manejará nuestra aplicación. Debido a que la programación GTK+

[4] Mesa: Implementación libre de OpenGL

es relativamente nueva, hacemos una breve introducción con ejemplos sencillos para mejor entendimiento de nuestro editor. Una vez hecho lo anterior ya contamos con todo lo necesario para realizar la programación de nuestra aplicación que es abordada en toda la parte final de este capítulo.

En el cuarto capítulo damos solución a un caso práctico sencillo, con el único fin de probar nuestra aplicación.

En el quinto y último capítulo analizamos los resultados del caso práctico contrastándolo con en desarrollo de una animación hecho en su totalidad con software preexistente, para determinar la eficiencia de nuestro editor de trayectorias.

OBJETIVO GENERAL

- Diseñar un editor de trayectorias tridimensionales que permita diferentes métodos para la edición de trayectorias, para ahorrar tiempo, sincronizar movimientos y mejorar la calidad de las animaciones.

OBJETIVOS PARTICULARES

- Definir los conceptos básicos necesarios de la animación por computadora, así como determinar las deficiencias del software existente.
- Definir y delimitar los tipos de trayectorias que el programa realizará.
- Determinar las estructuras de datos óptimas para la representación de trayectorias, así como la programación de estas en GTK+.
- Conocer las herramientas GUI que ofrece GTK+.
- Realizar ejemplos sencillos de animación para determinar la eficiencia de la aplicación.
- Evaluar resultados del caso práctico, tanto en tiempo de creación y calidad de la animación.

CAPÍTULO I .- ANTECEDENTES

Objetivo:

- Definir los conceptos básicos necesarios de la animación por computadora, así como determinar las deficiencias del software existente.

Antes de entrar más a detalle al tema que nos ocupa es necesario establecer conceptos para elaborar y definir claramente el proyecto de tesis [4]. A continuación se definirán los conceptos básicos necesarios que se utilizarán en capítulos posteriores.

1.1 .-ESPACIO TRIDIMENSIONAL

El espacio en tres dimensiones es empíricamente conocido por nosotros, ya que interactuamos en un mundo en tres dimensiones, este conocimiento empírico no es posible que una computadora lo adquiera, o por lo menos no se ha logrado que se iguale el nivel de abstracción del cerebro humano, es por eso que es necesario introducir el conocimiento a una computadora en forma matemática (el lenguaje que mejor maneja). A continuación definiremos el espacio en tres dimensiones formalmente a través de formulas matemáticas.

1.1.1 .-Definición formal

Para facilitar la representación del espacio tridimensional utilizamos un sistema coordenado que consta de tres rectas perpendiculares entre sí, que tienen la misma escala y se interceptan en un punto que llamaremos origen como se muestra en la figura 1.1(a), esta intersección forman tres planos coordenados: el plano xy, el plano xz y el plano yz [11]. Al espacio formado entre estos planos los llamaremos octantes, figura 1.1(b) y la pertenencia de un punto a cualquiera de estos octantes está determinado por el signo (+ ó -) de sus coordenadas. En la presente tesis se utilizará el primer octante para facilitar la comprensión de los ejemplos.

La representación matemática del punto **P** se puede hacer por medio de cualquiera de los sistemas de coordenadas, los mas empleados son: El cartesiano, el esférico y el cilíndrico.

El **sistema de coordenadas cartesiano** se basa en distancias expresadas en unidades de un punto **P(x,y,z)**, en donde **x,y** y **z** pertenecen a los números reales y son las distancias al origen sobre cada uno de los ejes respecto a tres planos perpendiculares entre sí, al eje de las **x** se le conoce como el eje de las ordenadas, al de las **y** como el de las abscisas y al eje **z** se le llama cota.

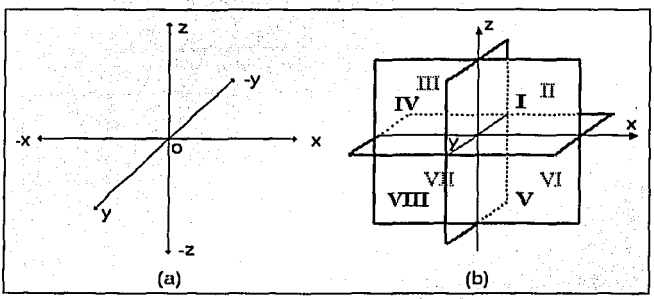

Figura 1.1: Representación del espacio tridimensional

En la figura 1.2(a) se ilustra la representación del punto, donde r es el vector del punto **P(x,y,z)** en un sistema de coordenadas de la "mano derecha" y en la figura 1.2(b) en el sistema de coordenadas de la "mano izquierda".

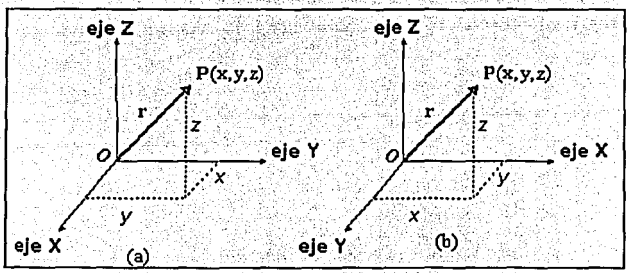

Figura 1.2 : Sistema de coordenadas cartesiano

Es importante hacer notar que el sistema de ejes coordenados utilizado para el desarrollo de esta tesis será el sistema de ejes coordenados de la mano izquierda, que se muestra en la figura 1.2(b).

La distancia entre 2 puntos **P(x,y,z)** y **Q(x,y,z)** esta dado por la fórmula:

$$|PQ| = \sqrt{(x_Q - x_P)^2 + (y_Q - y_P)^2 + (z_Q - z_P)^2} \quad \ldots (1)$$

Utilizando Esta fórmula podemos determinar la distancia del origen al punto P. Es decir:

$$|OP| = \sqrt{(x_P - 0)^2 + (y_P - 0)^2 + (z_P - 0)^2} = \sqrt{x_P{}^2 + y_P{}^2 + z_P{}^2} \quad \ldots(2)$$

Para encontrar las coordenadas del **punto que divide una recta en una razón dada**, figura 1.3, se utilizan las siguientes fórmula[8]:

$$x = \frac{x_{P1} + kx_{P2}}{1+k}, y = \frac{y_{P1} + ky_{P2}}{1+k}, z = \frac{z_{P1} + kz_{P2}}{1+k} \quad \ldots (3)$$

$$\text{Donde } k = \frac{i}{j} \quad \ldots(4)$$

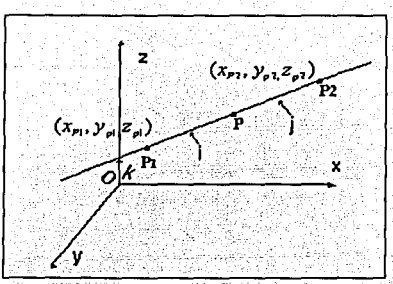

Figura 1.3: Punto que divide un segmento en una razón dada

El punto medio de un segmento de recta esta dado por la formula:

$$x = \frac{x_{P1} + x_{P2}}{2}, y = \frac{y_{P1} + y_{P2}}{2}, z = \frac{z_{P1} + z_{P2}}{2} \quad \ldots(5)$$

El **sistema de coordenadas cilíndricas** se basa en las coordenadas polares φ y ρ con respecto al plano principal (**x** ,**y**), **z** es la cota con el mismo valor del sistema cartesiano, figura 1.4.

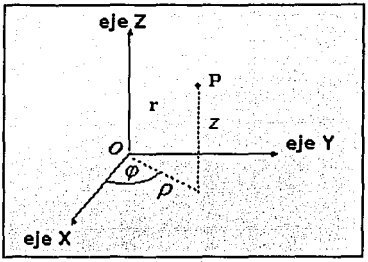

Figura 1.4 : Sistema de coordenadas polares

Las fórmulas para la obtención de las coordenadas cilíndricas(φ y ρ) a partir de las cartesianas y viceversa son:

$$\rho = \sqrt{x^2 + y^2}, \varphi = \tan^{-1}\frac{x}{y} = Sen^{-1}\frac{y}{\rho} \quad ...(6)$$

$$X = \rho \cos \varphi, \quad y = \rho \text{ sen } \varphi, \quad z = z \quad ...(7)$$

El sistema de coordenadas esféricas (o polares) es: r es la longitud del radio vector, φ es la longitud, θ es la distancia polar. En la figura 1.5 se puede observar este sistema y se muestran las direcciones positivas de los valores. Para la obtención univoca de todos los puntos en el espacio, los valores dados en las coordenadas esféricas deben estar en los siguientes límites:

$$0 \le r < \infty \quad , \quad -\pi < \varphi \le \pi \quad , \quad 0 \le \theta \le \pi \quad ...(8)$$

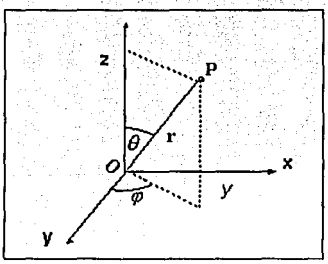

Figura 1.5 Sistema de coordenadas esféricas

Las fórmulas para la obtención de las coordenadas esféricas a partir de las cartesianas y viceversa son:

$$x = r sen\theta \cos\varphi, y = r sen\theta sen\varphi, z = r\cos\theta ...(9)$$

$$r = \sqrt{x^2 + y^2 + z^2}, \varphi = \tan^{-1}\frac{y}{x}, \theta = \tan^{-1}\frac{\sqrt{x^2 + y^2}}{z} \quad ...(10)$$

1.1.2 .-Representación en la computadora

De los sistemas de coordenadas explicados anteriormente, el cartesiano es el más utilizado por su simplicidad para encontrar la posición de un punto en el espacio, pero para la realización de transformaciones de escalamiento, rotación y translación de objetos, el sistema de coordenadas esférica parece ser la mejor opción. Pero para su representación en el monitor todos los sistemas presentan el mismo problema: dado que son sistemas en el espacio R^3 que no pueden ser directamente representados por un dispositivo que maneja gráficos en R^2 . Por lo tanto es necesario "engañar" la vista humana haciendo uso de técnicas de graficación por computadora, como a continuación se explica.

La técnica a utilizar es la graficación de objetos tridimensionales con *perspectiva caballera* [9] que se caracteriza por ser oblicua y cilíndrica; es decir, el efecto se obtiene proyectando el objeto sobre uno de los planos coordenados con líneas de visión paralelas y no perpendiculares a dicho plano. El eje saliente suele proyectarse de manera que forme un ángulo φ con el eje horizontal y además se somete a un factor de escala, que suele ser de 1/2 o 2/3.

Para obtener las coordenadas bidimensionales de la proyección caballera de un punto tridimensional dado por sus coordenadas, plantearemos una matriz de transformación a partir de la definición de perspectiva.

Para obtener la matriz que convierta cada vector de la base R^3 en una combinación lineal de R^2 se realiza el siguiente análisis. Como vemos en la figura 1.6 el eje ***Z*** en R^3 pasa a ser directamente el eje ***y*** en R^2 y el eje ***X*** en R^3 pasa a ser el eje ***x*** en R^2 , es decir:

$$[x \quad y \quad z]*\begin{bmatrix}1 & 0\\ 0 & 0\\ 0 & 1\end{bmatrix} = [X2d, Y2d]$$

como ejemplo convertiremos los dos siguientes vectores: (0,0,1) y (0,0,1)

$$(0,0,1) \rightarrow [0 \quad 0 \quad 1]*\begin{bmatrix} 1 & 0 \\ 0 & 0 \\ 0 & 1 \end{bmatrix} = [0,1] \qquad (1,0,0) \rightarrow [1 \quad 0 \quad 0]*\begin{bmatrix} 1 & 0 \\ 0 & 0 \\ 0 & 1 \end{bmatrix} = [1,0]$$

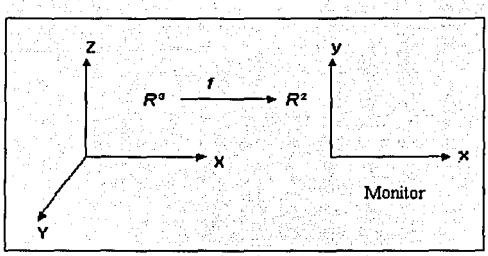

Figura 1.6 :Transformación de R^3 a R^2.

El eje en perspectiva de R^3 , *Y*, se convierte en una combinación lineal de (1,0) y (0,1) fácilmente identificable, teniendo en cuenta la reducción del factor de escala y ángulo con el eje horizontal f(0,1,0)=(d1,d2)=(cos φ,senφ)

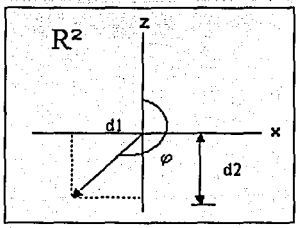

Figura 1.7: Proyección sobre R^2 del eje Y de R^3

Por lo tanto la matriz de conversión es la siguiente:

$$[x \quad y \quad z]*\begin{bmatrix} 1 & 0 \\ \cos\varphi & sen\varphi \\ 0 & 1 \end{bmatrix} = [X2d, Y2d] \ldots (11)$$

1.2 CONCEPTOS DE ANIMACIÓN POR COMPUTADORA.

Los pasos empleados para realizar una animación por computadora básicamente son los mismos que se emplean en la creación de dibujos animados, muñecos de plastilina e incluso las películas, ya que en el fondo el principio es el mismo: desplegar imágenes fijas secuenciadas cuadro por cuadro. Pero es en la generación de estos cuadros donde el método cambia.

La generación de los cuadros es parte importante, no es recomendable generar cuadros que carecen de información para nuestros propósitos, es decir es necesario determinar cuales y que cantidad de cuadros son suficientes para lograr una buena animación, pero no debemos exceder los necesarios. Hacer dicha determinación no es tarea fácil para el animador (o director), es por eso que ha predominado en el mundo de la animación una metodología para la creación de cada uno de los cuadros, basándose en la utilización de cuadros clave(story board). A continuación se explican las etapas más importantes en la realización de secuencias animadas, estas son: realización del guión, "story board", creación de objetos y escenas, generación de cuadros intermedios, edición y postproducción

1.2.1 .-Guión

En general en el mundo del cine, TV, teatro y demás, el guión es una descripción escrita de las escenas a detalle junto con los diálogos para cada personaje, características de los personajes y en algunos guiones se incluye cual es la parte importante de la escena. En el mundo de la animación (dibujada o digital) el concepto de guión no cambia en mucho, Donald Earn [2] lo define como "una descripción de la acción y definición de la secuencia de movimientos como un conjunto de eventos básicos que deben ocurrir...", esta definición es correcta, pero incompleta ya que en las animaciones también existen diálogos y es necesario describir las escenas.

En conclusión un guión lo podemos definir como una narración que describe las escenas y la forma en que los personajes interactúan con estas y con otros personajes (objetos).

1.2.2 .-Story Board

"Story board" o cuadros clave, es el guión contado en imágenes, en donde únicamente las partes más importantes son resaltadas y determinan la forma en que deben ser construidos los escenarios, objetos y trayectorias. Los cuadros clave sitúan a cada objeto en una parte de la escena, conociendo esta posición y su posición en el siguiente cuadro clave podemos determinar la trayectoria a seguir de cada objeto y de esta manera editar su trayectoria, así como sincronizarlos, es por eso que el Story board es muy importante para un animador.

Debe tenerse siempre presente que ninguno los cuadros clave que resulten de esta etapa no pueden ser excluidos de la animación, ya que al hacer esto se perdería continuidad en la misma.

1.2.3 .-Creación de escenas y objetos

La creación de escenas y objetos es una de las etapas principales y de mayor consumo de tiempo de la animación por computadora, ya que es en esta etapa donde se lleva mas de la mitad de tiempo trabajando, esto por los cálculos y edición de los diferentes objetos que contiene una sola escena, en concreto esta etapa consta de los siguientes pasos:

1.2.3.1 .- Modelado de los objetos y escenas

Un modelo es la representación de un fenómeno u objeto de tal forma que sea manejable para su estudio, en graficación por computadora la representación de objetos y superficies se puede hacer en cualquiera de los diferentes métodos que existen, los mas utilizados son por medio de ecuaciones paramétricas y mallas poligonales [1]. Existen diferentes programas que utilizan uno de estos métodos, en algunos la creación de modelos se puede llevar a cabo a través de una interfaz gráfica de usuario y en otras, como en POV-RAY es necesario que el usuario edite las líneas de código que definen a los objetos. Determinar que programa utilizar depende del diseñador y de las características que los modelos deben cubrir.

Una vez hecha la elección del software el siguiente paso es definir los escenarios apoyándose en el Story Board, también se determina el orden de los ejes coordenado (método de la mano derecha o izquierda) , la textura de cada objeto y la posición de cada fuente de luz de la escena para lograr la iluminación deseada.

El modelado de objetos como puede ser una silla, una puerta un árbol, un personaje[4], etc. puede llevarse a cabo al paralelo que el modelado de escenas. Para la realización de esta etapa es necesario establecer estándares en la utilización de texturas y escalas de medición para evitar que los modelos luzcan poco homogéneos.

1.2.3.2 .- Posición de cámaras

La cámara es un punto en el espacio a partir del cual se va a observar una escena, es la posición, orientación y ángulo de una cámara la que determinará la imagen a crear.
La sintaxis para la definición de una cámara dentro de un a escena en Pov-Ray la siguiente:

```
camera {
   location <-2, 3, -3>
   direction <0.0, 0.0, 2.0>
   up  <0.0, 1.0, 0.0>
   right <4/3, 0.0, 0.0>
   look_at <0, 0, 0>
}
```

Se puede observar que para la definición de las propiedades de una cámara se hace uso de un vector < x , y , z >. Para posicionar la cámara es necesario modificar el vector *location <x, y , z>*. La cámara es un objeto más de la escena y es posible agregarle movimiento a la misma cuando se crea la animación.

1.2.3.1 .- Edición de trayectorias de Objetos

Esta etapa es la que más nos interesa, en esta etapa el animador se encarga de editar las trayectorias de objetos, como es el caso de 3D Studio o programar las trayectorias, como es el caso de Pov-Ray.
En caso de contar con una interfaz gráfica y la facilidad de utilizar el ratón para el movimiento de objetos, la edición de trayectorias consume muy poco tiempo, ya que el animador recibe una retroalimentación inmediata de las trayectorias que esta describiendo de manera visual, ya sea por medio de líneas que representen la trayectoria que va a seguir el objeto, previsualizando en modelo de alambre[5] la animación o ambas.

[4] Se maneja a los personajes de una animación como objetos para simplificar los conceptos
[5] El volumen y características de los objetos es representado por medio de líneas.

En el caso que se tenga que editar las trayectorias de manera manual, es decir en papel y luego probarlas directamente en la generación de los cuadros intermedios, sin la capacidad de previsualizarlos, representa una gran perdida de tiempo si es que no se logran los resultados deseados en los primeros cálculos, esto debido a la cantidad de cuadros y el tiempo para la generación de los mismos para evaluar resultados.

1.2.4 Generación de cuadros (render[6])

Son las imágenes intermedias entre los cuadros clave. El número de cuadros intermedios que se necesitan se determina de acuerdo al tiempo que durará la animación y el número de cuadros por segundo de la animación. Una película de cine consta de 24 cuadros por segundo (fps[7]) para asegurar que el ojo humano perciba sin perdida de demasiados cuadros y evitar que la animación se vea pausada. Siguiendo este esquema y suponiendo que deseamos crear una animación de 2 minutos obtenemos que el número de cuadros por segundo que se necesitan para la animación son 120 seg. x 24 fps = 2880 cuadros.

La generación de estos cuadros corre a cargo de una máquina de "render" que es independiente al software de modelado. El método de "render" difiere de una máquina de "render" a otra.

1.2.5 Edición y postproducción

En esta etapa tomamos como material de trabajo todas las imágenes producidas en la etapa anterior, estas imágenes serán analizadas para determinar si se logró modelar lo que queríamos, de no ser así se pueden editar para mejorar los resultados. Otra parte importante de esta etapa es la postproducción que consiste en agregar información a las imágenes obtenidas anteriormente, las principales tareas de esta etapa son:

- Hacer una introducción, con el nombre de la animación y autores.
- Agregar efectos visuales.
- Agregar sonido: música de fondo, sonido de los objetos, efectos especiales, etc.
- Agregar los créditos al final de la animación

[6] Render: generación de imágenes.
[7] Siglas en ingles de Frames per second – cuadros por segundo.

Para la realización de estas tareas se hace uso de software especial totalmente diferente al utilizado anteriormente y como es de esperarse se requiere de experiencia en estas tareas para poder lograr los resultados esperados.

1.3 ANÁLISIS DEL SOFTWARE EXISTENTE

En la actualidad existen una cantidad importante de software para el modelado en 3D, pero son solo unos pocos los que realmente valen la pena analizar para determinar cuales son sus ventajas y desventajas, así como tener un punto de referencia para la evaluación de resultados del presente proyecto. De estos pocos, se analizaran Pov-Ray y 3D Studio Max los cuales son los más representativos de los programas existentes y finalmente se analizarán las características de Material 3D y se explicaran las características deseables en el editor de trayectorias.

1.3.1 Análisis de Pov-Ray

POV-RAY son las siglas de Persistence of Vision™ Ray-Tracer. Es un software Freeware[8] que se caracteriza por estar escrito en ANSI "C". No cuenta con una interfaz gráfica de usuario, en vez de eso proporciona su propio editor de pragramas que se muestra en la Figura 1.8 y como se puede observar para su utilización en todo momento se requiere de la programación directa del usuario, es decir, para representar un objeto en el espacio el usuario debe introducir un segmento de código que describirá dicho objeto.

Su editor es muy simple y fácil de entender, pero es su sencillez lo que le resta funcionalidad, lo cual obliga al usuario emplear más tiempo en el modelado de objetos debido al nivel de abstracción.

[8]Freeware: Software de código abierto con condiciones para su uso.

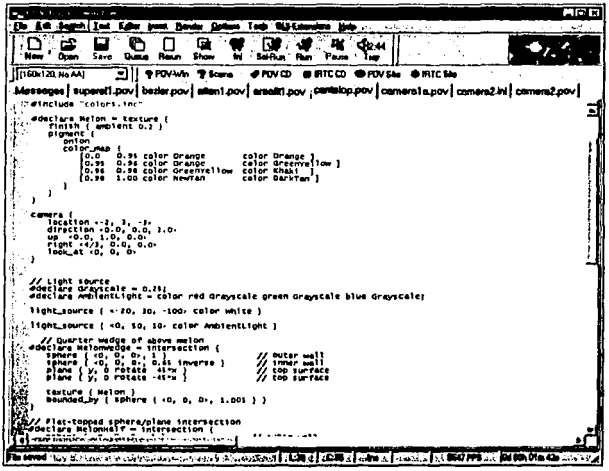

Figura 1.8: Editor de Pov-Ray

A continuación se hace un resumen de las ventajas y desventajas de este software.

Ventajas:

1. Es un programa gratuito, que se puede obtener de Internet.
2. Existen porciones de código de modelos disponibles en sitios de la red.
3. El código puede compilarse en diferentes plataformas, lo cual cumple con nuestro requerimiento de portabilidad.
4. Esta basado en el lenguaje C , que lo hace muy comprensible.
5. Su máquina de "render" es muy buena y logra texturas y pigmentos bastante buenos.
6. El tiempo de generación de imágenes es relativamente rápido.

Desventajas:

1. No cuenta con un interfaz gráfica de usuario que permita uso del ratón para editar los objetos.
2. Los programas pueden llegar a ser demasiado largos y su análisis se hace más difícil.
3. No cuenta con un editor de trayectorias.

4. Para animar es necesario editar un archivo de guión por cada secuencia de animación y los cálculos para la posición, trayectorias y transformaciones de los objetos deben realizarse manualmente, lo cual se traduce en pérdida de tiempo.
5. Las animaciones tienen que hacerse en base a una sola variable, la variable *clock*.

1.3.2 Análisis de 3D Studio Max.

3D Studio Max es un software comercial muy caro el cual sólo funciona en Windows, en general es muy bueno y contiene una serie de herramientas que facilitan mucho la animación por computadoras. Pero lo ideal es buscar un software gratuito y que pueda ser utilizado en diferentes plataformas.

En la figura 1.9 se muestra la interfaz principal de 3D Studio, como se puede observar este software cuenta con una interfaz gráfica de usuario más completa, resultado de varios años de desarrollo. La funcionalidad de este software es amplia, contando con una gran cantidad de botones y secciones en la interfaz. Esta característica es un inconveniente para el usuario, ya que de inmediato es saturado de controles que en un inicio no se sabe cual es su función. Para sacar buen provecho de este software se deben aprender las funciones de todos o la mayoría de ellos, ahora la inversión de tiempo se hará en el aprendizaje del funcionamiento de este software.

Este software es uno de los programas de modelado más popular en el mundo de la animación por computadora.

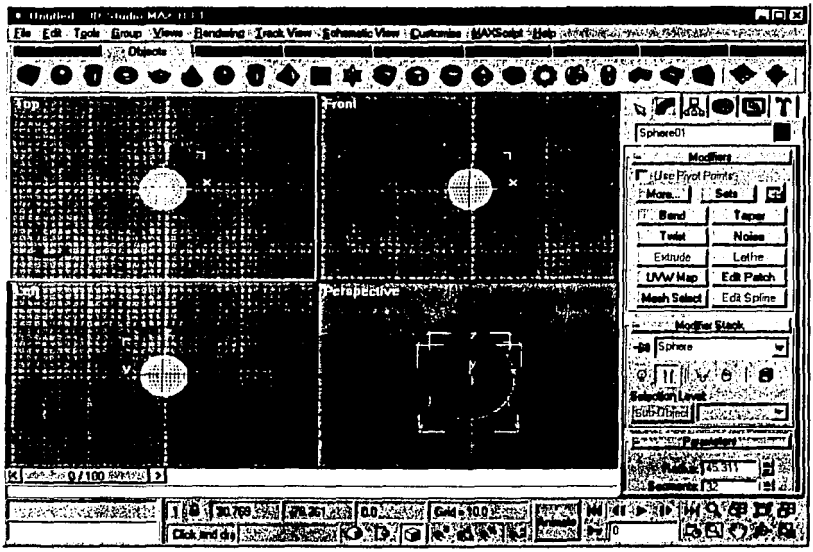

Figura 1.10: Interfaz de 3D Studio

Ventajas

1. Es un software con 14 años de desarrollo y contienen una gran cantidad de herramientas.
2. La interfaz gráfica de usuario es muy completa.
3. Si contiene un editor de trayectorias y diferentes ayudantes para la edición de la animación.
4. Genera las animaciones en muy poco tiempo .
5. Existen muchos libros para el aprender su uso.
6. Acepta la conexión de plug-ins[9].
7. Cuenta con secciones para la visualización de los diferentes planos y la perspectiva del modelo, y permite la manipulación de los objetos con el ratón sobre ellas.
8. Permite previsualizar los modelos en la interfaz.

[9] Tecnología que permite al Software tener la capacidad de aceptar la conexión de pequeños programas para aumentar sus funciones.

Desventajas

1. Es un software muy caro y es necesario pagar una licencia para su uso.
2. Sólo funciona bajo el sistema operativo Windows lo que limita su uso.
3. Su interfaz es muy buena, pero contiene un exceso de botones que saturan a un usuario no experimentado.
4. Se requiere invertir tiempo para aprender a utilizarlo.
5. La calidad de sus texturas no es muy buena.
6. Las animaciones creadas no tienen la calidad que se obtiene de Pov-Ray, ya que se notan los polígonos que forman los objetos.

1.3.2 Análisis de Material 3D.

Material 3D es un proyecto de para desarrollar software de modelado, este proyecto es bastante ambicioso ya que el desarrollo del mismo busca eliminar la gran cantidad de inconvenientes que tienen la mayoría de las aplicaciones para la creación de animaciones en 3D.

Material 3D busca desarrollar un software bajo la Licencia Publica General (GPL[10]) [R4], licencia que distribuye software del proyecto GNU[11]. La Licencia Pública General de GNU pretende garantizarle la libertad de compartir y modificar software libre, las condiciones de uso vienen explicadas a mayor detalle en el documento oficial de esta licencia [R5].

Otro inconveniente que busca resolver Material 3D es la portabilidad, esto se logra con la utilización de bibliotecas gtk+ que están construidas sobre la biblioteca glib como se puede observar en la figura 1.11, esta última define su propio conjunto de tipos de datos, este enfoque permite la portabilidad de las aplicaciones, dado que sólo hace falta cambiar la los tipos de datos con respecto a la plataforma que se quiera utilizar sin necesidad de rescribir la aplicación en gtk+.

Material 3D contará con tecnología de módulos insertables lo cual permitirá insertar módulos para complementar sus funcionalidades ya diseñadas. Dentro de los módulos insertables planeados para funcionar con este programa están el módulo de

[10] GPL: General Public License
[11] GNU: Proyecto para la distribución de software libre, para más detalles consultar las referencias en la bibliografía

metamorfosis tridimensional, el módulo detector de colisiones y el módulo editor de trayectorias (esta tesis).

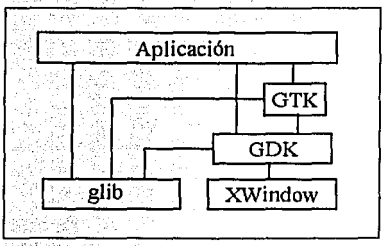

Figura 1.11: Niveles en una aplicación con GTK

En la figura 1.12 se muestra la interfaz gráfica de usuario de material 3D y como se puede observar es simple y bien organizada, lo cual lo hace agradable a la vista. Básicamente consta de 3 pestañas independientes: *model, texture* y *animate*, cada una de ellas con sus propias herramientas, esta organización simple le sugiere de inmediato al usuario los pasos a seguir para desarrollar una animación.

La retroalimentación hacia los eventos del usuario es importante y es por eso que Material 3D previsualiza los modelos en modelo de alambre, así como la perspectiva haciendo uso de bibliotecas Mesa3D [R6] que son un clon de OpenGL [R7].

Otro aspecto importante de este software es que esta orientado a aplicaciones comerciales, didácticas y científicas. Tal vez el punto fuerte de esta última orientación es la capacidad de combinar este tipo de software con aplicaciones científicas. Si en el pasado un grupo investigación deseaba realizar animaciones muy específicas, que requería modificaciones el software de modelado, se tenia que pagar grandes sumas de dinero (aparte de la licencia) para que la empresa propietaria del software satisficiera sus necesidades. Con la política de código abierto que adopta Material 3D permite al público en general modificar o ampliar el código a sus necesidades, lo cual se ve reflejado en ahorro de dinero.

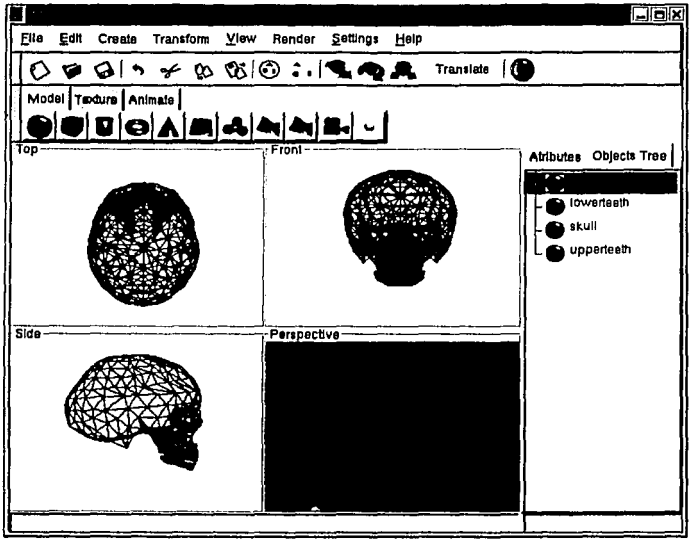

Figura 1.12 : Interfaz de Material 3D

Las características deseables del editor de trayectorias que funcionará como módulo de material 3D son:

- Permitir que el usuario grafique trayectorias por medio del uso del ratón
- Capacidad de leer las coordenadas de las trayectorias de los objetos de un archivo de texto, facilitando que los cálculos científicos o de ingeniería puedan ser visualizados en forma de animación.
- Permitir definir funciones para calcular las coordenadas de las rutas.
- Permitir definir rutas por medio de ecuaciones permitirá realizar animaciones basadas en física, de este modo definir que un objeto (una pelota por ejemplo) va a caer siguiendo un camino definido por la ecuación de caída libre.

CAPÍTULO II .- ANÁLISIS Y DELIMITACIÓN DE LAS TRAYECTORIAS A IMPLEMENTAR

Objetivo:

- Definir y delimitar los tipos de trayectorias que el programa realizará.

Para poder iniciar con el análisis de los elementos de programación necesarios para el desarrollo de la aplicación es necesario antes analizar y delimitar los tipos de trayectorias a implementar. En primer lugar debemos entender que es una trayectoria en un espacio tridimensional para después explicar las trayectorias que nuestra aplicación implementará.

Una trayectoria es una aplicación c:[a,b] → R^n, si n =3 entonces c es una trayectoria en el espacio expresada de forma paramétrica $c(t) = (x(t), y(t), z(t))$ siendo x(t), y(t) y z(t) las componentes de la curva, la variable t la utilizamos para denotar el tiempo, de manera tal que c(t) sirve para denotar la posición en el espacio en un tiempo t.
De esta forma podemos decir que una trayectoria se describir por medio de líneas continuas a trozos [11], cómo se muestra en la figura 2.1.

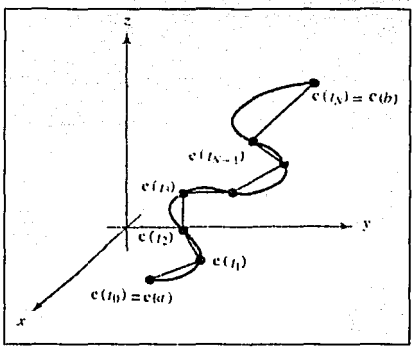

Figura 2.1: Aproximación poligonal de una trayectoria.

Ahora para poder hacer finito el número de puntos intermedios es necesario establecer el número de posiciones (c [tn])que se desean obtener por trayectoria, y de esa forma dividir la trayectoria en un conjunto de puntos equidistantes. Como ejemplo para la obtención de estos puntos se muestra la figura 2.2 que contiene la salida de la generación de cuadros de un programa hecho en Pov-Ray, con las siguientes características.

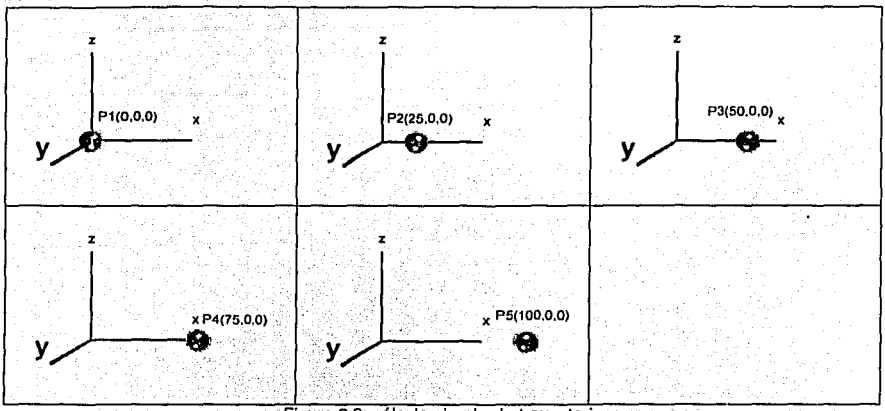

Figura 2.2: cálculo simple de trayectoria

Se trata de la trayectoria simple de una esfera sobre el eje x, el punto inicial es Pi(0,0,0) y el punto final es Pf(100,0,0). El número de puntos que se deseaban obtener de la trayectoria son 5. Con estos datos se calcularon los puntos intermedios de la trayectoria:

P1(0,0,0) – P2(25,0,0) – P3(50,0,0) – P4(75,0,0) – P5(100,0,0)

Como se puede observar el número de segmentos es igual a 4, esto quiere decir, que la longitud de la trayectoria fue dividida entre el numero de puntos intermedios menos uno, dándonos como resultado el tamaño del segmento igual a 25. En este caso la trayectoria era recta sobre un mismo eje, pero no en todos los casos es así, como en los siguientes subcapítulos se explica.

En la presente tesis se pretende permitir 3 formas diferentes de establecer una trayectoria para objetos tridimensionales. La primera de ellas se trata de trayectoria de objetos dirigidos por puntos definidos por el usuario, la segunda permite obtener datos desde otros archivos, estos pueden ser el resultado de software con aplicaciones científicas o cálculos hechos con otros programas, el tercer y último camino es la utilización de formulas matemáticas introducidas por el usuario de forma paramétrica.

2.1 .-TRAYECTORIAS DE OBJETOS DIRIGIDOS POR PUNTOS.

Para la implementación de este tipo de trayectoria se planeó permitir al usuario determinar la trayectoria a seguir haciendo uso del ratón sobre los tres planos en el espacio. Estos tres planos serán representados visualmente por tres áreas de dibujo independientes y la trayectoria será representada por líneas que unen puntos en el espacio.

A diferencia de la trayectoria simple del ejemplo anterior, una trayectoria de objetos dirigidos por puntos está formada por segmentos de recta en el espacio de diferentes longitudes y para determinar los puntos intermedios equidistantes se requiere de un procesamiento más elaborado.

Para explicar como se calculan los puntos intermedios de este tipo de trayectoria utilizaremos el siguiente conjunto de puntos correspondientes a la trayectoria que se muestra en la figura 2.3.

P = { P1(0,0,0),P2(50,50,50),P3(100,50,50),P4(150,50,150) }

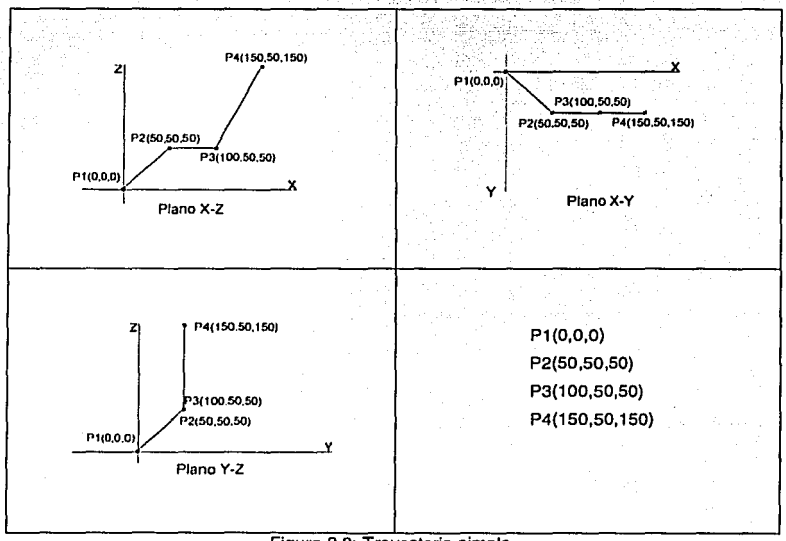

Figura 2.3: Trayectoria simple.

Lo primero que se tiene que hacer es calcular la longitud total de la trayectoria, y para eso se hace uso de la ecuación (1) de la distancia entre dos puntos en el espacio.

$$\text{Longitud} = \sum_{i=1}^{n-1} \sqrt{(X_{Pi+1} - X_{Pi})^2 + (Y_{Pi+1} - Y_{Pi})^2 + (Z_{Pi+1} - Z_{Pi})^2}$$

Aplicándola tenemos que longitud = 86.6 + 50 + 111.8 =248.4, ahora digamos que queremos 8 puntos que formen nuestra trayectoria, eso quiere decir, que el tamaño de cada segmento es de 248.4+(8-1)=35.485. Con este valor, el conjunto **P** de puntos y la ecuación 11 de el capítulo 1 podemos determinar los 8 puntos equidistantes en el espacio que describen esta trayectoria, a este conjunto de puntos lo llamaremos **T**.
Obviamente el valor de T1= P1, para calcular el segundo punto, T2 tenemos:

$$k = 35.485/(86.6 - 35.485) = 0.694$$

$$X_{t2} = \frac{0 + 0.69 * 50}{1 + 0.694}, Y_{t2} = \frac{0 + 0.694 * 50}{1 + 0.694}, Z_{t2} = \frac{0 + 0.694 * 50}{1 + 0.694}$$

$$X_{t2} = 20.49, Y_{t2} = 20.49, Z_{t2} = 20.49$$

Entonces el valor de T2 es T2(20.49, 20.49, 20.49).
Es este punto es importante hacer notar que la imagen (El conjunto P) cambia al introducir el nuevo punto T2 correspondiente a la representación de la trayectoria entre el punto P1 y P2. De esta forma obtenemos un nuevo conjunto de puntos base que a continuación se muestra.

P = { P1(0,0,0),P2(20.49, 20.49, 20.49),P3(50,50,50),P4(100,50,50),P5(150,50,150) }

Para obtener el punto T3 se sigue la misma metodología con los nuevos valores de **P**.
La longitud del segmento P2-P3=51.113, por lo tanto los cálculos son los siguientes:

$$k = 35.485/(51.113 - 35.485) = 2.270$$

$$X_{t2} = \frac{20.49 + 2.27 * 50}{1 + 2.27}, Y_{t2} = \frac{20.49 + 2.27 * 50}{1 + 2.27}, Z_{t2} = \frac{20.49 + 2.27 * 50}{1 + 2.27}$$

$$X_{t2} = 40.97, Y_{t2} = 40.97, Z_{t2} = 40.97$$

C = { P1(0,0,0) , P2(20.49,20.49,20.49),P3(40.97,40.97,40.97)
P4(50,50,50),P5(100,50,50),P6(150,50,150) }

Para calcular T4 los cálculos cambian un poco dado que como se puede observar el segmento T3-T4 es mayor al sobrante del segmento que se dividido anteriormente, es decir, el nuevo segmento P3-P4. Entonces tenemos que solo contamos con 15.63 unidades de este segmento para encontrar T4, para ello tomamos las 19.855 unidades del siguiente segmento de recta, P4-P5, como a continuación podemos ver.

$$k = 19.855/(51.113 - 19.855) = 0.659$$

$$X_{t2} = \frac{50 + 0.659 * 100}{1 + 0.659}, Y_{t2} = \frac{50 + 0.659 * 50}{1 + 0.659}, Z_{t2} = \frac{50 + 0.659 * 50}{1 + 0.659}$$

$$X_{t2} = 69.861, Y_{t2} = 50, Z_{t2} = 50$$

En general son los dos casos con los cuales nos podemos encontrar para determinar un punto **T**, el conjunto **T** final es el siguiente.

T= { T1(0,0,0), T2(20.49, 20.49, 20.49), T3(40.97, 40.97, 40.97),
T4(69.861,50,50),T5(100.238,50,54.762),T6(116.825,50,86.508),
T7(133.412,50,118.254),T8(150,50,150) }

La representación de este conjunto de puntos en los tres diferentes planos se muestra en la figura 2.4, como se puede observar afectivamente los puntos son equidistantes, lo que ayudará a que la trayectoria del objeto luzca más uniforme.

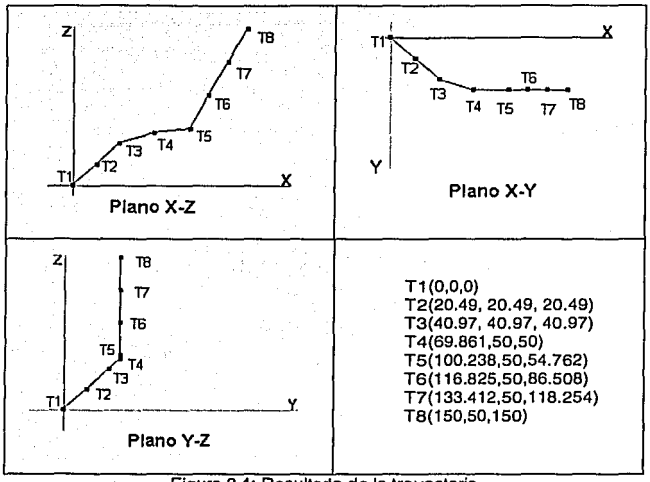

Figura 2.4: Resultado de la trayectoria

Como es de esperarse después de que se llega a este punto tenemos dos conjuntos diferentes de puntos en el espacio, lo ideal es poder guardarlos en un archivo para poder ser recuperados en un futuro. La estructura de este archivo es explicado en el siguiente capítulo.

2.2 .-TRAYECTORIAS OBTENIDAS DESDE UN ARCHIVO.

Como ya lo hemos planteado, resulta de gran utilidad poder obtener trayectorias desde archivos generados por otros programas. El modelado y animación por computadora en aplicaciones científicas son muy utilizadas para poder comprender mejor fenómenos, estructuras, etc. Cuantas veces no hemos visto documentales en donde se hace uso de ello.

Una de las ventajas que ofrece Material 3D y nuestro editor de trayectorias sobre los paquetes de modelado comerciales, es la capacidad de modificar el código fuente a nuestras necesidades. Esta es una gran noticia para todo aquel que necesite de modelar y no cuente con recursos suficientes para pagar el alto precio de las licencias, como es el caso de gran cantidad de grupos investigadores académicos, es decir, modificar el programa para que lea una estructura de archivo en especifico. A continuación se explica.

Como vimos anteriormente, inicialmente se requiere de un conjunto de puntos, para la obtención de los puntos intermedios de la trayectoria, pero puede ser el caso que no se desee hacer este cálculo y utilizar la trayectoria inicial, en tal caso solo es necesario la conexión de los objetos tridimensionales. Otra opción es modificar y agregar puntos con el uso del ratón(como en el método anterior) a los puntos ya obtenidos desde el archivo.

Como ejemplo tenemos el siguiente programa en "C" que corresponde al cálculo de la en caída en barrena de un objeto, como se muestra en la figura 2.5.

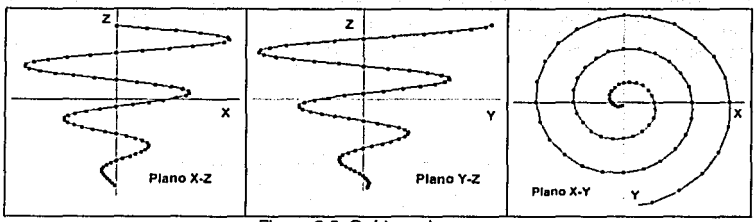

Figura 2.5: Caída en barrena.

```
#include <stdio.h>
#include <math.h>

int main(void)
{
   FILE *stream;
   int x,y,z=100,j,radio=150;

   stream = fopen("cap2.tra", "w+");
   fprintf(stream,"%d \n",72);

   for(j=0;j < 1080; j=j+15)
   {
      radio=radio-2;
      x= sin(j*0.017453)*radio;
      y= cos(j*0.017453)*radio;
      z=z-3;

      fprintf(stream, "%d %d %d\n", x ,y,z);
   }

   fclose(stream);
   return 0;
}
```

Este programa genera un archivo llamado cap2.tra con 72 puntos en el espacio que corresponden a la trayectoria antes mencionada, estos puntos son los mostrados en la siguiente tabla.

Puntos 1-12	Puntos 13-24	Puntos 25-36	Puntos 37-48	Puntos 49-60	Puntos 61-72
0 148 97	0 -123 61	0 99 25	0 -75 -11	0 51 -47	0 -27 -83
37 141 94	-31 -117 58	25 94 22	-19 -71 -14	12 48 -50	-6 -25 -86
71 124 91	-59 -103 55	47 83 19	-35 -62 -17	23 41 -53	-11 -20 -89
100 100 88	-83 -83 52	66 66 16	-49 -49 -20	32 32 -56	-15 -15 -92
121 70 85	-100 -58 49	79 46 13	-58 -34 -23	38 22 -59	-17 -10 -95
133 35 82	-110 -29 46	86 23 10	-63 -17 -26	40 10 -62	-17 -4 -98
135 0 79	-111 0 43	87 0 7	-63 0 -29	39 0 -65	-15 0 -101
129 -34 76	-106 28 40	83 -22 4	-59 16 -32	36 -9 -68	-13 3 -104
114 -65 73	-93 53 37	72 -41 1	-51 29 -35	31 -17 -71	-10 5 -107
91 -91 70	-74 74 34	57 -57 -2	-41 41 -38	24 -24 -74	-7 7 -110
64 -110 67	-52 90 31	40 -69 -5	-28 48 -41	16 -27 -77	-4 6 -113
32 -121 64	-26 98 28	20 -75 -8	-13 52 -44	7 -28 -80	-1 5 -116

Tabla 1: Puntos generados por el programa

La idea es que nuestro editor de trayectorias permita abrir este archivo, recupere estos datos hacia memoria y los grafique en los diferentes planos similar en la figura 2.4 y además que grafique en perspectiva tridimensional.

A partir de este punto es el usuario quien decide si es necesario modificar la trayectoria o directamente se generan los cuadros para la animación.

2.3 .-ESPECIFICACIÓN DEL MOVIMIENTO POR FORMULA MATEMÁTICA

Un inconveniente de la trayectoria de objetos dirigidos por puntos es el tedio de tener que ingresar punto por punto, este tipo de definición de trayectorias en útil cuando se trata de una trayectoria simple y un conjunto de puntos con cardinalidad pequeña. Como es de esperarse no siempre se van a manejar un conjunto pequeño de puntos, en tal caso es necesario de obtener un poco de ayuda de las matemáticas, y para ello permite al usuario generar puntos especificando fórmulas matemáticas como a continuación se explica.

Para permitir esto se tiene que introducir funciones para la generación de los puntos, la figura 2.6 muestra el formulario que permitirá la introducción de las mismas para ser evaluadas en los diferentes planos.

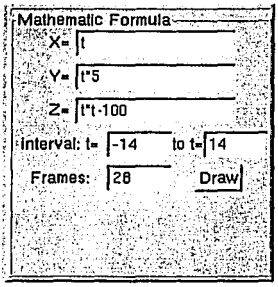

Figura 2.6: Formulario para el editor de trayectorias

Como se puede observar en cada uno de los primeros cuadros de entrada existe una expresión en función de **t**, que describen la trayectoria a crear. Los siguientes cuadros de entrada sirven para determinar el intervalo en que estas funciones se evaluarán y por último el número de cuadros a crear.

Utilizaremos estos mismos datos para ejemplificar el funcionamiento de la especificación de trayectorias por formula matemática:

Intervalo t=-14 a t=14

X(t)=t

Y(t)=t*5

Z=t * t -100

y el número de cuadros=29 con lo cual podemos deducir en donde se va a evaluar la función.

Para t=-14 tenemos que:

X(-14)= -14; Y(-14)=-70; Z(-14)=-96

Como el número de cuadros es exactamente igual al tamaño del intervalo el incremento es igual a 1, por lo tanto el siguiente valor en t es t=-13.

Para t=-13 tenemos que:

X(-13)= -13; Y(-13)=-65; Z(-13)=-69

Y así sucesivamente lo cual nos da como resultado el conjunto de puntos que se muestran en la tabla 2, el destino de estos es una estructura de datos residente en memoria (Point3D) que en el siguiente capítulo se explica.

Puntos 1 – 8	Puntos 9 - 16	Puntos 17 -24	Puntos 25 - 29
-14 -70 96	-6 -30 -64	2 10 -96	10 50 0
-13 -65 69	-5 -25 -75	3 15 -91	11 55 21
-12 -60 44	-4 -20 -84	4 20 -84	12 60 44
-11 -55 21	-3 -15 -91	5 25 -75	13 65 69
-10 -50 0	-2 -10 -96	6 30 -64	14 70 96
-9 -45 -19	-1 -5 -99	7 35 -51	
-8 -40 -36	0 0 -100	8 40 -36	
-7 -35 -51	1 5 -99	9 45 -19	

Tabla 2: Puntos resultantes

Los cuales ya graficados en los tres diferentes planos lucen como en la figura 2.7 se muestra.

Como podemos ver es una gran ventaja este tipo de generación de trayectorias, podemos partir de estos resultados para modificar los puntos con nuestro primer método (con el uso del ratón) y no crearlos manualmente desde el principio.

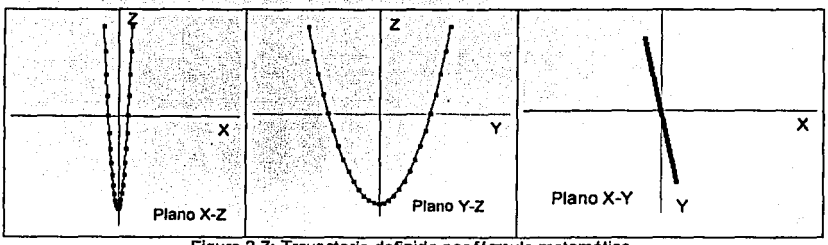

Figura 2.7: Trayectoria definida por fórmula matemática.

En el capítulo 4 se utilizará este mismo ejemplo para evaluar el funcionamiento de nuestro programa.

CAPÍTULO III .- ESTRUCTURAS DE DATOS PARA LA REPRESENTACIÓN DE TRAYECTORIAS Y PROGRAMACIÓN EN GTK+.

Objetivos:

- Determinar las estructuras de datos óptimas para la representación de trayectorias, así como la programación de estas en GTK+.
- Conocer las herramientas GUI que ofrece GTK+.

En este capítulo aprenderemos los programación con gtk+ y las estructuras de datos a utilizar, en primer lugar definimos los tipos de datos y la estructuras de los archivos que nuestro editor de trayectorias manejará, en la segunda consta de una pequeña introducción a la programación a la programación de aplicaciones bajo el sistema operativo Linux con bibliotecas gtk+, explicando los elementos de estas mismas que utilizaremos y en la tercera parte entramos de lleno a la programación de nuestra aplicación.

3.1 .- ESTRUCTURAS DE DATOS A UTILIZAR

Para el almacenamiento temporal en memoria de los puntos en el espacio definimos un tipo de dato llamado *type3D*, que consta de una estructura formada por tres variables del tipo gint[12] como a continuación se muestra.

```
typedef struct
{
gint x;
gint y;
gint z;
}type3D;
```

A partir de este tipo es posible declarar las variables a utilizar por nuestro programa, el siguiente segmento de código declara la variable *point3D* que servirá para almacenar la trayectoria inicial.

```
type3D point3D[];
```

Para el almacenamiento temporal de los puntos intermedios calculados se declara una variable llamada *trj*.

```
type3D trj[];
```

Esta ultima variable se utiliza para almacenar la trayectoria calculada como se vio en el capitulo 2.1, para posteriormente ser almacenadas en un archivo o asignarlos a los objetos para posteriormente generar las imágenes.
Para el almacenamiento en disco de las trayectorias utilizamos la siguiente estructura de archivo.

[12] Tipo entero equivalente en glib, para más detalles consultar el API de glib.[R3]

```
Primera parte:    Número de puntos
Segunda parte:    Puntos
Tercera parte:    Número de cuadros
Cuarta parte:     Puntos trayectoria
```

La extensión escogida para los nombres de los archivos es "tra", es decir que si tenemos un archivo que contenga los puntos calculados con nuestro programa, correspondientes a una caída en barrena (por ejemplo) se podría llamar *barrena.tra*.

3.2 .- BREVE INTRODUCCIÓN A GTK +

El origen de gtk+ viene íntimamente relacionado con el desarrollo de GIMP[13]. El GIMP es un programa de manipulación de gráficos planeado para ser ejecutado bajo sistema operativo Linux, pero en Linux solo existían bibliotecas de muy bajo nivel para el desarrollo de interfaces gráficas. Dado que este programa requería de una interfaz de usuario y el manejo de eventos (principalmente del ratón) el grupo de programación optó por desarrollar sus propias bibliotecas y las llamó GTK, GIMP Tool Kit (Conjunto de herramientas GIMP).

Estas bibliotecas están escritas en lenguaje C bajo el paradigma orientado a objetos, esto es usando la definición de clases y llamadas a métodos. La clase principal es GTKObject que contiene las características mas generales de los objetos gtk. Esta clase es heredada a GTKWidget que define todos los elementos necesarios para la construcción de interfaces gráficas de usuario como son ventanas, botones, menús, cuadros de dialogo, etcétera, llamados Widgets. De esta manera solo es necesario incluir la librería *gtk.h* y a partir de ella construir los diferentes widgets que se necesiten para la interfaz a construir.

Los pasos generales para la construcción de un objeto tipo widget son los siguientes:

1. Declarar una variable de tipo GtkWidget
2. Construir el tipo de widget a utilizar
3. Conectar todas sus señales con sus respectivos manejadores
4. Inicializar los atributos de los widgets
5. Empacar el widget en un contenedor , como lo es una ventana o un botón.
6. Mostrar el widget.

Para ejemplificar la construcción de aplicaciones en gtk+ tenemos el siguiente código de un programa que despliega una ventana con un botón, que al ser presionado imprime "Hola Mundo" en la terminal y cerrará la aplicación. Además cuando se cierra la ventana imprimirá un aviso en la terminal: " Evento: delete , ha ocurrido".

[13] Graphical Image Manipulation Program (programa para manipulación gráfica de imagenes)

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

/* Esta es una función de retrollamada y solo imprime Hola mundo en
la terminal */
void hello( GtkWidget *widget,
            gpointer   data )
{
    g_print ("Hola Mundo\n");
}

gint delete_event( GtkWidget *widget,
                   GdkEvent  *event,
                   gpointer   data )
{
    g_print ("Evento: delete , ha ocurrido \n");
    return(TRUE);
}

/* Retrollamada para cerrar la aplicación */
void destroy( GtkWidget *widget,
              gpointer   data )
{
    gtk_main_quit();
}

int main( int   argc, char *argv[] )
{
    /* Se crean las variable tipo  widget */
    GtkWidget *window;
    GtkWidget *button;

    /* Esta línea es llamada en todas las aplicaciones GTK+.*/
    gtk_init(&argc, &argv);

    /* se crea una ventana de alto nivel(principal) */
    window = gtk_window_new (GTK_WINDOW_TOPLEVEL);

    /* Se conecta la ventana con las señales a responder
      y la función que la manejará */
    gtk_signal_connect (GTK_OBJECT (window), "delete_event",
                        GTK_SIGNAL_FUNC (delete_event), NULL);

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

    /* Se establece el tamaño del borde de la ventana. */
    gtk_container_set_border_width (GTK_CONTAINER (window), 10);

    /* Se crea un nuevo botón con la etiqueta Hola Mundo. */
    button = gtk_button_new_with_label ("Hola Mundo");

    /* cuando el botón recibe una señal de clicked, este manda a
llamar
       a  la  funcion hello()  */
    gtk_signal_connect (GTK_OBJECT (button), "clicked",
```

```
                        GTK_SIGNAL_FUNC (hello), NULL);

    /* Esto manda a llamar a la función gtk_widget_destroy, que
elimina
     * el widget indicado */
    gtk_signal_connect_object (GTK_OBJECT (button), "clicked",
                               GTK_SIGNAL_FUNC
(gtk_widget_destroy),
                               GTK_OBJECT (window));

    /* esto empaqueta el botón en la ventana (un contenedor  gtk)
*/
    gtk_container_add (GTK_CONTAINER (window), button);

    /* el paso final es desplegar el widget creado */
    gtk_widget_show (button);

    /* y la ventana */
    gtk_widget_show (window);

    /* Todo programa en gtk debe tener la función gtk_main().
       Controla el fin de ejecución y espera por eventos a
       ocurrir (como presionar una tecla  o eventos del ratón). */

    gtk_main ();

    return(0);
}
/* Fin del ejemplo */
```

Como se puede observar en el ejemplo, la funcionalidad de todos los widgets que responden a sus respectivas señales residen en las funciones de retrollamada (callbacks), esto es una ventaja debido a que permite separarlos de la parte de creación y despliegue de los mismos.

Por lo tanto la mayor parte de la programación para el desarrollo de esta tesis se encuentra en estas funciones y es por eso que es necesario conocer los Widgets que vamos a utilizar antes de programar la aplicación en sí.

A continuación se explicaran los widgets: *window, table, drawing_area*, así como la escucha de eventos del ratón en el drawing_area, como parte importante de nuestro tema de tesis. El motivo por el cual no se explican todos los Widgets utilizados en esta tesis es debido a que Widgets como botones, etiquetas y demás son muy sencillos y fáciles de implementar. Si se desea saber más a detalle como programar este tipo de Widgets se recomienda consultar el API[14] de GTK+.

[14] siglas en ingles de Applications Programming Interface (Interfaz para programación de aplicaciones) [R3]

3.2.1 .- Widget window de gtk +

Este tipo de widget es un contenedor de widgets y sirve para ir agregando elementos como botones, recuadros (frames) y tablas para construir una aplicación. Para ello casi todas las aplicaciones crean una ventana principal que usualmente se denomina ventana de nivel superior, las ventanas de nivel superior no tienen ventana padre, por que no están contenidas en ninguna otra ventana. En GTK+ los Widgets tienen una relación padre-hijo en la que el padre es el widget contenedor y el hijo es el widget que es contenido en dicho contenedor. Dicho lo anterior podemos explicar como se crea una ventana de nivel superior y cómo se conecta a sus señales, para posteriormente explicar los Widgets que serán hijos en nuestra aplicación.

Lo primero que se debe de hacer y es un paso general para todos los Widgets debido a que lo que se hace es definir un puntero a una estructura Gtkwidget.

```
GtkWidget *ventana;
```

Después se invoca al constructor y le pasamos como parámetro el tipo de ventanas que queremos construir, en este caso GTK_WINDOW_TOPLEVEL.

```
ventana = gtk_widget_new(GTK_WINDOW_TOPLEVEL.);
```

Después podemos definir las propiedades de la ventana.

```
/* Tamaño de la ventana */
  gtk_widget_set_usize (ventana, 800, 600);

/* Tamaño del borde del contenedor */
  gtk_container_set_border_width (GTK_CONTAINER (ventana), 3);

/* se establece el nombre */
  gtk_window_set_title (GTK_WINDOW (ventana), "Nombre de la
ventana" );
```

Hasta ahora solo hemos definido apariencia, hace falta permitir a la aplicación responder a las acciones del usuario, como es hacer click en el icono de cerrar. A estas acciones GTK+ les llama eventos(GdkEvent), y están definidos en un tipo de

datos enumerado llamado GdkEventType y en el se encuentra un cierto número de eventos a los que deferentes Widgets pueden responder. En el caso de la ventana principal los eventos a los que puede responder son a *focus_in_event* que detecta cuando el cursor entra en la ventana principal, *focus_out_event* cuando sale de la ventana principal, *destroy_event* cuando el usuario cierra la ventana y otros mas. Para la ventana principal de nuestra aplicación sólo conectaremos el evento *delete_event* que ocurre cuando el usuario cierra la ventana, la función de retrollamada que atenderá a este evento se encargara de cerrar la aplicación y hacer la labor de recuperación de memoria si es el caso. El código para conectar el evento con la función es el siguiente.

```
gtk_signal_connect (GTK_OBJECT (ventana), "delete_event",
                    GTK_SIGNAL_FUNC (fnCerrar), NULL);
```

Como ya lo habíamos visto en el ejemplo, existe un bucle que se encarga de escuchar los eventos, que es el *gtk_main()*. El fragmento de código anterior le dice a *gtk_main()* que cuando el usuario cierre la ventana se debe ejecutar la función "*fnCerrar*". Esta función debe ser programada por nosotros y esto nos permite realizar las acciones que nos interese hacer, a continuación se muestra el código para una posible forma de programar esta función de retrollamada.


```
/*
 * fnCerrar
 *
 */
gint fnCerrar (GtkWidget *widget, gpointer *data)
{

/* Se puede hacer recolección de memoria o limpieza antes de cerrar
la aplicación */

/* Con esta función se cierra el programa gtk_main(). */
    gtk_main_quit ();

    return (FALSE);
}
```

Finalmente hacemos visible la ventana con la función gtk_widget_show (ventana), todos los Widgets hijos de la ventana que sean visibles se mostraran entonces.

Con esto ya aprendimos como crear la ventana principal, ahora necesitamos saber como agregar Widgets a la ventana y en que posición, para ello gtk+ tiene definido Widgets de empaquetado que únicamente sirven para administrar el diseño de la aplicación, como a continuación se explica.

3.2.2.- widget table de gtk +

Existen tres Widgets de empaquetado: la caja horizontal(*gtk_hbox_new*), la caja vertical (*gtk_vbox_new*) y las tablas(*gtk_table_new*) las cajas sirven para ir agregando Widgets a la misma y utilizan todo el espacio disponible horizontalmente o verticalmente según sea el caso.

Las tablas de empaquetado dividen un contenedor en columnas y renglones lo que permite agregar un widget hijo en la posición deseada y ocupando el espacio deseado, esto nos da más control sobre el diseño de nuestras aplicaciones lo cual es importante.

La creación de este tipo de Widgets es básicamente el mismo que el de los anteriores, solo es necesario explicar como se agrega un widget a este tipo de contenedores. Para ello se hace uso de la función *gtk_table_attach*.

Para ejemplificar de manera sencilla como se utilizan las tablas de empaquetado tenemos el siguiente código y la ejecución del mismo (figura 3.1).

```
GtkWidget *main_win;
GtkWidget *tabla;
GtkWidget *imagen2;
GtkWidget *imagen1;

main_win = gtk_window_new (GTK_WINDOW_TOPLEVEL);
gtk_object_set_data (GTK_OBJECT (main_win), "main_win", main_win);
gtk_widget_set_usize (main_win, 400, 300);
gtk_window_set_title (GTK_WINDOW (main_win), _("Ejemplo: Table"));

tabla = gtk_table_new (4, 4, FALSE);  //se define una cuadricula de 4x4


    //  se agrega la tabla a la ventana padre
gtk_container_add (GTK_CONTAINER (main_win), tabla);

    // se cargan las imagines imagenes
imagen1 = create_pixmap (main_win, "linux1.xpm", FALSE);
imagen2 = create_pixmap (main_win, "linux2.xpm", FALSE);
```

```
    /* En  esta parte se agregan las imagines de acuerdo a la
cuadricula */

  gtk_table_attach (GTK_TABLE (tabla), imagen1, 0, 2, 0, 2,
                    (GtkAttachOptions) (GTK_EXPAND | GTK_FILL),
                    (GtkAttachOptions) (GTK_EXPAND | GTK_FILL), 0, 0);

  gtk_table_attach (GTK_TABLE (tabla), imagen2, 2, 4, 2, 4,
                    (GtkAttachOptions) (GTK_EXPAND | GTK_FILL),
                    (GtkAttachOptions) (GTK_EXPAND | GTK_FILL), 0, 0);


  gtk_widget_show (imagen1);
  gtk_widget_show (imagen2);
  gtk_widget_show (tabla);  // se hacen visibles los Widgets
  gtk_widget_show (main_win);
```

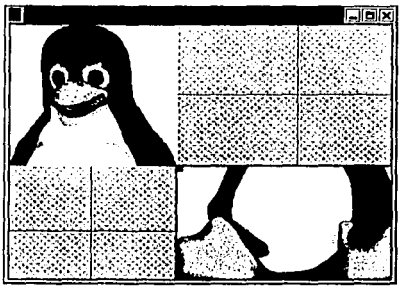

Figura 3.1.- Ejemplo de tablas

Para agregar un widget a una tabla se utiliza la función *gtk_table_attach()* en la cual el primer parámetro se le pasa el nombre de la tabla, en el segundo el nombre del widget que se desea agregar, en los cuatro siguientes parámetros se especifican las cuatro coordenadas a ocupar por el widget hijo, los dos siguiente parámetros son opciones como es la expansión o rellenado, los dos últimos widgets indican cuantos píxeles de ajuste hay que utilizar alrededor del widget.

3.2.3 .- widget drawing_area de gtk+

Este widget es muy importante para nuestra aplicación y en la explicación de su funcionamiento entraremos a más detalle. Este tipo de widget resulta ser muy sencillo dado que tiene muy pocas propiedades a comparación con otros widgets. El único propósito de este widget es proporcionar una área para emplear funciones GDK[15] para dibujar dentro del widget.

Antes de explicar como utilizar estas primitivas graficas GDK es necesario aprender como se crea una área de dibujo y los eventos importantes para ésta.

Con las siguientes líneas se crea una área de dibujo y se establece su tamaño.

```
GtkWidget *area_dibujo;
...
area_dibujo=gtk_drawing_area_new();
gtk_drawing_area_size(area_dibujo,100,100);
```

Con esto ya tenemos creada nuestra área en la cual dibujaremos, pero existen dos eventos importantes para este widget, el evento de exposición, *expose_event* y el evento de configuración, *configure_event*. El suceso *expose_event* ocurre cuando una área de dibujo requiere ser re-dibujada, esto sucede cuando se expone la ventana a un primer plano, o cuando se cambia de tamaño la ventana. Para atender este tipo de sucesos tenemos que programar nuestra función de retrollamada y conectarla con el widget *drawing_area*. La función de retrollamada sólo se encargará de limpiar y volver a dibujar el área de dibujo. El suceso *configure_event* ocurre por primera vez cuando el widget *drawing_area* es creado y mostrado, antes de hacerlo visible es necesario configurarlo, es decir establecer su tamaño y color de fondo. Este evento también ocurre cuando el tamaño de la ventana de nivel superior cambia y por lo tanto el tamaño de el área de dibujo también lo hace, para mantener la proporción es necesario liberar el espacio de memoria que ocupaba el área de dibujo y reservar nueva memoria al tamaño de la nueva área.

Otros eventos ligados a las áreas de dibujo que son importantes de estudiar son los eventos del ratón, no serviría de mucho una área de dibujo si no se puede

[15] Graphics Drawing Kit, para mas detalles consultar la bibliografía

dibujar en ella. A continuación se explica como se manejan los eventos de ratón específicamente con las áreas de dibujo.

3.2.4 .- eventos del ratón en gtk+

Hay tres variantes principales de este tipo de eventos, el *button_press_event* cuando se presiona cualquier botón del ratón, el *motion_notify_event* sucede cuando se mueve el ratón y el *button_release_event* sucede cuando se suelta el botón del ratón.

Para censar estos eventos debemos conectarlos con sus respectivas funciones de retrollamada, como ya se ha hecho en widgets anteriores, a continuación se muestra como se conectan los eventos con el widget "*drawing_area*".

```
  gtk_signal_connect (GTK_OBJECT (drawing_area),
"button_press_event",
                      GTK_SIGNAL_FUNC
(fnHandle_button_press_event),
                      NULL);
  gtk_signal_connect (GTK_OBJECT (drawing_area),
"button_release_event",
                      GTK_SIGNAL_FUNC
(fnHandle_button_release_event),
                      NULL);
  gtk_signal_connect (GTK_OBJECT (draw_area),
"motion_notify_event",
                      GTK_SIGNAL_FUNC
(fnHandle_motion_notify_event),
                      NULL);
```

Además es necesario delimitar que eventos deben ser escuchados por el widget, esto se hace con el siguiente código.

```
gtk_widget_set_events (drawing_area, GDK_EXPOSURE_MASK |
GDK_BUTTON_PRESS_MASK | GDK_BUTTON_RELEASE_MASK);
```

Cuando cualquier tipo de evento GDK sucede (no solo los del ratón), los datos asociados a dicho evento son guardados en una estructura para poder ser manipulados posteriormente por la función de retrollamada. Como ejemplo tenemos la estructura datos que se utiliza cuando un botón del ratón es presionado o liberado.

```
struct _GdkEventButton
{
  GdkEventType type;
  GdkWindow *window;
  gint8 send_event;
  guint32 time;
  gdouble x;
  gdouble y;
  gdouble pressure;
  gdouble xtilt;
  gdouble ytilt;
  guint state;
  guint button;
  GdkInputSource source;
  guint32 deviceid;
  gdouble x_root, y_root;
};
```

Una vez que se presiona cualquiera de los 3 botones del ratón sobre cualquiera el widget "*drawing_area*" se van a almacenar los datos en esta estructura y *gtk_main* mandara a llamar a la función *fnHandle_button_press_event*. A esta función se le pasa como parámetro un puntero a dicha estructura para que pueda manejar los datos. A continuación se muestra el código para esta función en la que se detecta cual de los tres botones fue presionado y las coordenadas del área de dibujo en donde se hizo.

```
fnHandle_button_press_event  (GtkWidget       *widget,
                              GdkEventButton  *event,
                              gpointer         user_data)
{

if (event->button == 1)
   {
      g_print("Presionaste el botón izquierdo \n");
      g_print("Las coordenadas son x=%d, y=%d \n", event->x, event-
>x);
   }
if (event->button == 2)
   {
      g_print("Presionaste el botón central \n");
      g_print("Las coordenadas son x=%d, y=%d \n" , event->x,
event->x);
   }
if (event->button == 2)
   {
      g_print("Presionaste el botón derecho \n");
      g_print("Las coordenadas son x=%d, y=%d \n" , event->x,
event->x);
      }

}
```

El puntero a la estructura *GdkEventButton* llamado "*event*" es utilizado para acceder a cualquier de sus campos, cada que el evento vuelve a ocurrir la estructura cambia sus valores. Para el evento de liberación del botón la estructura es la misma y para el movimiento del ratón difiere un poco pero la idea es la misma.

Ya tenemos las herramientas suficientes para desarrollar nuestra aplicación, en las siguientes líneas entramos a la programación del editor de trayectorias de objetos tridimensionales.

3.3 .- PROGRAMACIÓN DE LA APLICACIÓN.

3.3.1 .- Ventana principal

La ventana principal se trata de una ventana de nivel superior y su aspecto se muestra en la figura 3.2. Como se puede observar la interfaz cuenta con una barra de herramientas, cuatro áreas de dibujo contenidas en 4 marcos con etiqueta y dos carpetas con pestañas, todos ellos son administrados en posición y tamaño por una tabla de 16 renglones por 8 columnas. A continuación se explica a funcionalidad de cada uno de los widgets.

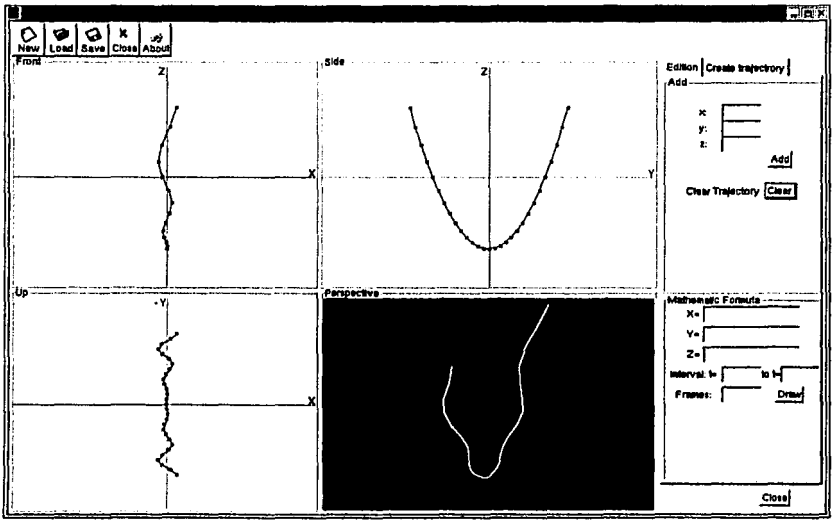

Figura 3.2: Ventana principal

La barra de herramientas cuenta con los siguientes botones:

Herramienta	Descripción
	Crear un nuevo archivo de trayectoria
	Carga una trayectoria desde una archivo generado por otro programa.
	Guarda en archivo la trayectoria que esta siendo editada.
	Cierra el archivo actual.
	Muestra los créditos.

Todos ellos responden al evento "*clicked*" con su respectiva función de retrollamada, las funciones como el save y load hacen uso de otro widget llamado *file_selection* el cual muestra un cuadro de dialogo para navegar en las carpetas y elegir un archivo a cargar o darle el nombre a un archivo nuevo.

Las áreas de dibujo sirven para visualizar la trayectoria en los tres diferentes planos y en perspectiva, además permite al usuario crear una trayectoria directamente sobre ellas haciendo uso del ratón, con el botón izquierdo el usuario puede dibujar puntos sobre cualquier área de dibujo y con el botón central es posible agarrar un punto y modificar su posición .

La última sección, el fólder con dos pestañas, nos permiten editar una trayectoria y establecer los parámetros para la generación de los puntos intermedios.

3.3.2 .- Áreas de dibujo y señales de ratón

Como ya habíamos dicho cada área de dibujo muestra las trayectorias en diferentes planos, la perspectiva frontal por ejemplo, muestra la trayectoria en el plano x-z haciendo a $\dot{y} = 0$, y así lo hacen respectivamente en los otros dos planos.

Dado lo anterior analizaremos el código de una área de dibujo y de esa forma cubriremos a las tres perspectivas. El siguiente fragmento de código corresponde a la creación y conexión de eventos del área de dibujo de la perspectiva frontal.

```
draw_area_Front = gtk_drawing_area_new ();

...

gtk_widget_show (draw_area_Front);
gtk_container_add (GTK_CONTAINER (fr_front), draw_area_Front);

...

gtk_widget_set_events (draw_area_Front, GDK_EXPOSURE_MASK |
GDK_POINTER_MOTION_MASK | GDK_POINTER_MOTION_HINT_MASK |
GDK_BUTTON2_MOTION_MASK | GDK_BUTTON_PRESS_MASK |
GDK_BUTTON_RELEASE_MASK);

...

gtk_signal_connect (GTK_OBJECT (draw_area_Front), "configure_event",
                    GTK_SIGNAL_FUNC
(on_draw_area_Front_configure_event),
                    NULL);
gtk_signal_connect (GTK_OBJECT (draw_area_Front), "expose_event",
                    GTK_SIGNAL_FUNC (on_draw_area_Front_expose_event),
                    NULL);
gtk_signal_connect (GTK_OBJECT (draw_area_Front), "button_press_event",
                    GTK_SIGNAL_FUNC
(on_draw_area_Front_button_press_event),
                    NULL);
gtk_signal_connect (GTK_OBJECT (draw_area_Front),
"motion_notify_event",

GTK_SIGNAL_FUNC(on_draw_area_Front_motion_notify_event),
                    NULL);
gtk_signal_connect (GTK_OBJECT (draw_area_Front),
"button_release_event",
                GTK_SIGNAL_FUNC
     (on_draw_area_Front_button_release_event),
                    NULL);
```

Esta parte ya a sido estudiada por nosotros, ahora resulta ser más importante analizar el código de las 5 funciones que han sido conectadas, estas serán analizadas en orden de aparición.

Para el despliegue de las áreas de dibujo hacemos uso de una técnica utilizada par eliminar el parpadeo en animaciones y juegos por computadora llamada doble buffer. Esta técnica consiste en reservar un espacio de memoria con las

dimensiones del área de dibujo y de esta forma utilizarlo como un lienzo, de esta forma evitamos el parpadeo de la pantalla al no dibujar directamente en el dispositivo de video, si no copiar el buffer hacia él cuando se haya terminado de dibujar. Los buffer son creados con la siguiente linea de código.

static GdkPixmap *pixmap_front = NULL;

Su inicialización corre por cuenta de la función configure_event como a continuación se muestra.

```
Gboolean
on_draw_area_Front_configure_event      (GtkWidget       *widget,
                                GdkEventConfigure *event,
                                gpointer          user_data)
{
      if (pixmap_front)                    //Si ya esta inicializada
         gdk_pixmap_unref(pixmap_front); // Liberar memoria

            /*Inicializarla con el tamaño adecuado*/
       pixmap_front = gdk_pixmap_new(widget->window,

                              widget->allocation.width,
                              widget->allocation.height,
                              -1);
                 /*Establecer el color del lienzo*/
       gdk_draw_rectangle (pixmap_front,
                           widget->style->white_gc,
                           TRUE,
                           0, 0,
                           widget->allocation.width,
                           widget->allocation.height);
  return TRUE;
}
```

Para el evento expose_event sólo se tiene que copiar el buffer a la área de dibujo, como a continuación se muestra.

```
gboolean
on_draw_area_Front_expose_event         (GtkWidget       *widget,
                                         GdkEventExpose  *event,
                                         gpointer         user_data)
{
      gdk_draw_pixmap(widget->window,
                      widget->style->fg_gc[GTK_WIDGET_STATE
(widget)],
                      pixmap_front,
```

```
                                event->area.x, event->area.y,
                                event->area.x, event->area.y,
                                event->area.width, event->area.height);


    return FALSE;
}
```

Con el puntero a la estructura *GdkEventExpose* se puede determinar la posición y tamaño del área de dibujo.

La siguiente función maneja el evento de pulsación de un botón, básicamente lo que hace es determinar que botón se presionó, si se presionó el botón 1 se introduce las nuevas coordenadas a la estructura de datos, se incrementa el contador y se manda a llamar a la función *Repaint*. Esta última función se encarga de limpiar y volver a pintar los puntos y líneas de la trayectoria en las cuatro áreas de dibujo, la explicación de esta se dejó para después.

```
gboolean
on_draw_area_Front_button_press_event  (GtkWidget        *widget,
                                        GdkEventButton   *event,
                                        gpointer          user_data)
{
      int temp=0;
      if (event->button == 1 && pixmap_front != NULL)
         {
       /*Introducimos un nuevo punto en la vista FRONTAL con Y =0*/
            point3D[cont].x=event->x-(widget->allocation.width/2);
            point3D[cont].z=-(event->y-(widget-
>allocation.height/2));
            point3D[cont].y=0;

            cont++;
            Repaint(widget); //se invoca a la función Repaint
         }

      if (event->button == 2 && pixmap_front != NULL)
         {

            clic.x=event->x - (widget->allocation.width/2);
            clic.z=-(event->y - (widget->allocation.height/2));
            for(temp;temp<cont;temp++)
            {
              if(clic.x < point3D[temp].x+4 && clic.x >
point3D[temp].x-4 && clic.z < point3D[temp].z+4 && clic.z >
point3D[temp].z-4)

                  {
                      sel=temp;
                    bandera=TRUE;
                  }
            }
```

```
        }
        return TRUE;
    }
```

Si el botón presionado es el número 2, es necesario capturar primero las coordenadas del evento y guardarlas en una variable del tipo *type3D* llamada clic, definida al principio de este capítulo. El siguiente paso es recorrer la estructura de datos *point3D* para determinar si alguno de los puntos coincide con el evento, de ser así se asigna su índice a la variable "*sel*" y la bandera se hace igual a verdadero. Esta bandera sirve para saber si el botón central sigue presionado, ya que va a ser usado en la función *on_draw_area_Front_motion_notify_event* que a continuación se explica.

La parte importante de esta función es la sentencia *if*, que en general lo que hace es verificar que el botón presionado sea el número 2(central), que el buffer este inicializado y que no se haya liberado el ratón. De ser así las coordenadas del evento sustituyen a las coordenadas indexadas por la variable "*sel*" e inmediatamente se manda llamar a la función *Repaint*, que permite visualizar el movimiento del punto conforme se va modificando.

```
gboolean
on_draw_area_Front_motion_notify_event (GtkWidget       *widget,
                                        GdkEventMotion  *event,
                                        gpointer         user_data)
{
      int x, y;
       GdkModifierType state;

...

 if (state & GDK_BUTTON2_MASK && pixmap_front != NULL && bandera==
TRUE)
      {
          point3D[sel].x=event->x-(widget->allocation.width/2);
          point3D[sel].z=-(event->y-(widget->allocation.height/2));
         Repaint (widget);
      }

...
}
```

La última función es muy simple, lo único que hace es detectar que la tecla que se liberó fue la tecla número dos, de ser así la bandera se inicializa al igual que el índice a la estructura *point3D*.

```
gboolean
on_draw_area_Front_button_release_event
                                        (GtkWidget       *widget,
                                        GdkEventButton   *event,
                                        gpointer         user_data)
{
      if (event->button == 2 )
         {
            bandera=FALSE;
            sel=0;
         }

  return FALSE;
}
```

3.3.3 .- Función Repaint

La función *Repaint* es extensa, pero en general realiza los mismos pasos para las cuatro áreas de dibujo, sólo en el área de perspectiva difiere, ya que es necesario desplegar las líneas en perspectiva caballera utilizando el método que se vio en el capítulo 1, es por eso que de igual forma solo analizaremos la parte de código correspondiente a el área frontal y como se logro visualizar la perspectiva.

Las primeras líneas corresponden a las variables y widgets utilizados para pintar las diferentes áreas de dibujo.

```
static void Repaint( GtkWidget *widget)
     {
           GtkWidget *front_to_repaint;
           GtkWidget *up_to_repaint;
           GtkWidget *side_to_repaint;
           GtkWidget *persp_to_repaint;

      GdkRectangle update_Draw_area;
      GdkPoint update_point[cont];
      GdkFont *font;

      GdkRectangle punto; //PARA LOS PUNTOS
      int c=0;


       front_to_repaint = lookup_widget(widget, "draw_area_Front");

...

font = gdk_font_load ("-abisource-arial-bold-r-normal-*-*-140-*-*-
p-*-iso8859-1");

       //borramos las áreas de dibujo

...
```

El siguiente paso es borrar el área de dibujo, para ello se utiliza la función *gdk_draw_rectangle.*

```
/*** Frontal ******/
gdk_draw_rectangle (pixmap_front,
                    penWhite,
                    TRUE,
                    0, 0,
                   front_to_repaint->allocation.width,
                   front_to_repaint->allocation.height);

...
```

Después se dibujan los ejes y etiquetas de diferentes colores, verde (*penGreen*) para el eje Y ,azul (*penBlue*) para el eje X y rojo (*penRed*) para el eje Z.

```
      //dibujamos los ejes coordenados
...
/*** Frontal ******/

gdk_draw_line (pixmap_front,
               penRed,
               front_to_repaint->allocation.width/2, 0,
               front_to_repaint->allocation.width/2,
               front_to_repaint->allocation.height);

gdk_draw_line (pixmap_front,
               penBlue,
               0,front_to_repaint->allocation.height/2,
               front_to_repaint->allocation.width,
               front_to_repaint->allocation.height/2);

      //etiquetas de los ejes
gdk_draw_string(pixmap_front,font,
                penRed,
               (front_to_repaint->allocation.width/2)-10,10,"Z");

gdk_draw_string(pixmap_front,font,
                penBlue,
               (front_to_repaint->allocation.width)-10,
               front_to_repaint->allocation.height/2,"X");

...
```

Después se dibujan los puntos y líneas, para ello se recorre la estructura desde 0 hasta el número total de puntos creados y se dibuja dicho punto con la función *gdk_draw_rectangle* en el buffer respectivo. De manera similar se actualizan las líneas que unen a estos puntos con la función *gdk_draw_lines.*

```
      //Dibujar líneas y puntos
...
/*** Frontal ***/
```

```
for(c=0;c<cont;c++)
  {
       punto.x = point3D[c].x -2;
       punto.y = -(point3D[c].z + 2);
       punto.width = 4;
       punto.height = 4;
       gdk_draw_rectangle (pixmap_front,
                           penBlack,
                           TRUE,
                          (front_to_repaint-
>allocation.width/2)+punto.x,
                          (front_to_repaint-
                          >allocation.height/2)+punto.y,
                           punto.width, punto.height);

update_point[c].x= punto.x + (front_to_repaint-
>allocation.width/2)+1;
update_point[c].y= punto.y + (front_to_repaint-
>allocation.height/2)+1;

  }

gdk_draw_lines(pixmap_front,
               penBlack,
               update_point,
               cont);
```

El siguiente código dibuja las líneas en el área de perspectiva y como ya habíamos dicho aquí difiere un poco con respecto a las otras áreas. Nótese que para obtener los puntos x e y los valores de la estructura *point3D* es multiplicada haciendo uso de la ecuación número 7 del capítulo 1.

$$[point3D[c].x \quad point3D[c].y \quad point3D[c].z] * \begin{bmatrix} 1 & 0 \\ \cos\varphi & sen\varphi \\ 0 & 1 \end{bmatrix} = [X2d, Y2d]$$

```
/*** Perspectiva ***/
for(c=0;c<cont;c++)
  {
       punto.x = point3D[c].x + (point3D[c].y * cos(ANGLE3D)*0.75) -
2;
       punto.y = -(point3D[c].z + (point3D[c].y *
sin(ANGLE3D)*0.75)+ 3);
       punto.width = 4;
       punto.height = 4;

    update_point[c].x=punto.x + (persp_to_repaint-
>allocation.width/2)+2;
    update_point[c].y=punto.y + (persp_to_repaint-
>allocation.height/2)+2;
```

```
    }

gdk_draw_lines(pixmap_persp,
                        penWhite,
                        update_point,
                        cont);
```

Con las siguientes líneas de código se obliga a llamar el evento *expose_event*, es decir se manda a llamar a la función *on_draw_area_Front_expose_event*, que ya sabemos como funciona.

```
...

      //forzar expose event
      update_Draw_area.x = 0;
      update_Draw_area.y = 0;
      update_Draw_area.width = front_to_repaint->allocation.width;
      update_Draw_area.height = front_to_repaint-
>allocation.height;


      //Expose event del area frontal
      gtk_widget_draw (front_to_repaint, &update_Draw_area);

}     //fin de la función Repaint
```

La figura 3.3 muestra la representación de un cubo regular para demostrar que efectivamente se logro el efecto de profundidad deseado y planteado en los capítulos anteriores.

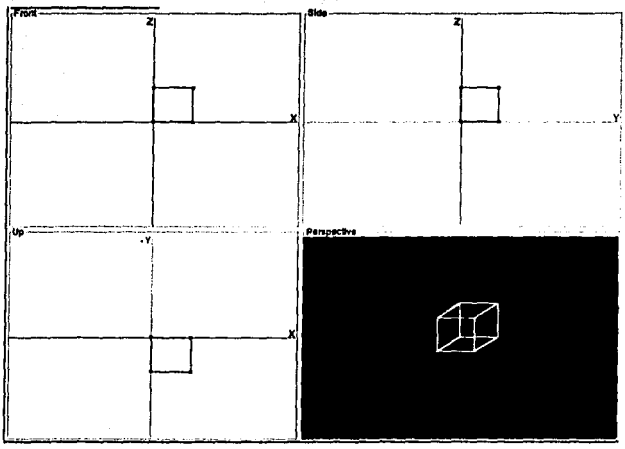

Figura 3.3 : Visualización de un cubo

3.3.4 .- Cálculo de puntos

Como habíamos visto en el capítulo anterior cuando se define una trayectoria (trayectoria de objetos dirigidos) es necesario calcular los puntos intermedios, para ello el programa necesita saber el número de puntos que se necesitan calcular. En la figura 3.4 tenemos un cuadro de entrada en el cual podemos proporcionarle este dato al editor de trayectorias y un botón cuya función de retrollamada se encargara de hacer dichos cálculos.

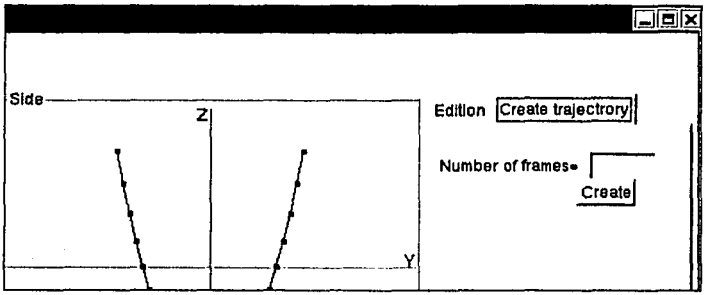

Figura 3.4 : Creación de trayectoria.

En sí esta función es el código en gtk+ de lo ya explicado en el sub-capítulo 2.1, el resultado de este cálculo es almacenado en un arreglo llamado trj de tipo *type3D*, a partir del cual se creará la animación.

```
void on_Btn_cal_trj_clicked(GtkButton       *button,
                            gpointer        user_data)
{

      GtkWidget *entry_frm;

      //Obtenemos los widgets

      gint j=0;
      gfloat longit=0.0,tmp=0.0,tmpx=0.0;
      gfloat  tam_seg=0.0, comp=0.0,interm=0.0,pasada=1.0;
      gfloat k=0.0;

      type3D aux;
      gint r=0,index_trj=0;

      entry_frm = lookup_widget(GTK_WIDGET(button), "num_frames");


      /* obtenemos el texto de entrada y se asigna a el arreglo
      convertimos a entero */

      num_frm=atoi(gtk_entry_get_text(GTK_ENTRY(entry_frm)));

      // validar que se introduce un valor
      if(num_frm < 1)num_frm=2;

/* Calcular longitud de la trayectoria */
```

```
        for(j=0;j<cont-1;j++)
        {

        longit=longit+sqrt(
                        pow(point3D[j+1].x-point3D[j].x , 2 ) +
                        pow(point3D[j+1].y-point3D[j].y , 2 ) +
                        pow(point3D[j+1].z-point3D[j].z , 2 ) );
         }

/*calculamos el tamaño de los segmentos
       tam_seg=longit/num_frm;

/* Calcular puntos intermedios de la trayectoria */

for(j=0;j<cont-1;j++)
{
       aux.x=point3D[j].x;
       aux.y=point3D[j].y;
       aux.z=point3D[j].z;

       tmp=sqrt(pow(aux.x-point3D[j+1].x , 2 ) +
                   pow(aux.y-point3D[j+1].y , 2 ) +
                   pow(aux.z-point3D[j+1].z , 2 ) );
       g_print("tmp=%f\n",tmp);

tmpx=tmp;
       while(tam_seg <= tmpx)
       {
       k=tam_seg*pasada;
       k=tmp-k;
       k=(tam_seg*pasada)/k;

       k=k+1.0;
        trj[index_trj].x=(aux.x+k*point3D[j+1].x)/k;
        trj[index_trj].y=(aux.y+k*point3D[j+1].y)/k;
        trj[index_trj].z=(aux.z+k*point3D[j+1].z)/k;

        index_trj++;
        pasada=pasada+1.0;


        tmpx=tmpx-tam_seg;

       }
       pasada=1.0;
}

/* se asigna el ultimo punto */
trj[index_trj].x=point3D[cont].x;
trj[index_trj].y=point3D[cont].y;
trj[index_trj].z=point3D[cont].z;

}
```

3.3.5 .- Manejo de archivos

Como habíamos visto la barra de herramientas cuenta con 5 botones, los cuatro primeros tienen que ver con el manejo de archivos, crear un archivo nuevo, cargar una trayectoria en un archivo. Las acciones como son crear uno nuevo y cerrar son realmente sencillos, nos concentraremos en la recuperación y guardado de datos en archivo.

Los botones de la barra de herramientas responden al evento *clicked* con lo cual es posible conectarlos con funciones de retrollamada. Para el botón *Load* tenemos la siguiente conexión.

```
gtk_signal_connect (GTK_OBJECT (Btn_Load), "clicked",
                    GTK_SIGNAL_FUNC (on_Btn_Load_clicked),
                    NULL);
```

La función de retrollamada *on_Btn_Load_clicked()* se encarga de crear y desplegar un widget del tipo *fileselection_open*, figura 3.5. El único objetivo de este widget es obtener la ruta completa de un archivo. Esto le permite al usuario buscar el archivo que contiene los puntos a cargar en memoria para posteriormente ser dibujado en las cuatro áreas de dibujo. El código para esto es el siguiente.

```
Void on_Btn_Load_clicked(GtkButton       *button,
                         gpointer        user_data)
{
   GtkWidget *fileselection_open;

      fileselection_open = create_fileselection_open ();
      gtk_widget_show (fileselection_open);
}
```

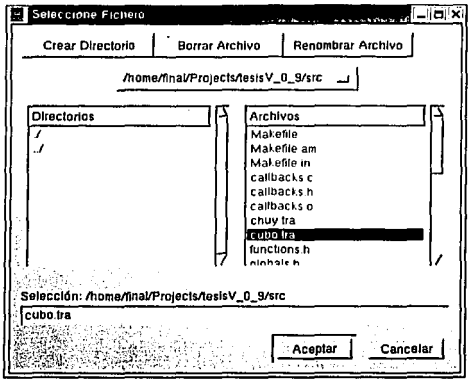

Figura 3.5 : Widget para la selección de archivo.

Igualmente el botón aceptar de este widget responde al evento clicked y la función para el tratamiento de este evento es el siguiente.

```
void on_ok_button_open_clicked (GtkButton       *button,
                                gpointer         user_data)
{
   GtkWidget *ObtainFileName;
   GtkWidget *window;

   FILE *FileRead;
   int i=0;

   window= lookup_widget(GTK_WIDGET(button), "fileselection_open");


   ObtainFileName = lookup_widget (GTK_WIDGET(button),
   "fileselection_open");

/* Obtener la trayectoria completa y guardarlo en la variable
correspondiente */
   fileName_trj = gtk_file_selection_get_filename
   (GTK_FILE_SELECTION(ObtainFileName));

/* Se abre dicho archivo en modo solo lectura */
   FileRead=fopen(fileName_trj,"r");

/* Se obtiene el numero de puntos y se asigna a la variable cont */
      /* Obtener el numero de puntos */
      fscanf(FileRead,"%d ",&cont);
```

```
/* Se leen todos los datos y se introducen en el arreglo point3D */
      for(i=0;i<cont;i++)
       {
        fscanf(FileRead,"%d ",&point3D[i].x);
        fscanf(FileRead,"%d ",&point3D[i].y);
        fscanf(FileRead,"%d ",&point3D[i].z);
       }


  fclose(FileRead);

/* Se manda a llamar a la función Repaint */
  Repaint(window);

  gtk_widget_destroy (window);


}
```

Los comentarios del código anterior lo explican claramente, como se puede observar el resultado es la sustitución de valores en memoria y una vez que hecho esto se dibujan los nuevos valores con la función *Repaint.*

Las funciones de retrollamada para guardar son prácticamente lo mismo que las anteriores, la diferencia radica en que el origen y el destino de los datos son inversos. El código de la función *on_ok_button1_clicked* perteneciente al widget de selección *file_sel _save es el siguiente.*

```
Void on_ok_button1_clicked(GtkButton       *button,
                           gpointer        user_data)
{
   GtkWidget *ObtainFileName;
   GtkWidget *window;

   FILE *FileSave;
   int i=0;

   window= lookup_widget(GTK_WIDGET(button), "file_sel_save");


   ObtainFileName = lookup_widget (GTK_WIDGET(button),
"file_sel_save");
   fileName_trj = gtk_file_selection_get_filename (
      GTK_FILE_SELECTION(ObtainFileName));

   FileSave=fopen(fileName_trj,"w+");


      /* Guardando el número de puntos */
      fprintf(FileSave,"%d \n",cont);
```

```
        /* Guardando los puntos */
        for(i=0;i<cont;i++)
         {
          fprintf(FileSave, "%d %d %d\n",
                    point3D[i].x,
                    point3D[i].y,
                    point3D[i].z);
         }

      fclose(FileSave);

      gtk_widget_destroy (window);
    }
```

En general ya hemos visto la parte de código más importante de nuestra aplicación el código completo se encuentra en el anexo A y disponible en el sitio del proyecto de Material 3D [R9].

CAPÍTULO IV .- IMPLEMENTACIÓN

Objetivo:

- Realizar un ejemplo sencillo de animación para determinar la eficiencia de la aplicación.

Para probar el funcionamiento del editor de trayectorias se va a crear una animación sencilla siguiendo las etapas explicadas en el capítulo 1, algunas de estas etapas son explicadas con el mayor detalle posible, otras no tanto dado que en algunas de ellas no es posible ilustrarlo en este documento.

4.1 .-PLANTEAMIENTO DEL PROBLEMA

Para la creación de la animación vamos a hacer uso de los datos de un proyecto de robótica que fue desarrollado en una asignatura correspondiente a la Maestría en Ciencia e Ingeniería de la Computación el semestre 2002-2 por los alumnos Salazar Franco Marco Antonio y Ramírez Guzmán Ramón. A continuación se explica la naturaleza de este proyecto y la forma en que se relacionará con nuestro editor de trayectorias.

El objetivo de este proyecto era programar un robot de tal forma que evitara obstáculos(mesas de trabajo) en un laboratorio de robótica, como primera entrada de dicho programa es la posición de donde partirá el robot en el laboratorio y como segunda la posición destino; la salida es un conjunto de datos que define la trayectoria calculada entre la posición origen y destino. Un diagrama del laboratorio se muestra en la figura 4.1.

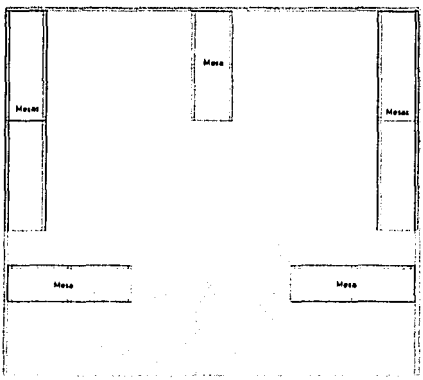

Figura 4.1: Diagrama del laboratorio de robótica

La forma en que dicho robot resuelve esta trayectoria es por medio de sensores que detectan cuando se ha hecho contacto con un obstáculo, en tal caso el robot se detiene, vuelve a calcular su trayecto y continúa. Aunque suene sencillo existen aspectos interesantes que hay que tomar en cuenta, pero sería profundizar en un tema que por el momento no nos ocupa, en vez de ello nos involucramos con los datos que arroja dicho robot al concluir su trayectoria en un archivo.

Nuestra prueba consistirá en hacer una animación a partir de estos datos, es decir, vamos a simular el trayecto de dicho robot en la habitación. Dicho lo anterior podemos describir nuestra escena a crear, es decir podemos escribir nuestro guión.

4.2 GUIÓN

Para la descripción del escenario hacemos uso del diagrama de la figura 4.1, todas las mesas son del mismo tamaño y tienen una estructura metálica con pintura naranja y un tablón de color negro.

El objeto principal de la escena es un robot que será representado por un cilindro. La iteración de este robot con el escenario viene descrito en el archivo antes mencionado.

Dado que es una escena muy simple no hace falta desarrollar un story board ya que todas las posiciones del robot vienen contenidas en el archivo de datos.

4.3 MODELADO DE OBJETOS Y ESCENAS

En la figura 4.2 se puede ver el modelado de escenas con Material 3D, a continuación se explica la construcción de los objetos.

Mesas: Para el modelado de cada mesa se hizo uso de 9 cubos de diferentes dimensiones, 8 para la estructura y 1 para el tablón. Las 7 mesas fueron posicionadas en la escena según la posición real en el laboratorio.

Robot: Como ya lo habíamos dicho se hizo uso de un cilindro del dimensionado según el robot real.

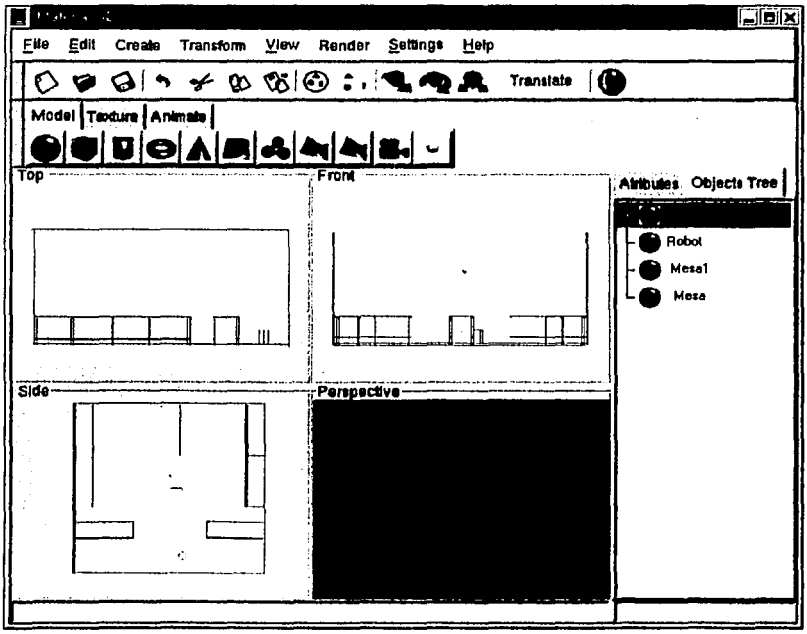

Figura 4.2: Modelado de laboratorio

Escenario (Laboratorio): El tamaño del laboratorio es de 10 metros de largo por 10 de fondo. El laboratorio real cuenta con ventanas pero para facilitar el modelado se omitieron.

La parte central del modelo es decir el centro del laboratorio fue situada en el origen de los ejes coordenados del visualizador de material 3D, esto es importante dado que utilizando esta referencia es como vamos a generar nuestra trayectoria.

Esta etapa es laboriosa y requiere de tiempo para finalizarla debido a la gran cantidad de primitivas utilizadas.

4.4 .-POSICIÓN DE LA CÁMARA.

La posición de la cámara será fija con coordenadas (0,-15,6), la orientación de la cámara será al centro de la escena, en la figura 4.3 se muestra una imagen generada para darnos la idea de la posición de la cámara.

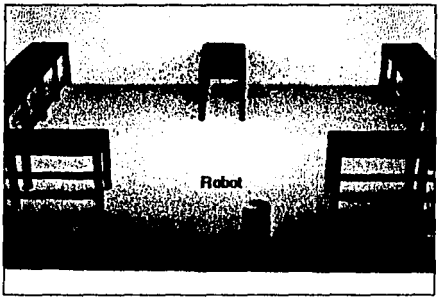

Figura 4.3: Posición de la cámara.

Con esta perspectiva podemos apreciar el movimiento del robot y no es necesario mover la cámara a través de la habitación. Esto simplifica nuestro trabajo.

4.5 .-EDICIÓN DE TRAYECTORIAS

Hemos llegado a la parte más importante para nosotros, la edición de trayectorias, para cuyo caso haremos uso del los datos del archivo del cual ya hemos platicado, dicho archivo se le ha dado formato para que nuestro editor de trayectorias pueda leerlo, el contenido del nuevo archivo es el siguiente:

```
11
10 10 0
9 9 0
3.8 7.9 0
0 6.0 0
-2.6 3.4 0
-2.3 0.9 0
1.5 0.4 0
3.7 -1.2 0
1.4 -3.2 0
5.3 -5.1 0
2.7 -7.6 0
```

Los llamaremos trayectoria inicial, debido a que haremos uso de nuestra aplicación para mejorarla, es decir calcular puntos intermedios a partir de estos puntos. La representación de dicha trayectoria con nuestra aplicación se pueden observar en la figura 4.4, podemos observar que efectivamente los puntos están muy separados y esto da como resultado una animación de baja calidad debido a que el movimiento de el robot luce poco uniforme en los virajes que realiza para evitar obstáculos.

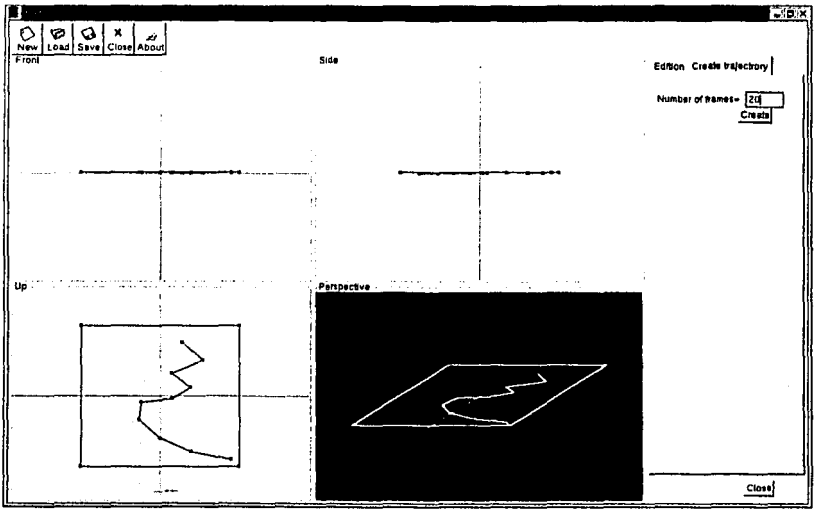

Figura 4.4: trayectoria inicial de nuestra animación.

Para mejorar el movimiento del robot vamos a aumentar el número de puntos y para ello introduciremos un número más grande de cuadros, en este caso queremos obtener 20 puntos intermedios. En la figura 4.5 podemos observar que introducimos este valor para después hacer clic en el botón *create* para obtener los resultados que se observan en la figura 4.6.

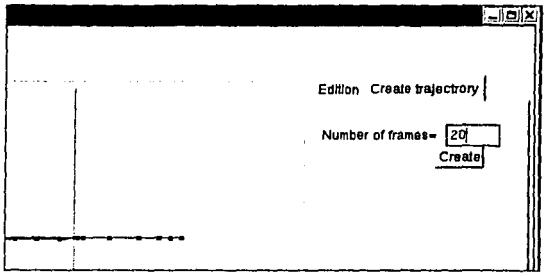

Figura 4.5:Introducción del número de cuadros

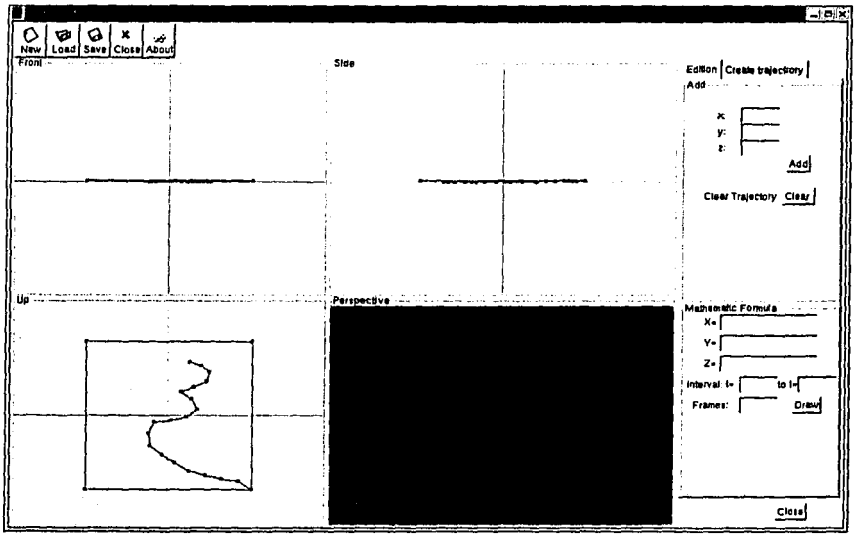

Figura 4.6: Resultado del cálculo de la trayectoria.

Este nuevo conjunto de puntos en el espacio efectivamente suaviza la trayectoria, pero se necesitan más puntos para lograr una animación de buena calidad, en este caso vamos a dejar este ejemplo con 20 puntos intermedios.

Como resultado obtendremos el siguiente conjunto de puntos, que pueden ser almacenados en un archivo.

```
20
10 10 0
8.4 8.9 0
6.5 8.6 0
4.6 8.1 0
2.6 7.5 0
0.9 6.4 0
-0.7 5.3 0
-2.1 4.1 0
-2.3 2.4 0
-1.6 0.9 0
0.4 0.7 0
2.3 0.2 0
3.6 -0.8 0
2.9 -2.2 0
1.6 -3.2 0
3.2 -3.8 0
4.7 -4.6 0
5.0 -5.9 0
4.1 -6.7 0
2.7 -7.2 0
```

4.6 GENERACIÓN DE LOS CUADROS INTERMEDIOS

Para la generación de cuadros intermedios hacemos uso de la máquina de render utilizada por material 3D. En este caso una muestra de las imágenes generadas se observa en la figura 4.7.

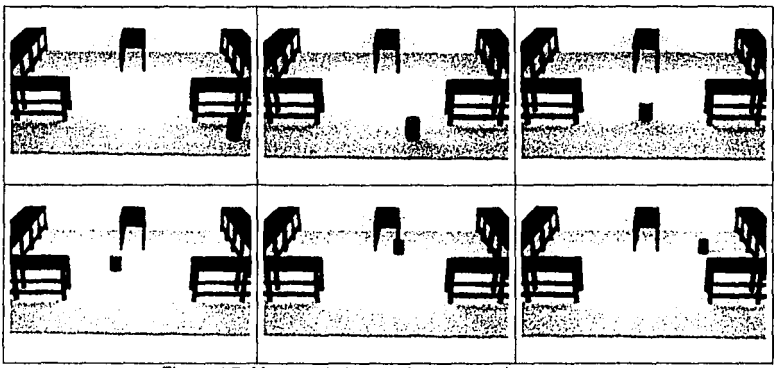

Figura 4.7: Muestra de los cuadros generados.

En la imagen de la figura se muestran 6 de los 20 cuadros intermedios generados, estos cuadros serán utilizados posteriormente para generar la animación.

4.7 .- EDICIÓN Y POSTPRODUCCIÓN

Para la edición y postproducción del video final se utilizo la versión de prueba de Adobe Premiere 6.0 el cual se muestra en la figura 4.8.

A grandes rasgos lo que se hizo fue tomar las 20 imágenes y editarlas, es decir se concatenaron a 12 cuadros por segundo, se le agregó un título de inicio y se exporto para obtener un archivo de video en formato "avi" con compresión de video MPEG-4.

El video resultante así como los archivos fuente del editor de trayectorias se puede obtener en *http://tigre.aragon.unam.mx/~jeshc/tesis/.*

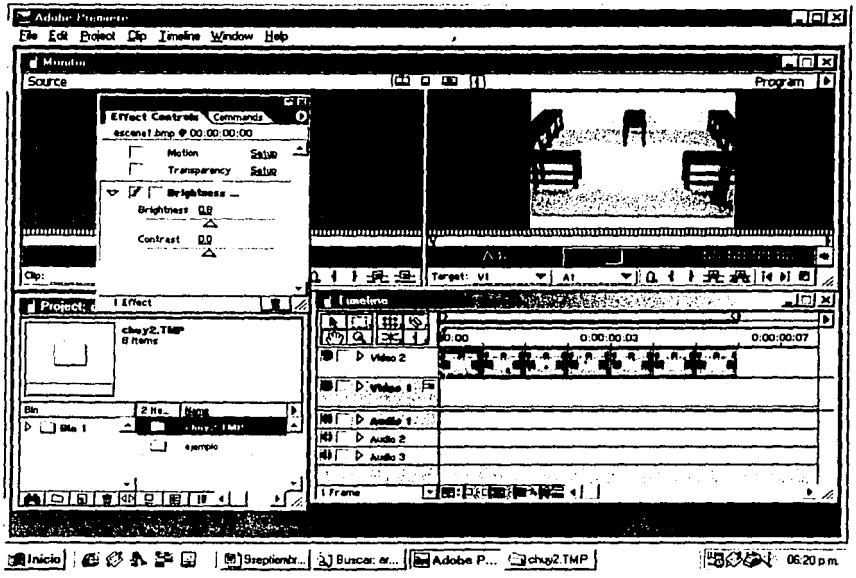

Figura 4.8: Editor de video Adobe Premiere 6.0,versión de prueba.

La ayuda que nos brindó nuestro editor de trayectorias disminuyó el tiempo empleado para la construcción de la animación, ya que eliminó por completo los cálculos en papel

y la realización de pruebas innecesarias. En el capítulo siguiente se analizarán los resultados de esta etapa para evaluar nuestra aplicación.

CAPÍTULO V .- EVALUACIÓN DE RESULTADOS

Objetivo:

- Evaluar resultados del caso práctico, tanto en tiempo de creación y calidad de la animación.

Para la evaluación de los resultados de dicha animación se llevó registro de los tiempos utilizados en cada una de las etapas y fue comparado con los tiempos de creación de una animación hecha en su totalidad con Pov-Ray y además se avaluaron aspectos de facilidad de uso y previsualización de modelos.

Esta animación se trata de la misma escena (explicada en el capítulo anterior) sólo que ha sido simplificada para ahorrar tiempo en el modelado, ya que como ya habíamos visto en Pov-Ray esta es una tarea complicada y tardada.

La simplificación consta en utilizar cubos en vez de construir las mesas a detalle, una muestra de la escena hecha con Pov-Ray se puede observar en l a figura 5.1.

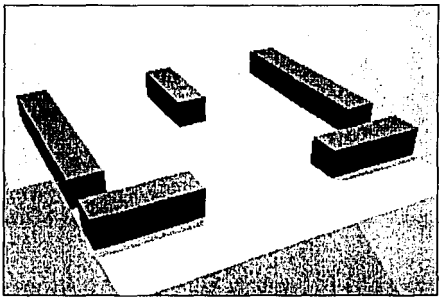

Figura 5.1: Escena simplificada, hecha en Pov-Ray.

Esto nos ahorró gran cantidad de tiempo, pero aún así la construcción de este modelo se llevo más tiempo que el creado en el capítulo anterior.

En los siguientes sub-capítulos analizamos esta y otras diferencias que nos darán una idea más clara de los beneficios obtenidos con nuestra aplicación.

5.1 .-TIEMPO DE CREACIÓN

Como se puede apreciar en la siguiente tabla el tiempo para el modelado de la escena llevo más tiempo(aún simplificada), con estos primeros datos podemos darnos cuenta de las ventajas de utilizar una interfaz gráfica par el diseño de modelos.

El ahorro de tiempo en esta etapa es de gran importancia ya que podemos utilizarlo en etapas que demandan también de el.

	Tiempo en minutos	
Etapa	**Material 3D**	**Pov-Ray**
Modelado	45	63
Edición de trayectorias	5	46
Creación de cuadros	2	12
Edición	30	30

El motivo de tal diferencia en tiempo en la etapa de modelado es la interfaz gráfica de usuario que proporciona material 3D para la creación de objetos, de esta forma se pueden previsualizar los objetos en modelo de alambre y así darnos una idea de los resultados finales.

A diferencia de Pov-Ray en el cual la única forma de evaluar que un modelo ha sido terminado y posicionado correctamente es generando constantemente(casi en cada cambio del código) una imagen.

En la edición de trayectorias es claro darse cuenta por qué es tal la diferencia, para el cálculo de los puntos intermedios en Pov-ray se tuvo que calcular con lápiz y papel los puntos intermedios, haciendo uso de la ecuación del punto que divide una recta en una razón dada, introducir los datos posición por posición e ir generando un cuadro a la vez. En cambio para la edición de trayectorias con Material 3D se modificó un módulo (Debido a que estamos en etapa de pruebas) para que leyera directamente los puntos generados por el editor de trayectorias y así generar las imágenes en conjunto.

El tiempo en la generación de cuadros en ambos casos fue realmente poco debido a que la máquina de "render" es independiente del modelador, por lo tanto la diferencia en esta etapa es poca. Pero tomando en cuenta que la generación de cuadros con Pov-ray fue hecha imagen por imagen.

5.2 .-CALIDAD DE LA ANIMACIÓN

Para la evaluación de la calidad se compararon ambos videos producidos y a primera vista pudimos ver que las trayectorias diferían en partes(tal ves por algún error humano de cálculo). Poniendo más atención en los datos producidos logramos determinar los puntos que difieren y es en esa parte en que el robot parece salirse de la línea recta entre un punto y otro.

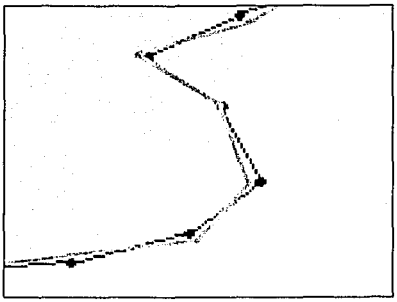

Figura 5.1: Error de cálculo

En esta figura graficamos un segmento de la trayectoria calculada a mano y podemos observar que esta no sigue una línea recta como debería ser, ello hace que la calidad de la animación se vea afectada. En este ejemplo la calidad se vio poco afectada debido a los pocos cuadros que se introdujeron (20), pero nos dimos cuenta que la tarea de calcular los puntos en papel es laboriosa y de ser el caso que los puntos intermedios deseados sean más, estamos propensos a cometer mayor número de errores y esto afectaría en mayor medida la calidad de la animación.

5.3 .- FACILIDAD DE USO

Con los apartados anteriores podemos justificar la facilidad de uso que tiene nuestra aplicación. El uso de nuestro editor de trayectorias se ve beneficiado en las características que a continuación se resumen.

La previsualización es importante en las herramientas de modelado por computadora, esto permite al usuario obtener información inmediata de lo que esta construyendo, de otra forma tendría que hacer uso de técnicas de prueba y error, esto disminuiría su facilidad de uso.

Con el uso del ratón para edición de trayectorias el usuario tiene la herramienta principal para la realización de su trabajo, de esta forma la trayectoria puede ser especificada o modificada a partir de la idea del usuario. Pero esto no es suficiente y otra característica que hace fácil su uso es la capacidad de generar trayectorias(o

aproximaciones) con el uso de ecuaciones matemáticas, en tal caso el usuario puede ahorrar mucho tiempo.

Además el editor de trayectorias permite la introducción de puntos por medio de cuadros de texto, esto aumenta su facilidad de uso ya que en ecuaciones el usuario deseará colocar puntos en coordenadas exactas y con el uso del ratón esta tarea seria difícil.

Una característica muy importante es el uso de archivos para el almacenado y recuperación de trayectorias, sobre todo la adaptación de archivos de salida de otras aplicaciones a nuestro editor de trayectorias. En la figura 5.2 se muestra el flujo de datos por archivos para la creación de animaciones a partir de datos generados por otros programas.

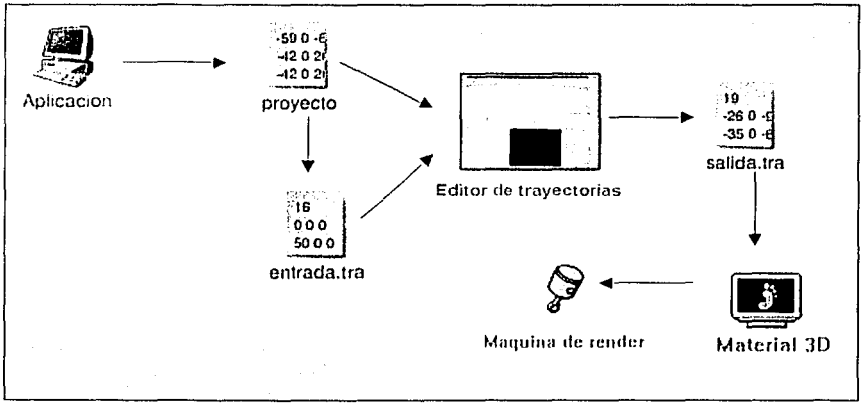

Figura 5.2: Diagrama de manejo de archivos

El último y más importante característica es el cálculo y generación automática de puntos.

CONCLUSIONES

Complementando la sección anterior podemos concluir que el editor de trayectorias logró su cometido simplificando las tareas y ofreciendo diferentes métodos para la definición de trayectorias, ya que como aprendimos las limitantes que antes existían además de consumir grandes cantidades de tiempo, aumentaban el riesgo de cometer un error. Además existen características que nos permiten darnos cuenta de las ventajas de utilizar nuestro software.

Una característica muy importante y medular de nuestro proyecto es la adopción de la licencia de software libre, lo cual lo provee de un valor agregado por las siguientes ventajas. La primera de ellas es la utilización gratuita del software sin ningún tipo de restricciones en la licencia, dicha ventaja, evidentemente no tiene comparación con el costo de los programas comerciales para la construcción de animaciones, lo cual beneficiará en un futuro áreas como la investigación, que comúnmente cuentan con poco presupuesto para el desarrollo de diversos proyectos. Una segunda ventaja es la disposición del código fuente para adaptarlas a las necesidades del usuario final.

Otra característica muy importante es su portabilidad con el uso de librerías gtk+ (al igual que material 3D)que permiten su compilación en diferentes plataformas, de esta forma satisfacemos una de nuestras necesidades planteadas al principio de nuestra tesis. Para comprobar la portabilidad de la que estamos hablando se hizo una prueba para la ejecución de Material 3D en Windows[17] y se obtuvieron los resultados esperados.

En la actualidad Material 3D y nuestro editor de trayectorias es una alternativa que en el futuro seguirá desarrollándose para ofrecer las funciones y las herramientas necesarias para la creación completa de una animación de calidad.

En general el proyecto es bastante ambicioso y se requiere invertir tiempo en el diseño y la programación para lograr alcanzar la calidad de los software comerciales. Aún queda mucho por hacer para hacer de la edición de trayectorias una tarea más fácil,

[17] Utilizando Visual C++, para mas detalles como compilar bajo Windows un proyecto gtk+ visite la pagina de gtk+ [R3]

pero esta primera propuesta mejora por mucho la creación de animaciones con software gratuito.

El objetivo de este proyecto es también invitar a la comunidad de software libre[18] a participar en el proyecto para su continua mejora y desarrollo, de esta forma el camino para llegar a una versión más completa de nuestro software será más corto.

Cómo conclusiones adicionales podemos agregar los siguientes puntos:

Con bases matemáticas relativamente sencillas es posible representar y calcular trayectorias reales de forma convincente.

Programas con buenas características si existen pero debido a su carácter comercial tienen limitaciones debido al alto costo de las licencias y su nulo acceso al código fuente.

Al utilizar GTK+ y "glib", librerías de fuente abierta facilitamos la migración a diferentes plataformas.

[18] Para depurar y programar otros módulos como el proyecto del sistema operativo Linux.

ANEXO A .- CÓDIGO FUENTE

Debido a que el código fuente de nuestra aplicación es bastante amplio solo incluiremos en el presente anexo el código fuente correspondiente al programa principal (main.c) y el correspondiente a las funciones de retrollamada (callbacks.c), que contienen toda la funcionalidad.

La lista de los archivos principales del código fuente se muestran a continuación.

Nombre	Descripción
main.c	Programa principal, crea la interfaz (haciendo uso del archivo interface.h) e inicia el bucle gtk_main() para la espera de eventos.
functions.h	Contiene funciones prototipo utilizadas en callbacks.h
globals.h	Contiene las variables globales y estructuras de datos.
iterface.c	Contiene funciones para la creación de ventanas y widgets para la interfaz de usuario.
interface.h	Funciones prototipo para interfaz.c
support.c	Contiene las funciones para crear la interface de la aplicación
support.h	Contiene funciones prototipo para support.c
callbacks.c	Contiene las funciones de retrollamada que manejan los eventos que son enviados por gtk_main()
callbacks.h	Contiene las funciones prototipo de callback.c

```
/*****************************************************************
  main.c
  Editor de trayectorias versión 0.9
  2 de Julio del 2002
  Jesús Hernández Cabrera
*****************************************************************/

#ifdef HAVE_CONFIG_H
 include <config.h>
#dif

#clude <gnome.h>


:erface.h
:tiene las funciones para crear la interface de la aplicacion

#clude "interface.h"
 support.c
:tiene utilerias para obtencion de caracteristicas de widgets*/

#clude "support.h"

:
in (int argc, char *argv[])

GtkWidget *main_window;
GtkWidget *about1;
GtkWidget *fileselection_open;

#def ENABLE_NLS
bindtextdomain (PACKAGE, PACKAGE_LOCALE_DIR);
:extdomain (PACKAGE);
#dif

  // nombre de la aplicación
gnome_init ("tesisv_0_9", VERSION, argc, argv);

  //Se crea la ventana principal
main_window = create_main_window ();

  //Se despliega
gtk_widget_show (main_window);

  //Se crea un cuadro de dialogo, Acerca de...
about1 = create_about1 ();
gtk_widget_show (about1);

  //Se crea un cuadro de dialogo para la sel de ficheros
fileselection_open = create_fileselection_open ();
  gtk_widget_show (fileselection_open);

  //Loop principal
gtk_main ();
return 0;
```

```
*****************************************************************
 callbacks.c                                                     *
 Editor de trayectorias versión 0.9                              *
 2 de Julio del 2002                                             *
 Jesús Hernández Cabrera                                         *
                                                                 *
*****************************************************************/

def HAVE_CONFIG_H
include <config.h>
iif

clude <gnome.h>
clude <stdio.h>

clude "callbacks.h"
clude "interface.h"
clude "support.h"
clude "functions.h"
clude "globals.h"


3D clic;
3D point3D[100];
3D trj[100];

*****************************************************************
                                                                 *
 Vista de perspectiva                                            *
                                                                 *
*****************************************************************/
lean
raw_area_Persp_configure_event      (GtkWidget       *widget,
                                     GdkEventConfigure *event,
                                     gpointer         user_data)

pixmap_persp)
      gdk_pixmap_unref(pixmap_persp);

    pixmap_persp = gdk_pixmap_new(widget->window,
                                  widget->allocation.width,
                                  widget->allocation.height,
                                  -1);
    gdk_draw_rectangle (pixmap_persp,
                        widget->style->black_gc,
                        TRUE,
                        0, 0,
                        widget->allocation.width,
                        widget->allocation.height);


turn TRUE;



lean
raw_area_Persp_expose_event         (GtkWidget       *widget,
                                     GdkEventExpose  *event,
                                     gpointer         user_data)


gdk_draw_pixmap(widget->window,
                widget->style->fg_gc[GTK_WIDGET_STATE (widget)],
                pixmap_persp,
                event->area.x, event->area.y,
                event->area.x, event->area.y,
                event->area.width, event->area.height);


turn FALSE;


lean
```

```
on_draw_area_Persp_button_release_event
                                        (GtkWidget       *widget,
                                        GdkEventButton  *event,
                                        gpointer         user_data)
{

  return FALSE;
}


gboolean
on_draw_area_Persp_motion_notify_event (GtkWidget       *widget,
                                        GdkEventMotion  *event,
                                        gpointer         user_data)
{

  return FALSE;
}


gboolean
on_drawingarea_Persp_button_press_event
                                        (GtkWidget       *widget,
                                        GdkEventButton  *event,
                                        gpointer         user_data)
{

  return FALSE;
}


/*************************************************************************
*                                                                        *
*   Vista Lateral                                                        *
*                                                                        *
*************************************************************************/


gboolean
on_draw_area_Side_button_press_event    (GtkWidget       *widget,
                                        GdkEventButton  *event,
                                        gpointer         user_data)
{

    int temp=0;
    if (event->button == 1 && pixmap_front != NULL)
        {
          //Introducimos un nevo punto en la vista frontal con las cadenas
         point3D[cont].y=event->x-(widget->allocation.width/2);
         point3D[cont].z=-(event->y-(widget->allocation.height/2));
         point3D[cont].x=0;


         cont++;
         Repaint(widget);
        }
    if (event->button == 2 && pixmap_side != NULL)
        {

                clic.y=event->x-(widget->allocation.width/2);
         clic.z=-(event->y-(widget->allocation.height/2));
         for(temp;temp<cont;temp++)
         {
           if(clic.y < point3D[temp].y+4 && clic.y > point3D[temp].y-4 &&
                clic.z < point3D[temp].z+4 && clic.z > point3D[temp].z-4)

             {
                        sel=temp;
               bandera=TRUE;
             }
         }

        }
g_print("Clic en Side: y=%d z=%d \n",point3D[cont-1].y,point3D[cont-1].z);
       return TRUE;
```

```
}

gboolean
on_draw_area_Side_configure_event       (GtkWidget         *widget,
                                         GdkEventConfigure *event,
                                         gpointer           user_data)

{
    if (pixmap_side)
        gdk_pixmap_unref(pixmap_side);

       pixmap_side = gdk_pixmap_new(widget->window,
                                    widget->allocation.width,
                                    widget->allocation.height,
                                    -1);
       gdk_draw_rectangle (pixmap_side,
                           widget->style->white_gc,
                           TRUE,
                           0, 0,
                           widget->allocation.width,
                           widget->allocation.height);


  return TRUE;
}

gboolean
on_draw_area_Side_expose_event          (GtkWidget         *widget,
                                         GdkEventExpose    *event,
                                         gpointer           user_data)

{

    gdk_draw_pixmap(widget->window,
                    widget->style->fg_gc[GTK_WIDGET_STATE (widget)],
                    pixmap_side,
                    event->area.x, event->area.y,
                    event->area.x, event->area.y,
                    event->area.width, event->area.height);


  return FALSE;
}

gboolean
on_draw_area_Side_button_release_event (GtkWidget         *widget,
                                        GdkEventButton    *event,
                                        gpointer           user_data)

{
    if (event->button == 2 )
       {
        bandera=FALSE;
        sel=0;
       }
    return FALSE;
}


gboolean
on_draw_area_Side_motion_notify_event  (GtkWidget         *widget,
                                        GdkEventMotion    *event,
                                        gpointer           user_data)

{
    int x, y;

       GdkModifierType state;

       if (event->is_hint)
         gdk_window_get_pointer (event->window, &x, &y, &state);
       else
         {
           x = event->x;
           y = event->y;
           state = event->state;
         }
```

```
if (state & GDK_BUTTON2_MASK && pixmap_front != NULL && bandera== TRUE)

     point3D[sel].y=event->x-(widget->allocation.width/2);
 point3D[sel].z=-(event->y-(widget->allocation.height/2));
  Repaint (widget);


return TRUE;



*************************************************************
                                                            *
sta superior                                                *
                                                            *
*************************************************************/


n
v_area_Up_button_press_event      (GtkWidget       *widget,
                                   GdkEventButton  *event,
                                   gpointer         user_data)

: temp=0;
(event->button == 1 && pixmap_front != NULL)
{
  //Introducimos un nevo punto en la vista FRONTAL CON LA COTA A CERO
 point3D[cont].x=event->x-(widget->allocation.width/2);
 point3D[cont].y=event->y-(widget->allocation.height/2);
 point3D[cont].z=0;


 cont++;
 Repaint(widget);
}
(event->button == 2 && pixmap_up != NULL)
{

         clic.x=event->x-(widget->allocation.width/2);
 clic.z=event->y-(widget->allocation.height/2);
 for(temp;temp<cont;temp++)
 {
   if(clic.x < point3D[temp].x+4 && clic.x > point3D[temp].x-4 &&
       clic.z < point3D[temp].y+4 && clic.z > point3D[temp].y-4)

     {
              sel=temp;
       bandera=TRUE;
     }
 }

)
("Clic en Up: x=%d y=%d \n",point3D[cont-1].x,point3D[cont-1].y);
return TRUE;



n
_area_Up_configure_event          (GtkWidget       *widget,
                                   GdkEventConfigure *event,
                                   gpointer         user_data)

(pixmap_up)
  gdk_pixmap_unref(pixmap_up);

pixmap_up = gdk_pixmap_new(widget->window,
                           widget->allocation.width,
                           widget->allocation.height,
                           -1);
gdk_draw_rectangle (pixmap_up,
                    widget->style->white_gc,
                    TRUE,
                    0, 0,
                    widget->allocation.width,
                    widget->allocation.height);
```

```
    if(penWhite == NULL) penWhite = GetPen (NewColor (0xffff, 0xffff, 0xffff));
    if(penBlack == NULL) penBlack = GetPen (NewColor (0, 0, 0));
    if(penBlue == NULL) penBlue = GetPen (NewColor (0, 0, 0xffff));
    if(penRed == NULL) penRed = GetPen (NewColor (0xffff,0, 0));
    if(penGreen == NULL) penGreen = GetPen (NewColor ( 0x00ff, 0xffffff,0x00ff));
    if(penGray == NULL) penGray = GetPen (NewColor (0x9000,0x9000,0x9000));

        Repaint(widget);

  return TRUE;
}


gboolean
on_draw_area_Up_expose_event            (GtkWidget        *widget,
                                         GdkEventExpose   *event,
                                         gpointer          user_data)
{

    gdk_draw_pixmap(widget->window,
                    widget->style->fg_gc[GTK_WIDGET_STATE (widget)],
                    pixmap_up,
                    event->area.x, event->area.y,
                    event->area.x, event->area.y,
                    event->area.width, event->area.height);


  return FALSE;
}


gboolean
on_draw_area_Up_button_release_event    (GtkWidget        *widget,
                                         GdkEventButton   *event,
                                         gpointer          user_data)
{
    if (event->button == 2 )
       {
        bandera=FALSE;
        sel=0;
       }
  return FALSE;
}


gboolean
on_draw_area_Up_motion_notify_event     (GtkWidget        *widget,
                                         GdkEventMotion   *event,
                                         gpointer          user_data)
{
    int x, y;

       GdkModifierType state;

       if (event->is_hint)
         gdk_window_get_pointer (event->window, &x, &y, &state);
       else
         {
           x = event->x;
           y = event->y;
           state = event->state;
         }

       if (state & GDK_BUTTON2_MASK && pixmap_front != NULL && bandera== TRUE)
    {
          point3D[sel].x=event->x-(widget->allocation.width/2);
        point3D[sel].y=event->y-(widget->allocation.height/2);
         Repaint (widget);
    }

       return TRUE;
}


/*************************************************************************
*                                                                        *
```

```
*   Vista de frente                                                    *
*                                                                      *
**********************************************************************/

gboolean
on_draw_area_Front_button_press_event  (GtkWidget       *widget,
                                        GdkEventButton  *event,
                                        gpointer         user_data)
{
    int temp=0;
    if (event->button == 1 && pixmap_front != NULL)
        {
          //Introducimos un nevo punto en la vista FRONTAL CON LA COTA A CERO
         point3D[cont].x=event->x-(widget->allocation.width/2);
         point3D[cont].z=-(event->y-(widget->allocation.height/2));
         point3D[cont].y=0;


         cont++;
         Repaint(widget);
        }
    if (event->button == 2 && pixmap_front != NULL)
        {

                clic.x=event->x - (widget->allocation.width/2);
         clic.z=-(event->y - (widget->allocation.height/2));
         for(temp;temp<cont;temp++)
         {
           if(clic.x < point3D[temp].x+4 && clic.x > point3D[temp].x-4 &&
                clic.z < point3D[temp].z+4 && clic.z > point3D[temp].z-4)

              {
                        sel=temp;
               bandera=TRUE;
              }
         }

        }
g_print("Clic en Front:x=%d z=%d \n",point3D[cont-1].x,point3D[cont-1].z);
       return TRUE;

}


gboolean
on_draw_area_Front_configure_event      (GtkWidget       *widget,
                                         GdkEventConfigure *event,
                                         gpointer         user_data)
{

    if (pixmap_front)
      gdk_pixmap_unref(pixmap_front);

       pixmap_front = gdk_pixmap_new(widget->window,
                                     widget->allocation.width,
                                     widget->allocation.height,
                                     -1);
       gdk_draw_rectangle (pixmap_front,
                           widget->style->white_gc,
                           TRUE,
                           0, 0,
                           widget->allocation.width,
                           widget->allocation.height);

  return TRUE;
}


gboolean
on_draw_area_Front_expose_event         (GtkWidget       *widget,
                                         GdkEventExpose  *event,
                                         gpointer         user_data)
{
    gdk_draw_pixmap(widget->window,
                    widget->style->fg_gc[GTK_WIDGET_STATE (widget)],
                    pixmap_front,
```

```
                        event->area.x, event->area.y,
                        event->area.x, event->area.y,
                        event->area.width, event->area.height);


  return FALSE;
}


gboolean
on_draw_area_Front_button_release_event
                                        (GtkWidget       *widget,
                                        GdkEventButton  *event,
                                        gpointer         user_data)
{
    if (event->button == 2 )
        {
         bandera=FALSE;
         sel=0;
        }


  return FALSE;
}


gboolean
on_draw_area_Front_motion_notify_event (GtkWidget       *widget,
                                        GdkEventMotion  *event,
                                        gpointer         user_data)
{
    int x, y;

       GdkModifierType state;

       if (event->is_hint)
         gdk_window_get_pointer (event->window, &x, &y, &state);
       else
          {
            x = event->x;
            y = event->y;
            state = event->state;
          }

       if (state & GDK_BUTTON2_MASK && pixmap_front != NULL && bandera== TRUE)
    {
          point3D[sel].x=event->x-(widget->allocation.width/2);
        point3D[sel].z=-(event->y-(widget->allocation.height/2));
         Repaint (widget);
    }

       return TRUE;
}



/*************************************************************************
*                                                                        *
*   Funciones de dibujo                                                  *
*                                                                        *
*                                                                        *
*************************************************************************/

/*************************************
*                                    *
*   Colores para los graficos        *
*                                    *
*************************************/


GdkGC *GetPen (GdkColor *c)
{
    GdkGC *gc;

    /* --- Crea un gc --- */
```

```
    gc = gdk_gc_new (pixmap_front);

    /* --- inicializa el color de primer plano --- */
    gdk_gc_set_foreground (gc, c);

    /* --- lo retorna --- */
    return (gc);
}


GdkColor *NewColor (long red, long green, long blue)
{
    /* --- Hacer el color --- */
    GdkColor *c = (GdkColor *) g_malloc (sizeof (GdkColor));

    /* --- llenar los campos --- */
    c->red = red;
    c->green = green;
    c->blue = blue;

    gdk_color_alloc (gdk_colormap_get_system (), c);

    return (c);
}









/****************************************
*                                       *
*   Calculo y pintado de vistas         *
*                                       *
****************************************/

static void Repaint( GtkWidget *widget)
     {
        GtkWidget *front_to_repaint;
        GtkWidget *up_to_repaint;
        GtkWidget *side_to_repaint;
        GtkWidget *persp_to_repaint;

        GdkRectangle update_Draw_area;
    GdkPoint update_point[cont];
    GdkFont *font;

    GdkRectangle punto; //PARA LOS PUNTOS

    int c=0;


     front_to_repaint = lookup_widget(widget, "draw_area_Front");
     up_to_repaint = lookup_widget(widget, "draw_area_Up");
     side_to_repaint = lookup_widget(widget, "draw_area_Side");
     persp_to_repaint = lookup_widget(widget, "draw_area_Persp");

font = gdk_font_load ("-abisource-arial-bold-r-normal-*-*-140-*-*-p-*-iso8859-1");

        //borramos las areas de dibujo

/*** superior ******/
       gdk_draw_rectangle (pixmap_up,
                           penWhite,
                           TRUE,
                           0, 0,
                           up_to_repaint->allocation.width,
                           up_to_repaint->allocation.height);


/*** Frontal ******/
       gdk_draw_rectangle (pixmap_front,
                           penWhite,
                           TRUE,
```

```
                                0, 0,
                                front_to_repaint->allocation.width,
                                front_to_repaint->allocation.height);

        /*** Lateral ******/
               gdk_draw_rectangle (pixmap_side,
                                penWhite,
                                TRUE,
                                0, 0,
                                side_to_repaint->allocation.width,
                                side_to_repaint->allocation.height);


        /*** Perspectiva ******/

               gdk_draw_rectangle (pixmap_persp,
                                penBlack,
                                TRUE,
                                0, 0,
                                side_to_repaint->allocation.width,
                                side_to_repaint->allocation.height);



            //dibujamos los ejes coordenados

        /*** Superior ******/

            gdk_draw_line (pixmap_up,
                                penGreen,
                                up_to_repaint->allocation.width/2, 0,
                                up_to_repaint->allocation.width/2,
                                up_to_repaint->allocation.height);

            gdk_draw_line (pixmap_up,
                                penBlue,
                                0,up_to_repaint->allocation.height/2,
                                up_to_repaint->allocation.width,
                                up_to_repaint->allocation.height/2);

            //etiquetas de los ejes
            gdk_draw_string(pixmap_up,font,penGreen,
                            (up_to_repaint->allocation.width/2)-15,10,"-Y");

            gdk_draw_string(pixmap_up,font,penBlue,
                            (up_to_repaint->allocation.width)-10,
                            up_to_repaint->allocation.height/2,"X");

        /*** Frontal ******/

            gdk_draw_line (pixmap_front,
                                penRed,
                                front_to_repaint->allocation.width/2, 0,
                                front_to_repaint->allocation.width/2,
                                front_to_repaint->allocation.height);

            gdk_draw_line (pixmap_front,
                                penBlue,
                                0,front_to_repaint->allocation.height/2,
                                front_to_repaint->allocation.width,
                                front_to_repaint->allocation.height/2);

            //etiquetas de los ejes
            gdk_draw_string(pixmap_front,font,penRed,
                            (front_to_repaint->allocation.width/2)-10,10,"Z");

            gdk_draw_string(pixmap_front,font,penBlue,
                            (front_to_repaint->allocation.width)-10,
                            front_to_repaint->allocation.height/2,"X");

        /*** Lateral ******/

            gdk_draw_line (pixmap_side,
                                penRed,
                                side_to_repaint->allocation.width/2, 0,
```

```
                        side_to_repaint->allocation.width/2,
                        side_to_repaint->allocation.height);

    gdk_draw_line (pixmap_side,
                        penGreen,
                        0,side_to_repaint->allocation.height/2,
                        side_to_repaint->allocation.width,
                        side_to_repaint->allocation.height/2);

    //etiquetas de los ejes
    gdk_draw_string(pixmap_side,font,penRed,
                    (side_to_repaint->allocation.width/2)-10,10,"Z");

    gdk_draw_string(pixmap_side,font,penGreen,
                    (side_to_repaint->allocation.width)-10,
                    side_to_repaint->allocation.height/2,"Y");

/*** Perspectiva ***/

    gdk_draw_line (pixmap_persp,
                        penRed,
                        persp_to_repaint->allocation.width/2,
                        persp_to_repaint->allocation.height/2,
                        persp_to_repaint->allocation.width/2,
                        persp_to_repaint->allocation.height/2-100);

    gdk_draw_line (pixmap_persp,
                        penBlue,
                        persp_to_repaint->allocation.width/2,
                        persp_to_repaint->allocation.height/2,
                        persp_to_repaint->allocation.width/2+100,
                        persp_to_repaint->allocation.height/2);



    gdk_draw_line (pixmap_persp,
                        penGreen,
                        persp_to_repaint->allocation.width/2,
                        persp_to_repaint->allocation.height/2,
                    persp_to_repaint->allocation.width/2-(80*cos(35*0.017453)),
                    persp_to_repaint->allocation.height/2+(80*sin(35*0.017453)));

/*
*   0.017453 es el equivalente en radianes de un grado.
*   Esta convesión es nescesaria ya que el compilador de
*   "C" sólo maneja radianes para las operaciones
*   trigonométricas.
*/

    //Dibujar lineas y puntos

/*** superior ***/

        for(c=0;c<cont;c++)
      {
        punto.x = point3D[c].x-2 ;
            punto.y = point3D[c].y ;
            punto.width = 4;
            punto.height = 4;
            gdk_draw_rectangle (pixmap_up,
                        penBlack,
                    TRUE,
                    (up_to_repaint->allocation.width/2)+punto.x,
                    (up_to_repaint->allocation.height/2)+punto.y-2,
                    punto.width,
                    punto.height);
        update_point[c].x=punto.x+(up_to_repaint->allocation.width/2)+2;
        update_point[c].y=punto.y+(up_to_repaint->allocation.height/2);
      }

    gdk_draw_lines(pixmap_up,
                    penBlack,
                    update_point,
                    cont);

/*** Frontal ***/
```

```
   for(c=0;c<cont;c++)
     {
       punto.x = point3D[c].x -2;
           punto.y = -(point3D[c].z + 2);
           punto.width = 4;
           punto.height = 4;
           gdk_draw_rectangle (pixmap_front,
                       penBlack,
                       TRUE,
                       (front_to_repaint->allocation.width/2)+
                       punto.x,(front_to_repaint->allocation.height/2)+punto.y,
                       punto.width, punto.height);

       update_point[c].x= punto.x + (front_to_repaint->allocation.width/2)+1;
       update_point[c].y= punto.y + (front_to_repaint->allocation.height/2)+1;

     }

       gdk_draw_lines(pixmap_front,
                       penBlack,
                       update_point,
                       cont);

*** Lateral ***/
   for(c=0;c<cont;c++)
     {
       punto.x = point3D[c].y-2 ;
           punto.y = -(point3D[c].z+2);
           punto.width = 4;
           punto.height = 4;
           gdk_draw_rectangle (pixmap_side,
                           penBlack,
                  TRUE,
                           (side_to_repaint->allocation.width/2)+punto.x,(side_to_repaint->allocat
on.height/2)+punto.y,
                           punto.width, punto.height);


       update_point[c].x=punto.x+(side_to_repaint->allocation.width/2)+1;
       update_point[c].y=punto.y+(side_to_repaint->allocation.height/2)+1;
      }

   gdk_draw_lines(pixmap_side,
                   penBlack,
                   update_point,
                   cont);



** Perspectiva ***/
   for(c=0;c<num_frm;c++)
     {
       punto.x = trj[c].x + (trj[c].y * cos(ANGLE3D)) - 2;
           punto.y = -(trj[c].z + (trj[c].y * sin(ANGLE3D))+ 3);
           punto.width = 4;
           punto.height = 4;


           gdk_draw_rectangle (pixmap_persp,
                           penGreen,
                           TRUE,
                           (side_to_repaint->allocation.width/2)+punto.x-2,
                           (side_to_repaint->allocation.height/2)+punto.y-4,
                           punto.width,
                           punto.height);


   }


   for(c=0;c<cont;c++)
     {
       punto.x = point3D[c].x + (point3D[c].y * cos(ANGLE3D)*0.75) - 2;
       punto.y = -(point3D[c].z + (point3D[c].y * sin(ANGLE3D)*0.75)+ 3);
```

```
        update_point[c].x=punto.x + (persp_to_repaint->allocation.width/2)+2;
        update_point[c].y=punto.y + (persp_to_repaint->allocation.height/2)+2;


      }

gdk_draw_lines(pixmap_persp,
               penWhite,
               update_point,
               cont);



    //forzar expose event
    update_Draw_area.x = 0;
    update_Draw_area.y = 0;
        update_Draw_area.width = front_to_repaint->allocation.width;
        update_Draw_area.height = front_to_repaint->allocation.height;

        //Expose event del area frontal

        gtk_widget_draw (front_to_repaint, &update_Draw_area);

        update_Draw_area.width = up_to_repaint->allocation.width;
        update_Draw_area.height = up_to_repaint->allocation.height;

        //Expose event del area superior

        gtk_widget_draw (up_to_repaint, &update_Draw_area);
        update_Draw_area.width = side_to_repaint->allocation.width;
        update_Draw_area.height = side_to_repaint->allocation.height;

        //Expose event del area lateral


        gtk_widget_draw (side_to_repaint, &update_Draw_area);
        update_Draw_area.width = persp_to_repaint->allocation.width;
        update_Draw_area.height = persp_to_repaint->allocation.height;

        //Expose event del area perspectiva
        gtk_widget_draw (persp_to_repaint, &update_Draw_area);



    }

void
on_button6_clicked                     (GtkButton       *button,
                                        gpointer         user_data)
{
    GtkWidget *entryX,*entryY,*entryZ;
//  gchar *Validate;

    //Obtenemos los widgets
    entryX = lookup_widget(GTK_WIDGET(button), "entry_new_x");
    entryY = lookup_widget(GTK_WIDGET(button), "entry_new_y");
    entryZ = lookup_widget(GTK_WIDGET(button), "entry_new_z");

// validar que se introduce un valor

    //obtenemos el texto de entrada y se asigna a el arreglo convertimos a entero
    point3D[cont].x=atoi(gtk_entry_get_text(GTK_ENTRY(entryX)));
    point3D[cont].y=atoi(gtk_entry_get_text(GTK_ENTRY(entryY)));
    point3D[cont].z=atoi(gtk_entry_get_text(GTK_ENTRY(entryZ)));
    cont++;

    //g_print("En Controles\n X=%d \n Y=%d\n Z=%d ",x,y,z);
    Repaint(GTK_WIDGET(button));
}


void
on_button7_clicked(GtkButton       *button,
                   gpointer         user_data)
{
```

```
    cont=0;
    Repaint(GTK_WIDGET(button));
}


void
on_Btn_Close_clicked (GtkButton        *button,
                      gpointer          user_data)
{
gtk_main_quit();
}


void
on_Btn_New_clicked (GtkButton        *button,
                    gpointer          user_data)
{

}


void
on_Btn_Load_clicked (GtkButton        *button,
                     gpointer          user_data)
{
   GtkWidget *fileselection_open;
//fileselection_open
    fileselection_open = create_fileselection_open ();
    gtk_widget_show (fileselection_open);
}


void
on_Btn_Save_clicked(GtkButton        *button,
                    gpointer          user_data)
{
GtkWidget *file_sel_save;
//fileselection_open
    file_sel_save = create_file_sel_save ();
    gtk_widget_show (file_sel_save);
}




void
on_Btn_About_clicked (GtkButton        *button,
                      gpointer          user_data)
{
  GtkWidget *About;

    About = create_about1 ( );
    gtk_widget_show (About);
}


void
on_Btn_tool_Close_clicked(GtkButton        *button,
                          gpointer          user_data)
{

}

/* Obtrener el nombre del archivo a abrir */
void
on_ok_button_open_clicked(GtkButton        *button,
                          gpointer          user_data)
{
   GtkWidget *ObtainFileName;
   GtkWidget *window;

   FILE *FileRead;
   int i=0;

window= lookup_widget(GTK_WIDGET(button), "fileselection_open");
```

```
    ObtainFileName = lookup_widget (GTK_WIDGET(button),
                                    "fileselection_open");

    fileName_trj = gtk_file_selection_get_filename
                              (GTK_FILE_SELECTION(ObtainFileName));

    FileRead=fopen(fileName_trj,"r");

    /* Obtener el numero de puntos */
    fscanf(FileRead,"%d ",&cont);

    for(i=0;i<cont;i++)
     {
      fscanf(FileRead,"%d ",&point3D[i].x);
          fscanf(FileRead,"%d ",&point3D[i].y);
          fscanf(FileRead,"%d ",&point3D[i].z);
     }


        fclose(FileRead);

    //Repaint(window);

        gtk_widget_destroy (window);


}


void
on_cancel_button1_clicked (GtkButton       *button,
                           gpointer         user_data)
{
 GtkWidget *window;

    window= lookup_widget(GTK_WIDGET(button), "fileselection_open");
    gtk_widget_destroy (window);
}


void
on_Btn_cal_trj_clicked (GtkButton       *button,
                        gpointer         user_data)
{


    GtkWidget *entry_frm;

    //Obtenemos los widgets

     gint j=0;
     gfloat longit=0.0,tmpx=0.0, tam_seg=0.0,comp=0.0,
     gfloat tmp=0,interm=0.0,pasada=1.0;
     gfloat k=0.0;
     type3D aux;
     gint r=0,index_trj=0;

    entry_frm = lookup_widget(GTK_WIDGET(button), "num_frames");

    // validar que se introduce un valor
    //obtenemos el texto de entrada y se asigna a el arreglo
    //convertimos a entero

    num_frm=atoi(gtk_entry_get_text(GTK_ENTRY(entry_frm)));
    if(num_frm < 1)num_frm=2;



    g_print("En Calcular  num_frm=%d \n",num_frm);

/* Clucular longitud de la trajectoria */

    for(j=0;j<cont-1;j++)
    {
```

```
      longit=longit+sqrt(
                        pow(point3D[j+1].x-point3D[j].x , 2 ) +
                        pow(point3D[j+1].y-point3D[j].y , 2 ) +
                        pow(point3D[j+1].z-point3D[j].z , 2 ) );

       }


      tam_seg=longit/num_frm;


/* Calcular puntos intermedios de la trajectoria */

for(j=0;j<cont-1;j++)
{
          aux.x=point3D[j].x;
     aux.y=point3D[j].y;
     aux.z=point3D[j].z;

     tmp=sqrt(pow(aux.x-point3D[j+1].x , 2 ) +
              pow(aux.y-point3D[j+1].y , 2 ) +
              pow(aux.z-point3D[j+1].z , 2 ) );
     g_print("tmp=%f\n",tmp);

tmpx=tmp;
     while(tam_seg <= tmpx)
     {
     //coef=(tmp-tam_seg*pasada)/(tam_seg*pasada);

     k=tam_seg*pasada;
     k=tmp-k;
     k=(tam_seg*pasada)/k;

g_print("coef=%f\n",k);
     k=k+1.0;

g_print("coeff=%f\n",k);

      trj[index_trj].x=(aux.x+k*point3D[j+1].x)/k;
      trj[index_trj].y=(aux.y+k*point3D[j+1].y)/k;
      trj[index_trj].z=(aux.z+k*point3D[j+1].z)/k;

          index_trj++;
          pasada=pasada+1.0;

g_print("xf=%d\n",trj[index_trj-1].x);
g_print("yf=%d\n",trj[index_trj-1].y);
g_print("zf=%d\n",trj[index_trj-1].z);

      tmpx=tmpx-tam_seg;
          g_print("tmpx=%f\n\n",tmpx);

     }
     pasada=1.0;
}



g_print("longitud=%f\n",longit);

g_print("Tam segmento=%f\n",tam_seg);

trj[index_trj].x=point3D[cont].x;
trj[index_trj].y=point3D[cont].y;
trj[index_trj].z=point3D[cont].z;

}


void
on_ok_button1_clicked(GtkButton       *button,
                      gpointer         user_data)
{
GtkWidget *ObtainFileName;
```

```
    GtkWidget *window;

    FILE *FileSave;
    int i=0;

window= lookup_widget(GTK_WIDGET(button), "file_sel_save");


     ObtainFileName = lookup_widget (GTK_WIDGET(button), "file_sel_save");
     fileName_trj = gtk_file_selection_get_filename
                    (GTK_FILE_SELECTION(ObtainFileName));
     FileSave=fopen(fileName_trj,"w+");

g_print("Ruta del archivo:%s",fileName_trj);

     /* Obtener el numero de puntos */

     fprintf(FileSave,"%d \n",cont);
     for(i=0;i<cont;i++)
      {
       fprintf(FileSave, "%d %d %d\n", point3D[i].x ,point3D[i].y,point3D[i].z);


      }


         fclose(FileSave);

         gtk_widget_destroy (window);
}


void
on_cancel_button2_clicked (GtkButton       *button,
                           gpointer         user_data)
{
GtkWidget *window;

     window= lookup_widget(GTK_WIDGET(button), "fileselection_save");
     gtk_widget_destroy (window);
}
```

BIBLIOGRAFÍA

[1] Donald Hearn y M. Pauline Beker, Graficas por computadora, segunda edición, Prentice Hall Interamericana ,1995.

[2] Darío Pescador Albiach, 3D Studio,guía práctica para usuarios, 1ª edición, Ediciones Anaya Multimedia, 1999.

[3] A. Cordero Luis, Fernández Mariza, Gray Alfredo, Geometría diferencial de curvas y superficies con matemática , segunda edición, Addison wesley iberoamericana,1995.

[4] Zorrilla Arena, Santiago, Introducción a la metodología de la Investigación, Octava Edición. Editorial Océano. México, 1992.

[5] Johnson, R. E. & F. L. Kiokemeister, Calculus with Analytic Geometry, 5th ed., Allyn & Bacon, 1974.

[6] Harlow Eric, Desarollo de aplicaciones Linux con GTK+ y GDK,1ª edición, Prentice Hall Iberia, Madrid 1999.

[7] Manuel Escribano Cauqui. Programación de gráficos en 3D, Addison-Wesley Iberoamericana,S.A. Madrid-España 1995.

[8] Zózimo Menna Goncalves, Geometría analítica del espacio, enfoque vectorial, Editorial Limusa, 1987.

[9] Weiskamp, Keith, Graficas poderosas con turbo c++, *Megabyte Mexico, 1994*

[10] Aaron m. tenenbaum, yedidyah langsam, moshe j. Augenstein, Data structures using c, Prentice Hall, c1990

[11] Marsden, Jerrold E. y Tromba, Anthony J , Cálculo Vectorial, Addison Wesley Iberoamericana, 3a. edición E.U.A., 1991

REFERENCIAS A PAGINAS DE INTERNET

[R1] http://www.stmuc.com/moray

Programa modelador en alambre para Pov-Ray

[R2] http://www.pov-ray.com

Software para la creación de gráficos tridimensionales de alta calidad.

[R3] http://www.gtk.org

librerias "C" para la creación de aplicaciones graficas en linux.

[R4] http://www.gnu.org

Página del proyecto GNU

[R5] http://www.opensource.org/licenses/gpl-license.php

Página de la licencia publica general.

[R6] http://www.mesa3d.org/

Implementación de las librerías gráficas de openGL.

[R7] http://www.opengl.org/

Ambiente para la creación interactiva y portable de aplicaciones en 2D y 3D

[R8] http://www.gnu.org/copyleft/gpl.html

[R9] http://tigre.aragon.unam.mx/m3d/

Pagina del proyecto material 3D

[R10] http://www.aliaswavefront.com/maya

Modelador avanzado de graficas en 3D.