Facultad de Informática
Universidad Politécnica de Valencia

Desarrollo, implementación y prueba de un algoritmo de reconstrucción de objetos a partir de una representación axonométrica, utilizando técnicas de optimización

Proyecto Final de Carrera presentado por:

D. Juan Vicente Andreu Hernández

Director:

Dr. D. Pedro Company Calleja

Catedrático de Expresión Gráfica en la Ingeniería

Departamento de Tecnología

Universitat Jaume I de Castellón

Tutor:

D. Antonio J. Sánchez Salmerón

Profesor Titular de Escuela Universitaria

Departamento de Ingeniería de Sistemas y Automática

Universidad Politécnica de Valencia

Valencia, Julio de 2000

ÍNDICE

ÍNDICE DE FIGURAS

ÍNDICE DE CÓDIGO FUENTE

Juan Vicente Andreu Hernández

Facultad de Informática

Universidad Politécnica de Valencia

Julio de 2000

1. RECONSTRUCCIÓN GEOMÉTRICA

1.1 Introducción

La comunicación entre los diseñadores y los sistemas de diseño asistido por ordenador (CAD) está desequilibrada a causa de las necesidades de programación. En su estado actual, los sistemas CAD fuerzan al diseñador a controlar el proceso de diseño asistido por medio de un flujo secuencial; que parte de las especificaciones y llega al diseño detallado. Desgraciadamente, dicha forma de proceder puede incidir muy negativamente en la creatividad del diseñador.

La causa de la estructura secuencial de los sistemas CAD es la naturaleza secuencial de los lenguajes algorítmicos de programación (y se debe resaltar que las interfaces gráficas de usuario –GUI's- no son ninguna excepción a esta regla). La necesidad de que los programadores definan un modelo del proceso de diseño (o del proceso de "modelado para diseño de productos") aumenta la tendencia a la secuenciación, porque, para los programadores, definir modelos *conceptuales* "lo que el sistema debe hacer", tan parecidos como sea posible a los modelos de *implementación* "la forma en la que el sistema debe hacerlo" es siempre la solución más simple. El resultado es que los sistemas CAD están constantemente pidiendo al usuario que especifique *acciones* (tareas secuenciales completamente especificadas). Y esta es una mala estrategia para la fase de síntesis, cuando el diseñador está tratando de fijar "visiones", es decir, ideas poco definidas y desordenadas.

Los comandos *transparentes* (esas órdenes que pueden ser invocadas en cualquier momento durante la ejecución de un programa interactivo guiado por comandos), pueden dar la falsa impresión de que el usuario puede hacer casi todo y en cualquier momento. De hecho, los sistemas CADD (los sistemas de *dibujo* asistido) son muy interactivos, porque imponen pocas limitaciones incluso a los usuarios más erráticos. Pero se debe recordar que esto es debido a que están basados en Geometría Descriptiva/Constructiva y en Dibujos Normalizados de Ingeniería, que son dos disciplinas no secuenciales. O, mejor dicho, dos disciplinas basadas en la utilización de dos aspectos de un *lenguaje gráfico* (no secuencial). Por tanto, lo que el usuario hace es

utilizar un ordenador para construir una representación gráfica que, una vez acabada, será un documento que contendrá información interpretable sólo por el ser humano. Es decir, que el ordenador será incapaz de interpretar la información contenida en su propia base de datos para utilizarla como información útil para automatizar otras tareas de diseño. Y conviene matizar que el ordenador es incapaz de interpretar la información cuando esta está aun incompleta (lo cual sería muy deseable para el diseñador, dado que de este modo el ordenador podría ayudarle realmente a concretar y fijar sus ideas en la fase de síntesis) pero también es incapaz de interpretar representaciones completas, que cualquier humano con formación técnica puede entender de forma unívoca.

Además, desde el punto de vista del diseñador, en los sistemas CAD de modelado tridimensional (los sistemas de *diseño* asistido) la situación también es precaria. Con los sistemas CAD 3D se pueden crear modelos virtuales tridimensionales, los cuales pueden ser posteriormente mostrados por medio de imágenes tan realistas como se desee y/o por medio de planos normalizados obtenidos de forma casi automática. Los modelos pueden ser utilizados incluso como información de entrada para todo tipo de análisis y evaluaciones del diseño (estudio de la compatibilidad geométrica, de la resistencia mecánica, del comportamiento térmico, etc.). Pero la construcción de dichos modelos es puramente secuencial. Se ejecuta una única acción tras cada comando, y el sistema vuelve al estado *neutro* (modo de espera para recibir y ejecutar una nueva orden). Además los modelos conceptuales utilizados para construir estos modelos están más próximos a los modelos internos del sistema CAD que a la forma de pensar del diseñador (modelos BRep, CSG, etc.).

En definitiva, no se cuenta con el soporte de un lenguaje que sirva para canalizar las ideas poco definidas y mal estructuradas que el diseñador tiene en las fases iniciales de especificación y síntesis de un nuevo diseño. Esta situación es opuesta a la que se da cuando los diseñadores realizan la fase de síntesis mediante herramientas gráficas "clásicas" (Bocetos, Geometría Descriptiva y Dibujo Normalizado). Entonces, se utiliza un lenguaje gráfico, no secuencial, siendo precisamente aquellas características derivadas de las posibilidades de dicho lenguaje, como la sobrexposición de ideas, los solapes visuales (sobremuestras) y, en definitiva, la coexistencia de diferentes

soluciones en un mismo marco de trabajo, las que hacen del mismo un vehículo muy apreciable para canalizar la creatividad del diseñador.

Resumiendo, el problema estriba en que el pensamiento gráfico (en el sentido de "no verbal") no puede expresarse cómodamente a través de un lenguaje verbal. Definiendo el término verbal como sinónimo de secuencial. Es decir, entendiendo que los lenguajes verbales están basados en la variación de un conjunto de signos a lo largo del tiempo, sin importar que los signos sean símbolos gráficos o sonidos. Por el contrario, los lenguajes no verbales o visuales, y dentro de éstos los "gráficos", son aquellos que basan la transmisión de la información, no sólo en el significado de un conjunto de símbolos gráficos predefinidos, sino también en las relaciones espaciales entre dichos símbolos. Es decir, que las relaciones de semejanza, orden, proporcionalidad y vecindad entre los símbolos tienen una importancia capital que hace inviable expresar oralmente una comunicación gráfica.

Por tanto, los potentes postprocesadores gráficos disponibles en la actualidad para tareas de diseño asistido, deben complementarse con los correspondientes preprocesadores. Entonces, los postprocesadores se concentrarán en su verdadera tarea de ayudar al diseñador en la manipulación de los modelos virtuales 3D, *después* de que dichos modelos hayan sido creados con la ayuda de preprocesadores capaces de leer la información expresada por el diseñador en el lenguaje gráfico que a él le resulta cómodo emplear (en lugar del lenguaje secuencial, que resulta más cómodo al sistema).

Dada la complejidad del lenguaje gráfico empleado en diseño, el objetivo explicado arriba es casi utópico. Se trata, ni más ni menos que de dotar a un sistema informático de la capacidad de "leer" dibujos de ingeniería. Podemos detallar el problema diciendo que necesitamos capacidad para:

1. Convertir un boceto (dibujo preliminar, con una descripción incompleta de la geometría) en una figura "vectorial" plana (figura bidimensional descrita por un conjunto de primitivas 2D). Se trata de obtener una base de datos que describa la figura bocetada en términos de elementos geométricos contenidos en el plano de la figura ("primitivas 2D"). Es importante recordar la distinción entre figuras *matriciales* (en donde la

imagen completa se descompone en una nube de puntos situados según una retícula de más o menos densidad) y las figuras *vectoriales* en las que la imagen se descompone en elementos geométricos más complejos (tales como segmentos de recta, arcos de circunferencia, etc.). La importancia de la distinción estriba en que en la actualidad es posible convertir imágenes en figuras matriciales de forma automática ("escaneando"), pero no es posible obtener una figura vectorial a partir de una imagen en papel.

2. Reconstruir un modelo geométrico tridimensional a partir de una o más figuras vectoriales. Se trata de recuperar la información sobre la geometría tridimensional de los objetos que está implícita o explícitamente contenida en las distintas representaciones bidimensionales de los mismos.

3. Añadir a la base de datos del modelo 3D generado en la reconstrucción, toda la información complementaria a su propia forma geométrica. Es decir, la información que está expresada por medio de los símbolos normalizados habituales en los dibujos de ingeniería (como tolerancias, procesos de fabricación, etc.).

Cada una de las tres tareas apuntadas es suficientemente compleja, por lo que la consideración simultanea de todas ellas está de momento fuera del alcance de desarrollos prácticos. Además, el problema es aun más completo, si se tiene en cuenta que lo ideal sería que la capacidad de "lectura" fuera interactiva. Es decir, que la información debería ser leída e incorporada al modelo tal como el diseñador la fuera introduciendo, de forma que se generasen modelos "provisionales" que permitiesen al diseñador ir comprobando algunos aspectos de la validez del diseño antes de tenerlo completamente especificado [1].

En este trabajo presentaremos la situación actual en la segunda de las tareas indicadas (la reconstrucción de los modelos geométricos) y nuestras propias aportaciones en dicha línea. En particular, resumiremos los resultados alcanzados en la reconstrucción a partir de varias vistas [2], y el método implementado por nosotros para obtener, de forma casi automática, un modelo 3D de un objeto poliédrico, a partir de una axonometría del mismo [3].

1.2 Ámbito de la reconstrucción geométrica

La descripción de objetos tridimensionales en un plano, utilizando proyecciones bidimensionales, se remonta a más de dos mil años. Fue Monge el primero que sistematizó y simplificó los métodos existentes, dando lugar al nacimiento de la *Geometría Descriptiva*. El problema contrario de cómo reconstruir automáticamente la estructura de un objeto tridimensional (estructura geométrica y topológica) a partir de su proyección, empezó a atraer la atención sólo a finales de los 60, motivado por el desarrollo de los ordenadores digitales.

La *reconstrucción*, implica determinar la relación geométrica y topológica de las partes atómicas de un objeto. No debe confundirse con el *reconocimiento* o *restitución*, que se usa en visión artificial y que implica la identificación de un objeto mediante algún sistema de acoplamiento de plantillas. De modo simple, se puede decir que la *restitución* sirve para determinar *qué* objeto hay en la imagen, mientras que la reconstrucción tiene por objetivo determinar *cómo* es ese objeto.

Las clasificaciones existentes de los métodos de reconstrucción tienden a destacar los aspectos más críticos del problema. Así, algunos autores basan sus clasificaciones en aspectos tales como la representación interna usada en el proceso de reconstrucción, la naturaleza de los objetos reconstruidos, el número de vistas 2D requeridas y las premisas y el grado de interacción necesario para obtener la reconstrucción.

1.2.1 Naturaleza de los objetos y los modelos

Los primeros intentos de interpretación de dibujos 2D se limitaron a modelos predefinidos que sólo pretendían identificar objetos cuyas formas habían sido previamente definidas. Es decir, que estaban más cercanos a la restitución que a la reconstrucción.

Posteriormente se obtuvo una solución general para la reconstrucción de objetos poliédricos. Aunque, algunos métodos necesitaban distinguir entre poliedros eulerianos y no eulerianos. Además, la complejidad del poliedro limitaba el alcance de algunos métodos. La complejidad se mide en función de la "valencia" de sus vértices, es decir, el número de aristas que confluyen en un vértice. Para ilustrar la facilidad con que un

objeto poliédrico puede tener vértices con una valencia elevada basta observar la Figura 1-1.

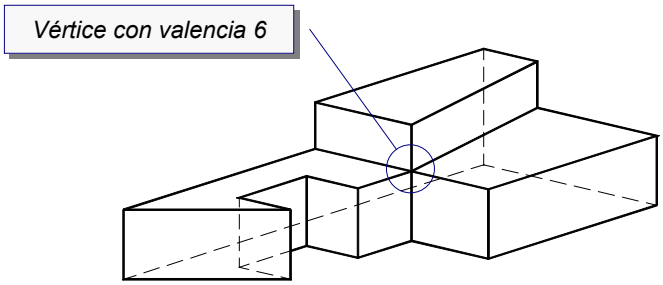


Figura 1-1. Objeto poliédrico con un vértice de valencia 6.

Otros intentos se encaminaron hacia la reconstrucción de objetos de revolución y objetos extruidos (dos casos especiales de la geometría de "barrido" tan típica de los sistemas de modelado 3D). Inicialmente se imponían restricciones muy grandes en cuanto a formas y orientaciones. Desarrollos posteriores han suavizado y/o eliminado parte de dichas restricciones.

Actualmente se pueden reconstruir objetos pertenecientes a un dominio bastante amplio. Sin embargo, los procesos de reconstrucción aumentan su porcentaje de fallos cuando los objetos implicados incluyen partes poliédricas complejas y/o un número creciente de superficies curvas.

En cuanto al modelo generado tras el proceso de reconstrucción, las representaciones utilizadas son tanto la Geometría Constructiva de Sólidos (CSG) como la Representación de Fronteras (BRep), aunque la mayor parte de los trabajos de reconstrucción usan BRep.

1.2.2 Premisas y grado de interacción

Para simplificar el problema de la reconstrucción, generalmente se asume que en cualquier proyección de una *escena* (un volumen tridimensional, generalmente cúbico, que contiene uno o varios modelos geométricos) sólo se representan las aristas y los contornos. Es decir, se trabaja con vistas *normalizadas* (en el sentido de que los objetos se representan por medio de aristas y contornos). No se consideran la textura, la gama, el sombreado y otros parámetros adicionales que sí que se están usando en el reconocimiento de objetos. Si que cabe matizar que algunos sistemas distinguen entre

aristas vistas y ocultas; otros no distinguen, y los hay que explícitamente exigen que no se empleen dichas aristas ocultas.

Se suele añadir una limitación adicional sobre la dirección de la proyección en proyecciones paralelas y sobre el centro de la proyección en proyecciones perspectivas. En las proyecciones paralelas, la dirección de la proyección no debe ser paralela a ninguna cara, ni paralela a ningún par de aristas no colineales. En las proyecciones perspectivas, el centro de la proyección no debe ser coplanar con ninguna cara ni coplanar con ningún par de aristas no colineales. A esta limitación se la denomina "Supuesto de Punto de Vista General" y generalmente elimina casos de degeneración potenciales en los que, por ejemplo, una cara se puede proyectar como una línea, o dos aristas distintas se pueden proyectar sobre una misma línea (ver Sugihara [4]). La reconstrucción a partir de una única proyección no tiene mucho sentido sin este supuesto. En la Figura 1-2 se muestran dos axonometrías distintas del mismo objeto poliédrico. Se puede observar que la representación isométrica es más difícil de "leer" que la dimétrica, porque no respeta la convención del punto de vista general.

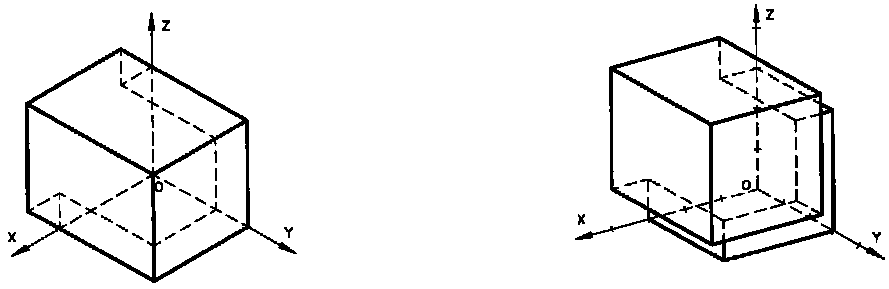


Figura 1-2. Proyecciones de un objeto poliédrico sin respetar (izquierda) y respetando (derecha) la convención del punto de vista general.

Los sistemas existentes también se pueden clasificar en función de la colaboración que requieren por parte del usuario. La distinción básica es entre sistemas automáticos y sistemas guiados. Aunque algunos sistemas guiados requieren tanta participación del usuario que son sistemas de modelado inteligentes, más que sistemas de reconstrucción.

1.2.3 Número de vistas y número de soluciones

Por el número de vistas requeridas al dibujo de entrada, existen dos grandes categorías: métodos de vistas múltiples y métodos de vista única.

Los métodos basados en vistas múltiples están más adelantados. Obviamente, en general, es mucho más fácil reconstruir objetos 3D a partir de proyecciones de vistas múltiples que a partir de proyecciones de vista única. No obstante dichos métodos suelen estar limitados a considerar las vistas en alzado, planta y perfil, y no aceptan convencionalismos normalizados (cortes, vistas particulares, etc.).

La reconstrucción a partir de una única vista presenta más ambigüedades.

En cuanto al número de soluciones, es importante destacar que, en algunos casos, la figura de partida puede corresponder a más de un objeto. Un método se llama de "solución múltiple" si encuentra todos los objetos que puedan corresponderse con la(s) vista(s) 2D, y de "solución única" si se para tras encontrar un primer objeto válido.

1.3 El estado del arte de la reconstrucción geométrica

1.3.1 Introducción

Los primeros estudios sobre este tema se pueden encontrar resumidos en unas pocas referencias [4-10]. El libro de Sugihara [4] es la referencia más completa a la historia inicial de la interpretación automática de dibujos técnicos. Najendra y Gujar [5] publicaron un resumen de varios artículos que trataban la reconstrucción de objetos tridimensionales a partir de sus vistas 2D. Wang y Grinstein [6] completaron el trabajo, realizando una taxonomía de la reconstrucción de objetos 3D a partir de dibujos lineales de proyección bidimensional. La referencia a Yan y otros [7] se debe a que estos autores completaron y sistematizaron uno de los métodos más desarrollados y efectivos hasta la fecha, para reconstrucción de poliedros a partir de vistas múltiples. Por último, las referencias [8-10] trazan los orígenes del prometedor método de reconstrucción por optimización de regularidades.

1.3.2 Análisis de los métodos de vistas múltiples

Los modelos obtenidos tras el proceso de reconstrucción son generalmente de tipo "Geometría Constructiva de Sólidos" (CSG) o "Representación de Fronteras" (BRep). Actualmente, la mayor parte de los trabajos de reconstrucción totalmente automática

usan BRep, mientras que las representaciones CSG son más habituales en los procesos de reconstrucción guiada.

Se han llevado a cabo distintos intentos para obtener modelos CSG a partir de vistas múltiples. Todos los métodos de este tipo tienen en común que asumen que el objeto 3D puede ser construido a partir de cierto conjunto de primitivas combinadas siguiendo una cierta jerarquía. Los métodos difieren en las estrategias que proponen para "extraer" dichas primitivas de las vistas de partida. Pero, hasta la fecha, en todos ellos se requiere mucha participación del usuario. Por tanto, se trata de métodos de construcción asistida, más que de reconstrucción automática.

Respecto a los métodos orientados hacia la consecución de un modelo BRep, el trabajo de Yan y otros [7], contiene una completa y detallada descripción del método de reconstrucción de poliedros más efectivo hasta la fecha. Dicho método sigue la secuencia de pasos que resumimos a continuación:

1. Generación de vértices "candidatos", a partir de los nodos 2D.

2. Generación de aristas "candidatas", a partir de los vértices candidatos y los segmentos lineales de la figura de partida.

3. Construcción de caras, a partir de las aristas. (Primero se construyen los "bucles de cara", candidatos a describir caras en el modelo 3D, y después se filtran dichos bucles).

4. Se forma el modelo BRep, a partir de los bucles de cara.

La secuencia descrita resume con bastante aproximación la mayoría de los métodos basados en "ensamblar" una representación BRep del objeto a partir de una colección de elementos geométricos, los cuales deben identificarse previamente en las figuras de partida. Las diferentes propuestas se centran en aumentar la efectividad de los algoritmos y en tratar los casos patológicos.

1.3.3 Análisis de los métodos de vista única

En el ámbito de los métodos de vista única, las primeras aproximaciones fueron los "métodos de etiquetado", que se basan en definir "circuitos" con todos los segmentos de la figura 2D que son candidatos a corresponder con aristas del modelo 3D. Los segmentos que forman cada circuito son clasificados (o "etiquetados") con unos códigos (signos + o -) que posteriormente sirven para decidir si los diferentes circuitos pueden corresponder con caras del modelo 3D (e incluso si la figura corresponde a la representación de algún modelo válido).

Los métodos de etiquetado son métodos de interpretación más que de reconstrucción: ofrecen sólo las condiciones necesarias para que un dibujo lineal 2D represente un sólido 3D válido. Además, un dibujo lineal que se puede etiquetar adecuadamente no necesariamente representa un sólido 3D verdadero.

Los métodos de *programación lineal* fueron los primeros métodos con una aplicación práctica en reconstrucción. Se basan en plantear y resolver las ecuaciones lineales que describen las condiciones que un poliedro debe satisfacer. Sugihara [4] consiguió dar una condición necesaria y suficiente que permitía que un dibujo lineal representase un objeto poliédrico en términos del problema de programación lineal. Su formulación permite resolver el problema de discriminar entre dibujos lineales correctos e incorrectos. Sin embargo, el método de Sugihara no es práctico para realizar la reconstrucción. La condición es tan precisa matemáticamente que algunos dibujos son rechazados simplemente porque los vértices se desvían ligeramente de las posiciones correctas. Además, la implementación del método requiere la solución de un gran problema de programación lineal de 3m+n variables, donde m es el número de caras visibles del poliedro y n el número de vértices visibles.

Frente a la rigidez de la programación lineal, se propuso el *método perceptual*. En este método, el algoritmo toma como dato un dibujo axonométrico, y utiliza los invariantes que aseguran que las líneas paralelas aparecen paralelas, y que las aristas paralelas a los ejes principales se dibujan con longitudes proporcionales a las dimensiones reales. El objetivo del algoritmo es asignar coordenadas 3D a todos los

vértices del grafo. El algolritmo preprocesa el dibujo generando un gráfico de adyacencia (un mapa de vértices y aristas) y después etiqueta el dibujo.

Este modelo tiene limitaciones en el uso de las reglas de percepción heurísticas. Estas son ciertas en muchos casos, pero no siempre. Por ejemplo, este método corregirá líneas con una ligera inclinación y las hará horizontales o verticales, incluso aunque esta inclinación sea intencionada. Otro problema surge de la determinación precisa de sí dos planos son paralelos, o si dos objetos son simétricos. A menos que exista algún supuesto sobre el mundo del objeto (como la rectitud), estas propiedades no se pueden determinar antes de que se asignen las coordenadas. Así pues, este modelo no ofrece una reconstrucción exacta del objeto 3D.

En el campo de la reconstrucción de una representación CSG *(*método de *identificación de primitivas*), se ha avanzado poco hasta fechas recientes. Ya se ha dicho que la construcción de una representación CSG a partir de vistas múltiples todavía depende mucho de la interacción con el usuario. En la reconstrucción automática de la representación CSG a partir de un sólo dibujo la aportación más importante es la de Wang y Grinstein [11], quienes presentaron un algoritmo para extraer automáticamente los bloques primitivos de un poliedro. Posteriormente, Wang amplió el algoritmo para abarcar objetos semi-rectilineos (los que tienen una dirección axial y cualquier superficie cuya normal no sea perpendicular al eje, es ella misma perpendicular al eje) y algunos objetos curvilineos. Wang también introdujo los cilindros como primitivas, y dio una solución completa para encontrar la representación CSG de un dibujo lineal de un poliedro general con el tetraedro como nueva forma primitiva.

El último método introducido es el método de *optimización*, donde la aportación más significativa es la de Lipson y Shpitalni [10]. Su método de reconstrucción por "inflado" de una representación plana se basa en definir como variables las coordenadas Z de todos los vértices definidos por sus coordenadas (X,Y) en la representación plana. La función objetivo es una formulación matemática de las "regularidades" observables en el dibujo 2D. Los autores distinguen tres tipos de regularidades: (a) las que reflejan alguna relación espacial entre entidades individuales (por ejemplo el paralelismo), (b) las que reflejan alguna relación espacial entre un grupo de entidades (por ejemplo una

simetría oblicua en las aristas que definen el contorno de una cara) y (c) las regularidades que afectan a todo el dibujo (como la proporcionalidad entre las longitudes del dibujo y las longitudes reales). Su método tolera las imperfecciones y permite reconstruir una gran variedad de objetos, incluyendo caras planas y cilíndricas. Pero el porcentaje de fallos aumenta al considerar objetos con superficies curvas.

2. OBJETIVOS DEL PROYECTO FINAL DE CARRERA

El presente Proyecto Final de Carrera (PFC) forma parte de las actividades del Grupo REGEO. Por tanto, los objetivos del PFC se enmarcan en los objetivos del grupo.

2.1 Grupo REGEO

REGEO es un grupo de investigación integrado por gente de diferentes Universidades Españolas: Universitat Jaume I, Universidad Politécnica de Valencia y la Universidad Politécnica de Cartagena.

El grupo tiene su sede en el Departamento de Tecnología, localizado en el edificio TC del Área de Tecnología y Ciencias Experimentales en el Campus de Riu Sec de la Universitat Jaume I.

Hay disponible una página Web sobre las actividades del grupo, sus miembros y demás temas de interés de acceso público (*ver 9.1. Direcciones Internet*).

2.2 Objetivos del Grupo REGEO

El **objetivo general** del grupo es conseguir una aplicación automática (o al menos, fácil de usar) para "reconstruir modelos geométricos", es decir, para generar automáticamente modelos virtuales 3D, partiendo de dibujos a mano alzada en 2D.

La Reconstrucción Geométrica es una pequeña parte del campo de la Visión Artificial, o Visión por Computador. En la Reconstrucción Geométrica la información de entrada son figuras geométricas en 2D y un conjunto de símbolos normalizados empleados en los planos de ingeniería, y la información de salida son modelos geométricos 3D como los utilizados por las aplicaciones CAD/CAM/CAE.

La reconstrucción no es tan sólo un proceso numérico basado en reglas geométricas. La información geométrica contenida en las figuras suele ser incompleta. Razón por la que las figuras 2D deben ser “percibidas” por el ordenador (en el sentido psicológico del término “percepción”). Además, la posible existencia de símbolos

normalizados (tales como condiciones geométricas y dimensiones) obliga a "interpretar" ese lenguaje e incorporar la información en el correspondiente modelo.

Por tanto, las palabras clave que describen el objetivo general del Grupo REGEO son: Reconstrucción geométrica, Modelado 3D automático, Percepción automática. Lenguaje gráfico. Y la clasificación UNESCO de su campo de trabajo es: 1203.9.

El objetivo general se concibe como un paso necesario para alcanzar el objetivo más ambicioso de generar un lenguaje gráfico de comunicación entre el ser humano y el ordenador. En concreto, el interés se centra en la comunicación entre el diseñador/proyectista y las aplicaciones CAD. En otras palabras, la generación de modelos virtuales partiendo de dibujos estandarizados es el camino más eficiente para establecer una comunicación entre los diseñadores y las aplicaciones CAD. Y, en este desafío, la Reconstrucción 3D de modelos geométricos obtenidos a partir de dibujos de ingeniería es un aspecto fundamental; al igual que la vectorización, la interpretación de símbolos y otros aspectos relacionados con el tema.

El Objetivo general propuesto arriba se ha desarrollado en dos objetivos programáticos, a saber:

- Integrar las fases iniciales de diseño en el entorno de los sistemas CAD/CAM/CAE

- Llegar a la reconstrucción automática de planos de ingeniería, rescatando la información acumulada en planos en las oficinas de ingeniería

A su vez, estos dos objetivos programáticos se han desarrollado en los siguientes **objetivos específicos**:

1. Desarrollar un algoritmo de reconstrucción a partir de una única vista e integrar en él el resto de algoritmos desarrollados (algoritmo de reconstrucción a partir de vistas múltiples y uno de los módulos de vectorización). Dicho entorno deberá ser el marco de los sucesivos trabajos que se emprendan.

2. Mejorar el algoritmo de reconstrucción a partir de una única vista. Por un lado deberá de prever la incorporación de datos a partir de la digitalización de bocetos que sucesivamente deberán ser vectorizados y reconstruidos en 2D. Por otro lado las prestaciones del actual algoritmo deberán ampliarse dando entrada a objetos con formas cuadráticas.

3. Mejorar el algoritmo de reconstrucción a partir de vistas múltiples a partir del que actualmente se encuentra desarrollado. Incrementar las prestaciones del actual algoritmo dando entrada a objetos con formas cuadráticas.

4. Integrar la vectorización de planos en la reconstrucción de modelos a partir de sus vistas principales.

5. Explorar la consideración de la reconstrucción de toda la información contenida en los planos de ingeniería en forma de normas y convencionalismos. Es decir, diseñar un algoritmo capaz de reconstruir a partir de un conjunto variable de vistas, que pueden incluir diferentes combinaciones de vistas principales y auxiliares, además de vistas cortadas y convencionalismos.

6. Incorporar la información dimensional contenida en la acotación, y la información de fabricación contenida en todo tipo de símbolos normalizados (tolerancias, acabados superficiales, procedimientos de fabricación, etc.) en el proceso de reconstrucción a partir de vistas múltiples.

7. Revisar la normalización de las representaciones gráficas actualmente en uso, para eliminar las ambigüedades, redundancias e indefiniciones, que un técnico entrenado es capaz de detectar y corregir, pero que resultaría mucho más complicado conseguir que un sistema de reconstrucción adquiriese dicho grado de "experiencia" y capacidad de decisión. Tal línea de trabajo redundaría en una simplificación de la tarea de reconstrucción, y, lo que es más importante, en una mejora del lenguaje gráfico normalizado. Se obtendría una "normalización de la normalización".

8. En el método de vista única, conseguir una herramienta que permita al diseñador, una vez haya plasmado sus ideas en bocetos "virtuales" (trabajando directamente sobre el sistema CAD), pedir una "solidificación" de los mismos. Ello permitiría

comprobar la "bondad geométrica" y/o otro tipo de requisitos funcionales del diseño (por ejemplo cualquier tipo de análisis resistente, térmico, etc.). De este modo, el sistema permitiría actuar sobre el boceto y/o sobre el modelo 3D, en refinamientos posteriores, hasta fijar el diseño definitivo.

También en el método de vista única, integrar todo el proceso de diseño en un mismo entorno, permitiendo así un tratamiento simultáneo de todos los aspectos del proceso iterativo del diseño. Y eliminando cualquier tipo de "exportación" de la información generada durante dicho proceso, ya que dicha información estaría totalmente integrada en la base de datos del sistema. A pesar de los importantes avances en CAD, los diseñadores todavía prefieren el lápiz y el papel. Especialmente en las fases más conceptuales del diseño, en las que se baraja una colección incompleta de requisitos e ideas abstractas sobre lo que el producto diseñado deberá ser. Sin embargo la integración del trabajo de estas fases en los actuales sistemas integrados de diseño y fabricación es del todo deseable tanto desde el punto de vista productivo como desde el equipo humano que ha de llevar a cabo las tareas en su conjunto. Conseguir dicha integración constituye la finalidad básica de la reconstrucción 3D a partir de representaciones mediante vista única plasmadas mediante perspectivas o bocetos.

2.3 Objetivo del PFC

El objetivo de este Proyecto Final de Carrera es desarrollar un algoritmo para reconstruir modelos 3D de objetos a partir de una representación axonométrica de los mismos, utilizando la información sobre geometría tridimensional de los objetos que está implícita o explícitamente contenida en las figuras de partida. Por tanto, corresponde con el objetivo específico nº 1 descrito anteriormente.

Las especificaciones generales que se imponen como requisitos para el objetivo propuesto son:

- Los objetos a reconstruir serán de forma poliédrica, y la reconstrucción deberá ser independiente de la axonometría de partida.

- El modelo 2D se obtendrá a partir de dibujos preexistentes, almacenado en algún formato gráfico estándar como puede ser DWG, DGN ó DXF.

- El método de reconstrucción se basará en la determinación de "regularidades" de la imagen de partida, la formulación matemática de las mismas y la optimización numérica del problema de dichas formulaciones.

- La implementación se realizará en lenguaje C y se utilizará la herramienta de programación Microsoft Visual C++ utilizando las Microsoft Foundation Classes, buscando obtener un entorno de interacción amigable como cualquier aplicación Windows. Este entorno tendrá funciones básicas de edición de vértices, aristas y caras.

- La visualización de los modelos 2D y 3D se obtendrá utilizando la librería gráfica estándar OpenGL. Los modelos 3D se podrán ampliar y mover libremente en el espacio, para poder apreciar el resultado final de la reconstrucción.

Se propone, por tanto, implementar un algoritmo que permita obtener un modelo tridimensional a partir de una figura vectorial plana representativa de un diseño en perspectiva axonométrica. El desarrollo de dicho algoritmo estará basado en métodos de optimización cuya función objetivo quedará definida por las regularidades que se definan a partir de las relaciones geométricas existentes en la perspectiva de partida, que deberán ser detectadas automáticamente por el algoritmo.

Los objetivos específicos en los que podemos descomponer el objetivo general del PFC son:

1. Desarrollo de un programa "preprocesador" para la lectura, almacenamiento y visualización de figuras geométricas planas. Dichas figuras entrarán como datos del preprocesador, en formato vectorial y estructuradas según algunos de los protocolos más habituales en las principales aplicaciones CAD (ficheros tipo DXF, DWG, DGN, IGES, etc.).

2. Creación de un módulo de edición que le permita al usuario modificar y manipular las figuras leídas como datos. La capacidad de edición deberá permitir al usuario "retocar" las imágenes de partida para facilitar la tarea posterior de reconstrucción. Se prevé que pueda ser necesario modificar los atributos de algunos elementos de la figura, eliminar algunas partes (por ejemplo ocultar una capa, o quitar un texto), etc.

3. Generación de un módulo de reconstrucción basado en la metodología de optimización por inflado. Dicho módulo debe constar de dos partes principales:

    a) Establecimiento de una función objetivo válida para un rango de modelos lo más general posible, que contemple todas las características geométricas del modelo que están contenidas de forma explícita o implícita en la figura que lo representa.

    b) Selección e integración en el sistema de los algoritmos de optimización que se consideren mejor adaptados a las particularidades del problema (problema complejo con muchos mínimos locales en la función objetivo, etc.)

4. Desarrollo de un programa "postprocesador" para la visualización, selección y almacenamiento de los modelos resultantes. Dichos modelos deberán almacenarse en el método elegido (BRep) y la información que los define deberá quedar estructurada según algunos de los protocolos más habituales en las principales aplicaciones CAD (OpenGL, IGES, DXF, etc.).

3. MÉTODO DE RECONSTRUCCIÓN BASADO EN OPTIMIZACIÓN

3.1 Introducción

Desde el punto de vista de la Geometría, siempre se ha sabido que la recuperación plena de un modelo geométrico 3D a partir de una sola proyección de él no es posible. No obstante, en el campo de la Psicología es también muy bien conocido el hecho que los humanos parecen no tener ningún problema para identificar modelos 3D a partir de imágenes 2D. Lo que es más, parece haber un gran acuerdo general sobre cuál es el *"único modelo correcto"* que todos los humanos ven en cada imagen; aunque pueden existir varias interpretaciones de un único modelo, lo cierto es que a nuestra vista siempre nos llega una única interpretación válida, la más psicológicamente plausible. La razón viene del hecho que los humanos, cuando "leen" los dibujos, efectúan acciones de recuperación implícitas. Según la escuela de Gestalt, esto es porque la percepción humana mantiene algunas características comunes, llamadas "los principios de organización."

Es más, se realizan suposiciones implícitas de un modo iterativo y tentativo, según el método de "prueba y error". Es decir, desde la primera mirada, los humanos hacen algunas suposiciones acerca de cuál es el modelo 3D que se representa en una imagen. Poco a poco, esta suposición inicial se validando o modificando, mediante un proceso mental de aproximación. Este proceso depende entre otros de la experiencia e intuición previa que el espectador tiene sobre qué tipo de objetos puede corresponder a la imagen dada y cuál era la intención del autor.

Por tanto, un proceso iterativo dónde alguna solución inicial es refinada de acuerdo a unas características percibidas, parece ser una buena estrategia para conseguir lo que la sola aplicación de reglas de la geometría no puede obtener: un modelo 3D psicológicamente creíble. Este proceso considera las mismas suposiciones que un humano puede hacer, y desecha esas suposiciones iniciales que se demuestra que no son válidas.

La optimización numérica constituye, a nuestro entender, uno de los caminos más prometedores para *reconstruir modelos geométricos*, es decir, para generar automáticamente modelos virtuales 3D, partiendo de figuras en 2D que se consideran proyecciones de dicho modelo. La razón es que los procesos iterativos característicos de la optimización guardan una cierta similitud con la manera de operar de la percepción humana [12],[13], como acabamos de ver en la introducción.

En la reconstrucción geométrica por optimización, el modelo a reconstruir se "infla", es decir, que se expresa haciendo coincidir las coordenadas (X,Y) de sus vértices con las de la imagen de partida (por lo que permanecen inalteradas) mientras las coordenadas Z de dichos vértices constituyen las variables de diseño del problema. Como consecuencia, un numero infinito de modelos son válidos, porque todos ellos se pueden proyectar sobre la misma imagen de partida. Marill [8] llamó **extensión ortográfica** al conjunto infinito de modelos tridimensionales cuya proyección ortogonal es igual a la figura inicial. En la Figura 3-1 podemos ver una figura 2D de partida y algunos modelos 3D que forman parte de su extensión ortográfica.

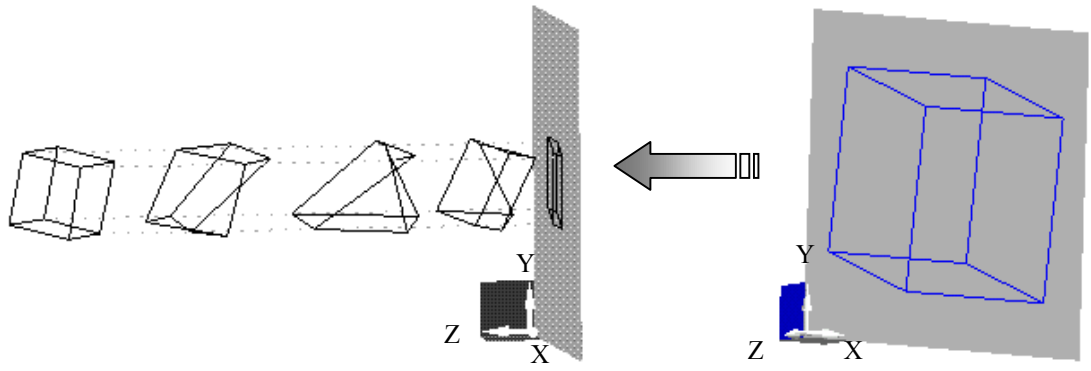


Figura 3-1. Extensión ortográfica (izquierda) de una imagen de partida (derecha)

Por consiguiente se precisa un mecanismo capaz de seleccionar, de entre el conjunto de modelos de la extensión ortográfica, aquel que resulte relevante y acorde con la percepción visual humana.

Marill [8] propuso que el problema de reconstrucción geométrica de modelos alámbricos de poliedros a partir de una vista proyección axonométrica del modelo podía

ser descrito en términos de un problema de optimización matemática, partiendo de la base que:

a) La figura está formada por segmentos de recta que se unen en vértices.

b) Los vértices del modelo corresponden con los vértices de la figura.

c) Cada uno de los segmentos de la figura se corresponde con una arista del modelo.

d) Las coordenadas (X,Y) de los vértices del modelo reconstruido coinciden con las coordenadas (X,Y) de los vértices de la figura.

e) Las coordenadas Z se determinan de modo que el modelo resultante no sea una *solución valida*, sino la *solución buena* o el *único modelo correcto* que hemos visto en la introducción a este capítulo.

Para encontrar la solución buena, necesitaremos una *Función Objetivo* definida de tal manera que esta solución buena sea "barata" o la solución de mejor coste ("solución óptima" hablando en términos de optimización matemática).

3.2 Formulación del problema de Reconstrucción mediante Optimización

La propuesta de Marill fue definir una función objetivo cuyas variables fueran las coordenadas Z de los vértices y que tomara un valor mínimo cuando los ángulos formados por todas las parejas de aristas que concurren en cada vértice fuesen iguales. Para ello, expresó la desviación estándar de los ángulos en función de las coordenadas Z y minimizó dicha desviación.

En el método propuesto posteriormente por Leclerc y Fischler [9], la función objetivo era mejorada al incluir la exigencia de que las caras del poliedro cumplieran la condición de planicidad. Esta aportación iba acompañada del correspondiente algoritmo capaz de detectar las caras en la figura de partida.

Posteriormente Lipson y Sphitalni [10] propusieron una ampliación del método mediante la definición de un conjunto de "regularidades" basadas en reglas heurísticas. Se trata de aquellas propiedades que se pueden descubrir inspeccionando la figura y que se puede presuponer que deben cumplirse *también* en el modelo final. Estas propiedades se denominan genéricamente regularidades porque las primeras que se propusieron eran características propias de las figuras regulares. Las regularidades son la forma de hacer explícita la percepción visual humana. Por ello se deben aplicar como características que *es probable* que no sean accidentales. Es decir, que no se deben formular como ecuaciones matemáticas de obligado cumplimiento, porque ello daría lugar a sistemas de ecuaciones mal definidos e irresolubles. Por el contrario, se debe "premiar" su cumplimiento, pero aceptando como posibles las soluciones que no cumplan todas las condiciones impuestas. En consecuencia, se formula una Función Objetivo, en lenguaje matemático, con las coordenadas Z de los vértices como variables independientes:

$$F= \Sigma \alpha_j R_j(z)$$

Expr 3-1

donde,

- α_j es el j-esimo coeficiente de ponderación, y
- $R_j(z)$ es la j-esima regularidad.

En definitiva, el problema se plantea como la optimización de una función objetivo F(z), donde z es el vector de coordenadas (z_1, z_2, z_3, … , z_n) de los vértices de la imagen.

Se puede decir que la función objetivo propuesta por Marill estaba formada por una única regularidad, la MSDA (mínima desviación estándar de ángulos), que primaba la obtención de modelos muy "regulares" en el sentido clásico del término. Pero Marill no la aplicó como una auténtica regularidad, dado que la exigía indiscriminadamente, sin deducir previamente si la figura a la que se aplicaba tenía tal cualidad.

3.3 Mínimos locales en la Reconstrucción mediante Optimización

Los algoritmos de optimización parten de una solución inicial, que se supone mala, y van escogiendo iterativamente soluciones mejores que la anterior, aproximándose poco a poco a la solución óptima, y deteniéndose cuando ya no encuentran ninguna solución mejor. Puede suceder que en un momento dado se haya escogido un camino que desemboque en un *mínimo local*, algo así como una calle sin salida de la que el algoritmo no sabe salir.

Veamos la Figura 3-2. Suponemos que el algoritmo de optimización parte inicialmente de la posición 1, y ha optado por dirigirse hacia la izquierda llegando a la posición 2. Si el algoritmo es simple y sólo busca soluciones mejores, no sabrá salir de 2. Este es un *mínimo local*. El mínimo global está en la posición 3, a la que nunca llegará. Por tanto este algoritmo nunca nos dará la mejor solución.

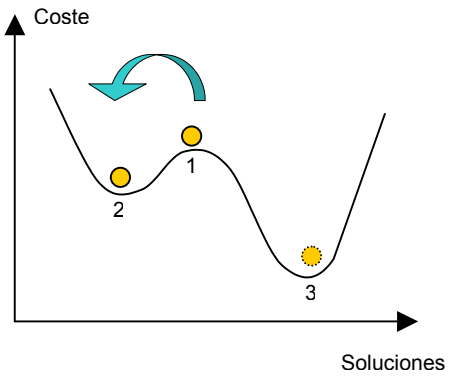


Figura 3-2. Ejemplo de mínimo local

La Reconstrucción difiere de otros problemas de optimización en un hecho capital: los mínimos locales son soluciones no deseadas, porque para el usuario sólo es válido el modelo 3D que él mismo habría reconstruido mentalmente.

Dicho de otro modo, no importa cuán cerca puede estar la Función Objetivo de un mínimo local respecto de la solución "real", porque la solución deseada debe ajustarse a parámetros psicológicos, no a diferencias numéricas.

Aquí, mientras un mínimo local, o simplemente una mejora significativa en la Función Objetivo, puede ser un buen resultado en otros campos, una reducción similar en la Función Objetivo de una Reconstrucción Geométrica no soluciona ningún problema. Porque el mínimo local corresponderá a un componente de la extensión ortográfica cuya forma puede diferir de la solución psicológicamente plausible.

Pongamos por ejemplo el caso de un problema de optimización de una estructura que hay que construir, donde la Función Objetivo mide el coste monetario de la estructura. Cualquier reducción en su valor es un buen resultado, porque nos lleva a una construcción más barata. Pero este no es el caso de la Reconstrucción Geométrica, porque un mínimo local representa un modelo que difiere significativamente del que el usuario espera obtener. Por tanto, el usuario no aceptará el proceso de reconstrucción como un método valido y confortable de comunicación con el ordenador.

Así pues, necesitamos algoritmos que sean capaces de encontrar mínimos globales.

Para evitar los mínimos locales, la primera estrategia que implementamos es la de Leclerc y Fischler [9], consistente en que la función objetivo era balanceada por un parámetro λ cuyo valor decrecía desde 1 a 0 en la expresión:

$$F= F_A + \lambda F_I + (1-\lambda)\, F_O$$

Expr 3-2

donde F_I representa la parte de inflado inicial, falsas regularidades para escapar del óptimo trivial de la imagen y que no tienen porqué ser verificadas en el óptimo global, Fo representa el conjunto de verdaderas regularidades que son inicialmente verificadas en la imagen, y F_A representa verdaderas regularidades que no son inicialmente nulas en la imagen.

La estrategia aportada en este trabajo es la de implementar métodos de optimización diseñados específicamente para evitar los mínimos locales y encontrar los mínimos globales.

En concreto, en este PFC hemos optado por un algoritmo de optimización novedoso y prometedor llamado *Simulated Annealing* que en teoría es capaz de escapar de mínimos locales. Es la primera vez que se utiliza este algoritmo en el ámbito de la Reconstrucción Geométrica. Se estudia con detalle en el Capitulo 4.2.3

3.4 Métodos de Optimización para Reconstrucción

Se han propuesto muchos algoritmos para la optimización numérica [14]. Los criterios para seleccionar el algoritmo apropiado en cada caso pasan por considerar aspectos muy dispares y, a veces, contrapuestos:

- La búsqueda puede ser **exhaustiva** o **iterativa**. Explorar todas las soluciones posibles no es viable más que en los casos más elementales de la reconstrucción geométrica. Por tanto, la búsqueda exhaustiva no se emplea.

  Los métodos iterativos difieren en la estrategia de búsqueda. En base a la información utilizada, se distingue entre **búsqueda no guiada** (de orden cero) y **búsqueda guiada** (de primer o segundo orden, dependiendo de que se disponga de las primeras o las segundas derivadas de la función objetivo). Leclerc y Fischler implementaron y estudiaron el comportamiento del algoritmo del "Gradiente Conjugado" descrito en [14], usando el cálculo numérico para determinar las derivadas parciales de la función objetivo.

  La búsqueda no guiada puede ser aleatoria o puede estar basada en un criterio simple que decida la dirección de búsqueda (como mantener una dirección mientras la solución no empeore, o como el algoritmo de descenso "Hill Climbing" utilizado por Marill).

  En la búsqueda iterativa también es importante definir el criterio de parada: cuando se considera que se ha encontrado la mejor solución.

- Atendiendo a las variables se distingue entre **programación continua** (las variables pueden tomar cualquier valor real), **programación entera** (sólo pueden tomar valores enteros) y **programación discreta** (sólo pueden tomar valores de un conjunto predefinido). Los métodos de búsqueda continua son eficientes y se adaptan bien a la Reconstrucción Geométrica por inflado.

- Atendiendo a la complejidad de la función objetivo se puede distinguir entre **programación lineal** y **no lineal**. Las regularidades son no lineales respecto a las coordenadas Z de los vértices. Por tanto, el problema es siempre no lineal.

  El tamaño del problema aun no constituye un obstáculo serio, dado que los desarrollos son experimentales y los casos tratados hasta la fecha son abordables sin tener que recurrir a técnicas de reducción del tamaño, tales como la condensación de variables.

- Atendiendo a las restricciones se consideran métodos que las tratan **explícitamente** (optimización con restricciones) y otros que las consideran **implícitamente** (optimización con penalizaciones). Las regularidades son restricciones implícitas, dado que no se deben cumplir imperativamente.

- La existencia de mínimos locales provoca la distinción entre **algoritmos locales** (que son aquellos que encuentran óptimos en las cercanías de la solución inicial) y **algoritmos globales** (que buscan en todo el espacio de soluciones).

Tras exponer los diferentes tipos de algoritmos existentes, podemos resumir los antecedentes de su aplicación al problema de Reconstrucción. Para ello, basta indicar que los métodos aplicados hasta la fecha en reconstrucción han sido iterativos, continuos, no lineales y sin restricciones.

Marill utilizó un algoritmo simple de descenso (Hill Climbing).

Posteriormente Lipson y Sphitalni exploraron tres algoritmos diferentes de optimización, “la interpolación parabolica (Minimización de Brent)”, el método del “Gradiente Conjugado” y un “Algoritmo Genético”. De sus experiencias extrayendo las siguientes conclusiones:

- El algoritmo Cíclico de Minimización de Brent parecía ser el mas prometedor.

- El Algoritmo del Gradiente Conjugado convergía rápidamente pero en algunos casos no converge.

- El método Genético evitaba los mínimos locales pero precisaba de un elevado número de iteraciones.

En este trabajo se presentan dos algoritmos de optimización, por un lado se ha implementado el algoritmo de Hill Climbing para estudiar y contrastar los ejemplos propuestos por Marill y por Leclerc y Fischler. Por otro lado, se ha implementado el algoritmo Simulated Annealing para comprobar si es capaz de escapar de mínimos locales y encontrar el óptimo global.

4. MÉTODOS DE OPTIMIZACIÓN

4.1 Optimización Hill-Climbing

El algoritmo Hill-Climbing fue el primero utilizado como herramienta para la reconstrucción mediante procesos de optimización.

Se trata de un algoritmo iterativo "de descenso". Es decir, que la elección de la dirección de búsqueda se hace de forma exhaustiva (se calculan todas las posibles direcciones), y se elige aquella que consigue un mayor descenso.

El algoritmo se debe completar con un "criterio de parada". El criterio habitual es el que se denomina "de equilibrio", que consiste en parar cuando no se puede encontrar una solución que mejore a la actual con un porcentaje de mejora mayor que un valor fijado por el usuario.

Se suele añadir un segundo criterio de parada fijando el número máximo de iteraciones permitidas.

4.1.1 Descripción del algoritmo Hill-Climbing

Dada una función no lineal F = F(z) donde ***z*** es el vector de las variables a optimizar de ***n*** componentes: $z = (z_0, z_1, \ldots z_{n-1})$, la función es evaluada de manera iterativa para los 2n vectores siguientes:

$$
\begin{aligned}
z_1 &= (z_0 + s, z_1, \ldots z_{n-1}) \\
z_2 &= (z_0 - s, z_1, \ldots z_{n-1}) \\
z_3 &= (z_0, z_1 + s, \ldots z_{n-1}) \\
z_4 &= (z_0, z_1 - s, \ldots z_{n-1}) \\
&\ldots\ldots\ldots\ldots\ldots\ldots\ldots \\
z_{2n-1} &= (z_0, z_1, \ldots z_{n-1} + s) \\
z_{2n} &= (z_0, z_1, \ldots z_{n-1} - s)
\end{aligned}
$$

donde ***s*** toma valores decrecientes para cada "escalón" del algoritmo.

El proceso se ejecuta repetidamente para los 2n vectores de cada escalón mientras no se alcance alguno de los siguientes criterios de equilibrio:

- La evaluación de la función no disminuye su coste en un cierto valor.
- El número de iteraciones realizado en cada escalón sobrepasa un máximo establecido.

Obtenido el equilibrio para un determinado escalón *k*, el proceso se repite para el siguiente escalón *k +1*.

Se ha establecido un coeficiente de aceptación de soluciones independiente para el último escalón del proceso, porque permite exigir distintos grados de refinado en la solución final. El diagrama de flujo del algoritmo puede verse en la Figura 4-1.

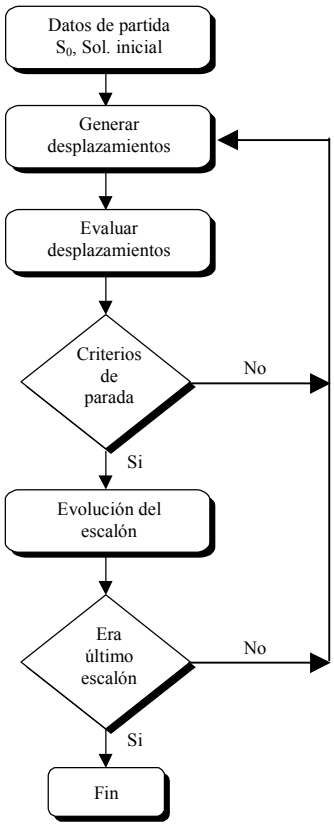


Figura 4-1. Diagrama de flujo del algoritmo Hill-Climbing

En la Figura 4-2 y Figura 4-3 se muestran las gráficas de la evolución del coste para distintos escalones del algoritmo de Hill-Climbing. En la Figura 4-2 se han

establecido los escalones propuestos por Marill (1, 0.5, y 0.1). En la Figura 4-3 se han establecido los escalones utilizados por Leclerc y Fischler (0.125, 0.0625, 0.03125, 0.015 y 0.007). Estos escalones son totalmente arbitrarios. En el Capítulo 4.1.2 se discutirá sobre la conveniencia de éstos y la implementación que proponemos nosotros.

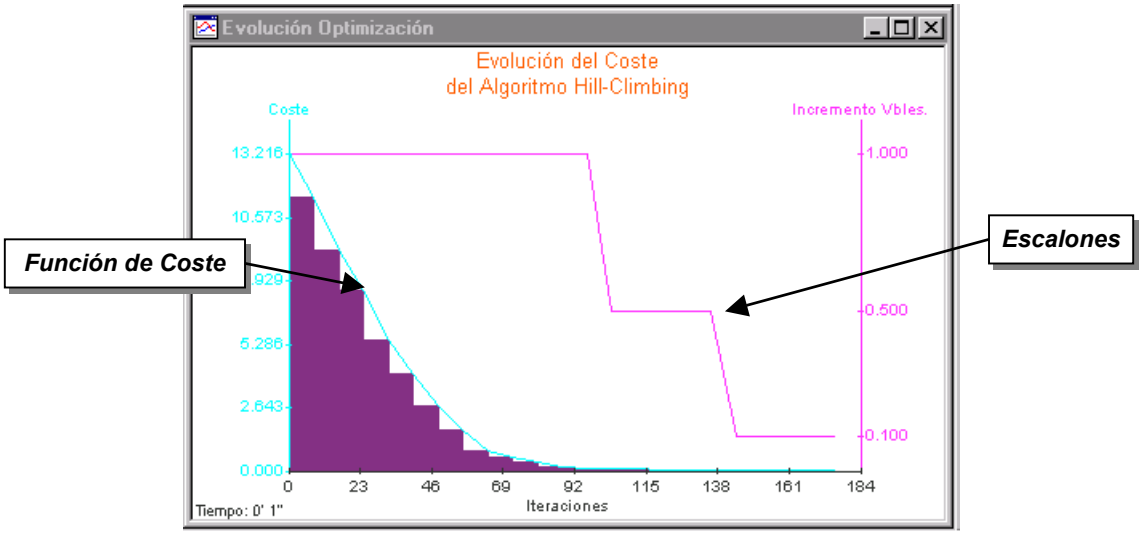


Figura 4-2. Diagrama de evolución del coste para escalones propuestos por Marill

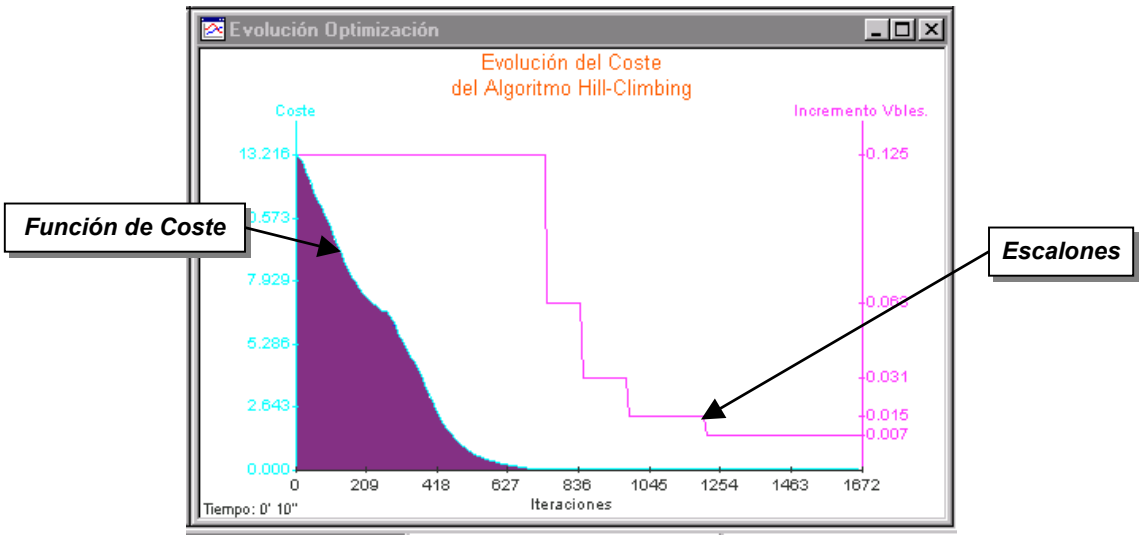


Figura 4-3. Diagrama de evolución del coste para escalones según Leclerc y Fischler

La función analizada ha sido definida con ocho variables (reconstrucción de un cubo), y optimizada fijando un número máximo de 800 iteraciones por escalón.

En las representaciones puede observarse el carácter siempre descendiente de la función coste definido por los criterios de aceptación del algoritmo y que pueden ocasionar la obtención mínimos locales como soluciones al problema de optimización. La función decrece de manera más brusca durante el primer escalón (escalón con mayor incremento de variables), suavizándose su monotonía para escalones más pequeños.

4.1.2 Implementación del Hill-Climbing en la Reconstrucción Geométrica

4.1.2.1 Generación de soluciones y Evolución de los escalones

Como ya hemos visto en el capitulo anterior, el funcionamiento del algoritmo Hill-Climbing consiste en que a partir de una solución, se busca la solución mejor (si la hay) del conjunto de soluciones que formamos a partir de incrementar o decrementar en una cantidad Δz cada una de las variables z. Este Δz es fijo para cada escalón, y tiene sentido que disminuya cada vez que se salta al escalón siguiente. El salto entre escalones se produce cuando ya no encontramos soluciones que mejoren la actual. Después del último escalón, el algoritmo se detiene.

Por tanto, vemos que el número de escalones puede ser determinante en el comportamiento del algoritmo. También apreciamos intuitivamente que el Δz para cada escalón debe ser un factor crítico. Veamos como se utilizan estos parámetros tan importantes en los trabajos de investigación de Marill por un lado y Leclerc y Fischler por otro.

- **Marill** aplica un conjunto fijo de 3 escalones: **Δz = (1, 0.5, 0.1)**. Si nos fijamos en los ejemplos que utiliza, nos damos cuenta que todos tienen un *rango máximo de x (Δx)* = 7.79 y un *rango máximo de y (Δy)* = 8.16.

  Esto significa que el escalón inicial (que es la máxima variación permitida a las coordenadas z en una sola iteración) es el 12% del máximo(Δx, Δy).

- **Leclerc y Fischler** aplican 5 escalones: **Δz = (0.125, 0.0625, 0.03125, 0.015 y 0.007)**. En sus ejemplos vemos que todos tienen un *rango máximo de x (Δx)* = 4.78 y un *rango máximo de y (Δy)* = 3.50.

Esto significa que el escalón inicial es de sólo el 2% del máximo(Δx, Δy). Ellos argumentan que escalones más grandes fuerzan al algoritmo a salirse del valle de atracción del mínimo local actual.

Vemos que Leclerc y Fischler utilizan más escalones que Marill. Esto incrementa el tiempo de cálculo, pero ayuda a conseguir soluciones más precisas. Además, ellos decrementan Δz en un factor de 2.

Según nuestra experiencia, no hay diferencias significativas si el decremento de Δz se hace constante, porque el algoritmo por sí mismo corta estos niveles en donde no se consiguen mejoras significativas. No obstante, evitar diferencias elevadas en los escalones es una garantía contra fallos en el algoritmo, y el mejor descenso en la función objetivo parece que se obtiene cuando los decrementos siguen una curva exponencial en lugar de lineal. También consideramos que utilizar más de diez escalones parece que no tiene efecto en el resultado final. Lo que si que afecta a la precisión final es el valor del Δz más pequeño.

Llegamos a la conclusión que si definimos la magnitud de los escalones como una proporción del máximo(Δx, Δy) conseguiremos que el algoritmo sea completamente independiente de las dimensiones de la figura de partida. También concluimos que de 5 a 10 escalones parece ser un buen rango para obtener buenas soluciones en un tiempo razonable, asumiendo que el Δz mayor debe ser menor que el 100% del máximo(Δx, Δy). Nosotros sugerimos el rango 10-20%, porque es un modo indirecto de imponer la suposición psicológica de *proporcionalidad*. En nuestras pruebas, si utilizamos Δz grandes, la solución no respeta esta proporcionalidad. Por ejemplo, en la Figura 4-4, podemos ver que partiendo del gráfico inicial, con escalones iniciales mayores que el máximo(Δx, Δy) se obtiene un prisma alargado de sección cuadrada, en lugar del cubo que cualquier persona “puede ver”.

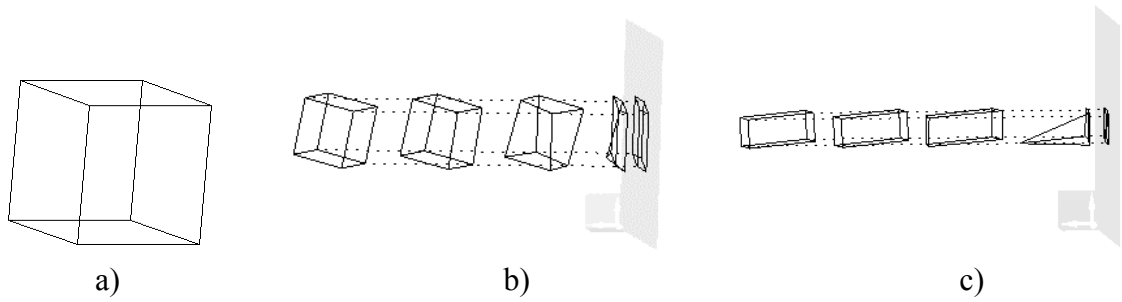


Figura 4-4. Ejemplo de la influencia de los escalones en la Proporcionalidad de los resultados.
a) gráfico 2D de partida
b) modelo psicológicamente plausible (obtenido con max(Δz)=0.2*max(Δx,Δy))
c)modelo no proporcional (obtenido con max(Δz)=2.0*max(Δx,Δy))

4.1.2.2 Programación del algoritmo Hill-Climbing.

A continuación presentaremos el código del algoritmo Hill-Climbing, subrayando aquellas líneas que son las realmente importantes y que forman parte del esqueleto del mismo. Este código esta dentro de las clases C++ de la aplicación, porque necesitamos informar al usuario de cómo va la ejecución. Obsérvese que de no ser por esta necesidad, el código está programado con lenguaje C.

Código Fuente 4-1. Algoritmo Hill-Climbing

```
ESTADO CReferDoc::OptimizacionHillClimbing(
         TListaDouble *pListaVariables,
         TBDEntidades *pBD,
         TParametrosOptimizacion *pParametrosOpt,
         long *pNumeroIteracionesRealizadas,
         TListaDouble *pListaEvolucionCoste,
         TListaDouble *pListaTodasSoluciones,
         TListaDouble *pListaEvolucionZ,
         CDialogProgreso *pDialogoProgreso )

{
ESTADO Estado;
int i, iNumeroVariables,
         iContadorIteraciones,
         iNumeroMaximoIteraciones,
         iSolucion,
         iMejorSolucion,
         iContadorIncrementosVariables,
         lContadorTotalIteraciones;
double dMejorCoste,
         dCosteActual,
         dIncrementoVariable,
         dCoefAceptacionMejorCoste,
         dDato;
TListaDouble NuevaSolucion,
              MejorSolucion;
BOOLEAN bHayMejorSolucion;
```

```
/* INICIALIZACION DE VARIABLES */

iNumeroMaximoIteraciones = pParametrosOpt->HillClimbing.iNumeroMaximoIteraciones;
dCoefAceptacionMejorCoste = 1 - pParametrosOpt->HillClimbing.dCoefRechazoMejorCoste;
iNumeroVariables = pParametrosOpt->iNumeroVariables;

if (pParametrosOpt->HillClimbing.iEstiloIncrementoVariables ==
    INCREMENTO_VARIABLE_PROPORCIONAL)
        EstimaIncrementosVariablesHillClimbing( pBD,
                &pParametrosOpt->HillClimbing.ListaIncrementosVariables,
                pParametrosOpt->HillClimbing.dCoefIncrementoVariables,
                pParametrosOpt->HillClimbing.iNumeroIncrementoVariables,
                pParametrosOpt->HillClimbing.dCoefPasoIncrementoVariables );

/* Reinicia las listas de soluciones y mejores soluciones */
NuevaListaDouble( &NuevaSolucion );
NuevaListaDouble( &MejorSolucion );
CopiaListaDouble( pListaVariables, &MejorSolucion );

ActualizaCoeficientesRegularidades( pParametrosOpt->Regularidades.ListaCoeficientes, 0);

/* REPETIR PARA CADA INCREMENTO DE VARIABLE */
iContadorIncrementosVariables = 0;
lContadorTotalIteraciones = 0;
long lContadorIteraciones_ParaCoeficientesRegularidades = 0;
do {
        pParametrosOpt->HillClimbing.iIncrementoVariableActual =
        iContadorIncrementosVariables;

        /* REPETIR HASTA QUE NO SE PUEDA MEJORAR O SE ALCANCE EL MAXIMO DE ITERACIONES */
        iContadorIteraciones = 0;
        do {
                ActualizaCoeficientesRegularidades(
                        pParametrosOpt->Regularidades.ListaCoeficientes,
                        lContadorIteraciones_ParaCoeficientesRegularidades );

                /* Guarda el Incremento de Variable utilizado ahora. */
                ObtenListaDouble( &pParametrosOpt->HillClimbing.ListaIncrementosVariables
                        , pParametrosOpt->HillClimbing.iIncrementoVariableActual,
                         &dIncrementoVariable);
                Estado = AnyadeListaDouble( pListaEvolucionCoste, dIncrementoVariable );
                ASSERT( Estado == OK );

                /* Voy a buscar alguna solucion que mejore la que tengo hasta ahora */
                dMejorCoste = CalculaCoste( &MejorSolucion, pBD, pParametrosOpt,
                pListaEvolucionCoste, pListaEvolucionZ, TRUE );
                iMejorSolucion = -1;

                /* REPETIR PARA CADA VARIABLE */
                /* Las soluciones pares seran obtenidas con incrementos positivos, y las
                   impares, negativos. */
                for (iSolucion = 0; iSolucion < iNumeroVariables*2; iSolucion++ ) {
                        pParametrosOpt->HillClimbing.iSolucionActual = iSolucion;

                        /* Dada la solucion 'iSolucion' obtiene la solucion 'iSolucion+1' */
                        GeneraNuevaSolucionHillClimbing( &MejorSolucion, &NuevaSolucion,
                                pParametrosOpt );
                        dCosteActual = CalculaCoste( &NuevaSolucion, pBD, pParametrosOpt,
                                pListaEvolucionCoste, pListaEvolucionZ, FALSE );

                        /* Comprueba si es la mejor solucion hasta el momento */
                        if ( dCosteActual < dMejorCoste * dCoefAceptacionMejorCoste ) {
                                dMejorCoste = dCosteActual;
                                iMejorSolucion = iSolucion;
                        };
                };
```

```
                /* Si hay una solucion mejor, se guarda. */
                if (iMejorSolucion != -1) {
                        pParametrosOpt->HillClimbing.iSolucionActual = iMejorSolucion;
                        GeneraNuevaSolucionHillClimbing( &MejorSolucion, &NuevaSolucion,
                                pParametrosOpt );
                        DestruyeListaDouble( &MejorSolucion );
                        NuevaListaDouble( &MejorSolucion );
                        CopiaListaDouble( &NuevaSolucion, &MejorSolucion );
                        bHayMejorSolucion = TRUE;
                }
                else
                        bHayMejorSolucion = FALSE;


/* ------------------------------------------------------------------------- */
/* Con esta funcion se actualiza la barra de progreso, y se permite que las otras
   aplicaciones Windows funcionen en multitarea. (Ver 'ActualizaProgreso').
   Si 'Estado' no es OK, entonces hay que cancelar el algoritmo.                */

Estado = pDialogoProgreso->ActualizaProgreso(
        lContadorIteraciones_ParaCoeficientesRegularidades );

// Dibuja el objeto segun la solucion de menor coste conseguida hasta este momento.
if (pParametrosOpt->bVerEvolucionPasoAPaso) {
        ActualizaZconSolucionesBDEntidades( &MejorSolucion, pBD );
        m_bCalcularVolumenOpenGL = true;
        RefrescaVentanaReconstruccion3D();
};
/* ------------------------------------------------------------------------- */

        /* Guarda TODAS las MEJORES soluciones en CADA ITERACION. */
                for( i=0; i < iNumeroVariables; i++ ) {
                        ObtenListaDouble( &MejorSolucion, i, &dDato );
                        AnyadeListaDouble( pListaTodasSoluciones, dDato );
                };

                lContadorIteraciones_ParaCoeficientesRegularidades++;
                iContadorIteraciones++;

        } while ((iContadorIteraciones < iNumeroMaximoIteraciones) &&
                (bHayMejorSolucion == TRUE) &&(Estado==OK));

        iContadorIncrementosVariables++;

} while ((iContadorIncrementosVariables < TamanyoListaDouble( &pParametrosOpt->
        HillClimbing.ListaIncrementosVariables )) && (Estado==OK));

if (Estado != CANCELADO_POR_USUARIO) {
        /* Se devuelve el resultado en 'pListaVariables' */
        DestruyeListaDouble( pListaVariables );
        NuevaListaDouble( pListaVariables );
        CopiaListaDouble( &MejorSolucion, pListaVariables );
};

DestruyeListaDouble(&NuevaSolucion);
DestruyeListaDouble(&MejorSolucion);

*pNumeroIteracionesRealizadas = lContadorTotalIteraciones;
return Estado;
};
```

El algoritmo principal utiliza dos importantes funciones **GeneraNuevaSolucionHillClimbing** y **CalculaCoste**. La primera paso a describirla

ahora, y la segunda se puede ver en el *Capitulo 5.6. Implementación de la función objetivo.*

Código Fuente 4-2. Función GeneraNuevaSolucionHillClimbing

```
void GeneraNuevaSolucionHillClimbing(
        TListaDouble *pSolucion,
        TListaDouble *pNuevaSolucion,
        TParametrosOptimizacion *pParametros)

/* Dada la solucion 'pSolucion', calcula la solucion 'pNuevaSolucion'. */
{
        int iVariableModificada;
        double dDato, dIncremento;
        div_t ResultadoDivision;

        /* Las soluciones pares seran obtenidas con incrementos positivos, y las impares, negativos. */

        /* Primero prepara la nueva solucion, a partir de los valores de la solucion anterior. */
        DestruyeListaDouble( pNuevaSolucion );
        NuevaListaDouble( pNuevaSolucion );
        CopiaListaDouble( pSolucion, pNuevaSolucion );

        /* Selecciona QUE variable sera modificada */
        ResultadoDivision = div(pParametros->HillClimbing.iSolucionActual, 2);
        iVariableModificada = ResultadoDivision.quot;
        ObtenListaDouble( pNuevaSolucion, iVariableModificada, &dDato );

        /* Selecciona CUANTO se modifica la variable */
        ObtenListaDouble( &pParametros->HillClimbing.ListaIncrementosVariables,
                          pParametros->HillClimbing.iIncrementoVariableActual, &dIncremento);

        /* Selecciona COMO se modifica la variable: sumando o restando */
        if ( ResultadoDivision.rem == 0 )
                dDato += dIncremento;
        else
                dDato -= dIncremento;

        SustituyeListaDouble( pNuevaSolucion, iVariableModificada, dDato );
};
```

4.2 Optimización Simulated-Annealing

4.2.1 Antecedentes históricos

Las técnicas de *búsqueda aleatoria dirigida* (ver Figura 4-5) están basadas en métodos enumerativos, pero utilizan información adicional para guiar la búsqueda. Son técnicas de propósito general, que se pueden aplicar a un amplio número de problemas.

Se dividen en dos familias principales: técnicas de *"simulated annealing"* y *"algoritmos evolutivos"*. Ambos tipos de técnicas son procesos evolutivos, ya que las técnicas SA utilizan un proceso de evolución termodinámica para buscar el estado de mínima energía, y los algoritmos evolutivos utilizan el principio de selección natural darwiniano.

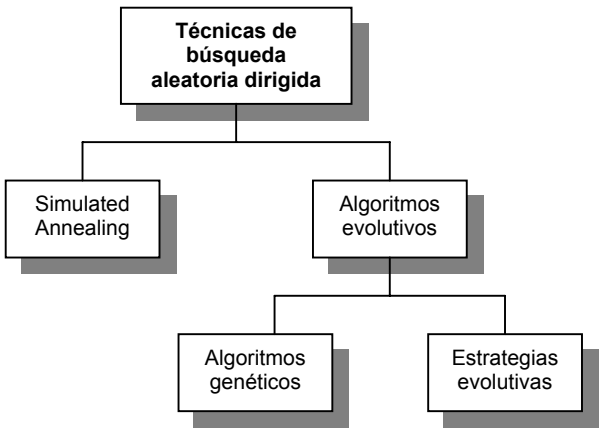


Figura 4-5. Técnicas de búsqueda aleatoria dirigida

La técnica del *Simulated Annealing* es una técnica de búsqueda aleatoria dirigida, que surge en 1983 con la publicación de un artículo de Kirkpatrick y otros [15].

Desde entonces, se viene empleando con profusión en el campo de la resolución de problemas de optimización de tipo combinatorio con excelentes resultados.

4.2.1.1 Símil termodinámico

El término "Simulated Annealing", que se podría traducir al castellano por "Recocido Simulado" hace referencia a un tratamiento térmico como es el "recocido" que según la acepción recogida en el Diccionario de la Lengua Española de la Real Academia Española significa "caldear los metales para que adquieran de nuevo la ductilidad o el temple que suelen perder al trabajarlos". A continuación se justificará la razón de utilizar este símil cuando realmente se está tratando de resolver un problema de optimización.

Como se indica en [15], existe una relación muy estrecha entre la Mecánica Estadística, que estudia el comportamiento de sistemas con muchos grados de libertad en equilibrio térmico, y los problemas de Optimización Combinatoria, que buscan minimizar una función objetivo que depende de muchos parámetros. La Mecánica Estadística es la base fundamental de la Física de la Materia Condensada. Dado el elevado número de átomos que constituyen una pequeña cantidad de materia (basta para ello recordar que doce gramos de carbono contienen $6{,}023 \times 10^{23}$ átomos), solamente se observa en los experimentos aquel estado más probable del sistema en equilibrio térmico a una temperatura dada.

Cada configuración del sistema, definida por un conjunto de posiciones de los átomos $\{r_i\}$, tiene asociada su factor de probabilidad de Boltzmann, $\exp(-E(\{r_i\})/k_B T)$, donde $E(\{r_i\})$ representa la energía de la configuración, k_B es la constante de Boltzmann y T es la temperatura.

Una cuestión fundamental de la Mecánica Estadística es el comportamiento del sistema a bajas temperaturas. Los estados fundamentales y las configuraciones próximas a ellos, representan un número extremadamente pequeño frente a todas las configuraciones de un cuerpo macroscópico, aunque determinan sus propiedades a bajas temperaturas, ya que al disminuir T, la distribución de Boltzmann tiende al estado de mínima energía. En las situaciones prácticas, una baja temperatura no es un condición suficiente para encontrar los estados fundamentales de la materia. Los experimentos que determinan el estado a bajas temperaturas de un material se realizan mediante un cuidadoso recocido del material, fundiendo primero la substancia, realizando a continuación un enfriamiento muy lento hasta la vecindad del punto de congelación.

Si no se emplea este método, y se permite que la substancia salga del punto de equilibrio, el cristal resultante tendrá muchos defectos, o se alcanzará un estado metaestable, con estructuras locales óptimas.

El encontrar el estado de baja temperatura de un sistema, calculando su energía, es un problema similar a uno de optimización combinatoria. Sin embargo el concepto de temperatura de un sistema físico no tiene un equivalente claro en el problema de optimización.

4.2.1.2 El algoritmo Metropolis

Metropolis et al. [16] presentaron en su artículo "Equation of state calculations by fast computing machines" uno de los primeros métodos de cálculo que aprovechaba las capacidades computacionales de los ordenadores. En su artículo se propone un método para calcular las propiedades de una substancia que se pueda considerar compuesta de moléculas individuales. Para reducir el tamaño del problema a uno abordable para los medios de cálculo disponibles, se considera un número finito de moléculas N, que se puede tomar hasta de varios cientos. El sistema consiste en una celda cuadrada que contiene las N partículas. Para evitar los efectos de borde, se supone que la substancia completa es periódica, consistiendo en muchas celdas cuadradas, cada celda conteniendo N partículas con la misma configuración.

Se define la distancia $\underline{d}_{AB}$ como la mínima distancia entre la partículas A y cualquiera de las partículas B, de las cuales hay una en cada una de las celdas en que se divide la substancia completa. En la Figura 4-6 se puede ver más claramente este concepto.

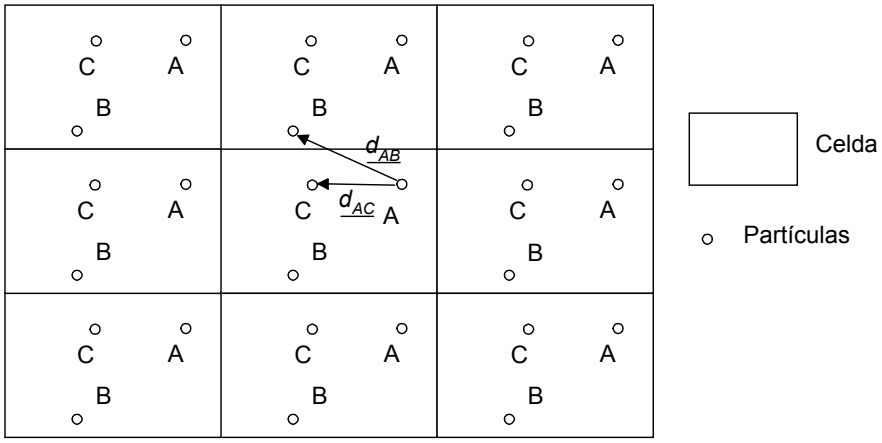


Figura 4-6. Substancia compuesta de celdas individuales cuadradas con N partículas

Si existe un potencial que disminuye muy rápidamente con la distancia, entonces solamente una de las distancias AB contribuirán de una manera significativa al potencial total; por ello sólo se considera la mínima distancia $\underline{d}_{AB}$.

Conocidas las posiciones de las N partículas de la celda, se puede calcular fácilmente la energía potencial del sistema:

$$E = \frac{1}{2}\sum_{i=1}^{N}\sum_{\substack{j=1 \\ j\neq i}}^{N} V(\underline{d}_{ij})$$

siendo $V(d_{ij})$ el potencial asociado a las partículas "*i*" y "*j*".

Cualquier variable macroscópica F del sistema, se obtiene a través de la expresión[1]:

$$\overline{F} = \frac{\int F e^{\frac{-E}{kT}} dpdq}{\int e^{\frac{-E}{kT}} dpdq}$$

donde *dp* y *dq* (extensión en fase) es el diferencial de superficie en el espacio de configuraciones 4-dimensional utilizado en Mecánica cuántica. Dada la dificultad de resolver la integral para los cientos de partículas existentes Metropolis propuso el siguiente algoritmo:

1) En primer lugar se distribuyen las N partículas siguiendo por ejemplo una distribución regular, por ejemplo en forma de rejilla.

2) Se desplaza cada partícula de su posición según la siguiente expresión: $\begin{matrix} X \longrightarrow X + \alpha\xi_1 \\ Y \longrightarrow Y + \alpha\xi_2 \end{matrix}$, donde α es el máximo desplazamiento permitido y ξ_1 y ξ_2 son números aleatorios entre -1 y 1. De esta forma después de realizar el desplazamiento, la posición de la partícula se encuentra dentro de un cuadrado de lado 2α centrado alrededor de su posición original.

[1] El objetivo de presentar esta expresión es observar como la aportación de cada elemento está ponderada por el factor de Boltzmann.

3) A continuación se calcula variación de energía provocada por el movimiento. Si $\Delta E<0$ entonces, ya que el movimiento conduce al sistema a un estado con menor nivel energético, dicho movimiento es permitido. Si $\Delta E>0$ entonces el desplazamiento es permitido con una probabilidad exp-$\Delta E/kT$); para ello se genera[2] un número aleatorio ξ_3 entre 0 y 1, de tal forma que si $\xi_3<\exp(-\Delta E/kT)$ se desplaza la partícula a su nueva posición. En caso contrario la partícula se devuelve a su posición inicial. Este criterio de aceptación se denomina **criterio Metropolis** en la bibliografía.

4) El algoritmo se va aplicando sucesivamente a diferentes partículas.

De esta forma repitiendo el proceso de selección de desplazamientos una y otra vez, el sistema evoluciona hasta alcanzar una distribución de Boltzmann.

Utilizando este símil sería posible el calcular una serie de magnitudes como podría ser el calor específico $C(T)$ que tendría por expresión:

$$C(T) = \frac{d < E(T) >}{dT} = \frac{< E(T)^2 > - < E(T) >^2}{kT^2}$$

Expresión 4-1

El calor específico da una idea de la variación de la energía del sistema respecto a la variación del parámetro de control T. Un valor elevado del mismo muestra un cambio en el estado de orden del sistema, que podría utilizarse para detectar un rango de temperaturas en el que el enfriamiento se debe realizar lentamente. Además se podría utilizar para calcular la entropía S del sistema utilizando la relación:

$$\frac{dS(T)}{dT} = \frac{C(T)}{T}$$

Expresión 4-2

De todas formas, las analogías entre el enfriamiento de un fluido y un problema de optimización puede fallar en un aspecto importante. En los fluidos ideales todos los átomos son similares y el estado fundamental es un cristal regular. En un problema de

[2] Utilizando una distribución uniforme.

optimización pueden intervenir un conjunto de elementos muy distintos y no intercambiables, por lo que el obtener una solución regular es muy improbable.

La técnica del "Simulated Annealing", que de aquí en adelante se denominará genéricamente **SA** difiere de las técnicas tradicionales de mejora iterativa, en su capacidad para escapar de mínimos locales gracias al empleo del criterio Metrópolis para la aceptación de los desplazamientos, tal como se puede observar en la Figura 4-7.

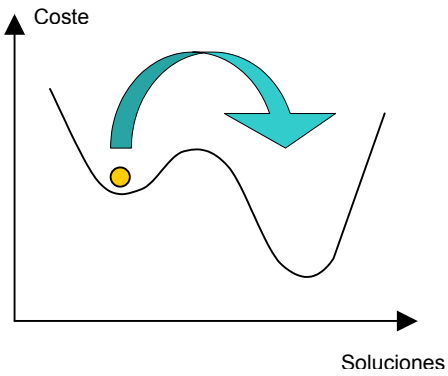


Figura 4-7. Simulated Annealing escapa de mínimos locales

Si se aplica el algoritmo Metrópolis a un problema de optimización, el símil físico sería el siguiente: la función de coste del problema de optimización equivaldría a la energía del sistema, y la configuración del sistema correspondería al conjunto de parámetros a optimizar. En cuanto a la temperatura pasaría a ser simplemente un parámetro de control del algoritmo, que tendría las mismas unidades que la función de coste.

4.2.2 Descripción del algoritmo Simulated Annealing

Todo algoritmo SA (Figura 4-8) se puede caracterizar por una serie de parámetros, unos específicos del problema en cuestión y otros genéricos, independientes de la naturaleza del problema.

Como ***parámetros específicos*** del problema tenemos:

- un ***espacio de soluciones S***, que es el conjunto finito de todas las soluciones del problema.
- una ***función de coste f*** definida como: $f: S \rightarrow \mathrm{R}$
- un ***mecanismo de generación***, que permite el desplazamiento desde una solución a otra de su entorno, entendiendo como entorno de una solución aquellas soluciones a las que podemos llegar desde una solución dada, aplicando el mecanismo de generación.

y un conjunto de ***parámetros genéricos***, denominados **"esquema de enfriamiento"**:

- ***temperatura inicial T_0*** .
- ***ley de evolución de la temperatura***.
- ***criterio de equilibrio***.
- ***criterio de congelación***

Este conjunto de parámetros es independiente del tipo de problema que se esté resolviendo, y controla el funcionamiento del algoritmo.

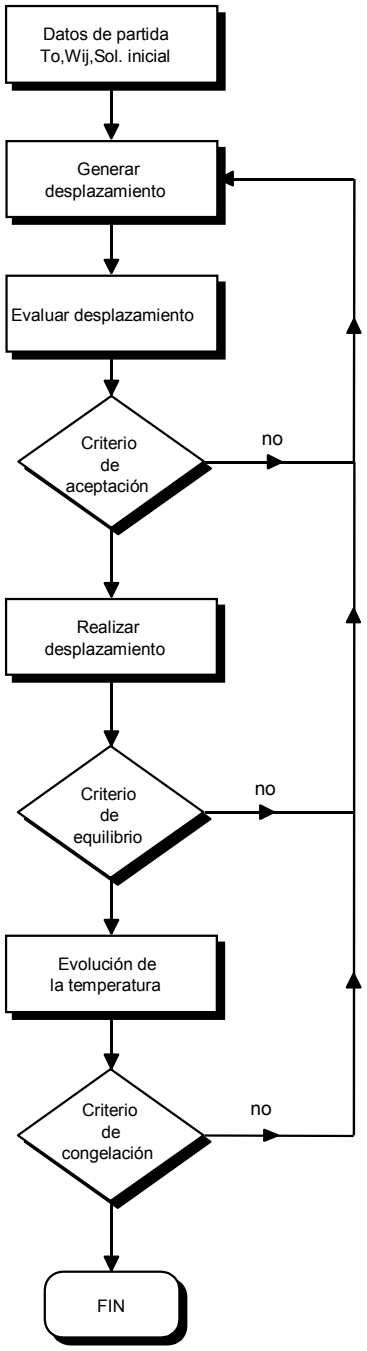


Figura 4-8. Diagrama de flujo del algoritmo Simulated Annealing

Como se indica en el diagrama de bloques de la Figura 4-8 el algoritmo realiza una serie de iteraciones a partir de una temperatura inicial. Existen diferentes técnicas para fijar esta temperatura inicial que posteriormente se analizarán.

En cada escalón de temperatura, según se indica en la Figura 4-9, se realizan una serie de iteraciones hasta que se verifica el criterio de equilibrio, condición que provoca evolucionar el valor de la temperatura mediante la ley de enfriamiento que se haya elegido. El sistema va pasando por una serie de escalones de temperatura hasta que finalmente se verifica el criterio de congelación que marca la parada en el proceso de iteración. Como se puede observar en la Figura 4-9 dada la característica del criterio de aceptación, en la fase inicial del enfriamiento, cuando la temperatura es elevada, se produce un elevado número de transiciones que provocan un incremento de coste, como se observa en la gráfica.

Progresivamente al avanzar el proceso de enfriamiento, el sistema va gradualmente evolucionando hacia estados de menor coste, hasta que en la fase final prácticamente se estabiliza en torno a la solución final.

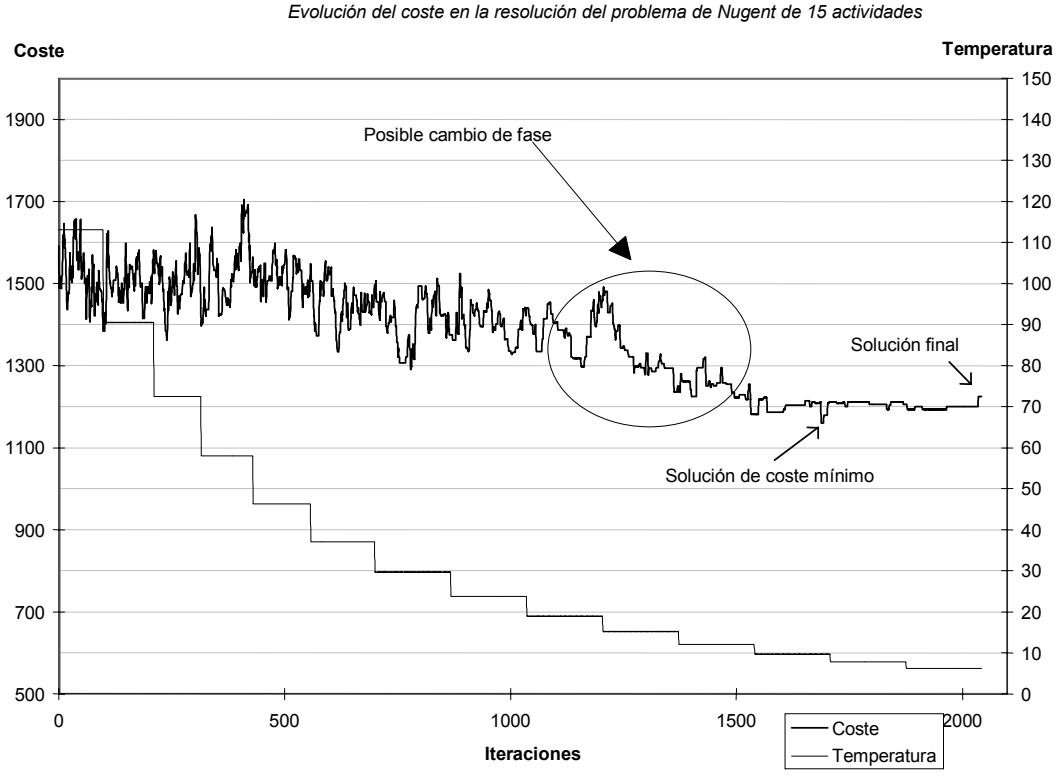


Figura 4-9. Ejemplo de evolución del coste en un algoritmo Simulated Annealing

En la Figura 4-9 se observa además otro fenómeno, que con frecuencia aparece en la utilización del algoritmo SA: dado que la implementación real de los algoritmos SA no satisface todos los requisitos teóricos de convergencia del algoritmo, en muchas ocasiones la solución final obtenida no es la mejor solución por la que ha pasado el sistema. Por ello, ciertos autores proponen diversas mejoras prácticas para la utilización del algoritmo. Una de ellas consiste en almacenar a lo largo del proceso de búsqueda la mejor solución obtenida por el sistema, de tal forma que si la solución final no es la mejor, el sistema recupera de la memoria la mejor solución por la que ha pasado. Éste sería el caso del ejemplo de la Figura 4-9.

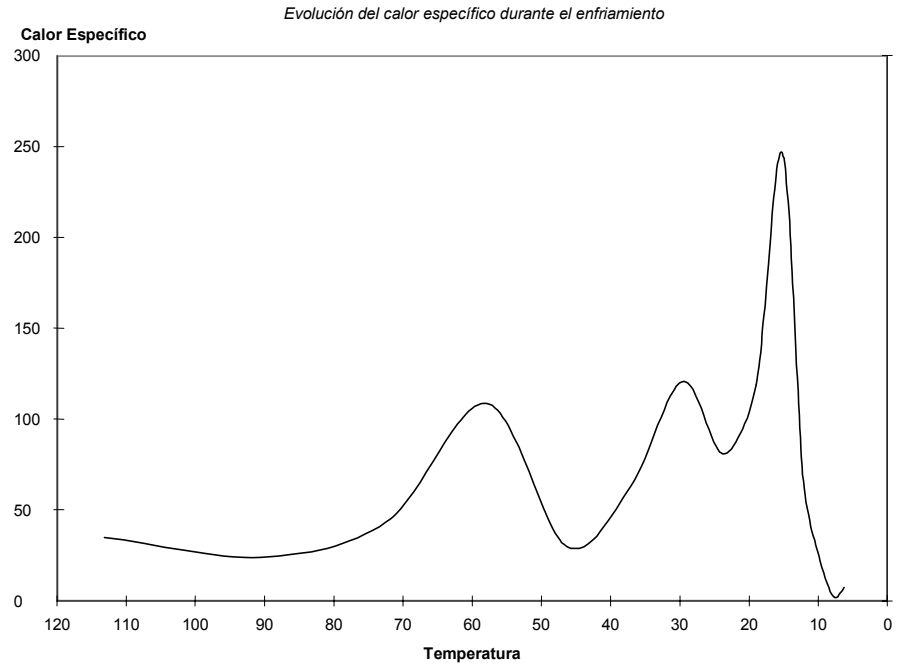


Figura 4-10. Ejemplo de evolución del calor específico en un algoritmo Simulated Annealing

En la gráfica de la Figura 4-10 se representa la evolución del calor específico calculado según la Expresión 4-1. Como se puede observar, en torno al intervalo de temperaturas [10,20] se produce un pico muy pronunciado en la evolución del calor específico, que algunos autores consideran como asociado a un "cambio de fase" que serviría como indicio de una zona en la que se debe realizar un enfriamiento más lento, pues de esta forma consideran que sería probable que el sistema encontrara "buenas soluciones".

En la Figura 4-11 se presenta un diagrama de flujo detallado de lo que sería un algoritmo SA estándar, incluyendo un contador *M* de los escalones de temperatura realizados, ya que en muchas implementaciones del algoritmo se emplea para determinar la condición de congelación o calcular la ley de enfriamiento.

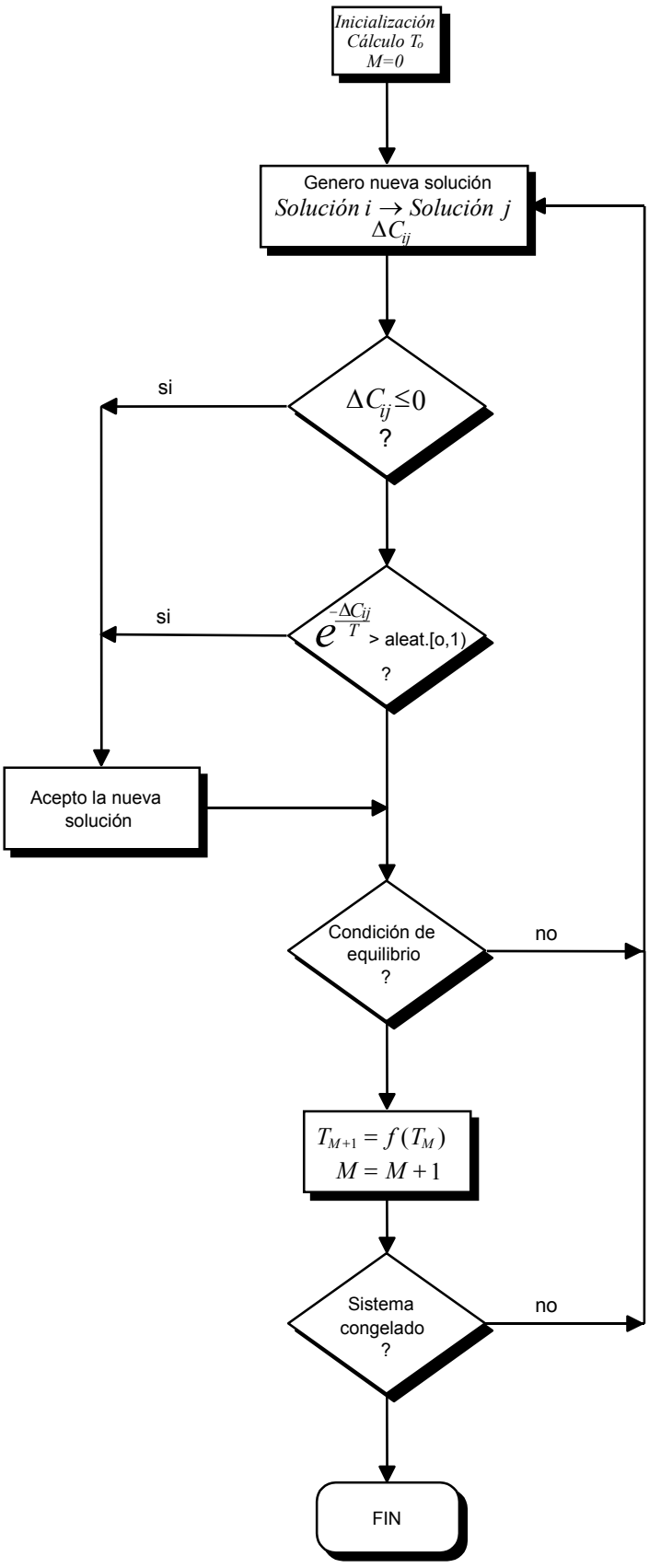


Figura 4-11. Diagrama de flujo detallado del algoritmo Simulated Annealing

4.2.2.1 Modelo matemático del algoritmo

Se puede considerar que establecido el espacio de soluciones, el mecanismo de generación y la estructura de la vecindad a una solución dada, el funcionamiento de un algoritmo SA. consiste en un intento continuo de transformación de la configuración actual en alguna de sus configuraciones vecinas. Este mecanismo es matemáticamente asimilable a una *cadena de Markov*, es decir corresponde a una secuencia de experimentos, en la que el resultado de cada experimento depende únicamente del resultado del experimento anterior. En el caso de un algoritmo SA, los experimentos equivalen a las transiciones, y es evidente que el resultado de una transición depende únicamente del resultado de la transición previa.

Una **cadena de Markov** se describe por medio de un conjunto de probabilidades condicionales $P_{ij}(k\text{-}1,k)$ para cada par de resultados (i,j) donde $P_{ij}(k\text{-}1,k)$ es la probabilidad de que el resultado del experimento k-ésimo sea j, supuesto que el resultado del experimento $(k\text{-}1)$-ésimo sea i.

Si se define como $a_i(k)$ la probabilidad de obtener como resultado i en el k-ésimo intento, se podrá calcular $a_i(k)$ mediante la siguiente expresión:

$$a_i(k) = \sum_{l=1}^{L} a_l(k-1) \cdot P_{li}(k-1,k) \qquad k = 1,2,\ldots$$

siendo L el número de posibles soluciones.

Si se denota como $X(k)$ el resultado del experimento k, entonces:

$$P_{ij}(k-1,k) = \Pr\{X(k) = j \quad | \quad X(k-1) = i\}$$

y que

$$a_i(k) = \Pr\{X(k) = i\}$$

Si las probabilidades condicionales no dependen de k, la correspondiente cadena de Markov se denomina ***homogénea***, y en caso contrario, ***no homogénea***.

En el caso de los algoritmos SA, la probabilidad condicional $P_{ij}(k\text{-}1,k)$ representa la probabilidad de que la k-ésima transición sea una transición de la configuración i a la j. De esta forma, $X(k)$ es la configuración obtenida después de k transiciones. A la vista de esto, $P_{ij}(k\text{-}1,k)$ se denomina *probabilidad de transición* y la matriz $|S| \times |S|$ formada por $P_{ij}(k\text{-}1,k)$, *matriz de transición*.

Las probabilidades de transición dependen del valor del parámetro de control T, ó *temperatura del sistema*. De esta forma si se mantiene constante T, la correspondiente cadena de Markov es homogénea y la matriz de transición $P=P(T)$ se puede definir como:

$$P_{ij}(T) = \begin{cases} G_{ij}(T)\,A_{ij}(T) & \forall j \neq i \\ 1 - \sum_{l=1, l\neq i}^{S} G_{il}(T)\,A_{il}(T) & j = i \end{cases}$$

Expresión 4-3

Según la última expresión, cada probabilidad de transición se define como el producto de dos probabilidades condicionales: la *probabilidad de generación* $G_{ij}(T)$, que proporciona la probabilidad de generar la configuración j a partir de la configuración i, y la *probabilidad de aceptación* $A_{ij}(T)$, que indica la probabilidad de aceptar la configuración j una vez generada a partir de la configuración i. Las respectivas matrices $G(T)$ y $A(T)$ se denominan respectivamente matriz de generación y matriz de aceptación.

De la Expresión 4-3 se deduce que la matriz $P(T)$ es una matriz estocástica:

$$\forall i \qquad \sum_{j} P_{ij}(T) = 1$$

El utilizar la Expresión 4-3, supone una generalización de la idea de algoritmo SA, ya que éste sería un caso particular de un tipo de algoritmos más general, siendo en este caso $G_{ij}(T)$ una distribución uniforme de la vecindad (las transiciones de realizan escogiendo aleatoriamente una configuración vecina j de la configuración actual i), y $A_{ij}(T)$ está dada por el algoritmo Metropolis.

Puesto que el parámetro *T* varía en el transcurso del algoritmo se pueden distinguir dos tipos de formulaciones:

A) ***Algoritmo homogéneo***: el algoritmo es descrito por una serie de cadenas de Markov homogéneas. Cada cadena de Markov es generada para un valor fijo de T, siendo T decrementada entre cadenas de Markov consecutivas. En la Figura 4-12 se puede observar la evolución del coste durante el proceso de enfriamiento. Se trata de un algoritmo homogéneo, ya que el enfriamiento se realiza a traves de una serie de escalones, durante los cuales la temperatura permanece constante.

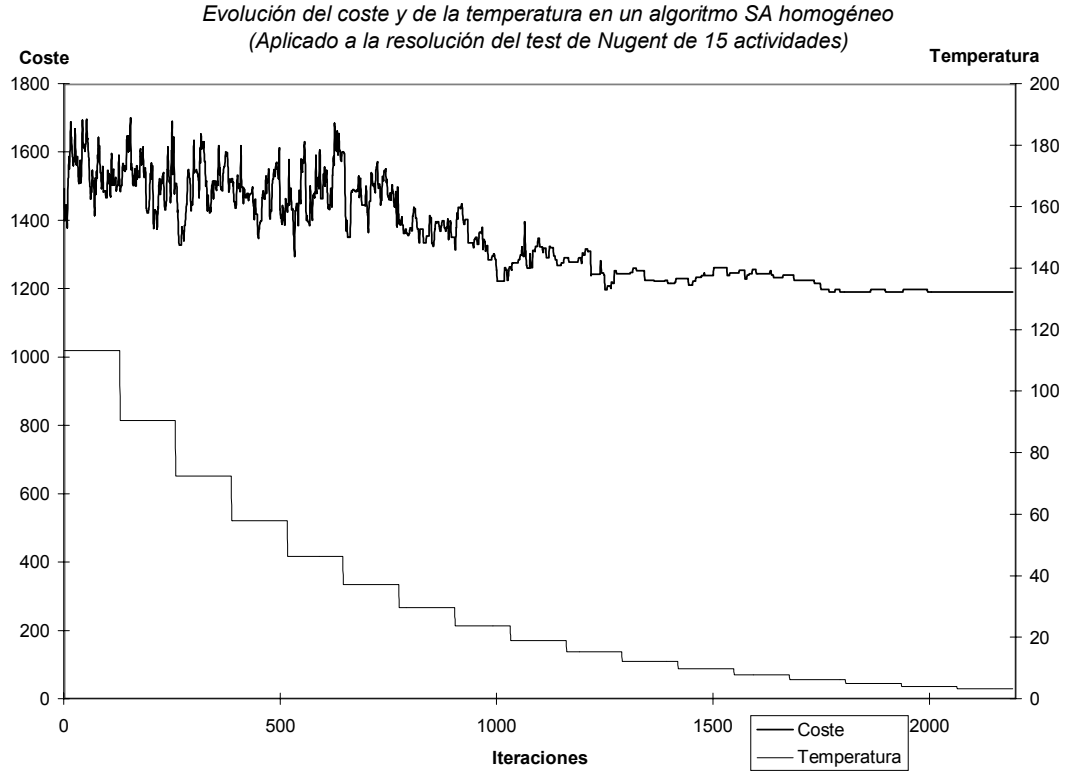


Figura 4-12. Ejemplo de evolución del coste en un algoritmo Simulated Annealing Homogéneo

B) ***Algoritmo no homogéneo***: el algoritmo es descrito por una única cadena de Markov. El valor de T es decrementado entre transiciones consecutivas, lo que equivale a considerar que se realiza el enfriaiento a través de una serie de escalones de longitud unidad. En la Figura 4-13 se presenta la evolución que sufre el coste, para el mismo problema tipo de la Figura 4-12. Como se puede

observar, la curva de enfriamiento es continua, a diferencia del caso anterior. En ambos ejemplo se ha utilizado la ley de enfriamiento $T_M = c \cdot T_{M-1}$

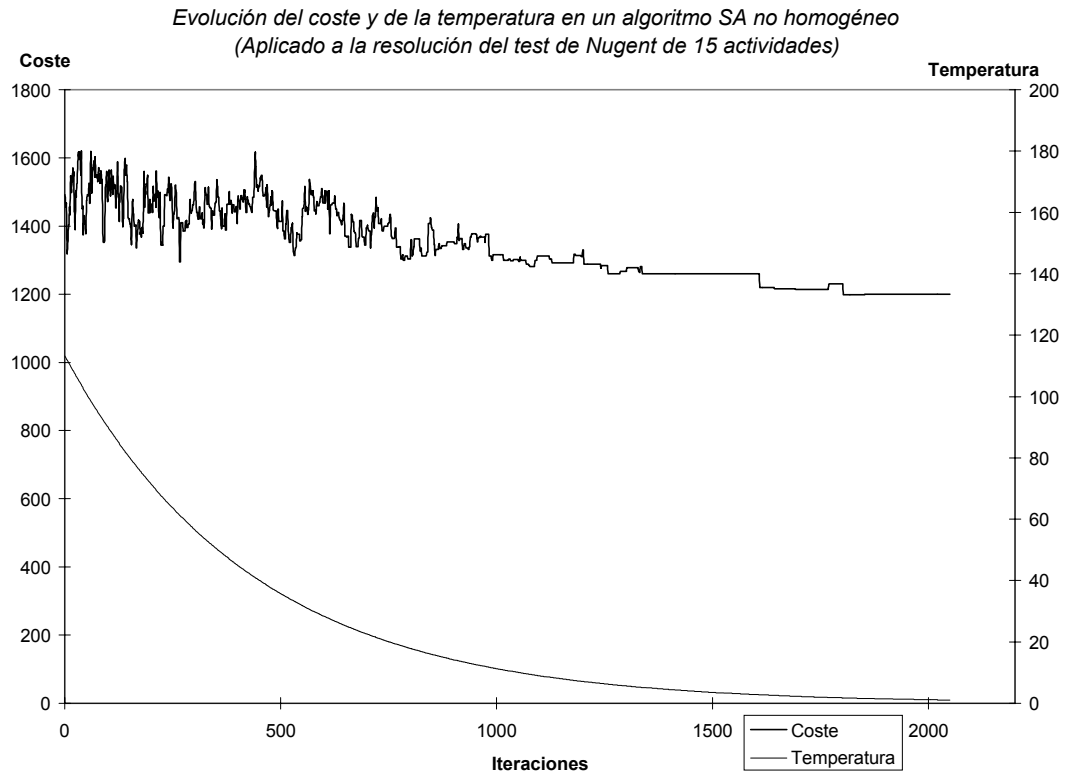


Figura 4-13. Ejemplo de evolución del coste en un algoritmo Simulated Annealing no Homogéneo

El algoritmo SA obtiene un mínimo global si, después de un número de transiciones (normalmente bastante elevado), supóngase K, se verifica la siguiente relación:

$$\Pr\left\{ X(K) \in S_{opt} \right\} = 1$$

donde S_{opt} es el conjunto de configuraciones de los mínimos globales.

Se puede demostrar que el algoritmo homogéneo presenta convergencia asintótica es decir $\lim_{K \to \infty} \Pr\left\{ X(K) \in S_{opt} \right\} = 1$ si:

- Cada cadena de Markov es de longitud infinita.
- Se verifican ciertas condiciones en las matrices $A_{ij}(t)$ y $G_{ij}(t)$

- $\lim_{l\to\infty} T_l = 0$ donde T_l es el valor de la temperatura en la l-ésima cadena de Markov.

Para el algoritmo no homogéneo se pueden establecer otro conjunto de condiciones que garantizan la convergencia.

Esencial para la existencia de convergencia es el hecho de que exista bajo determinadas condiciones una "*distribución estacionaria*" de la cadena de Markov homogénea. La "*distribución estacionaria*" es la distribución de probabilidad de las configuraciones tras un número infinito de transiciones. Posteriormente, se indicará como afecta este concepto a la implementación práctica del algoritmo.

La distribución estacionaria está definida por el vector q, cuya i-ésima componente está dada por:

$$q_i = \lim_{k\to\infty} \Pr\{ X(k) = i \quad | \quad X(0) = j \} \text{ para un } j \text{ arbitrario}$$

Esto es equivalente a decir que:

$$q = \lim_{k\to\infty} a(0)^T P^k$$

Expresión 4-4

donde P^k es la k-ésima potencia de la matriz de transición y $a(0)^T$ representa el vector traspuesto con la distribución inicial de probabilidad, es decir $a(0)=(a_i(0))$, con $i \in S$ cumpliéndose además que

$$\forall i \in S \qquad a_i(0) \geq 0, \qquad \sum_{i\in S} a_i(0) = 1$$

La prueba de la convergencia del algoritmo se basa en los siguientes puntos:

1. Justificación de la existencia de la distribución estacionaria, para lo cual la cadena de Markov debe ser aperiódica e irreducible.

   El vector q, queda determinado por las siguientes ecuaciones:

$$\forall i \quad : \quad q_i > 0, \quad \sum_i q_i = 1$$
$$\forall i \quad : \quad q_i = \sum_j q_j P_{ji}$$

Obsérvese que el vector q es el vector propio de la matriz P correspondiente al valor propio 1.

Una matriz de Markov es *irreducible* si y sólo si para cualquier par de configuraciones (*i*,*j*) hay una probabilidad positiva de alcanzar la configuración *j* desde *i*, en un número finito de transiciones, es decir:

$$\forall i, j \quad \exists n \quad / \quad 1 \le n < \infty \quad \wedge \quad \left(P^n\right)_{ij} > 0$$

Además se define una matriz de Markov como aperiódica si y sólo si para cualquier configuración i del espacio de soluciones S el máximo común divisor de todos los enteros mayores o iguales a 1, tales que $\left(P^n\right)_{ii} > 0$ sea igual a 1.

Una expresión de la distribución estacionaria es la siguiente:

$$q_i(T) = \frac{e^{\frac{-C(i)}{T}}}{\sum_{j \in S} e^{\frac{-C(j)}{T}}}$$

Expresión 4-5

2. Una vez demostrada la existencia de la distribución estacionaria, se debe justificar que para T decrecientes $q(T)$ converja a la distribución uniforme del conjunto de configuraciones de los mínimos globales, es decir:

$$\lim_{T \to 0} q(T) = \pi$$

Expresión 4-6

donde el vector π de dimensión $|S|$ (que indica la cardinalidad del conjunto *S*, es decir, el número de elementos que lo componen) está definido como:

$$\pi_i = \begin{cases} |S|^{-1} & si \quad i \in S_{opt} \\ 0 & en\ caso\ contrario \end{cases}$$

siendo S_{opt} el conjunto de las configuraciones globales mínimas.

Combinando la Expresión 4-3 y la Expresión 4-6 la condición que se obtiene es:

$$\lim_{T \to 0} \left(\lim_{k \to \infty} \Pr\{ X(k) = i \} \right) = \pi_i$$

4.2.2.2 Implementación del algoritmo

En cualquier implementación del algoritmo, la convergencia asintótica puede ser solamente aproximada. De esta forma, aunque el algoritmo es asintóticamente un algoritmo de optimización, **cualquier implementación del mismo resulta en un algoritmo de aproximación**. Como muestra de estas aproximaciones, es evidente que el número de transiciones en cada escalón de temperatura sólo puede ser finito, y que la condición de que $\lim_{k \to \infty} T_k = 0$ sólo puede ser aproximada con un número finito de valores de T. Debido a estas aproximaciones, no se puede garantizar que el algoritmo converja al mínimo global con probabilidad 1.

La implementación habitual del algoritmo es mediante una secuencia de cadenas de Markov homogéneas de longitud finita con valores decrecientes del parámetro de control o temperatura.

Como ya se ha mencionado previamente en este capítulo, los parámetros que van a definir el comportamiento del algoritmo, constituyen el esquema de enfriamiento, y son:

- Temperatura inicial.
- Temperatura final (criterio de congelación).
- Condición de equilibrio (longitud de la cadena de Markov).

- Ley de evolución de la temperatura.

El concepto de "quasi-equilibrio" es determinante para la construcción de muchos esquemas de enfriamiento. Si L_k es la longitud de la k-ésima cadena de Markov, entonces se dice que el algoritmo SA está en quasi-equilibrio en T_k si $a(L_k,T_k)$ está próxima a $q(T_k)$ (distribución estacionaria). Este concepto de proximidad es uno de los puntos donde los esquemas de enfriamiento difieren unos de otros.

4.2.2.2.1 Cálculo de la temperatura inicial

En la bibliografía se han presentado diferentes sistemas para determinar la temperatura inicial. Por ejemplo escoger T_0 de tal forma que prácticamente sean aceptadas todas las transiciones, es decir que $\exp(-\Delta C/T_0)\approx 1$. Kirkpatrick y otros [15] en su artículo proponen una regla empírica, escoger un valor grande de T_0 y realizar cierto número de transiciones. Si se define el coeficiente de aceptación χ como el cociente entre el número de transiciones aceptadas y el número de transiciones intentadas, entonces si el valor obtenido de χ es menor de una cierta cantidad, por ejemplo 0.8, entonces se dobla el valor de T_0, procediendo así sucesivamente hasta verificar la condición impuesta.

Otros autores como Kouvelis [17], utilizan la expresión $T_0 = \dfrac{\overline{\Delta C}^{(+)}}{\ln(\chi_0^{-1})}$ donde se calcula el incremento de coste medio de una serie de transiciones, y se prefija de antemano un valor de χ.

4.2.2.2.2 Temperatura final (criterio de congelación).

La condición de parada del algoritmo se puede establecer de diferentes maneras. Una de ellas es fijar un número determinado de valores de T para los cuales se ejecuta el algoritmo. Otra condición de parada puede ser, si fijado un determinado número de escalones de temperatura, las configuraciones obtenidas al finalizar cada una de ellos son iguales entonces se decide la parada del algoritmo. Otro criterio puede ser, el establecer un coeficiente de aceptación χ mínimo, de tal forma que en caso de no alcanzarse se detiene el algoritmo.

4.2.2.2.3 Longitud de la cadena de Markov (condición de equilibrio).

La elección más sencilla para L_k, longitud de la *k*-ésima cadena de Markov, es elegir un valor dependiente (de forma polinómica) del tamaño del problema. Así, L_k es independiente de *k*. Existen propuestas más elaboradas, en las que se establece que para cada valor T_k se debería realizar un número mínimo de transiciones. Entonces en este caso se determina L_k de tal forma que el número mínimo de transiciones sea η_{min} (siendo η_{min} un número fijo). Sin embargo al aproximarse *T* al valor 0, las transiciones son aceptadas cada vez con una probabilidad menor de tal forma que $L_k \to \infty \quad si \quad T_k \to 0$. Por ello L_k debe estar acotado superiormente por una cierta constante $\overline{L}$, para evitar cadenas de Markov excesivamente largas para bajos valores de *T*.

En esta línea Kirkpatrick [15] propone $\overline{L} = n$ siendo *n el* número de variables a resolver. Otra opción es tomar $\overline{L} = m \cdot R$ como un múltiplo del tamaño de la clase de vecindad.

Otro sistema de definir la longitud de la cadena de Markov, es establecer el concepto de "época". Se define una "época" como un número de transiciones con un número fijo de transiciones aceptadas, siendo el coste de una "época" el coste de la última configuración de la misma. En el momento que el coste de una "época" está a una determinada distancia de las "épocas" precedentes, la cadena de Markov se acaba. De esta forma, la condición de término de una cadena de Markov, y consecuentemente su longitud, queda ligada a las fluctuaciones de la función de coste observada en esa cadena.

4.2.2.2.4 Ley de evolución de la temperatura.

Existen diferentes leyes de evolución de temperatura que se clasifican en función de su complejidad. Un primer grupo agruparía a los esquemas sencillos de enfriamiento. Dentro de este grupo, aparacen en la bibliografía expresiones de la forma:

$$T_M = c \cdot T_{M-1}$$

Expresión 4-7

oscilando el valor *c* entre 0,50 y 0,99.

En virtud de la definición anterior también se puede escribir:

$$T_M = c^M T_0$$

con lo que se obtiene una curva de enfriamiento de tipo potencial, produciéndose un enfriamiento más rápido a temperaturas elevadas, que posteriormente se hace más lento a bajas temperaturas.

Otro tipo de ley de enfriamiento que aparece en la bibliografía es la de establecer a priori un número determinado de escalones, por ejemplo un número K, de tal forma que se mantiene el salto de temperatura constante de forma que se tiene la siguiente expresión:

$$T_M = \frac{K - M}{K} T_0$$

Expresión 4-8

Otras leyes de enfriamiento utilizadas por diversos autores pueden ser las siguientes:

$$T_{M+1} = \frac{T_M}{1 + cT_M}$$

Expresión 4-9

$$T_{M+1} = \frac{T_0}{1 + cM}$$

Expresión 4-10

Resumiendo, se puede concluir, que respecto a la temperatura inicial, se propone que inicialmente todas las transiciones sean prácticamente aceptadas. Sin embargo existen dos tendencias en cuanto a la elección de la ley de evolución de la temperatura y las longitud de la cadena de Markov o condición de equilibrio.

Estas dos alternativas se podrían clasificar como:

- Alternativa A: se establece una longitud variable de la cadena de Markov, y un decremento constante en la ley de evolución de la temperatura.

- Alternativa B: se establece una longitud fija de la cadena de Markov, y un decremento variable de la temperatura.

En este caso, se entiende como fija o variable, la independencia o dependencia respecto de la evolución del algoritmo.

En cuanto a la temperatura final o condición de congelación, también se observan dos alternativas. La primera, usada en los esquemas sencillos de enfriamiento, consiste en establecer la no aceptación de un número mínimo de transiciones en una secuencia de cadenas de Markov, mientras que la otra alternativa consistiría en realizar extrapolaciones del coste medio de las configuraciones en un determinado número de cadenas de Markov consecutivas.

4.2.3 Implementación del Simulated Annealing en la Reconstrucción Geométrica

Hasta aquí se ha visto la teoría de un algoritmo SA genérico. Es fundamental tener claro el concepto de funcionamiento de un algoritmo SA, al margen de sus parámetros.

La filosofía del algoritmo SA consiste en modificar suavemente todas las variables, durante muchas iteraciones, de modo que se acepten soluciones que mejoren la mejor solución en curso para ir aproximándose a la mejor solución global, con la **posibilidad de que se admitan soluciones no válidas** (peores que la mejor solución en curso) y así tener la **capacidad de salir de un mínimo local**. Esta posibilidad de aceptar soluciones "malas" va disminuyendo a medida que avanza el algoritmo.

El comportamiento del algoritmo al principio parece un tanto *alocado*, aceptando muchas soluciones *malas*, pero ahí radica su potencia pues es capaz de remontar un mínimo local para buscar el mínimo global. Recordemos que lo que buscamos es la

mejor solución, no una solución buena. Conforme va avanzando, el algoritmo va centrándose más en una solución, de modo que consigue afinarla al máximo.

Una vez comprendido el algoritmo, es el momento de aplicarlo al objetivo de este proyecto, esto es, la reconstrucción 3D. Recordemos en qué punto estamos del proyecto global:

Tenemos un boceto 2D que es la proyección de algún objeto 3D, por ejemplo la Figura 4-14. Este dibujo ha sido procesado y limpiado, de modo que cumple una serie de requisitos vistos anteriormente (ver capítulo 1.2.2 más atrás en pag.14 y capítulo 2.3 en pag.24). Representa TODOS los vértices y aristas del objeto. Cada uno de los vértices contiene información de su posición *(X, Y, Z)* en el espacio, siendo *Z*=0 por estar en el plano. Ahora pretendemos buscar valores de *Z* en cada vértice de modo que al final, cumpliendo las regularidades, el objeto alcance una representación 3D posible. Esto es lo que llamamos *"proceso de inflado"* (ver Figura 4-15), en que vemos como los vértices van despegándose del papel para alcanzar su sitio correcto en el espacio.

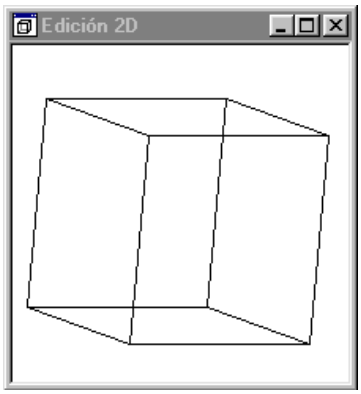


Figura 4-14. Boceto 2D de partida

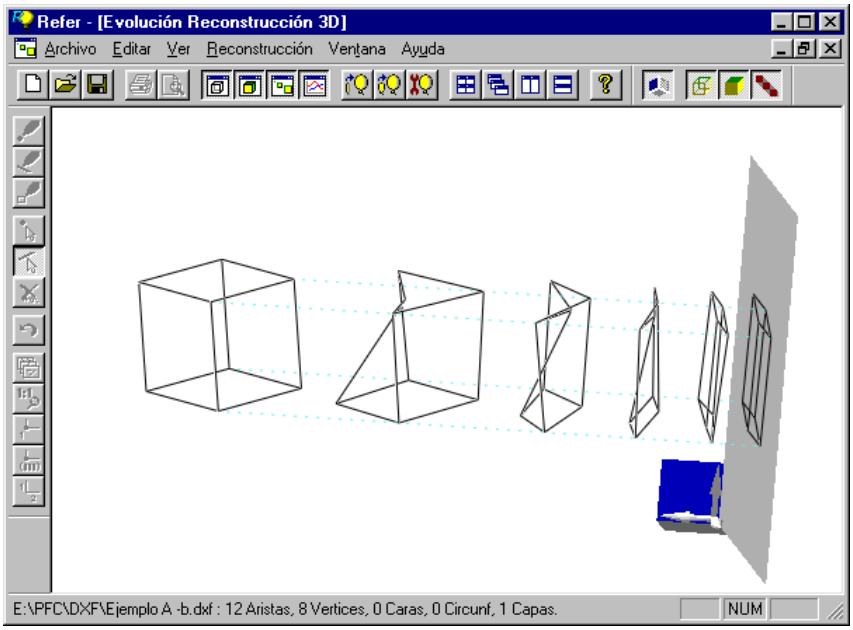


Figura 4-15. Ejemplo del proceso de inflado

Recordemos cuáles son los parámetros que necesita el algoritmo S.A:

Parámetros específicos	*Parámetros genéricos o "esquema de enfriamiento"*
• Espacio de soluciones S • Función de coste • Mecanismo de generación	• Temperatura Inicial T_0 • Ley de evolución de la temperatura • Criterio de equilibrio • Criterio de congelación o temperatura final

Ahora veremos paso a paso qué criterios de diseño se han elegido, y por qué.

4.2.3.1 Espacio de soluciones S

Así pues, las variables que manejaremos serán las *Z* de los vértices. Por tanto, dado un instante ***i*,** definiremos la solución *i* como:

$S_i = \{Z_{i0}, Z_{i1}, Z_{i2}, \ldots, Z_{in}\}$, siendo *n* el nº de vértices

La solución inicial S_o será aquella en que el objeto está completamente *desinflado*, esto es, todas sus Z valen 0:

$S_0 = \{0, 0, 0, \ldots, 0\}$

La solución final será aquella en que el objeto esté *inflado*, de modo que se cumplan las regularidades.

4.2.3.2 Mecanismo de generación

El mecanismo de generación nos permite pasar de una solución a otra **muy próxima** de su entorno. Dada una solución S_i , obtendremos la solución S_{i+1} modificando (incrementando o decrementando) alguna de las variables *Z* de la solución S_i. La elección de la variable a modificar será aleatoria, al igual que la decisión de incremento o decremento.

Es decir, dada la solución $S_i = \{Z_{i0}, Z_{i1}, Z_{i2}, \ldots, Z_{in}\}$, y un incremento de *Z* llamado ΔZ, el conjunto de soluciones candidatas a ser S_{i+1} serán:

$$\{Z_{i0} + \Delta Z, Z_{i1}, Z_{i2}, \ldots, Z_{in}\}$$
$$\{Z_{i0} - \Delta Z, Z_{i1}, Z_{i2}, \ldots, Z_{in}\}$$

$$\{Z_{i0}, Z_{i1} + \Delta Z, Z_{i2}, \ldots, Z_{in}\}$$
$$\{Z_{i0}, Z_{i1} - \Delta Z, Z_{i2}, \ldots, Z_{in}\}$$

$$\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots$$

$$\{Z_{i0}, Z_{i1}, Z_{i2}, \ldots, Z_{in} + \Delta Z\}$$
$$\{Z_{i0}, Z_{i1}, Z_{i2}, \ldots, Z_{in} - \Delta Z\}$$

de las cuales se elegirá una de forma totalmente aleatoria. Esta forma de operar hace que el comportamiento del algoritmo sea **no determinista**: la ejecución del mismo ejemplo dos veces no tiene porqué dar el mismo resultado en las dos ocasiones.

La elección del ΔZ es un aspecto crítico en el diseño del algoritmo. De él depende que la aproximación a la solución sea más o menos rápida. Además determina la precisión del resultado. Veamos:

- Si el ΔZ es muy pequeño tenemos mucha precisión, pero el algoritmo avanza muy lento hacia la solución, necesitando más iteraciones.

- Si el ΔZ es demasiado grande avanzaremos rápido hacia la solución, pero perderemos precisión, lo que nos puede impedir alcanzar la mejor solución.

Un ejemplo del segundo caso lo tenemos en la Figura 4-16: Supongamos que estamos en la solución S_i, y que con el ΔZ aplicado pasamos a la solución S_{i+1}. Como se puede ver la solución con coste menor sería alguna situada entre ambas, pero con el ΔZ actual no la podremos alcanzar.

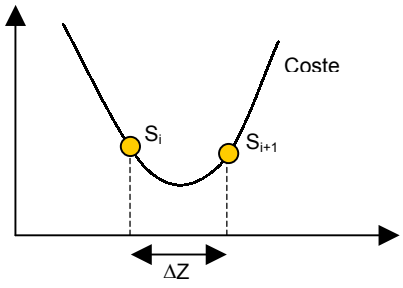


Figura 4-16. Ejemplo de ΔZ demasiado grande en SA

En la práctica parece buena la opción de establecer un ΔZ que sea muy pequeño pero proporcional al tamaño del objeto, dejando que el algoritmo se ejecute durante un número elevado de iteraciones. En el programa se deja abierto este parámetro para que pueda ser modificado. Empíricamente se ha demostrado

como bueno un ΔZ que sea del orden del 1% del rango de los datos[3] del objeto 2D. En la Figura 4-17 podemos ver cómo se ajusta este parámetro.

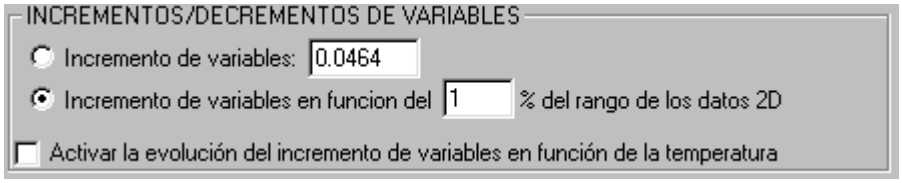


Figura 4-17. Ajuste del Incremento de variables en algoritmo SA

También podría considerarse que el ΔZ no fuera constante, sino que fuera variable, de modo que tuviera un valor elevado al principio para acelerar la llegada a una solución buena y que fuera disminuyendo para que al final se tuviera la suficiente precisión. En mi proyecto he optado por poder ajustar el ΔZ en función de la temperatura que, como veremos más adelante, comienza elevada para ir descendiendo. No está clara la razón, pero en la práctica parece que el algoritmo no mejora sus resultados con esta última opción activada. Este sería un aspecto a estudiar con más detalle.

4.2.3.3 Función de coste

La función de coste servirá para evaluar cómo de buena es una solución. Esta función es necesaria para comparar dos soluciones, y decidir cuál es la mejor.

Este parámetro es muy importante, y en este proyecto es especialmente delicado, ya que no siempre es posible escoger entre dos soluciones. La razón está en el modo en que el ser humano es capaz de reconocer un boceto 2D y reconstruirlo mentalmente generando un modelo 3D. Intuitivamente nos podemos dar cuenta que un mismo boceto puede tener varias soluciones perfectamente válidas. Esto está ampliamente detallado en el Capitulo 3.1, pag. 27.

[3] Defino el *Rango X del objeto 2D* como la máxima distancia entre los componentes X de los vértices del objeto 2D, y defino el *Rango Y del objeto 2D* como la máxima distancia entre los componentes Y de los vértices del objeto 2D.
Así pues, defino el *Rango de los datos del objeto 2D* como el mínimo entre el *Rango X* y el *Rango Y*.

Como ya vimos, nos serviremos de las Regularidades para poder calcular el coste de una solución. Ahora comprenderemos hasta qué punto es necesario tener bien definidas estas Regularidades, y las relaciones e interacciones entre todas ellas.

Hay que tener en cuenta que en este proyecto solo emplearemos dos regularidades, ambas explicadas detalladamente en el Capítulo 5, pag. 85:

- *Mínima Desviación Estándar de los Ángulos*

- *Paralelismo de Líneas*

Se pretende que el algoritmo haga lo siguiente: supongamos que estamos en la solución S_i , y mediante el mecanismo de generación hemos encontrado la solución S_{i+1}. Ahora hay que decidir con cuál de las dos soluciones nos quedamos. Para tomar esta decisión hay que saber cuál de las dos es mejor, es decir, cuál de las dos "cumple más" las regularidades. En este momento calculamos el coste de cada regularidad, según vimos, y lo tenemos en cuenta en función de la importancia que tenga cada regularidad (hay regularidades que parecen más importantes que otras, como se verá más adelante[4]). Es obvio que cada regularidad tendrá su propio orden de magnitud, y por tanto no se puede operar con varias regularidades entre sí. Además, conceptualmente tampoco tiene ningún sentido, igual que tampoco sumaríamos peras con manzanas.

Así pues, vemos que aquí tenemos un punto muy importante a estudiar, que es el tema de **la ponderación y las relaciones entre las regularidades**. Este asunto debería tratarse con profundidad en estudios posteriores, saliéndonos del ámbito de este proyecto.

Sin embargo, dado que sólo se utilizan dos regularidades, se han podido conseguir resultados francamente buenos con un método sencillo y eficaz, que consiste simplemente en considerar el coste de una solución como la suma de los costes de las dos regularidades, y ponderar cada una de las regularidades para, de algún modo, homogeneizar o normalizar sus magnitudes.

[4] Ver *Capítulo 5. Regularidades*

Esto es, **definimos el coste de una solución como la suma de los costes de cada una de las regularidades, multiplicado por un factor de corrección**.

Podemos ver la implementación de la función **CalculaCoste** en el *Capítulo 5.6 Implementación de la función objetivo*

¿Hasta qué punto es buena esta opción? Empíricamente se demuestra que el sistema funciona considerando que la regularidad del *Paralelismo Lineal* es del orden del 0.01% de la *Desviación Estándar de Angulos*. Podemos ver cómo queda en el programa (Figura 4-18):

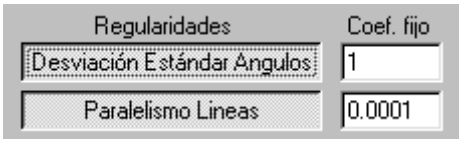


Figura 4-18. Ponderación de Regularidades

4.2.3.4 Temperatura inicial. Método Refer.

En un algoritmo SA típico se tomaría una temperatura inicial T_0 de modo que se cumpla que en las primeras iteraciones se acepten prácticamente todas las transiciones, es decir, se acepten casi todas las soluciones nuevas. Sin embargo, en esta aplicación el método propuesto por Kirkpatric[5] no funciona correctamente; debe modificarse para que pueda ser operativo.

Replanteémonos nuestro objetivo: inicialmente partimos de un objeto 3D que se caracteriza por tener todos sus vértices en el mismo plano (Z=0). Queremos inflarlo de modo que los vértices se separen del plano, para que el objeto tenga *volumen*, y siempre con la condición de que se sigan los criterios de las regularidades. Es decir, las regularidades se deben de cumplir *más* al final que al principio: la solución debe ir mejorando.

Y aquí es donde se nos plantea la paradoja: hay algunas regularidades que en la solución inicial ya se cumplen al 100%, y por tanto, cualquier modificación en alguna de las variables lleva a una solución que empeora dichas regularidades. Así que estas regularidades nos impiden inicialmente que el algoritmo avance.

[5] Ver el apartado *4.2.2.2.1Cálculo de la temperatura inicial* en la página 63.

Un ejemplo muy claro de esto lo tenemos con la regularidad del *Paralelismo de Líneas*, que básicamente consiste en que si dos aristas son paralelas en el plano 2D entonces deben serlo también en el espacio 3D; si no lo son, esto tiene un coste. Supongamos que tenemos el boceto del cubo (Figura 4-14). En esta figura todas las aristas que son paralelas en el plano 2D también lo son en el espacio 3D. Por tanto su coste es 0. Es decir, el coste de esta regularidad en la solución inicial es nulo. Si ahora el algoritmo modifica cualquier variable (cualquier vértice), vemos que en el espacio 3D al menos una arista ha dejado de tener la misma relación de paralelismo que tenia anteriormente; y ahora si que tendrá un coste. Por tanto esta solución encontrada será peor que la anterior, y no se aceptará.

Si el algoritmo se ejecutara sólo con esta regularidad activada, veríamos que no avanza, dando la primera solución como buena. Es más, no sólo es buena, es la mejor.

Aquí vemos nuevamente hasta qué punto es necesario comprender el funcionamiento de las regularidades y sus interrelaciones.

En la práctica vemos que el empeoramiento de una regularidad se compensa con la mejora de la otra, y además no olvidemos que el criterio Metropolis admite *soluciones malas*, dando opción al avance en la búsqueda del óptimo.

El método Kirkpatric parte de la solución inicial para calcular la temperatura inicial. En esta aplicación no podemos partir de la solución inicial, porque la solución inicial puede ser *demasiado buena*, por las razones vistas anteriormente.

El método Juanvi-Kirkpatric se basa en modificar aleatoriamente algunas de las variables hasta que se tenga una solución con coste no nulo. En este momento se le da a la temperatura T_o un valor del orden del coste de la solución encontrada[6], y se comprueba en este primer escalón de temperatura cual es el coeficiente de aceptación χ.

Definimos el **coeficiente de aceptación** como el cociente entre el número de transiciones aceptadas y el número de transiciones intentadas.

[6] Concretamente del orden de la décima parte del coste de la solución encontrada.

Si dicho coeficiente χ es menor que una cierta cantidad entonces se dobla el valor de T_o, procediendo así sucesivamente hasta verificar la condición impuesta. Empíricamente se considera bueno un coeficiente de aceptación superior al 80%, como se puede ver en la Figura 4-19:

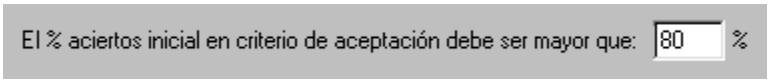


Figura 4-19. Coeficiente de aceptación

4.2.3.5 Criterio de congelación o Temperatura final

El criterio de congelación es la condición de parada del algoritmo. Por un lado se debe poner una cota superior al número de iteraciones del algoritmo, para asegurarse que éste termina en tiempo finito. Pero además podemos considerar que se puede llegar a la solución buena antes de agotar todas las iteraciones. Por eso hemos implementado dos criterios de congelación:

- **Criterio de congelación** propiamente dicho. Es simplemente un valor arbitrario a gusto del usuario, que indica el máximo de escalones de temperatura que recorrerá el algoritmo. Nosotros hemos obtenido buenos resultados con un valor de 200 escalones de temperatura como máximo. Ver figura siguiente.

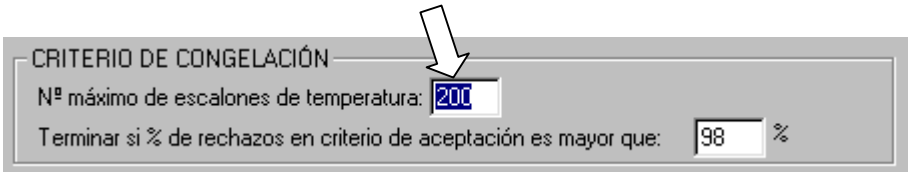


Figura 4-20. Temperatura final o Criterio de congelación propiamente dicho

- Una segunda condición, que consiste en el siguiente razonamiento: si llegamos a un punto en que prácticamente es imposible conseguir una solución mejor que la solución actual, podemos considerar que ya tenemos la solución buena. Esto sucederá cuando el porcentaje de aceptaciones de soluciones mejores sea muy bajo, o lo que es lo mismo, cuando el porcentaje de rechazos de soluciones mejores sea muy elevado. Nosotros consideramos que este porcentaje de rechazos debe ser del orden del 98%. Ver Figura 4-20

4.2.3.6 Condición de equilibrio

La condición de equilibrio es el parámetro que define cuánto tiempo (o mejor dicho, cuantas iteraciones) estará el algoritmo buscando soluciones en un escalón de temperatura. Si es muy grande se eleva el número de cálculos, mientras que si es reducido se corre el riesgo de ser insuficiente.

Para la condición de equilibrio, hemos implementado dos condiciones:

- **Condición de equilibrio** propiamente dicha, que dependerá de forma polinómica del número de variables.

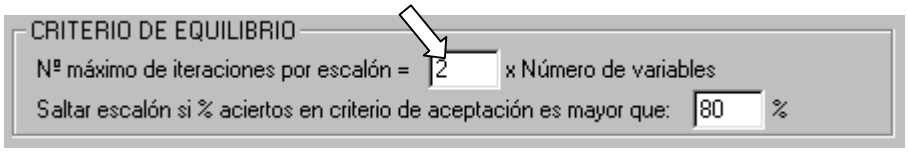


Figura 4-21. Condición de equilibrio propiamente dicha

- Una segunda condición, que consiste en el siguiente razonamiento: si estamos en un escalón y conseguimos un *porcentaje muy elevado* de soluciones que mejoran la solución actual, podemos considerar que estamos en el camino correcto para encontrar la solución buena, y en ese caso parece lícito saltar ya al escalón siguiente para acelerar el proceso de búsqueda. Ahora bien, ¿qué entendemos por un *porcentaje muy elevado*? En la práctica parece bueno un porcentaje del 80%. Ver figura siguiente:

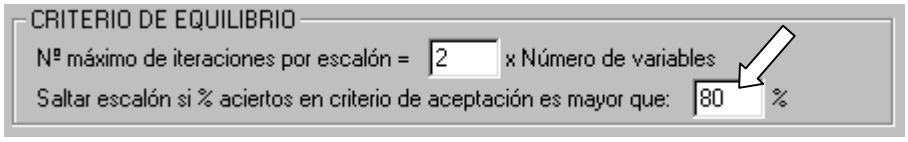


Figura 4-22. Condición de equilibrio. Segunda opción dependiente de % aciertos

Es obvio que en el caso que no interese esta segunda opción, basta con poner un valor del 100%.

4.2.3.7 Ley de evolución de la temperatura

La ley de evolución de la temperatura es importante en el sentido que afecta a la propiedad del algoritmo de aceptar soluciones malas. Mientras la temperatura sea alta,

habrá mas probabilidad de coger soluciones malas y, por tanto, de escapar de mínimos locales.

Existen diferentes leyes de evolución de temperatura. Nosotros hemos implementado dos leyes que se pueden escoger en el mismo programa:

- **Evolución de la temperatura constante**. Se debe establecer a priori un número determinado de escalones. En nuestra implementación cogeremos será el nº máximo de escalones. Se ha programado según la fórmula siguiente:

$$Temperatura = \frac{NumeroMaximoEscalones - ContadorEscalones}{NumeroMaximoEscalones} TemperaturaInicial$$

  siendo *ContadorEscalones* una variable cuyo valor equivale al escalón actual; es el nº de escalones por los que hemos pasado.

- **Evolucion de la temperatura según una curva de tipo potencial.** Es un método sencillo en el que en cada iteración la temperatura se multiplica por un factor *c* que oscila entre 0,5 y 0,99. La práctica nos indica que un valor bueno puede ser c=0.96, como podemos ver en la Figura 4-23.

$$Temperatura_M = c \cdot Temperatura_{m-1}$$

  Este tipo de evolución de temperatura produce un enfriamiento más rápido a temperaturas elevadas, y un enfriamiento más lento a temperaturas bajas.

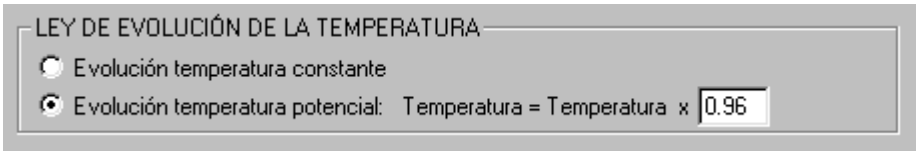


Figura 4-23. Ley de evolución de la temperatura

Después de probar ambos métodos con nuestros ejemplos, hemos apreciado diferencias significativas en los resultados que nos indican que el método de la evolución según una curva de tipo potencial es mejor. El método de la evolución constante hace que el algoritmo tarde mucho en encontrar el camino correcto hacia la

solución buena, y cuando encuentra este camino ya lleva muchas iteraciones , por lo que apenas le queda tiempo para afinar la solución buena.

4.2.3.8 Programación del algoritmo Simulated Annealing

A continuación presentaremos el código del algoritmo Simulated Annealing, subrayando aquellas líneas que son las realmente importantes y que forman parte del esqueleto del mismo. Este código esta dentro de las clases C++ de la aplicación, porque necesitamos informar al usuario de cómo va la ejecución. Obsérvese que de no ser por esta necesidad, el código está programado con lenguaje C.

Código Fuente 4-3. Algoritmo Simulated Annealing

```
ESTADO CReferDoc::OptimizacionSimulatedAnnealing(
        TListaDouble *pListaVariables, TBDEntidades *pBD,
        TParametrosOptimizacion *pParametrosOpt,
        long *pNumeroIteracionesRealizadas,
        TListaDouble *pListaEvolucionCoste,
        TListaDouble *pListaTodasSoluciones,
        TListaDouble *pListaEvolucionZ,
        CDialogProgreso *pDialogoProgreso )
/* Aplica el Algoritmo de Optimizacion "Simulated Annealing". */
{
        ESTADO Estado;
        int i,
            iNumeroVariables,
            iContadorEscalones,
            iNumeroMaximoEscalones,
            iNumeroMaximoIteracionesEnEscalon,
            iCriterioEquilibrio,
            iSolucion,
         iContadorRechazos;
        long lTamanyoListaEvolucionCoste, lContadorIteracionesRealizadas;
        double dDato,
                dCriterioCongelacion,
                dCosteActual,
                dCosteAnterior,
                dMejorCoste,
                dIncrementoCoste,
                dTemperatura,
                dTemperaturaInicial,
                dIncrementoVariablesInicial;
        TListaDouble Solucion,
                NuevaSolucion,
                MejorSolucion;
        BOOLEAN bTerminarAlgoritmo;

        /* INICIALIZACION DE VARIABLES */

        iContadorEscalones = 0;         /* Contador de escalones */
        iNumeroMaximoEscalones = pParametrosOpt->SimulatedAnnealing.iNumeroMaximoEscalones;   /* Numero
maximo de escalones de temperatura */
        iNumeroVariables = pParametrosOpt->iNumeroVariables;
        iNumeroMaximoIteracionesEnEscalon = pParametrosOpt-
>SimulatedAnnealing.iNumeroMaximoIteracionesEnEscalon * iNumeroVariables;
        dCriterioCongelacion = pParametrosOpt->SimulatedAnnealing.dCriterioCongelacion;  /* Terminar si %
rechazos > dCriterioCongelacion */
```

```
        iCriterioEquilibrio = pParametrosOpt->SimulatedAnnealing.dCriterioEquilibrio *
                              (double)iNumeroMaximoIteracionesEnEscalon;    /* Saltar escalon si Nº
aciertos > iCriterioEquilibrio */

        if (pParametrosOpt->SimulatedAnnealing.iEstiloIncrementoVariables == INCREMENTO_VARIABLE_FIJO)
                dIncrementoVariablesInicial = pParametrosOpt->SimulatedAnnealing.dIncrementoVariables;
        else
                dIncrementoVariablesInicial =
                        EstimaIncrementoVariableSimulatedAnnealing( pBD, pParametrosOpt-
>SimulatedAnnealing.dCoefIncrementoVariables );

        pParametrosOpt->SimulatedAnnealing.dIncrementoVariablesActual = dIncrementoVariablesInicial;

        /* Reinicia las listas de soluciones y mejores soluciones */
        NuevaListaDouble( &Solucion );
        NuevaListaDouble( &NuevaSolucion );
        NuevaListaDouble( &MejorSolucion );

        CopiaListaDouble( pListaVariables, &Solucion );
        CopiaListaDouble( pListaVariables, &MejorSolucion );

        srand( (unsigned)time( NULL ) );  /* Reinicia los numeros aleatorios con el reloj del sistema */


        /* SE CALCULA LA TEMPERATURA INICIAL */
        /* METODO REFER */

        ActualizaCoeficientesRegularidades( pParametrosOpt->Regularidades.ListaCoeficientes, 0);

        dCosteActual = CalculaCoste( &Solucion, pBD, pParametrosOpt, pListaEvolucionCoste,
                                     pListaEvolucionZ, FALSE);

        /* Si la solucion ya es optima el coste vale 0, y entonces el algoritmo no empezara.
           Para obligarle a comenzar, calculare el coste inicial habiendo modificado alguna variable, y
           forzandole asi a que el coste no sea cero.                                                    */

        while (dCosteActual == 0) {
                GeneraNuevaSolucionSimulatedAnnealing( &Solucion, &NuevaSolucion, pParametrosOpt );
                DestruyeListaDouble( &Solucion );
                NuevaListaDouble( &Solucion );
                CopiaListaDouble( &NuevaSolucion, &Solucion );
                dCosteActual = CalculaCoste( &Solucion, pBD, pParametrosOpt, pListaEvolucionCoste,
                                                                        pListaEvolucionZ, FALSE);
        };

        dCosteAnterior = dMejorCoste = dCosteActual;
        dTemperaturaInicial = ((dCosteAnterior / 10.0) / 2.0);

        do {
                dTemperaturaInicial *= 2.0;   // Temperatura inicial demasiado baja; hay que aumentarla
                i = 0;
                for( iSolucion = 0 ; iSolucion < iNumeroMaximoIteracionesEnEscalon ; iSolucion++ ) {
                        GeneraNuevaSolucionSimulatedAnnealing( &Solucion, &NuevaSolucion,
pParametrosOpt );
                        dIncrementoCoste = CalculaCoste( &NuevaSolucion, pBD, pParametrosOpt,
                                        pListaEvolucionCoste, pListaEvolucionZ, FALSE ) - dCosteAnterior;
                        if ( exp( -dIncrementoCoste/dTemperaturaInicial ) >
                                                        ((double)rand() / (double)RAND_MAX_MAS_1))
                                i++;
                };
        } while ( ((double)i / (double)iNumeroMaximoIteracionesEnEscalon) <
                                pParametrosOpt->SimulatedAnnealing.dCoefAciertosTemperaturaInicial );

        /* FIN METODO REFER */


        dTemperatura = dTemperaturaInicial;
```

```
        lTamanyoListaEvolucionCoste = 0;
        lContadorIteracionesRealizadas = 0;
        bTerminarAlgoritmo = FALSE;

        /* HACER-MIENTRAS EL SISTEMA NO ESTE CONGELADO */
        do {
                iSolucion = 0;
                iContadorRechazos = 0;
                ActualizaCoeficientesRegularidades( pParametrosOpt->Regularidades.ListaCoeficientes,
                                                    iContadorEscalones);

                /* HACER-MIENTRAS NO SE ALCANCE LA CONDICION DE EQUILIBRIO */
                do {
                        lContadorIteracionesRealizadas++;

                        /* Guarda la evolucion de la Temperatura */
                        Estado = AnyadeListaDouble( pListaEvolucionCoste, dTemperatura );
                        ASSERT( Estado == OK );

                        /* Dada la solucion 'Solucion' obtiene la solucion 'Solucion+1' */
                        GeneraNuevaSolucionSimulatedAnnealing( &Solucion, &NuevaSolucion,
                                                               pParametrosOpt );
                        dCosteActual = CalculaCoste( &NuevaSolucion, pBD, pParametrosOpt,
                                                     pListaEvolucionCoste, pListaEvolucionZ, TRUE );
                        dIncrementoCoste = dCosteActual - dCosteAnterior;

                        /* Comprueba si el incremento de coste es negativo: ha disminuido el coste */
                        if ( dIncrementoCoste <= 0 ) {

                                /* Acepta nueva solucion:  pSolucion = pNuevaSolucion  */
                                dCosteAnterior = dCosteActual;
                                DestruyeListaDouble( &Solucion );
                                NuevaListaDouble( &Solucion );
                                CopiaListaDouble( &NuevaSolucion, &Solucion );

                                /* Comprueba si es la mejor solucion hasta el momento */
                                if ( dCosteActual < dMejorCoste ) {
                                        dMejorCoste = dCosteActual;
                                        DestruyeListaDouble( &MejorSolucion );
                                        NuevaListaDouble( &MejorSolucion );
                                        CopiaListaDouble( &Solucion, &MejorSolucion );
                                };
                        }
                        else
                                /* Realiza otro criterio de aceptacion : CRITERIO METROPOLIS */
                                if ( exp( -dIncrementoCoste/dTemperatura ) >
                                                        ( (double)rand() / (double)RAND_MAX_MAS_1) ) {

                                        /* Acepta nueva solucion:  pSolucion = pNuevaSolucion  */
                                        dCosteAnterior = dCosteActual;
                                        DestruyeListaDouble( &Solucion );
                                        NuevaListaDouble( &Solucion );
                                        CopiaListaDouble( &NuevaSolucion, &Solucion );

                                        /* Comprueba si es la mejor solucion hasta el momento */
                                        if ( dCosteActual < dMejorCoste ) {
                                                dMejorCoste = dCosteActual;
                                                DestruyeListaDouble( &MejorSolucion );
                                                NuevaListaDouble( &MejorSolucion );
                                                CopiaListaDouble( &Solucion, &MejorSolucion );
                                        };
                                }
                                else
                                        iContadorRechazos++;

                        iSolucion++;
```

```
                /* COMPRUEBA LA CONDICION DE EQUILIBRIO */
                } while ( (iSolucion < iNumeroMaximoIteracionesEnEscalon) &&
                          ((iSolucion - iContadorRechazos) < iCriterioEquilibrio) ) ;

                /* EVOLUCION DE LA TEMPERATURA */
                if (pParametrosOpt->SimulatedAnnealing.iEvolucionTemperatura ==
                                                                EVOLUCION_TEMPERATURA_CONSTANTE)
                        /* Evolucion de temperatura constante */
                        dTemperatura = (iNumeroMaximoEscalones-iContadorEscalones) *
                                                dTemperaturaInicial / iNumeroMaximoEscalones;
                else
                        /* Evolucion de temperatura segun una curva de tipo potencial */
                        dTemperatura *=
                                pParametrosOpt->SimulatedAnnealing.dDecrementoTemperaturaPotencial;

                /* El incremento de variable va evolucionando conforme a la temperatura, si procede. */
                if (pParametrosOpt->SimulatedAnnealing.bHayEvolucionIncrementoVariables == TRUE )
                        pParametrosOpt->SimulatedAnnealing.dIncrementoVariablesActual =
                                (iNumeroMaximoEscalones-iContadorEscalones) *
                                dIncrementoVariablesInicial / iNumeroMaximoEscalones;

                /* Guarda TODAS las MEJORES soluciones en CADA ESCALON de temperatura */
                for( i=0; i < iNumeroVariables; i++ ) {
                        ObtenListaDouble( &MejorSolucion, i, &dDato );
                        AnyadeListaDouble( pListaTodasSoluciones, dDato );
                };

/* ------------------------------------------------------------------------------ */
/* Con esta funcion se actualiza la barra de progreso, y se permite que las otras
   aplicaciones Windows funcionen en multitarea. (Ver 'ActualizaProgreso').
        Si 'Estado' no es OK, entonces hay que cancelar el algoritmo.                  */

                Estado = pDialogoProgreso->ActualizaProgreso( iContadorEscalones );

                // Dibuja el objeto segun la solucion de menor coste conseguida hasta este momento
                if (pParametrosOpt->bVerEvolucionPasoAPaso) {
                        ActualizaZconSolucionesBDEntidades( &MejorSolucion, pBD );
                        m_bCalcularVolumenOpenGL = true;
                        RefrescaVentanaReconstruccion3D();
                };
/* ------------------------------------------------------------------------------ */

                iContadorEscalones++;

        /* COMPRUEBA EL CRITERIO DE CONGELACION (TEMPERATURA FINAL) */
                if ((double)iContadorRechazos/(double)iSolucion) > dCriterioCongelacion)
                        bTerminarAlgoritmo = TRUE;

        } while ((iContadorEscalones < iNumeroMaximoEscalones) &&
                 (bTerminarAlgoritmo == FALSE) &&
                 (Estado == OK));


        if (Estado != CANCELADO_POR_USUARIO) {        /* Se devuelve el resultado en 'pListaVariables' */
                DestruyeListaDouble( pListaVariables );
                NuevaListaDouble( pListaVariables );
                CopiaListaDouble( &MejorSolucion, pListaVariables );
        };

        DestruyeListaDouble(&Solucion);
        DestruyeListaDouble(&NuevaSolucion);
        DestruyeListaDouble(&MejorSolucion);

        *pNumeroIteracionesRealizadas = lContadorIteracionesRealizadas;
        return Estado;
};
```

El algoritmo principal utiliza dos importantes funciones **GeneraNuevaSolucionSimulatedAnnealing** y **CalculaCoste**. La primera paso a describirla ahora, y la segunda se puede ver en el *Capitulo5.6. Implementación de la función objetivo.*

Código Fuente 4-4. Función GeneraNuevaSolucionSimulatedAnnealing

```
void GeneraNuevaSolucionSimulatedAnnealing(
        TListaDouble *pSolucion,
        TListaDouble *pNuevaSolucion,
        TParametrosOptimizacion *pParametros)
/* Dada la solucion 'pSolucion', calcula la solucion 'pNuevaSolucion' que corresponde a 'i'. */
{
int iVariableModificada;
double dDato;

/* Primero prepara la nueva solucion, a partir de los valores de la solucion anterior. */
DestruyeListaDouble( pNuevaSolucion );
NuevaListaDouble( pNuevaSolucion );
CopiaListaDouble( pSolucion, pNuevaSolucion );

/* Selecciona aleatoriamente la variable que sera modificada */
iVariableModificada = ((double)rand() / (double)RAND_MAX_MAS_1) * (double)pParametros->iNumeroVariables;
ObtenListaDouble( pNuevaSolucion, iVariableModificada, &dDato );

/* Selecciona aleatoriamente como se modifica la variable: sumando o restando */
if ( pParametros->iAlgoritmoOptimizacion == ALGORITMO_SIMULATED_ANNEALING )
        if ( ((double)rand() / (double)RAND_MAX_MAS_1) < 0.5 )
                dDato += pParametros->SimulatedAnnealing.dIncrementoVariablesActual;
        else
                dDato -= pParametros->SimulatedAnnealing.dIncrementoVariablesActual;
else
        if ( ((double)rand() / (double)RAND_MAX_MAS_1) < 0.5 )
                dDato += pParametros->SimulatedAnnealingMulti.dIncrementoVariablesActual;
        else
                dDato -= pParametros->SimulatedAnnealingMulti.dIncrementoVariablesActual;

SustituyeListaDouble( pNuevaSolucion, iVariableModificada, dDato );
};
```

4.3 Optimización Simulated-Annealing Multicriterio

En la implementación del Simulated Annealing tenemos un aspecto que no es fácil de resolver, y es la correcta definición de la función objetivo, en concreto la correcta ponderación de las regularidades. En una misma función estamos operando con conceptos distintos, distintas magnitudes e incluso distintas unidades de medida. Utilizamos un factor de ponderación para cada regularidad de modo que más o menos cada una tenga el peso que le corresponde.

Nos planteamos si sería más útil utilizar un algoritmo Simulated Annealing Multicriterio, que básicamente consiste en una modificación de un clásico Simulated Annealing en el que el criterio Metrópolis se calcula para cada regularidad, y se acepta una solución cuando todas las regularidad cumplen el criterio Metropolis. Hay por tanto una temperatura por cada regularidad.

Aquí el concepto de función objetivo como una suma de regularidades ponderadas no existe. Hay que ver la función objetivo como un conjunto de regularidades que tienen que mejorar todas a la vez. El problema es que así formulado este criterio es demasiado restrictivo. Hay regularidades que entran en conflicto con otras, en el sentido que no permiten el avance de éstas porque tienen objetivos distintos.

Probamos a implementar este algoritmo, pero los resultados no mejoraron con respecto al Simulated Annealing clásico, así que para este PFC nos centramos en este último, con el cual estábamos teniendo buenos resultados.

A continuación mostraremos el esquema del algoritmo en pseudo-código.

Figura 4-24. Pseudo-código del algoritmo Simulated Annealing Multicriterio

- INICIALIZACION DE VARIABLES
- SE CALCULA LA TEMPERATURA INICIAL. METODO Refer Multicriterio
  - Hay que determinar la temperatura inicial de cada regularidad
  - Para cada regularidad, si su coste vale 0 su solucion particular ya es optima, y entonces el algoritmo no empieza.
    - Para obligarle a comenzar, calculare el coste inicial habiendo modificado algunas variables hasta que ninguna regularidad tenga coste 0
  - Ahora ya tienen todas las regularidades un coste positivo. En funcion de este coste, podemos estimar el orden de magnitud de su temperatura asociada
  - Hay que ajustar la temperatura de cada regularidad para que efectivamente cumpla las condiciones iniciales del criterio metrópolis
    - Temperatura inicial demasiado baja; hay que aumentarla, hasta que sea adecuada.
- FIN METODO Refer Multicriterio
- HACER-MIENTRAS EL SISTEMA NO ESTE CONGELADO
  - HACER-MIENTRAS NO SE ALCANCE LA CONDICION DE EQUILIBRIO
    - Dada la solucion 'Solucion' obtiene la solucion 'Solucion+1'

- Segun el algorimo Simulated Annealing Multicriterio se aceptara una nueva solucion cuando TODAS las regularidades cumplan el criterio Metropolis. Comprobamos cuantas regularidades cumplen el criterio Metropolis.
    - Si todas regularidades cumplen criterio Metropolis entonces Acepta nueva solucion y Comprueba si es la mejor solucion hasta el momento

- COMPRUEBA LA CONDICION DE EQUILIBRIO
- EVOLUCION DE LA TEMPERATURA (para cada temperatura de cada Reguladidad)
- Guarda la ULTIMA solucion de CADA ESCALON de temperatura

- COMPRUEBA EL CRITERIO DE CONGELACION (TEMPERATURA FINAL)
- FIN DEL ALGORITMO

Sería un buen tema de investigación futura el analizar con más detenimiento este algoritmo. Hemos visto que utilizamos un criterio muy restrictivo. Quizá se podría probar suavizándolo un poco del modo siguiente: exigir que al menos un porcentaje de las regularidades mejoren su coste, no todas a la vez. De este modo se permitiría que algunas regularidades mejorasen a costa de las otras.

5. REGULARIDADES

5.1 Introducción

Las regularidades son el camino para ser explícito con las percepciones humanas. Para el fundamento de este concepto podemos referenciar el libro de la psicología clásica de Koffka [18].

Sin embargo, la noción de **regularidad** requiere una explicación más detallada. De hecho las primeras apariciones de la palabra *regularidades* asociada con representaciones gráficas se deben a Gestalt psicólogos que llamaban regularidades a aquellas relaciones que no pueden ser un accidente.

Es decir, se supone que inspeccionando la imagen, es posible deducir el número necesario de propiedades que describen un modelo, y formular en lenguaje matemático el conjunto de condiciones suficientes para adoptar como "buena" una solución en la reconstrucción geométrica.

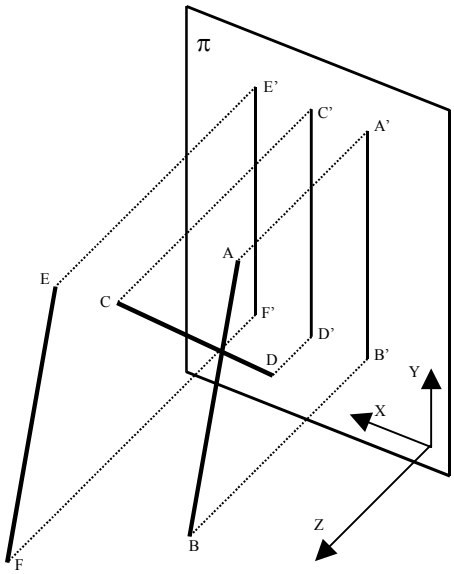


Figura 5-1. Regla heurística de la regularidad de paralelismo

Las regularidades pueden ser definidas como relaciones geométricas entre entidades o grupos de entidades. Sin embargo, la formulación de regularidades está sujetas a reglas heurísticas y por tanto basadas en probabilidades. Un ejemplo típico de regularidad es el *paralelismo*. La regla heurística que rige dicha regularidad nos dice que si dos líneas son paralelas en el plano, probablemente dichas líneas representen dos aristas del modelo tridimensional, pero esto no tiene por que ser cierto con probabilidad uno. De hecho, si nos fijamos en la representación dada en la Figura 5-1 las rectas A'B', C'D' y E'F' dadas en la imagen proyección de tres aristas tridimensionales resultan ser paralelas en sí y de acuerdo con la regla heurística anteriormente citada debería corresponder con tres aristas tridimensionales entre sí. Sin embargo como puede verse si

bien dichas regla se verifica para las aristas AB y CD no se cumplen con la arista CD. Por consiguiente dichas reglas están restringidas a los puntos de vista de la imagen de partida.

5.2 Interpretación de las imágenes.

Además de los problemas anteriormente derivados de las reglas heurísticas que rigen las regularidades, es preciso añadir el problema de la rigurosidad que se presenta al tratar las regularidades de forma matemática. En otras palabras y siguiendo con el ejemplo de la regularidad de paralelismo anteriormente expuesto, cuando dos aristas no son exactamente paralelas, pueden ser interpretadas por el observador humano como si lo fuesen, mientras que matemáticamente, dichas aristas serían consideradas como concurrentes y por consiguiente no estarían afectadas de la regularidad de paralelismo.

Para paliar dichas diferencias, la mayoría de los sistemas CAD aceptan un valor de 7º como diferencia angular permisible para determinar el paralelismo. Como consecuencia de ello se establece un factor de confianza μ(a) donde "a" es la diferencia angular existente entre las dos líneas en cuestión.

Dicho factor de confianza μ está definido de manera que alcance su máximo valor 1.0 para una condición de paralelismo exacta (a = 0) y decreciente hasta cero de acuerdo a una curva de distribución normal con σ = 7 cuando "a" se aproxima a 90º (ver *Figura 5-2*)

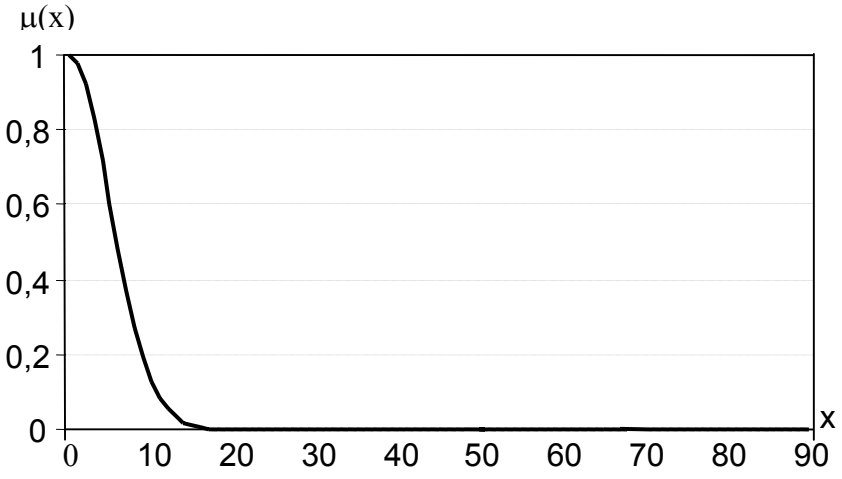


Figura 5-2. Representación gráfica de la función de confianza

La formulación general de la función de confianza viene dada por la expresión:

$$\mu_{a,b}(x) = e^{-((x-a)/b)^2}$$

Expresión 5-1

donde:

- "x" representa el valor obtenido de la imagen.
- "a" es un valor nominal de referencia (por ejemplo 0º para el paralelismo).
- "b" es la desviación límite permisible.

Con propósitos prácticos, la Expresión 5-1 puede ser modificada para eliminar los valores muy próximos a cero de acuerdo quedando así:

$$\mu_{a,b}(x) = \max [0, 1.1\, e^{-((x-a)/b)^2} - 0.1]$$

Expresión 5-2

5.3 Formulación de las regularidades.

En este PFC sólo vamos a implementar dos regularidades:

- Mínima desviación estándar de los ángulos
- Paralelismo de lineas

sin embargo hay que tener en cuenta que los objetivos de este PFC se engloban dentro de los objetivos del Grupo REGEO (ver Capítulo 2.2, pag. 21). Por tanto hay que prever la implementación del resto de regularidades, preparando los menús de diálogo de Refer y dejando el código fuente dispuesto para la programación de nuevas regularidades sin que sea una tarea traumática. A título orientativo se verán todas las regularidades tal y como las formulan Lipson y Shpitalni [10].

Las regularidades pueden ser agrupadas en tres grandes grupos en función del número y tipo de entidades que relacionan [10]:

1. Regularidades que reflejan relaciones espaciales entre entidades individuales, como la regularidad de paralelismo ya referida anteriormente.

2. Regularidades que reflejan relaciones espaciales entre grupos de entidades, como pueden ser las entidades de forman el contorno de una imagen o una cadena de entidades, este es el caso de la regularidad de posible simetría existente en una determinada cara.

3. Regularidades que reflejan relaciones espaciales que afectan a todas las entidades del dibujo como la regularidad de isometría.

Las siguientes notaciones van a ser usadas para la formulación de las regularidades:

5.3.1 Regularidad paralelismo de líneas.

Esta regularidad esta basada en la hipótesis de que todos los pares de aristas paralelas en la imagen deben corresponder a proyecciones de pares de aristas paralelas en el espacio. La expresión analítica para formular esta regularidad es evaluada de la siguiente forma:

$$R_{paralelismo} = \omega\,[\cos^{-1}(u_1, u_2)]^2 \quad \text{siendo} \quad \omega = \mu_{0^o,7^o}(\cos^{-1}(u_1', u_2'))$$

Expresión 5-3

donde:

- u_1 y u_2 representan los vectores unitarios tridimensionales según aristas 1 y 2.
- u_1' y u_2' representan los vectores unitarios de la imagen según aristas 1 y 2.
- por último ω un factor de peso dado por la función de confianza $\mu(x)$.

Los problemas derivados de esta regla heurística han sido expuestos ya con antelación en la Expresión 5-1.

5.3.2 Regularidad verticalidad de aristas.

Se establece como hipótesis que toda línea vertical de la imagen (paralela al eje Y) corresponde con una arista vertical en el espacio (sus dos extremos finales deben tener igual coordenada Z).

La expresión que evalúa la verticalidad de las aristas es de la forma:

$$R_{verticalidad} = \omega\,[\cos^{-1}(u_Y)]^2$$

Expresión 5-4

siendo $\omega = \mu_{0^o,7^o}(\cos^{-1}(u_Y'))$

donde:

- u_Y es la componente "y" del vector unitario en el espacio.
- u_Y' es la componente "y" del vector unitario de la imagen en la dirección de la arista.

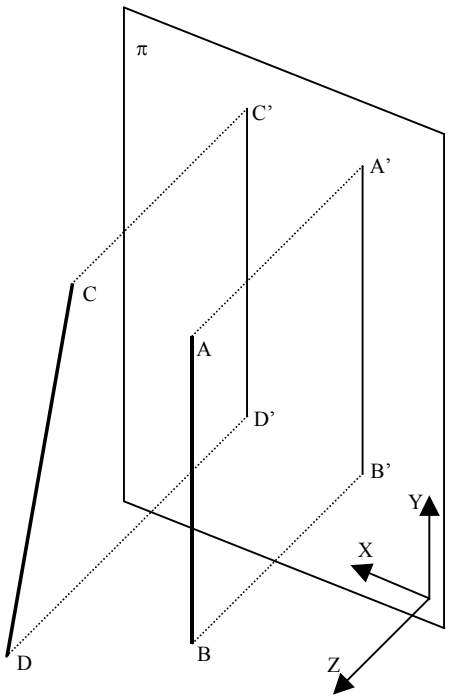


Figura 5-3. Regla heurística de la regularidad de verticalidad

Nuevamente la regla heurística esta sometida a inconsistencias como se demuestra en la Figura 5-3, donde si bien la línea proyectada A'B' paralela al eje "Y" si se corresponde con una arista vertical del modelo AB (los puntos A y B tienen igual coordenada Z), la línea C'D' que también resulta ser paralela al eje "Y" no corresponde a ninguna vertical del modelo, dado que las coordenadas C_Z y D_Z difieren entre sí.

5.3.3 Regularidad mínima desviación estándar de ángulos

La regularidad de mínima desviación estándar de ángulos a la que a partir de ahora nos referiremos como MSDA, se trata en realidad de una falsa regularidad, en tanto que no supone una hipótesis de cumplimiento de una forma generalizada para cualquier tipo de modelo, si no mas bien, una idea basada en las propiedades que verifican los poliedros regulares.

Dicha regularidad propone que los ángulos formados entre todos los pares de aristas que convergen en un mismo vértice deben ser similares. Esta hipótesis es formulada en términos matemáticos de acuerdo con las siguientes expresiones:

$$R_{MSDA} = n\,\sigma^2\,(\cos^{-1}(u_1, u_2))^2$$

Expresión 5-5

donde:

- u_1 y u_2 representan los vectores unitarios tridimensionales según aristas de todos los posibles pares de lineas que concurren en un mismo vértice del modelo.

5.3.4 Regularidad ortogonalidad de líneas.

Esta regularidad simplificada como MSDP, al igual que la anteriormente comentada, resulta una falsa regularidad en tanto que nuevamente la hipótesis de la que parte no se justifica en la gran mayoría de los modelos, si bien como se analizará al final de este capítulo, su formulación tiene propósitos relevantes.

De acuerdo con esta regla heurística, todos los pares de líneas de la imagen que concurren en un mismo vértice deben corresponderse con aristas perpendiculares en el modelo tridimensional, salvo que dichas aristas resulten colineales. Los términos matemáticos usados para su evaluación son expresados en la forma:

$$R_{MSDP} = \Sigma\,\omega\,[\sin^{-1}(u_1, u_2)]^2 \quad \text{siendo} \quad \omega = \mu_{0^o,7^o}(\cos^{-1}(u_1', u_2'))$$

Expresión 5-6

donde:

- n es el número de pares de líneas no colineales que convergen en un mismo vértice.
- u_1 y u_2 representan los vectores unitarios tridimensionales según aristas 1 y 2
- u_1' y u_2' representan los vectores unitarios de la imagen según aristas 1 y 2
- un factor de peso dado por la función de confianza $\mu(x)$.

5.3.5 Regularidad colinealidad de aristas.

Las aristas colineales en el plano del boceto son colineales en el espacio. El término usado para plasmar esta heurística es

$$R_{colinealidad} = \sum_{i=1}^{n} w_i \cdot \max_{j=1,\ldots,4} \times \left[\frac{\det\left|\bar{v}_j, \bar{v}_{j+1}, \bar{v}_{j+2}\right|}{\max\left(\left\|\bar{v}_j - \bar{v}_{j+1}\right\|, \left\|\bar{v}_{j+1} - \bar{v}_{j+2}\right\|, \left\|\bar{v}_{j+2} - \bar{v}_j\right\|\right)} \right]^2$$

Expresión 5-7

$$w_i = 1 = \mu_{0^o,7^o} \cdot \left(\cos^{-1}\left(\hat{l}_1' \cdot \hat{l}_2'\right)\right)$$

donde:

- n es el número de los pares colineales
- $v_{j=1,\ldots,4}$ son los cuatro vértices finales de las dos aristas.

5.3.6 Regularidad isometría.

Las longitudes de las entidades en el modelo 3D son uniformemente proporcionales a sus longitudes en el plano del boceto. El término a considerar para la no uniformidad corresponde a la desviación típica de las escalas.

$$R_{isometria} = n \cdot \sigma^2\left(r_{i=1,\ldots\ldots,N_e}\right)$$

Expresión 5-8

$$r_i = \frac{longitud\left(entidad_i\right)}{longitud'\left(entidad_i\right)}$$

donde:

- n es el número de entidades
- r_i es el ratio entre la actual longitud de la entidad i y su longitud en el plano del boceto
- s es la desviación típica de las series de r_i .

5.3.7 Regularidad esquinas ortogonales.

Una unión de tres líneas que matemáticamente sea una proyección de una esquina ortogonal 3D es ortogonal en el espacio. Para determinar si una unión de tres líneas de un plano puede ser una proyección de una esquina ortogonal, se aplica la siguiente prueba, basada en el hecho de que la proyección de una esquina ortogonal tiene un arco mínimo de 90º.

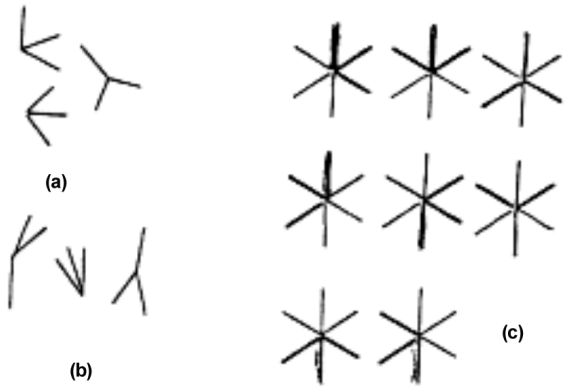


Figura 5-4. Unión de tres lineas

(a) Algunas uniones de tres líneas pueden aparecer formando esquinas ortogonales. (b) algunas no. (c) una unión de tres líneas tiene ocho variantes

Una unión de tres líneas tiene ocho variantes, creadas cambiando la dirección de cada línea y considerando las 8 permutaciones resultantes (ver Figura 5-4). Se prueban las ocho variantes de la unión en el *plano del boceto*; para cada variante, existen tres líneas $l'_{l=1,\ldots,3}$, formando tres pares entre ellas mismas, $l'_{l=1,2}$, $l'_{l=2,3}$, $l'_{l=3,1}$. Cada línea se describe por medio de un vector de dirección en 2D, $\hat{l}'_i$ en el plano del boceto, apuntando desde la unión hacia fuera. Si una variante de la unión forma un arco de menos de 90º (p. ej. no es una proyección de una esquina ortogonal), todos los tres puntos de sus pares de vectores directores serán positivos. Si una unión de tres líneas es una proyección de una esquina ortogonal, todas sus ocho variantes deben tener un arco de al menos 90º.En consecuencia, si alguno de las ocho variantes parece formar un arco menor de 90º (aparece como una condición todo-positiva), la unión analizada es improbable que represente una esquina ortogonal.

Consecuentemente, el término usado para evaluar la condición de ortogonalidad de la esquina es:

$$R_{esquina} = w_{esquina} \sum_{par=1}^{3} \left[sen^{-1} \left(\hat{l}_1 \cdot \hat{l}_2 \right) \right]^2$$

Expresión 5-9

$$w_{esquina} = \begin{cases} 1 & \text{si } \beta \leq 0 \\ \mu_{0,\,0.1} & \text{si } \beta > 0 \end{cases}$$

$$\beta = \max_{8 \text{ var} iantes} \left[\min_{3\ pares} \left(\hat{l}'_a \cdot \hat{l}'_b \right) \right]$$

5.3.8 Regularidad planicidad de caras.

Un contorno de cara consistente enteramente en líneas rectas es el reflejo de una cara plana en 3D. La evaluación de esta relación se desarrolla en dos etapas: la primera es encontrar la superficie que mejor se adapte a los vértices del contorno; la segunda consiste en calcular, elevar al cuadrado y sumar las desviaciones de cada vértice de esa superficie. El plano que mejor se adapte vendrá dado por:

$ax + by + cz + d = 0$

Los coeficientes del plano *a, b y c* son calculados resolviendo el sistema lineal mediante la lista de puntos (x_i ; y_i, ; z_i *i = 1, ... , n)* que están en el plano, y asumiendo d = 1.

$$\sum \begin{bmatrix} x_i^2 & x_i y_i & x_i z_i \\ x_i y_i & y_i^2 & y_i z_i \\ x_i z_i & y_i z_i & z_i^2 \end{bmatrix} \cdot \begin{bmatrix} a \\ b \\ c \end{bmatrix} = \sum \begin{bmatrix} x_i \\ y_i \\ z_i \end{bmatrix}$$

Los coeficientes son entonces normalizados haciendo $\sqrt{a^2+b^2+c^2}=1$ con d escalado apropiadamente y la distancia de un punto al plano calculada como el valor absoluto $| a\,x_i + b\,x_i + c\,x_i\ |$.

Cabe resaltar que para usar esta regularidad, se necesita una etapa de preproceso en la cual los circuitos de vértices correspondientes a las caras del objeto 3D representado, son identificadas en un grafo 2D.

Hay otros métodos para evaluar la planicidad de un conjunto de vértices en el espacio. Sin embargo, los métodos que se basan en examinar la planicidad de las secuencias de cuatro puntos que estén alrededor del contorno (ej. Leclerc[10]) no son suficientes porque la planicidad local de los puntos de contorno no asegura su planicidad global. Esta situación puede ocurrir siempre que hay una secuencia de tres puntos colineales.

5.3.9 Regularidad ortogonalidad de caras.

Un contorno de una cara que muestra ortogonalidad oblicua es probable que sea ortogonal en el espacio. Si las entidades del contorno de una cara plana se unen solo en los ángulos rectos, entonces el contorno puede decirse que es ortogonal. Si el contorno se ve desde un punto de vista arbitrario, mostrará ortogonalidad oblicua, como se muestra en la Figura 5-5a. Las caras o cadena de entidades que muestren ortogonalidad oblicua se detectan fácilmente alternando sus líneas del perímetro entre dos direcciones principales que correspondan a las direcciones de los ejes principales del boceto original.

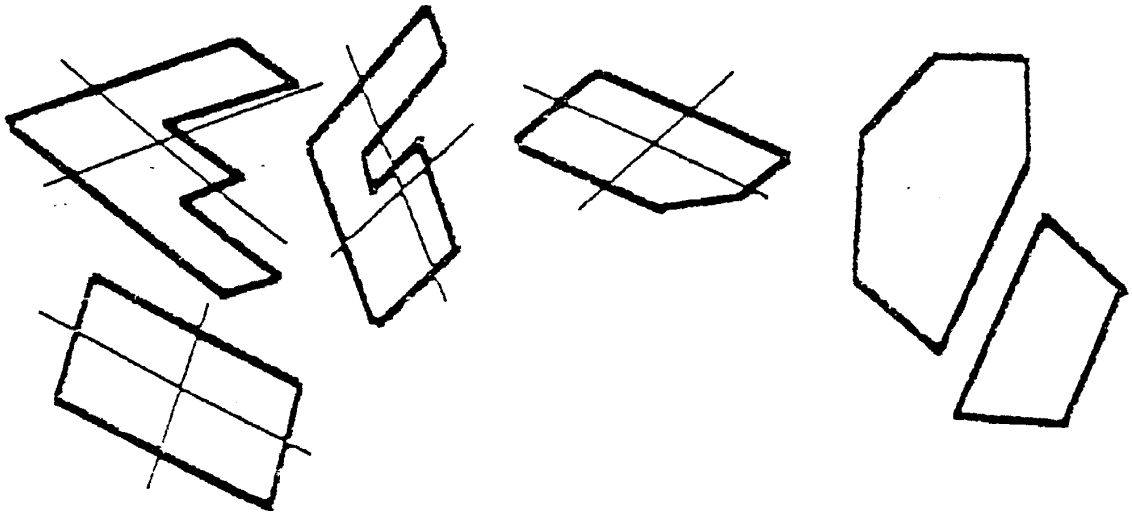


Figura 5-5. Caras ortogonales
a) Caras mostrando ortogonalidad oblicua con sus ejes principales oblicuos b) Ortogonalidad parcial y c) Sin ortogonalidad

Se usa para la detección, el comportamiento estadístico de los valores alternantes producidos por la multiplicación del producto escalar y el producto vectorial de líneas adyacentes en 2D en el plano del boceto. El comportamiento consistente es probable que represente ortogonalidad oblicua. La cantidad por la cual la cara se considera que tiene ortogonalidad oblicua se representa por el valor de el coeficiente de peso $w_{\text{ortogonalidad oblicua}}$. Los términos para evaluar lo anterior para una cara son

$$R_{\substack{ortogonalidad \\ oblicua}} = w_{\substack{ortogonalidad \\ oblicua}} \cdot \sum_{i=1}^{n} \left[sen^{-1}\left(\hat{l}_i \cdot \hat{l}_{i+1} \right) \right]^2$$

Expresión 5-10

$$w_{\substack{ortogonalidad \\ oblicua}} = \mu_{0,\,0.2}\left(\sigma\left(\beta_{i=1,\ldots,n} \right) \right)$$

$$\beta_i = (-1)^i \cdot \left[\hat{l}_i' \cdot \hat{l}_{i+1}' \right] \cdot \left[\hat{l}_i' \times \hat{l}_{i+1}' \right]$$

donde:

- *n* representa el número de líneas a lo largo del contorno de la cara

Las caras en las cuales solo parte de su contorno se muestra ortogonalidad oblicua, como en la Figura 5-5b, pueden ser también aceptadas, con la lista de líneas $l'_{i=1,\ldots,n}$ reducida a un subconjunto a lo largo del perímetro. Esto requiere que se pueda detectar subconjuntos con b_i consistente (baja σ).

5.3.10 Regularidad perpendicularidad de caras.

La hipótesis aquí establecida, y que nuevamente constituye una falsa regularidad, establece que todas las caras adyacentes en la imagen (caras que compartan alguna arista en común) deben ser perpendiculares entre sí.

El término utilizado para la evaluación es:

$$R_{PerpendCaras} = (sen^{-1}(n_1, n_2))^2$$

Expresión 5-11

donde:

- n_1 y n_2 representan los vectores unitarios tridimensionales normales a todas las caras adyacentes del modelo.

5.3.11 Regularidad simetría de caras.

Una cara que muestre simetría oblicua en 2D denota una verdadera cara simétrica en 3D (véase Figura 5-6). Los algoritmos para la detección de simetría oblicua han sido el tema de muchas investigaciones extensas.

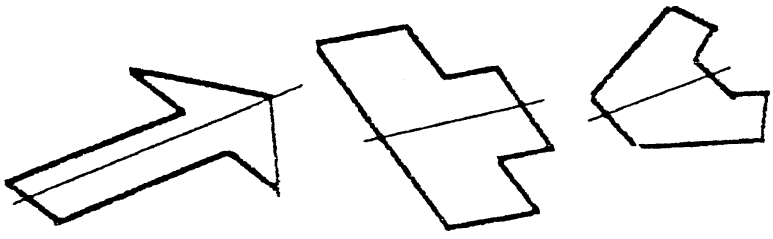


Figura 5-6. Caras mostrando simetría oblicua

Una simplificación usada aquí, es que si existe simetría oblicua en una figura poligonal, su eje intersecta el contorno en dos puntos, "cada vértice o punto medio de una entidad". Asumiendo también que el número de entidades en ambos lados del eje de

simetría en una figura totalmente simétrica es igual, el número de posibles candidatos al eje de simetría se reduce significativamente a *n*, donde *n* es el número de vértices de la figura. Cada posible candidato a eje de simetría pasa por los vértices v_k y $v_{k+n/2}$, donde $k=1/2, 2/2, 3/2, ..., n/2$ y, por ejemplo, $v_{2.5} = (v_3 + v_2) / 2$.

La relación entre los vértices de una figura y el candidato a eje de simetría determina si el eje puede servir como eje de simetría oblicua. Esta relación se representa, mediante el eje *k*, a través del coeficiente de peso w_k. El máximo w_k determina el eje de simetría elegido si la cara posee la característica de simetría oblicua por completo.

$$w_k = \mu_{0,\,0.2}\left(\sigma_{i=1/2,2/2,3/2,\ldots n/2}\left(\text{oblicua}_i\right)+\sigma_{i=1/2,2/2,3/2,\ldots n/2}\left(\text{sim}_i\right)\right)$$

$$w_{\substack{simetria\\oblicua}} = \max_{k=1/2,2/2,3/2,\ldots,n/2}\left[w_k\right]$$

$$oblicua_i = \left[\frac{v_k' - v_{k+n/2}'}{\left\|v_k' - v_{k+n/2}'\right\|}\cdot\frac{v_{k+1}' - v_{k-1}'}{\left\|v_{k+1}' - v_{k-1}'\right\|}\right]\times\left[\frac{v_k' - v_{k+n/2}'}{\left\|v_k' - v_{k+n/2}'\right\|}\times\frac{v_{k+1}' - v_{k-1}'}{\left\|v_{k+1}' - v_{k-1}'\right\|}\right]$$

$$sim_i = \frac{distancia\ \text{de}\ v_{k+i}\ \text{al eje}}{\text{distancia de}\ v_{k-i}\ \text{al eje}} - 1$$

Cabe decir que los vértices v_k' están en el plano 2D del boceto. Dos condiciones se necesitan para que ocurra simetría oblicua. La primera, los puntos correspondientes deben estar equidistantes del eje de simetría, una condición que hemos llamado *sim* en las fórmulas anteriores; segunda, las líneas que se extienden entre los correspondientes puntos deben formar un ángulo consistente con el eje de simetría, una condición que hemos llamado *oblicua* en las fórmulas anteriores. Si se ha detectado simetría oblicua, el término de optimización será:

$$R_{\substack{simetria\\oblicua}} = w_{\substack{simetria\\oblicua}}\sum_{i=1}^{n/2}\left[sen^{-1}\left(\left[\frac{\left(v_{k+1} - v_{k-1}\right)}{\left\|v_{k+1} - v_{k-1}\right\|}\right]\times\left[\frac{\left(v_k - v_{k+n/2}\right)}{\left\|v_k - v_{k+n/2}\right\|}\right]\right)\right]^2$$

donde:

- *k* denota el eje que ha sido seleccionado.

5.4 Implementación de las regularidades

5.4.1 Mínima Desviación Estándar de los Ángulos

A continuación presentaremos el código de la función **CalculaCosteDesviacionEstandarAngulos,** que dada una solución, nos devuelve su coste en función de la *Regularidad Mínima Desviación Estándar de los Ángulos.* Se subrayan aquellas líneas que son las realmente importantes y que forman parte del esqueleto de la función.

Código Fuente 5-1. Regularidad: Mínima Desviación Estándar de los Angulos

```
double CalculaCosteDesviacionEstandarAngulos( TListaDouble *pSolucion, TBDEntidades *pBD )
/* Devuelve el coste de la 'pSolucion' teniendo en cuenta la Base de Datos 'pBD' y la
        siguiente regularidad:                                                    */
/*      MINIMA DESVIACION ESTANDAR DE LOS ANGULOS DE LAS ARISTAS.                 */
{
long i, j, k, n,
     lIndiceArista1, lIndiceArista2;
TVertice Vertice, VerticeCabeza, VerticeCola, VectorUnitario1, VectorUnitario2;
TArista Arista1, Arista2;
double dDato, dProductoEscalar,
       dAnguloRadianes, dAnguloGrados,
       dSumatorioAngulos, dSumatorioAngulosAlCuadrado;

dSumatorioAngulos = dSumatorioAngulosAlCuadrado = 0.0;
n = 0;

/* Para cada vertice, analiza el angulo que forman sus aristas incidentes */
for( i=0; i < TamanyoListaVertices( &pBD->ListaVertices ) ; i++ ) {
         NuevoVertice( &Vertice );
         ObtenListaVertices( &pBD->ListaVertices, i, &Vertice );

         for( j=0 ; j < (TamanyoListaLong( &Vertice.ListaAristas )-1) ; j++ ) {
                  ObtenListaLong( &Vertice.ListaAristas, j, &lIndiceArista1 );

                  /* Dada una arista del vertice, analiza el angulo que forma con las aristas que le
                     siguen */
                  for( k=j+1 ; k < TamanyoListaLong( &Vertice.ListaAristas ) ; k++ ) {
                           ObtenListaLong( &Vertice.ListaAristas, k, &lIndiceArista2 );

                           /* Voy a calcular el angulo formado entre Arista1 y Arista2. */
                           ObtenListaAristas( &pBD->ListaAristas, lIndiceArista1, &Arista1 );
                           ObtenListaAristas( &pBD->ListaAristas, lIndiceArista2, &Arista2 );

                           /* Calculo el Vector Unitario de la Arista 1 */
                           NuevoVertice( &VerticeCabeza );
                           ObtenListaVertices( &pBD->ListaVertices, Arista1.lCabeza, &VerticeCabeza );
                           ObtenListaDouble( pSolucion, Arista1.lCabeza, &dDato );
                           VerticeCabeza.z = dDato;
```

```
                        NuevoVertice( &VerticeCola );
                        ObtenListaVertices( &pBD->ListaVertices, Arista1.lCola, &VerticeCola );
                        ObtenListaDouble( pSolucion, Arista1.lCola, &dDato );
                        VerticeCola.z = dDato;

                        VectorUnitario1 = CalculaVectorUnitario3D( &VerticeCabeza, &VerticeCola );

                        DestruyeVertice( &VerticeCabeza );
                        DestruyeVertice( &VerticeCola );

                        /* Calculo el Vector Unitario de la Arista 2 */
                        NuevoVertice( &VerticeCabeza );
                        ObtenListaVertices( &pBD->ListaVertices, Arista2.lCabeza, &VerticeCabeza );
                        ObtenListaDouble( pSolucion, Arista2.lCabeza, &dDato );
                        VerticeCabeza.z = dDato;

                        NuevoVertice( &VerticeCola );
                        ObtenListaVertices( &pBD->ListaVertices, Arista2.lCola, &VerticeCola );
                        ObtenListaDouble( pSolucion, Arista2.lCola, &dDato );
                        VerticeCola.z = dDato;

                        VectorUnitario2 = CalculaVectorUnitario3D( &VerticeCabeza, &VerticeCola );

                        DestruyeVertice( &VerticeCabeza );
                        DestruyeVertice( &VerticeCola );

                        /* Calculo el Producto Escalar del Vector Unitario 1 y el Vector Unitario 2 */
                        dProductoEscalar = PRODUCTO_ESCALAR( VectorUnitario1, VectorUnitario2 );
                        assert( !_isnan(dProductoEscalar) );

                        if (dProductoEscalar > 1) dProductoEscalar=1;
                        if (dProductoEscalar < -1) dProductoEscalar=-1;

                        /* Calculo el Angulo, en grados, que forman el Vector Unitario 1 y
                           el Vector Unitario 2 */
                        dAnguloRadianes = acos( dProductoEscalar );
                        dAnguloGrados = RADIANES_A_GRADOS( dAnguloRadianes );

                        dSumatorioAngulos += dAnguloGrados;
                        dSumatorioAngulosAlCuadrado += (dAnguloGrados*dAnguloGrados);
                        assert( !_isnan(dSumatorioAngulosAlCuadrado) );
                        n++;
                    };
                };
                DestruyeVertice( &Vertice );
            };
            /* Devuelve la Desviacion Estandar de los Angulos de cada arista que confluye en un vertice.*/
            return sqrt( ( dSumatorioAngulosAlCuadrado - (dSumatorioAngulos*dSumatorioAngulos)/n ) / (n-0) );
        };
```

5.4.2 Paralelismo de Líneas.

A continuación presentaremos el código de la función **CalculaCosteParalelismoLineas,** que dada una solución, nos devuelve su coste en función de la *Regularidad Paralelismo de Líneas.* Se subrayan aquellas líneas que son las realmente importantes y que forman parte del esqueleto de la función.

```
double CalculaCosteParalelismoLineas( TListaDouble *pSolucion, TBDEntidades *pBD )
/* Devuelve el coste de la 'pSolucion' teniendo en cuenta la Base de Datos 'pBD' y la
        siguiente regularidad:                                                    */
/*      PARALELISMO DE LINEAS.                                                    */
{
long i, j;
TVertice VerticeCabeza1, VerticeCola1,
         VerticeCabeza2, VerticeCola2,
         VectorUnitario2D1, VectorUnitario2D2,
         VectorUnitario3D1, VectorUnitario3D2;
TArista Arista1, Arista2;
double dDato, dPesoW12,
       dAnguloRadianes, dAnguloGrados,
       dSumatorioParalelismo, dProductoEscalar;

dSumatorioParalelismo = 0;

/* Comprueba el paralelismo de cada arista con todas las demas. */
for( i=0; i<TamanyoListaAristas( &pBD->ListaAristas ); i++ ) {
         ObtenListaAristas( &pBD->ListaAristas, i, &Arista1 );

         NuevoVertice( &VerticeCabeza1 );
         ObtenListaVertices( &pBD->ListaVertices, Arista1.lCabeza, &VerticeCabeza1 );
         ObtenListaDouble( pSolucion, Arista1.lCabeza, &dDato );
         VerticeCabeza1.z = dDato;

         NuevoVertice( &VerticeCola1 );
         ObtenListaVertices( &pBD->ListaVertices, Arista1.lCola, &VerticeCola1 );
         ObtenListaDouble( pSolucion, Arista1.lCola, &dDato );
         VerticeCola1.z = dDato;

         VectorUnitario2D1 = CalculaVectorUnitario2D( &VerticeCabeza1, &VerticeCola1 );
         VectorUnitario3D1 = CalculaVectorUnitario3D( &VerticeCabeza1, &VerticeCola1 );

         /* Dada la arista 'Arista1', comprueba su paralelismo con todas las demas. */
         for( j=i+1; j<TamanyoListaAristas( &pBD->ListaAristas ); j++ ) {
                  ObtenListaAristas( &pBD->ListaAristas, j, &Arista2 );

                  NuevoVertice( &VerticeCabeza2 );
                  ObtenListaVertices( &pBD->ListaVertices, Arista2.lCabeza, &VerticeCabeza2 );
                  ObtenListaDouble( pSolucion, Arista2.lCabeza, &dDato );
                  VerticeCabeza2.z = dDato;

                  NuevoVertice( &VerticeCola2 );
                  ObtenListaVertices( &pBD->ListaVertices, Arista2.lCola, &VerticeCola2 );
                  ObtenListaDouble( pSolucion, Arista2.lCola, &dDato );
                  VerticeCola2.z = dDato;

                  VectorUnitario2D2 = CalculaVectorUnitario2D( &VerticeCabeza2, &VerticeCola2 );
                  VectorUnitario3D2 = CalculaVectorUnitario3D( &VerticeCabeza2, &VerticeCola2 );

                  dProductoEscalar = PRODUCTO_ESCALAR( VectorUnitario2D1, VectorUnitario2D2 );

                  /* Por las caracteristicas de los numeros flotantes, puede ser que dProductoEscalar
                     quede ligeramente por encima (o debajo) de 1 (o -1). Esto lo evitare con las
```

```
                lineas siguientes.                                                    */
                assert( !_isnan(dProductoEscalar) );
                if (dProductoEscalar > 1) dProductoEscalar=1;
                if (dProductoEscalar < -1) dProductoEscalar=-1;

                dAnguloRadianes = acos( dProductoEscalar );
                dAnguloGrados = RADIANES_A_GRADOS(dAnguloRadianes);

                dPesoW12 = CalculaUab( 0, 7, dAnguloGrados );

                dProductoEscalar = PRODUCTO_ESCALAR( VectorUnitario3D1, VectorUnitario3D2 );
                assert( !_isnan(dProductoEscalar) );
                if (dProductoEscalar > 1) dProductoEscalar=1;
                if (dProductoEscalar < -1) dProductoEscalar=-1;

                dAnguloRadianes = acos( dProductoEscalar );
                dAnguloGrados = RADIANES_A_GRADOS(dAnguloRadianes);

                dSumatorioParalelismo += ( dPesoW12 * (dAnguloGrados*dAnguloGrados) );
                assert( !_isnan(dSumatorioParalelismo) );

                DestruyeVertice( &VerticeCabeza2 );
                DestruyeVertice( &VerticeCola2 );
        };
        DestruyeVertice( &VerticeCabeza1 );
        DestruyeVertice( &VerticeCola1 );
};
return dSumatorioParalelismo;
};
```

5.5 Formulación de la función objetivo.

Cada regularidad está expresada por los términos matemáticos anteriormente expuestos y son incluidos en la función a objetivo tal y como se ha comentado en el capítulo anterior en la forma:

$$F = \Sigma \alpha_j R_j (z)$$

Expresión 5-12

donde:

- αj es el j-ésimo coeficiente de peso
- Rj(z) es la j-ésima regularidad.

De esta manera, los distintos términos de las regularidades dan ahora un valor que especifica las regularidades de la imagen de acuerdo con la configuración del modelo tridimensional asociado y por consiguiente son el fundamento que dirigen el problema

de optimización hacia la consecución de la forma tridimensional más adecuada, sirviendo como criterio de selección en la extensión de la proyección.

Ahora bien, dado que los algoritmos de optimización pretenden minimizar la función global, existe una mayor influencia de aquellas regularidades que por sus unidades supongan un mayor costo en la función global, a modo de ejemplo, el peso de una regularidad que depende directamente de las unidades de longitud será mucho mayor que el de aquellas regularidades que dependan directamente de las unidades de desviación angular.

Para paliar dicho problema se han establecido las siguientes soluciones:

- Imponer distintos factores ("pesos α_j") independientes para cada una de las regularidades formuladas en la función objetivo.
- Normalizar los costos de cada regularidad.
- Implementar un algoritmo de optimización que además de optimizar función objetivo, asegure minimizar cada uno de las regularidades de la función objetivo (algoritmos SAM).

Pero además dada la formulación de regularidades propuesta, puede ocurrir que el proceso de optimización no se inicie dado que muchas regularidades son ya verificadas en la solución de partida (solución trivial), por consiguiente algunos autores han incorporado las anteriormente denominadas falsas regularidades que provocan un costo inicial y por consiguiente el inicio del algoritmo de optimización.

Como justificación general de la formulación de falsas regularidades, diremos que dichas regularidades provocan un medio de salida de la solución trivial, iniciando de esta manera el proceso de optimización.

Por otra parte regularidades como la perpendicularidad de caras conducen al modelo inicial hacia un modelo convexo facilitando la búsqueda del proceso de optimización.

5.6 Implementación de la función objetivo

A continuación presentaremos el código de la función **CalculaCoste,** que dada una solución, nos devuelve su coste en función de las regularidades que tengamos activadas y del coeficiente de ponderación fijo asignado a cada una de ellas. Se subrayan aquellas líneas que son las realmente importantes y que forman parte del esqueleto de la función.

Código Fuente 5-2. Función CalculaCoste

```
double CalculaCoste(
        TListaDouble *pSolucion, TBDEntidades *pBD,
        TParametrosOptimizacion *pParametros,
        TListaDouble *pListaEvolucionCoste,
        TListaDouble *pListaEvolucionZ,
        BOOLEAN GuardarEvolucionVariables )

/* Esta es la "Funcion de Coste" del Algoritmo de Optimizacion.                */
/* Devuelve el coste de la 'pSolucion' teniendo en cuenta la Base de Datos 'pBD',*/
/* las regularidades activadas y sus coeficientes.                             */
{
        double dCosteTotal,
                aCostesRegularidades[REGULARIDAD_ULTIMA+1],
                dDato,
                dValorCoeficiente;
        int iRegularidad, i;
        TParametrosCoeficienteRegularidad Coeficiente;
        TRegularidad Regularidad;

        /* Calcula el coste de cada Regularidad que este activada */
        for( iRegularidad=0; iRegularidad < REGULARIDAD_ULTIMA+1; iRegularidad++ ) {

                Regularidad = pParametros->Regularidades.ListaRegularidades[ iRegularidad ];

                if ( Regularidad.bActivada == TRUE ) {
                        switch(iRegularidad) {
                                case REGULARIDAD_DESVIACION_ESTANDAR_ANGULOS:
                                        aCostesRegularidades[ iRegularidad ] =
CalculaCosteDesviacionEstandarAngulos( pSolucion, pBD );
                                        break;
                                case REGULARIDAD_PARALELISMO_LINEAS:
                                        aCostesRegularidades[ iRegularidad ] =
CalculaCosteParalelismoLineas( pSolucion, pBD );
                                        break;
                                case REGULARIDAD_VERTICALIDAD_LINEAS:
                                        aCostesRegularidades[ iRegularidad ] =
CalculaCosteVerticalidadLineas( pSolucion, pBD );
                                        break;
                                case REGULARIDAD_PLANICIDAD_CARAS:
                                        aCostesRegularidades[ iRegularidad ] =
CalculaCostePlanicidadCaras( pSolucion, pBD );
                                        break;
                                case REGULARIDAD_ISOMETRIA:
                                        aCostesRegularidades[ iRegularidad ] = CalculaCosteIsometria(
pSolucion, pBD );
                                        break;
                                case REGULARIDAD_ORTOGONALIDAD_ESQUINAS:
                                        aCostesRegularidades[ iRegularidad ] =
CalculaCosteOrtogonalidadEsquinas( pSolucion, pBD );
```

```
                                        break;
                                case REGULARIDAD_ORTOGONALIDAD_FACIAL_OBLICUA:
                                        aCostesRegularidades[ iRegularidad ] =
CalculaCosteOrtogonalidadFacialOblicua( pSolucion, pBD );
                                        break;
                                case REGULARIDAD_ORTOGONALIDAD_LINEAS:
                                        aCostesRegularidades[ iRegularidad ] =
CalculaCosteOrtogonalidadLineas( pSolucion, pBD );
                                        break;
                                case REGULARIDAD_PERPENDICULARIDAD_CARAS:
                                        aCostesRegularidades[ iRegularidad ] =
CalculaCostePerpendicularidadCaras( pSolucion, pBD );
                                        break;
                                case REGULARIDAD_COLINEALIDAD_LINEAS:
                                        aCostesRegularidades[ iRegularidad ] =
CalculaCosteColinealidadLineas( pSolucion, pBD );
                                        break;
                        };

        ........

                        /* Finalmente, multiplica por el coeficiente fijo. */
                        aCostesRegularidades[ iRegularidad ] *= Regularidad.dCoeficienteFijo;
                }
                else
                        /* Si la Regularidad no esta activada, su coste es 0. */
                        aCostesRegularidades[ iRegularidad ] = 0;
        };


        /* EL COSTE TOTAL ES LA SUMA DE LOS COSTES DE TODAS LAS REGULARIDADES */
        dCosteTotal = 0;
        for( iRegularidad=0; iRegularidad < REGULARIDAD_ULTIMA+1; iRegularidad++ )
                dCosteTotal += aCostesRegularidades[ iRegularidad ];

        ........

        return dCosteTotal;
};
```

6. DISEÑO, IMPLEMENTACIÓN Y PRUEBA DE REFER

6.1 Planteamiento estratégico

El objetivo de este Proyecto Final de Carrera es diseñar y construir una aplicación que permita implementar adecuadamente los algoritmos de reconstrucción geométrica vistos en los Capítulos anteriores. Esta aplicación la bautizamos como **REFER**.

La aplicación tiene estos objetivos básicos:

1. LEER IMAGEN 2D: Leer un dibujo desde un fichero con algún formato gráfico estandarizado.

2. VISUALIZAR IMAGEN 2D: Mostrar el dibujo en pantalla, y poder tener funciones de edición básicas, para poder corregir errores en el dibujo y filtrar todo lo que no sean aristas y vértices, que son las únicas entidades de partida que utilizarán los algoritmos.

3. GENERAR MODELO 3D: Implementar la reconstrucción geométrica, con algún método de optimización y utilizando regularidades.

4. VISUALIZAR MODELO 3D: Visualizar el objeto 3D resultante, pudiendo mover el objeto para ver el resultado desde cualquier ángulo.

5. Un último objetivo no escrito, pero presente desde el principio, era que esta aplicación debía ser el punto de partida de un ambicioso proyecto de investigación, por tanto tenía que estar diseñada pensando en ampliaciones futuras por parte de otra gente.

Esta ultima reflexión fue muy importante, porque condicionó en gran medida las decisiones iniciales de diseño.

Más tarde se ha comprobado la validez de este planteamiento, ya que hasta el momento se están desarrollando dos tesis doctorales que se apoyan en esta aplicación.

6.2 Planteamiento táctico

6.2.1 Introducción

Cuando mi Director y yo decidimos iniciar este proyecto, mediados de 1997, nos planteamos sobre que plataforma de programación y ejecución trabajaríamos. Dadas las características de este PFC en cuanto a su previsible larga duración y dedicación, y para facilitar el trabajo en casa, decidimos que lo más práctico sería utilizar la plataforma "Wintel", esto es, Microsoft Windows y procesador compatible Intel. Además es la que ofrecía mayor variedad de herramientas, documentación e información disponible.

Con esta situación de trabajo veamos con detalle la elección en los apartados más significativos: Hardware, Sistema Operativo, Herramientas de Programación y Librería Gráfica.

6.2.1.1 Hardware

La elección del Hardware venía condicionada por la necesidad de poder trabajar en casa, con un ordenador compatible PC. De no ser así, quizá nos hubiéramos planteado trabajar con potentes estaciones de trabajo Unix de la Universidad. Esta primera elección del Hardware nos condicionó el Software que teníamos disponible. Pero como ya hemos dicho antes, la plataforma Wintel es la que ofrecía mayor cantidad de software y documentación.

La aplicación debía correr en un PC estándar del año que se inició el proyecto. En aquel año un PC típico era un Pentium 166-200 MMX con 16-32 MB de memoria RAM, disco duro de 2 GB, con tarjeta gráfica aceleradora 2D (sin aceleración 3D) de 2MB.

Cuando comenzó este PFC, mediados de 1997, la situación informática difería bastante de la actual, mediados de 2000. Según la conocida *"Ley de Moore"*[7] los PC eran 4 veces menos potentes que los actuales. Y es cierto, pues en el momento de escribir esta memoria un PC típico es un Pentium III 600-800 con 64-128 MB de memoria RAM, disco duro de 10 GB, y tarjeta gráfica aceleradora 2D/3D de 8-16 MB.

[7] Moore era el presidente de Intel que a finales de los 70 vaticinó que "cada 18 meses los procesadores duplicarán su potencia". La experiencia demuestra que hasta la fecha se ha cumplido esta "ley no escrita".

6.2.1.2 Sistema Operativo. Microsoft Windows.

Nuestra aplicación funcionaría bajo el sistema operativo con interfaz gráfico de usuario más popular del momento, Microsoft Windows 95. Durante la realización de este PFC apareció en el mercado el Microsoft Windows 98, que en la práctica no es sino una mejora o actualización del anterior Windows 95. Funcionalmente son idénticos, por lo que comúnmente se les ha dado en llamar Windows 9x. Estos son sistemas operativos orientados al usuario doméstico, las aplicaciones multimedia y los juegos. A su vez, Microsoft ofrece para los entornos empresariales Windows NT (en sus distintas versiones: 3.x y 4), el "auténtico" sistema operativo de Microsoft, diseñado para ser fiable, seguro y estable, y con muchísimo mejor futuro por delante que su hermano menor Windows 9x.

Ya se hablaba del sistema operativo Linux (Unix gratuito), pero todavía era complicado de instalar y no era muy conocido. Para Windows había mejores herramientas de programación y más bibliografía.

6.2.1.3 Elección del Entorno de Programación

La aplicación estaría compilada según una arquitectura de 32 bits para obtener el máximo rendimiento, a costa de que no pudiera funcionar en el ya obsoleto Microsoft Windows 3.1. Así también se podría ejecutar en Microsoft Windows NT 4.0.

Para el usuario, Windows es un entorno multitarea basado en ventanas que se corresponden con programas. Esto significa que permite ejecutar concurrentemente programas especialmente escritos para dicho entorno y también incluso programas escritos para el antediluviano MS-DOS.

Para desarrollar programas, Windows provee rutinas que permiten utilizar componentes como menús, cuadros de diálogo y barras de desplazamiento entre otros. Así mismo, el programador puede tratar el ratón, el teclado, el monitor, la impresora, los puertos de comunicaciones y el reloj del sistema sin tener en cuenta las particularidades del dispositivo periférico.

Para programar una aplicación que corriera en Windows, con una interfaz gráfica de usuario amigable, había varias opciones, entre las que destacaban:

- Borland Delphi
- Microsoft Visual Basic
- Microsoft Visual C++

Yo personalmente prefería Borland Delphi por varias razones. En primer lugar, porque era una herramienta que conocía bien ya que anteriormente había realizado un programa visualizador gráfico 3D, para unas prácticas de una asignatura de la Facultad. Es un muy buen entorno de desarrollo. Y el lenguaje que utiliza es una evolución del Turbo Pascal orientado a objetos, muy utilizado en nuestra Facultad de Informática.

Si no lo elegimos fue principalmente por el objetivo 5 de REFER. Tenía el problema de la compatibilidad futura. Nosotros queríamos que el programa se pudiera ampliar posteriormente, y no quedar anclado a una herramienta de programación del momento. Es más, pretendíamos que se pudiera aprovechar parte del código y llevarlo a una estación gráfica Unix, si fuera necesario. Esta ultima razón nos dirigió hacia el lenguaje C.

También teníamos el Microsoft Visual Basic, similar al Delphi, pero mucho menos serio. Además, tampoco permitiría evolucionar a Unix.

Por tanto, si queríamos máxima compatibilidad, teníamos que utilizar C, lo que nos llevaba irremediablemente al Microsoft Visual C++.

Esta herramienta es un excelente entorno de desarrollo, que permite programar todo tipo de aplicaciones, utilizando C y C orientado a objetos (C++). Puede compilar aplicaciones de 32 bits (Win32), que son las que consiguen mayor rendimiento. Nos ofrece la librería MFC ó Microsoft Foundation Classes que es un conjunto de clases base de C++ para desarrollar aplicaciones para Windows que utilicen la interfaz gráfica de usuario.

Así pues, para programar la interfaz se utilizaría el C++ y las MFC, y en el resto se utilizaría C estándar.

6.2.1.3.1 Microsoft Visual C++

Microsoft Visual C++ es un entorno de programación en el que se combinan la programación orientada a objetos (C++) y el sistema de desarrollo diseñado especialmente para crear aplicaciones gráficas para Windows (SDK).

Aunque las aplicaciones Windows son sencillas de utilizar, el desarrollo de las mismas no es fácil. Por ello, para hacer más asequible esta tarea, Visual C++ incluye, además de varias herramientas que lo convierten en un generador de programas C++, un conjunto completo de clases (Microsoft Foundation Classes, MFC) que permiten crear de una forma intuitiva las aplicaciones para Windows y que permiten manejar los componentes de Windows según a su naturaleza de objetos. Esto es, la librería MFC es una implementación que utiliza las funciones de la API de Windows, encapsulando todas las estructuras y llamadas a dichas funciones en objetos fáciles de utilizar.

Visual C++, orientado al desarrollo de aplicaciones para Windows está centrado en dos tipos de objetos, *ventanas* y *controles*, que permiten diseñar sin programar, una interfaz gráfica para una aplicación. Para realizar una aplicación se crean *ventanas*, a veces llamados *formularios*, y sobre ellas se dibujan otros objetos llamados *controles*. A continuación se escribe el código fuente relacionado con cada objeto.

Quiere esto decir, que cada *objeto* (*ventanas* y *controles*) está ligado a un código que permanece inactivo hasta que se dé el suceso que lo activa. Por ejemplo, podemos programar un botón (objeto que se puede pulsar) que responda a un clic del ratón.

La versión que hemos utilizado durante todo el proyecto ha sido Microsoft Visual C++ 5.0

6.2.1.3.2 Librería Microsoft Foundation Class (MFC)

Windows fue diseñado mucho antes que el popular lenguaje C++. Por este motivo hasta que apareció C++, prácticamente la totalidad de las aplicaciones para Windows se desarrollaban utilizando el lenguaje C y la librería de funciones de la API de Windows. No obstante, estas aplicaciones fueron construidas pensando en objetos, por tanto, un lenguaje orientado a objetos como C++ es lo más idóneo para construir una interfaz natural para desarrollar este tipo de aplicaciones.

La librería MFC (*Microsoft Foundation Class library* – librería de clases base de Microsoft), constituye verdaderamente una interfaz orientada a objetos para Windows, que permite desarrollar aplicaciones para Windows de una forma más intuitiva que la forma tradicional. Una aplicación desarrollada utilizando las MFC comparada con una versión de la misma en el sistema tradicional, contiene menos código, tiene una velocidad de ejecución comparable y también, permite llamadas a las funciones del lenguaje C y de la API de Windows.

Las MFC en ningún momento tratan de reemplazar a las funciones de la API de Windows. Una función de Windows es cubierta por una función miembro de una clase sólo si hacerlo supone una clara ventaja. Esto significa que en algunas ocasiones tendremos que hacer llamadas a las funciones nativas de Windows.

Las MFC están estrechamente ligadas con objetos tales como ventanas (ventanas marco, ventanas de diálogo, botones, cajas de texto, etiquetas, etc.), menús, contextos de dispositivo (pantallas, impresoras, etc.) y dispositivos gráficos (*bitmaps*, *fonts*, *pens*, etc.), generalmente utilizados en el diseño de una aplicación.

En la figura siguiente podemos apreciar la extensión y la complejidad del árbol de clases y sus herencias. Es una estructura realmente completa.

Figura 6-1. Estructura de las MFC

6.2.1.4 Librería Gráfica. OpenGL

Uno de los puntos clave del PFC era mostrar en pantalla el objeto 3D recién generado y poder moverlo interactivamente, para ver si el resultado de la optimización era el esperado.

Podíamos habernos hecho nosotros mismos una librería de funciones que nos permitiese dibujar objetos 3D en un programa Windows con Visual C++. Pero no tenía mucho sentido crearnos una librería nueva, y perder tiempo en programarla (de hecho yo ya había programado algo similar con Borland Delphi en la Universidad). Pensamos que era mejor, más rápido y más didáctico aprender y utilizar alguna librería estándar que estuviera disponible en el mercado.

Cuando se inició este PFC, Microsoft ofrecía la librería DirectX versión 3.0, fundamentalmente para programar juegos. Como pasa en casi todos los productos de Microsoft, las primeras versiones de DirectX dejaban bastante que desear y no convencían a casi nadie[8]. También existía una librería para juegos llamada *Glide* , propietaria del fabricante de las mejores tarjetas 3D para juegos del momento: *3dfx.*

Frente estas dos alternativas orientadas al mercado lúdico, teníamos a OpenGL como la mejor librería de gráficos 3D en cuanto a calidad visual, además de ser una norma abierta controlada por la Plataforma para Revisiones de la Arquitectura OpenGL (*OpenGL Arquitecture Review Board,* ARB), cuyos miembros fundadores son SGI, Digital Equipament Corporation, IBM, Intel y Microsoft.

Así pues, por calidad y por futuro, elegimos a OpenGL como la librería que utilizaríamos para visualizar nuestros modelos 3D.

El 1 de julio de 1992 se presentó la versión 1.0 de la especificación OpenGL. Justo cinco días después, en una de las primeras jornadas para desarrolladores de Win32, SGI realizó algunas demostraciones de OpenGL funcionando en su hardware IRIS Indigo. La proyección de secuencias de películas como *Terminator2*, *El Juicio Final* y aplicaciones médicas gráficas fueron atracciones populares en la sala de

[8] Pero como acaba pasando en casi todos los productos de Microsoft, la compañía supo reaccionar y mejorar su producto; actualmente la librería DirectX 6.1 y la más reciente DirectX 7.0 es el auténtico estándar en cuanto a gráficos 3D para juegos dentro de la plataforma Wintel.

exhibición de empresas. Mientras tanto, SGI y Microsoft estaban trabajando conjuntamente para incluir OpenGL en una futura versión de Windows NT.

OpenGL se define estrictamente como "una interfaz software para gráficos por hardware". En esencia, es una librería de gráficos 3D y modelado portátil y muy rápida. Usando OpenGL, se pueden crear gráficos 3D con una calidad visual cercana a la de un ray-tracer (o trazador de rayos) con la gran ventaja de ser mucho más rápido.

Emplea algoritmos cuidadosamente desarrollados y optimizados por Silicon Graphics, Inc. (SGI), un reconocido líder mundial en los gráficos y animaciones por ordenador.

OpenGL se propone para el empleo de hardware informático diseñado y optimizado para la visualización y manipulación de gráficos 3D. Las implementaciones genéricas de OpenGL, que sólo constan de software sin hardware, también son posibles, y en esta categoría entran las implementaciones de Windows NT y Windows 9x.

Muy pronto no será este el caso excepcional, puesto que cada vez más y más empresas de hardware gráfico para PCs están añadiendo aceleración 3D a sus productos, influenciados principalmente por las exigencias del mercado de juegos 3D, hecho este que tiene un paralelismo cercano a la evolución de las aceleradoras gráficas 2D, basadas en Windows, que optimizan operaciones como el trazado de líneas y el relleno y manipulación de mapas de bits. Al igual que hoy nadie se plantea el uso de una tarjeta VGA ordinaria para ejecutar Windows, en poco tiempo las tarjetas aceleradoras de gráficos 3D se convertirán en algo común. De hecho la mayoría de tarjetas 3D lanzadas actualmente ofrecen drivers que soportan en mayor o menor medida la aceleración hardware de OpenGL.

Microsoft tiene sus propias librerías gráficas 3D para el mercado multimedia y de juegos. Son las Direct3D, y están incluidas dentro de la Microsoft DirectX Foundation.

La DirectX Foundation proporciona a los desarrolladores un sencillo conjunto de APIs (Interfaces de programación de Aplicaciones) que proporcionan mejoras en el acceso a las características avanzadas del hardware de alto rendimiento, como son las aceleradoras 3D y las tarjetas de sonido. Estas APIs que son llamadas «de bajo nivel»

incluyen aceleración 2D, soporte para dispositivos de entrada como joysticks, teclados y ratones, y controlan la mezcla del sonido y la salida de éste. Las funciones de bajo nivel de DirectX son soportadas por los siguientes componentes que conforman la DirectX Foundation Layer: DirectDraw, Direct3D, Directlnput, DirectSound, DirectPlay y DirectMusic. Antes de la aparición de DirectX, los programadores que creaban contenido multimedia para ordenadores Windows habían de adaptar sus productos para que funcionasen en un amplio abanico de configuraciones hardware.

DirectX Foundation proporciona algo llamado *Hardware Abstraction Layer* (HAL), que es un software que hace de intermediario entre la aplicación y el ordenador. Como resultado, los desarrolladores pueden escribir una simple versión de su producto que utilice DirectX sin tener que preocuparse sobre el amplio rango de dispositivos hardware y configuraciones que existan para la plataforma Windows.

La DirectX Foundation proporciona también a los desarrolladores herramientas que pueden ayudarles a conseguir el mejor rendimiento de la máquina que están usando. Automáticamente determina las posibilidades del hardware del ordenador y entonces ajusta los parámetros de la aplicación para que funcione correctamente. DirectX además permite correr aplicaciones multimedia que requieren el soporte de características que no posee el sistema, emulando ciertos dispositivos hardware mediante *Hardware Emulation Layer* (HEL), que son unos drivers por software que funcionan como si se tratase de hardware. Por ejemplo, un juego DirectX que utilice funciones 3D podrá funcionar en un ordenador que no tenga aceleración 3D porque DirectX emula los servicios que proporciona una aceleradora 3D.

Todas las tarjetas 3D que se venden en el mercado tienen drivers DirectX para explotar completamente sus posibilidades. Algunos fabricantes, pocos pero cada vez más, también proporcionan drivers OpenGL completos. Hay otros que proporcionan drivers OpenGL que internamente hacen llamadas a la librería DirectX; esta es una forma rápida, aunque no óptima, de desarrollar y proporcionar el soporte OpenGL.[9]

[9] En el momento de escribir esta memoria se anuncia el acuerdo entre SGI y Microsoft para crear unas librerías gráficas de gran potencia, que cristalizarán en la siguiente versión de DirectX. Parece ser que OpenGL quedará integrado en la próxima DirectX versión 8.0

OpenGL no estaba disponible en las versiones de 16 bits de Microsoft Windows (3.1, 3.11 y otras). Microsoft añadió OpenGL a Windows NT 3.5 y a Windows 95 a través de una distribución separada de algunas DLLs. La primera entrega de DLLs realizada por Microsoft para Windows NT soportaba todas las funciones de OpenGL 1.0, que también están disponibles bajo Windows NT 3.5 y 3.51. Las funciones OpenGL 1.1 han sido añadidas posteriormente a Windows NT 4.0 y Windows 98.

Desde el punto de vista de la programación, el uso de OpenGL bajo Windows 9x es el mismo que el de OpenGL bajo Windows NT.

OpenGL es un lenguaje procedimental más que un lenguaje descriptivo. En lugar de diseñar la escena y cómo debe aparecer, el programador describe los pasos necesarios para conseguir una cierta apariencia o efecto. Estos "pasos" implican llamadas a una API altamente portátil que incluye aproximadamente 120 comandos y funciones. Estos se emplean para dibujar primitivas gráficas como puntos, líneas y polígonos en tres dimensiones. Además, OpenGL soporta iluminación y sombreado, mapeado con texturas, animación y otros efectos especiales.

OpenGL no incluye ninguna función para la gestión de ventanas, interactividad con el usuario ni manejo de ficheros. Cada entorno anfitrión (como Microsoft Windows) tiene sus propias funciones para este propósito y es responsable de implementar alguna manera de facilitar a OpenGL la facultad de dibujar en una ventana o mapa de bits.

6.2.2 Planificación del trabajo

De acuerdo con el desglose de objetivos propuestos anteriormente, el trabajo de desarrollo del presente PFC se descompone en el conjunto de actividades siguientes:

1. Aprendizaje del Entorno de Desarrollo:
    - 1.1. Aprendizaje del Microsoft Visual C++ y la librería MFC.
    - 1.2. Aprendizaje de la librería gráfica OpenGL.
2. Recopilación de información
    - 2.1. Recopilación de información sobre reconstrucción geométrica en general, y, en especial, sobre reconstrucción a partir de vista única y por el método de optimización.

2.2. Recopilación de información sobre vectorización y sobre identificación de información de diseño normalizada por medio de símbolos gráficos.

3. Estudio:

3.1. de modelos de optimización.

⇒ Establecimiento de una función objetivo válida para un rango de modelos lo más general posible, que contemple todas las características geométricas del modelo que están contenidas de forma explícita o implícita en la figura que lo representa.

⇒ Especificación de la estructuración de dichas tareas en un Modelo de Usuario.

3.2. de métodos de optimización.

⇒ Análisis y selección del, o de los, métodos de optimización apropiados.

⇒ Selección de información tratable por los formatos de intercambio.

3.3. de protocolos más habituales de intercambio de ficheros de las principales aplicaciones CAD.

4. Implementación:

4.1. Sistema preprocesador.

⇒ Lectura de ficheros tipo DXF, DWG, DGN, IGES, etc.

⇒ Visualización de las imágenes vectoriales capturadas en los formatos anteriores.

⇒ Almacenamiento de las imágenes en los formatos considerados.

4.2. Módulo de edición.

⇒ Manipulación del modelo (eliminar primitivas representadas, añadir primitivas al diseño, ...)

⇒ Manipulación de la visualización del modelo (zoom, encuadres, desplazamientos y giros).

4.3. Módulo de reconstrucción.

⇒ Reconocimiento de relaciones geométricas de la imagen vectorial.

⇒ Implementación de los algoritmos de optimización adecuados de acuerdo las particularidades del modelo.

⇒ Definición de la función objetivo y restricciones para la optimización.

5. Pruebas:

5.1. Análisis de los resultados obtenidos para modelos poliédricos.

6. Conclusiones finales y propuestas de actividades futuras.

6.2.3 Coste económico de la aplicación

Por lo que se refiere a la temporalidad, el conjunto de actividades descrito arriba se desarrollarán de forma básicamente consecutiva, aunque dada la naturaleza de algunas actividades, se considera conveniente desarrollarlas en paralelo. Por tanto, la

temporalización estimada para las actividades propuestas se resume en la siguiente tabla:

TABLA DE ACTIVIDADES

ACTIVIDAD	DURACIÓN (MES)																	
	1º	2º	3º	4º	5º	6º	7º	8º	9º	10º	11º	12º	13º	14º	15º	16º	17º	18º
1.1 Aprendizaje Visual C++ y MFC																		
1.2 Aprendizaje OpenGL																		
2.1 Recopilación Inf. de reconstrucción																		
2.2 Recopilación Inf. de reconstrucción																		
3.1 Estud. de modelos de optimización																		
3.2 Estud. métodos de optimización																		
3.3 Estud. protocolos ficheros CAD																		
4.1 Sistema preprocesador																		
4.2 Módulo de edición																		
4.3 Módulo de reconstrucción																		
5.1 Análisis modelos poliédricos																		
6 Conclusiones finales																		

Periodo de dedicación prioritaria a la correspondiente actividad

Periodo de dedicación complementaria a la correspondiente actividad

Periodo sin dedicación a la actividad

En la tabla anterior hemos supuesto una jornada laboral de 8 horas, 20 días/mes de trabajo, y que estos 18 meses sería el tiempo que le costaría a un solo programador dedicado completamente a esta tarea.

Supongamos que se contrata a una empresa externa la realización de este proyecto., y que el coste de un programador de esta empresa es de 5000 pts/hora.

El coste total del proyecto sería de:

18 meses x 20 dias/mes x 8 horas/dia x 5000 pts/hora = **14.400.000 pts**

que equivalen a **86.545'74 €** *(1€ = 166'386 pts)*

Hay que tener en cuenta que se dedican 6 meses al aprendizaje de Visual C++ y OpenGL. Si se contratara una empresa con experiencia en estas herramientas, el coste se reduciría quedando en 9.600.000 pts.

En cualquier caso, es más rentable tener un programador contratado en la empresa dedicado exclusivamente a esta tarea. Con un salario bruto de, digamos, 350.000 pts./mes el coste de estos 18 meses sería de 6.300.000 pts.

6.3 Programación de REFER

6.3.1 La Base de Datos de Entidades

6.3.1.1 Necesidad de una Base de Datos de Entidades

Como veremos luego, el formato de fichero gráfico estándar que decidí utilizar fue el DXF de AutoCAD. En AutoCAD los objetos gráficos como las lineas, los circulos, los bloques de texto, etc, reciben el nombre de *entities*. Así que decidí mantener la nomenclatura y llamarlos **entidades.**

Los datos de estas entidades los tendría que manejar a lo largo de todo el programa: para pintar los objetos en pantalla, para calcular las regularidades, para aplicar el algoritmo de optimización, para manejar los objetos 3D ya reconstruidos, etc.

Por tanto, necesitaba un mecanismo lo bastante potente para gestionar fácilmente estas entidades. En concreto, los requisitos eran:

1. Estructura de datos de tamaño flexible, ya que a priori no sabemos cuantas entidades podremos encontrar. No es recomendable utilizar arrays, pues no sabremos su tamaño, así que hay que utilizar asignación dinámica de memoria.

2. Funciones autoexplicativas que permitan manejar las entidades de una forma lo más sencilla posible, sin tener que manipular directamente las estructuras de datos internas. La utilización de estas funciones previene errores y asegura la integridad de las entidades: por ejemplo, si se elimina un vértice, hay que eliminar su arista asociada.

3. Estructura y Funciones fácilmente transportables a otras plataformas, utilizando el lenguaje C. Estas entidades son además independientes del formato gráfico empleado, sea DXF o cualquier otro en el futuro.

Por su gran parecido a una base de datos, lo bauticé precisamente así: **Base de Datos de Entidades**. Está programada enteramente en C, y básicamente la estructura consiste en un conjunto de listas de entidades.

Llamaré **lista de entidades** a una estructura de tipo lista, gestionada mediante punteros para conseguir una asignación dinámica de memoria, en función de las necesidades en tiempo real del algoritmo.

6.3.1.2 Diseño de una Base de Datos de Entidades

Como se puede ver en el cuadro siguiente, en una estructura de tipo TBDEntidades tendremos una lista de Vértices, una lista de Aristas, una lista de Caras, una lista de Cadenas de Texto, una lista de Circunferencias, y una lista de Capas.

BDdatos.h

```
/* BASE DE DATOS GENERAL DE TODAS LAS ENTIDADES : 'TBDEntidades' */
typedef struct {
      TListaVertices        ListaVertices;
```

```
        TListaAristas          ListaAristas;
        TListaCaras            ListaCaras;
        TListaCadenasTexto     ListaCadenasTexto;
        TListaCircunferencias  ListaCircunferencias;
        TListaCapas            ListaCapas;
} TBDEntidades;
```

A partir del fichero de entrada DXF se crean todas estas listas menos la lista de Caras, porque esta información no se encuentra en dicho fichero, ni la necesitamos. Las regularidades que implementaremos no necesitan información de caras. Hay regularidades que si necesitaran conocer las caras, y se tendrá que hacer un algoritmo de detección de caras, pero esto se sale del ámbito de este PFC[10].

Para hacer pruebas, REFER tiene una utilidad de edición de caras, que permite definirlas editando directamente el dibujo 2D.

Para mostrar básicamente cómo funcionan estas estructura de datos, mostraremos aquí la definición de datos, y cómo se ha programado una lista de enteros (números enteros). No se pretende mostrar mucho más código porque se puede ir directamente a los ficheros fuentes que están completamente comentados y explicados. En cambio, si se ve la estructura y las funciones disponibles se puede apreciar la utilidad de estos módulos y el trabajo dedicado a desarrollarlos.

Código Fuente 6-1. Estracto de BDatos.h

```
/* CAPA (LAYER) DE DIBUJO */
typedef struct {
        char sTexto[TAMANYO_TEXTO_CAPA];
        BOOLEAN bActiva;
} TCapa;

/* LISTA DE CAPAS.
        Esta lista varia su tamaño reservando memoria en tiempo de ejecucion.
        NOTA: Aunque 'aLista' se define como puntero, se utiliza como ARRAY */
typedef struct {
        TCapa *aLista;
        long lTamanyo;
} TListaCapas;

/* VERTICE EN COORDENADAS "REALES" */
typedef struct {
        double x, y, z;
        TListaLong ListaAristas;
        BOOLEAN bPresente;
```

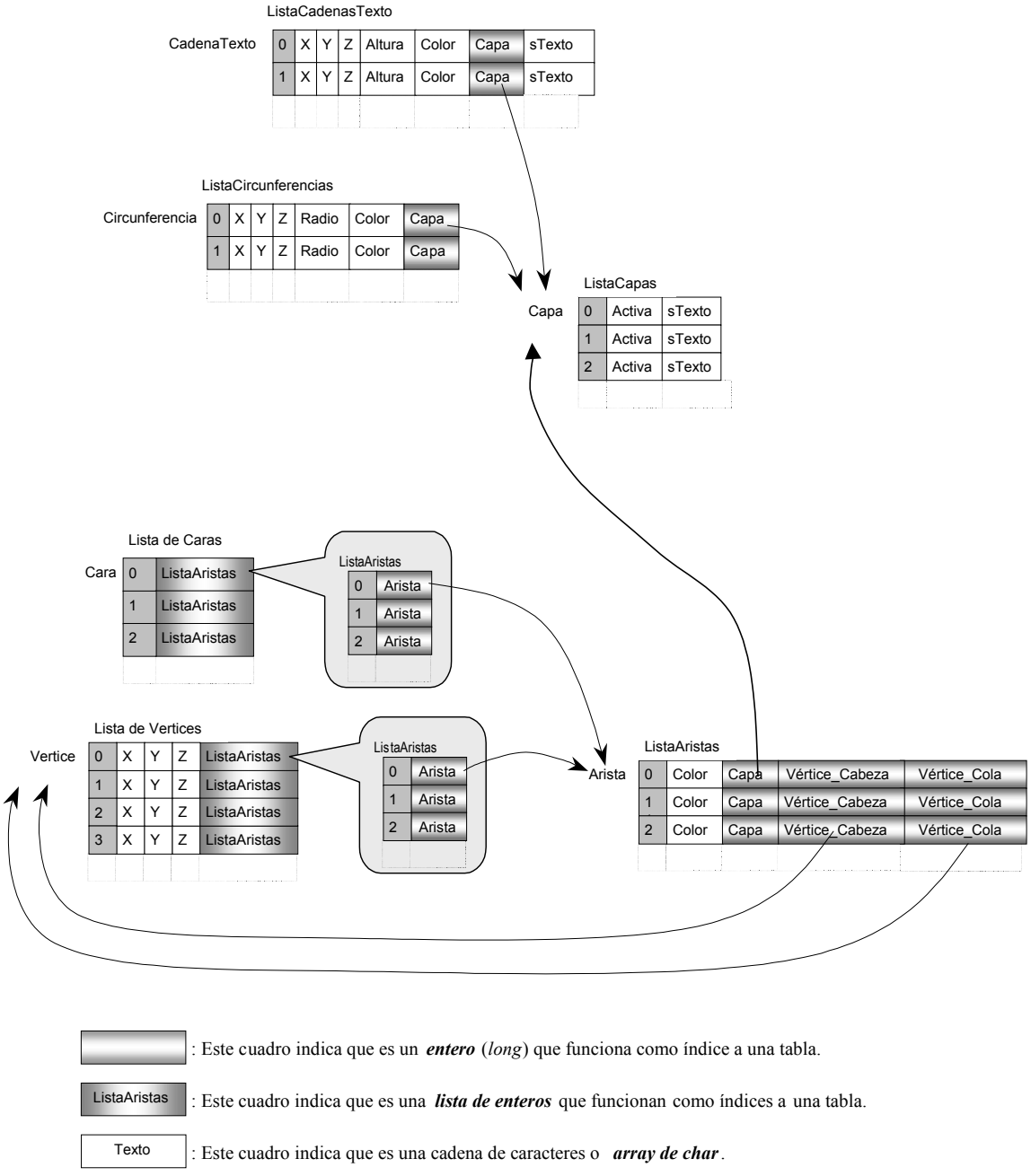
[10] Este algoritmo de detección de caras lo desarrolla Vicente Camarena en su PFC.

```
        BOOLEAN bSeleccionado;
} TVertice;

/* LISTA DE VERTICES.
        Esta lista varia su tamaño reservando memoria en tiempo de ejecucion.
        NOTA: Aunque 'aLista' se define como puntero, se utiliza como ARRAY */
typedef struct {
        TVertice *aLista;
        long lTamanyo;
} TListaVertices;

/* ARISTA :  Cola ---> Cabeza  */
typedef struct {
        long lCabeza, lCola;
        long lCapa;
        int iColorR, iColorG, iColorB,
            iEstilo;
        BOOLEAN bPresente;
        BOOLEAN bSeleccionada;
} TArista;

/* LISTA DE ARISTAS.
        Esta lista varia su tamaño reservando memoria en tiempo de ejecucion.
        NOTA: Aunque 'aLista' se define como puntero, se utiliza como ARRAY */
typedef struct {
        TArista *aLista;
        long lTamanyo;
} TListaAristas;

/* CARA : Basicamente es una lista de Aristas */
typedef struct {
        TListaLong ListaAristas;
        BOOLEAN bSeleccionada;
} TCara;

/* LISTA DE CARAS.
        Esta lista varia su tamaño reservando memoria en tiempo de ejecucion.
        NOTA: Aunque 'aLista' se define como puntero, se utiliza como ARRAY */
typedef struct {
        TCara *aLista;
        long lTamanyo;
} TListaCaras;

/* CIRCUNFERENCIA */
typedef struct {
        double x, y, z, dRadio;
        long lCapa;
        int iColorR, iColorG, iColorB,
            iEstilo;
} TCircunferencia;

/* LISTA DE CIRCUNFERENCIAS.
        Esta lista varia su tamaño reservando memoria en tiempo de ejecucion.
        NOTA: Aunque 'aLista' se define como puntero, se utiliza como ARRAY */
typedef struct {
        TCircunferencia *aLista;
        long lTamanyo;
} TListaCircunferencias;

/* CADENA DE CARACTERES O TEXTO  */
typedef struct {
        double x, y, z, dAltura;
        char sTexto[TAMANYO_TEXTO_CADENATEXTO];
        long lCapa;
        int iColorR, iColorG, iColorB,
            iEstilo;
} TCadenaTexto;
```

```
/* LISTA DE CADENAS DE CARACTERES.
         Esta lista varia su tamaño reservando memoria en tiempo de ejecucion.
         NOTA: Aunque 'aLista' se define como puntero, se utiliza como ARRAY */
typedef struct {
         TCadenaTexto *aLista;
         long lTamanyo;
} TListaCadenasTexto;
```

En la figura siguiente se puede ver una representación gráfica que permite comprender las relaciones que existen entre los componentes de la Base de Datos de Entidades.

Esquema básico de la Base de Datos de Entidades



: Este cuadro indica que es un ***entero*** (*long*) que funciona como índice a una tabla.

ListaAristas : Este cuadro indica que es una ***lista de enteros*** que funcionan como índices a una tabla.

Texto : Este cuadro indica que es una cadena de caracteres o ***array de char***.

Figura 6-2. Esquema básico de la Base de Datos de Entidades

Código Fuente 6-2. Estracto de ListaLong.h

```
/* LIBRERIA QUE CONTIENE LA ESTRUCTURA DE DATOS Y FUNCIONES PARA EL DESARROLLO Y
   MANTENIMIENTO DE UNA LISTA DE ENTEROS "LONG", SIN TAMAÑO INICIAL Y QUE AUMENTA Y
   DISMINUYE DE TAMAÑO CONFORME VA NECESITANDO MAS MEMORIA EN TIEMPO DE EJECUCION.  */
/* Autor: Juan Vicente Andreu Hernandez   - Castellon'98 -                          */


/*-----------------------------------------------*/
/* DEFINICION DE "TIPOS" Y ESTRUCTURAS DE DATOS */

/* Esta lista varia su tamaño reservando memoria en tiempo de
   ejecucion.
   NOTA: Aunque 'aLista' se define como puntero, se utiliza como ARRAY */

typedef struct {
        long *aLista;
        long lTamanyo;
} TlistaLong;
```

Código Fuente 6-3. Estracto de ListaLong.c

```
#include <stdlib.h>
#include "ListaLong.h"


/*-------------------------------------------------------------*/
/* PROCEDIMIENTOS DE MANEJO DEL TIPO LISTA DE ENTEROS "LONG" */


void NuevaListaLong(TlistaLong *pL)
/* INICIALIZA LA LISTA.
   ¡¡IMPORTANTE!! :ES LA PRIMERA OPERACION A REALIZAR. POSTERIORMENTE NO SE DEBE UTILIZAR. */
{
        pL->lTamanyo = 0;
        pL->aLista = NULL;
};


void DestruyeListaLong(TlistaLong *pL)
/* ELIMINA TODOS LOS ENTEROS DE LA LISTA, LIBERANDO TODA SU MEMORIA OCUPADA. */
{
        free(pL->aLista);
        pL->lTamanyo=0;
};


long TamanyoListaLong(TlistaLong *pL)
/* DEVUELVE EL TAMAÑO (NUMERO DE ELEMENTOS) DE LA LISTA */
{
        return (pL->lTamanyo);
};


ESTADO AnyadeListaLong(TlistaLong *pL, long lDato)
/* OBTIENE MAS MEMORIA DINAMICAMENTE, Y AÑADE EL ENTERO 'lDato' AL FINAL DE LA LISTA.
```

```
   DEVUELVE 'OK' SI TODO ES CORRECTO. */
{
        if ( (pL->aLista = (long *) realloc( pL->aLista, sizeof(long)*(pL->lTamanyo+1) )) == NULL )
                return ERROR_ASIGNACION_MEMORIA;
        pL->aLista[pL->lTamanyo] = lDato;
   pL->lTamanyo++;
   return OK;
};


ESTADO ObtenListaLong(TListaLong *pL, long lPosicion, long *lDato)
/* DEVUELVE EN 'lDato' EL ENTERO DE LA POSICION 'lPosicion'.
   DEVUELVE 'OK' SI TODO ES CORRECTO. */
{
   if ( (lPosicion<0) || (lPosicion >= pL->lTamanyo) )
                return ERROR_ELEMENTO_NO_ENCONTRADO;
   *lDato = pL->aLista[lPosicion];
   return OK;
};


ESTADO InsertaListaLong(TListaLong *pL, long lPosicion, long lDato)
/* OBTIENE MAS MEMORIA DINAMICAMENTE, E INSERTA EL ENTERO 'lDato' EN LA POSICION 'lPosicion'.
   DEVUELVE 'OK' SI TODO ES CORRECTO. */
{
   long i;

   if ( (lPosicion<0) || (lPosicion >= pL->lTamanyo) )
                return ERROR_ELEMENTO_NO_ENCONTRADO;
   if ( (pL->aLista = (long *) realloc( pL->aLista, sizeof(long)*(pL->lTamanyo+1) )) == NULL )
                return ERROR_ASIGNACION_MEMORIA;
   pL->lTamanyo++;

   /* Desplaza todos los elementos hacia atras a partir de 'lPosicion' */
   for( i = pL->lTamanyo-1 ; i > lPosicion; i-- )
                pL->aLista[i] = pL->aLista[i-1];
   pL->aLista[lPosicion] = lDato;
   return OK;
};


ESTADO EliminaListaLong(TListaLong *pL, long lPosicion)
/* ELIMINA EL ENTERO EN LA POSICION 'lPosicion', LIBERANDO MEMORIA.
   DEVUELVE 'OK' SI TODO ES CORRECTO. */
{
   long i;

   if ( (lPosicion<0) || (lPosicion >= pL->lTamanyo) )
                return ERROR_ELEMENTO_NO_ENCONTRADO;
   pL->lTamanyo--;

   /* Desplaza todos los elementos hacia adelante a partir de 'lPosicion' */
   for( i = lPosicion; i < pL->lTamanyo; i++ )
       pL->aLista[i] = pL->aLista[i+1];

   if ( ((pL->aLista = (long *) realloc( pL->aLista, sizeof(long)*pL->lTamanyo )) == NULL ) &&
                  (pL->lTamanyo != 0) )
                return ERROR_ASIGNACION_MEMORIA;
   return OK;
};


ESTADO SustituyeListaLong(TListaLong *pL, long lPosicion, long lDato)
/* SUSTITUYE EL ELEMENTO EN 'lPosicion' POR EL ENTERO 'lDato'.
```

```
   DEVUELVE 'OK' SI TODO ES CORRECTO. */
{
   if ( (lPosicion<0) || (lPosicion >= pL->lTamanyo) )
                return ERROR_ELEMENTO_NO_ENCONTRADO;
   pL->aLista[lPosicion] = lDato;
   return OK;
};


ESTADO CopiaListaLong( TListaLong *pOrigen, TListaLong *pDestino )
/* HACE UNA COPIA EXACTA DE 'pDestino' EN 'pOrigen'.
   DEVUELVE 'OK' SI TODO ES CORRECTO.               */
{
         long i;

         pDestino->lTamanyo = pOrigen->lTamanyo;
         if ((pDestino->aLista = (long *) calloc( pDestino->lTamanyo, sizeof(long) ))== NULL)
                  return ERROR_ASIGNACION_MEMORIA;
         for( i=0 ; i < pDestino->lTamanyo ; i++ )
                  pDestino->aLista[i] = pOrigen->aLista[i];
         return OK;
};


BOOLEAN BuscaListaLong( TListaLong *pL, long lDato, long *pPosicion)
/* BUSCA EN LA LISTA 'pL' EL ELEMENTO 'lDato', Y DEVUELVE EN 'pPosicion' SU POSICION.
   DEVUELVE 'OK' SI SE ENCUENTRA EL ELEMENTO, Y 'FALSE' SI NO SE ENCUENTRA.          */
{
         long i;
         BOOLEAN bEncontrado;

         bEncontrado=FALSE;
         for( i=0 ; i < pL->lTamanyo ; i++ ) {
                  if (pL->aLista[i] == lDato) {
                           bEncontrado = TRUE;
                           break;
                  };
         };
         if (bEncontrado == TRUE)
                  *pPosicion = i;
         else
                  *pPosicion = -1;   /* Da un valor imposible, para provocar errores */
         return bEncontrado;
};
```

6.3.1.3 Funciones de la Base de Datos de Entidades

En la figura siguiente se muestra una tabla con las funciones básicas disponibles para los distintos tipos de listas. Además puede consultarse sobre los archivos cabecera (.h) porque sobre la marcha se han ido incorporando -y se incorporarán- más funciones que han sido necesarias.

FUNCIONES DISPONIBLES	Long	Vertices	Aristas	Capas	Caras	CadenasTexto	Circunferencias
NuevaLista... (**Lista*)	X	X	X	X	X	X	X
DestruyeLista... (**Lista*)	X	X	X	X	X	X	X
TamanyoLista... (**Lista*)	X	X	X	X	X	X	X
AnyadeLista... (**Lista*, **Elemento*)	X	X	X		X	X	X
AnyadeLista...SR (**Lista*, **Elemento*)		X		X			
ObtenLista... (**Lista*, *Posicion*, **Elemento*)	X	X	X	X	X	X	X
InsertaLista... (**Lista*, *Posicion*, **Elemento*)	X						
EliminaLista... (**Lista*, *Posicion*)	X	X	X	X	X		X
SustituyeLista... (**Lista*, *Posicion*, **Elemento*)	X	X	X		X	X	X
CopiaLista... (**ListaOrigen*, **ListaOrigen*)	X	X	X	X	X	X	X

Figura 6-3. Tabla-resumen de funciones para manejar las listas de una una BB.DD de Entidades

A continuación veremos el listado de las funciones disponibles para manejar la Base de Datos de Entidades, con sus parámetros, y los valores que devuelven. Sólo deberían utilizarse estas funciones, y evitar en la medida de lo posible acceder a los datos directamente.

Código Fuente 6-4. Funciones de la Base de Datos de Entidades

BDatos.h

```
/*-------------------------------------------------*/
/* FUNCIONES DE MANEJO DE LA ESTRUCTURA 'TVertice' */

void NuevoVertice( TVertice *pV );
/* INICIALIZA LOS DATOS DEL VERTICE, CONCRETAMENTE SU LISTA DE ARISTAS.
   ¡¡IMPORTANTE!! :ES LA PRIMERA OPERACION A REALIZAR. POSTERIORMENTE NO SE DEBE
   UTILIZAR. */

void DestruyeVertice( TVertice *pV );
/* Libera la memoria ocupada por la estructura de un Vertice.
   MUY IMPORTANTE: NO UTILIZAR CON VERTICES NO INICIALIZADOS. Hay que utilizar siempre
   esta funcion con cada vertice utilizado, antes de finalizar los algoritmos. Ojo con
   los bucles que hacen 'ObtenListaVertices' repetidamente sobre el mismo vertice.     */

ESTADO CopiaVertice( TVertice *pOrigen, TVertice *pDestino );
/* HACE UNA COPIA EXACTA DE 'pDestino' EN 'pOrigen'.
   DEVUELVE 'OK' SI TODO ES CORRECTO.               */

BOOLEAN ComparaVertices( TVertice *pV1, TVertice *pV2);
/* Devuelve TRUE si los dos vertices son iguales; FALSE en caso contrario */


/*-----------------------------------------------------------------------*/
/* PROCEDIMIENTOS ESPECIFICOS DE LAS LISTAS DE VERTICES (TListaVertices). */
```

```
void NuevaListaVertices(TListaVertices *pL);
/* INICIALIZA LA LISTA.
   ¡¡IMPORTANTE!! :ES LA PRIMERA OPERACION A REALIZAR. POSTERIORMENTE NO SE DEBE
   UTILIZAR. */

void DestruyeListaVertices(TListaVertices *pL);
/* ELIMINA TODOS LOS ELEMENTOS DE LA LISTA, LIBERANDO TODA SU MEMORIA OCUPADA. */

long TamanyoListaVertices(TListaVertices *pL);
/* DEVUELVE EL TAMAÑO (NUMERO DE ELEMENTOS) DE LA LISTA */

ESTADO AnyadeListaVertices(TListaVertices *pL, TVertice *pVertice);
/* OBTIENE MAS MEMORIA DINAMICAMENTE, Y AÑADE EL VERTICE 'pVertice' AL FINAL DE LA
   LISTA.
   DEVUELVE 'OK' SI TODO ES CORRECTO. */

ESTADO ObtenListaVertices(TListaVertices *pL, long lPosicion, TVertice *pVertice);
/* DEVUELVE EN 'pVertice' EL ELEMENTO DE LA POSICION 'lPosicion'.
   DEVUELVE 'OK' SI TODO ES CORRECTO.
   MUY IMPORTANTE: Despues de obtener un vertice, cuando este no haga falta hay que
   Destruirlo antes de salir del proceso que lo utiliza. Ver la instruccion
   "DestruyeVertice".            */

ESTADO EliminaListaVertices(TListaVertices *pL, long lPosicion);
/* ELIMINA EL ELEMENTO EN LA POSICION 'lPosicion', LIBERANDO MEMORIA.
   DEVUELVE 'OK' SI TODO ES CORRECTO. */

long AnyadeListaVerticesSR( TListaVertices *pL, TVertice *pV );
/* Si no existe uno igual, añade el vertice 'pV' y devuelve su posicion en la lista.
   Si existe uno igual devuelve la posicion del que ya estaba.
   En caso de error (por no poder asignar memoria) devuelve -1. */

ESTADO SustituyeListaVertices(TListaVertices *pL, long lPosicion, TVertice *pVertice);
/* SUSTITUYE EL ELEMENTO EN 'lPosicion' DE LA LISTA 'pL' POR EL VERTICE 'pVertice'.
   DEVUELVE 'OK' SI TODO ES CORRECTO. */

void XYZMaxMinListaVertices(TListaVertices *pL, double *Xmin, double *Ymin, double *Zmin, double *Xmax,
double *Ymax, double *Zmax);
/* Devuelve las X's, Y's, y Z's mas grandes y mas pequeñas de todos los vertices de la
   lista */

ESTADO CopiaListaVertices( TListaVertices *pOrigen, TListaVertices *pDestino );
/* HACE UNA COPIA EXACTA DE 'pDestino' EN 'pOrigen'.
   DEVUELVE 'OK' SI TODO ES CORRECTO.              */


/*-------------------------------------------------*/
/* FUNCIONES DE MANEJO DE LA ESTRUCTURA 'TCapa' */

void CopiaCapa( TCapa *pOrigen, TCapa *pDestino );
/* HACE UNA COPIA EXACTA DE 'pDestino' EN 'pOrigen'.
   DEVUELVE 'OK' SI TODO ES CORRECTO.              */


/*-----------------------------------------------------------------*/
/* PROCEDIMIENTOS ESPECIFICOS DE LAS LISTAS DE CAPAS (TListaCapas). */

void NuevaListaCapas(TListaCapas *pL);
/* INICIALIZA LA LISTA.
   ¡¡IMPORTANTE!! :ES LA PRIMERA OPERACION A REALIZAR. POSTERIORMENTE NO SE DEBE
   UTILIZAR. */

void DestruyeListaCapas(TListaCapas *pL );
/* Destruye la lista, destruyendo cada Capa con su cadena de caracteres asociada */

long TamanyoListaCapas(TListaCapas *pL);
```

```
/* DEVUELVE EL TAMAÑO (NUMERO DE ELEMENTOS) DE LA LISTA */

ESTADO ObtenListaCapas(TListaCapas *pL, long lPosicion, TCapa *pCapa);
/* DEVUELVE EN 'pCapa' EL ELEMENTO DE LA POSICION 'lPosicion'.
   DEVUELVE 'OK' SI TODO ES CORRECTO. */

long AnyadeListaCapasSR(TListaCapas *pL, char *sLinea);
/* Si no existe una igual, crea una nueva capa con el título 'sLinea', la añade en la
   lista, y devuelve su posicion.  Si existe una igual, devuelve la posicion de la que
   ya estaba.
   En caso de error (por no poder asignar memoria) devuelve -1. */

ESTADO EliminaListaCapas(TListaCapas *pL, long lPosicion);
/* ELIMINA EL ELEMENTO EN LA POSICION 'lPosicion', LIBERANDO MEMORIA.
   DEVUELVE 'OK' SI TODO ES CORRECTO. */

ESTADO CopiaListaCapas(TListaCapas *pOrigen, TListaCapas *pDestino );
/* HACE UNA COPIA EXACTA DE 'pDestino' EN 'pOrigen'.
   DEVUELVE 'OK' SI TODO ES CORRECTO.              */

ESTADO ActivaCapaListaCapas(TListaCapas *pL, long lPosicion, BOOLEAN bActiva);
/* SUSTITUYE EL VALOR DE LA VARIABLE 'Capa.bActiva' DE LA CAPA EN LA POSICIÓN
   'lPosicion' DE LA LISTA POR LA VARIABLE 'bActiva'.
   DEVUELVE 'OK' SI TODO ES CORRECTO. */


/*-------------------------------------------------------------------------*/
/* PROCEDIMIENTOS ESPECIFICOS DE LAS LISTAS DE ARISTAS (TListaAristas). */

void NuevaListaAristas(TListaAristas *pL);
/* INICIALIZA LA LISTA.
   ¡¡IMPORTANTE!! :ES LA PRIMERA OPERACION A REALIZAR. POSTERIORMENTE NO SE DEBE
   UTILIZAR. */

void DestruyeListaAristas(TListaAristas *pL);
/* ELIMINA TODOS LOS ELEMENTOS DE LA LISTA, LIBERANDO TODA SU MEMORIA OCUPADA. */

long TamanyoListaAristas(TListaAristas *pL);
/* DEVUELVE EL TAMAÑO (NUMERO DE ELEMENTOS) DE LA LISTA */

ESTADO AnyadeListaAristas(TListaAristas *pL, TArista *pArista);
/* OBTIENE MAS MEMORIA DINAMICAMENTE, Y AÑADE LA ARISTA 'pArista' AL FINAL DE LA LISTA.
   DEVUELVE 'OK' SI TODO ES CORRECTO. */

ESTADO ObtenListaAristas(TListaAristas *pL, long lPosicion, TArista *pArista);
/* DEVUELVE EN 'pArista' EL ELEMENTO DE LA POSICION 'lPosicion'.
   DEVUELVE 'OK' SI TODO ES CORRECTO. */

ESTADO EliminaListaAristas(TListaAristas *pL, long lPosicion);
/* ELIMINA EL ELEMENTO EN LA POSICION 'lPosicion', LIBERANDO MEMORIA.
   DEVUELVE 'OK' SI TODO ES CORRECTO. */

ESTADO SustituyeListaAristas(TListaAristas *pL, long lPosicion, TArista *pArista);
/* SUSTITUYE EL ELEMENTO EN 'lPosicion' DE LA LISTA 'pL' POR LA ARISTA 'pArista'.
   DEVUELVE 'OK' SI TODO ES CORRECTO. */

ESTADO CopiaListaAristas(TListaAristas *pOrigen, TListaAristas *pDestino );
/* HACE UNA COPIA EXACTA DE 'pDestino' EN 'pOrigen'.
   DEVUELVE 'OK' SI TODO ES CORRECTO.              */


/*-------------------------------------------------*/
/* FUNCIONES DE MANEJO DE LA ESTRUCTURA 'TCara'    */

void NuevaCara( TCara *pC );
/* INICIALIZA LOS DATOS DE LA CARA, CONCRETAMENTE SU LISTA DE ARISTAS.
   ¡¡IMPORTANTE!! :ES LA PRIMERA OPERACION A REALIZAR. POSTERIORMENTE NO SE DEBE
```

```
   UTILIZAR. */

void DestruyeCara( TCara *pC );
/* Libera la memoria ocupada por la estructura de una Cara.
   MUY IMPORTANTE: NO UTILIZAR CON CARAS NO INICIALIZADAS. Hay que utilizar siempre
   esta funcion con cada cara utilizada, antes de finalizar los algoritmos. Ojo con
   los bucles que hacen 'ObtenListaCaras' repetidamente sobre la misma cara.       */

ESTADO CopiaCara( TCara *pOrigen, TCara *pDestino );
/* HACE UNA COPIA EXACTA DE 'pDestino' EN 'pOrigen'.
   DEVUELVE 'OK' SI TODO ES CORRECTO.              */


/*-------------------------------------------------------------------*/
/* PROCEDIMIENTOS ESPECIFICOS DE LAS LISTAS DE CARAS (TListaCaras).   */

void NuevaListaCaras(TListaCaras *pL);
/* INICIALIZA LA LISTA.
   ¡¡IMPORTANTE!! :ES LA PRIMERA OPERACION A REALIZAR. POSTERIORMENTE NO SE DEBE
   UTILIZAR. */

void DestruyeListaCaras(TListaCaras *pL);
/* ELIMINA TODOS LOS ELEMENTOS DE LA LISTA, LIBERANDO TODA SU MEMORIA OCUPADA. */

long TamanyoListaCaras(TListaCaras *pL);
/* DEVUELVE EL TAMAÑO (NUMERO DE ELEMENTOS) DE LA LISTA */

ESTADO AnyadeListaCaras(TListaCaras *pL, TCara *pCara);
/* OBTIENE MAS MEMORIA DINAMICAMENTE, Y AÑADE LA CARA 'pCara' AL FINAL DE LA LISTA.
   DEVUELVE 'OK' SI TODO ES CORRECTO. */

ESTADO ObtenListaCaras(TListaCaras *pL, long lPosicion, TCara *pCara);
/* DEVUELVE EN 'pCara' EL ELEMENTO DE LA POSICION 'lPosicion'.
   DEVUELVE 'OK' SI TODO ES CORRECTO. */

ESTADO EliminaListaCaras(TListaCaras *pL, long lPosicion);
/* ELIMINA EL ELEMENTO EN LA POSICION 'lPosicion', LIBERANDO MEMORIA.
   DEVUELVE 'OK' SI TODO ES CORRECTO. */

ESTADO SustituyeListaCaras(TListaCaras *pL, long lPosicion, TCara *pCara);
/* SUSTITUYE EL ELEMENTO EN 'lPosicion' DE LA LISTA 'pL' POR LA CARA 'pCara'.
   DEVUELVE 'OK' SI TODO ES CORRECTO. */

ESTADO CopiaListaCaras(TListaCaras *pOrigen, TListaCaras *pDestino );
/* HACE UNA COPIA EXACTA DE 'pDestino' EN 'pOrigen'.
   DEVUELVE 'OK' SI TODO ES CORRECTO.              */


/*-----------------------------------------------------*/
/* FUNCIONES DE MANEJO DE LA ESTRUCTURA 'TCadenaTexto' */

void CopiaCadenaTexto( TCadenaTexto *pOrigen, TCadenaTexto *pDestino );
/* HACE UNA COPIA EXACTA DE 'pDestino' EN 'pOrigen'.
   DEVUELVE 'OK' SI TODO ES CORRECTO.              */


/*-------------------------------------------------------------------------------*/
/* PROCEDIMIENTOS ESPECIFICOS DE LAS LISTAS DE CADENAS DE TEXTO (TListaCadenasTexto). */

void NuevaListaCadenasTexto(TListaCadenasTexto *pL);
/* INICIALIZA LA LISTA.
   ¡¡IMPORTANTE!! :ES LA PRIMERA OPERACION A REALIZAR. POSTERIORMENTE NO SE DEBE
   UTILIZAR. */

void DestruyeListaCadenasTexto(TListaCadenasTexto *pL);
/* ELIMINA TODOS LOS ELEMENTOS DE LA LISTA, LIBERANDO TODA SU MEMORIA OCUPADA. */
```

```
long TamanyoListaCadenasTexto(TListaCadenasTexto *pL);
/* DEVUELVE EL TAMAÑO (NUMERO DE ELEMENTOS) DE LA LISTA */

ESTADO AnyadeListaCadenasTexto(TListaCadenasTexto *pL, TCadenaTexto *pCadenaTexto);
/* OBTIENE MAS MEMORIA DINAMICAMENTE, Y AÑADE LA CADENA 'pCadenaTexto' AL FINAL DE LA
   LISTA.
   DEVUELVE 'OK' SI TODO ES CORRECTO. */

ESTADO ObtenListaCadenasTexto(TListaCadenasTexto *pL, long lPosicion, TCadenaTexto *pCadenaTexto);
/* DEVUELVE EN 'pCadenaTexto' EL ELEMENTO DE LA POSICION 'lPosicion'.
   DEVUELVE 'OK' SI TODO ES CORRECTO. */

ESTADO SustituyeListaCadenasTexto(TListaCadenasTexto *pL, long lPosicion, TCadenaTexto *pCadenaTexto);
/* SUSTITUYE EL ELEMENTO EN 'lPosicion' DE LA LISTA 'pL' POR LA CADENA 'pCadenaTexto'.
   DEVUELVE 'OK' SI TODO ES CORRECTO. */

ESTADO CopiaListaCadenasTexto(TListaCadenasTexto *pOrigen, TListaCadenasTexto *pDestino );
/* HACE UNA COPIA EXACTA DE 'pDestino' EN 'pOrigen'.
   DEVUELVE 'OK' SI TODO ES CORRECTO.               */


/*-------------------------------------------------------------------------------*/
/* PROCEDIMIENTOS ESPECIFICOS DE LAS LISTAS DE CIRCUNFERENCIAS (TListaCircunferencias)*/

void NuevaListaCircunferencias(TListaCircunferencias *pL);
/* INICIALIZA LA LISTA.
   ¡¡IMPORTANTE!! :ES LA PRIMERA OPERACION A REALIZAR. POSTERIORMENTE NO SE DEBE
   UTILIZAR. */

void DestruyeListaCircunferencias(TListaCircunferencias *pL);
/* ELIMINA TODOS LOS ELEMENTOS DE LA LISTA, LIBERANDO TODA SU MEMORIA OCUPADA. */

long TamanyoListaCircunferencias(TListaCircunferencias *pL);
/* DEVUELVE EL TAMAÑO (NUMERO DE ELEMENTOS) DE LA LISTA */

ESTADO AnyadeListaCircunferencias(TListaCircunferencias *pL, TCircunferencia *pCircunferencia);
/* OBTIENE MAS MEMORIA DINAMICAMENTE, Y AÑADE LA CIRCUNFERENCIA 'pCircunferencia' AL
   FINAL DE LA LISTA.
   DEVUELVE 'OK' SI TODO ES CORRECTO. */

ESTADO ObtenListaCircunferencias(TListaCircunferencias *pL, long lPosicion, TCircunferencia *pCirc);
/* DEVUELVE EN 'pCircunferencia' EL ELEMENTO DE LA POSICION 'lPosicion'.
   DEVUELVE 'OK' SI TODO ES CORRECTO. */

ESTADO EliminaListaCircunferencias(TListaCircunferencias *pL, long lPosicion);
/* ELIMINA EL ELEMENTO EN LA POSICION 'lPosicion', LIBERANDO MEMORIA.
   DEVUELVE 'OK' SI TODO ES CORRECTO. */

ESTADO SustituyeListaCircunferencias(TListaCircunferencias *pL, long lPosicion, TCircunferencia *pCirc);
/* SUSTITUYE EL ELEMENTO EN 'lPosicion' POR LA CIRCUNFERENCIA 'pCirc'.
   DEVUELVE 'OK' SI TODO ES CORRECTO. */

ESTADO CopiaListaCircunferencias( TListaCircunferencias *pOrigen, TListaCircunferencias *pDestino );
/* HACE UNA COPIA EXACTA DE 'pDestino' EN 'pOrigen'.
   DEVUELVE 'OK' SI TODO ES CORRECTO. */


/*-------------------------------------------------------------------------------*/
/* PROCEDIMIENTOS ESPECIFICOS DE LAS BASES DE DATOS DE ENTIDADES (TBDEntidades). */

ESTADO NuevaBDEntidades( TBDEntidades *pBD );
/* Crea una nueva Base de Datos, inicializando sus listas y componentes.
   Devuelve 'OK' si la operacion se realiza con exito.                  */

ESTADO DestruyeBDEntidades( TBDEntidades *pBD );
/* Destruye la Base de Datos, destruyendo sus listas y componentes.
   Devuelve 'OK' si la operacion se realiza con exito.             */
```

```
ESTADO AnyadeEntidadBDEntidades( TBDEntidades *pBD, TEntidad *pE );
/* Inserta la Entidad 'pE' en la Base de Datos 'pBD', manteniendo la integridad de la
   BB.DD.
   Devuelve 'OK' si la operacion se realiza con exito.                                    */


/* ============================================================================
   MUY IMPORTANTE:
   El orden de dependencia de las Listas de la Base de Datos es el siguiente:

            ListaCaras --> ListaAristas <==> ListaVertices
                                |
                                +---------->  \
                                              |
             ListaCadenasTexto ------------>  +--> ListaCapas
                                              |
             ListaCircunferencias --------->  /

   Por tanto, para MANTENER LA INTEGRIDAD de la Base de Datos tendre que cumplir:

* Si quiero eliminar un Vertice, primero tendre que modificar ListaCaras, despues
modificare ListaAristas, y por ultimo eliminare el Vertice de ListaVertices.

* Si quiero eliminar una Arista, primero tendre que modificar ListaCaras y luego podre
eliminar la Arista de ListaAristas.
Automaticamente se modificaran las ListaAristas de cada Vertice de ListaVertices.

* Si quiero eliminar una Capa, primero tendre que modificar ListaAristas, ListaCadenasTexto y
ListaCircunferencias, y por ultimo eliminare la Capa de ListaCapas.

* Puedo modificar ListaCaras o ListaCadenasTexto o ListaCircunferencias sin modificar
ninguna otra Lista.

============================================================================ */


ESTADO EliminaAristaBDEntidades( TBDEntidades *pBD, long lPosicion );
/* Elimina la Arista de la posicion 'lPosicion' de la Lista de Aristas de la BB.DD.
   MUY IMPORTANTE: ESTA FUNCION MANTENIENE LA INTEGRIDAD DE LA BASE DE DATOS, REVISANDO
   EL RESTO DE LISTAS Y ACTUALIZANDOLAS CONVENIENTEMENTE.

* La Arista eliminada no deberia pertenecer a ninguna Cara.
* Al eliminar la Arista, se elimina tambien de la ListaAristas de los dos vertices
         ( Cabeza y Cola ). Si estas listas quedan a cero, se eliminan los vertices.
   Devuelve 'OK' si la operacion se realiza con exito.                                 */

ESTADO EliminaVerticeBDEntidades( TBDEntidades *pBD, long lPosicion );
/* Elimina el Vertice de la posicion 'lPosicion' de la Lista de Vertices de la BB.DD.
   MUY IMPORTANTE: ESTA FUNCION MANTENIENE LA INTEGRIDAD DE LA BASE DE DATOS, REVISANDO
   EL RESTO DE LISTAS Y ACTUALIZANDOLAS CONVENIENTEMENTE.
   * El Vertice eliminado no deberia pertenecer a ninguna Arista.
   Devuelve 'OK' si la operacion se realiza con exito.                                 */

ESTADO EliminaCapaBDEntidades( TBDEntidades *pBD, long lPosicion );
/* Elimina la Capa de la posicion 'lPosicion' de la Lista de Capas de la BB.DD.
   MUY IMPORTANTE: ESTA FUNCION MANTENIENE LA INTEGRIDAD DE LA BASE DE DATOS, REVISANDO
   EL RESTO DE LISTAS Y ACTUALIZANDOLAS CONVENIENTEMENTE.
   * No deberia existir ninguna Arista, Circunferencia, CadenaTexto que pertenezca a la
     Capa que se quiere eliminar.
   Devuelve 'OK' si la operacion se realiza con exito.                                 */

ESTADO EliminaSeleccionadosBDEntidades( TBDEntidades *pBD, int iEntidad);
/* Elimina de la Base de Datos 'pBD' las entidades de tipo 'iEntidad' que tengan
   el campo 'bSeleccion' a TRUE, manteniendo la integridad de la BB.DD.
   MUY IMPORTANTE: ESTA FUNCION MANTENIENE LA INTEGRIDAD DE LA BASE DE DATOS, REVISANDO
   EL RESTO DE LISTAS Y ACTUALIZANDOLAS CONVENIENTEMENTE.
```

```
   * Si se elimina un Vertice, tambien se eliminan las Aristas que lo contengan.
   * Si se elimina una Arista, tambien se eliminan los Vertices de esta.
   Devuelve 'OK' si la operacion se realiza con exito.                              */

ESTADO EliminaNoVisiblesBDEntidades( TBDEntidades *pBD, TOpcionesVisibilidad *pOpciones );
/* Elimina de la Base de Datos 'pBD' las entidades no visibles, manteniendo la
   Integridad de la BB.DD.
   Devuelve 'OK' si la operacion se realiza con exito.                                */

ESTADO CopiaBDEntidades( TBDEntidades *pBDOrigen, TBDEntidades *pBDDestino );
/* Hace una copia exacta de la Base de Datos 'pBDOrigen' en 'pBDDestino'.
   Devuelve 'OK' si la operacion se realiza con exito.  */
```

6.3.2 Leer imagen 2D. Archivos DXF

REFER tendrá que iniciar la reconstrucción geométrica a partir de un dibujo en 2D. Lo más cómodo y práctico es que este dibujo se pueda obtener a partir de un archivo realizado en algún programa gráfico de los conocidos, como AutoCAD, MicroStation, etc. Esto nos da la flexibilidad de no estar atados a un formato propio. Por tanto tiene que ser capaz de leer un fichero gráfico estándar, como puede ser DWG, DXF, IGES, etc.

El programa de diseño para PC más popular en el mercado es el AutoCAD de Autodesk. Cualquier otro programa puede utilizar los mismos ficheros que AutoCAD, que son el DWG y el DXF. Ahora bien ¿cuál de los dos utilizar?

Así pues busqué información de Autodesk acerca de ambos formatos. Encontré la descripción de la versión 12, en Internet [19]. Lo que sigue es la traducción literal de un párrafo que me despejó cualquier duda:

"(...)Como la base de datos de un dibujo de AutoCAD (un fichero .dwg) está escrita en un formato compacto que cambia significativamente cuando se añaden nuevas funciones a AutoCAD, no documentamos este formato y no recomendamos que se hagan programas para leerlo directamente. Para permitir el intercambio de dibujos entre AutoCAD y otros programas, se ha definido el formato DXF (Drawing Interchange File). Todas las versiones de AutoCAD aceptan este formato y son capaces de convertir hacia y desde su representación interna del dibujo.(...)"

Así que seguí el consejo de Autodesk y elegí el formato DXF; además tiene la ventaja de ser un fichero ASCII estándar, que puede ser abierto y estudiado con cualquier sencillo editor de texto.

Ahora bien, una vez ya tenía un archivo por el que empezar, tenía que saber cómo guardarme internamente la información que iba a leer. Información relativa a los vértices y aristas que componen las figuras.

El programa que lee el fichero DXF y nos genera la base de datos de entidades de dibujo que nos interesan lo he llamado **TurboInterpreteDXF**, en honor al compilador *Turbo Pascal* de Borland.

6.3.2.1 Estructura general de un fichero DXF

Básicamente un fichero DXF tiene varias secciones:

1. HEADER: Contiene información general sobre el dibujo.
2. TABLES: Contiene definiciones de varios items:
    - 2.1. LTYPE: Tabla de tipos de linea.
    - 2.2. LAYER: Tabla de capas.
    - 2.3. STYLE: Tabla de estilos de texto.
    - 2.4. VIEW: Tabla de vistas.
    - 2.5. UCS: Tabla del sistema de coordenadas del usuario.
    - 2.6. VPORT: Tabla de configuración de ventana.
    - 2.7. DIMSTYLE: Tabla de estilos de dimensiones.
    - 2.8. APPID: Tabla de identificación de la aplicación.
3. BLOCKS: Contiene entidades de definición de bloques.
4. ENTITIES: Contiene las entidades de dibujo, incluyendo cualquier bloque.
5. END OF FILE.

6.3.2.2 Funcionamiento básico del lector DXF

El propósito fundamental de nuestro lector de DXF es ser capaz de obtener los vértices y las aristas. Esta información se encuentra en la sección ENTITIES. Así que lo primero que hacemos es buscar esta sección en el fichero DXF.

A partir de aquí son varias las entidades de AutoCAD que nos podemos encontrar. A nosotros nos interesan las siguientes:

- LINE: Define una linea recta (o arista) que une dos vértices
- CIRCLE: Define una circunferencia con un centro y un radio.
- TEXT: Define una linea de texto, de hasta 256 caracteres, y la sitúa en un punto en el espacio. Puede tener información del tipo de letra.

POLYLINE: Define un conjunto de aristas, definidas por un conjunto de vértices consecutivos.

Todas estas entidades tienen campos para definir con qué color, qué estilo de linea, y en qué capa del dibujo se deben representar.

Cuando tenemos completamente definida una entidad de AutoCAD, la insertamos en nuestra Base de Datos de Entidades con:

```
AnyadeEntidadBDEntidades( pBDEntidades, &Entidad );
```

Esta función se encarga de ampliar dinámicamente las listas de entidades según sea necesario, insertar la nueva entidad, y mantener la coherencia entre la distintas listas, sin que nos tengamos que preocupar en este nivel de cómo se almacenan internamente estos datos.

6.3.3 Visualizar imagen 2D

En este punto ya tenemos un dibujo 2D almacenado de modo conveniente en nuestra Base de Datos de Entidades. Para mostrarlo en pantalla utilizaremos las clases de dibujo básicas que nos ofrecen las Microsoft Foundation Classes del Microsoft Visual C++ que nos proporcionan todos los recursos de manejo de ventanas, pinceles,

colores, tipos de linea, etc. que podamos necesitar. Además esto nos da luego facilidades extra como puede ser imprimir directamente sobre cualquier impresora que tengamos definida en Windows, sin apenas esfuerzo.

En cualquier representación de figuras 2D o 3D, tendremos que pasar de una **Ventana del Mundo Real** (en donde se "mueven" dichas figuras) a una **Ventana de un Dispositivo**, como puede ser una ventana de Windows, o podría ser una hoja A4 de una impresora, etc. Cada ventana puede tener su propio sistema de coordenadas. Hay que hacer una **transformación al Dispositivo** para pasar correctamente de una ventana a otra.

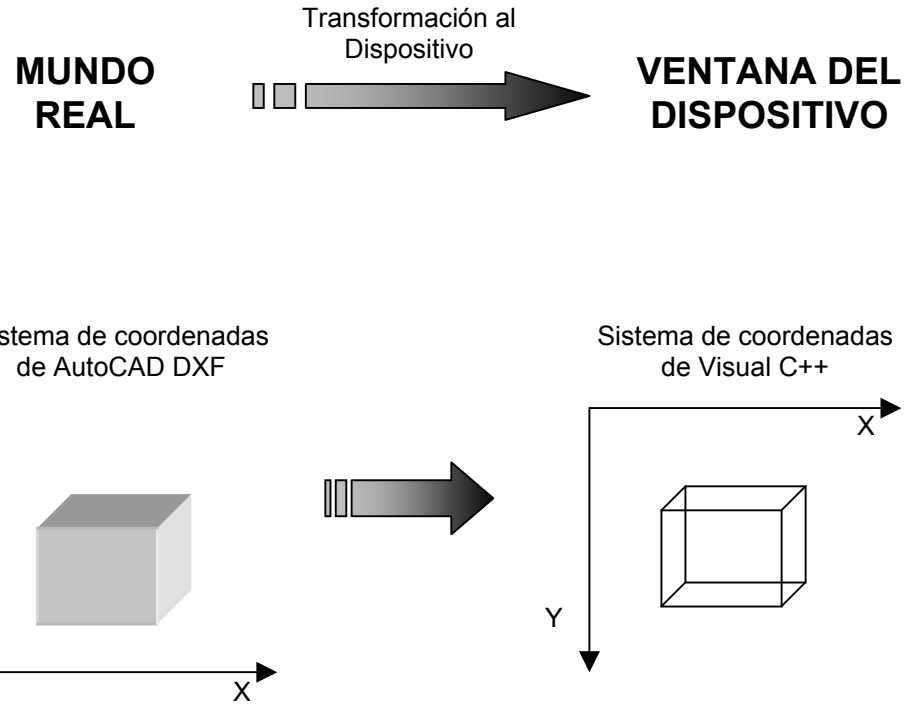


Figura 6-4. Transformación al Dispositivo

Para esto creamos y utilizamos la clase *CTransformacionVentanaDispositivo*. Así nos definiremos un objeto de esta clase, y lo inicializaremos estableciendo cómo es la Ventana del mundo Real y la Ventana ó Marco del Dispositivo. Posteriormente, realizando llamadas a su método *TransformacionDispositivo(x,y)* pasándole un par de coordenadas *(x,y)* del Mundo Real, nos devuelve un punto de pantalla (dispositivo) que podemos pintar directamente con los métodos de dibujo de las MFC.

Estas son las funciones de la clase que disponemos para hacer transformaciones de la ventana al dispositivo:

Código Fuente 6-5. Clase y funciones para la Transformación al Dispositivo

MISCELANEA.CPP

```
/*-------------------------------------------------------------------------*/
// Define la Clase para hacer Transformaciones de una Ventana del mundo real
// a un Dispositivo como pueda ser una ventana Windows.
class CTransformacionVentanaDispositivo {

public:
        struct {
                double dXmax, dXmin, dYmax, dYmin;
        } m_VentanaDispositivo;

        struct {
                double dXmax, dXmin, dYmax, dYmin;
        } m_VentanaReal;

        void VentanaReal(double Xmin, double Ymin, double Xmax, double Ymax);
        // Establece la VENTANA del MUNDO REAL

        void MarcoDispositivo(double Xmin, double Ymin, double Xmax, double Ymax);
        // Establece el MARCO del DISPOSITIVO (ventana)

        TPunto TransformacionDispositivo(double x, double y);
        //Transformacion de coordenadas REALES a coordenadas del DISPOSITIVO
};
/*-------------------------------------------------------------------------*/
```

6.3.4 Visualizar modelo 3D. Librería gráfica OpenGL

Antes de comenzar a añadir cualquier código OpenGL a esta aplicación, hay que añadir las librerías OpenGL al proyecto (*Projetc Settings* de Visual C++). Esto supone entrar en *Build / Settings / Link / Object-library modules* y agregar las librerías **opengl32.lib**, **glu32.lib**.

También hay que añadir los archivos de cabecera de OpenGL al proyecto. El lugar más fácil en donde ponerlos (y así olvidarnos de ellos), es en *stdafx.h* y así quedarán incluidos en la cabecera precompilada:

```
// stdafx.h

#include <gl\gl.h>          // Librerias OpenGL
#include <gl\glu.h>         // Librerias OpenGL GLU
```

Para cualquier aplicación es una buena práctica de diseño mantener código fuente tan modular como sea posible. Al aislar trozos funcionales, hace mucho más fácil reutilizar y mantener el código. Al aislar el código OpenGL "puro" en un módulo separado, se puede reemplazar eficientemente con un código específico, conservando la funcionalidad del resto de la aplicación.

Empiezo por definir tres funciones en un fichero en fuente C llamado *GLCodigo.c*. El fichero *GLCodigo.h* contiene la definición de estas funciones y está incluido para el acceso en nuestro fichero de clase derivado de *CView*.

Código Fuente 6-6. Funciones disponibles en GLCodigo.h

```
// GLCodigo.h

void GLSetupRC(void *pData);
void GLRenderScene(TBDEntidades *pBBDD, TCamara *pCamara,
                   Tvolumen *pVolumen, bool bPlanoXY);
void GLResize(GLsizei w, GLsizei h, TVolumen *pVolumen);
```

La función *GLSetupRC* es donde se colocará cualquier código que inicialice nuestro contexto de generación. Esto puede ser tan simple como determinar el color de limpieza, o tan complejo como establecer las condiciones de iluminación. Se llamará a la función *GLRenderScene* por medio de la función miembro *OnDraw* de la clase derivada de *CView* para generar la escena. Por último, se llamará a *GLResize* por medio del gestor *WM_SIZE*, pasando los nuevos alto y ancho del área cliente de la ventana. Aquí se pueden hacer todos los cálculos necesarios para establecer el volumen de visualización y la vista.

Las funciones *GLSetupRC* y *GLRenderScene* toman punteros *void*. Esto me permite pasar datos de cualquier tipo al código de generación sin cambiar la interfaz. Aunque se podría haber hecho el fichero *GLCodigo* un fichero C++ en vez de un fichero C, es más fácil eliminar el código C existente desde cualquier fuente e incluirlo en el programa MFC. Visual C++ compilará este módulo como un fichero C y lo vinculará al resto de la aplicación.

6.3.5 Arquitectura de una aplicación Windows

La librería MFC permite construir aplicaciones SDI (*Single Document Interface*) y aplicaciones MDI (*Multiple Document Interface*).

Una aplicación SDI, sólo permite tener abierta una ventana marco con la vista del documento que tiene abierto, y que también es único por cada ejemplar activo de la aplicación.

Una aplicación MDI permite tener abiertas varias ventanas marco dentro de la ventana principal de la aplicación. Esto es, una aplicación MDI tiene una ventana marco principal dentro de la cual pueden abrirse varias ventanas marco hijas, de las cuales una solo estará activa, la que tiene la barra de título resaltada. Cada ventana hija contiene una vista de un documento, lo que permite disponer de diferentes tipos de ventanas; por ejemplo, ventanas de texto y ventanas de hojas de cálculo. Ninguna de las ventanas hija tiene menú, ya que comparten el menú de la ventana principal (ventana padre).

Los objetos fundamentales de una aplicación en ejecución son los siguientes:

- **Objeto aplicación** (objeto de una clase derivada de *CWinApp*). Controla al resto de los objetos que forman parte de la aplicación y tiene como cometido inicializar, ejecutar y finalizar la aplicación. Sólo hay un objeto aplicación por cada aplicación Windows. También crea y gestiona las plantillas de documento para los distintos tipos de documentos que la aplicación soporte.

- **PlantilIa de documento** (objeto de una clase derivada dc las clases derivadas de *CDocTemplate*). Una plantilla de documento es un mecanismo para integrar documentos, vistas y ventanas. La plantilla de documento conecta un objeto documento con sus vistas asociadas y con las ventanas en las que las vistas visualizan los datos del documento, lo que asegura su utilización conjunta cuando el usuario abre o crea un determinado tipo de documento.

Un clase plantilla de documento concreta, crea y gestiona todos los documentos de un mismo tipo abiertos. Las aplicaciones que soportan más de un tipo de documento tienen varias plantillas de documento. Para crear plantillas de documento se utiliza la clase *CSingleDocTemplate* en aplicaciones SDI y la clase *CMultiDocTemplate* en aplicaciones MDI.

El objeto aplicación mantiene una lista de todas las plantillas de documento de la aplicación, de forma que cada nueva plantilla de documento se añade automáticamente a esta lista. Por consiguiente, el objeto aplicación conoce en todo momento qué tipo de ficheros están asociados con los documentos y con las vistas. La aplicación puede entonces registrar estos tipos de ficheros en el fichero *win.ini,* de esta forma un usuario puede hacer doble clic en un fichero del *Administrador de programas* y llamar a la aplicación para abrirla.

- **Documento** (objeto de una clase derivada de *CDocument*)**.** La clase documento especifica los datos de la aplicación. Proporciona toda la funcionalidad necesaria para la apertura y administración de ficheros en el disco. Por lo tanto, se puede utilizar un objeto de una clase derivada de *CDocument* para soportar órdenes como *Fichero nuevo, Abrir fichero, Guardar jwhero* y *Guardar como.* Para cada tipo de fichero que utilice una aplicación, hay que derivar una clase de *Cdocument*.

  Cuando se crea una aplicación, AppWizard crea una clase derivada de *CDocument***.** Para crear más clases derivadas de *CDocument* hay que utilizar ClassWizard.

- **Ventana marco** (objeto de una clase derivada de *CFrameWnd***,** derivada de *CWnd***)**. Las ventanas marco proporcionan, a las vistas utilizadas para visualizar los datos, los controles estándar de una ventana; esto es, el marco, la barra de título, el menú de control y los botones de maximizar y minimizar la ventana. En una aplicación SDI, la ventana marco de la vista del documento es también la ventana marco principal de la aplicación, en cambio, en una aplicación MDI, es una ventana hija de la ventana marco

principal. En este caso son las ventanas hija las que contienen los objetos vista de los documentos abiertos.

Para aplicaciones SDI, la ventana marco principal debe derivarse de *CFrameWnd*. Si la aplicación es MDI, la ventana marco principal debe derivarse de *CMDIFrameWnd* y las ventanas hijas, ventanas marco de documento, se derivarán de *CMDIChildWnd***.** Por cada tipo de ventana marco de documento que soporte la aplicación se derivará una clase de *CMDChildWnd*.

- **Vista** (objeto de una clase derivada de *CView*). Una vista es la ventana que hace de interfaz entre el usuario y los datos del objeto documento. Por lo tanto, determina cómo se visualizan los datos y cómo puede actuar el usuario sobre ellos. Un documento puede tener varias vistas de los datos.

En una aplicación en ejecución, estos objetos responden de forma conjunta a las acciones del usuario, comunicándose entre sí por medio de mensajes. Un único objeto aplicación gestiona una o más plantillas de documento. Cada plantilla de documento crea y gestiona uno o más documentos (dependiendo de si la aplicación es SDI o MDI). El usuario ve y manipula un documento a través de la vista contenida dentro de una ventana marco. La figura siguiente muestra las relaciones entre los objetos para una aplicación SDI.

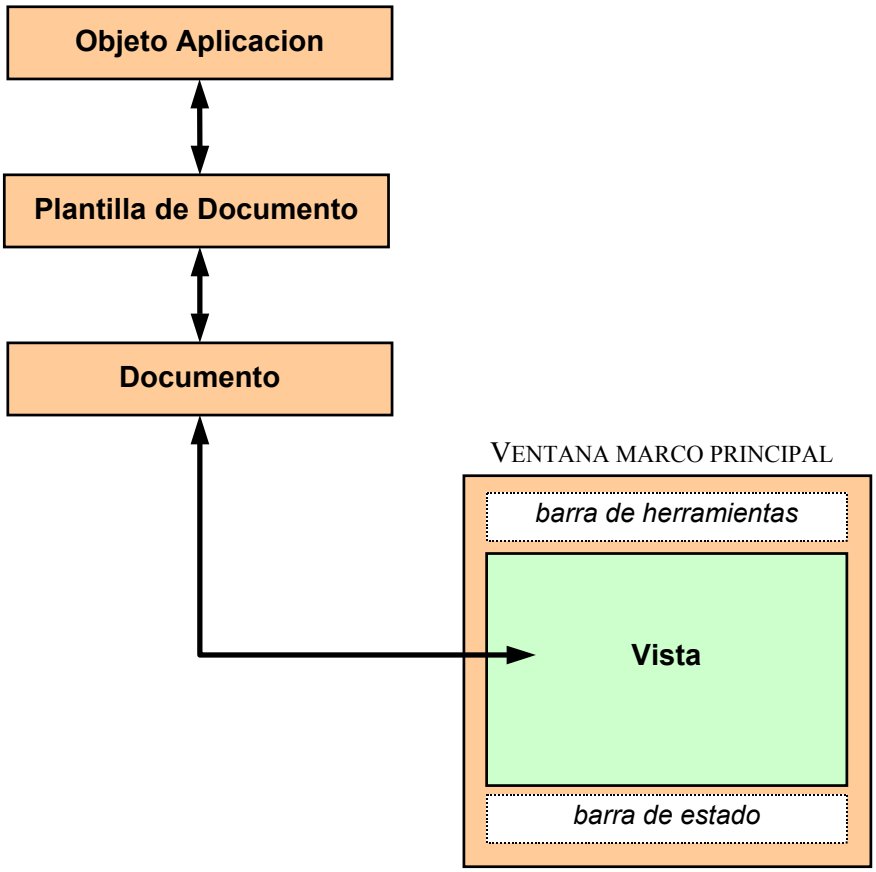


Figura 6-5. Relaciones entre objetos en una aplicación SDI

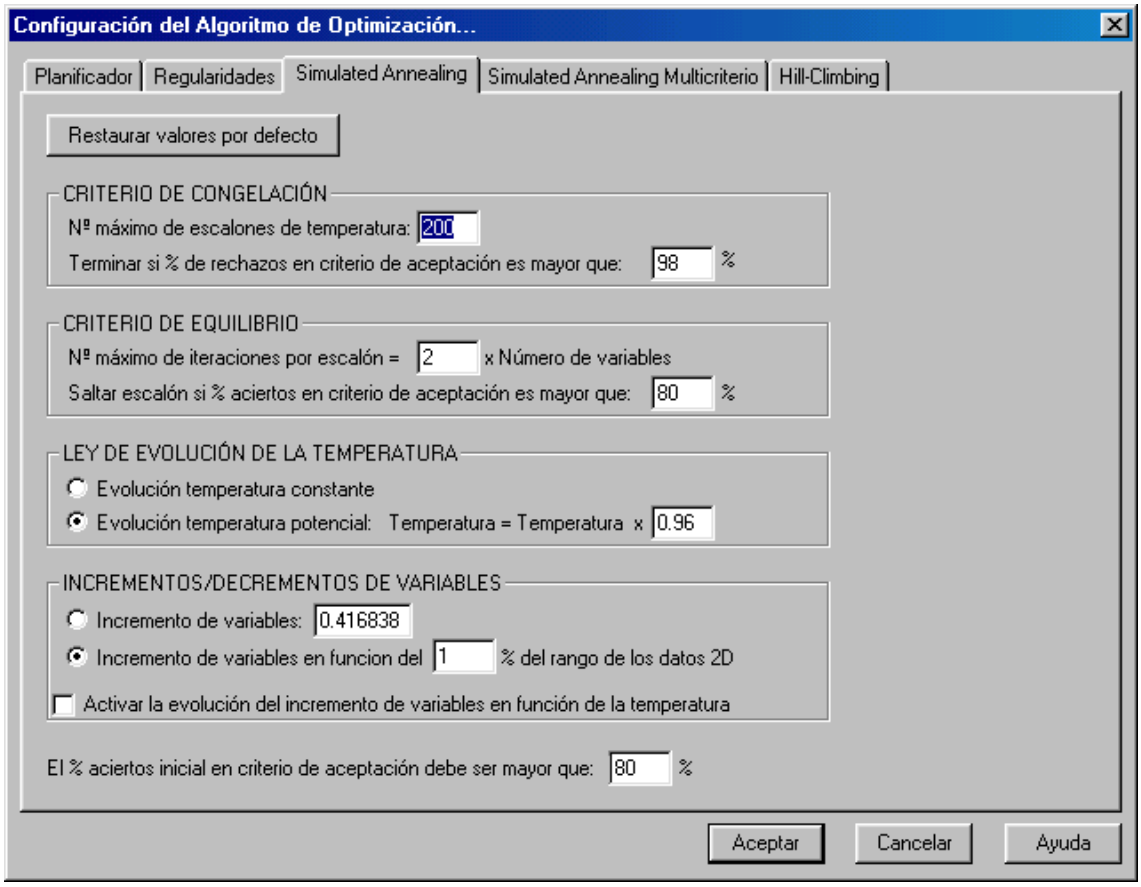
6.4 Pruebas y resultados obtenidos con REFER

Hemos probado los ejemplos de Marill y "Leclerc y Fischler" y hemos podido comprobar la validez de nuestro algoritmo de optimización Simulated Annealing, con las dos regularidades que se han implementado en este PFC.

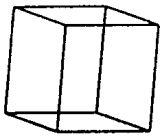
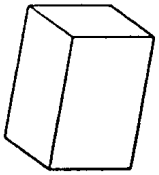
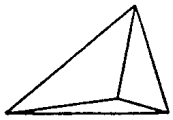
Los parámetros que se han utilizado para probar todos los ejemplos son los que tiene el programa por defecto:

- Se utilizan las dos regularidades que hemos implementado, con sus coeficientes de ponderación correspondientes:

- Se utiliza el Algoritmo Simulated Annealing, con los siguientes parámetros:



6.4.1 Pruebas con los ejemplos de Marill:

					
Ejemplo	**A**	**B**	**C**	**D**	**E**
Puntos	(-2.89 -1.62) (0.57 -1.62) (0.92 2.32) (-2.54 2.32) (-0.92 -2.32) (2.54 -2.32) (2.89 1.62) (-0.57 1.62)	(-3.43 -1.01) (1.06 4.63) (-2.4 4.63) (-1.55 -2.38) (1.91 -2.38) (2.94 3.26) (-0.53 3.26)	(1.52 -3.08) (1.52 2.12) (0.48 5.08) (0.48 -0.12)	(-4.33 -1.56) (1.79 -1.53) (0.54 2.54) (-0.14 -0.98)	(-0.67 1.5) (3.67 1.07) (1.67 0.47) (-2.67 0.9) (-0.67 -0.47) (3.6 -0.9) (1.67 -1.5) (-2.67 -1.07)
Lineas	(0 1) (1 2) (2 3) (3 0) (4 5) (5 6) (6 7) (7 4) (0 4) (1 5) (2 6) (3 7)	(1 2) (2 0) (3 4) (4 5) (5 6) (6 3) (0 3) (1 5) (2 6)	(0 1) (1 2) (2 3) (3 0) (0 2) (1 3)	(0 1) (0 2) (0 3) (1 2) (1 3) (2 3)	(0 1) (1 2) (2 3) (3 0) (0 4) (1 5) (2 6) (3 7)
Rango X	5.78	6.37	1.04	6.12	6.34
Rango Y	4.64	7.01	8.16	4.10	3.00

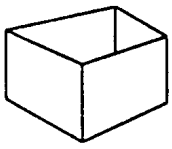
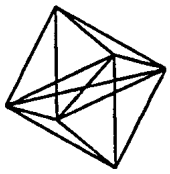
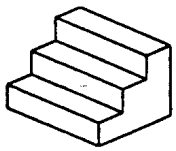
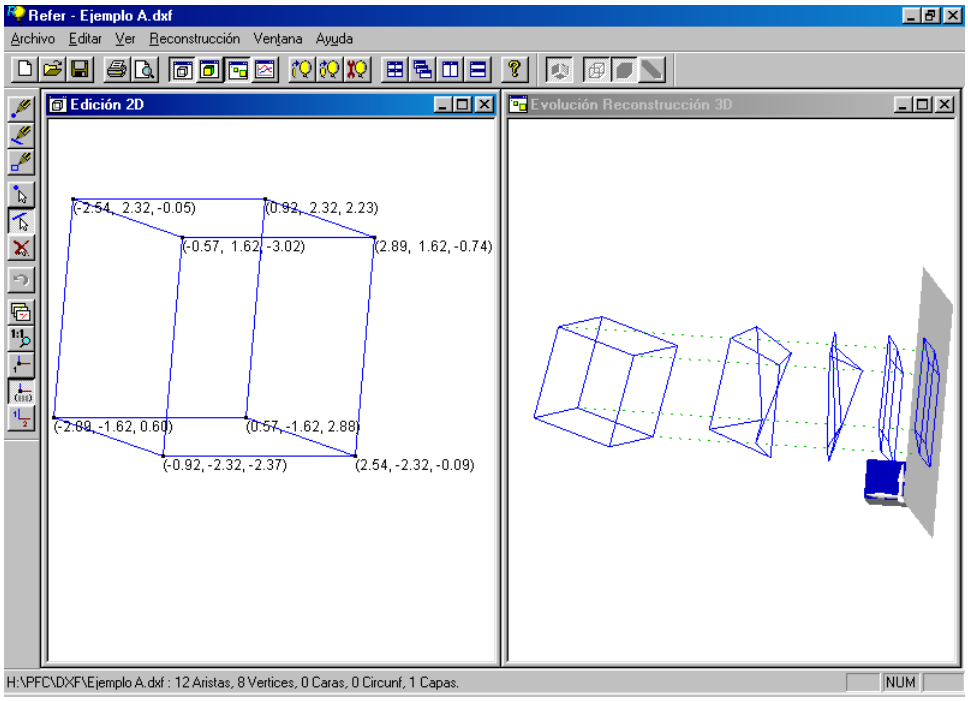
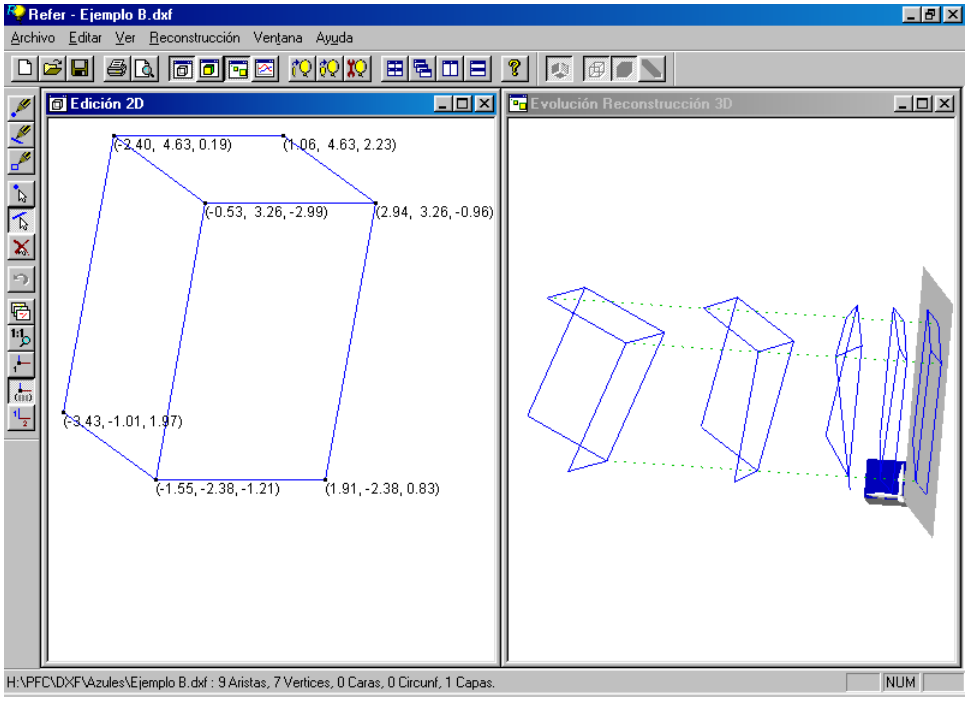
			
Ejemplo	**F**	**G**	**H**
Puntos	(-2.67 -0.47) (0.8 -2.47) (0.8 1.07) (-2.67 2.67) (2.33 1.55) (4.7 -0.7) (4.7 2.33) (2.33 3.7)	(-2.51 -0.59) (0.9 1.03) (2.51 0.59) (-0.9 -1.03) (0.93 -2.57) (-0.93 2.57)	(-2.4 -1.45) (0.14 2.07) (-0.71 1.71) (-0.71 0.9) (-1.56 0.54) (-1.56 -0.28) (-2.4 -0.63) (0.78 -0.79) (3.32 -1.72) (3.32 0.73) (2.47 0.37) (2.47 -0.45) (1.63 -0.8) (1.63 -1.62) (0.78 1.98)
Lineas	(0 3) (0 1) (1 2) (2 3) (1 5) (5 6) (2 6) (3 7) (7 4) (7 6)	(1 2) (2 3) (0 1) (0 3) (4 0) (4 1) (4 2) (4 3) (5 0) (5 1) (5 2) (5 3) (0 2) (1 3)	(1 2) (2 3) (3 4) (4 5) (5 6) (6 0) (7 9) (8 9) (9 10) (10 11) (11 12) (12 13) (13 14) (14 7) (0 7) (1 9) (2 10) (3 11) (4 12) (5 13) (6 14)
Rango X	7.37	5.02	5.72
Rango Y	6.17	5.14	4.77

Figura 6-6. Ejemplos de Marill

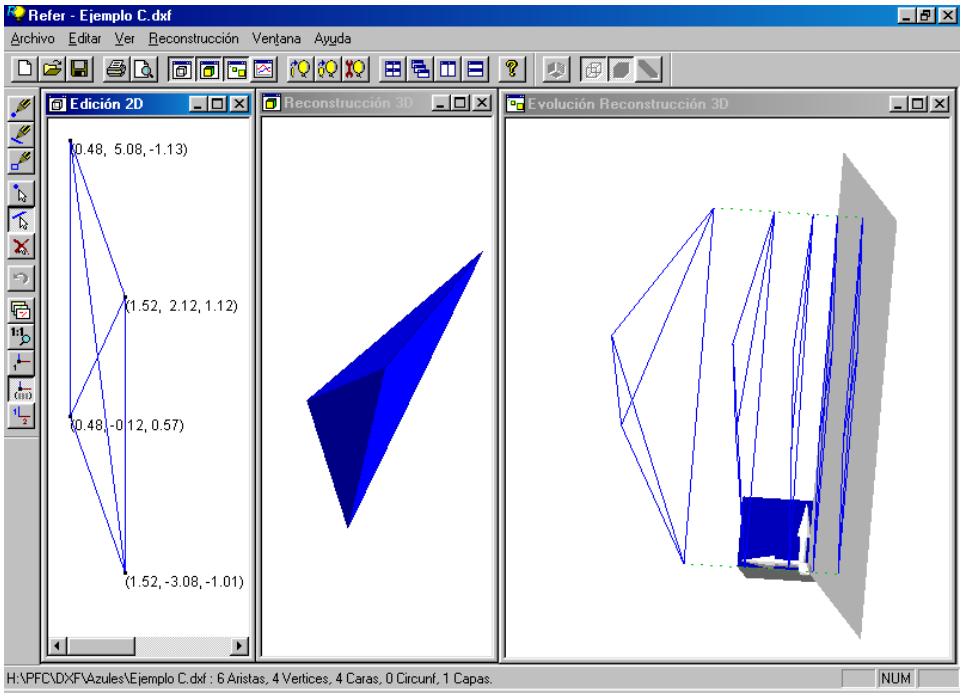
Ejemplo A



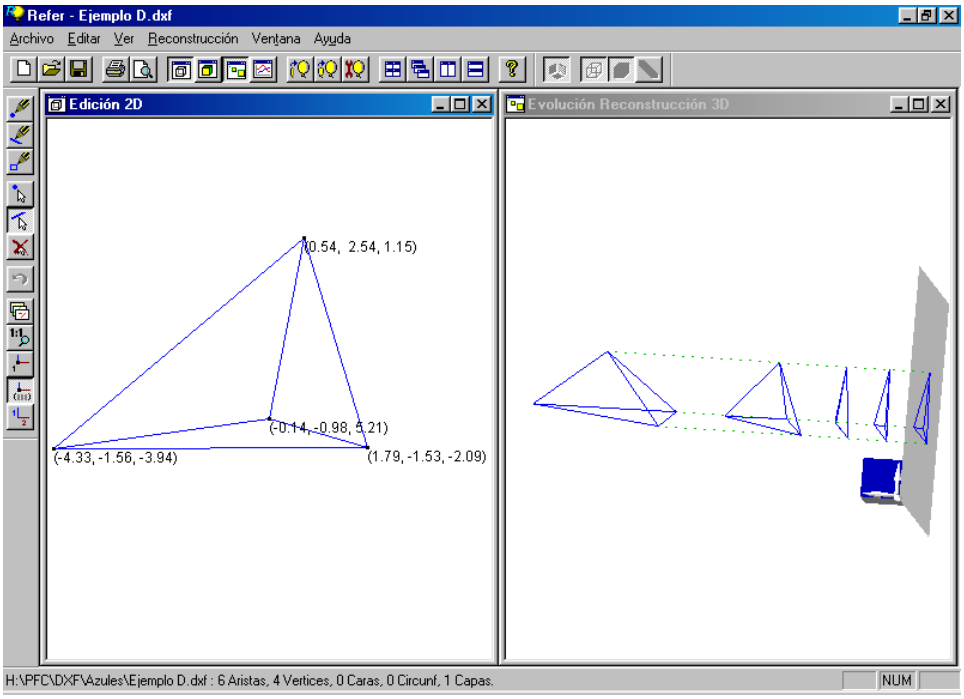
Ejemplo B



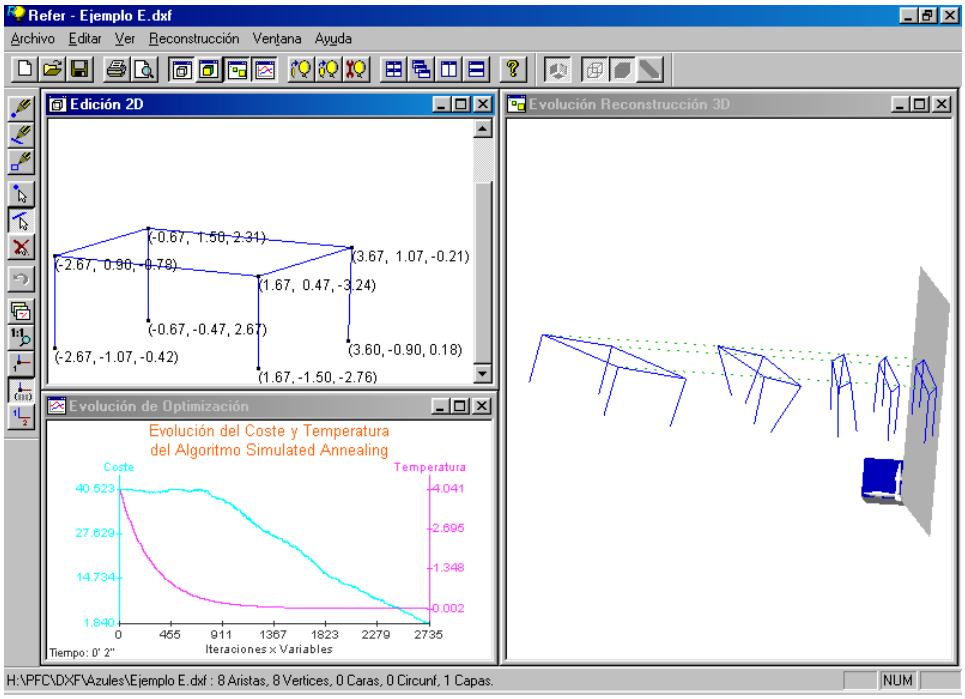
Ejemplo C



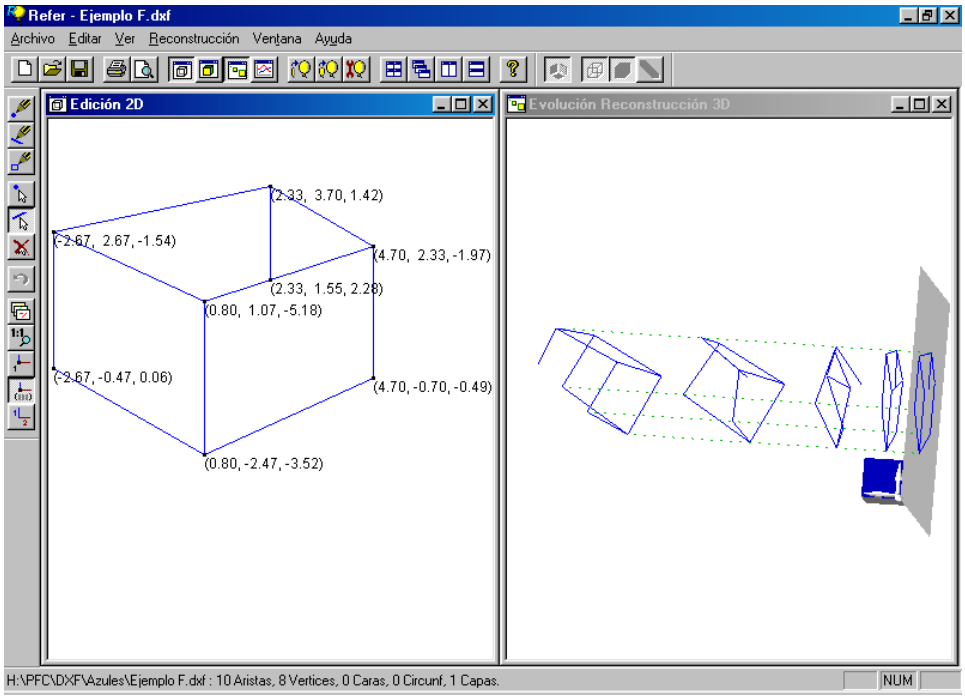
Ejemplo D



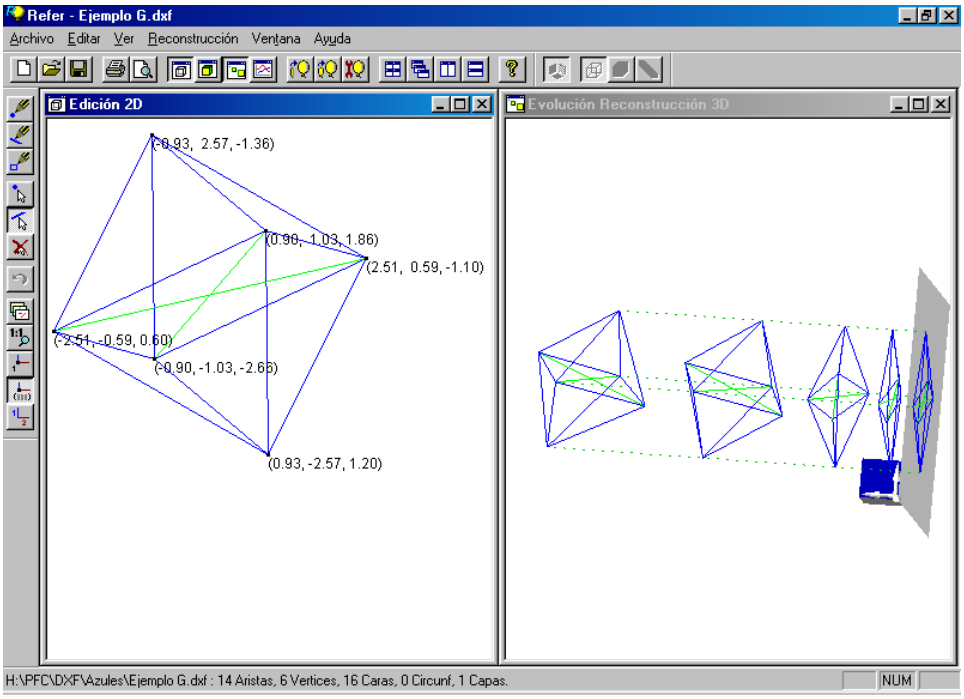
Ejemplo E



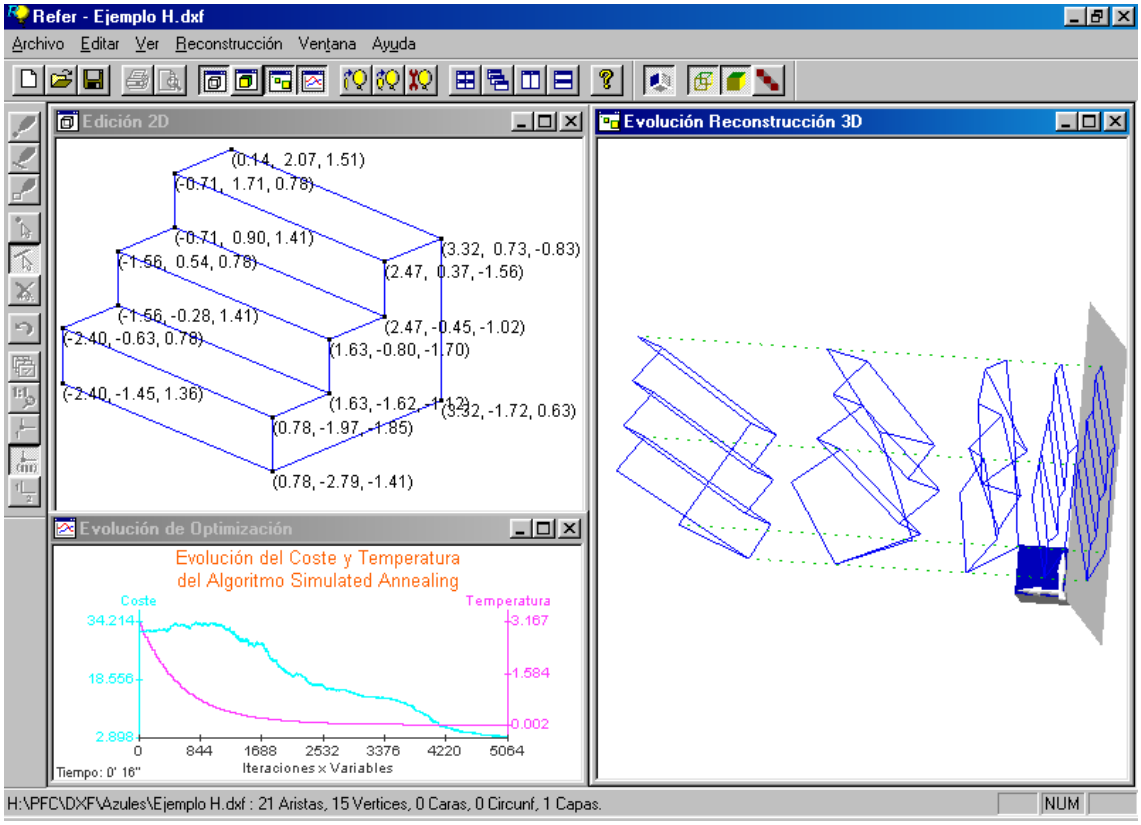
Ejemplo F



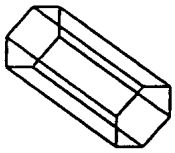
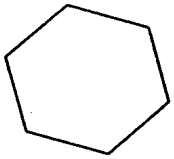
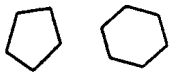
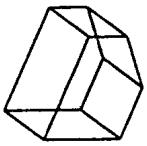
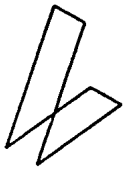
Ejemplo G



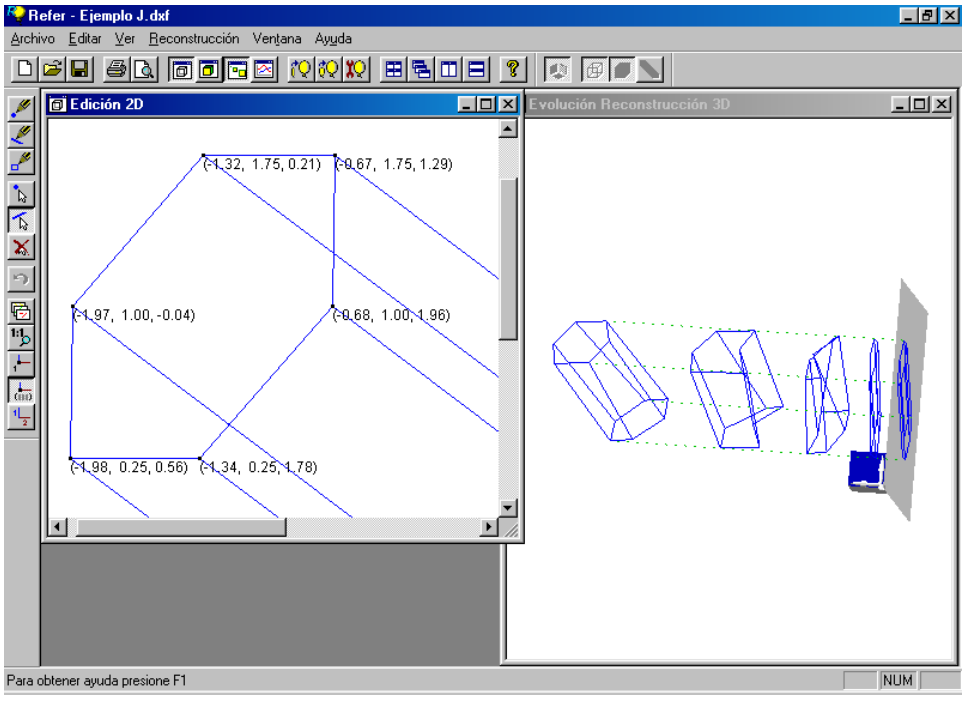
Ejemplo H



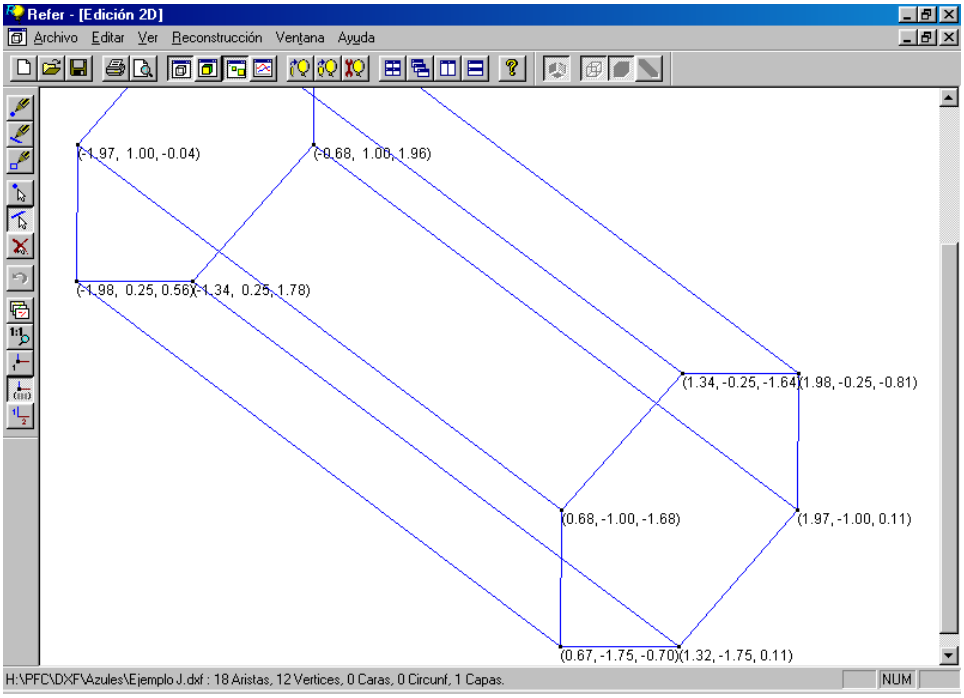
6.4.2 Pruebas con los ejemplos de Leclerc y Fischler

					
Ejemplo	**J**	**K**	**L**	**M**	**N**
Puntos	(1.97 -1.00) (1.32 -1.75) (0.67 -1.75) (0.68 -1.00) (1.34 -0.25) (1.98 -0.25) (-0.68 1.00) (-1.34 0.25) (-1.98 0.25) (-1.97 1.00) (-1.32 1.75) (-0.67 1.75)	(0.96 -0.27) (0.24 -0.89) (-0.72 -0.61) (-0.96 0.27) (-0.24 0.89) (0.72 0.61)	(0.81 -0.19) (-0.02 -0.96) (-0.82 -0.41) (-0.49 0.71) (0.52 0.85) (3.96 -0.27) (3.24 -0.89) (2.28 -0.61) (2.04 0.27) (2.76 0.89) (3.72 0.61)	(0.15 -0.06) (0.80 -0.06) (0.99 0.38) (0.86 0.81) (0.54 0.81) (-0.18 0.19) (0.46 0.19) (0.66 0.63) (0.53 1.06) (0.20 1.06)	(-0.58 0.24) (0.95 1.36) (1.50 1.04) (-0.02 -0.08) (0.30 2.89) (0.86 2.56)
Lineas	(0 1) (1 2) (2 3) (3 4) (4 5) (5 0) (6 7) (7 8) (8 9) (9 10) (10 11) (11 6) (0 6) (1 7) (2 8) (3 9) (4 10) (5 11)	(0 1) (1 2) (2 3) (3 4) (4 5) (5 0)	(0 1)(1 2)(2 3) (3 4) (4 0) (5 6) (6 7) (7 8) (8 9) (9 10) (10 5)	(0 1) (1 2) (2 3)(3 4)(4 0) (5 6)(6 7)(7 8) (8 9)(9 5)(0 5) (1 6)(2 7)(3 8) (4 9)	(0 1)(1 2) (2 3)(3 5) (5 4)(4 0)
Rango X	3.96	1.92	4.78	1.17	2.08
Rango Y	3.50	1.78	1.85	1.12	2.97

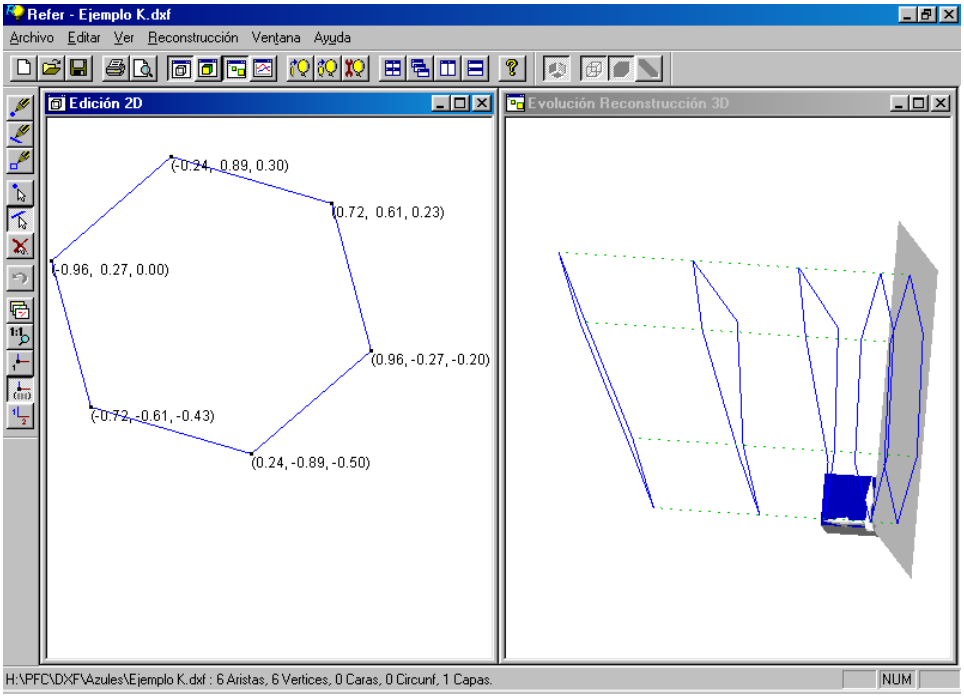
Ejemplo J



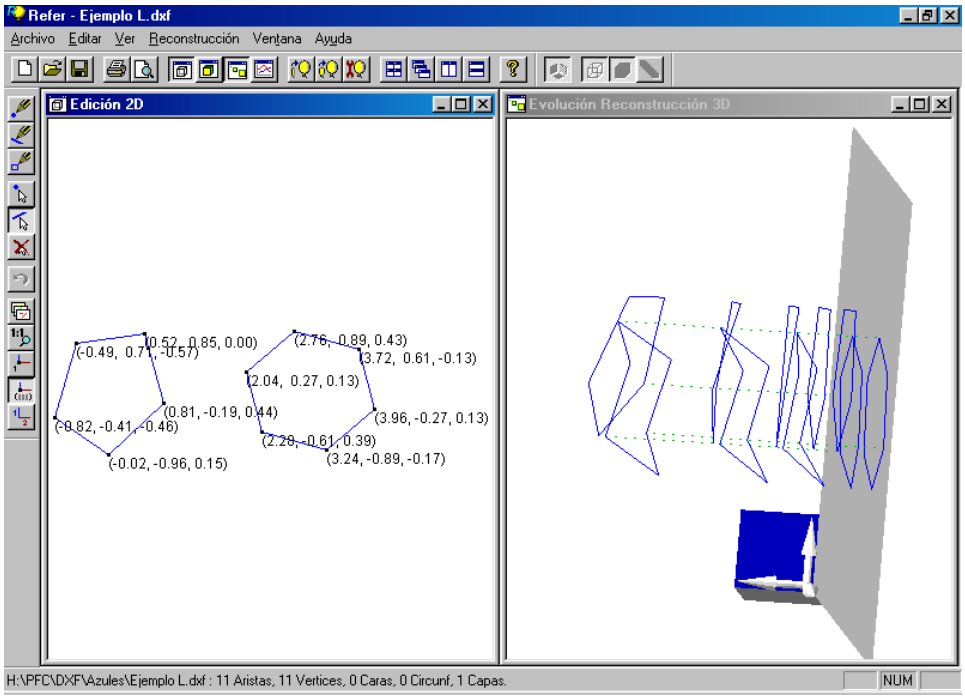
Aquí se muestra otra ventana con el resto del modelo del Ejemplo J para que se puedan apreciar las coordenadas Z de los vértices:



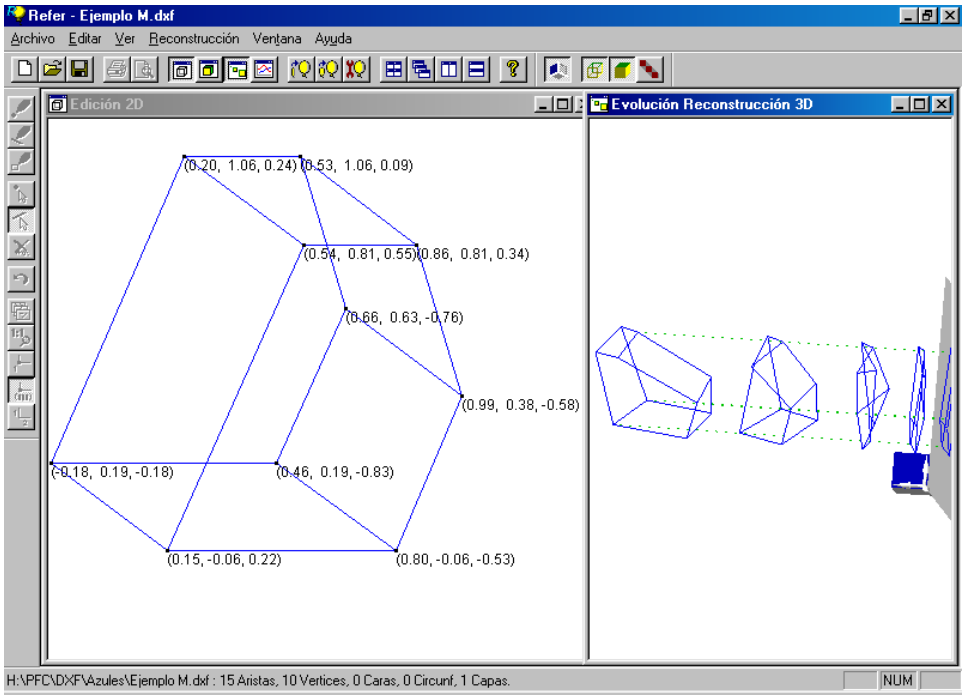
Ejemplo K



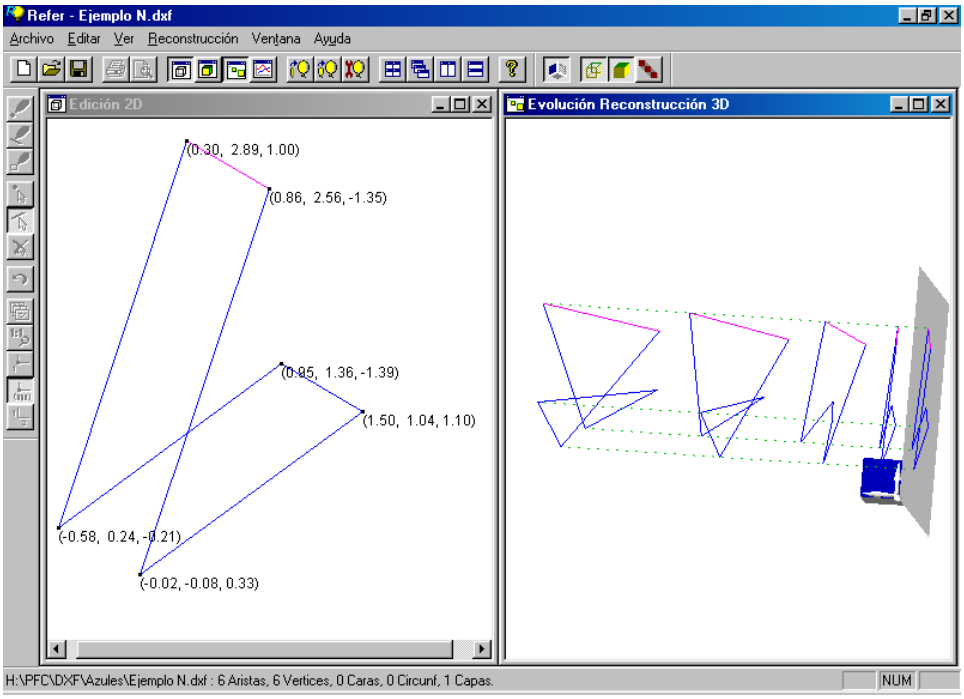
Ejemplo L



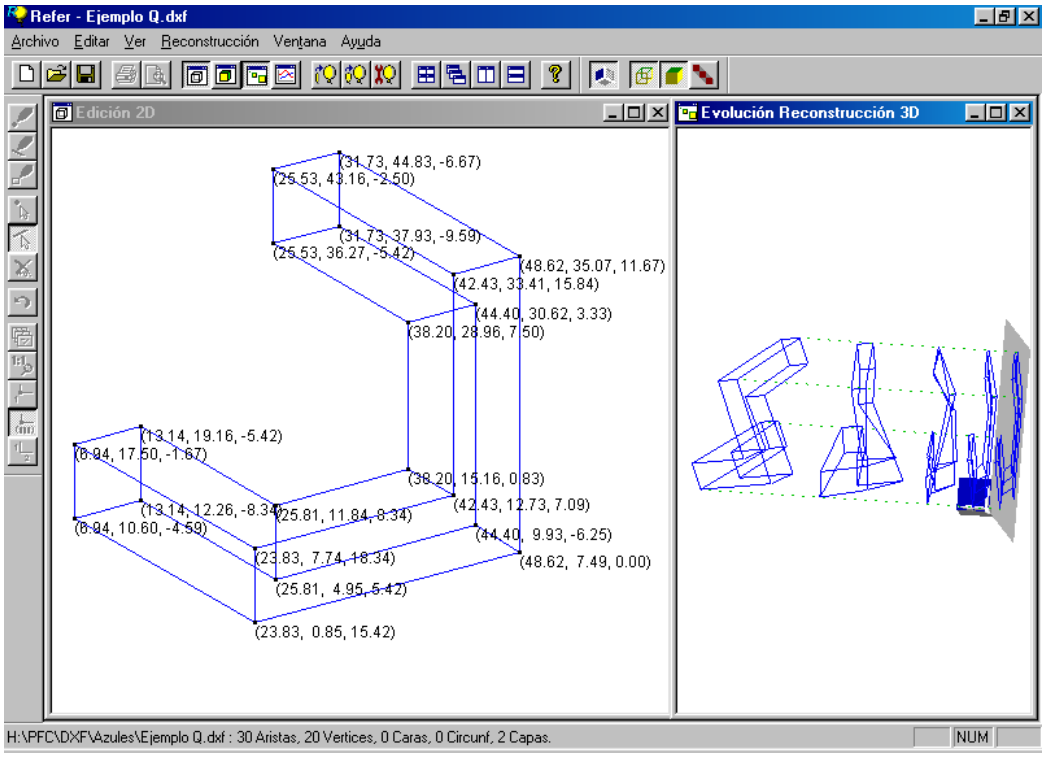
Ejemplo M



Ejemplo N



Y por último, un ejemplo de los nuestros:



7. RESULTADOS Y CONCLUSIONES

Los resultados obtenidos han sido altamente satisfactorios para este Proyecto Final de Carrera. Se ha implementado el algoritmo de Hill-Climbing, y ha funcionado bien, incluso mejor de lo que se esperaba. Es un algoritmo sencillo pero muy robusto, y por las pruebas demuestra ser útil para refinar la solución final. Es decir, con el algoritmo Simulated Annealing nos aproximaríamos rápidamente a la solución buena, y a partir de aquí, una ejecución del Hill-Climbing sobre la solución anterior nos daría un resultado exacto, la solución buena que andamos buscando.

Como se deduce del párrafo anterior, hemos conseguido implementar el algoritmo Simulated Annealig con resultados también muy buenos. Por sí solo es capaz de encontrar la solución buena muchísimo más rápido que el Hill-Climbing cuando la figura comienza a complicarse con muchos vértices y aristas. A veces, cuando está a punto de llegar a la solución buena, se detiene porque ha llegado al máximo de iteraciones. Esto pasa cuando al principio de la ejecución pierde mucho tiempo aceptando soluciones malas, y claro, al final le falta tiempo.

Pensamos que es necesario un estudio detallado de los parámetros de este algoritmo, porque son muchos e influyen unos con otros.

Por otro lado, se ha implementado una ventana de edición 2D que ha servido para crear todos los ejemplos, cumpliendo el objetivo previsto.

Se han diseñado dos ventanas de representación 3D, una que muestra la solución final y la solución “on-line” en tiempo de ejecución. Y otra ventana que muestra el dibujo sobre un plano 2D, la solución final y algunas de las soluciones intermedias proyectadas sobre dicho plano. En el aspecto de la visualización, OpenGL se ha mostrado como una herramienta muy potente y perfectamente adecuada a este tipo de necesidades.

Por último, se ha diseñado una ventana (no estaba prevista inicialmente) en la que se muestra una gráfica de la evolución de los costes de los algoritmos, para comprobar el correcto funcionamiento de los mismos.

8. DESARROLLOS FUTUROS

El algoritmo Simulated Annealing tiene muchos parámetros que se pueden ajustar, y dependen unos de otros, con lo que es difícil probar todas las posibles combinaciones dentro del ámbito de este Proyecto Final de Carrera. Este estudio en profundidad debería hacerse en alguna Tesis futura.

Por ejemplo, en un algoritmo Simulated Annealing típico se le da un valor constante al ΔZ. También podría considerarse que el ΔZ no fuera constante, sino que fuera variable, de modo que tuviera un valor elevado al principio para acelerar la llegada a una solución buena y que fuera disminuyendo para que al final se tuviera la suficiente precisión. En nuestro PFC hemos optado por poder ajustar el ΔZ en función de la temperatura que, como veremos más adelante, comienza elevada para ir descendiendo. No está clara la razón, pero en la práctica parece que el algoritmo no mejora sus resultados con esta última opción activada. Este sería un aspecto a estudiar con más detalle.

Otro punto muy importante a estudiar es el tema de **la ponderación y las relaciones entre las regularidades**. Esta cuestión tiene una gran importancia porque de ella depende directamente la validez y la exactitud de la Función Objetivo. Este asunto también debería tratarse con profundidad en estudios posteriores.

Un línea de investigación futura podría ser el algoritmo Simulated Annealig Multicriterio. En este PFC se ha realizado una primera aproximación, sin obtener resultados positivos. En el *Capítulo 4.3. Optimización Simulated-Annealing Multicriterio* , pag. 81. analizamos el algoritmo y sus posibles mejoras.

9. BIBLIOGRAFÍA

Programación Avanzada con Visual C++
David J. Kruglinski
McGraw Hill – Serie Microsoft Press. 1996

Visual C++. Aplicaciones para Windows
Fco. Javier Ceballos Sierra
Ra-ma.

Cómo se hace con Visual C++ 4
Scott Stanfield
Inforbook's, S.L. 1996

El lenguaje de programación C. Segunda Edición.
Brian W. Kernighan, Dennis M. Ritchie
Prentice-Hall Hispanoamericana, S.A. 1995

Programación en Windows
Charles Petzold
Anaya Multimedia, S.A. – Serie Microsoft Press

Programación en OpenGL
Richard S. Wright Jr., Michael Sweet
Anaya Multimedia, S.A. 1997

OpenGL Programming Guide: The official guide to Learning OpenGL, Release 1
OpenGL Architecture Review Board
Jackie Neider, Tom Davis, Mason Woo
Addison-Wesley. 1993

Graphical User Interface Design and Evaluation
David Redmond-Pyle, Alan Moore
Prentice Hall. 1995

Enciclopedia of Graphics File Formats. Second Edition.
James D. Murray and William vanRyper
O'Reilly & Associates, Inc. 1996

9.1 Direcciones Internet

Web del Grupo REGEO
http://www.tec.uji.es/regeo

The Graphics File Formats Page
HTTP://WEB.DCS.ED.AC.UK/HOME/MXR/GFX/

DXF Release 12 specification
HTTP://WEB.DCS.ED.AC.UK/HOME/MXR/GFX/3D/DXF12.SPEC

MSDN Online Member Area
http://msdn.microsoft.com
http://premium.microsoft.com/msdn/library

OPENGL Coding/Programming Courses & Tutorials
HTTP://WWW.OPENGL.ORG/CODING/CODING.HTML

9.2 Referencias

[1] Company P. "Integrating Creative Steps in CAD Process". *International Seminar on Principles and Methods of Engineering Design*, Napoli, 1.997. Vol. 1, pp. 295-322.

[2] Gomis J.M.; Company P., and Contero M. "Reconstrucción de modelos poliédricos a partir de sus vistas normalizadas". *Anales de Ingeniería Mecánica*, Año 11, vol. 1, 1.997, pp. 383-391.

[3] Gomis J.M., Company P. y García J., "Preprocesador para modelado geométrico tridimensional a partir de la delineación 2D de axonometrías". *Actas del IX Congreso internacional de Expresión Gráfica en la Ingeniería*. Volumen 2, 1997, pp. 345-354.

[4] Sugihara K. Machine interpretation of Line Drawings. MIT Press 1986.

[5] Nagendra I.V. and Gujar U.G.. "3-D Objects From 2-D Orthographic Views – A Survey". *Computers & Graphics*. Vol 12, No. 1, 1988. pp. 111-114.

[6] Wang W. and Grinstein G. "A Survey of 3D Solid Reconstruction from 2D Projection Line Drawings". *Computer Graphics Forum*. Vol. 12, No 2, 1993, pp. 137-158.

[7] Yan Q.W., Philip Chen C.L. and Tang Z. "Efficient algoritm for the reconstruction of 3D objects from orthographic projections". *Computer Aided Design*. Vol. 26, No 9, 1994, pp. 699-717.

[8] Marill, T. "Emulating the Human Interpretation of Line-Drawings as Three-Dimensional Objects". *International Journal of Computer Vision*. Vol. 6, No. 2, 1991, pp. 147-161.

[9] Leclerc, Y. and Fischler M. "An Optimization-Based Approach to the Interpretation of Single Line Drawings as 3D Wire Frames". *International Journal of Computer Vision*. Vol. 9, No. 2, 1992, pp. 113-136.

[10] Lipson H. and Shpitalni M. "Optimization-Based Reconstruction of a 3D Object from a Single Freehand Line Drawing". *Computer Aided Design*. Vol. 28, No. 8, 1996, pp. 651-663.

[11] Wang, W. and Grinstein, G. "A polyhedral object's CSG-Rep reconstruction from a single 2D line drawing," *Proc. of 1989 SPIE Intelligent Robots and Computer Vision III.- AIgorithms and Techniques,* vol. 1192, pp. 230-238, (Nov 1989).

[12] Company, P., Gomis, J.M. and Contero, M. , "Geometrical Reconstruction from Single Line Drawings Using Optimization-Based Approaches". WSCG'99. Conference proceedings, edited by Vaclav Skala (ISBN 80-7082-490-5), Volume II, 1999, pp. 361-368.

[13] Conesa, J., Company, P., y Gomis, J.M., "Initial modeling strategies for geometrical reconstruction optimization-based approaches". 11th ADM International conference, Volume B, 1999, pp. 161-171.

[14] Press, W.H., Flannery, B.P., Teukolsky, S.A. and Vetterling, W.T. "Numerical Recipes in C". Cambridge University Press, 1988 (reprinted 1991).

[15] Kirkpatric S., Gelatt C.D. and Vecchi M.P., 1983. "Optimization by simulated annealing". *Science*, vol. 220, No. 4598, pp. 671-680.

[16] Metropolis N., Rosenbluth A.W., Rosenbluth M.N. and Teller A.H., 1953. Equation of state calculations by fast computing machines. *The Journal of Chemical Physics*, vol. 21, no. 6, pp. 1087-1091.

[17] Kouvelis P., Chiang W. and Fitzsimmons J., 1992. Simulated annealing for machine layout problems in the presence of zoning constraints. *European Journal of Operational Research*, vol. 57, pp. 203-223.

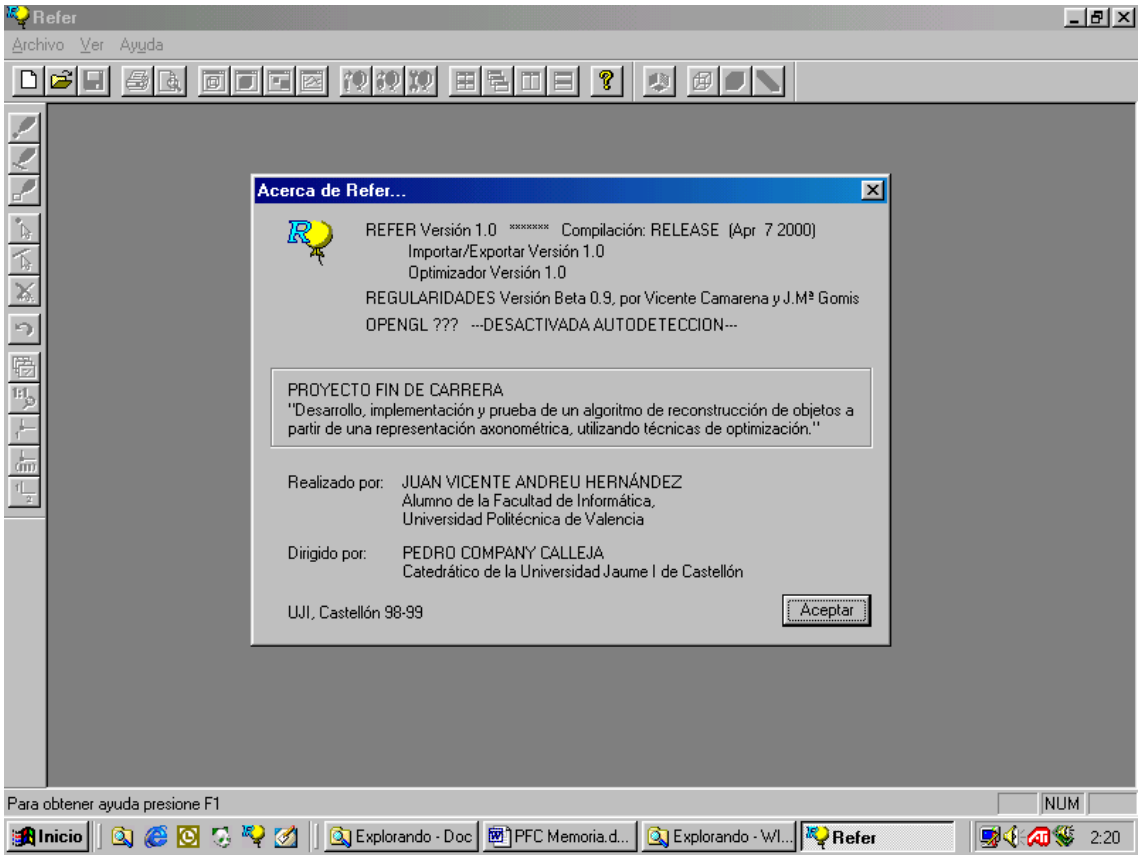
[18] Koffka K., 1935 (1967). Principles of Gestalt Psychology. Harcourt Brace, New York.

[19] *DXF Release 12 specification.* http://web.dcs.ed.ac.uk/home/mxr/gfx/3d/DXF12.spec

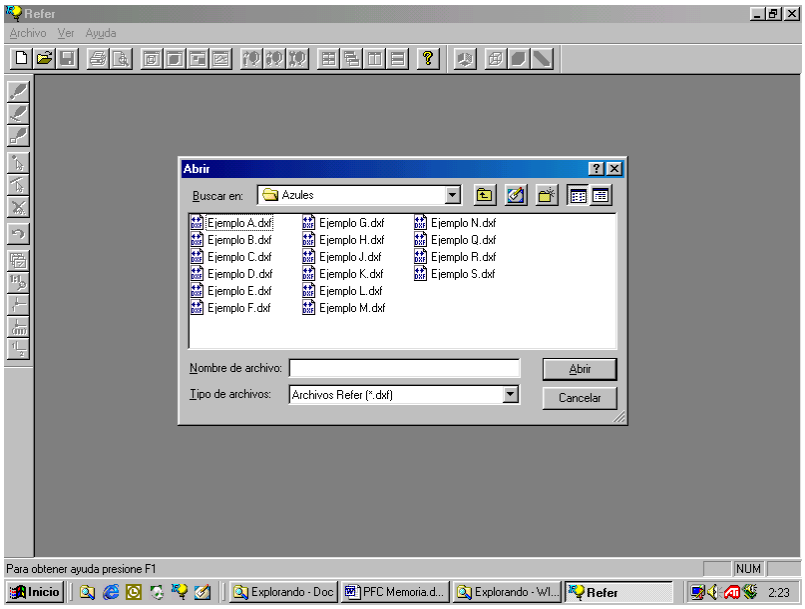
10. APÉNDICE A: FUNCIONAMIENTO DE REFER

Refer es una aplicación creada con Microsoft Visual C++ 5.0 y MFC, y utiliza la librería gráfica OpenGL, disponible para Windows 95 como un add-on, y que viene integrado en el Sistema Operativo a partir de la versión Windows 95 SR-2, Windows 98 y Windows NT 4.0.

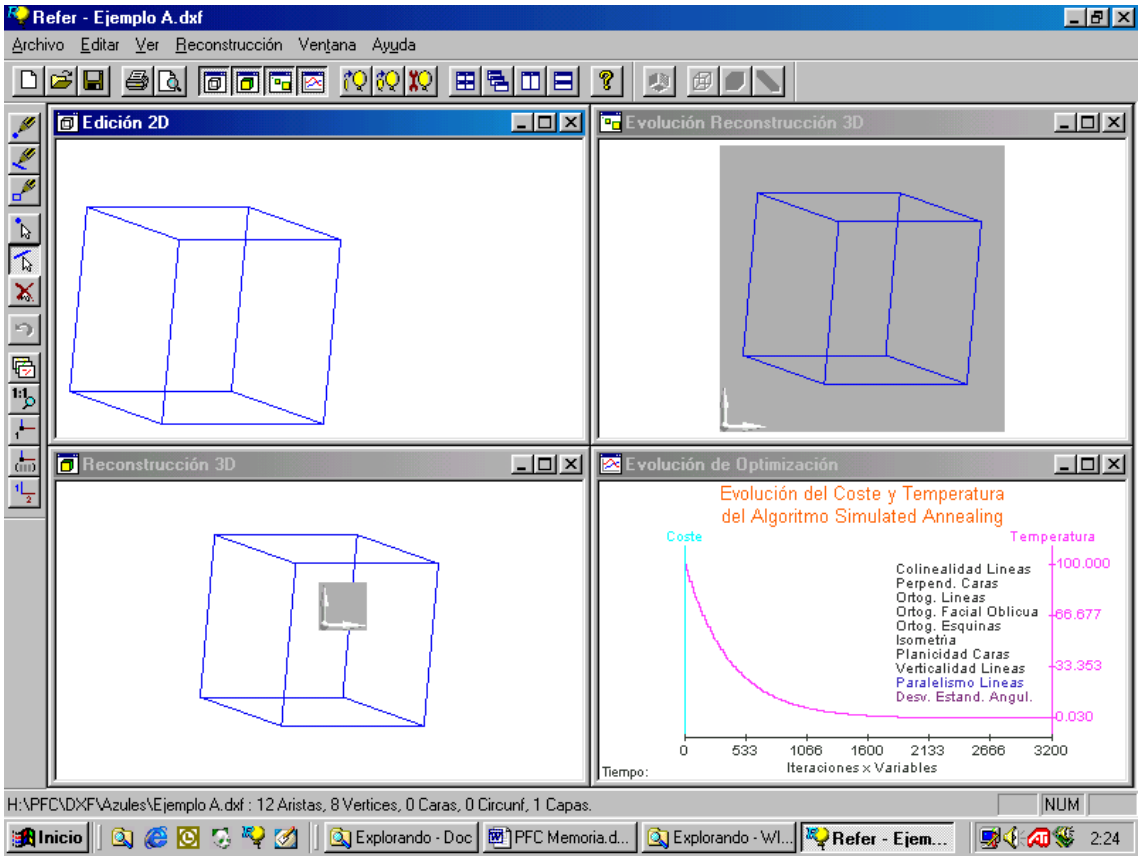
Por tanto tiene todo el aspecto y funcionalidad de una aplicación Windows estándar. Este es su aspecto nada más abrirse y pulsar *Ayuda + Acerca de Refer*



El procedimiento para abrir un archivo es el mismo que cualquier programa de Windows:



Después de abrir un fichero, este es el aspecto que presenta:



hay 4 ventanas, cada una tiene su nombre que la identifica:

- **Edición 2D.** Como su nombre indica, esta es la ventana 2D en la que podemos crear, visualizar, retocar y guardar los dibujos 2D.

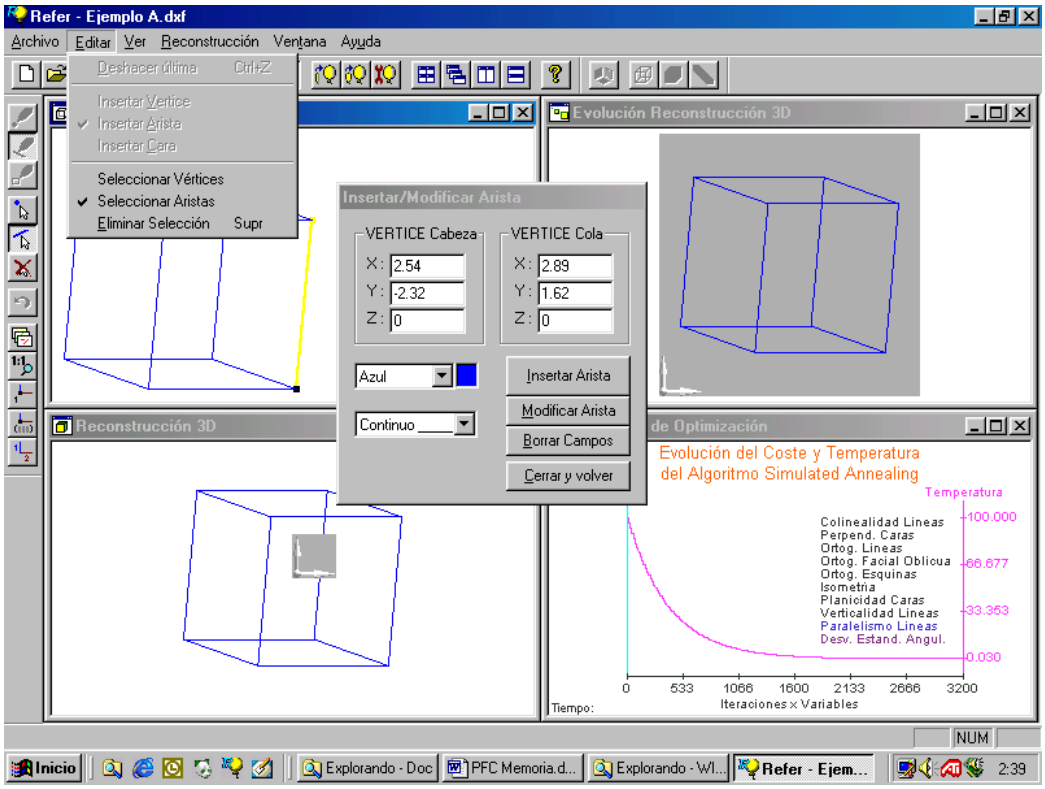
- **Reconstrucción 3D.** Esta es la ventana que nos muestra en tiempo real como se está reconstruyendo la figura 3D. Cuando el algoritmo de reconstrucción ha terminado, podemos pinchar con el ratón sobre la ventana y mover libremente el objeto en el espacio sobre sí mismo.

- **Evolución Reconstrucción 3D.** Cuando el algoritmo de reconstrucción ha terminado, esta ventana nos muestra el dibujo de la ventana de *Edición 2D* sobre un plano 2D, la solución final de la ventana de *Reconstrucción 3D* y algunas de las soluciones intermedias proyectadas sobre dicho plano. También podemos pinchar con el ratón sobre la ventana y mover libremente estos objetos en el espacio sobre sí mismo.

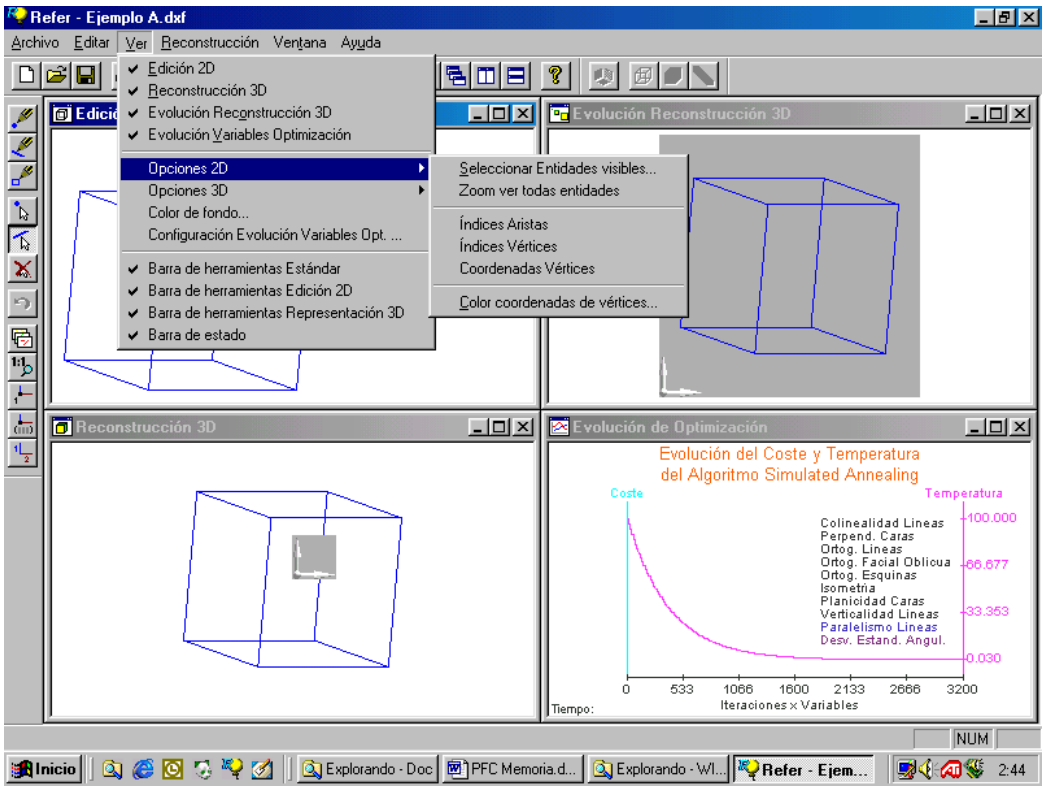
- **Evolución de Optimización.** Cuando el algoritmo de reconstrucción ha terminado, esta ventana nos muestra mediante una gráfica la evolución del coste de los algoritmos de optimización, la evolución de la temperatura en el algoritmo Simulated Annealing, y la evolución de los escalones en el algoritmo Hill-Climbing. Hay una opción para representar los valores de cada variable z.

Hay varios menus, estructurados de forma similar a cualquier programa Windows, y agrupados según su función. La mayoría de funciones más importantes se pueden acceder desde las barras de botones que hay disponibles.

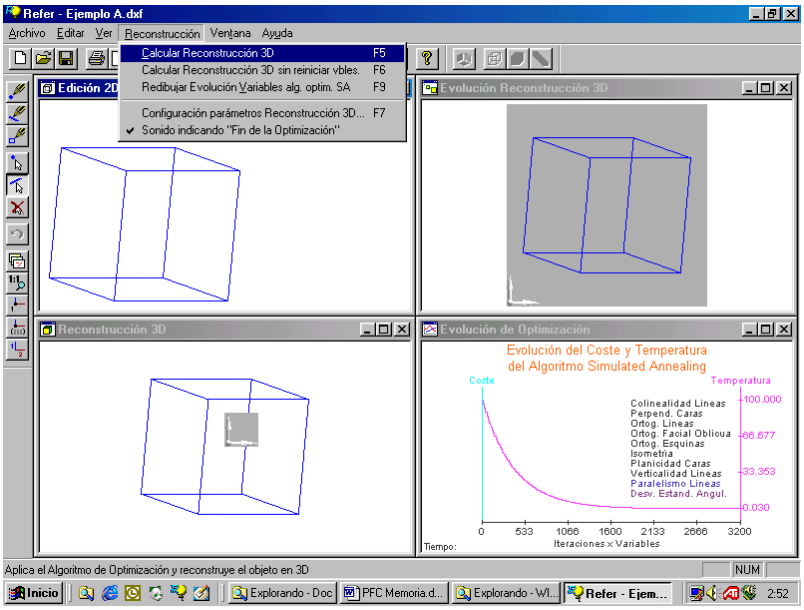
En la figura siguiente se puede ver el menú de *Edición* abierto, al tiempo que se está *Insertando/Modificando* una arista 2D seleccionada en la ventana de *Edición 2D*.



En la figura siguiente se puede ver desplegado el menú de *Ver* y además el submenú de *Opciones 2D*.

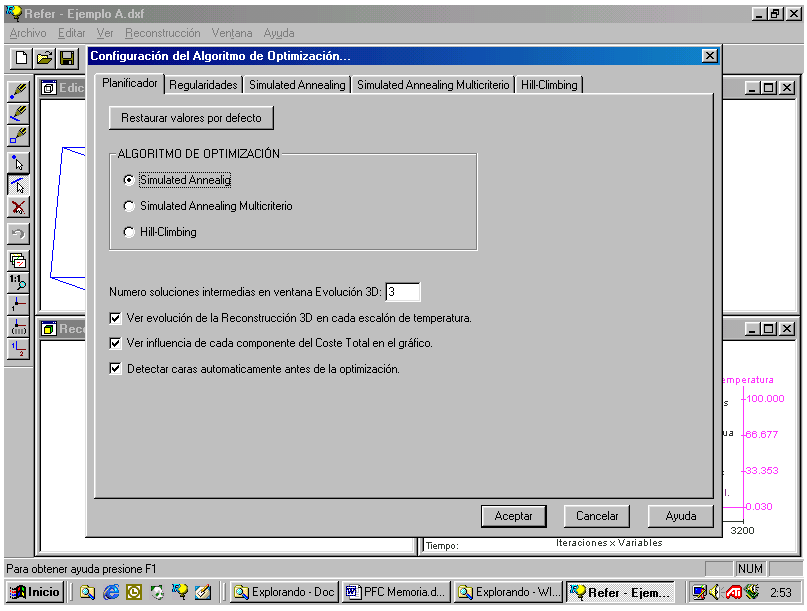


Este es el menú de *Reconstrucción*:

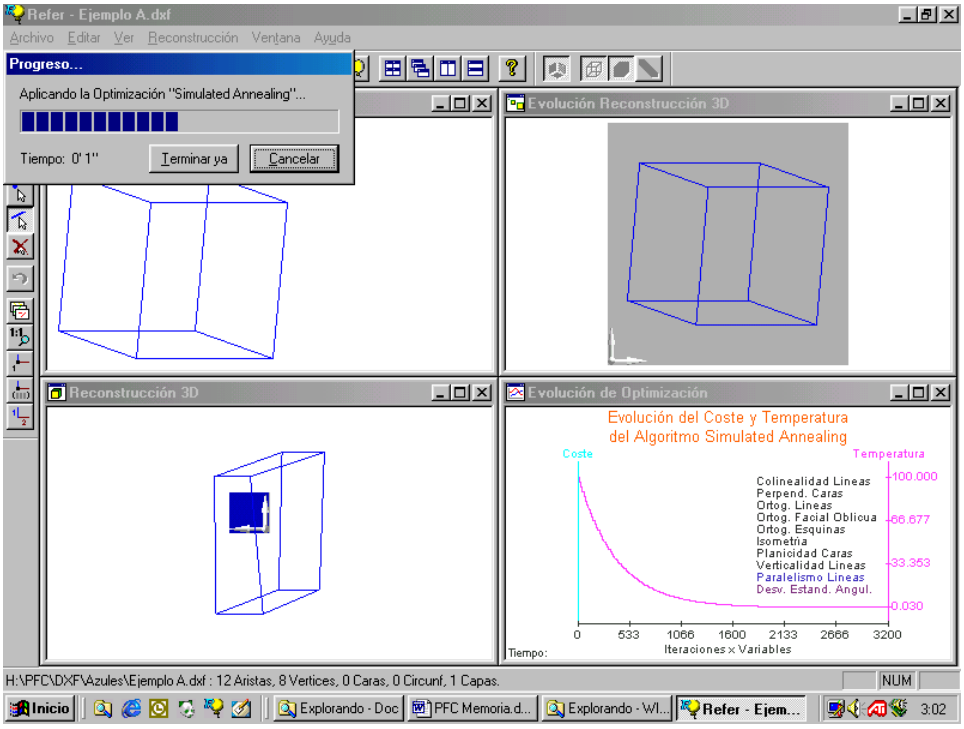


Desde este menú, si elegimos la primera opción (que vemos coincide con pulsar F5) se lanza ya la reconstrucción 3D, con los parámetros que estén configurados.

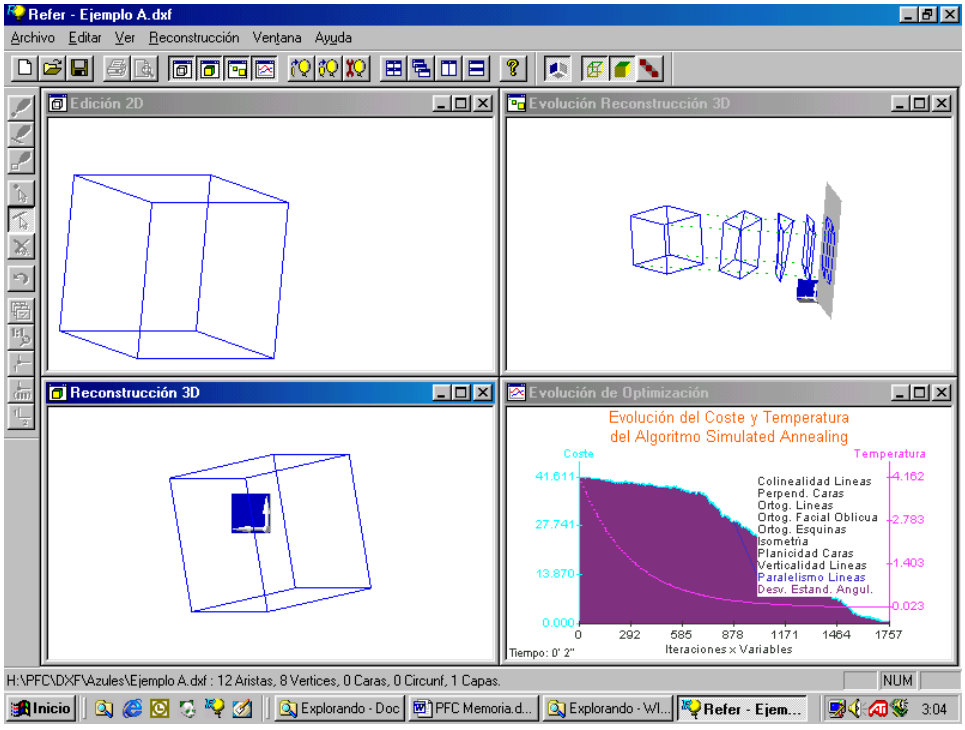
Y este es el diálogo principal desde el que se elige el algoritmo de optimización, los parámetros de éste, las regularidades, etc. Se accede a él desde *Reconstrucción* + *Configuración parámetros reconstrucción 3D...*



En la siguiente figura podemos ver el algoritmo en pleno funcionamiento. Notar como la ventana de *Reconstrucción 2D* está mostrando la mejor solución encontrada hasta ese momento en que capturamos la pantalla.



Por último vemos como quedan todas las ventanas con el algoritmo terminado:



Finalmente, si activamos la detección automática de caras, el resultado es espectacular: