INTERFAZ DE USUARIO GRÁFICA PARA PROGRAMA DE ELEMENTOS FINITOS

CAMILO AKIMUSHKIN VALENCIA

UNIVERSIDAD INDUSTRIAL DE SANTANDER

FACULTAD DE INGENIERÍAS FÍSICO-MECÁNICAS

ESCUELA DE INGENIERÍA MECÁNICA

BUCARAMANGA, COLOMBIA

2009

INTERFAZ DE USUARIO GRÁFICA PARA PROGRAMA DE ELEMENTOS FINITOS

TRABAJO DE GRADO PARA OPTAR AL TITULO DE INGENIERO MECÁNICO

PRESENTADO POR

CAMILO AKIMUSHKIN VALENCIA

DIRECTOR

Ing. OMAR GELVEZ

CO-DIRECTOR

Ing. DAVID FUENTES

UNIVERSIDAD INDUSTRIAL DE SANTANDER

FACULTAD DE INGENIERÍAS FÍSICO-MECÁNICAS

ESCUELA DE INGENIERÍA MECÁNICA

BUCARAMANGA, COLOMBIA

2009

DEDICATORIA

Éste trabajo está dedicado a mi familia, en especial a mi madre quien siempre creyó en mí.

CONTENIDO

LISTA DE FIGURAS

LISTA DE TABLAS

CAPÍTULO 1

RESUMEN

TÍTULO: INTERFAZ DE USUARIO GRÁFICA PARA PROGRAMA DE ELEMENTOS FINITOS.[*]

AUTOR: AKIMUSHKIN VALENCIA, Camilo.[†]

PALABRAS CLAVE: Interfaz de usuario gráfica, elementos finitos, región discretizada o malla, programación orientada a objetos.

DESCRIPCIÓN:

La solución del problema de valor inicial mediante el método numérico de elementos finitos, requiere utilizar programas de computadora. Uno de los problemas que se pueden atacar con tal m'etodo es la conducción de calor en un sólido.

Cuando los programas se vuelven mas complejos, se distribuyen independientemente de tal forma que cada uno se pueda utilizar por aparte. En el caso de los elementos finitos, se requiere una serie de datos para obtener la solución deseada. Partiendo de la geometría y condiciones de frontera del problema, se debe definir una región discretizada o malla que sustituirá a la región verdadera por un gran conjunto de elementos (finitos) y sobre la cual se resolverá una serie de ecuaciones algebraicas que sustituyen a las ecuaciones diferenciales. Esta tarea la realiza un programa de enmallado o pre-procesador, leyendo archivos que contienen la geometría y las condiciones de frontera y escribiendo un archivo que contiene la región discretizada o malla. Con la malla obtenida, un programa de solución o *solver*, plantea y obtiene las ecuaciones algebraicas en elementos finitos.

Se evidencia la necesidad de una interfaz de usuario gráfica que permita introducir la geometría y las condiciones de frontera sin tener que escribir manualmente los archivos de entrada al pre-procesador y sin conocer la sintaxis utilizada por el programa de enmallado. En éste trabajo se desarrolla tal interfase para mallas bi-dimensionales. El programa tambien es un editor gráfico que permite crear, editar y combinar figuras. La geometría, las condiciones de frontera y los puntos que lee el pre-procesador se pueden modificar en cualquier momento para comparar los resultados bajo diferentes parámetros. Finalmente se escriben los archivos, se invoca al pre-procesador y se dibuja la malla generada por éste, todo ello automaticamente ocultando los detalles al usuario.

---

[*] Trabajo de Grado.

[†] Escuela de Ingeniería Mecanica, Facultad de Ingenierías Fisico-Mecánicas, Universidad Industrial de Santander. Director: Omar A. Gélvez Arocha.

CAPÍTULO 2

ABSTRACT

TITLE: GRAPHIC USER INTERFACE FOR FINITE ELEMENTS SOFTWARE.[*]

AUTHOR: AKIMUSHKIN VALENCIA, Camilo.[†]

KEY WORDS: Graphic user interface, finite elements, grid, object oriented programming.

DESCRIPTION:

The solution of the initial value problem through the finite elements method requires using computer programs. One of the problems that could be treated with such method is the heat conduction in solids.

When programs become more complex, they are distributed independently so each one could be used apart. With the finite elements method, a set of data is required in order to obtain the desired solution. From the geometry and boundary conditions of the problem, a discrete region or grid must be defined which will substitute the real region with a great amount of (finite) elements on which will be solved a set of algebraic equations that replaces the differential equations. This work is done by a discretization program or pre-processor, reading files containing the geometry and boundary conditions and writing a file containing the grid. With the grid obtained, a solution program or solver, formulates and solves the finite elements algebraic equations.

Becomes evident the necessity of a graphic user interface which permits to introduce the geometry and boundary conditions without having to write manually the input files for the pre-processor and without knowing the syntax used by the pre-processor. In this work, such interface is developed for bi-dimensional grids. The program is also a graphic editor to create, edit and combine figures. The geometry, the boundary conditions and the points that the pre-processor reads, can be modified in any moment to compare the results with different parameters. Finally, the files are written, the pre-processor is invoked and the grid generated by it is drawn. All of it is done automatically hiding the details to the user.

[*] Degree work.

[†] Escuela de Física, Facultad de Ciencias, Universidad Industrial de Santander. Director: Omar A. Gélvez Arocha.

Parte I

Problema

CAPÍTULO 3

INTRODUCCIÓN

En éste trabajo se presenta un programa que consiste en un editor gráfico de figuras en dos dimensiones con el cual se puede definir una serie de condiciones de frontera sobre un amplio espectro de geometrías para plantear problemas de conducción de calor al interior de un sólido. Con la geometría y condiciones de frontera ingresadas se obtiene una región discretizada o *malla* que se puede utilizar para plantear y resolver un sistema de ecuaciones algebraicas o *ecuaciones en elementos finitos*.

El programa genera y dibuja la región discretizada o *malla* lo cual se utiliza para observar los efectos de cada variable instantaneamente y permite ilustrar el funcionamiento de la herramenta de discretización o *enmallado* llamada *pre-procesador*. Además, si el usuario lo requiere, puede copiar el archivo de malla generado para utilizarlo manualmente con una herramienta de solución de ecuaciones en elementos finitos y dibujar el gradiente de temperaturas resultante mediante otra herramienta[*].

La capacidad de crear herramientas le permitió al hombre dominar su entorno y multiplicar el beneficio por su trabajo, gracias a esto no se tuvo que preocupar más por algunas de sus necesidades básicas pudiendo dedicarse a las tareas que su propio criterio le indicó como mas importantes. Desde entonces, las herramientas se han vuelto cada vez mas especializadas, tecnificadas e indispensables.

El conocimiento y refinamiento de una técnica ha marcado cada una de las etapas de desarrollo de la humanidad. Desde la edad de hierro hasta la revolución industrial, el dominio de la técnica ha logrado modificar la relación del ser humano con su entorno y sus semejantes. La técnica utilizada en éste proyecto es mas antigua de lo que parece, son los métodos numéricos. Éstos aparecieron con el cálculo diferencial pero existen algunos que son mucho mas antiguos[†].

Por otra parte, el uso de computadoras ha marcado el desarrollo actual en todo sentido y seguramente definirá una etapa de desarrollo, por lo que parece ser que la utilización de las computadoras en ingeniería ahora es mas importante que nunca. Afortunadamente, la capacidad de las computadoras personales actuales permite manejar la gran cantidad de datos requerida si se desea hacer, por ejemplo, una simulación en ingeniería y además hacerlo en un entorno de trabajo adecuado a la

[*] El paquete comercial *diffpack* para solucionar el sistema de ecuaciones y el paquete *plotmtv* para graficar.

[†] Para mas información ver la sección 6.3.

tarea, lo cual aumenta la productividad.

CAPÍTULO 4

OBJETIVOS

4.1. Objetivo general

Contribuir a la misión de la Universidad Industrial de Santander de formar personas con alta competencia profesional al desarrollar una interfase gráfica para una herramienta computacional para la solución de la conducción de calor basada en elementos finitos.

4.2. Objetivos específicos

Crear una interfase gráfica para usuarios como parte de una herramienta computacional para modelar problemas de transferencia de calor bidimensional usando el entorno de programación Visual C++. La interfase gráfica planteada se encargará básicamente de:

- Poder introducir geometrías básicas, como líneas, círculos, rectángulos.
- Poder combinar las geometrías básicas para formar geometrías más complejas.
- Guardar y cargar (abrir) de archivo las geometrías introducidas.
- Introducir las condiciones de frontera para problemas de conducción de calor: temperatura fija, flujo de calor fijo y flujo de calor convectivo con coeficiente de convección y temperatura ambienete fijos.
- Obtener y dibujar la región discretizada o *malla* del problema.

Crear una interfase a un proceso de generación de malla no estructurado, de manera que el programa de generación de malla lea de un archivo la frontera del dominio a discretizar, realice el proceso de enmallado, y el programa de interfase de usuario lea de un archivo la malla generada.

Permitir la inclusión de condiciones de frontera donde se especifique o bien la temperatura en unos nodos específicos, o bien la temperatura a lo largo de una frontera, el flujo de calor o una condición donde se conozca el coeficiente de convección de calor y la temperatura ambiente asociada.

Elaborar manuales de usuario que incluyan la forma de uso del programa creado.

CAPÍTULO 5

DISEÑO DE UNA INTERFAZ GRÁFICA

5.1. Sistemas operativos

Los sistemas operativos (en adelante llamados SO) son los programas que se encargan de administrar recursos y permisos en una computadora para permitirle a los usuarios utilizar los demás programas y datos en general sin tener que tratar directamente con los procesos que se llevan a cabo en el equipo utilizado. Además un SO suele incluir no solo utilidades agregadas sino también una **interfase de programación de aplicaciones** (API), diseñada para servir de ayuda a los programas que se ejecutan en el SO.

Existe una gran variedad de sistemas operativos como el Windows de Microsoft, El MacOS de Apple, El OS2 de Sun Microsystems, Inc., el UNIX desarrollado en Bell Labs o el Linux ya sea alguna de las muchas distribuciones gratuitas o de una de las cada vez menos distribuciones pagadas. Vale la pena recordar la historia de los mismos.

El SO UNIX puede considerarse como el primero usado masivamente, empezó a desarrollarse en los laboratorios Bell Labs en 1969. Antes de UNIX, cada computadora (*mainframe*) tenía su propio SO que los usuarios debían aprender y peor aun, los programas solamente funcionaban en una computadora. UNIX resolvió este problema ya que era simple y sofisticado, capaz de reciclar código y estaba escrito en lenguaje C de alto nivel. Gracias a esto se pudieron crear sistemas operativos UNIX para prácticamente cualquier computadora.

El MS-DOS fue presentado en 1984 por la empresa Microsoft y consistía en una interfase por línea de comandos sobre la cual se desarrollarían los siguientes sistemas operativos, Windows 3.1, Windows 95 y Windows 98, paralelamente, Microsoft ofrecía Windows NT para servidores de internet. Luego aparecerían los sistemas operativos Windows XP y Windows Vista.

El SO LINUX fue creado por Linus Benedict Torvalds a comienzos de los años 90s como una version académica gratuita para computadoras personales de UNIX. Ya existían versiones comerciales de UNIX para cualquier *hardware* incluyendo computadoras personales pero eran muy costosas y muy lentas. La popularidad de LINUX hizo que mucha gente contribuyera al proyecto y a los pocos años convirtió a LINUX en una réplica de UNIX con toda su funcionalidad y capaz de

utilizarse en un creciente rango de maquinas.

5.2. Interfaces de usuario

La comunicación entre una persona y una computadora, como cualquier comunicación requiere esencialmente de cuatro elementos indispensables:

- Emisor : aquel que emite la información.
- Receptor : aquel que recibe la información.
- Canal : medio a través del cual la información es transmitida.
- Código : lenguaje común a ambos interlocutores.

El código o lenguaje utilizado va a depender de la **interfaz de usuario** manejada, y puede ser mas o menos rico en significado, sencillo o adaptable. Cada programa ofrece su propia interfaz[*] incluyendo a los sistemas operativos. Los programas que funcionan bajo un sistema operativo normalmente utilizan la interfaz ofrecida por el mismo.

Las interfaces de usuario se encargan de manejar los canales que tengan relación con el usuario (teclado, ratón, pantalla, impresora, etc.) usando los respectivos códigos comunes al programa y al usuario. Existen otras interfaces que manejan la comunicación del programa con otros dispositivos, afortunadamente los sistemas operativos también pueden manejar estas interfaces por lo cual no se hace necesario que el programa se relacione directamente con los dispositivos sino con el sistema operativo. En definitiva, el programa debe comunicarse solamente con el usuario y con el sistema operativo, esta comunicación se logra a través de **eventos** que son manejados mediante **mensajes**.

5.3. Interfaz de usuario gráfica

El termino **interfaz de usuario gráfica**, traducción del inglés *Graphic User Interface*, o GUI fue acuñado como contraposición de las **interfaces de usuario por línea de comandos**, o CUI[†]. En las CUI los canales de entrada y salida de información son flujos de cadenas de caracteres (*standard input/output stream*) mientras que en las GUI los canales de entrada son ratones, *joysticks*, etc. además del teclado y la salida en pantalla no solo es texto, también incluye elementos gráficos como menús, ventanas, botones, etc. Actualmente éstas dos son las interfaces de usuario mas utilizadas.

Si bien un programa puede ofrecer la interfaz que quiera mientras sea el programa seleccionado con el foco, la interfaz de usuario de un programa va a estar limitada por la interfaz de usuario del sistema operativo, ya que éste también es un programa quien controla a los dispositivos periféricos. Antes, existían programas que tenían una interfaz de usuario (gráfica) independiente del SO y mas sofisticada, algunos programas (como los juegos) aun conservan esta practica pero como

[*] No se debe confundir con la interfaz ofrecida por una clase aunque hay casos en que ambas se relacionan, las clases se comentan en la sección 7.2 y en el glosario.

[†] *Command User Interface*, también llamadas *Command Line Interface*.

se verá, ahora que los SO han evolucionado, se prefiere una mayor integración con el escritorio (SO)[‡].

Los primeros sistemas operativos como UNIX o MS-DOS, ofrecían una interfaz de usuario por línea de comandos ya que éstas requieren la mínima infraestructura para funcionar, con solo un teclado y una pantalla monocromática como periféricos se pueden realizar todas las tareas, pero a medida que las capacidades de los equipos crecieron, cada vez mas sistemas operativos empezaron a ofrecer versiones con interfaces de usuario gráficas. Al comienzo éstas se usaron principalmente para herramientas CAD, CAE, etc.

En la actualidad casi todos los sistemas operativos ofrecen la posibilidad de usar interfaces de usuario gráficas debido a que son mas fáciles de manejar y aprender por la gran mayoría de los usuarios finales. El éxito comercial de algunos sistemas operativos como el Windows 3.1 de Microsoft Corp. se debió principalmente a su interfaz de usuario gráfica.

Incluso los sistemas operativos que se comunican con el usuario mediante línea de comandos ofrecen diferentes paquetes de utilidades para interfaces de usuario gráficas, en Linux por ejemplo[§] se utilizan GTK+, GNOME y XFCE. Las interfaces gráficas siempre requieren mas recursos de la computadora y hay ocasiones en las que son innecesarias, por ejemplo para mantener un servidor de internet.

5.4. Guías y recomendaciones de diseño

Según algunas organizaciones especializadas[¶], existen ciertas recomendaciones que pueden seguir aquellos que desarrollan nuevos programas, relacionadas con el funcionamiento, interacción con el usuario, manejo de datos, etc. Estos son algunos de los parámetros con los cuales se mide el desempeño de un programa.

5.4.1. Accesibilidad

La accesibilidad significa permitir a una persona con algún tipo de discapacidad participar de las actividades de la vida. Al utilizar un software, los colores son inútiles para distinguir información para los usuarios daltonicos, igualmente lo son los sonidos para los sordos y los usuarios con movilidad limitada no pueden usar una aplicación sin las adecuadas equivalencias en el teclado.

Una aplicación verdaderamente accesible debe permitir además, lectores de voz, lectores de pantalla y dispositivos de entrada/salida alternos.

5.4.2. Internacionalización y localización

La internacionalización significa diseñar aplicaciones que funcionen bajo diferentes entornos de lenguaje. La localización se refiere al acto de traducir los mensajes y símbolos a otro idioma. En

[‡]Es lo que en programación orientada a objetos se conoce como “re-inventar la rueda”.

[§]Hasta antes de Windows XP, Windows también se comunicaba con el usuario mediante línea de comandos ya que el verdadero sistema operativo era MS-DOS que quedaba oculto bajo la interfaz de usuario de Windows.

[¶]La mayoría de las guías y recomendaciones fueron tomadas de la referencia [3], Human Interface Guidelines 2.0.

particular se facilita la tarea de localización si se utilizan recursos como las tablas de cadenas de caracteres[‖].

5.4.3. Respuesta (*response*)

Es la rapidez con que responde el programa ante una solicitud (mensaje) del usuario.

5.4.4. Entrada del usuario

Se refiere a la comodidad con la cual debe comunicarse el usuario al introducir información al programa, esto se debe tener en cuenta especialmente si el programa va a ser utilizado muchas veces.

5.4.5. Otras recomendaciones

- Evitar los cuadros de diálogo modales, ya que con ellos el usuario solo tiene dos opciones, los botones OK y Cancel. Nunca utilizar cuadros de diálogo modales de sistema (*System Modal*) ya que bloquean todo el sistema operativo.
- No utilizar alertas sonoras en general.
- Utilizar los aceleradores estandarizados para las tareas comunes en vez de definir o traducir otras combinaciones de teclas.
- Cuando se agrupen diferentes opciones en un cuadro de diálogo, debe existir un botón aplicar para cada grupo.

Una aplicación diseñada para funcionar en un sistema operativo gráfico puede contener elementos accesorios, se recomienda utilizar principalmente dos elementos: Una entrada en el menú aplicaciones (o botón de inicio en Windows) y una entrada en el registro de documentos del sistema para que la aplicación se ejecute automáticamente cada vez que el usuario escoja un documento con una extension del tipo apropiado. Se recomienda no poner ningún icono en el escritorio.

En la entrada en el menú se debe poner el nombre de la aplicación y una pequeña *descripción funcional* de su comportamiento a menos que el nombre de la aplicación sea lo suficientemente descriptivo. Se recomienda poner un *tooltip* (comentario) para cada entrada en el menú, el cual puede ser un poco mas extenso y descriptivo.

Otros elementos pueden ser necesarios en sistemas operativos particulares, por ejemplo, las GConfKeys usadas en GNOME.

[‖] Ver capitulo 7.

CAPÍTULO 6

EL MÉTODO DE ELEMENTOS FINITOS

En Ingeniería Mecánica existen sistemas que requieren algún análisis y cuyo comportamiento se puede modelar con ecuaciones diferenciales. En algunos casos éstas ecuaciones se pueden plantear para una sola variable independiente (ecuaciones diferenciales ordinarias) pero en otros es necesario utilizar mas variables -por ejemplo, al modelar sistemas físicos que dependen de varias coordenadas espaciales o que cambian en el tiempo- en tal caso se deben plantear ecuaciones diferenciales que incluyan derivadas parciales respecto a las variables independientes (ecuaciones diferenciales parciales).

Un sistema modelado con ecuaciones diferenciales (ordinarias o parciales) y que satisface una serie de valores en la frontera de la región es también conocido como un **problema de valor inicial**[*].

Los problemas de valor inicial se pueden encontrar en las mas diversas situaciones, especialmente cuando se utilizan modelos para describir la realidad. Algunos de los ejemplos mas habituales en ingeniería son:

- La conducción de calor al interior de un sólido.
- El momento de inercia de torsión de la sección perpendicular de un perfil empotrado.
- El esfuerzo soportado y la deformación en cada punto de un sólido.
- El perfil de velocidades o la presión en un fluido.
- Análisis de vibraciones en un medio.
- Electrodinámica.
- Simulación de reacciones químicas.
- Dispositivos de estado sólido.

[*] Traducción del inglés *Initial Value Problem.* o *Boundary Conditions Problem.*.

Muchas veces el modelado de un sistema se simplifica reduciendo el número de variables independientes si se aprovechan las simetrías, ya sea porque el sistema es en verdad simétrico o porque se puede utilizar un modelo simétrico para describir un sistema sin éstas simetrías.

Existen numerosos sistemas que se comportan de acuerdo a ecuaciones diferenciales parciales (PDE[†]) y muchas veces se pueden conocer estas ecuaciones junto con ciertas condiciones iniciales pero la dificultad para resolverlas (incluso para resolver ecuaciones diferenciales ordinarias, ODE) en la práctica, dificultó el modelado con ecuaciones en el pasado.

6.1. Ecuaciones de campo

Los sistemas físicos que son descritos por PDEs respecto a las coordenadas se suelen llamar problemas de campo ya que se desea encontrar el valor de un campo escalar en una región del espacio. Los problemas de campo se pueden plantear mediante formalismos equivalentes, uno de ellos, el formalismo de Lagrange, se caracteriza por encontrar las simetrías en un sistema.

El formalismo de Lagrange permite obtener las ecuaciones que rigen la evolución de un sistema físico sin importar su complejidad partiendo de un único principio de mínima acción. La acción es la integral desde un tiempo t_1 hasta otro t_2 de una función llamada función de Lagrange o Lagrangiana, es decir que la acción es un **funcional** de la función Lagrangiana.

$$S \equiv \int_{t_1}^{t_2} L(t)dt \tag{6.1}$$

Según el principio de mínima acción, un sistema físico evoluciona desde un tiempo t_1 hasta otro t_2 cualesquiera por la sucesión de estados tal que la acción desde t_1 hasta t_2 sea mínima[‡],

$$\delta S = \delta \int_{t_1}^{t_2} L(t)dt = 0 \tag{6.2}$$

Si el sistema está descrito mediante una función ϕ de las coordenadas y del tiempo, la ecuación dinámica del sistema llamada *ecuación de movimiento de Euler-Lagrange* se obtiene fácilmente a partir del principio de mínima acción,

$$\frac{\partial L}{\partial \phi} - \frac{d}{dt}\left(\frac{\partial L}{\partial \phi_t}\right) = 0 \tag{6.3}$$

Si se tiene un problema de campo, es decir que se debe conocer el valor de un campo escalar $\phi(\mathbf{x}, t)$ en una región durante un tiempo, se puede generalizar el principio de mínima acción. Se define una variable $\mathfrak{L} = \mathfrak{L}(\mathbf{x}, t, \phi, \phi_{\mathbf{x}}, \phi_t)$ llamada densidad Lagrangiana, tal que,

$$L(t) = \int \mathfrak{L}(\mathbf{x}, t)d^3x. \tag{6.4}$$

[†]Del inglés, *Partial Differential Equations*.

[‡]La ecuación (6.2) implica que el funcional S tiene un valor extremo, no necesariamente un mínimo.

Aplicando el principio de mínima acción a la densidad Lagrangiana,

$$\delta S = \int_V \left(\frac{\partial \mathfrak{L}}{\partial \phi} \delta\phi + \sum_i \frac{\partial \mathfrak{L}}{\partial \phi_{x_i}} \delta\phi_{x_i} \right) dV = 0 \tag{6.5}$$

y teniendo en cuenta que la variación no depende de las coordenadas, $\delta\phi_{x_i} = \delta(\frac{\partial \phi}{\partial x_i}) = \frac{\partial}{\partial x_i}(\delta\phi)$, se reemplazan los términos de la sumatoria, los cuales se pueden integrar por partes,

$$\delta S = \int_V \left(\frac{\partial \mathfrak{L}}{\partial \phi} + \sum_i \frac{\partial \mathfrak{L}}{\partial \phi_{x_i}} \frac{\partial}{\partial x_i} \right) \delta\phi dV$$

$$\delta S = \int_V \left(\frac{\partial \mathfrak{L}}{\partial \phi} - \sum_i \left(\frac{\partial}{\partial x_i} \frac{\partial \mathfrak{L}}{\partial \phi_{x_i}} \right) + \sum_i \frac{\partial}{\partial x_i} \frac{\partial \mathfrak{L}}{\partial \phi_{x_i}} \right) \delta\phi dV$$

la última suma se puede transformar en una integral de superficie usando la ley de Gauss[§] ya que es la integral de volumen de un gradiente,

$$\int_V \sum_i \frac{\partial}{\partial x_i} \left(\frac{\partial \mathfrak{L}}{\partial \phi_{x_i}} \delta\phi \right) dV = \int_S \sum_i l_i \frac{\partial \mathfrak{L}}{\partial \phi_{x_i}} \delta\phi dS \tag{6.6}$$

donde l_i es el coseno entre el vector superficie dS y el eje x_i y S es la superficie que rodea a V. Combinando los resultados se obtiene:

$$\delta S = \int_V \left[\frac{\partial \mathfrak{L}}{\partial \phi} - \sum_i \frac{\partial}{\partial x_i} \left(\frac{\partial \mathfrak{L}}{\partial \phi_{x_i}} \right) \right] \delta\phi dV + \int_S \left[\sum_i l_i \frac{\partial \mathfrak{L}}{\partial \phi_{x_i}} \right] \delta\phi dS \tag{6.7}$$

Un valor estacionario de S ocurre solo si los términos entre los paréntesis cuadrados son iguales a cero, con lo cual se obtienen tanto las ecuaciones de movimiento como las condiciones de frontera:

$$\frac{\partial \mathfrak{L}}{\partial \phi} - \sum_i \frac{\partial}{\partial x_i} \left(\frac{\partial \mathfrak{L}}{\partial \phi_{x_i}} \right) = 0 \tag{6.8}$$

$$\sum_i l_i \frac{\partial \mathfrak{L}}{\partial \phi_{x_i}} = 0 \tag{6.9}$$

La densidad Lagrangiana se obtiene a partir de la homogeneidad e isotropía dependiendo del campo que se necesite estudiar. Por ejemplo, la acción que describe la conducción de calor en un cuerpo esta dada por,

$$S_C = \int_V \frac{1}{2} \left[K_{xx} \left(\frac{\partial \phi}{\partial x} \right)^2 + K_{yy} \left(\frac{\partial \phi}{\partial y} \right)^2 + K_{zz} \left(\frac{\partial \phi}{\partial z} \right)^2 - 2 \left(\dot{q} - \lambda \frac{\partial \phi}{\partial t} \right) \phi \right] dV + \int_{s1} q\phi dS + \int_{s2} \frac{h}{2} \left[\phi - 2\phi\phi_\infty \right]^2 dS \tag{6.10}$$

[§] $\int_V d^n x \partial A / \partial x_i = \int_S d^{n-1} x S A.$

donde ϕ es la temperatura al interior del cuerpo. A partir de este funcional se deducen las ecuaciones de conducción de calor, utilizando el principio de mínima acción,

$$K_{xx}\frac{\partial^2\phi}{\partial x^2} + K_{yy}\frac{\partial^2\phi}{\partial y^2} + K_{zz}\frac{\partial^2\phi}{\partial z^2} + \dot{q} = \lambda\frac{\partial\phi}{\partial t} \tag{6.11}$$

Dado que la mayoría de los sistemas que se utilizan en la práctica se pueden describir conociendo el valor de una función (el campo) y su derivada en un instante, las ecuaciones diferenciales que éste obedece (ecuaciones de campo) son de segundo orden como máximo.

6.2. Conducción de calor

La conducción de calor al interior de un sólido fue modelada por Joseph Fourier mediante la siguiente ecuación,

$$\overrightarrow{\mathbf{q}} = -K\nabla T \tag{6.12}$$

donde $\overrightarrow{q}$ es el flujo de calor sobre una superficie, T es la temperatura y K es un tensor de segundo orden (una matriz) que representa el coeficiente de conducción de calor del material del sólido. Para obtener una expresión en términos de T, se reemplaza el flujo de calor $\overrightarrow{q}$,

$$\frac{dU}{dt} = -\int_S (-K\nabla T)\cdot(\overrightarrow{\mathbf{n}}\cdot d\mathbf{S}) + \int_V \dot{q}dV$$

El primer término se puede convertir en una integral de superficie utilizando la ley de Gauss,

$$\frac{dU}{dt} = \int_V (\nabla\cdot K\nabla T + \dot{q})\, dV \tag{6.13}$$

igualando (6.13) con la ley de conservación de la energía en ausencia de trabajo,

$$\frac{dU}{dt} = \int_V \left(\rho c\frac{\partial T}{\partial t}\right) dV \tag{6.14}$$

se obtiene una ecuación sobre una integral de volumen V. Como la expresión debe ser válida para cualquier volumen V, la integral no es necesaria y se obtiene la ecuación diferencial,

$$\nabla\cdot K\nabla T + \dot{q} = \rho c\frac{\partial T}{\partial t} \tag{6.15}$$

llamada *Ecuación de difusión de calor* para un sólido incompresible. Observe que ésta es la misma ecuación (6.11) obtenida con la acción (6.10), donde ρ y c son la densidad y la conductividad térmica del material y el parámetro $\lambda = \rho c$.

Para obtener una solución aproximada de los problemas de campo como la conducción de calor, se utilizan métodos numéricos mediante programas de computadora. Los métodos numéricos en muchos casos son la única manera de obtener resultados cuantitativos, especialmente si se consideran regiones con geometrías complicadas.

6.3. Métodos numéricos de solución

Los métodos numéricos se han utilizado desde hace mas de dos mil años, una de las muestras mas antiguas de su utilización es la tableta de Babilonia *YBC 7289*, que describe el cálculo de la raíz cuadrada de dos. Los métodos numéricos se utilizan en ingeniería en las siguientes especialidades:

Calcular valores de funciones, era la aplicación principal de los métodos numéricos antes de las calculadoras y computadoras y aun hoy es muy importante.

Interpolación, extrapolación y regresión, consiste en obtener valores de funciones a partir de otros valores, incluyendo el error en los datos en el caso de la regresión.

Solución de ecuaciones y sistemas de ecuaciones, se relaciona con el problema de hallar las raíces de una ecuación algebraica.

Problema del valor propio o *Eigenvalue*, muchos problemas se pueden plantear como el sistema de ecuaciones lineales para x, $Ax = \lambda x$ donde A es una matriz y λ es una constante y cuya solución son un conjunto de valores x_i llamados vectores propios o *eigenvectores* del sistema.

Optimización, consiste en encontrar los puntos en los cuales una función es maximizada o minimizada.

Integración numérica, también es llamada cuadratura. Es mas confiable que la diferenciación numérica.

Ecuaciones diferenciales ordinarias, entre los métodos mas utilizados para aproximar una ecuación diferencial se menciona el método de Euler que es un método limitado al primer orden y el método de Runge-Kutta, popular por la aproximación de cuarto orden.

Ecuaciones diferenciales parciales, en éste caso es necesario discretizar la ecuación, esto se hace mediante una de las siguientes opciones:

Método de diferencias finitas. Es el método mas simple y consiste en aproximar la ecuación diferencial por una ecuación en diferencias incluyendo un error de truncamiento. Se destaca el método de Crank-Nicolson que es numéricamente estable y convergente. Estos métodos requieren mayor cantidad de cómputo.

Método de elementos finitos. Consiste en discretizar la geometría requerida y aproximar la solución de la ecuación, por ejemplo usando polinomios.

Método de volúmenes finitos. Al igual que en el método de elementos finitos, se discretiza la región, pero en cambio se definen volúmenes que rodean a los nodos y ecuaciones de continuidad sobre estos volúmenes (volumen de control). El método es utilizado principalmente en dinámica de fluidos computacional.

Método espectral. Bajo determinadas condiciones, un problema se puede resolver descomponiendo la función requerida en componentes o armónicos (por ejemplo mediante una transformación de Fourier) y resolviendo para cada componente.

Los métodos mas populares en ingeniería son los de elementos finitos para ecuaciones diferenciales parabólicas y los de volúmenes finitos para ecuaciones diferenciales hiperbólicas. Ambos se caracterizan por converger a la solución y por ser numéricamente estables, es decir que los errores no se acumulan. Se prefieren estos métodos ya que manejan mejor las geometrías mas complicadas.

En ambos casos la discretización se puede hacer tanto con mallas estructuradas como no estructuradas, solo las últimas se pueden utilizar con una geometría arbitraria.

6.4. Ecuaciones en elementos finitos

El método de elementos finitos es el mas apropiado para resolver numéricamente la ecuación (parabólica) de conducción de calor dentro de un sólido. Junto con un método de discretización no estructurado se puede garantizar que se obtendrá la solución de la conducción de calor en cualquier geometría.

Una función continua ***f(x)*** definida en una región se puede aproximar con otra función ***F(x)***, construida con funciones continuas definidas cada una en un trozo o **elemento** de la región. El valor de la función construida en un punto será igual al valor de la función correspondiente al elemento i ($F(\mathbf{x}) = F_i(\mathbf{x})$ para $\mathbf{x} \in e_i$) dentro del cual se encuentra el punto.

Utilizando la aproximación anterior, el método numérico de los elementos finitos obtiene las ecuaciones que debe cumplir $F(\mathbf{x})$ equivalentes a las ecuaciones de campo que debe cumplir $f(x)$. A diferencia de las ecuaciones parciales de campo, las ecuaciones para $F(x)$ van a ser ecuaciones algebraicas matriciales, ideales para resolver en una computadora.

Se pueden obtener ecuaciones para un número finito de valores porque al aproximar, se utilizan funciones con un número finito de constantes, por ejemplo polinomios. El número de constantes que tiene la función aproximada de un elemento es igual al número de **nodos** del elemento.

El número de nodos también define el **orden del elemento** el cual impone el algoritmo a utilizar para la solución. No es necesario tener elementos de orden superior para obtener una buena aproximación[¶] ya que se puede disminuir el tamaño de los elementos, a *grosso modo*, dos elementos de primer orden aproximan una función casi tan bien como uno de segundo orden en el mismo espacio.

El desarrollo siguiente explica como se hace esto con los elementos mas sencillos llamados *simplex*, que son los elementos con el menor número de nodos. Los elementos *simplex* tienen un número de nodos igual a uno mas dimensiones del elemento. Las únicas funciones que se pueden escoger en éste caso son polinomios de primer grado.

[¶]Excepto cuando se imponen condiciones especiales de continuidad de las derivadas superiores en las fronteras entre elementos.

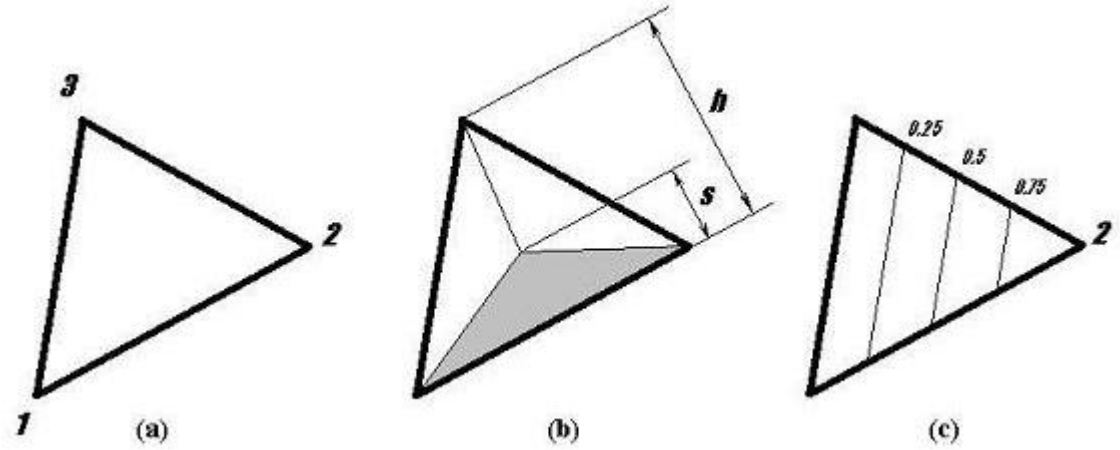

Figura 6.1: Elemento *simplex* en dos dimensiones. (a) Nodos. (b) Coordenadas de area. (c) Líneas de la función de forma N_2 constante.

Considere un elemento *simplex* en dos dimensiones[‖], el cual tendrá tres nodos. Es conveniente situar los nodos en los extremos del elemento y se va a seguir la convención de enumerarlos en sentido contrario al giro del reloj. El campo $\phi(x, y)$ lineal,

$$\phi(x, y) = \alpha_1 + \alpha_2 x + \alpha_3 y \tag{6.16}$$

debe satisfacer las condiciones en los nodos:

$$\phi(x_i, y_i) = \Phi_i, i = 1, 2, 3 \tag{6.17}$$

con lo cual se obtiene el sistema lineal de ecuaciones para las constantes α_i,

$$\begin{aligned}\Phi_1 &= \alpha_1 + \alpha_2 x_1 + \alpha_3 y_1\\ \Phi_2 &= \alpha_1 + \alpha_2 x_2 + \alpha_3 y_2\\ \Phi_3 &= \alpha_1 + \alpha_2 x_3 + \alpha_3 y_3\end{aligned}$$

que se puede escribir en forma compacta como $\Phi_i = C\alpha_i$ y cuya solución es $\alpha_i = C^{-1}\Phi_i$, explícitamente:
$\alpha_1 = \frac{1}{2A}\left[(x_2y_3 - x_3y_2)\Phi_1 + (x_3y_1 - x_1y_3)\Phi_2 + (x_1y_2 - x_2y_1)\Phi_3\right]$
$\alpha_2 = \frac{1}{2A}\left[(y_2 - y_3)\Phi_1 + (y_3 - y_1)\Phi_2 + (y_1 - y_2)\Phi_3\right]$
$\alpha_3 = \frac{1}{2A}\left[(x_3 - x_2)\Phi_1 + (x_1 - x_3)\Phi_2 + (x_2 - x_1)\Phi_3\right]$
donde A es el área del elemento y el determinante,

$$2A = \begin{vmatrix} 1 & x_1 & y_1 \\ 1 & x_2 & y_2 \\ 1 & x_3 & y_3 \end{vmatrix} \tag{6.18}$$

[‖]Las ecuaciones se pueden generalizar fácilmente para cualquier número de dimensiones, se escogen dos por concordancia con el programa. Por otra parte, el pre-procesador utilizado por el programa genera elementos *simplex*.

Es conveniente definir las **funciones de forma** N_i tales que para cualquier punto interior al elemento se cumple,

$$\phi(x,y) = \sum_i N_i \Phi_i \tag{6.19}$$

reemplazando α_i, se obtiene:

$$\begin{aligned}\phi(x,y) = \frac{1}{2A}\big[\,&((x_2y_3 - x_3y_2) + (y_2 - y_3)x + (x_3 - x_2)y)\,\Phi_1 + \\ &((x_3y_1 - x_1y_3) + (y_3 - y_1)x + (x_1 - x_3)y)\,\Phi_2 + \\ &((x_1y_2 - x_2y_1) + (y_1 - y_2)x + (x_2 - x_1)y)\,\Phi_3\big]\end{aligned} \tag{6.20}$$

comparando las anteriores dos ecuaciones, las tres funciones de forma se pueden escribir como,

$$N_i = \frac{1}{2A}\left[a_i + b_i x + c_i y\right], \tag{6.21}$$

$$a_i = (x_j y_k - x_k y_j), b_i = (y_j - y_k), c_i = (x_k - x_j).\ i,\ j,\ k\ permutaciones\ cclicas\ de\ 1,2,3$$

En general, las funciones de forma N_i se caracterizan por valer uno en el nodo i y cero en los demás nodos con valores intermedios en el resto del elemento. Las funciones de forma obtenidas para el elemento *simplex* casualmente resultan ser las mismas coordenadas de área de un punto definidas como la razón entre el área total y el área del triángulo formado por el punto y los vértices restantes como se muestra en la figura 6.1. Las coordenadas de área se caracterizan porque su suma es igual a uno en cualquier punto del triángulo y por ser constantes sobre las líneas paralelas a los bordes del elemento. Además existen ecuaciones que se van a utilizar para resolver integrales (Eisenberg y Malervin, 1973) que contengan potencias de las coordenadas de área,

$$\int_L N_1^a N_2^b dL = \frac{a!b!}{(a+b+1)!}L \tag{6.22}$$

$$\int_A N_1^a N_2^b N_3^c dA = \frac{a!b!c!}{(a+b+c+2)!}2A \tag{6.23}$$

Las funciones de forma se pueden usar para obtener el gradiente de la función escalar ya que son las únicas que dependen de las coordenadas, se obtiene que para un elemento *simplex* el gradiente dentro de un elemento es constante,

$$\frac{\partial \phi}{\partial x} = \frac{1}{2A}\left(b_1\Phi_1 + b_2\Phi_2 + b_3\Phi_3\right) \tag{6.24}$$

$$\frac{\partial \phi}{\partial y} = \frac{1}{2A}\left(c_1\Phi_1 + c_2\Phi_2 + c_3\Phi_3\right) \tag{6.25}$$

Considérese por ejemplo el problema de la conducción de calor al interior de un sólido. En la formulación funcional, el problema consiste en minimizar el funcional,

$$\chi = \int_V \frac{1}{2}\left[K_{xx}\left(\frac{\partial\phi}{\partial x}\right)^2 + K_{yy}\left(\frac{\partial\phi}{\partial y}\right)^2 + K_{zz}\left(\frac{\partial\phi}{\partial z}\right)^2 - 2Q\phi\right] dV + \int_S \left[q\phi + \frac{1}{2}h(\phi-\phi_\infty)^2\right] dS \tag{6.26}$$

donde ϕ es la temperatura, con esta formulación se incluyen las condiciones de frontera mediante la segunda integral. El método de elementos finitos consiste en reemplazar la función continua ϕ a minimizar por un conjunto $\{\Phi\}$ de valores de la función en los nodos. Para esto, definimos los arreglos,

$$\{g\} = \begin{pmatrix} \frac{\partial\phi}{\partial x} \\ \frac{\partial\phi}{\partial y} \\ \frac{\partial\phi}{\partial z} \end{pmatrix},\ D = \begin{pmatrix} K_{xx} & 0 & 0 \\ 0 & K_{yy} & 0 \\ 0 & 0 & K_{zz} \end{pmatrix} \tag{6.27}$$

con los cuales la ecuación (6.26) se puede escribir como,

$$\chi = \int_V \frac{1}{2}\left[\{g\}^T[D]\{g\} - 2\phi Q\right] dV + \int_{S_1} q\phi dS + \int_{S_2} \frac{h}{2}\left[\phi^2 - 2\phi\phi_\infty + \phi_\infty^2\right] dS \tag{6.28}$$

Teniendo en cuenta que ϕ esta definida mediante las funciones $\phi^{(e)}$ en cada elemento individual separadamente, las anteriores integrales también deben sumarse para todos los elementos,

$$\chi = \sum_{e=1}^{E} \chi^{(e)} \tag{6.29}$$

El funcional χ se minimiza cuando,

$$\frac{\partial\chi}{\partial\{\Phi\}} = \frac{\partial}{\partial\{\Phi\}}\sum_{e=1}^{E}\chi^{(e)} = \sum_{e=1}^{E}\frac{\partial}{\partial\{\Phi\}}\chi^{(e)} = 0. \tag{6.30}$$

A continuación se reemplaza la función continua a trozos por el conjunto de valores en los nodos $\{\Phi\}$, teniendo en cuenta (6.19),

$$\Phi^{(e)} = [N^{(e)}]\{\Phi\} \tag{6.31}$$

se obtiene,

$$\{g^{(e)}\} = \begin{bmatrix} \frac{\partial N_1^{(e)}}{\partial x} & \frac{\partial N_2^{(e)}}{\partial x} & \cdots & \frac{\partial N_p^{(e)}}{\partial x} \\ \frac{\partial N_1^{(e)}}{\partial y} & \frac{\partial N_2^{(e)}}{\partial y} & \cdots & \frac{\partial N_p^{(e)}}{\partial y} \\ \frac{\partial N_1^{(e)}}{\partial z} & \frac{\partial N_2^{(e)}}{\partial z} & \cdots & \frac{\partial N_p^{(e)}}{\partial z} \end{bmatrix} \begin{pmatrix} \Phi_1 \\ \Phi_2 \\ \vdots \\ \Phi_p \end{pmatrix} = [B^{(e)}]\{\Phi\} \tag{6.32}$$

reemplazando (6.31) y (6.32) en (6.28), se obtiene para cada elemento,

$$\begin{aligned}\chi^{(e)} = &\int_{V^{(e)}} \frac{1}{2}\{\Phi\}^T[B^{(e)}]^T[D^{(e)}][B^{(e)}]\{\Phi\}dV \\ &- \int_{V^{(e)}} Q[N^{(e)}]\{\Phi\}dV + \int_{S_1^{(e)}} q[N^{(e)}]\{\Phi\}dS \\ &+ \int_{S_2^{(e)}} \frac{h}{2}\{\Phi\}^T[N^{(e)}]^T[N^{(e)}]\{\Phi\}dS \\ &- \int_{S_2^{(e)}} h\phi_\infty[N^{(e)}]\{\Phi\}dS + \int_{S_2^{(e)}} \frac{h}{2}\phi_\infty^2 dS\end{aligned} \tag{6.33}$$

Los valores de los coeficientes Q, q, ϕ_∞ y h son conocidos y se conservan dentro de la integral ya que pueden variar dentro del elemento. Derivando (6.33) respecto a $\{\Phi\}$ se obtiene,

$$\frac{\partial \chi^{(e)}}{\partial\{\Phi\}} = [k^{(e)}]\{\Phi\} + f^{(e)} \tag{6.34}$$

donde,

$$[k^{(e)}] = \int_{V^{(e)}} [B^{(e)}]^T[D^{(e)}][B^{(e)}]dV + \int_{S_2^{(e)}} h[N^{(e)}]^T[N^{(e)}]dS \tag{6.35}$$

$$\{f^{(e)}\} = -\int_{V^{(e)}} Q[N^{(e)}]^T dV + \int_{S_1^{(e)}} q[N^{(e)}]^T dS - \int_{S_2^{(e)}} h\phi_\infty[N^{(e)}]^T dS \tag{6.36}$$

el sistema final de ecuaciones algebraicas se obtiene reemplazando en (6.30),

$$\frac{\partial \chi}{\partial\{\Phi\}} = \sum_{e=1}^{E} \left([k^{(e)}]\{\Phi\} + f^{(e)}\right) = 0 \tag{6.37}$$

el cual sustituye la ecuación diferencial de conducción de calor (ecuación de Fourier).

Parte II

Estrategias de solución

CAPÍTULO 7

ESTRUCTURA INTERNA

El objetivo del trabajo es presentar una forma cómoda de intercambiar información entre una persona y una computadora, lo cual se logra mediante un **programa** con interfase de usuario gráfica (GUI). El programa presentado recibe las ordenes del usuario y genera información que muestra en la pantalla y que guarda en diferentes archivos.

Con el programa, el usuario puede dibujar y editar una serie de figuras geométricas y definir una serie de condiciones de frontera con las cuales se llama a las funciones ofrecidas para que el programa genere y dibuje una región discretizada o *malla*. Todo lo anterior se utiliza para atacar los problemas de conducción de calor bidimensionales en ingeniería con métodos computacionales de elementos finitos (ver capítulo anterior).

A continuación se presenta el código que hace esto posible. En este capítulo se muestran las características generales y las herramientas de dibujo y en el capítulo siguiente se presentan las herramientas para generar y dibujar la malla. En la siguiente sección (7.1) se muestran las principales características del programa presentado al usuario (ver figura 7.1).

Para entender la estructura general del programa es necesario conocer la Programacion Orientada a Objetos, ya que el código se encuentra organizado en unidades fundamentales (objetos o **clases**) que cumplen labores específicas. Las clases se explican brevemente en la sección 7.2. Habiendo definido lo que es una clase y como se relacionan entre si las clases, en la sección 7.3 se presenta la estructura interna del programa (ver figura 7.3) y como se relaciona esta con el usuario.

A partir de la sección 7.4, se muestran los detalles técnicos del código escrito. Como el programa está creado a partir de ciertas clases disponibles, las clases de MFC y de ReportCreator[*], lo primero que se debe hacer es modificar estas clases para que tengan las caracteristicas requeridas (ver secciones 7.5 y 7.6). Luego se pueden crear nuevas clases similares a las existentes como se muestra en la sección 7.7.

Cada una de las cuatro secciónes siguientes, explica una función que se utiliza para modificar

[*] Las clases de ReportCreator, disponibles en www.codeproject.com a su vez estan basadas en las clases MFC de Microsoft VisualC++.

los objetos existentes. La sección 7.8 para dibujar figuras, 7.9 para editar, 7.10 para hallar los puntos de intersección y 7.11 para ver la barra de estado. Muchas de éstas funciones son llamadas directamente por el usuario, utilizando la clase View como es acostumbrado (ver 7.5.1), la organización (ubicación) de las funciones dentro de las clases se explica en las secciones de las clases (7.6.1 y 7.6.2).

En la sección 7.12 se muestra como se definen las condiciones de frontera y como se crean las clases que contienen tal información, éstas clases en particular, fueron creadas aisladamente sin referencia a ninguna otra clase existente. Finalmente la sección 7.13 habla de la forma en que se guardan lor archivos (*.efn) en las clases MDI de MFC.

7.1. Generalidades

Para describir el programa en términos de las partes que lo constituyen es necesario conocer ciertas abstracciones de la programación orientada a objetos, principalmente que son las clases (ver siguiente sección).

Las clases son los datos que internamente maneja el programa, en un programa orientado a objetos todo el código existente se encuentra en las clases[†]. En una clase los datos pueden ser o *variables*, que son otras clases o *funciones*, que son algoritmos que modifican variables y devuelven un valor al terminar.

El programa es una aplicación ejecutable en Windows llamada `elfin.exe`. Al iniciar el programa se abre una ventana en el escritorio con menús, botones y un area de dibujo (Ver figura 7.1). El programa permanece abierto, manejando los **eventos** generados por el usuario (como mover el ratón, utilizar el menú, seleccionar un objeto u oprimir una tecla) hasta que el usuario decida cerrar la ventana. Esto es lo que se denomina la **interfase grafica** presentada al usuario por el programa.

Para ofrecer dicha interfase gráfica y mantener abierto el programa mediante un bucle (ciclo) de mensajes, se utilizan las clases de Microsoft (MFC, ver glosario). Específicamente se usa una *Arquitectura de Documento Múltiple (MDI)*[‡]. Se puede generar automaticamente un programa con arquitectura MDI utilizando *AppWizard* en *Microsoft Visual Studio*. Al iniciar un nuevo proyecto en *Visual C++*, aparece el cuadro de diálogo de *AppWizard* (Ver figura 7.2, en el cual se escoge el tipo de proyecto a iniciar para que se generen los archivos necesarios funcionando correctamente. Para obtener un programa ejecutable con arquitectura MDI se escoge *MFC AppWizard (exe)* de la lista presentada.

Después de recorrer los cuadros de diálogo presentados, se muestra una lista con los archivos y las clases que se van a crear. Al aceptar, *AppWizard* genera el programa y se tiene un conjunto de clases relacionadas entre si las cuales se encargan de manejar los diferentes aspectos de la arquitectura MDI[§], ver figura 7.1. A grandes rasgos, cuando se inicia el programa se crea una clase

[†] se puede considerar clase como sinónimo de objeto, para mas información ver el glosario.

[‡] la arquitectura de documento multiple se refiere a la capacidad del programa de mantener abiertos diferentes documentos con un solo programa (ver glosario).

[§] la arquitectura MDI se explica en los textos de Microsoft Visual C++ así como en la documentación de otras clases

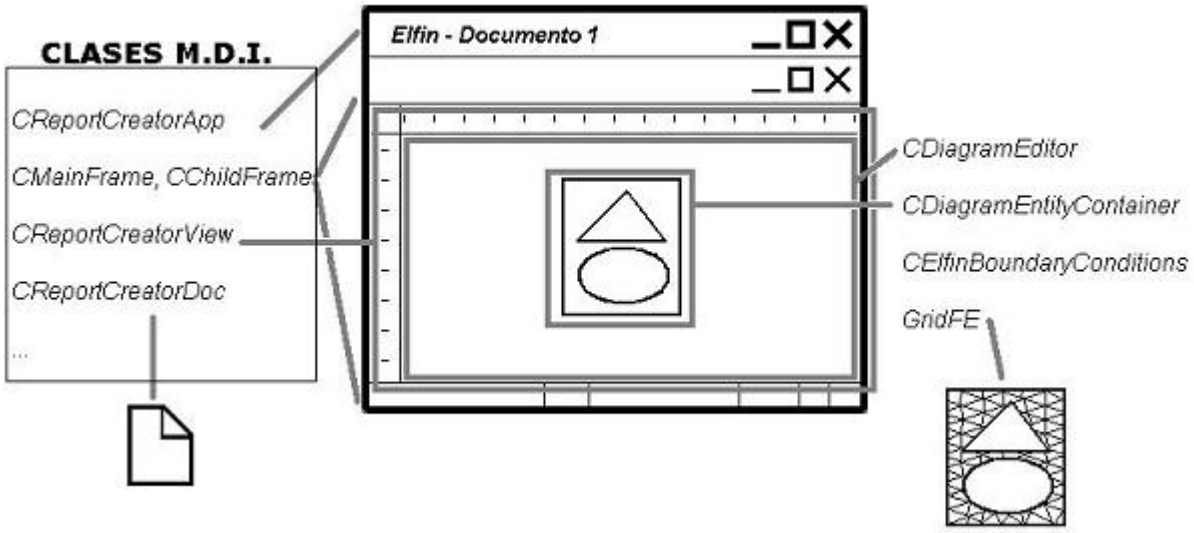

Figura 7.1: Ventana del programa *elfin.exe*

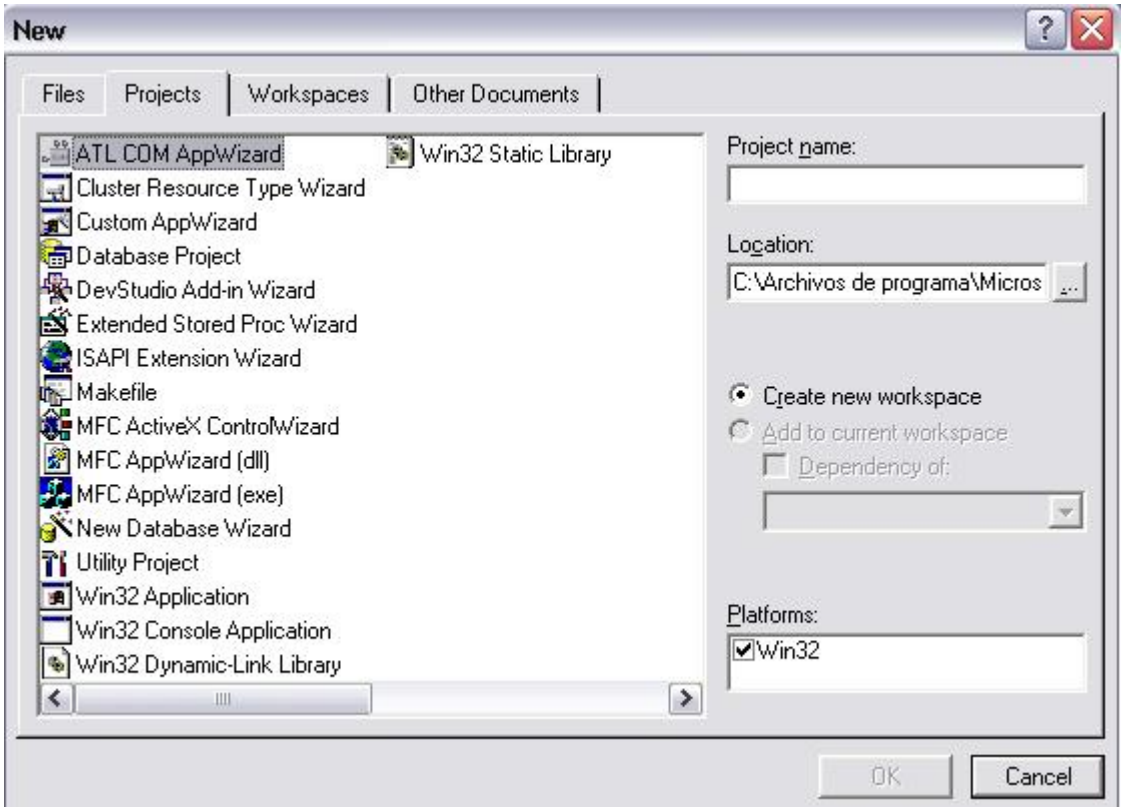

Figura 7.2: Cuadro de diálogo de *AppWizard*.

App (*CReportCreatorApp*) que maneja todo el programa, en un programa MDI tambien se crean las clases *CMainFrame* y *CChildFrame* que se encargan de manejar los bordes de la ventana. Por cada vez que el usuario decide abrir o crear un proyecto (archivo), la clase *App* crea dos nuevas clases

MDI, ver por ejemplo [4].

de tipos *C...View* y *C...Doc* que manejan la ventana principal y el documento (archivo guardado) respectivamente. Como es practica habitual, se introduce todo el codigo dentro de la clase de vista (*CReportCreatorView*).

Cuando el usuario maneja el programa, tiene a su disposición una serie de funciones que se pueden acceder desde el menú, los aceleradores del teclado o los botones de la barra de herramientas. Además los eventos (tales como movimientos del ratón, etc.) generados por el usuario tienen cada uno una función que lo administra. Normalmente la mayoría de las funciones se encuentra dentro de la clase View.

El programa antes que todo es una aplicación gráfica con la cual un usuario puede crear y editar figuras geométricas, con estas figuras se obtiene la geometría que define la región requerida así que el programa es también una sencilla herramienta de diseño asistido por computadora. Entre las utilidades más indispensables para este tipo de herramienta, se mencionan:

- **Creación de líneas sencillas.** Con el programa se pueden crear líneas rectas y arcos de elipses sin giro. Esto se hace dibujando con el ratón sobre el área de edición.
- **Creación de areas sencillas.** Como se obtienen las líneas, se pueden obtener areas como rectángulos y elipses.
- **Edición de líneas sencillas para crear polilineas.** Además de poder editar gráficamente el tamaño y posición de las figuras sencillas, también es posible obtener líneas compuestas o "poli-líneas". El usuario utiliza las poli-líneas indirectamente al pasar a la etapa de enmallado (ver capitulo 8).
- **Edición de areas y líneas para crear areas "vectoriales".** La poli-línea es en realidad un arreglo de figuras, la cual posee métodos para insertar éstas figuras correctamente. Si después de insertar una figura, la poli-línea queda cerrada, inmediatamente se define un area vectorial.

La función *Divide* se utiliza para obtener cualquier región cerrada, en vez de definir sofisticadas funciones para obtener las posibles intersecciones entre areas, se deja la responsabilidad al usuario para que construya una región a partir segmentos unidos por sus extremos. No obstante la geometría ingresada se verifica posteriormente, cuando el usuario pasa a la etapa de discretización y enmallado. Después de que el usuario ha ingresado una serie de figuras unidas por sus extremos que definen una región cerrada, el programa realiza su labor especifica mediante los siguientes pasos consecutivos:

- El usuario ingresa las condiciones de frontera mediante el cuadro de diálogo correspondiente (Ver sección 7.12).
- Cuando el usuario ha dibujado una región y pasa a la etapa de discretización o enmallado (Shift+2), se construyen todos los contornos posibles a partir de los segmentos existentes revisando a la vez que la geometría sea correcta (Ver sección 8.2.1).
- El usuario define el numero de puntos o la distancia entre ellos (Ver sección 8.2.2) y las condiciones de frontera presentes sobre los puntos para todos los segmentos de los contornos cerrados. Esto es necesario porque en los archivos de entrada del pre-procesador se define la geometría mediante una serie de puntos.

- El usuario debe abrir el cuadro de diálogo *Región* e introducir la información requerida para escribir los archivos, como nombre y numero de elementos de la región e información de los agujeros o refinamientos presentes. Al oprimir el botón *OK* se escriben los archivos de entrada (*geom.i* y *prepro.part*), se llama al pre-procesador que genera el archivo *grid.grid* y se lee el archivo *grid.grid* y se dibuja la malla (Ver sección 8.2.3).
- Cuando el usuario pasa a la etapa de solución (Shift+3), el programa de solución o *solver* lee el archivo *grid.grid* y obtiene la solución de las ecuaciones algebraicas de la malla (Ver capitulo 8.3).

7.2. Programación Orientada a Objetos

El desarrollo de programas ha pasado de ser una curiosidad a ser una de las industrias mas importantes en medio siglo. Esto solo fue posible cuando se evidenció la necesidad de utilizar el código desarrollado anteriormente en vez de caer en el síndrome de *re-inventar la rueda* y se establecieron mecanismos eficientes para permitir a los programadores manejar el código importado. El último y mas importante de estos mecanismos es el paradigma de la **programación orientada a objetos** (*Object Oriented Programming*, OOP) el cual no ha sido destronado durante más de dos décadas[¶]. La OOP define las reglas que se deben seguir al utilizar código en un nuevo programa y al escribir código para que sea re-utilizado por otros, así que sera necesaria al trabajar con el código escrito. La programación orientada a objetos le permite a una persona desarrollar programas de cientos de miles de líneas de código en lenguaje de alto nivel gracias a la re-utilización de código anterior. La re-utilización de código tiene la doble ventaja de la modularidad y de evitar los errores ocultos basándose en la confianza en el código reutilizado, el cual en lo posible, debe no modificarse.

La sola definición de la programación orientada a objetos es muy abstracta y sujeta a diferentes interpretaciones[‖], basta con tener en cuenta que el código escrito con éste paradigma esta organizado en unidades fundamentales llamadas **objetos** (que pueden ser **clases** o **estructuras**). Los objetos contienen variables y funciones llamadas **miembros** del objeto, que solo se pueden acceder si se declaran como *public*, los miembros que se declaran como *protected* o *private* solo se pueden acceder por otros miembros de la clase o sus clases descendientes, lo cual se conoce como encapsulado de la información. La programación orientada a objetos se fundamenta en dos pilares, la **herencia** y el **polimorfismo**. Con el primero se pueden definir clases hijas o **descendientes** de otras clases padre o **base** creando una vasta genealogía de objetos con propiedades comunes, llamada **jerarquía**[**]. Con el segundo se pueden escoger funciones en tiempo de ejecución utilizando funciones declaradas *virtual* en la clase base y volviéndolas a definir en las clases descendientes.

Por ejemplo, la clase Figura contiene variables y funciones para manejar información de una figura geométrica como posición, tamaño, nombre, etc., se pueden definir clases herederas de la clase Figura que contengan funciones (de dibujo, calculo de intersecciones, etc.) para manejar específicamente un tipo de figura, en particular las clases Linea, Arco, Rectángulo, Elipse y Polilinea son clases hijas o herederas de la clase Figura, en las clases hijas se re-definen las funciones Draw(...)

[¶] El paradigma anterior era la programación estructurada y se specula que el siguiente estará relacionado con el desarrollo sincronizado en red, además existen iniciativas como el código libre que están modificando la forma de trabajo y la industria.

[‖] se recomienda consultar la referencia [4] y el glosario.

[**] Véase las figuras 7.7 y 7.6.

que se consideran funciones polimorfas (virtuales).

7.3. Organización de los objetos

A continuación se presenta un esquema general de la estructura interna del programa (figura 7.3), en donde cada rectángulo representa una clase. Cada clase contiene a su vez variables o funciones clasificadas en tres (*public*, *protected* y *private*). cada variable (o función) se muestra como el tipo de clase y el nombre de la clase. Por ejemplo, La clase CReportEditorView tiene una variable de tipo *CDiagramEditor* llamada *m_editor*.

Según el diagrama, el programa esta compuesto por las clases MDI como es practica habitual. La clase de vista, (la clase CReportCreatorView) contiene una serie de funciones (publicas), las funciones afx_msg, que manejan los mensajes generados por el usuario. A diferencia de los programas MDI mas sencillos, el código no se escribe directamente sobre esta clase, en vez de esto, la clase tiene un miembro llamado m_editor que hace las veces de clase de vista. Este se encarga de manejar la ventana de dibujo y en general, todas las **utilidades** disponibles.

Entonces, todo el código se va a incluir en la clase MDI de vista (CReportCreatorView) a traves de un único objeto de tipo CDiagramEditor, que representa la ventana de dibujo (sin las reglas de referencia, ver figura 7.1). Dentro de la clase DiagramEditor hay tres contenedores de tipo CDiagramEntityContainer, cada contenedor es un conjunto ordenado de figuras y contiene funciones para agregar, quitar u organizar figuras. Las figuras pertenecen a la clase CDiagramEntity y a sus clases hijas. El Editor (la clase DiagramEditor) se encarga de todas las tareas y si es necesario llama a las figuras geométricas llamando a la función GetAt de CDiagramEntityContainer.

Según el tipo de tarea solicitada se utilizan las funciones del editor (la clase CDiagramEditor), el cual puede modificar las figuras que se encuentran en uno de los tres contenedores, escribir los archivos de entrada del pre-procesador o llamar al pre-procesador o al *solver*.

En resumen, un mensaje es generado por el usuario y manejado por alguna función On...(...) en la clase de vista, esta se encarga de llamar a la función correspondiente en la clase editor el cual administra un conjunto de figuras guardadas en los contenedores CDiagramEntityContainer. La descripción detallada de como se logra esto es el tema de los siguientes capítulos.

7.4. Características

Las caracteristicas mas importantes del programa son los metodos que él utiliza para relacionarse con el usuario (GUI), las herramientas disponibles (programas y algoritmos) y los archivos utilizados. El programa (elfin.exe) maneja múltiples archivos, los cuales se requieren por los diferentes programas utilizados internamente como se muestra en la figura 7.4,

Una vez que el usuario ingresa la geometría y las condiciones de frontera utilizando el ratón y el teclado, cuando se solicita crear la malla, el programa escribe los archivos *geom.i* y *prepro.part*

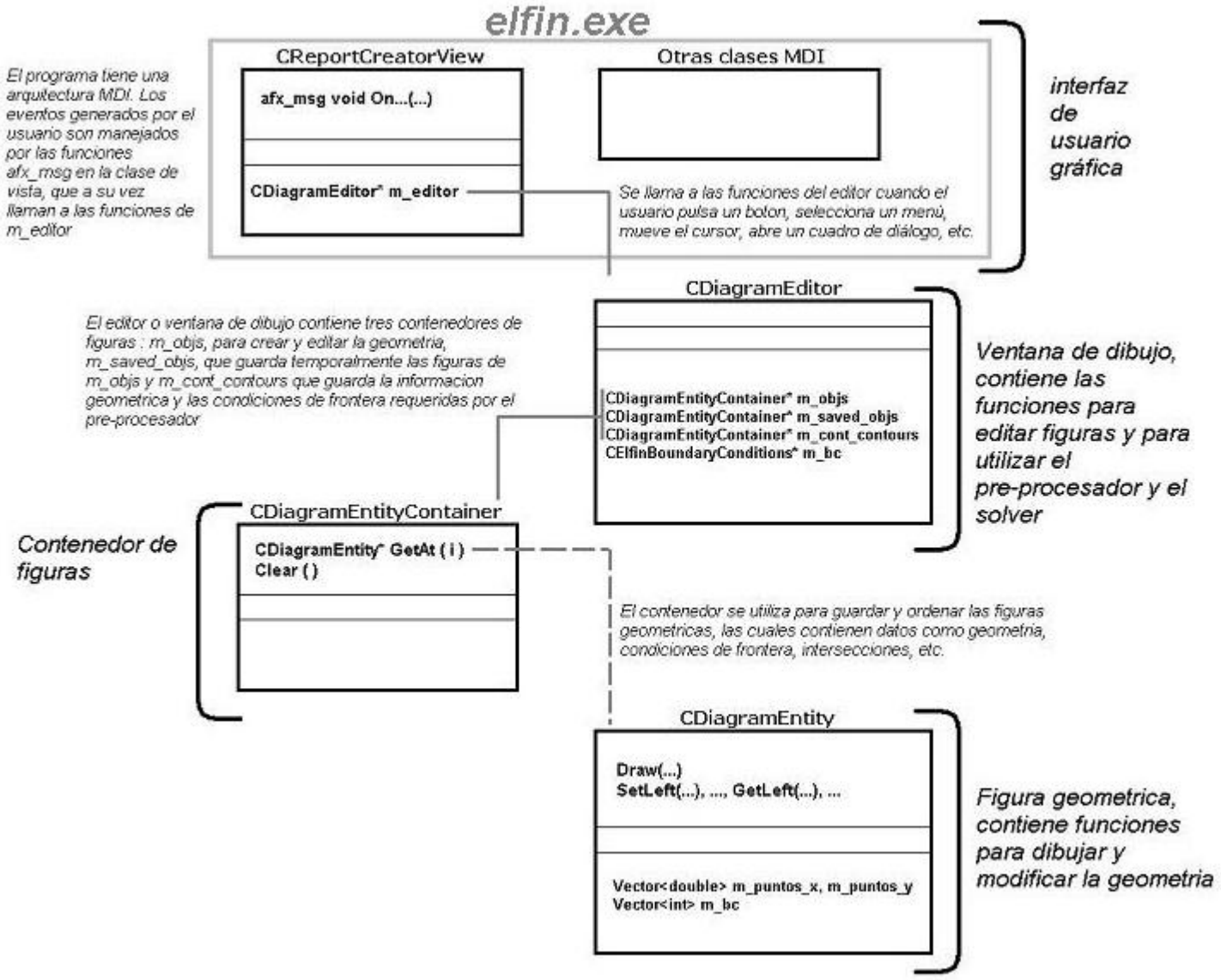

Figura 7.3: Estructura general del programa

que contienen la geometría y las condiciones de frontera del problema planteado.

El **pre-procesador** es el programa que genera la región discretizada y escribe el archivo correspondiente. Utilizando los archivos *geom.i* y *prepro.part*, el pre-procesador escribe el archivo *grid.grid* que contiene la región discretizada o malla. El programa lee el archivo *grid.grid* para dibujar la malla dentro de la ventana. Igualmente, si se tiene un **solver** o programa para obtener la solución a partir de la malla, el *solver* debe leer el archivo *grid.grid* y generar el archivo *grid.out*,

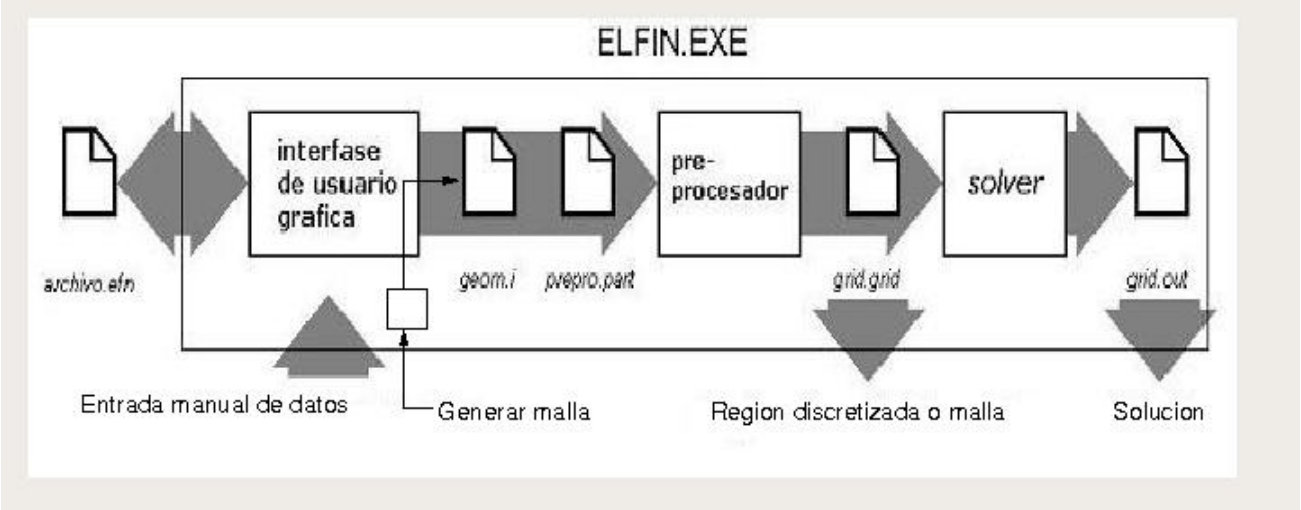

Figura 7.4: Archivos utilizados

el cual contiene la solución numérica de las ecuaciones, es decir, el valor de la temperatura en los nodos de la malla.

Es importante tener en cuenta que toda esta secuencia queda oculta al usuario en un único programa (elfin.exe). El usuario puede abrir o guardar diferentes archivos con la extensión *.efn que son independientes del formato del pre-procesador o del solver[††] y que se asocian con el programa en el sistema operativo[‡‡]. Entre otras características generales se mencionan:

- Se ejecuta bajo el sistema operativo Windows y ha sido desarrollado y probado bajo las versiones Windows 2000 y Windows XP, ésta elección es la más apropiada considerando que el programa esta hecho principalmente para los estudiantes de la universidad.
- Utiliza una arquitectura MDI. Fue escrito en Visual C++ 6.0.
- Utiliza clases de *Report Creator* disponible en el portal Code Project (www.codeproject.com) algunas de ellas descendientes de objetos de MFC.
- Utiliza los algoritmos de calculo de la recopilación *Graphics Gems* (www.graphicsgems.org) para calcular las intersecciones entre líneas rectas y entre una línea y una elipse para aumentar la rapidez de ejecución del programa.
- Utiliza la clase Vector (un contenedor) del proyecto de grado titulado, *Creación de objetos en C++ para la solución de problemas de ingeniería*.
- Utiliza un pre-procesador para la discretización de la geometría llamado PreproGeomPack desarrollado por Goe Barri de Numerical Objects de Noruega (www.nobjects.org).

Antes de continuar, se establecerá la notación utilizada para el código en C++ presentado en este capitulo:

[††]si el usuario lo requiere, puede también copiar los archivos *geom.i*, *prepro.part*, *grid.grid* o *grid.out* del directorio en donde se encuentra el programa.

[‡‡]es decir, los archivos que se pueden *abrir con* el programa.

- El código esta escrito con un tipo de fuente diferente (`Verbatim`).
- El código presentado se clasifica en dos grupos, el que fue escrito anteriormente se muestra con fondo sombreado (por ejemplo, `CDiagramEditor()`) mientras que el código desarrollado en éste trabajo se muestra sin color de fondo (por ejemplo, `CReportEntityArc()`)[§§].
- No se muestran los comentarios del código desarrollado ni muchos de los espacios en blanco.
- El código desarrollado se presenta en conjuntos de funciones o **utilidades**. El orden de presentación depende del uso de la función. Se explica la definición de las funciones según sea necesario.

La interacción del programa con el usuario, el manejo de los archivos, los mensajes del programa con el sistema operativo e incluso la organización interna de sus datos y rutinas dependen de ciertas clases que son creadas automáticamente al crear el proyecto, estas clases funcionan sincronizadas bajo lo que se suele llamar una arquitectura de documento, la biblioteca MFC ofrece aplicaciones con las arquitecturas de documentos multiples (MDI) y de documento único (SDI) además de una arquitectura basada en diálogos.

El proyecto utiliza una arquitectura MDI ya que es la más desarrollada. Las clases más importantes generadas automáticamente son:

- class CReportCreatorApp : public CWinApp
- class CReportCreatorDoc : public CDocument
- class CReportCreatorView : public CView
- class CChildFrame : public CMDIChildWnd
- class CMainFrame : public CMDIFrameWnd

Estas clases son modificadas por sus usuarios, normalmente CWinApp para registrar la aplicación, CDocument para abrir y guardar un documento y CView para agregar el resto del programa, se incluyen recursos gráficos que se editan junto a otras clases. Los autores de ReportCreator que generaron las anteriores clases, las modificaron y crearon muchas otras que se utilizan a través de CReportCreatorView, se toman estas clases ya modificadas y se agrega el código que se muestra a continuación. Para ver estas clases recién creadas se puede crear un nuevo proyecto con arquitectura MDI con AppWizard, si hacen falta funciones se pueden sobrecargar usando ClassWizard (Ver Figura 7.5.

7.5. Modificando las clases MDI

El código insertado en las clases MDI se muestra a continuación comenzando con las clases menos modificadas. Primero se debe agregar una entrada en el registro del sistema. En la función CReportCreatorApp::InitInstance() se debe cambiar la linea 106 en el archivo CReportCreatorApp.cpp por

[§§]También se reemplaza código sin importancia por tres puntos(...).

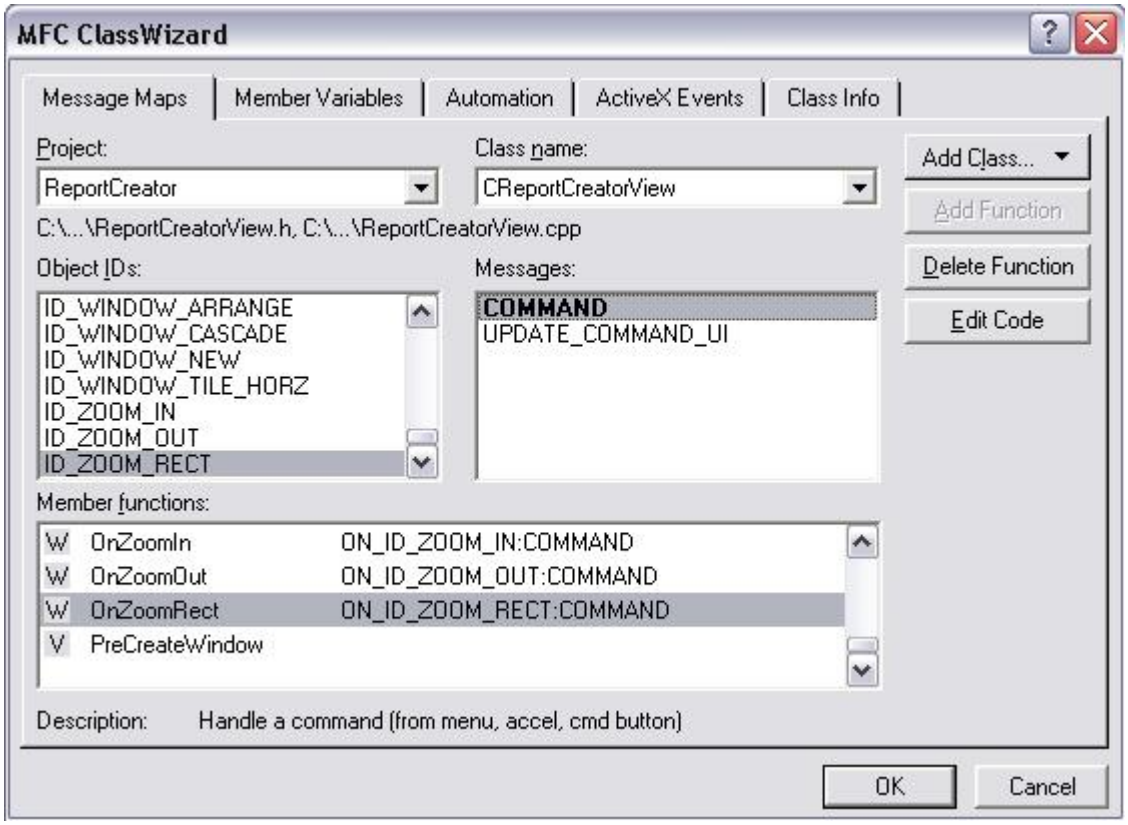

Figura 7.5: Cuadro de diálogo de *ClassWizard* mostrando los mensajes manejados.

```
    SetRegistryKey( _T( "ElFin" ));
```

Continuando con la clase CChildFrame, vamos a sobrecargar la función publica ActivateFrame declarandola en ChildFrm.h,

```
virtual void ActivateFrame( int nCmdShow );
```

y definiéndola en ChildFrm.cpp,

```
void CChildFrame::ActivateFrame( int nCmdShow ){
    nCmdShow = SW_SHOWMAXIMIZED;
    CMDIChildWnd::ActivateFrame( nCmdShow );
}
```

Esta función es llamada desde CWinAp::OnFileNew y es la encargada de fijar el tamaño de la ventana de cada documento al ser abierto, por defecto, para mostrar que se esta utilizando una aplicación de documento multiple esta ventana aparece reducida, pero el programa tiene mejor aspecto si se muestra ampliada agregando la primera linea, igualmente existe una función de CMainFrame para fijar el tamaño de la ventana de la aplicación respecto al escritorio pero no se va a modificar. La clase del cuadro de diálogo *About* no se modifica, pero si su recurso asociado.

7.5.1. La clase de vista

Se muestra una parte del archivo ReportCreatorView.h que muestra las variables privadas y las funciones generadas mediante *message map* para los eventos manejados:

```
    afx_msg void OnButtonAddArc();
    afx_msg void OnButtonAddPoly();
    afx_msg void OnButtonTrim();
    afx_msg void OnButtonSegment();
    afx_msg void OnButtonDivide();
    afx_msg void OnButtonIntersections();
    afx_msg void OnGroup();
    afx_msg void OnUngroup();
    afx_msg void OnUndo();
    afx_msg void OnUpdateGroup(CCmdUI* pCmdUI);
    afx_msg void OnUpdateUngroup(CCmdUI* pCmdUI);
    afx_msg void OnUpdateButtonDivide(CCmdUI* pCmdUI);
    afx_msg void OnUpdateButtonSegment(CCmdUI* pCmdUI);
    afx_msg void OnUpdateButtonTrim(CCmdUI* pCmdUI);
    afx_msg void OnUpdateButtonProperties(CCmdUI* pCmdUI);
    afx_msg void OnSelectAll();
    afx_msg void OnSelectNext();
    afx_msg void OnSelectPrevious();
    afx_msg void OnUpdateSelectNext(CCmdUI* pCmdUI);
    afx_msg void OnUpdateSelectPrevious(CCmdUI* pCmdUI);
    afx_msg void OnOnUndo();
    afx_msg void OnZoomRect();
    afx_msg void OnButtonAddHeat();
    afx_msg void OnButtonTemperature();
    afx_msg void OnButtonAddConvection();
    afx_msg void OnContour();
    afx_msg void OnShowConditions();
    afx_msg void OnObtainContours();
    afx_msg void OnEtapeEdit();
    afx_msg void OnEtapeMesh();
    afx_msg void OnEtapeSolve();
    afx_msg void OnMostrarElemento();
    afx_msg void OnSaveIn();

private:
 CReportEditor m_editor;
 CHorzRuler m_horzRuler;
 CVertRuler m_vertRuler;
 CCornerBox m_cornerBox;
 int m_screenResolutionX;
```

La variable mas importante es m_editor, ésta y las tres siguientes son clases hijas de CWnd y representan el cuadro de dibujo, una regla horizontal, una regla vertical y un cuadro ubicado en la esquina superior izquierda que al ser pulsado muestra un cuadro de diálogo para escoger las unidades de distancia utilizadas.

A pesar de que la clase CReportCreatorView tiene muchas funciones nuevas, en su mayoría generadas para manejar eventos, no se mostrará la implementación de todas las funciones. Debido a que la mayoría de las funciones generadas simplemente re-dirigen el manejo del evento a la variable m_editor como se muestra en el extracto del archivo ReportCreatorView.cpp siguiente:

```
void CReportCreatorView::OnSelectAll() {
    m_editor.SelectAll();
}
```

Sin embargo hay otras funciones que es necesario explicar:

```
void CReportCreatorView::OnButtonAddArc() {
    m_editor.StartDrawingObject( new CReportEntityArc );
}
```

Las funciones OnButtonDrawArc y OnButtonDrawPolyline al igual que otras similares agregan un nuevo objeto del tipo apropiado al contenedor, así que se deben incluir los archivos,

```
#include "..\\ReportEntityArc.h"

#include "..\\ReportEntityPolyline.h"
```

Las anteriores funciones se incluyen junto con un botón en la barra de herramientas. Otras funciones muy similares entre si son las que manejan los casos en que se habilita la pulsación sobre un botón, por ejemplo para los botones que permanecen hundidos, una de las funciones escritas es la siguiente,

```
void CReportCreatorView::OnUpdateGroup(CCmdUI* pCmdUI){
    pCmdUI->Enable( ( m_editor.GetSelectCount() > 1 ) );
}
```

con el cual se habilita un botón para agrupar objetos siempre que hayan al menos dos seleccionados. La función OnUpdateUngroup es idéntica, OnUpdateButtonDivide reemplaza el 1 por 0. Las funciones OnUpdateButtonSegment, OnUpdateButtonTrim, OnUpdateButtonProperties, OnUpdateSelectNext y OnUpdateSelectPrevious se activan cuando hay una sola figura seleccionada.

Cuando se selecciona alguna de las etapas, las tres funciones para los botones llaman a una misma función en m_editor con diferente argumento, el cual es una constante definida en CDiagramEditor.h, por ejemplo la función OnEtapeEdit,

```
void CReportCreatorView::OnEtapeEdit(){
    m_editor.Etape( ETAPE_EDIT );
}
```

igualmente se llama a la función Etape con los valores ETAPE_MESH y ETAPE_SOLVE.

Por último se utiliza una función que genera y guarda los archivos de entrada del preprocesador e inmediatamente llama al mismo para obtener la malla partir de estos archivos,

```
void CReportCreatorView::OnSaveIn() {
    m_editor.SaveInputFile();
    int retcod = system( "gengrid.exe geom.i prepro.part grid.grid" );
    GridFE grid;
    ifstream fcin("grid.grid");
    grid.scan(fcin);
}
```

En la primera linea se llama a la función SaveInputFile, la cual crea los archivos geom.i y prepro.part, a continuación se llama al programa gengrid.exe con estos archivos, el programa crea el archivo grid.grid que se lee en la penúltima linea para construir la malla.

7.6. Modificando las clases de *ReportCreator*

Normalmente una utilidad contiene código en las clases CDiagramEditor y CDiagramEntity[¶¶], además se pueden necesitar nuevas clases, modificar la clase CDiagramEntityContainer y la clase CReportCreatorView para que la utilidad se encuentre disponible para el usuario. Se observará que las siguientes utilidades son similares a los bloques agregados a la clase DiagramEditor.

7.6.1. Figuras

Las figuras utilizadas pertenecen a la jerarquía de la clase CDiagramEntity la cual a su vez desciende de la clase CObject, esta clase base debe ser modificada para añadir funciones para tareas comunes a todas las clases o para definir funciones virtuales que serán utilizadas en las clases descendientes (Ver figura 7.6).

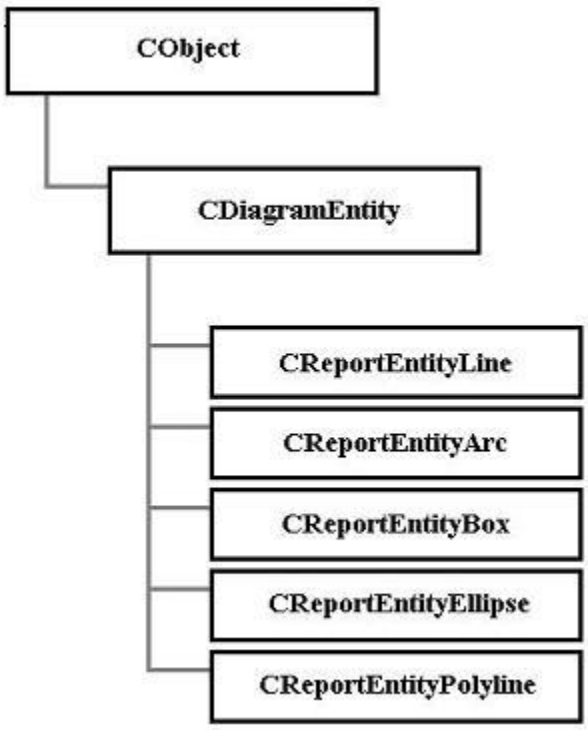

Figura 7.6: La genealogía de figuras

La clase base, CDiagramEntity, se modificó de la siguiente forma:

```
#include <vector.h>
#include "NativeFunctions.h"

...

#define DEHT_ENDMARKER_BE 10
#define DEHT_ENDMARKER_EN 11

...
```

[¶¶]que se presentan en las dos siguientes secciones.

```
#define DETY_ARC            4
#define DETY_POLINE         5

#define DIR_DIRECT          0
#define DIR_INVERT          1

#define REGION_EXTERIOR     0
#define REGION_HOLE         1
#define REGION_HOLE_INTER   2

#define REGION_DIR_DIR      0
#define REGION_DIR_INV      1

#ifndef pi
#define pi      3.14159265358979323846264338327 95
#endif

...

friend class CReportEntityPolyline;

...

public:
////Preprocessor data
    int             m_bc;
    double          m_points;
    CString         m_points_units;
    bool            m_region_construct;
    int             m_region_dir;
    int             m_region;
    int             m_region_inner;//hook
    double          m_region_inner_x;
    double          m_region_inner_y;
    int             m_region_outter;
    double          m_region_outter_x;
    double          m_region_outter_y;
    bool            m_in_a_contour;

////Division points
    Vector<double>  m_div_x;
    Vector<double>  m_div_y;
    bool            n_div_vis;
    virtual void    DrawDivisionMarkers( CDC* dc, double zoom );
    virtual void    DrawDivisionMarkers( CDC* dc, CRect rect );
    virtual void    DrawSense( CDC* dc, double zoom );
    virtual void    DrawSense( CDC* dc, CRect rect );
    virtual void    DrawHook( CDC* dc, double zoom );
    virtual void    DrawHook( CDC* dc, CRect rect );
    virtual void    GetDivisions( Vector<double>& x, Vector<double>& y );
    virtual void    GetDivisionsFromBegin( Vector<double>& x, Vector<double>& y );
    virtual void    GetDivisionsFromEnd( Vector<double>& x, Vector<double>& y );
    virtual bool    SetDivisions( int num );
    virtual bool    SetDivisionsDistance( double s );

    CSize           m_markerTrimSize;
    bool            intersected;
    bool            m_closed;
    Vector<double>  interx;
    Vector<double>  intery;
    Vector<double>  m_puntosx,m_puntosy;

    virtual void    DrawEndMarkers( CDC* dc );
    virtual int     HasDrawEndMarkers();
    virtual void    clearintersection();
    virtual void    SetIntersectionPoint( double px, double py );
    void            AddIntersections( Vector<double> interx, Vector<double> intery );

    virtual double  GetBeginX();
    virtual double  GetBeginY();
    virtual double  GetEndX();
```

```
    virtual double  GetEndY();
    bool            IsBeginIntersected();
    bool            IsEndIntersected();
    bool            IsRepeated( double x, double y );
    bool            AddRepeated( double x, double y );
    bool            IsEqual( double x1, double y1, double x2, double y2 );//
    virtual bool    GotExtremes();
    virtual void    RemoveRepeated();
    virtual void    RemoveSameIntersections( Vector<double>& interx, Vector<double>& intery );
    virtual void    SetFirst( CDiagramEntity* obj );
    virtual void    SetLast( CDiagramEntity* obj );

    virtual void    DrawIntersectionMarkers( CDC* dc, CRect rect ) const;
    virtual void    DrawIntersectionMarkers( CDC* dc, double zoom );
    virtual void    SetBorderColor( COLORREF value );

// Trimming
    virtual CDiagramEntityContainer*    GetTrims( CDiagramEntityContainer *ret ) ;
    virtual void    SortTrims();
    virtual void    Trim( CDiagramEntityContainer* orig, CDiagramEntity* trim, CDiagramEntityContainer* ret );

    double          Interval ( double x1, double y1, double x2, double y2 );//

protected:
    virtual CPoint  GetDrawPoint( CRect selrect, double x, double y ) const;

};
```

Primero se incluyen los archivos de cabecera que necesitaremos más adelante. Luego se definen las constantes, entre ellas se agregan DEHT_ENDMARKER_BE y DEHT_ENDMARKER_EN, utilizadas por la clase arco para identificar los puntos de edición (los cuadros amarillos), DETY_ARC y DETY_POLINE se asocian con los tipos de las respectivas figuras y los últimos tres conjuntos de constantes se utilizan para definir el tipo de contorno que se utiliza al generar los archivos de entrada (ver sección 8.2.3), siguen, la función round y las clases, los miembros de la clase DiagramEntity agregados se encuentran al final, por ahora no tiene sentido explicar cada uno de ellos, vale notar que casi todas las funciones son virtuales así que serán redefinidas en las clases descendientes.

Para manejar una colección de figuras existe una clase llamada CDiagramEntityContainer descendiente de CObArray que sera muy utilizada en el trabajo desarrollado, especialmente con contornos y polilineas. En ésta clase se escribieron las siguientes funciones:

```
    virtual void    SelectNext();
    virtual void    SelectPrevious();
    virtual void    Add( CDiagramEntityContainer* source );
    void            RemoveZeroSized();
    void            RemoveRepeatedPoints();
```

Las dos primeras permiten usar las teclas Tabs y Shift+Tabs para seleccionar la figura siguiente y anterior del contenedor. La función Add agrega una copia de cada elemento de source al contenedor.

7.6.2. Editor gráfico

La clase más importante es aquella que contiene todos los datos utilizados, ésta clase es CDiagramEditor, que se encuentra dentro de la clase CReportCreatorView como “CReportEditor m_-editor;” (CReportEditor desciende de CDiagramEditor). La mayoría de las utilidades se incorporan como miembros de esta clase, como se verá adelante.

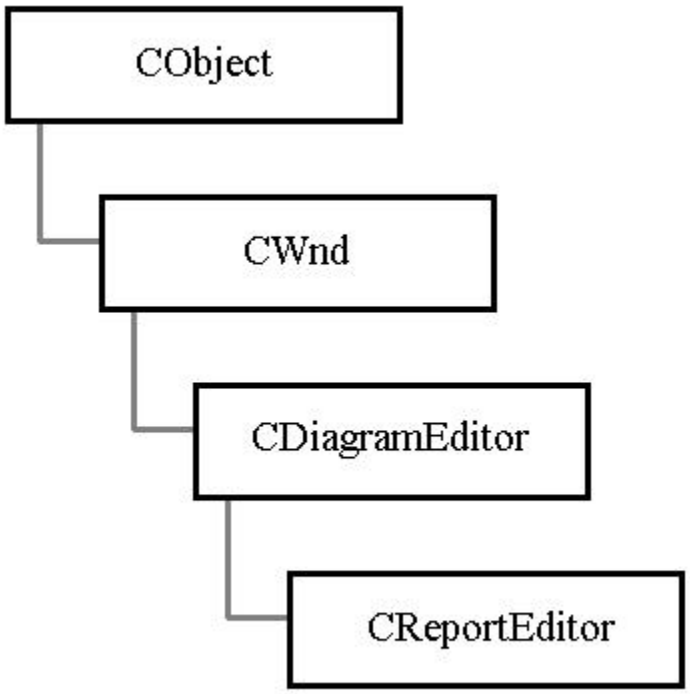

Figura 7.7: La genealogía de la ventana de edición

Este objeto maneja la ventana de dibujo, incluyendo los eventos del ratón y del teclado, además va a contener a todas las figuras. Como no vamos a modificar las reglas ni las unidades de distancia, los datos serán incorporados como miembros de esta clase. Pero los mensajes generados mediante *message map* son enviados a la clase de vista (como se hace habitualmente) la cual los re-envía a la clase del editor.

Por lo anterior, ésta clase es la que más elementos nuevos contiene, a continuación se muestran nuevas funciones declaradas en el archivo DiagramEditor.h,

```
#include "..\\ReportEntityArc.h"
#include "..\\ReportEntityPolyline.h"
#include "..\\..\\IntersectionsMatrix.h"
#include "..\\..\\ElFinConditionsDialog.h"
#include "..\\..\\ElFinElement2D.h"
#include "..\\..\\ElFinInputFormat.h"

...

#define MODE_EDITARC        6
#define MODE_TRIM           7
#define MODE_SEGMENT        8
#define MODE_DRAWING_POLY   9
#define MODE_RUBBER_ZOOM    10
#define MODE_CONTOUR        11

// Etapes
#define ETAPE_EDIT          0
#define ETAPE_MESH          1
#define ETAPE_SOLVE         2

// Boundary conditions
#define CONDITION_HEAT       0
#define CONDITION_CONV       1
#define CONDITION_TEMP       2

...

private:
```

```
    class CElFinConditionsDialog;

public:
    virtual void DrawStatus() const;

////    Trimming
    virtual void DivideSelected();
    virtual void TrimSelected();
    virtual void SegmentSelected();
    virtual void TrimPoints();

////    Intersections
    void AddIntersections( CDiagramEntity* first, CDiagramEntity* second, Vector<double>& ix, Vector<double>& iy );
    void AddExtremes( CDiagramEntity* first, CDiagramEntity* second );
    bool IntersectionExtremes   ( CDiagramEntity* first, CDiagramEntity* second, Vector<double>& ix, Vector<double>& iy );
    virtual bool intersection   ( CDiagramEntity* first, CDiagramEntity* second, Vector<double>& ix, Vector<double>& iy );
    bool intersected( double& x1, double& y1, double& x2, double& y2, double& x3, double& y3, double& x4, double& y4, double& ix, double& iy);
    bool intersectedlines       ( CReportEntityLine* first, CReportEntityLine* second, Vector<double>& ix, Vector<double>& iy );
    bool intersectedlinescolineal( double x1,double y1, double x2,double y2, double x3,double y3, double x4,double y4, double &px,double &py );
    bool intersectedlinerect    ( CReportEntityLine* line, CReportEntityBox* rect, Vector<double>& ix, Vector<double>& iy );
    bool intersectedlinecircle  ( CReportEntityLine* line, CReportEntityEllipse* ellipse, Vector<double>& ix, Vector<double>& iy );
    bool intersectedlinearc     ( CReportEntityLine* first, CReportEntityArc* second, Vector<double>& ix, Vector<double>& iy );
    bool intersectedrects       ( CReportEntityBox* first, CReportEntityBox* second, Vector<double>& ix, Vector<double>& iy );
    bool intersectedrectcircle  ( CReportEntityBox* rect, CReportEntityEllipse* ellipse, Vector<double>& ix, Vector<double>& iy );
    bool intersectedrectarc     ( CReportEntityBox* rect, CReportEntityArc* arc, Vector<double>& ix, Vector<double>& iy );
    int  interellipses          ( CReportEntityEllipse* ellf, CReportEntityEllipse* ells, Vector<double> &ix, Vector<double> &iy);
    bool intersectedcircles     ( CReportEntityEllipse* first, CReportEntityEllipse* second, Vector<double>& ix, Vector<double>& iy );
    bool intersectedcirclearc   ( CReportEntityEllipse* ellipse, CReportEntityArc* arc, Vector<double>& ix, Vector<double>& iy );
    bool intersectedarcs        ( CReportEntityArc* first, CReportEntityArc* second, Vector<double>& ix, Vector<double>& iy );

protected:
    CDiagramEntity* m_trim; // Temporary pointer to object that should be cut
    CDiagramEntity* GetHitTrim( CPoint point );
    void        AddContainer( CDiagramEntityContainer* cont );

private:
    CDiagramEntityContainer*    m_trim_objs;

public:
////    Rectangle Zoom
    CReportEntityBox* m_zoom_rect;
    virtual void ZoomRect();
    virtual void ZoomToRect();
    virtual void ZoomRectZoom();
    virtual void SelectNext();
    virtual void SelectPrevious();

////    Contours
    bool        m_show_divisions;
    void        SelContour();
    void        AllContours();
    bool        InContour( CDiagramEntity* obj );
    void        RefreshContour();
    void        ContoursModal();
    void        GetContours();
    int         FindExtreme( double x, double y, CDiagramEntityContainer* fill );
    CReportEntityPolyline*      m_contour;
    CDiagramEntityContainer*    m_contour_available;//filling
    CDiagramEntityContainer*    m_cont_contours;
    void        RefreshArc( CReportEntityArc* obj );

////    Boundary conditions
    CElFinBoundaryConditions*   m_boundary_conditions;
    void        BoundaryConditions();
    void        BoundaryConditionsModal();

private:
    CElFinConditionsDialog*     m_bc_dialog;
    CListCtrl                   m_bc_listctrl;

public:
////    Etapes
    int         m_etape;
    void        Etape( int etape );
    void        UpdateObjs( bool fromsaved );
    CDiagramEntityContainer*    m_saved_objs;   //m_objs response
    bool        canmove;
    bool        IsUndoPossible();

////    Elements
    CDiagramEntityContainer*    m_elements;
    CDiagramEntityContainer*    m_saved_elements;
    void        Elemental();
    void        GetElements();

////    Region Meshing
    CString     m_region_name;
    int         m_requested_no_of_triangles;
    void        RegionModal();
    bool        SaveInputFile();
```

del código antiguo se debe tener en cuenta la variable miembro,

```
CDiagramEntityContainer* m_objs; // Pointer to data
```

esta variable privada es un apuntador a un contenedor de objetos (de una clase que se verá en breve) el cual contiene las figuras creadas en el editor, nuestro programa utiliza éste e incluye otros contenedores, como m_contour_available, m_cont_contours, m_saved_objs, m_elements, m_-saved_elements.

Además se tendrán que modificar las siguientes funciones,

```
virtual void Draw( CDC* dc, CRect rect ) const;
virtual afx_msg void OnLButtonDown(UINT nFlags, CPoint point);
virtual afx_msg void OnLButtonUp(UINT nFlags, CPoint point);
virtual afx_msg void OnMouseMove(UINT nFlags, CPoint point);
virtual afx_msg void OnKeyDown(UINT nChar, UINT nRepCnt, UINT nFlags);
```

para dibujar en pantalla y para manejar los eventos del ratón sobre el area de dibujo y tecla pulsada.

Los anteriores miembros agregados a la clase se agrupan en bloques de acuerdo a su utilidad y están precedidos por un comentario, a *grosso modo*, se mencionan:

- DrawStatus, que actualiza la barra de estado con la información geométrica (ver la sección 7.11).
- Las funciones *trimming*, las tres primeras manejan los eventos de pulsado de los botones para recortar figuras desde CReportCreatorView y la última, TrimPoints, calcula los puntos de intersección entre las figuras (ver la sección Editar figuras, 7.9).
- Las funciones *Intersections*, que contienen los algoritmos geométricos utilizados para calcular los puntos de intersección entre las figuras. La labor de TrimPoints es seleccionar todas las parejas de objetos posibles y llamar a las primeras cuatro funciones de este bloque, AddIntersections, AddExtremes, IntersectionExtremes e intersection. Esta última llama selectivamente al resto de acuerdo a una verificación del tipo de las figuras (ver la sección Intersecciones entre figuras, 7.10).
- La función AddContainer que agrega los elementos de un contenedor a los objetos existentes.
- El programa incluye una utilidad para ampliar la vista a un cuadro seleccionado, para esto se utilizan m_zoom_rect y las siguientes tres funciones. Las siguientes dos funciones se utilizan para cambiar el objeto seleccionado por el siguiente y anterior.
- A continuación, precedidas por *Contours*, se encuentran las variables y funciones utilizadas para crear un conjunto ordenado de contornos a partir de los objetos disponibles, en estas funciones se verifican las uniones entre las figuras y finalmente se construye la variable m_-cont_contours, además se incluyen las funciones para definir condiciones de frontera y los puntos de division para el pre-procesador mediante un cuadro de diálogo modal (ver la sección 8.2.1).
- *Boundary conditions*, o condiciones de frontera se relaciona con los cinco siguientes miembros, m_boundary_conditions, que es un contenedor creado en este trabajo y dos funciones que inician el cuadro de diálogo de condiciones de frontera, ya sea de forma no modal o modal. En el caso modal se necesitan las variables m_bc_dialog y m_bc_listctrl (ver la sección 7.12).

- Precedidos por *Etapes* se encuentran los miembros que administran la etapa de trabajo en la que se encuentra el usuario. El primero es la variable m_etape, que conserva la etapa actual, sigue Etape, la función para fijar la etapa, la cual utiliza la siguiente función, UpdateObjs, que copia los datos desde o hasta m_saved_objs según las etapas de trabajo actual y requerida (ver la sección 8.1).

- Continuando con el bloque *Region Meshing*, se encuentran las variables que, junto con m_cont_contours permiten obtener la región discretizada, m_region_name y m_requested_no_of_triangles son variables que conservan el nombre y el numero requerido de triángulos para guardar. La función RegionModal abre el cuadro de diálogo para modificar estas variables así como otras características de m_cont_contours. Finalmente SaveInputFile escribe los dos archivos que necesita el pre-procesador, esta función se puede llamar desde el botón OK del cuadro de diálogo región (ver la sección 8.2.3).

- Por último se declaran las funciones que son necesarias en caso de querer dibujar los elementos finitos dentro del editor, como modelo se propone la función Elemental que dibuja un elemento con valores predefinidos a partir de tres objetos seleccionados y unidos por sus extremos.

7.7. Creación de objetos

Antes de proceder con las utilidades presentes en el programa, sera necesario definir nuevos objetos. Estos objetos son necesarios para manejar diferentes datos o simplemente para reunir una serie de tareas. En este trabajo se crearon las siguientes clases:

- class CReportEntityArc : public CDiagramEntity
- class CReportEntityPolyline : public CDiagramEntity
- class CElFinBC
- class CElFinBoundaryConditions
- class CElFinConditionsDialog : public CDialog
- class CElFinConstructRegionDialog : public CDialog
- class CElFinContoursDialog : public CDialog
- class CElFinElement2D : public CDiagramEntity
- class CElFinInputFormat

y otras cuyo uso se espera descontinuar. Por ahora, CReportEntityArc y CReportEntityPolyline son imprescindibles para la mayoría de las utilidades del resto del capítulo ya que representan figuras geométricas descendientes de CDiagramEntity.

7.7.1. La clase CReportEntityArc

Es necesario crear la figura arco si se van a editar las figuras disponibles actualmente, la linea, el rectángulo y la elipse. A esta clase se tendrá que asociar un cuadro de diálogo y una instancia en la clase CReportEntitySettings, se comienza por copiar y renombrar los archivos *.h y *.cpp de otra clase[||||].

En el archivo .h copiado se define un *guard* con un nombre único, este se debe reemplazar o modificar, por ejemplo, se cambia _CREPORTENTITYBOX_ por _CREPORTENTITYARC_-.Además se debe incluir el archivo CReportEntityArc.h en los siguientes lugares donde aparece incluido CReportEntityBox.h.

- ReportCreatorView.cpp (linea 25).
- ReportControlFactory.cpp (linea 21).
- ReportEntitySettings.h (linea 5).
- DiagramEditor.h (linea 15).

También se debe incluir el archivo de cabecera en ReportEntityEllipse.cpp (linea 25) y se debe reemplazar CReportEntityBox.h en ReportArcProperties.cpp (linea 21).

Esta clase ya se puede utilizar y se comporta igual que una CReportEntityBox. Para obtener características propias de un arco se deben modificar algunas funciones existentes pero antes es necesario agregar miembros privados particulares del arco, en CReportEntityArc.h

```
...

#include "ReportArcProperties.h"
#include "ReportEntityEllipse.h"

...

private:
    CReportArcProperties    m_dlg;

...

public:
    friend class    CDiagramEditor;
    friend class    CReportEntityEllipse;
```

```
 virtual void Draw( CDC* dc, CRect rect );
```

```
   virtual void    DrawEndMarkers( CDC* dc );
   virtual int     HasDrawEndMarkers();
   virtual HCURSOR GetCursor( int hit ) const;
   virtual CRect   GetDrawRect( CRect selrect ) const;
   virtual CPoint  GetPointFromNormalized( CRect selrect, double x, double y ) const;
   virtual void    SetFromNormalized( double l, double t, double r, double b );
   virtual void    SetRect( double left, double top, double right, double bottom );
   CPoint          GetStart();
```

[||||] se utilizó la clase CReportEntityBox.

```
    CPoint          GetEnd();
    double          GetCutx( double x, double y );
    double          GetCuty( double x, double y );
    virtual int     GetHitCode( const CPoint& point, const CRect& rect ) const;
    virtual double  GetBeginX();
    virtual double  GetBeginY();
    virtual double  GetEndX();
    virtual double  GetEndY();
    void            SetStart( double xa, double ya );
    void            SetEnd( double xb, double yb );
    void            FromEllipse( CReportEntityEllipse* ellipse, double xa, double ya, double xb, double yb );
    void            FromArc( CReportEntityArc* arc, double xa, double ya, double xb, double yb );
    double          centercoordx( double x );
    double          centercoordy( double y );
    double          centercoordxreverse( double x );
    double          centercoordyreverse( double y );
    void            resetsel();

////    Trims
    virtual CDiagramEntityContainer* GetTrims( CDiagramEntityContainer *ret ) ;
    virtual void    SortTrims();
    virtual void    SetFirst( CDiagramEntity* obj );
    virtual void    SetLast( CDiagramEntity* obj );
    virtual void    Trim( CDiagramEntityContainer* orig, CDiagramEntity* trim, CDiagramEntityContainer* ret );
    CPoint          p_begin;
    CPoint          p_end;
    CRect           drawrect;   //rectangulo para el trazado del arco
    double          startx;
    double          starty;
    double          endx;
    double          endy;

////    Divisions
    virtual bool    SetDivisions( int num );
    virtual bool    SetDivisionsDistance( double s );
    virtual void    GetCoordsFromAngle( double& x, double& y, double ang );
    virtual double  GetArcChord( double iniang, double finang );
    virtual double  GetBeginAngle();
    virtual double  GetEndAngle();
    virtual bool    GotExtremes();

private:
    double ax;
    double ay;
    double bx;
    double by;
    double selleft;
    double seltop;
    double selright;
    double selbottom;
};
```

Se debe agregar un miembro de CReportArcProperties y borrar el anterior para que el programa pueda asociar la clase correcta a su diálogo de propiedades, luego se declaran las clases amigas para poder acceder a los miembros privados particulares del arco.

Además de las funciones virtuales comunes a todas las clases descendientes de CDiagramEntity, la clase CReportEntityArc, en particular, requiere las siguientes funciones:

- GetCursor, que se sobrecarga para devolver el cursor particular de edición del arco.
- GetDrawRect, que calcula el rectángulo donde se inscribe el arco, a partir del rectángulo de selección del objeto.
- GetPointFromNormalized, SetFromNormalized, GetCutx y GetCuty son utilizadas internamente. GetStart, GetEnd, SetStart y SetEnd son las predecesoras de las funciones GetBeginX, etc. ahora pertenecientes a la clase base.
- FromEllipse y FromArc permiten obtener la geometría del arco a partir de aquella de una elipse o arco respectivamente.

- centercoordx, centercoordy, centercoordxreverse y centercoordyreverse se utilizan para convertir desde y hacia las coordenadas normalizadas propias del arco, resetsel, SetFirst y SetLast se utilizan internamente.
- p_begin, p_end, drawrect, startx, starty, endx, endy representan la geometría del arco mostrada en pantalla, los dos primeros para los puntos de edición (los cuadros amarillos), y los cuatro últimos para los puntos inicial y final, en coordenadas normalizadas respecto al centro de la elipse del arco, es decir de drawrect.
- La función GetCoordsFromAngle permite obtener las coordenadas x y y a partir del ángulo introducido respecto al arco.
- La función GetArcChord obtiene la longitud de una cuerda sobre el arco a partir de dos ángulos.
- Las funciones GetBeginAngle y GetEndAngle devuelven los ángulos del punto inicial y final del objeto.
- Por ultimo, las variables ax, ay, bx, by, selleft, seltop, selright, selbottom se utilizan internamente por el arco.

7.7.2. La clase CReportEntityPolyline

El procedimiento para crear una poli-línea es igual al de un arco como se mostró en la sección anterior, la declaración de la clase se encuentra en CReportEntityPolyline.h,

Además, la poli-línea declara sus propias variables y funciones para acceder a ellas, se destacan m_polyline_segments, que es un contenedor con los segmentos que constituyen la poli-línea y las funciones AddSegment y GetSegment para agregar y obtener, respectivamente un segmento. La primera función solo agrega un segmento si es el primero o alguno de sus extremos esta unido con algún extremo de la poli-línea. La segunda función obtiene el segmento con el índice i.

7.8. Funciones de dibujo

Ya se tiene un objeto arco o poli-línea, sin embargo el usuario no puede crear ni modificar uno a menos que se le ofrezca una función de dibujo. Para agregar un botón en la barra de herramientas se siguen éstos pasos:

Desde el IDE de Visual Studio, en la ventana ResourceView, la entrada Toolbar $>$ IDR_MAINFRAME se debe abrir el mapa de bits de la barra de herramientas. Se dibuja un arco como icono, y haciendo doble click sobre el botón creado se agrega el nuevo ID: ID_BUTTON_ADD_ARC.

Para generar las funciones que manejan los botones se abre Class Wizard con las teclas Ctrl+W, primero aparece el diálogo “Adding a Class” así que se oprime “Cancel”. En el diálogo “MFC Class Wizard”, en la pestaña “Message Maps”, en el cuadro combinado “Object ID’s” se selecciona ID_BUTTON_ADD_ARC, en Class Name, CReportCreatorView y en “Messages”, COMMAND. Al seleccionar este ultimo se debe resaltar el botón “Add Function...”, que debe oprimirse.

En el Diálogo: “Add Member Function” al aceptar el nombre (por omisión “OnButtonAddArc”) se regresa al Diálogo “MFC Class Wizard” y se da click en el botón “Edit code...”. Inmediatamente aparece el código de la función para ser editado, se escribe la línea,

```
m_editor.StartDrawingObject ( new CReportEntityArc );
```

en la función CReportCreatorView::OnButtonAddArc tal como muestra en la sección 7.5.1 donde se explica aquella clase. Se repite el procedimiento para la función OnButtonAddPolyline, usando el ID, ID_BUTTON_ADD_POLY.

7.8.1. Dibujar un arco

Cuando se incluye una nueva figura, siempre es necesario modificar la función Draw de la misma. Normalmente la clase contexto de dispositivo (CDC) ofrece todas las funciones de dibujo necesarias de tal manera que sólo es necesario cambiar una línea de código en Draw.

Sin embargo, la función disponible para dibujar un arco requiere como parámetro el rectángulo dentro del cual se inscribe el arco y no el actual rectángulo de selección. Además se requiere dibujar otro arco punteado superpuesto para al editar los puntos extremos del arco.

La función Draw del arco finalmente es la siguiente,

```
void CReportEntityArc::Draw( CDC* dc, CRect rect ){
    CRect drect = GetDrawRect( rect );
    CPoint start= GetPointFromNormalized( rect, startx, starty );
    CPoint end  = GetPointFromNormalized( rect, endx, endy );
    CPoint a    = GetPointFromNormalized( rect, ax, ay );
    CPoint b    = GetPointFromNormalized( rect, bx, by );
    p_begin = start;
    p_end = end;
    CPen dotpen;
    dotpen.CreatePen( PS_DOT, 1, RGB ( 0, 0, 0 ) );
    dc->SelectObject( &dotpen );
    dc->Arc( drect, start, end );
    dc->SelectObject( dotpen );
    dc->SelectStockObject( BLACK_PEN );
    CPen pen;
    if( GetBorder() ){
        pen.CreatePen( GetBorderStyle(), round( static_cast< double >
        ( GetBorderThickness() ) * GetZoom() ), GetBorderColor() );
        dc->SelectObject( &pen );
    }
    else
        dc->SelectStockObject( NULL_PEN );
    CBrush brush;
    if( GetFill() ){
        brush.CreateSolidBrush( GetFillColor() );
        dc->SelectObject( &brush );
    }
    else
        dc->SelectStockObject( NULL_BRUSH );
    dc->Arc( drect, a, b );
    dc->SelectStockObject( BLACK_PEN );
```

```
    dc->SelectStockObject( WHITE_BRUSH );
}
```

La diferencia entre ésta y otras funciones Draw es la inclusión de las siete primeras líneas de código y la repetición del procedimiento para dibujar el arco, la primera vez, con start y end como extremos y la segunda usando a y b.

La función GetDrawRect fue escrita para obtener el rectángulo en el que se inscribe el arco a partir del rectángulo de selección (el rectángulo señalado cuando se selecciona el objeto)

Además de modificar la función Draw del arco se escribe la función DrawEndMarkers para poder dibujar los cuadrados amarillos de edición, ésta función es prácticamente idéntica a CDiagramEntity::DrawSelectionMarkers y es llamada desde el mismo lugar.

7.8.2. Dibujar una poli-línea

La solución mas obvia para Polyline::Draw seria llamar sucesivamente a las funciones Draw de cada segmento, sin embargo existe un problema en la práctica, la función Draw utiliza el rectángulo del objeto para dibujar, como no hay relación entre el rectángulo de la poli-línea y el del segmento, no se puede utilizar este rectángulo para hacer un escalado y obtener el rectángulo del segmento porque el resultado es impredecible cada vez que se dibuja la ventana.

El problema del escalado se resuelve si en vez de llamar a Draw de cada segmento desde Draw de la poli-línea con un rectángulo incorrecto, se llama a Draw de cada segmento desde CReportEntityPolyline::DrawObject calculando el rectángulo correcto para cada segmento, para esto se aprovecha el hecho de que la función CDiagramEntity::DrawObject es virtual y se escribe la siguiente función sobrecargada para CReportEntityPolyline,

```
void CReportEntityPolyline::DrawObject( CDC* dc, double zoom ){
    SetZoom( zoom );
    CRect rect;
    CDiagramEntity* obj;
    for ( int i = 0; i < m_polyline_segments.GetSize(); i++ ){
        obj = m_polyline_segments.GetAt( i );
        rect = CRect( round( obj->GetLeft() * zoom ),
                      round( obj->GetTop() * zoom ),
                      round( obj->GetRight() * zoom ),
                      round( obj->GetBottom() * zoom ) );
        obj->Draw( dc, rect );
        if( IsSelected() ){
            obj->DrawSelectionMarkers( dc, rect );
            obj->DrawEndMarkers( dc );
        }
    }
}
```

7.9. Editar figuras

Además de dibujar y cambiar su tamaño, vamos a dividir los objetos en segmentos definidos por los puntos de intersección entre ellos, los algoritmos necesarios para obtener estos puntos se

explican en la siguiente sección, por ahora se tratan las funciones disponibles para el usuario final desde los botones “Dividir, Recortar y Segmento”. Como tarea preliminar, es necesario crear estos botones y asociar una función en CReportEntityView para cada uno, también se deben asociar los eventos *Update*, dividir se activa con al menos un objeto seleccionado y Recortar y Segmento con exactamente un objeto seleccionado.

Se desarrollan tres funciones en CDiagramEditor, DivideSelected, TrimSelected y SegmentSelected. Estas actúan sobre los objetos que se encuentren seleccionados, con DivideSelected todos los objetos seleccionados son cortados en segmentos definidos por los puntos de intersección mientras que con TrimSelected y SegmentSelected se entra en un modo interactivo en donde el usuario debe escoger el segmento que desea quitar o mantener respectivamente.

La más rápida y confiable de las funciones es DivideSelected ya que obtiene los objetos recortados simultáneamente, garantizando que el punto de corte siempre es igual al inicial. Esta función se llama desde la única linea de CReportCreatorView::OnButtonDivide y está definida de la siguiente forma,

```
void CDiagramEditor::DivideSelected(){
    if( m_objs != NULL ){
        m_objs->Snapshot();
        int con = 0;
        CDiagramEntity* objtemp;
        CDiagramEntityContainer* selectos = new CDiagramEntityContainer;
        CDiagramEntityContainer* divisiones = new CDiagramEntityContainer;
        CDiagramEntityContainer* temp = new CDiagramEntityContainer;
        divisiones->Clear();
        while ( objtemp = m_objs->GetAt( con++ ) ){
            if ( objtemp->IsSelected() ){
                temp = objtemp->GetTrims( temp );
                divisiones->Add( temp );
            }
        }
        objtemp = NULL;
        m_objs->RemoveAllSelected();
        m_objs->Add( divisiones );
        TrimPoints();
        m_objs->RemoveZeroSized();
        SetModified( true );
        delete selectos;
        delete divisiones;
        delete temp;
        RedrawWindow();
    }
}
```

Primero se deben crear varias variables locales, luego, dentro del bucle, se llena el contenedor divisiones con los segmentos obtenidos a partir de los objetos seleccionados llamando a la función GetTrims de cada objeto. Se llama a m_objs->RemoveAllSelected y m_objs->Add(divisiones) para reemplazar los objetos seleccionados por los segmentos divididos. Finalmente se actualizan los puntos de intersección con TrimPoints (no se retiran los objetos de tamaño nulo) y se dibuja la ventana.

Las funciones SegmentSelected y TrimSelected son más complicadas ya que requieren modificaciones en las funciones CDiagramEditor::OnLButtonDown, CDiagramEditor::OnLButtonUp y CDiagramEditor::OnMouseMove, la definición de las funciones es sencilla:

```
void CDiagramEditor::SegmentSelected() {
    m_objs->RemoveZeroSized();
    CDiagramEntityContainer* stack_objs = new CDiagramEntityContainer;
    int count = 0;
    CDiagramEntity* obj;
    if( m_objs ){
        while( ( obj = m_objs->GetAt( count++ ) ) )
            if( obj->IsSelected() )
                stack_objs->Add ( obj->Clone() );
        if( stack_objs->GetSize() == 1 ){
            CDiagramEntity* selobj = GetSelectedObject();
            if( m_subModeSelected )
                obj = m_subModeSelected;
            if ( selobj ){
                selobj->SetBorderColor( RGB ( 250, 20, 0 ) );
                selobj->SortTrims();
                m_trim = NULL;
                m_drawing = TRUE;
                m_interactMode = MODE_SEGMENT;
                SetModified( TRUE );
            }
        } else {
            m_drawing = FALSE;
            m_interactMode = MODE_NONE;
        }
        RedrawWindow();
    }
    delete stack_objs;
}
```

después de varias verificaciones se toma el objeto y se cambia su color a rojo, luego se ordenan los puntos de intersección mediante la función SortTrims y se cambia la variable m_interactMode a MODE_SEGMENT, esta variable define el modo en el que se encuentra el editor, se agregaron las constantes MODE_SEGMENT y MODE_TRIM al comienzo de DiagramEditor.h junto a los otros modos.

Para evitar repetir las funciones OnLButtonDown, OnMouseMove, OnLButtonUp se muestra la función TrimSelected que es practicamente identica a SegmentSelected

```
void CDiagramEditor::TrimSelected() {
    CDiagramEntityContainer* stack_objs = new CDiagramEntityContainer;
    int count = 0;
    CDiagramEntity* obj;
    if( m_objs != NULL ){
        m_objs->Snapshot();
        m_objs->RemoveZeroSized();
        while( ( obj = m_objs->GetAt( count++ ) ) )
            if( obj->IsSelected() )
                stack_objs->Add ( obj->Clone() );
        if( stack_objs->GetSize() == 1 ){
```

```
            CDiagramEntity* selobj = GetSelectedObject();
            if( m_subModeSelected )
                obj = m_subModeSelected;
            if ( selobj ){
                selobj->SetBorderColor( RGB ( 33, 130, 255 ) );
                selobj->SortTrims();
                m_trim = NULL;
                m_drawing = TRUE;
                m_interactMode = MODE_TRIM;
                SetModified( TRUE );
            }
        } else {
            m_drawing = FALSE;
            m_interactMode = MODE_NONE;
        }
        RedrawWindow();
    }
    delete stack_objs;
}
```

En esta función solo se cambia el color del objeto a un azul y el modo se cambia a MODE_TRIM.

7.10. Intersecciones entre figuras

En todos los casos anteriores se deben obtener los puntos de intersección entre los objetos existentes, esto se hace creando la función TrimPoints que será llamada por las tres funciones anteriores y en muchas otras ocasiones, esta se encargara de actualizar los puntos de intersección los cuales se van a guardar en los objetos.

```
void CDiagramEditor::TrimPoints() {
    HCURSOR tcurs = GetCursor();
    ::SetCursor( m_cursorWait );
    m_objs->Snapshot();
    if( m_objs ){
        int countf = 0;
        int counts = 0;
        CDiagramEntity* objfirst,*objsecond;
        Vector<double> interx, intery;
        int ctemp = 0;
        CDiagramEntity* objtemp;
        while( objtemp = m_objs->GetAt( ctemp++ ) ){
            objtemp->clearintersection();
            objtemp->intersected = FALSE;
        }
        while( objfirst = m_objs->GetAt( countf++ ) ){
            counts = countf;
            while( objsecond = m_objs->GetAt( counts++ ) ){
                if ( objfirst != objsecond ){
                    if ( intersection ( objfirst, objsecond, interx, intery ) ){
                        AddIntersections( objfirst, objsecond, interx, intery );
                    }
                    AddExtremes( objfirst, objsecond );
                }
            }
```

```
        }
        m_objs->RemoveRepeatedPoints();
    }
    SetModified( TRUE );
    ::SetCursor( tcurs );
}
```

Como la función es una de las que más tiempo puede demorar, primero se muestra el cursor m_cursorWait el cual es un característico reloj de arena. Después de ciertas verificaciones se recorre el contenedor de objetos borrando todos los puntos de intersección existentes.

A continuación se prueban todas las parejas posibles de objetos, para evitar repeticiones innecesarias cada pareja se debe escoger una sola vez, lo cual se logra barriendo todos los objetos con objfirst y todos los objetos ubicados por encima de objfirst con objsecond, como se comienza desde counts = countf (para evitar un espurio counts = countf + 1), se verifica que los dos objetos son diferentes. Para cada pareja se llama a la función intersection que retorna verdadero si las figuras están intersectadas, en tal caso los puntos hallados se agregan a los puntos de intersección de cada objeto, luego se llama a la función AddExtremes que compara los puntos extremos de los objetos y los une a los puntos de intersección en caso de ser iguales.

Desde la función TrimPoints se llaman principalmente tres funciones con cada pareja de objetos, AddExtremes, AddIntersections e intersection que se definen a continuación:

```
void CDiagramEditor::AddExtremes( CDiagramEntity* first,
CDiagramEntity* second ){
    Vector<double> tx, ty;
    tx.chaSize( 0 );
    ty.chaSize( 0 );
    if ( first->GetBeginX() != 0.0 && first->GetBeginY() != 0.0 && first->GetEndX() != 0.0 && first->GetEndY() != 0.0 ){
        if ( ( ( first->GetBeginX() == second->GetBeginX() ) && ( first->GetBeginY() == second->GetBeginY() ) ) ||
            ( ( first->GetBeginX() == second->GetEndX() ) && ( first->GetBeginY() == second->GetEndY() ) ) ){
            tx.addVal( first->GetBeginX() );
            ty.addVal( first->GetBeginY() );
            first->intersected = TRUE;
            second->intersected = TRUE;
        }
        if ( ( ( first->GetEndX() == second->GetBeginX() ) && ( first->GetEndY() == second->GetBeginY() ) ) ||
            ( ( first->GetEndX() == second->GetEndX() ) && ( first->GetEndY() == second->GetEndY() ) ) ){
            tx.addVal( first->GetEndX() );
            ty.addVal( first->GetEndY() );
            first->intersected = TRUE;
            second->intersected = TRUE;
        }
        addVector( first->m_puntosx, tx );
        addVector( first->m_puntosy, ty );
        addVector( second->m_puntosx, tx );
        addVector( second->m_puntosy, ty );
    }
}
```

En esta función se hace una (desactualizada) verificación de que los objetos tienen extremos, es decir son líneas, arcos o poli-líneas abiertas. Si tal es el caso se comparan los puntos iniciales y finales de los objetos y si alguno se repite, se añade a los puntos de división de los objetos, m_puntosx y m_puntosy.

```
void CDiagramEditor::AddIntersections( CDiagramEntity* first,
CDiagramEntity* second, Vector<double>& ix, Vector<double>& iy ) {
    first->AddIntersections( ix, iy );
    second->AddIntersections( ix, iy );
    first->intersected = TRUE;
    second->intersected = TRUE;
}
```

Esta función simplemente agrega simultáneamente los puntos a las intersecciones de ambos objetos. Por último la función CDiagramEditor::intersection,

```
bool CDiagramEditor::intersection( CDiagramEntity* first,
CDiagramEntity* second, Vector<double>& ix, Vector<double>& iy ) {
    int typefirst,typesecond;
    first->GetType(typefirst);
    second->GetType(typesecond);
    CDiagramEntity* temp;
    int typetemp;
    if (typesecond < typefirst){
        typetemp=typefirst;
        typefirst=typesecond;
        typesecond=typetemp;
        temp=first;
        first=second;
        second=temp;
    }
    double fl = first->GetLeft();
    double ft = first->GetTop();
    double fr = first->GetRight();
    double fb = first->GetBottom();
    double sl = second->GetLeft();
    double st = second->GetTop();
    double sr = second->GetRight();
    double sb = second->GetBottom();
    if ( !(
        ( ( fl < sl && fl < sr && fr < sl && fr < sr ) || ( fl > sl && fl > sr && fr > sl && fr > sr ) ) ||
        ( ( ft < st && ft < sb && fb < st && fb < sb ) || ( ft > st && ft > sb && fb > st && fb > sb ) ) ) ){
        switch (typefirst){
            case DETY_LINE :
                switch (typesecond){
                    case DETY_NONE:
                        return false;
                    case DETY_LINE:
                        return intersectedlines ( (CReportEntityLine*) first, (CReportEntityLine*) second, ix, iy );
                    case DETY_RECT:
                        return intersectedlinerect ( (CReportEntityLine*) first, (CReportEntityBox*) second, ix, iy );
                    case DETY_CIRCLE:
                        return intersectedlinecircle ( (CReportEntityLine*) first, (CReportEntityEllipse*) second, ix, iy );
                    case DETY_ARC:
                        return intersectedlinearc ( (CReportEntityLine*) first, (CReportEntityArc*) second, ix, iy );
                }
            case DETY_RECT :
                switch (typesecond){
                    case DETY_RECT:
                        return intersectedrects( (CReportEntityBox*) first, (CReportEntityBox*) second, ix, iy );
                    case DETY_CIRCLE:
                        {
                            CReportEntityBox* rect = ( CReportEntityBox* ) first;
                            CReportEntityEllipse* ellipse = ( CReportEntityEllipse* ) second;
                            return intersectedrectcircle( rect, ellipse, ix, iy );
                        }
                    case DETY_ARC:
                        {
                            CReportEntityBox* rect = ( CReportEntityBox* ) first;
                            CReportEntityArc* arc = ( CReportEntityArc* ) second;
                            return intersectedrectarc ( rect, arc, ix, iy );
                        }
                }
            case DETY_CIRCLE :
                switch (typesecond){
                    case DETY_CIRCLE:
                        return intersectedcircles( (CReportEntityEllipse*) first, (CReportEntityEllipse*) second, ix, iy );
                    case DETY_ARC:
                        return intersectedcirclearc( (CReportEntityEllipse*) first, (CReportEntityArc*) second, ix, iy );
                }
            case DETY_ARC :
                switch (typesecond){
                    case DETY_ARC:
                        return intersectedarcs ( (CReportEntityArc*) first, (CReportEntityArc*) second, ix, iy );
                }
        }
        return false;
    }
    return false;
}
```

Como se ve, esta es solo la punta del iceberg de la colección de funciones que se incorporan. Primero se organizan los objetos, luego se verifica que los rectángulos de ambos se intersecan (mejorando la eficiencia), y por último se ingresa a un *switch* que selecciona la función de intersección correcta entre ambos objetos. Se resalta que los argumentos de todas las funciones son los mismos argumentos de intersection, esto es posible gracias al uso de apuntadores.

7.10.1. Dos líneas

Cuando se tienen dos líneas se llama a la función CDiagramEditor::intersectedlines, en ésta función se realizan dos llamadas adicionales, la primera a la función CDiagramEditor::intersected que parece ser más eficiente que una regla de tres. Sin embargo esta función entrega un punto equivocado al utilizarse líneas colineales, es decir paralelas y unidas por un extremo así que sera necesario llamar a la función CDiagramEditor::intersectedlinescolineal en ese caso.

La función intersectedlinescolineal devuelve un valor verdadero si las líneas son paralelas y se tocan en sus extremos, en ese caso también devuelve el punto a través de px y py.

7.10.2. Línea con rectángulo

Se utiliza la función CDiagramEditor::intersectedlinerect, esta llama a la función miembro del rectángulo,

```
rect->linesfromrect( *left, *top, *right, *bottom );
```

con la cual se obtienen cuatro líneas a partir del rectángulo. Luego se llama a la función intersectedlines con la línea y cada una de éstas cuatro líneas locales.

7.10.3. Línea con elipse

La función CDiagramEditor::intersectedlinecircle se encarga de calcular el punto de intersección. Primero, se escalan las coordenadas a las coordenadas al sistema de referencia escalado a la elipse en donde ésta es un circulo de radio uno. A continuación se resuelve la ecuación de segundo grado obtenida. Existen tres casos, si no hay solución en los reales, se devuelve falso, si existe una única solución, es decir un solo punto de corte, se devuelve verdadero así como el punto, igualmente si hay dos soluciones pero en ese caso se devuelven dos puntos.

7.10.4. Línea con arco

Se utiliza CDiagramEditor::intersectedlinearc, la función primero crea una elipse a partir del arco, para esto es necesario fijar el rectángulo de la misma con GetDrawRect:

```
elipse.SetRect( second->GetDrawRect( second->GetRect() ) );
```

después de esto se llama a la función,

```
intersectedlinecircle( (CReportEntityLine*) first, &elipse, dummyx,
dummyy );
```

si existen intersecciones, aún es necesario que estas se encuentren sobre el arco, para esto se encuentra el ángulo de cada punto y se usa la función,

```
if ( IsAngleBetween ( ang, ini, fin ) )
```

definida en NativeFunctions.cpp, con el ángulo de la intersección y los ángulos inicial y final del arco, ésta función verifica que el ángulo se encuentre entre el ángulo inicial y el final inclusive, lo cual es complicado si el ángulo inicial tiene un valor mayor al ángulo final.

7.10.5. Dos rectángulos

la función CDiagramEditor::intersectedrects es muy similar a intersectedlinerect y utiliza esta última, esta vez se obtienen cuatro líneas de un rectángulo y se obtienen las intersecciones entre cada una de ellas y el rectángulo restante usando intersectedlinerect.

7.10.6. Rectángulo con elipse

La función asignada se llama CDiagramEditor::intersectedrectcircle y lo que hace es obtener las cuatro líneas habituales del rectángulo y encontrar las intersecciones de cada una de ellas y la elipse mediante intersectedlinecircle.

7.10.7. Rectángulo con arco

CDiagramEditor::intersectedrectarc es igual a intersectedrectcircle solo que se cambian las cuatro llamadas a intersectedlinecircle por intersectedlinearc.

7.10.8. Dos elipses

La función que calcula los puntos de intersección entre dos elipses merece ser explicada con detalle, a pesar de ser una función aproximada.

```
int CDiagramEditor::interellipses ( CReportEntityEllipse* ellf,
CReportEntityEllipse* ells, Vector<double> &ix, Vector<double> &iy){
    int ret = 0;
    double d = 0;
    double fcx = ( ellf->GetLeft() + ellf->GetRight() ) / 2.0;
    double fcy = ( ellf->GetTop() + ellf->GetBottom() ) / 2.0;
    double maxdist = max( ellf->GetLeft() + ellf->GetRight(),
                        ellf->GetTop() + ellf->GetBottom() );
    int veces = 360;
    CReportEntityLine lin;
    CReportEntityLine df;
    Vector<double> tx, ty;
    lin.SetLeft( fcx );
    lin.SetTop( fcy );
    lin.SetRight( fcx + maxdist * cos( 0.0 ) );
    lin.SetBottom( fcy + maxdist * sin( 0.0 ) );
    if ( intersectedlinecircle( &lin, ellf, tx, ty ) && tx.size() == ty.size() == 1 ){
        df.SetLeft( tx( 1 ) );
        df.SetTop( ty( 1 ) );
    } else {
```

```
        df.SetLeft( ellf->GetRight() );
        df.SetTop( ( ellf->GetTop() + ellf->GetBottom() ) / 2 );
    }
    for ( int j = 0; j < veces + 1; j++ ){
        lin.SetRight( fcx + maxdist * cos( 2.0 * pi * j / veces ) );
        lin.SetBottom( fcy + maxdist * sin( 2.0 * pi * j / veces ) );
        if ( intersectedlinecircle( &lin, ellf, tx, ty ) && tx.size() == 1 ){
            if ( j > 0 ){
                df.SetLeft( df.GetRight() );
                df.SetTop( df.GetBottom() );
            }
            df.SetRight( tx( 1 ) );
            df.SetBottom( ty( 1 ) );

            if ( intersectedlinecircle( &df, ells, tx, ty ) ){
                addVector( ix, tx );
                addVector( iy, ty );
                ret += tx.size();
            }
        }
    }
    return ret;
}
```

Esta función se encuentra habilitada en lugar de otros códigos más sofisticados que buscan resolver las ecuaciones matriciales de las superficies cónicas, y que permanecen escritos en alguna parte del código, desafortunadamente los puntos generados por éstas funciones pueden multiplicarse o desaparecer eventualmente, por lo cual se prefiere la función anterior.

En la función interellipses se define la constante entera veces con un valor constante de 360. Se crea una nueva línea a la cual se le fijan las coordenadas para que sea un minúsculo segmento de cuerda de una de las elipses tal que el ángulo de la cuerda sea igual a $\pi/360$. Esto se repite en el bucle un numero "veces" y en cada paso se calcula la intersección entre este segmento y la otra elipse. Al final obtiene la intersección entre una elipse y el polígono de "veces" lados que aproxima a la otra elipse. Se entiende que se escogió este numero porque el polígono es prácticamente igual a la elipse original.

La función CDiagramEditor::intersectedcircles es la que realmente se llama desde intersection, esta aun tiene la tarea de re-direccionar la llamada a la función más eficiente, es decir interellipses, así se evita cambiar el nombre en cada llamado si se llegara a cambiar de función predilecta.

7.10.9. Elipse y arco

La intersección entre una elipse y un arco utiliza la intersección entre dos elipses, primero se obtiene la elipse necesaria y luego se revisa que los puntos obtenidos se encuentren entre los ángulos inicial y final del arco (ver intersección entre una linea un arco).

7.10.10. Dos arcos

Esta función es igual a la anterior, solo que esta vez hay que obtener dos elipses llamando a first->GetDrawRect y second->GetDrawRect. Se hallan las intersecciones entre ellas y finalmente

se verifica que el ángulo sea correcto, esta vez se llama dos veces a la función IsAngleBetween de esta manera,

```
if ( IsAngleBetween ( angf, inif, finf ) && IsAngleBetween ( angs, inis, fins ) )
```

7.11. La barra de estado

En una aplicación con ventanas, la barra de estado se encuentra en el extremo inferior de la ventana, esta se utiliza para mostrar información al usuario de la misma. Por defecto la barra de estado contiene un campo el cual muestra información adicional cuando el cursor se encuentra sobre un botón de la barra de herramientas o un menu. Además necesitamos mostrar nuestra propia información geométrica.

Primero se crean tres campos además del existente, esto se hace agregando las siguientes tres líneas en la función CMainFrame::OnCreate antes de la llamada a EnableDocking,

```
    m_wndStatusBar.SetPaneInfo( 1, 0, 0, 70 );
    m_wndStatusBar.SetPaneInfo( 2, 0, 0, 260 );
    m_wndStatusBar.SetPaneInfo( 3, 0, 0, 70 );
```

```
CDiagramEntityContainer* m_wndToolBar.EnableDocking(CBRS_ALIGN_ANY);
```

con lo cual obtenemos tres campos de 70, 260 y 70 pixels de ancho a la derecha del actual. Se debe escribir la siguiente línea de código en CDiagramEditor::DrawObjects:

```
if(m_objs)
{

        DrawStatus();
```

La función DrawStatus se define de la siguiente manera

```
void CDiagramEditor::DrawStatus() const {
    CMainFrame* pFrame = ( CMainFrame* ) AfxGetApp()->m_pMainWnd;
    CStatusBar* pStatus = &pFrame->m_wndStatusBar;
    if ( pStatus ){
        CString label;
        CDiagramEntity* tempobj = GetSelectedObject();
        int count = 0;
        int zsized = 0;
        int sumax = 0;
        int sumay = 0;
        int selindex = -1;
        CDiagramEntity* obj;
        while( ( obj = m_objs->GetAt( count++ ) ) ){
            if ( obj->GetLeft()==obj->GetRight()&&obj->GetTop()==obj->GetBottom() )
                zsized++;
            sumax +=obj->m_puntosx.size();
            sumay +=obj->m_puntosy.size();
            if ( obj->IsSelected() ){
                selindex = count;
```

```
            }
        }
    label.Format ( "%d/%d", GetSelectCount(), m_objs->GetSize() );
        CString tamano;
        if ( GetSelectCount() == 1 ){
            if ( selindex > 0 ){
                CString hp;
                hp.Format( " %d:", selindex );
                label += hp;
            }
            double tl = tempobj->GetLeft();
            double tt = tempobj->GetTop();
            double tr = tempobj->GetRight();
            double tb = tempobj->GetBottom();
            tamano.Format( "[%f,%f][%f,%f]", tl, tt, tr, tb );
            if ( tl == tr && tt == tb )
                tamano += "z";
            if ( tr - tl == tb - tt )
                tamano += " D";
            CString delbug;//intersecciones
            if ( tempobj->m_puntosx.size() != tempobj->m_puntosy.size() )
                delbug += "ERROR N INTERSECT";
            if ( tempobj->m_puntosx.size() > 0 ){
                delbug.Format( "i%d", tempobj->m_puntosx.size() );
                if ( tempobj->IsBeginIntersected() )
                    delbug += "<";
                if ( tempobj->IsEndIntersected() )
                    delbug += ">";
                label += delbug;
            }
        } else {
            tamano.Format( " " );
        }
        pStatus->SetPaneText( 1, label );
        pStatus->SetPaneText( 2, tamano );
    }
}
```

En las primeras dos líneas se pide la barra de estado de la aplicación, luego se calculan las variables locales, zsized y selindex con el número de objetos de tamaño cero y seleccionados. A continuación se construyen las cadenas label y tamaño.

En label se muestran los objetos seleccionados y totales como seleccionadas/totales, si hay una sola figura seleccionada se agrega a label el numero de intersecciones y "<" y/o ">" si la figura se interseca en su punto inicial y/o final, en tamaño se escribe el tamaño del objeto seleccionado con el formato "[izquierda superior][derecha inferior]" y se agrega "z" si la figura tiene temaño cero y "D" si es diagonal (ancho igual al alto). Finalmente las cadenas label y tamaño se escriben en los campos 1 y 2 (el inicial es cero).

Hay que escribir la coordenada actual del cursor en el campo 3, sin embargo la función Draw-Status no es el lugar adecuado para esto ya que la coordenada no se actualizaría a tiempo, en su lugar, se agregan las siguientes líneas al comienzo de CDiagramEditor::OnMouseMove,

```
    CString str;
    CMainFrame* pFrame = ( CMainFrame* ) AfxGetApp()->m_pMainWnd;
```

```
    CStatusBar* pStatus = &pFrame->m_wndStatusBar;
    if ( pStatus ){
        CString xpos, ypos;
        CPoint vip( point );
        ScreenToVirtual( vip );
        xpos.Format( "%d", vip.x );
        ypos.Format( "%d", vip.y );
        str.Format( "( " + xpos + "," + ypos + " )" );
        pStatus->SetPaneText( 3, str );
    }
```

que son similares a DrawStatus, hay un detalle importante, es necesario obtener el verdadero punto a partir de las coordenadas de pantalla suministradas por OnMouseMove lo cual se hace con la función ScreenToVirtual.

7.12. Condiciones de frontera

Las condiciones de frontera son un conjunto de valores que debe satisfacer la solución obtenida o su derivada en determinados puntos y que restringen las posibles soluciones del problema. Un problema debe ser planteado con el número de condiciones adecuado para que tenga solución única.

Sin importar el planteamiento del problema, el programa debe permitirle al usuario introducir el número de condiciones de frontera que considere necesarios y hacerlo en cualquier momento. Eventualmente se podrían guardar ciertas condiciones comúnmente utilizadas, por ejemplo la Temperatura de Fusion del hielo, 0º C.

Una condiciones de frontera se puede definir como un dato compuesto de un numero, un nombre, un tipo (en este caso solamente Temperatura fija, Flujo de calor fijo o Flujo de calor por convección), un valor(o dos para convección) y las unidades en que está(n) dado(s).

Se crean dos clases para manejar este conjunto de datos, la clase **CElFinBC** para cada condición de frontera y la clase **CElFinBoundaryConditions** como el contenedor de todas las condiciones de frontera CElFinBC. Estas clases solo están relacionadas con los tipos de sus variables miembro: int, double y CString.

La clase CElFinBC es una sencilla clase declarada de la siguiente manera:

```
class CElFinBC {
    int     m_numb;
    CString m_name;
    CString m_type;
    double  m_value;
    CString m_value_units;
    double  m_value_2;
    CString m_value_2_units;
public:
    int     Get_numb();
    CString Get_name();
    CString Get_type();
    double  Get_value();
```

```
    CString Get_value_units();
    double  Get_value_2();
    CString Get_value_2_units();
    void    Set( int n, CString bc_name, CString bc_type, double bc_value,
            CString bc_value_units, double bc_value_2, CString bc_value_2_units );
};
```

Como se puede ver esta clase fija los datos mediante la función Set(...) y los obtiene mediante las funciones Get...(), ya que es más común fijar todos los datos al mismo tiempo. Estas funciones están definidas de la siguiente manera:

```
void CElFinBC::Set( int n, CString bc_name, CString bc_type, double
bc_value, CString bc_value_units, double bc_value_2, CString
bc_value_2_units ) {
    m_numb = n;
    m_name = bc_name;
    m_type = bc_type;
    m_value_units = bc_value_units;
    m_value_2_units = bc_value_2_units;
    m_value = bc_value;
    m_value_2 = bc_value_2;
    return;
}

int CElFinBC::Get_numb() {
    return m_numb;
}
```

Las otras funciones Get son similares a GetNumb(). La clase CElFinBC no necesita ni constructores ni destructor ya que todas sus variables miembro tienen los respectivos, además la clase no maneja la memoria.

A diferencia de CElFinBC, la clase siguiente, CElFinBoundaryConditions, maneja la memoria utilizada (muy primitivamente) y asigna el espacio requerido dinamicamente durante la ejecución del programa.

La clase CElFinBoundaryConditions es un contenedor de condiciones de frontera (clases CElFinBC), que ofrece una interfase más extensa como se puede ver en la declaración de la clase:

```
class CElFinBoundaryConditions {
    int         m_size;
    CElFinBC*   m_conditions;// [];
public:
    CElFinBoundaryConditions();
    CElFinBoundaryConditions( CElFinBC bc );
    ~CElFinBoundaryConditions();
    bool        Add( CElFinBC bc );
    bool        Del( CElFinBC bc );
    bool        DelAll();
    CElFinBC*   Get( int i );
    int         GetSize();
};
```

Esta clase va a tener una única instancia miembro del editor y va a ser manipulada únicamente por el diálogo de condiciones de frontera. Los miembros de la clase son un puntero al primer elemento de un *array* y el tamaño del array size.

7.13. Interacción con archivos

La forma en que el programa se relaciona con los archivos esta definida por la arquitectura MDI y es la principal razón para manejar el documento como una clase independiente. La clase documento se encarga de escribir y leer los datos utilizados de/en una cadena de caracteres utilizando la función **serialize**.

La clase documento utilizada ofrece la función serialize para que nosotros la sobre-carguemos, esto se hace usando la arquitectura del editor, llamando a las funciones CDiagramEntityContainer::GetString y CDiagramEntity::GetString para guardar un archivo y CReportControlFactory::CreateFromString para leer un archivo.

Para agregar información del proyecto se modifica GetString del contenedor y para agregar información de una figura se modifica GetString de la correspondiente entidad gráfica, en general no se utiliza la función virtual de la clase base CDiagramEntity sino que cada clase heredera tiene su propia función GetString.

La función GetString llama a GetDefaultGetString y esta a GetType, GetLeft, GetTop, GetRight, GetBottom, GetTitle, GetName, y GetGroup.

Para leer las nuevas clases de un archivo hay que agregar las siguientes líneas al final de CReportControlFactory::CreateFromString(CString),

```
    if( !obj )
        obj = CReportEntityArc::CreateFromString( str );
```

para cada nueva clase.

Hay que crear las funciones CreateFromString para cada clase, por ejemplo se tiene que escribir la siguiente función para crear un arco a partir de una cadena de caracteres:

```
CDiagramEntity* CReportEntityArc::CreateFromString( const CString
&str ){
    CReportEntityArc* obj = new CReportEntityArc;
    if(!obj->FromString( str ) ){
        delete obj;
        obj = NULL;
    }
    return obj;
}
```

Para mantener consistencia entre los datos escritos y los que se leen se deben conservar las siguientes equivalencias en los pares de funciones:

Para las poli-líneas se debe tener cuidado ya que los datos almacenados dentro del objeto son contenedores de más objetos. Puede existir conflicto si la cadena de caracteres anexada por GetString se confunde con la cadena del objeto o con la cadena del contenedor del documento cuando sea leída por FromString. En la función GetString de la poli-línea se deben incluir estas líneas para guardar la información de los segmentos:

Funcion de Escritura	Funcion de Lectura
GetString	FromString
ar.WriteString	ar.ReadString

Tabla 7.1: Funciones equivalentes de lectura y escritura

```
CDiagramEntity* obj;
while( ( obj = m_objs.GetAt( count++ ) ) )
    ar.WriteString( obj->$GetString() + _T("\r\n") );
```

CAPÍTULO 8

ENTRADA AL PRE-PROCESADOR

El pre-procesador es el programa encargado de generar una región discretizada o *malla* a partir de cierta geometría y condiciones de frontera. Como se mostró anteriormente, la región discretizada se utiliza en los metodos de elementos finitos. En este capítulo se muestran las funciones que utiliza el usuario para definir una región para discretizar. Antes de esto, en la sección 8.1 se muestra la secuencia que debe seguir el usuario mediante etapas de trabajo. Existen dos etapas consecutivas:

- **Etapa de edición.** En la cual se define la geometría y una lista con las condiciones de frontera del problema.
- **Etapa de discretización o enmallado.** Donde se deben definir características especiales para el pre-procesador como número de puntos de la geometría (ver 8.2.2), condiciones de frontera sobre cada punto y numero de elementos de la región (triangulos). Tambien se escriben los archivos y se obtiene y dibuja la región discretizada (ver 8.2.3).

Las funciones mostradas en el capítulo anterior están en la etapa de edición. El programa inicia en la etapa de edición por defecto asi que éstas se encuentran disponibles. Para acceder a las funciones de enmallado (principalmente a través de cuadros de diálogo) es necesario pasar a la **etapa de discretización**. En la sección 8.2.3 se muestra como se obtienen los archivos que requiere el pre-procesador y como se llama al mismo para obtener la región discretizada.

8.1. Etapas

La función Etape es llamada directamente por el usuario del programa utilizando un argumento que identifica la etapa a la que se va a acceder (Ver sección 7.5.1),

```
void CDiagramEditor::Etape( int etape ) {
    CString adv;
    adv.Format("Si regresa a la etapa de edicion perdera la informacion de enmallado");
    CString pasetape;
    pasetape.Format( "Debe acceder desde otra etapa" );
    if ( etape == ETAPE_EDIT ){
        if ( m_etape != ETAPE_EDIT )
            AfxMessageBox( adv );
```

```
        m_etape = ETAPE_EDIT;
        delete m_cont_contours;
        m_cont_contours = NULL;
        UpdateObjs( true );
    } else if ( etape == ETAPE_MESH ){
        if ( m_etape == ETAPE_MESH )
            AfxMessageBox( pasetape );
        else{
            m_etape = ETAPE_MESH;
            AllContours();
            UpdateObjs( false );
            m_objs->Clear();
        }
    } else if ( etape == ETAPE_SOLVE ){
        m_etape = ETAPE_SOLVE;
    }
    RedrawWindow();
}
```

8.2. Funciones para el pre-procesador

Para obtener la región discretizada, además de definir una geometrá (figuras) y una lista de condiciones de frontera, es necesario presentar tal información en el formato requerido por el pre-procesador.

- **Hallar todos los contornos cerrados.** Es una función que ejecuta el programa automaticamente cuando se pasa a la etapa de enmallado (ver 8.2.1), para verificar que la geometria con la cual se va a trabajar representa una región cerrada.
- **Hallar los puntos de división.** Como el pre-procesador lee una serie de puntos en vez de una geometrá contínua, es necesario definir y obtener una serie de puntos sobre los contornos disponibles. Sobre estos puntos se definen las condiciones de frontera.
- **Generar archivos de entrada.** Toda la información anterior debe escribirse en un formato comprensible por el pre-procesador dentro de dos archivos `geom.i` y `prepro.part`.

El ultimo paso se realiza cuando el usuario solicita crear la malla, es decir cuando pulsa el boton OK del cuadro de diálogo Región. Inmediatamente se escriben los archivos de entrada para el pre-procesador, se llama al mismo (se ejecuta el programa) para obtener el archivo de malla y se lee el archivo de malla para dibujarla en pantalla.

8.2.1. Hallar todos los contornos cerrados

Cuando se pasa de la etapa de edición a la etapa de mallado (discretización) es necesario tener uno o más contornos cerrados y no intersecandose entre sí los cuales definen la región a dividir. El proceso debe realizarse automáticamente, la interfase gráfica le informa al usuario que el contorno está correctamente definido dibujándolo de color verde, en caso contrario, los objetos que no definan una geometría cerrada se dibujarán de color amarillo.

Internamente, el programa llena las variables privadas m_saved_objs y m_contours desde la funcion Etape. La variable m_contours es llenada con poli-líneas construidas a partir de los objetos presentes (Ver la primera sección de este capitulo).

De la función Etape, creada para administrar los modos de trabajo del usuario, se requiere la llamada a la función AllContours para encontrar todos los contornos cerrados (y abiertos) posibles. La función AllContours llena el contenedor m_contours con poli-líneas, cada una de ellas representando un contorno ya sea cerrado o abierto, para esto llama repetidamente a la función RefreshContour cada vez con una figura diferente.

RefreshContour construye una poli-línea a partir de la figura disponible buscando consecutivamente las figuras se encuentren unidas por el extremo, como se muestra a continuación,

```
void CDiagramEditor::RefreshContour() {
    double m_contour_search_x = -1.0;
    double m_contour_search_y = -1.0;
    if ( m_contour ){
        m_contour->GetEnd( m_contour_search_x, m_contour_search_y );
        if ( m_contour_search_x >= 0.0 && m_contour_search_y >= 0.0 ){
            int borrame = 0;
            delete m_contour_available;
            m_contour_available = new CDiagramEntityContainer;
            borrame = FindExtreme( m_contour_search_x, m_contour_search_y, m_contour_available );
            bool ya = false;
            while ( borrame == 2 ){
                double savedx = m_contour_search_x;
                double savedy = m_contour_search_y;
                for ( int j = 0; j < borrame; j++ ){
                    if ( InContour( m_contour_available->GetAt( j ) ) == false ){
                        if ( m_contour_search_x == (m_contour_available->GetAt( j ))->GetBeginX() &&
                        m_contour_search_y == (m_contour_available->GetAt( j ))->GetBeginY() ){
                            m_contour->AddSegment( m_contour_available->GetAt( j ) );
                            m_contour->SetBorderColor( RGB( 255, 255, 0 ) );
                            m_contour->SetEnd( (m_contour_available->GetAt( j ))->GetEndX(),
                                (m_contour_available->GetAt( j ))->GetEndY() );
                            m_contour->GetEnd( m_contour_search_x, m_contour_search_y );
                            delete m_contour_available;
                            m_contour_available = new CDiagramEntityContainer;
                            borrame = FindExtreme( m_contour_search_x, m_contour_search_y, m_contour_available );
                            ya = true;
                        } else {
                            if ( m_contour_search_x == (m_contour_available->GetAt( j ))->GetEndX() &&
                            m_contour_search_y == (m_contour_available->GetAt( j ))->GetEndY() ){
                                m_contour->AddSegment( m_contour_available->GetAt( j ) );
                                m_contour->SetBorderColor( RGB( 255, 255, 0 ) );
                                m_contour->SetEnd( (m_contour_available->GetAt( j ))->GetBeginX(),
                                    (m_contour_available->GetAt( j ))->GetBeginY() );
                                m_contour->GetEnd( m_contour_search_x, m_contour_search_y );
                                delete m_contour_available;
                                m_contour_available = new CDiagramEntityContainer;
                                borrame = FindExtreme( m_contour_search_x, m_contour_search_y, m_contour_available );
                                ya = true;
                            } else
                                ya = false;
                        }
                        double cbx, cby, cex, cey;
                        m_contour->GetBegin( cbx, cby );
                        m_contour->GetEnd( cex, cey );
                        if ( cbx == cex && cby == cey ){
                            m_contour->SelectAll();
                            m_contour->SetBorderColor( RGB( 0, 255, 0 ) );
                            borrame = 0;
                        }
                    }
                }
            }
        }
    }
}
```

Existen dos posibilidades que contempla la función RefreshContour, que la figura sea una figura cerrada, como un rectángulo, una elipse o una poli-línea cerrada o que sea una figura abierta como

una línea, un arco o una poli-línea abierta.

En el primer caso la función no hace nada, en el segundo, la función entra en un bucle en el que se agregan segmentos (figuras) a la poli-línea siempre que **un solo** segmento y la poli-linea estén conectados en el extremo, cuando se cierra la poli-linea se cambia el color de la poli-linea de amarillo a verde.

Del anterior algoritmo se resaltan éstas características, la poli-línea crece por un solo extremo (también podría hacerlo por los dos) y no se permiten bifurcaciones (mas de dos objetos unidos por un mismo punto) ya que la poli-línea resultante queda abierta.

8.2.2. Hallar los puntos de división

Teniendo funciones para calcular las intersecciones, es sencillo obtener puntos de división sobre las figuras, matemáticamente, se parametriza el perímetro de la figura y se busca el punto en el plano correspondiente al valor del parámetro. Los puntos se almacenan como m_div_x y m_div_y.

Se presentan tres situaciones principales, hallar los puntos de division de una línea recta, hallar los puntos de una elipse y hallar los puntos de una poli-línea. Se desarrollaron las funciones virtuales SetDivisions y SetDivisionsDistance para fijar los puntos a partir de un numero solicitado por el usuario o de una distancia entre puntos solicitada.

Para las líneas rectas, se escribe la función SetDivisions, que calcula y fija los puntos de división de la línea a partir de un número entero y la función SetDivisionsDistance que calcula el número entero de puntos más adecuado para la distancia y a continuación llama a SetDivisions con éste.

Para las líneas curvas, la situación es más difícil y es necesario escribir tanto SetDivisions como SetDivisionsDistance, ambas son similares, sin embargo la función SetDivisions tiene una dificultad adicional y es que no conoce con anterioridad la distancia del punto actual así que hay que calcularla. Esta es la definición de la función SetDivisions del arco,

```
bool CReportEntityArc::SetDivisions( int num ) {
    bool ret = false;
    m_div_x.chaSize( 0 );
    m_div_y.chaSize( 0 );
    if ( num > 1 ){
        double abeg = GetBeginAngle();
        double aend = GetEndAngle();
        double l = GetArcChord( aend, abeg );
        if ( IsAngleBetween( pi, -abeg, -aend ) )
            aend += 2 * pi;
        m_div_x.addVal( GetBeginX() );
        m_div_y.addVal( GetBeginY() );
        int conta = 1;
        int nu = 20 * num;
        double s;
        double x1 = GetBeginX();
        double y1 = GetBeginY();
        double tempx, tempy, tempang, dista;
        for ( int i = 1; i < nu; i++ ){
```

```
            s = (double) i / (double) nu ;
            tempang = abeg + ( s * ( aend - abeg ) );
            tempang = AngleInDom( tempang );
            GetCoordsFromAngle( tempx, tempy, tempang );
            dista = pow( Interval( tempx, tempy, x1, y1 ), 0.50 );
            if ( dista > ( (double)l / (double)num ) ){
                m_div_x.addVal( tempx );
                m_div_y.addVal( tempy );
                x1 = tempx;
                y1 = tempy;
                conta++;
            }
        }
        m_div_x.addVal( GetEndX() );
        m_div_y.addVal( GetEndY() );
        conta++;
        ret = true;
    } else {
        if ( num = 1 ){
            m_div_x.addVal( GetBeginX() );
            m_div_y.addVal( GetBeginY() );
            m_div_x.addVal( GetEndX() );
            m_div_y.addVal( GetEndY() );
            ret = true;
        }
    }
    return ret;
}
```

Lo más importante de la función anterior es la llamada a GetArcChord, que obtiene numéricamente la longitud del arco*, con este valor se puede hacer prácticamente el mismo procedimiento de una línea recta. Se observa que hay un numero veinte (20), este es el numero de veces que se calcula la longitud entre un punto de división y el siguiente†, cuando la longitud calculada supera a l/num, se cambia el primer punto por el siguiente.

8.2.3. Generar archivos de entrada

Implícitamente la meta del programa es definir una región con condiciones de frontera, para esto se deben crear dos archivos (llamados archivos de entrada, geom.i y prepro.part) que son leídos por el programa que genera la región enmallada (llamado **pre-procesador**). Estos archivos deben ser cuidadosamente escritos para que sean interpretados correctamente por el pre-procesador, el usuario del programa no necesita tener conocimiento de la existencia de estos archivos y mucho menos modificarlos manualmente.

La clase que escribe estos archivos a partir de los datos existentes se llama CElFinInputFormat y se define en el archivo ElFinInputFormat.h:

```
#include "REPORTEDITOR\DIAGRAMEDITOR\DiagramEntityContainer.h"
#include "ElFinConditionsDialog.h" #include "..\\..\\PuntoMalla.h"
#include "REPORTEDITOR\ReportEntityPolyline.h"
```

*El calculo analítico exacto de la longitud de un arco implica resolver la llamada *integral elíptica*.

†es decir que cada punto tiene un error máximo de 5 %

```
class CElFinInputFormat { public:
    CString m_region_name;
    CString m_no_of_space_dimensions;
    CString m_case_type;
    CString m_no_of_boundary_vertices;
    CString m_no_of_boundary_curves;
    CString m_no_of_vertices_per_boundary_curve;
    CString m_indicate_type_and_location_of_boundary_curves;
    CString m_distinct_boundary_coor;
    CString m_indices_of_vertices_of_boundary_curves;
    CString m_no_of_boundary_indicators;
    CString m_boundary_indicator_names;
    CString m_boundary_indicator_list;

    CElFinInputFormat();
    CElFinInputFormat( CDiagramEntityContainer* conta );
    virtual ~CElFinInputFormat();
    virtual void    Save( CString name );
    virtual void    SavePart( CString name, int requested_n_o_t );
    virtual void    Update();
    virtual bool    Check( Vector<double>& outx, Vector<double>& outy,
                        const Vector<double>& inx, const Vector<double>& iny,
                        double& fstx, double& fsty,
                        double& lstx, double& lsty );
    virtual void    Invert( Vector<double>& x, Vector<double>& y );
    virtual CString GetDBC( CDiagramEntity* obj );
    virtual CString GetIOVOBC( CDiagramEntity* obj );
    virtual CString GetIOVOBC( int beg, int end, bool reverse );
    virtual CString GetBIL( int beg, int end, CDiagramEntity* obj );
    virtual CDiagramEntity* GetSorted( int i );
    virtual void    GetIndexes( int& beg, int& end, CDiagramEntity* obj );
    CElFinBoundaryConditions* bcs;
private:
    CDiagramEntityContainer* segments;
};
```

Las variables miembros de las clase son cadenas de caracteres cuyos nombres corresponden con los campos que se escriben en los archivos. Además se necesitan conocer las condiciones de frontera existentes mediante CElFinBoundaryConditions* bcs y los objetos de los cuales se obtiene la geometría mediante CDiagramEntityContainer* segments. Esta clase se utiliza con las funciones Save y SavePart, que guardan los archivos de entrada (el archivo de geometría y el de las condiciones de frontera, respectivamente).

La función Save toma como argumento el nombre del archivo que se quiere guardar (geom.i) y la función SavePart toma como argumentos, el nombre del archivo (prepro.geom) y el numero de triángulos (elementos de la región) requerido.

8.3. Solución de las ecuaciones

Ya que en los archivos de entrada también se definen las condiciones físicas del problema, las cuales son utilizadas por el pre-procesador para desarrollar la malla, se dispone de todos los elementos para obtener la solución al problema de transferencia de calor así que la solución de las

ecuaciones se limita a llamar a la herramienta disponible con los datos generados previamente. El manejo y la visualización de la solución obtenida se sale del dominio de éste trabajo[‡].

Eventualmente es posible, unir un programa para solucionar las ecuaciones o *solver* con las herramientas disponibles. Se pueden obtener ideas derivadas de la forma en que se utilizó el pre-procesador. Se puede escoger entre dos posibilidades: llamar a las librerias necesarias para definir un objeto que se cominicaría con el objeto de malla disponible para obtener la solución (recomendado) o llamar a un programa con el comando system,

```
system( "programa_de_solucion grid.grid archivo_de_salida.out" );
```

donde `programa_de_solucion` es el programa que obtiene y soluciona las ecuaciones en elementos finitos o *solver* y `archivo_de_salida.out` es el nombre del archivo generado que contiene la solución, en este caso, el perfil de temperaturas. Es necesario poner el programa de solución en el directorio del programa *elfin.exe* o instalarlo en el sistema.

[‡]para obtener las gráficas de temperatura se puede utilizar el programa plotmtv en UNIX/LINUX

Parte III

Resultados

CAPÍTULO 9

CONCLUSIONES

A partir de las metas trazadas inicialmente y el trabajo realizado se deriva una serie de conclusiones y observaciones de interés. Se obtuvo una interfaz de usuario gráfica (GUI) integrada al sistema operativo Windows con la cual se pueden plantear problemas de conducción de calor bidimensionales para que una herramienta de enmallado acoplada a la GUI obtenga el correspondiente archivo de *malla* que contiene información de la región definida, discretizada en *elementos finitos*. Finalmente la GUI se encarga de leer el archivo de malla y mostrar la región discretizada de tal forma que el usuario pueda modificar sus parámetros de forma interactiva mientras observa los efectos de las modificaciones.

Específicamente, se obtuvo una GUI como un programa ejecutable (.exe) que muestra una ventana principal (al estilo de los editores gráficos de dibujo) que tiene la capacidad de manejar múltiples archivos simultaneamente. Dibujando con el ratón dentro del área de dibujo de la ventana, el usuario puede crear las siguientes figuras geométricas:

- Líneas rectas.
- Arcos de elipse, con la posibilidad de modificar los ángulos inicial y final.
- Rectángulos.
- Círculos y elipses, con la restricción de que sus ejes son paralelos a los cartesianos.

Mas aún, éstas figuras se pueden intersectar, unir y cortar entre si para obtener una gran variedad de figuras nuevas, las cuales tambien se pueden guardar internamente como un objeto tipo poli-línea (por ejemplo los objetos, *Contorno* en el cuadro de diálogo *Región*).

El programa permite abrir y guardar la geometría ingresada gracias a las funciones incluidas en las clases MFC, lo cual se hace mediante los cuadros de diálogo habituales de Windows (*Abrir* y *Guardar como*). Los archivos se guardan con la extensión `.efn` y el sistema operativo asigna un ícono y un programa (elfin.exe) cada vez que encuentra un archivo con dicha extensión. Además se escriben los archivos `geom.i` y `prepro.part`, requeridos por la herramienta de enmallado (pre-procesador) y `grid.grid` generado con la misma.

No solo se escribe el archivo de malla, también se obtiene un *objeto* que permanece en la memoria durante la ejecución del programa. Dicho objeto de malla contiene una serie de variables, que se utilizan por ejemplo, para dibujar la malla en la ventana, y una serie de funciones que pueden utilizarse según la interfaz que haya dispuesto el autor del objeto (*Diffpack*).

Igualmente las condiciones de frontera permanecen como un objeto asociado a un cuadro de diálogo, así que se pueden utilizar permanentemente, el archivo `prepro.part` contiene la información de las condiciones de frontera utilizadas por el pre-procesador. Existen diferentes posibilidades para las condiciones de frontera que puede ofrecer la malla generada, las cuales deben ser exploradas con más detenimiento. Actualmente el programa permite las mas representativas que son: una temperatura fija o un flujo de calor fijo sobre un punto (nodo) y flujo de calor por convección fija a través de una cara.

El programa permite manejar los parámetros en tiempo real de forma rápida e intuitiva. También se elaboraron manuales de usuario y ejemplos ilustrativos para facilitar el aprendizaje del uso del programa.

9.1. Observaciones

En éste proyecto se logró obtener beneficio del convenio hecho entre la Universidad Industrial de Santander y Microsoft Corporation utilizando el paquete comprado por la Universidad para desarrollar un programa con arquitectura de documento múltiple y clases de MFC para el sistema operativo Windows, por esto el programa tiene la misma vigencia que el paquete (Entorno Integrado de Desarrollo, IDE) disponible en la Universidad a la fecha. Esto debe garantizar también que el código es compilable siempre que se disponga de la misma versión del paquete.

Se puede prescindir de importar otros formatos de archivos en éste trabajo ya que el programa tiene la capacidad de funcionar sin ésta utilidad, además implementarla pasaría por traducir la información incluida, especialmente las figuras, a las limitadas línea, arco, rectángulo, elipse y poli-línea perdiendo información valiosa por ejemplo de figuras tipo "*Bezier*" o "*BSplines*" a pesar de poder editar gráficamente las figuras restantes. Esto no resta importancia a la tarea de desarrollar una herramienta de intercambio de formatos, la cual por si misma, a juicio personal, es más que suficiente trabajo para otro proyecto de pregrado. Entre otras consideraciones, tal herramienta muy bien desarrollada integraría completamente el programa actual con el sistema operativo, y por ejemplo, bajo sistemas Linux se pueden aprovechar las utilidades puestas a disposición por el sistema operativo simplificando y asegurando enormemente el trabajo que podría realizarse. Utópicamente no existen límites respecto a los formatos de archivos que el programa podría manejar algún día como archivos de mallas y de soluciones extranjeros entre ellos algunos de marcas registradas.

El desarrollo de la interfaz gráfica se facilitó enormemente al trabajar en el IDE (Entorno Integrado de Desarrollo) Visual C++ de Microsoft ya que se pudieron utilizar las clases y funciones de MFC (*Microsoft Foundation Classes*) entre las que se incluyen ventanas, menús, cuadros de diálogo, botones, cursores, etc.

La construcción de una interfaz gráfica completa implica al menos dos etapas de desarrollo:

primero, se requiere crear una minima interfase pero que le permita al usuario manejar los eventos durante ejecución de manera interactiva y no secuencial. Segundo, se desarrollan diversos recursos, entre ellos los recursos gráficos para hacer la interacción del usuario con el programa mas cómoda e intuitiva.

Ya que la labor mas importante del ingeniero mecánico es el diseño de sistemas mecánicos, hoy en día es innegable la utilidad de las herramientas para CAD, tanto mas si se tiene en cuenta que estas han ampliado enormemente la capacidad de diseño ya que permiten que el ingeniero dedique sus esfuerzos mas al análisis de los resultados de un diseño que a su solución, evidentemente, esto también permite estudiar condiciones mas realistas.

Gracias al auge de las computadoras, un gran porcentaje de la población tiene una "intuición establecida" de algunas de las utilidades típicas de un programa (eventos del ratón o comandos como Ctrl-C, Ctrl-V, etc.) incluso comunes a diferentes sistemas operativos, con lo cual se hace mas fácil manejar un programa que ofrezca estas utilidades u otras similares. A pesar de que los elementos finitos fueron reconocidos como un método de gran alcance en ingeniería, solo recientemente se ha empezado a ahondar en su desempeño como herramientas, ya que su utilización siempre ha estado limitada por una complicada interacción con el usuario.

La programación orientada a objetos es el paradigma actual que ha impulsado la explosión de programas de alto nivel como éste, gracias a que permite manejar una gigantesca cantidad de líneas de código e incluir ordenadamente código creado por otras personas, logrando compartir los resultados y beneficiando a todos.

Entre los lenguajes orientados a objetos se destaca C++ el cual tiene la capacidad de incluir una sólida librería estándar y otras añadidas por el usuario. El lenguaje C++ es ampliamente utilizado para todo tipo de *software* especialmente el comercial (AUTOCAD, MATLAB, etc.), además tiene una característica especial: al ser un lenguaje de alto nivel, se puede usar para crear programas útiles en un amplio rango de *hardware*, no solo computadoras.

Uno de los errores mas habituales cuando se construye una herramienta o utilidad es no tener en cuenta el uso y el usuario final, en éste caso ese riesgo se reduce ya que tanto el autor como los usuarios finales son estudiantes de la misma escuela. Así, se puede considerar que un programa como el presentado resulta ser de mucha utilidad en la Escuela de Ingeniería Mecánica de la UIS y además cumple los requisitos para ser una herramienta computacional profesional.

Los objetivos del proyecto se cumplieron satisfactoriamente. La herramienta presentada genera archivos de malla los cuales se pueden utilizar con el paquete *Diffpack* creado por *Numerical Objects* de Noruega para obtener las respectivas soluciones. Además las mallas o regiones discretizadas se pueden graficar y sus parámetros se pueden modificar inmediatamente, así, un usuario puede utilizar el programa como una poderosa herramienta didáctica para el análisis de mallas.

9.2. Resultados obtenidos

Se obtuvo un programa con el cual se pudieron plantear varios problemas prácticos en poco tiempo evidenciando la utilidad de la herramienta propuesta.

La construcción de la herramienta mostrada resultó relacionada con una de las áreas de trabajo más actuales en ingeniería mecánica, la construcción de aplicaciones CAD, CAM, CAE, etc. A pesar de que la aplicación construida no pueda considerarse completamente acorde al estándar, es novedosa ya que logra la meta de crear un programa ejecutable integrado a un sistema operativo igual que cualquier editor gráfico de elementos finitos ofrecido en el comercio. La aplicación desarrollada tiene todas las utilidades típicas de un programa incluyendo el registro de los tipos de archivo.

9.3. Recomendaciones

En caso de continuar el trabajo realizado en este proyecto agregando herramientas y mejorando la funcionalidad del programa, se establecen ciertas recomendaciones nacidas de las experiencias.

Entre las mejoras pendientes que acepta el programa en futuros proyectos queda la utilidad para exportar e importar a otros formatos de archivos utilizados, lo cual le da un gran alcance. Se recomienda utilizar código libre antes que intentar escribir todas las traducciones posibles para evitar errores ocultos, también, es más fácil tener un formato estándar preferido (no *.efn) y utilidades de traducción confiables de y a este formato antes que tener una traducción independiente para cada pareja posible, sobre todo si hay un gran numero de formatos.

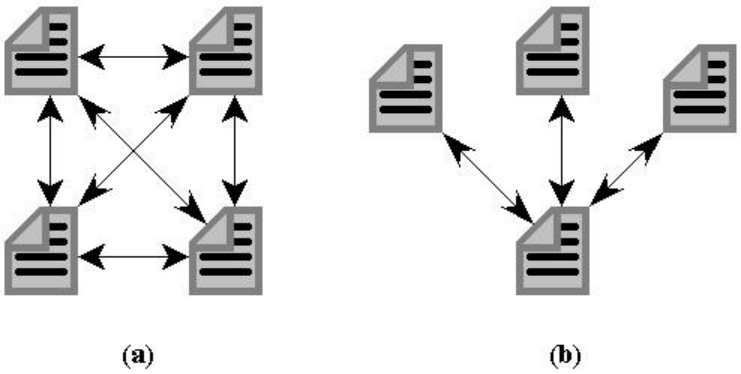

Figura 9.1: Traducciones necesarias (a) sin y (b) con formato preferido.

Una de las características que debe ofrecer un programa es la facilidad de uso, es decir que un usuario pueda utilizarlo intuitivamente en poco tiempo y en lo posible sin recurrir a ayudas, esto se hace muy importante ahora que el flujo de información en Internet ofrece programas desde cualquier parte del mundo haciendo que las personas eviten aprender información innecesaria acerca de como

utilizar un programa, ésta es la razón más importante para incluir una interfaz de usuario gráfica.

Se recomienda cambiar toda referencia a la clase Vector por su equivalente estándar, vector definido en la cabecera estándar <vector.h> ya que esto permitiría convertir otras clases en plantillas y cambiar, por ejemplo, los vectores x y y por el vector puntos. Si no se hace esto se debe tener cuidado al usar indices al recorrer los elementos de un objeto Vector.

Las herramientas facilitan el trabajo por lo cual una buena herramienta computacional debe simplificar, antes que dificultar, la labor del usuario, ocupándose correctamente de los detalles y ofreciendo una interfaz para el usuario, confiable y libre de errores bajo todas las condiciones de uso supuestas durante la etapa de diseño.

ANEXO A

MANUAL DE USUARIO

Con el código escrito en el capitulo 7 se obtiene el programa ***ElFin - Elementos Finitos***, el cual es un editor gráfico vectorial de figuras geométricas sencillas acoplado con un pre-procesador para enmallado. En éste capitulo se muestran las utilidades del programa *ElFin* relevantes al usuario final. A continuación se explican las herramientas disponibles, primero, se trata la interacción básica entre el programa y el usuario, luego se muestran las herramientas disponibles en el orden en el que son utilizadas normalmente.

El programa consiste en dos **etapas**, se inicia en la etapa de **edición** al abrir o crear un archivo nuevo, posteriormente el usuario accede a la etapa de **enmallado** para obtener la región discretizada o *malla*. Gran parte de la información se ingresa mediante cuadros de diálogo, los cuales tienen características típicas.

A.1. Entrada de datos

Existen cuatro formas de llamar a las funciones una vez que el programa está funcionando y se muestra la ventana principal: el menú de la ventana, la barra de herramientas, el menú de contexto y los aceleradores o combinaciones de teclas. En todos los casos las funciones se activan o desactivan dependiendo de la situación al momento de ser llamadas.

A.1.1. Barra de menú

En programas con arquitectura de documento multiple (ver glosario), es el lugar donde se recomienda poner todas las funciones para el usuario incluso sin estar habilitadas, ya que su ID será utilizado por otros recursos como la barra de tareas. Resulta mas conveniente ordenar la barra de menú tradicionalmente y como recomiendan. Cada entrada del menú contiene una letra para acceso rápido con la tecla Alt y el acelerador correspondiente a la función,

El menú *Archivo* contiene las utilidades características del programa y una lista de sus ocho archivos mas recientes.

Figura A.1: Barra de menú.

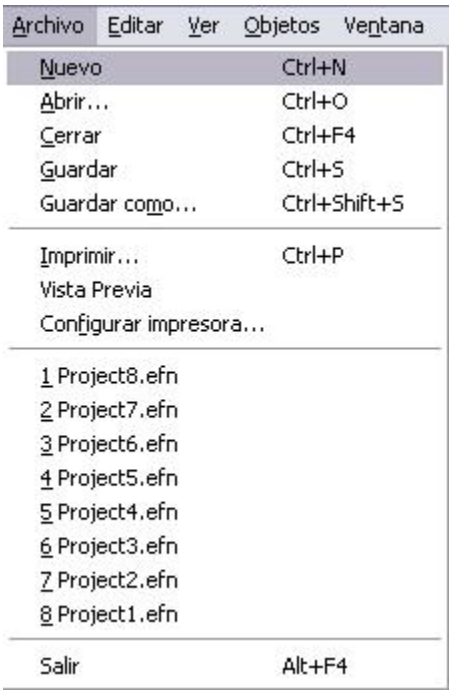

Figura A.2: Menú Archivo.

El menú *Editar* contiene opciones básicas de edición, además se utiliza para acceder a los diferentes modos de trabajo (etapas) y a los cuadros de diálogo.

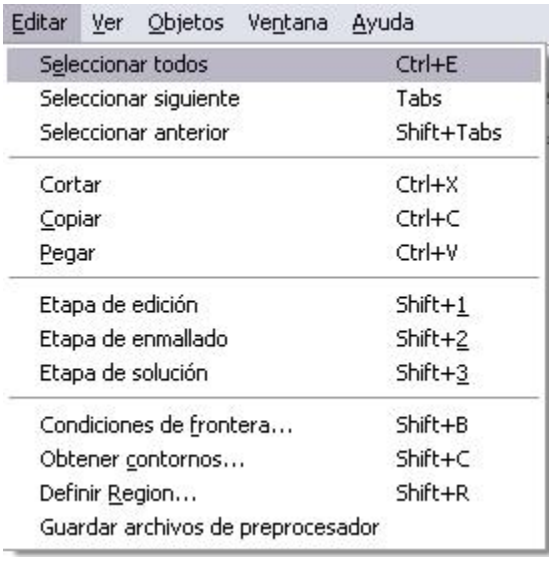

Figura A.3: Menú Editar.

El menú *Ver* permite definir la ampliación (*zoom*) de la página y mostrar u ocultar la cuadricula, la margen, la barra de estado y la barra de herramientas. También se accede al cuadro de diálogo "Preferencias".

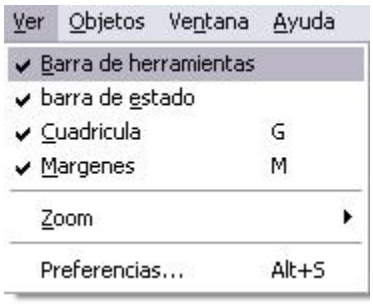

Figura A.4: Menú Ver.

El menú *Objetos* permite agregar las diferentes figuras, fijar los objetos a la cuadricula (*snap*), agrupar, desagrupar y ordenar las figuras. Además incluye las funciones Dividir, Segmento y Recortar. Si se selecciona un objeto, desde aquí se accede al cuadro de diálogo propiedades del mismo.

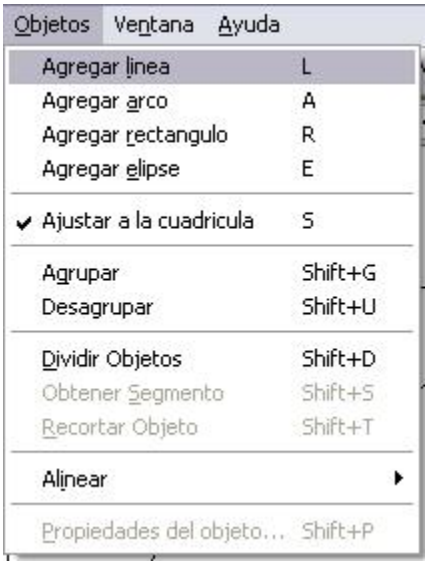

Figura A.5: Menú Objetos.

El menú *Ventana* maneja las ventanas de una aplicación de documento multiple.

El menú *Ayuda* muestra el cuadro de diálogo "Acerca de". Observe que en todos los menús, un cuadro de diálogo se denota agregando tres puntos "..." al final.

A.1.2. Barra de herramientas

Está ubicada debajo de la barra de Menú por defecto, pero se puede poner como una ventana flotante, se puede ver/ocultar en el Menú Ver. Contiene botones para acceder a las funciones mas

Figura A.6: Menú Ventana.

Figura A.7: Menú Ayuda.

importantes, entre las que se encuentran agrupadas las herramientas (funciones) del archivo, botones de visualizacion, botones para agregar figuras geometricas, botones de edición (las funciones dividir, recortar, segmento, agrupar, desagrupar y alinear) asi como un grupo de botones para dibujar otros objetos. Los botones tienen diagramas en alto contraste con un *tool-tip* que se despliega cuando el ratón esta sobre ellos.

Figura A.8: Botones de la barra de herramientas.

A.1.3. Menú de contexto

Es un menú que se despliega cuando se da click con el botón derecho del ratón sobre una figura, contiene un grupo de funciones para copiar, pegar y duplicar el objeto, otro para ordenar el objeto respecto a los demás y una función para acceder al cuadro de diálogo propiedades del objeto.

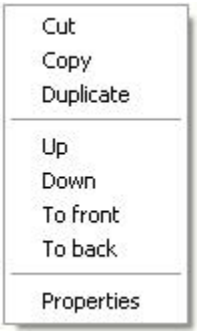

Figura A.9: Menú de contexto dando clic derecho sobre una figura.

A.1.4. Aceleradores

Los aceleradores son la forma mas rápida de utilizar un programa ya que llaman a una función cuando la persona oprime una o mas teclas del teclado. No todas las funciones tienen acelerador y no necesariamente al usar una tecla se usa un acelerador. En la siguiente tabla se muestran los aceleradores disponibles.

Comando	Acelerador	Comando	Acelerador
Nuevo documento	Ctrl+N	Acercar Zoom	+
Abrir documento	Ctrl+O	Alejar Zoom	-
Guardar	Ctrl+S	Mostrar/Ocultar cuadricula	G
Guardar como	Shift+Ctrl+S	Mostrar/Ocultar margen	M
Imprimir	Ctrl+P	Activar/Desactivar *snap*	S
Salir	Alt+F4	Agregar linea	L
Cortar	Ctrl+X	Agregar arco	A
Copiar	Ctrl+C	Agregar rectángulo	R
Pegar	Ctrl+V	Agregar elipse	E
Seleccionar todos	Ctrl+E	Agregar polilinea	P
Seleccionar siguiente	Tabs	Dividir objeto	Shift+D
Seleccionar anterior	Shift+Tabs	Obtener segmento del objeto	Shift+S
Etapa de edición	Shift+1	Recortar objeto	Shift+T
Etapa de enmallado	Shift+2	Agrupar objetos	Shift+G
Diálogo Condiciones de frontera	Shift+B	Desagrupar objetos	Shift+U
Diálogo Contornos	Shift+C	Borrar objetos	Delete
Diálogo Región	Shift+R	Duplicar objeto	Insert
Diálogo Propiedades del objeto	Shift+P	Diálogo Preferencias	Alt+S

Tabla A.1: Aceleradores

A.2. Salida de datos

El programa utiliza tres canales de salida: la pantalla, la impresora y los archivos en disco. Se evitaron los sonidos u otras formas de suministrar información.

A.2.1. La pantalla

El programa utiliza los recursos visuales que el sistema operativo pone a su disposición, normalmente, un programa solo debe dibujar en un area limitada del escritorio conocida como el area cliente (Ver figura A.10). El area cliente dispone de una barra de estado ubicada en el extremo inferior, reglas de referencia, horizontal y vertical, la ventana de dibujo y un area en la esquina superior izquierda donde se escogen las unidades.

La barra de estado contiene tres campos con información de relevancia, además del habitual, el primero desde la izquierda contiene información acerca de los objetos seleccionados y totales, si hay un elemento seleccionado también muestra el numero de intersecciones y los caracteres "<" y/o ">" si el objeto esta intersecando sobre su punto inicial o final (líneas y arcos). El segundo campo muestra información geométrica del objeto seleccionado con el formato, [izquierda superior] [derecha inferior], a la derecha también pueden aparecer las letras "D" si el objeto tiene igual ancho que alto y "z" si el objeto tiene tamaño nulo.

Las reglas sirven de referencia para la ventana de dibujo, haciendo click en la región de la esquina superior izquierda se muestra un cuadro de diálogo donde se fijan sus unidades.

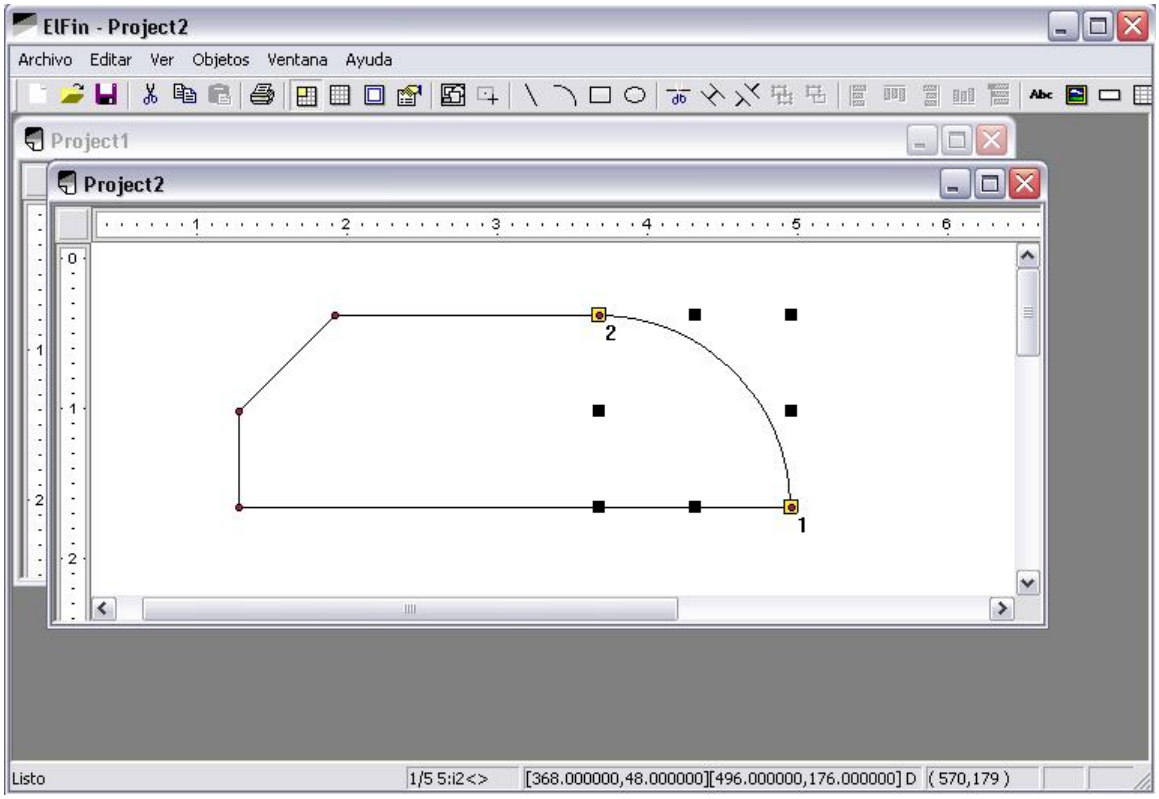

Figura A.10: Ventana de *ElFin* con dos documentos abiertos.

A.2.2. La impresora

El programa tiene una interfase con la API del Sistema Operativo mediante la cual puede utilizar una impresora. Para poder utilizar esta herramienta es necesario disponer de una impresora instalada, también es posible instalar una impresora tipo "Generic"para formato PS (postscript) para guardar un archivo en el disco.

A.2.3. Los archivos

El programa permite abrir y guardar archivos con la extension `.efn` y que contienen la información geométrica de los modelos. Si el usuario necesita los archivos de entrada del pre-procesador puede copiar los dos archivos llamados `geom.i` y `prepro.part` desde la carpeta donde se encuentra el programa. Igualmente puede copiar la región discretizada o malla generada que se encuentra en el archivo `grid.grid` para utilizarla con el paquete *Diffpack*.

A.3. Etapas

Al utilizar un programa de elementos finitos, los datos se manejan en forma secuencial ya que la geometría introducida tiene que ser verificada antes de utilizarse (ver figura A.11). Para realizar esta verificación se definen las etapas de trabajo por las que tiene que pasar un usuario del programa.

Este programa tiene dos etapas, la etapa de edición y la etapa de enmallado*. Al iniciar el programa, este se encuentra en la etapa de edición, el usuario debe dibujar una geometria que describa una región cerrada y pasar a la etapa de enmallado, donde se construye una región discretizada a partir de la geometría.

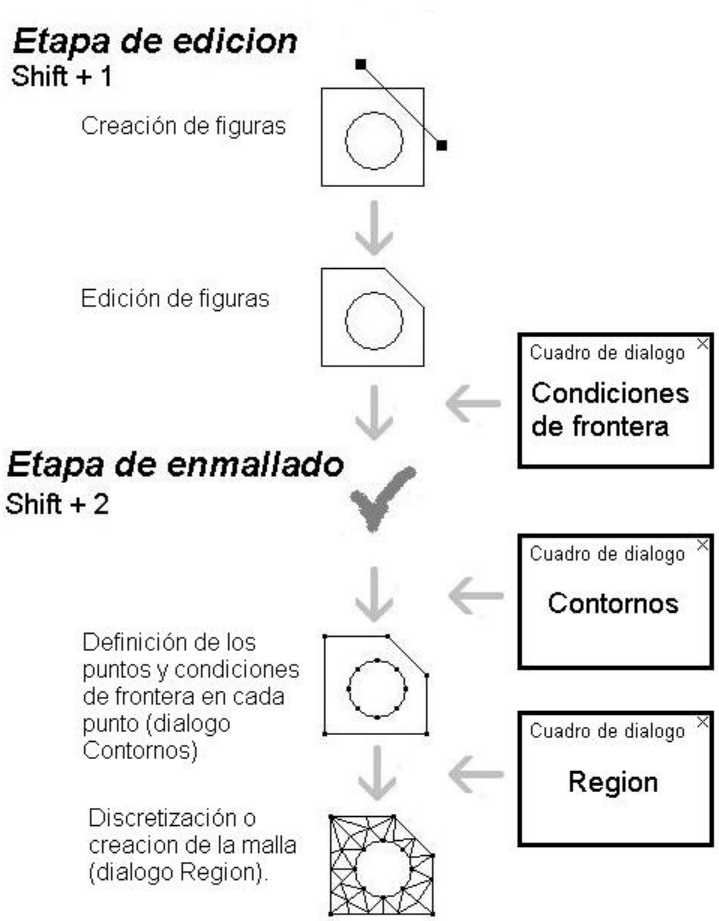

Figura A.11: Etapas consecutivas. Para obtener cada paso, antes se debe suministrar información en un cuadro de diálogo. Al pasar a la etapa de enmallado se verifica que la geometría ingresada define una región cerrada.

Inicialmente se parte de la etapa de edición y se deben construir una serie de contornos cerrados que se verifican al pasar a la etapa de enmallado. Para pasar a la etapa de enmallado se utilizan las teclas `Shift+2` o el menu Editar, para regresar a la etapa de edición (por ejemplo para modificar la geometría) se utilizan las teclas `Shift+1` o el menu Editar.

* al obtener la malla, están dadas las condiciones para pasar a una tercera etapa, la solución.

A.3.1. Etapa de edición

En esta etapa el usuario debe definir toda la geometría del problema planteado, para esto puede dibujar y editar las figuras geométricas disponibles. Entre las utilidades mas importantes que ofrece el programa se encuentra la edición de las figuras mediante las funciones Dividir, Segmento y Recortar para obtener areas mas complejas. Lo que finalmente se debe obtener es uno o mas conjuntos de objetos conectados en parejas por sus puntos extremos, de esta manera se define correctamente una región cerrada, un objeto de tipo elipse o rectángulo es también un contorno cerrado. No es necesario dibujar los objetos en un orden especifico respecto a los puntos de conexión. Inicialmente el programa se encuentra en la etapa de edición, esta también se puede acceder con las teclas `Shift+1`.

A.3.1.1. Creación y edición de figuras

El programa dispone de las herramientas usuales de un editor gráfico de figuras entre las que se encuentran: dibujar y modificar el tamaño y la posición de líneas, arcos de elipse, rectángulos y elipses cerradas. Se puede modificar la mayoría de los colores u otras características y también acercar, alejar o escoger un punto de vista determinado.

Para crear un una nueva figura se debe llamar a la correspondiente función (linea, arco, etc.), lo cual se puede hacer mediante la barra de menú de la ventana, desde los botones de la barra de herramientas o utilizando los aceleradores con el teclado. Al llamar a una de las funciones de dibujo, el cursor cambia a una figura en forma de cruz, el usuario debe ubicarse en donde desea que se encuentre la esquina superior izquierda[†] de la figura que desea crear, luego se debe oprimir el botón izquierdo del ratón y arrastrar (mantener oprimido) hasta la esquina opuesta (la inferior derecha). Si se tiene activada la opción *Ajustar a la cuadricula*, las esquinas quedaran ubicadas sobre los puntos de la cuadricula. Después de crear una figura se puede seleccionar con el ratón o con la tecla *tabs* y se puede mover con las flechas o arrastrando la figura con el ratón.

Además de las funciones básicas mencionadas, el programa cuenta con funciones avanzadas de edición que consisten en dividir los objetos a través de sus puntos comunes de intersección. Estas funciones se utilizan para generar correctamente geometrías complejas y se explican en detalle mas adelante.

A.3.1.2. Editar un arco

La figura arco se puede editar manualmente no solo para cambiar de tamaño o posición, también se puede modificar el ángulo del arco tendido sobre una elipse constante,

Para cambiar los ángulos inicial y final del arco, este debe estar seleccionado (tecla *tabs*), a continuación ponga el cursor del ratón sobre uno de los dos cuadrados amarillos, la forma del cursor debe cambiar a una flecha recortada. Finalmente mueva el cursor dentro de la ventana para modificar el arco.

[†]puede ser cualquier esquina, ya que las figuras se pueden invertir vertical y horizontalmente.

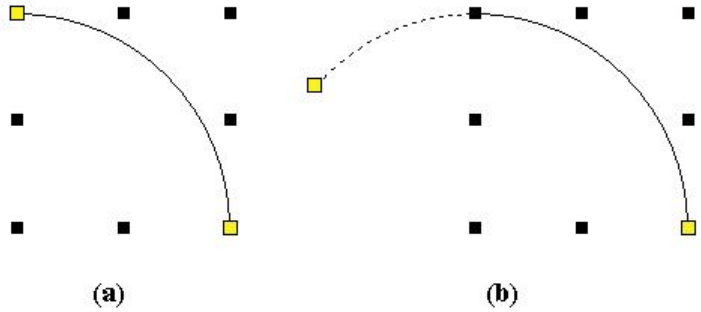

Figura A.12: Edición del arco, (a) al situarse sobre un cuadro y (b) moviendo el punto.

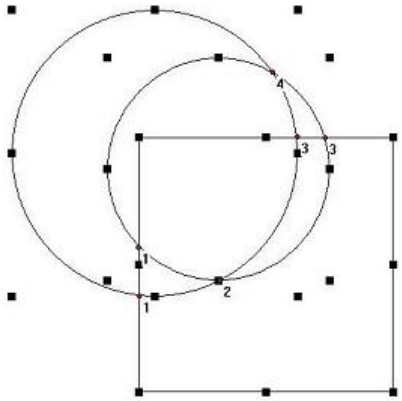

Figura A.13: Seleccionar todas las figuras antes de oprimir el botón Dividir.

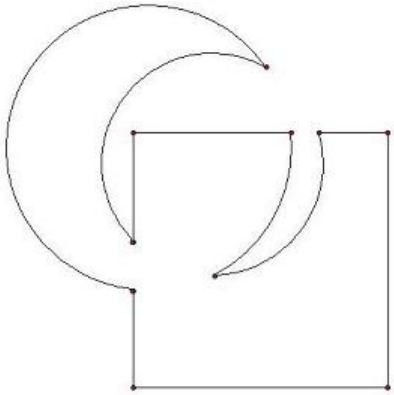

Figura A.14: Después de dividir y borrar los segmentos sobrantes (seleccionar con *Tabs*).

A.3.1.3. La función Dividir

El programa ofrece tres utilidades para editar una figura y convertirla en otra u otras a partir de sus puntos de intersección que son los funciones **Dividir**, **Segmento** y **Recortar**. con estos comandos el usuario tiene la posibilidad de obtener cualquier area definida por dos o mas figuras simplemente dividiendo las figuras y eliminando los segmentos innecesarios.

El método mas seguro para obtener un contorno cerrado es escoger simultáneamente todas las figuras a dividir y llamar a la función Dividir (Shift+D). Luego se recorren todos los objetos utilizando las teclas Tabs o Shift+Tabs para eliminar los segmentos sobrantes. Esto se debe a que es muy difícil utilizar las funciones Recortar y Segmento sin mover alguna de las figuras con el ratón.

Para llamar a estas utilidades primero se debe escoger un segmento (con intersecciones), La función Dividir, divide todos los objetos seleccionados por sus puntos de intersección.

A.3.1.4. Las funciones Segmento y Recortar

Las funciones Segmento y Recortar son un poco mas complejas, primero se debe seleccionar un elemento, a continuación se llama a la función Segmento o Recortar. El usuario pasa a un modo interactivo de edición en el cual tiene que ubicar el cursor sobre algún segmento del objeto seleccionado y escogerlo con el botón izquierdo del ratón.

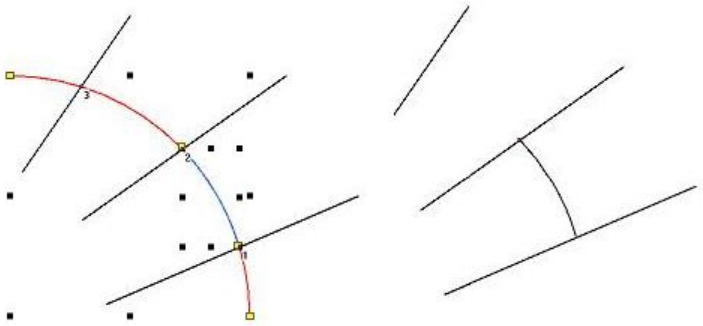

Figura A.15: Utilizando la función Segmento. (a) En modo interactivo. (b) Resultado.

En la función **Segmento** (ver figuras A.15 (a) y (b)) se elimina el objeto inicial y se reemplaza por el segmento seleccionado. Primero se debe escoger un objeto con puntos de intersección, a continuación se llama a la función segmento mediante el menu Objetos, con las teclas `Shift+S` o el botón en la barra de herramientas.

Inmediatamente, la figura se dibuja de color rojo y el usuario pasa a un modo interactivo en el cual, el programa le solicita escoger con el ratón uno de los segmentos que conforman la figura, al oprimir el botón izquierdo del ratón sobre un segmento, se borran todos los segmentos restantes del objeto y se termina el modo interactivo.

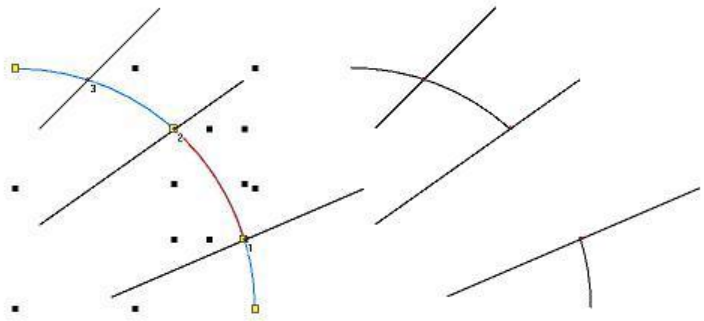

Figura A.16: Utilizando la función Recortar. (a) En modo interactivo. (b) Resultado.

En la función **Recortar** se elimina el segmento seleccionado y se conserva el resto de la figura. Igual que con la función Segmento, se debe escoger un objeto con puntos de intersección, a continuación se llama a la función mediante el menu Objetos, con las teclas `Shift+T` o el botón en la

barra de herramientas.

Inmediatamente, la figura se dibuja de color azul claro y el usuario pasa a un modo interactivo en el cual, el programa le solicita escoger con el ratón uno de los segmentos que conforman la figura, al oprimir el botón izquierdo del ratón sobre un segmento, se borra el segmento seleccionado y se termina el modo interactivo.

A.3.1.5. Diálogo Preferencias

Es el lugar donde el usuario puede escoger las características generales de la ventana como los colores, la vista (*zoom*), la cuadricula y la margen. Las diferentes opciones se muestran agrupadas, después de ser modificadas, el usuario debe oprimir el botón *Aceptar* o el botón *Cancelar*, ubicados en el extremo inferior izquierdo del cuadro (ver figura A.17) para hacer efectivos o ignorar los cambios realizados y regresar a la ventana de dibujo.

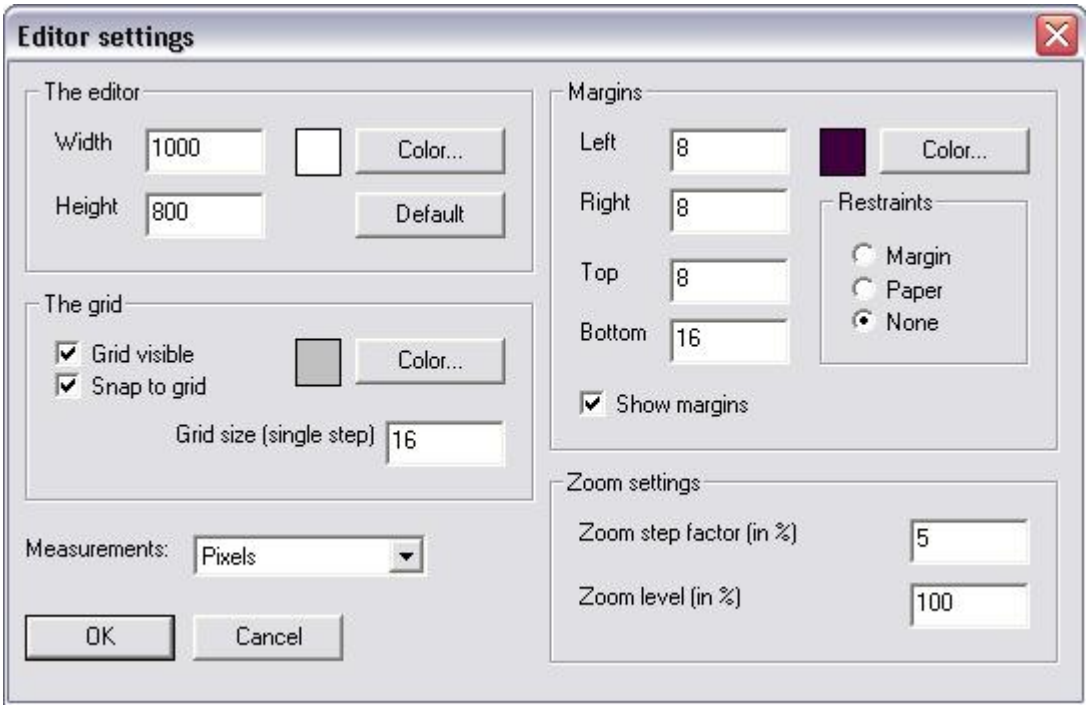

Figura A.17: Diálogo Preferencias.

El primer grupo incluye las opciones para personalizar el color de fondo y el tamaño del area de dibujo. El segundo grupo modifica la cuadricula, su color, el tamaño del paso (de un cuadro) y tiene dos botones de verificación, con el primero se muestra o se esconde la cuadricula y con el segundo se escoge si se utiliza o no la herramienta para forzar a los objetos a tomar el tamaño definido por la cuadricula (Herramienta Ajustar a la cuadricula). El tercer grupo especifica el tamaño y apariencia de la margen, también si se muestra o no. Por ultimo, el cuarto grupo define el aumento por cada acercamiento (Tecla +, o rueda del botón intermedio del ratón) en porcentaje, y el acercamiento (*zoom*) utilizado en porcentaje. Este cuadro se puede llamar en cualquier etapa (edición o enmallado) con las teclas `Alt+S` o desde el menu edición.

A.3.1.6. Diálogo Condiciones de frontera

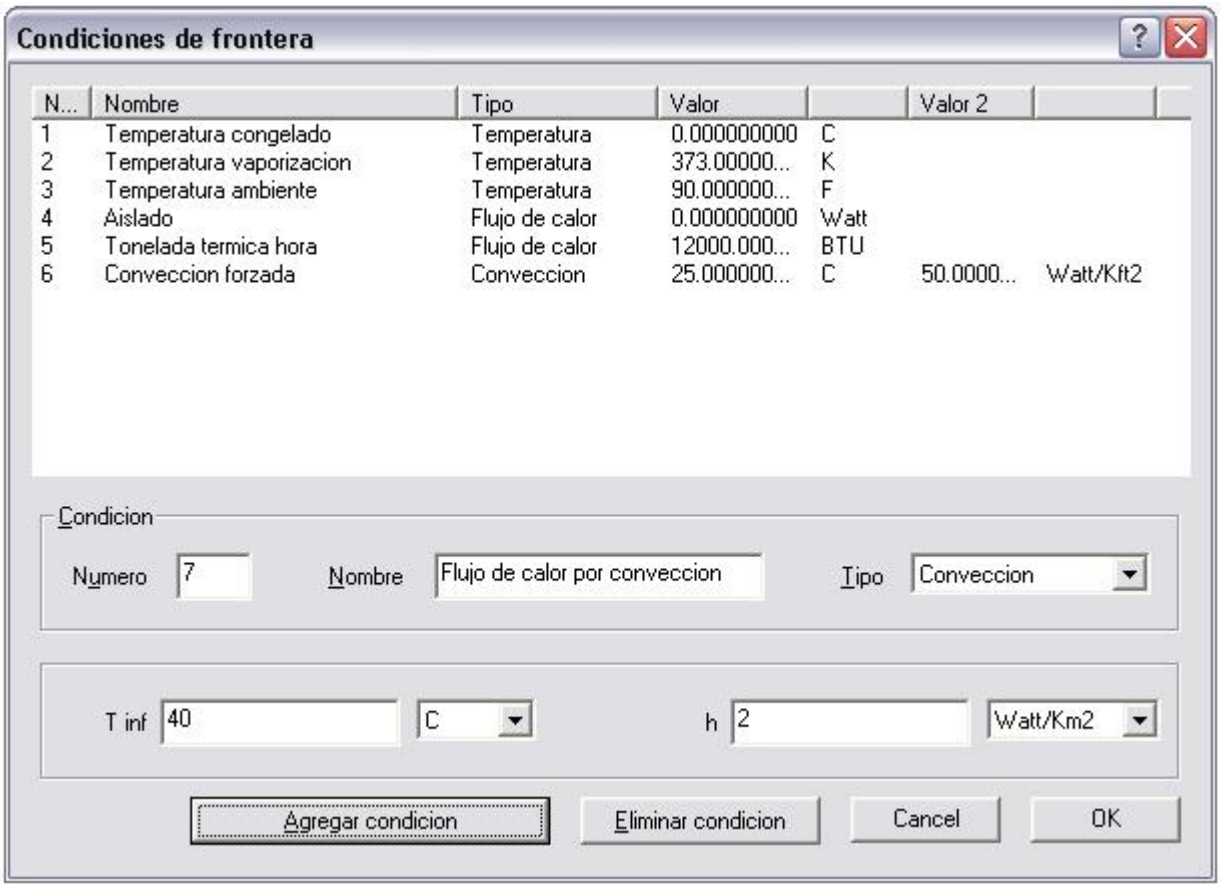

Figura A.18: Diálogo Condiciones de frontera.

Al igual que el diálogo Preferencias, éste diálogo se puede acceder desde cualquier etapa (utilizando las teclas `Shift+B`). La información introducida en este cuadro de diálogo consiste en las condiciones físicas de frontera del problema (Dirichlet o Neumann). El cuadro muestra una lista de selección única (solo se puede seleccionar un elemento a la vez) que se encuentra en la mitad superior (ver figura A.18) y en donde se muestran todas las condiciones de frontera definidas para el problema, en la parte inferior se muestran los diferentes campos que se deben llenar al definir una nueva condición, estos son: Numero, debe ser positivo y no repetirse; Nombre, para uso del usuario; Tipo, uno de los tres tipos que se explican mas adelante; Valor o valores numéricos junto con las unidades utilizadas. Los últimos campos solo se muestran cuando se selecciona el tipo de condición. El programa permite tres tipos de condiciones:

- **Temperatura.** Se fija la temperatura que tendrán los nodos (puntos) con ésta condición.
- **Flujo de calor.** Se fija una fuente o sumidero de calor que puede estar distribuido en un area, un contorno o un punto, actualmente el programa esta restringido a fuentes puntuales de calor sobre los nodos por lo cual es necesario obtener el calor total equivalente por cada nodo cuando el calor esta distribuido sobre un contorno o una superficie.
- **Flujo de calor por convección.** Se relacionan el flujo de calor y la temperatura sobre un

contorno de la región. En este caso se requieren dos valores: el coeficiente de conveccion y la temperatura del medio junto con las respectivas unidades de cada uno.

Cuando se ingresa una sola condición de frontera, ésta se guarda al oprimir el botón Aceptar. Si se desea agregar mas condiciones se oprime el botón agregar condición con lo cual se guardara un nuevo item en la lista con los valores ingresados. El numero de la condición aumenta automáticamente una unidad al ingresar una nueva condición. Si se desea eliminar una condición se debe escoger de la lista y se debe oprimir el botón eliminar condición. Finalmente, oprimir el botón Aceptar.

A.3.2. Etapa de enmallado

Al acceder a la etapa de enmallado se deben tener uno o mas contornos cerrados que no se intersecten entre si, que se mostraran de color verde, en caso contrario los contornos incorrectos se señalan con otro color (amarillo). Cuando se pasa de la etapa de edición a la etapa de enmallado, los objetos se guardan y no se pueden modificar hasta que se regrese a la etapa de edición.

En la etapa de enmallado se trabaja con los cuadros de diálogo Condiciones de frontera, Contornos y Región. Se accede a la etapa de enmallado mediante las teclas `Shift+2`.

A.3.2.1. Diálogo Contornos

El diálogo edición de contornos, o contornos (`Shift+C`) define los puntos utilizados por el pre-procesador para generar la malla a partir de los contornos cerrados disponibles. Inicialmente, el usuario se encuentra en la etapa de edición, allí, debe dibujar una serie de figuras y verificar que sean cerradas, es decir que los diferentes segmentos de un contorno (las figuras) se encuentren correctamente conectadas (ver secciones anteriores), entonces, al pasar a la etapa de enmallado (`Shift+2` o el menu edición), el programa crea una serie de contornos (de color verde) que se muestran en la lista del cuadro de diálogo Contornos (ver figura A.19).

Por lo expuesto anteriormente, el cuadro de diálogo Contornos (y el cuadro de diálogo Región) solo se pueden acceder desde la etapa de edición. El cuadro se compone de una lista de selección multiple, es decir que se pueden seleccionar varios elementos para ser modificados simultáneamente (se seleccionan varios elementos manteniendo oprimidos los botones `Ctrl` o `Shift`) cada una de las filas muestra un segmento perteneciente a alguno de los contornos (una de las figuras dibujadas), el primer campo es un índice, el segundo contiene el nombre del segmento (si el usuario fijo algún nombre en la etapa de edición) o por defecto, el nombre de la figura, además se muestra el contorno al que pertenece. El tercer campo contiene las condiciones de frontera asignadas los puntos del segmento y los campos cuarto y quinto contienen la información de los puntos.

El usuario debe seleccionar uno o mas segmentos de la lista, en la lista desplegable *Condición*, se selecciona la condición de frontera (una de las condiciones previamente incluidas en el cuadro Condiciones de frontera, sección anterior) que se desea asignar sobre los segmentos y se oprime el botón *Aplicar condición*. Luego se define el numero de puntos sobre los contornos. El usuario puede escoger entre definir una distancia (aproximada) entre los puntos o definir un numero total de puntos sobre el segmentos, en el primer caso, el usuario debe fijar también las unidades de la

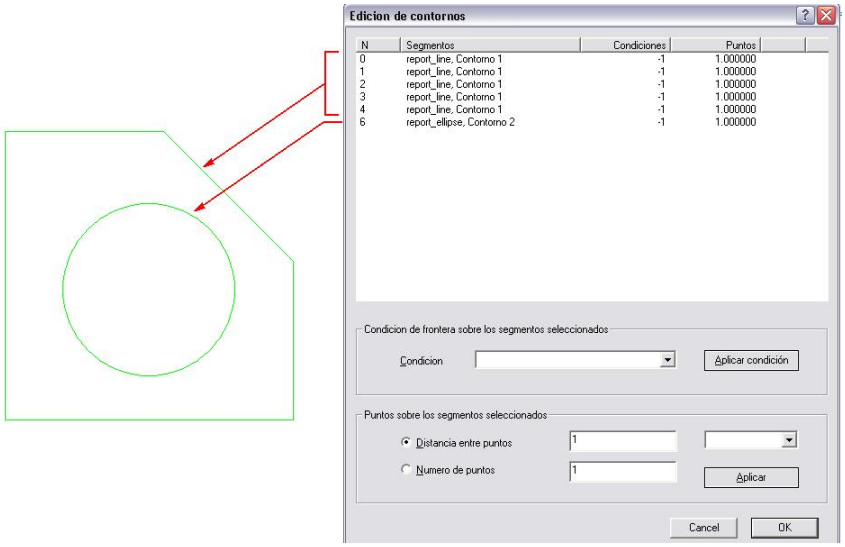

Figura A.19: Diálogo Contornos.

distancia entre puntos, luego debe oprimir el botón *aplicar* para modificar la lista. Finalmente al oprimir el botón *Aceptar*, se dibujan los nuevos puntos sobre los contornos.

A.3.2.2. Diálogo Región

En la etapa de enmallado, y después de definir los puntos en el cuadro de diálogo Contornos, se debe llamar al cuadro de diálogo Región (`Shift+R`), donde se especifícan los últimos detalles de la región y se llama al pre-procesador para generar la malla o región discretizada.

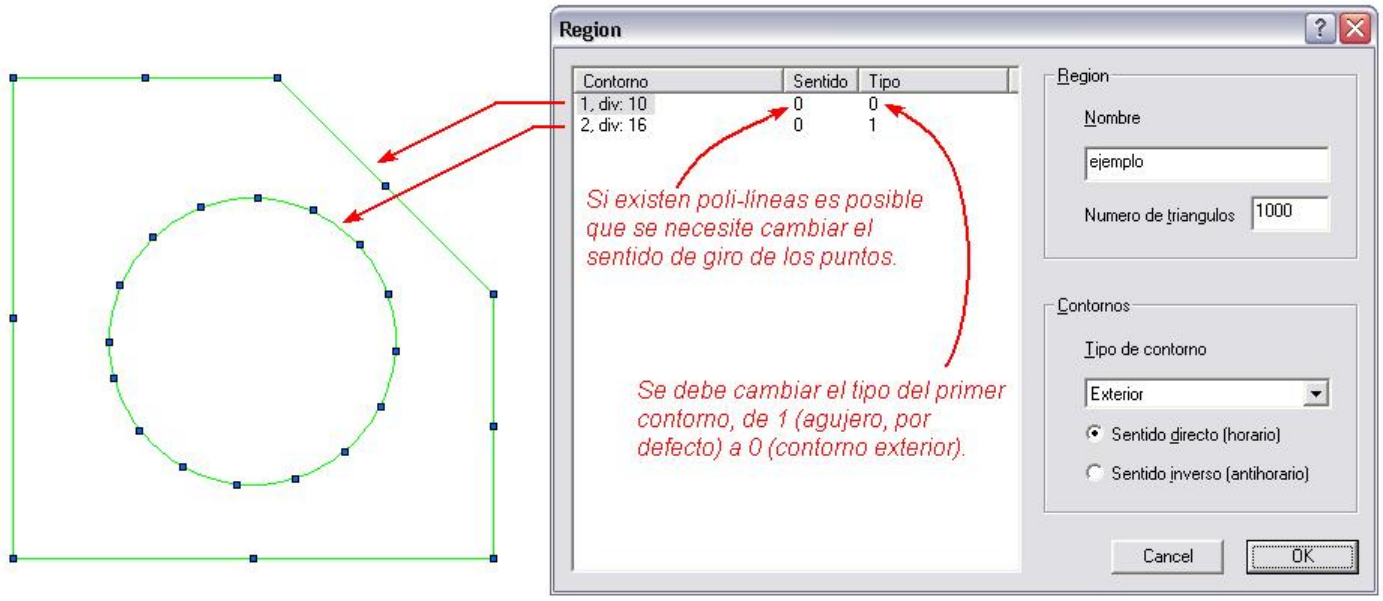

Figura A.20: Diálogo Región.

Al abrir el cuadro de diálogo Región, se muestra una lista de selección única con los contornos (no los segmentos como en el cuadro Contornos) disponibles. Automáticamente se asigna un sentido y un tipo a cada contorno (agujero). El sentido se puede modificar en los botones de selección ubicados en la parte inferior izquierda, esto puede ser necesario en caso de tener un contorno formado can varios segmentos, **no es necesario modificar el sentido de los contornos que consisten en un rectángulo o una elipse**, en caso de que los contornos resulten definidos en sentido anti-horario.

Se asigna el tipo *agujero* a todos los contornos ya que normalmente la mayoría de los contornos representan agujeros. **Obligatoriamente el usuario debe modificar el tipo de al menos un contorno, aquel que va a ser el contorno exterior**, también existe un tercer tipo, el agujero de interfase, que representa una región llena sobre la cual se desea hacer un refinamiento de la malla.

Al oprimir el botón *Aceptar* se debe mostrar la región generada o malla, en caso de error se muestran las flechas (rojas) que indican la dirección de la malla para que el usuario verifique que todas apuntan en sentido horario, (si no, esta es la causa del error). Si todas las flechas apuntan en sentido horario, la causa del error es otra, por ejemplo que el numero de puntos o el numero de elementos es demasiado grande.

ANEXO B

EJEMPLOS

En éste capítulo se presenta una serie de ejemplos estilo tutorial que pueden realizar los usuarios del programa **ElFin - Elementos Finitos**, éstos se incluyen en la ayuda del programa.

Los ejemplos del programa se clasifican por el nivel requerido y el tiempo de realización, después de realizar los ejemplos sencillos, un usuario debe estar en la capacidad de seguir con los avanzados.

B.1. Simple

Nivel: Básico
Objetivos:

- Dividir objetos.
- Definir condiciones de frontera.
- Definir una región.
- Obtener una región discretizada.

Tiempo: 15 min.
Procedimiento:

El primer paso es crear un nuevo proyecto vacio, si no se tiene una ventana del programa abierta, se debe iniciar el programa pulsando doble clic o *Enter* sobre el icono del archivo ejecutable (.exe) o el acceso directo (.lnk) disponible[*], al iniciar el programa se muestra un proyecto nuevo. Si ya existe una ventana del programa abierta, oprima `Ctrl+N` o el menú Archivo > Nuevo, para abrir un nuevo proyecto dentro de la ventana existente.

[*] los accesos directos incluidos conducen a una copia que se encuentra en el directorio del programa VisualStudio, `/MyProjects/ElFin-1.0.2.5/Release`.

Seleccione la opción ajustar a la cuadricula activo, esto se hace desde el menú Ver, con la tecla `S` o con el botón de la barra de herramientas. El botón correspondiente debe mostrarse hundido como en la figura. Dibuje un rectángulo seleccionando del menú Objetos > Agregar rectángulo, pulsando la tecla `R` o con el botón de la barra de herramientas "Rectángulo", luego arrastre el ratón a través del area de dibujo. Igualmente, dibuje una línea que corte al rectángulo tal como en la figura B.1.

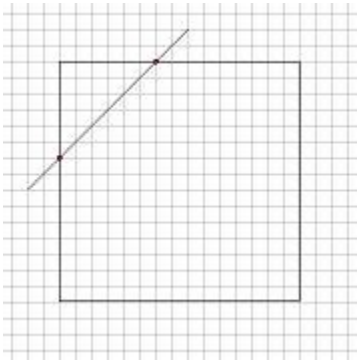

Figura B.1: Ejemplo 1. Rectángulo y línea.

Seleccione ambos objetos ya sea arrastrando con el ratón un area de selección que contenga una parte de cada uno, desde el menú Editar > Seleccionar todo, o mediante las teclas `Ctrl+E`. Oprima el botón Dividir, el menú Objetos > Dividir o las teclas `Shift+D`. A continuación escoja los segmentos libres y eliminelos usando la tecla `Delete` o el botón cortar de la barra de herramientas. Si se escogen los objetos pulsando repetidamente la tecla `Tabs` en vez de utilizar el ratón, se elimina el riesgo de mover accidentalmente las figuras. La geometría resultante debe ser como la mostrada en la figura B.2.

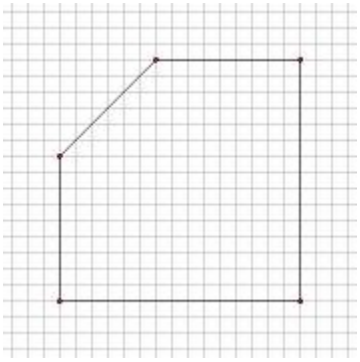

Figura B.2: Ejemplo 1. Figuras obtenidas al dividir el rectángulo y la línea.

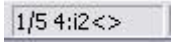

Figura B.3: Una figura conectada en ambos extremos.

Ya se dispone de la geometría necesaria para pasar a la segunda etapa, antes de esto el usuario puede, si lo considera conveniente, revisar los objetos seleccionándolos con la tecla `Tabs` (`Shift+Tabs` para recorrer las figuras al revés) y observando la barra de estado ubicada en el extremo inferior de la ventana, todos los objetos deben tener dos intersecciones ubicadas en el punto inicial y final del objeto, en tal caso, con todas las figuras deben mostrarse los siguientes caracteres en el segundo campo

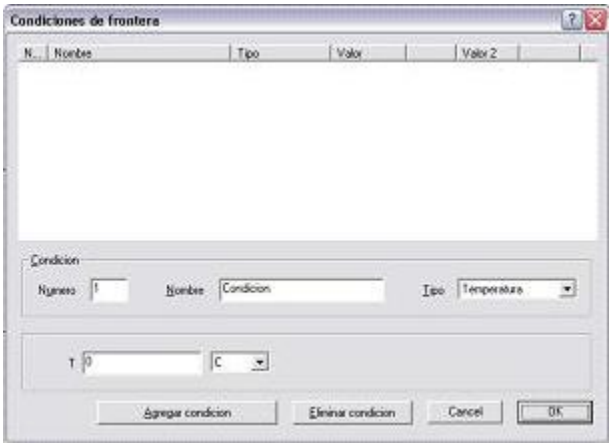

Figura B.4: Ejemplo 1. Diálogo Condiciones de frontera.

de la barra de estado (ver figura B.3) cambiando solamente el numero antes de ":", en la figura se encuentra escogido el objeto numero 4 del contenedor. Los caracteres "i2" significan que el objeto tiene dos puntos de intersección y los caracteres "<" y ">" significan que éstos se encuentran en sus puntos extremos inicial y final respectivamente. Recuerde que se debe tener cuidado de no intentar escoger los objetos haciendo clic con el ratón sobre ellos ya que se pueden mover y perder los puntos de intersección.

A continuación se definen las **condiciones de frontera** (este paso se puede hacer en cualquier momento durante la ejecución del programa). Con las teclas `Shift+B` o en el menú "Editar > Condiciones de frontera...", se muestra el cuadro de diálogo correspondiente. Se deben ingresar los datos en los campos de texto ubicados en la parte inferior del cuadro (figura B.4). El número de cada condición debe ser positivo y no repetirse, en este campo se deja el valor por defecto (1), se escoge un nombre para la condición y se define el tipo de condición del cuadro desplegable (temperatura) con lo cual se muestran los ultimos campos de texto donde se ingresa el valor de la temperatura (0) y sus unidades (grados C), después de llenar los campos se pulsa OK o `Enter`, con lo cual la información ingresada se agrega a la lista (en este caso, vacia) mostrada en la parte superior del cuadro de diálogo.
Si se desean incluir varias condiciones de frontera, se utiliza el botón Agregar condición, si se desea eliminar alguna de ellas se utiliza el botón Eliminar condición. No se puede repetir el número que define una condición de frontera. Pasamos a la **etapa de enmallado**, esto se hace pulsando `Shift+2`, o en el menú "Editar > Etapa de enmallado". Si tenemos éxito, el contorno cerrado debe cambiar su color a verde, pero si la geometría no esta cerrada se mostrará con color amarillo, en cualquier caso ya podemos definir los puntos de la región.
Para definir los puntos de la región así como las condiciones de frontera a las que esta sometido cada punto, abrimos el cuadro de diálogo **Edición de Contornos** usando las teclas `Shift+C` o desde el menú "Editar > Obtener Contornos...", ver figura B.5. El cuadro de diálogo Edición de contornos contiene una lista con los segmentos de la lista. Con todos los objetos seleccionados escoja una de las condiciones mostradas en el primer cuadro combinado y oprima el botón Aplicar condición. Aun con todos los objetos seleccionados escoja la opción "Numero de puntos" y escriba 2. oprima el botón Aplicar. Finalmente el diálogo debe verse como la figura B.5.

Al pulsar el botón OK se deben mostrar los puntos seleccionados (ver figura B.6), ahora necesi-

Figura B.5: Ejemplo 1. Diálogo Edición de contornos.

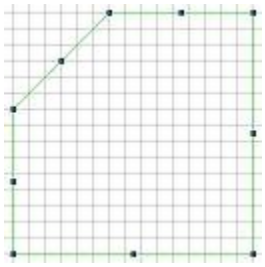

Figura B.6: Ejemplo 1. Geometría correctamente definida.

tamos abrir el cuadro de diálogo **Región**, esto se hace pulsando `Shift+R` o desde el menú "Editar > Región...".

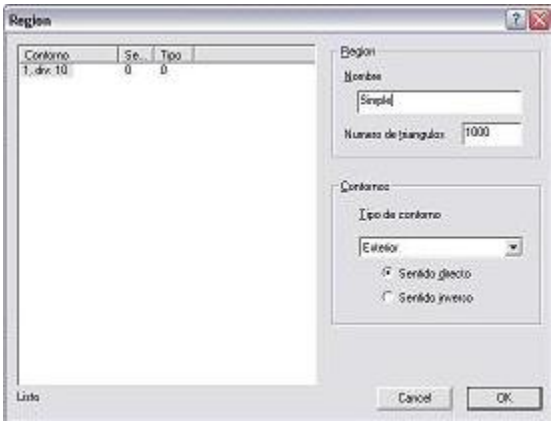

Figura B.7: Ejemplo 1. Diálogo Región.

En el cuadro de diálogo región se debe escoger el único contorno de la lista, para el mismo,

seleccione el tipo de contorno como “Exterior” y pulse el marcador “Sentido directo”. En el cuadro de edición Nombre escriba “Simple” y conserve el numero de triángulos en su valor predefinido, 1000. Al pulsar OK u oprimir `Enter` se guardan los archivos de entrada e inmediatamente se muestra la malla (ver figura B.8). Si se desea se pueden llevar los archivos `geom.i` y `prepro.part` generados, los cuales se encuentran en el directorio del programa.

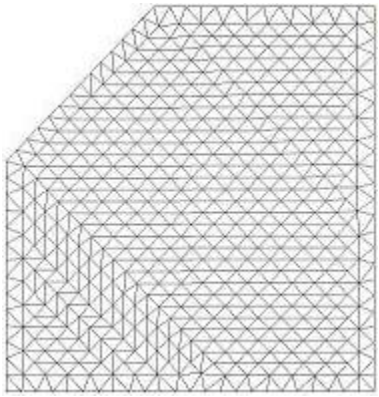

Figura B.8: Ejemplo 1. Región discretizada o malla.

Finalmente, si existen errores, no se genera la malla y se dibuja una flecha desde el primer punto del contorno al siguiente, como sabíamos que el sentido del contorno debía ser directo, la flecha mostrada deberia ir en el sentido de las manecillas del reloj, este es el sentido que requieren tener nuestros contornos para que el pre-procesador genere la región correctamente. Si alguna de las flechas se muestra en sentido anti-horario, se debe invertir el sentido del contorno en el cuadro de diálogo Región. Si todas las flechas apuntan en sentido horario, la malla no se puede generar por alguna razón diferente, por ejemplo, un mumero de elementos (triangulos) muy grande, o una geometria abierta.

B.2. Completo

Nivel: Intermedio
Objetivos:

- Redefinir un problema.
- Obtener una región discretizada con agujeros y refinamientos.

Tiempo: 10 min.
Procedimiento:

Los archivos generados en el ejemplo anterior están guardados en la carpeta `ejemplos/`. Abra el archivo Simple.efn para modificar la geometría.

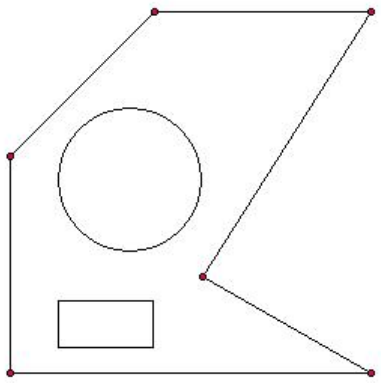

Figura B.9: Ejemplo 2. Geometría.

Alternativamente puede dibujar las figuras que conforman la geometría mostrada en la figura B.9. Seleccione la opción ajustar a la cuadricula activo, esto se hace con la tecla `S`, con el botón de la barra de herramientas o desde el menu Ver. El botón correspondiente debe mostrarse hundido (ver el ejemplo anterior).
Trate de dibujar las líneas rectas de tal forma que apunten en la dirección horaria del respectivo contorno. Seleccione la línea vertical de la derecha y ubíquese en uno de los el otro aun conectado y luego dibuje una línea para cerrar el contorno. En todo momento puede conocer como están conectadas las líneas observando la barra de estado. Finalmente, todas las líneas rectas deben mostrar el mensaje, `i2<>`, que significa que el objeto seleccionado tiene 2 intersecciones, una en su punto inicial ("<") y una en su punto final (">").

Adicionalmente dibuje la elipse y el rectángulo como se muestra en la figura B.9. Utilizando `Shift+B`, o desde el menú Editar abra el cuadro de diálogo condiciones de frontera y defina las condiciones de frontera que considere necesarias (ver el ejemplo anterior), para el ejemplo solo sera necesario definir una condicion de frontera llenando las campos de texto en el cuadro de diálogo condiciones de frontera y pulsando OK o *Enter*. Pase a la **Etapa de enmallado** usando `Shift+2`, los tres contornos deben aparecer resaltados de color verde. Luego, abra el cuadro de diálogo Edición de contornos, con `Shift+C` o en el menú " Editar > Obtener contornos...".

Aplique las condiciones de cualquier manera (observe que se puede aplicar una condición diferente a cada uno de los segmentos de los contornos), pero especifique los puntos de la siguiente manera, seleccione todos los segmentos menos los dos últimos e ingrese "Numero de puntos" y el numero 2, luego seleccione la elipse e ingrese "Numero de puntos" y 16, finalmente seleccione el rectángulo e ingrese "Numero de puntos" y 10 (por cada lado) como se muestra en la figura B.9.

Al pulsar OK se deben mostrar los puntos definidos. Ahora abra el diálogo Región usando Shift+R o desde el menu Edit, "Región". Escoja cada uno de los tres contornos de la lista y llene los campos como se muestra en la figura. Observe que definimos el rectángulo como un agujero de interfase identificado con el tipo #2.

Figura B.10: Ejemplo 2. Diálogo Edición de contornos.

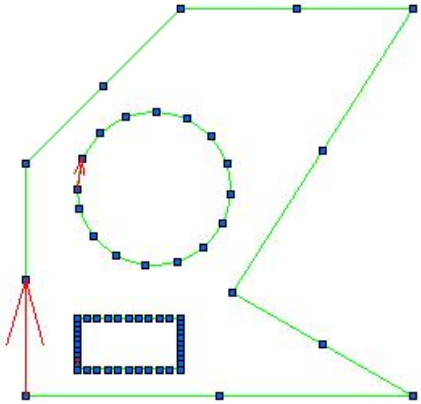

Figura B.11: Ejemplo 2, Contornos cerrados con sus puntos definidos.

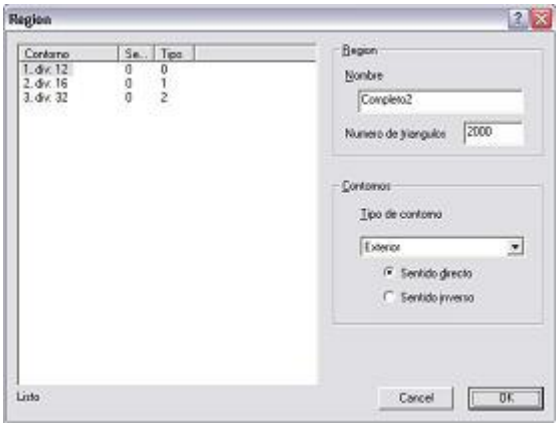

Figura B.12: Ejemplo 2. Diálogo Región.

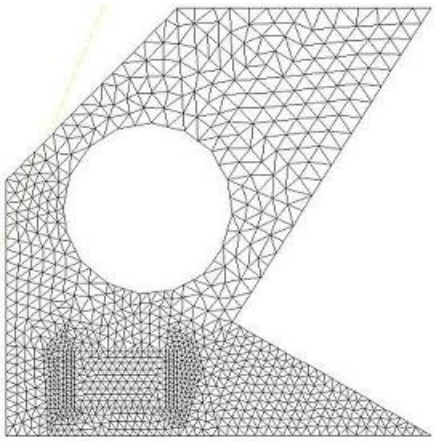

Figura B.13: Ejemplo 2, Región discretizada.

Al pulsar OK o Enter se generan los archivos y la región buscada. Los archivos de este ejemplo se encuentran en la carpeta Ejemplos/Completo.

B.3. UIS

Nivel: Avanzado
Objetivos:

- Trabajar con una región arbitraria.
- Definir un problema.
- Definir una dirección.
- Adaptar el numero de puntos.

Tiempo: 10 min.
Procedimiento:
Abra el archivo `UIS.efn` que se encuentra en `/ejemplos` en la carpeta del programa. Se deben mostrar las figuras geométricas con las cuales se debe obtener un contorno cerrado del escudo (Ver figura B.14), también puede dibujar las figuras por cuenta propia.

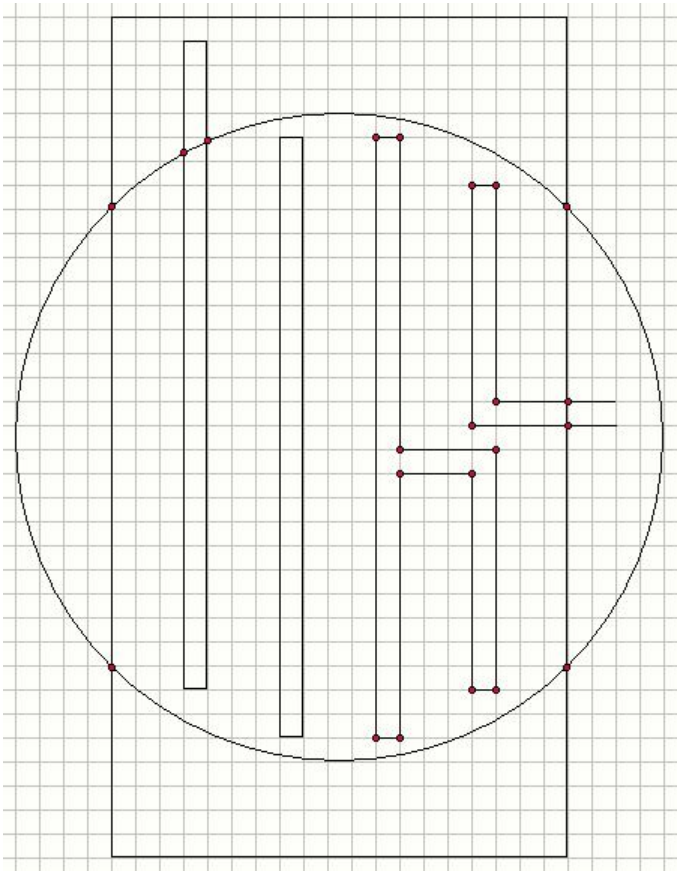

Figura B.14: Ejemplo 3. Geometría inicial del escudo.

Seleccione todas las figuras con las teclas `Ctrl+E` o seleccionando con el ratón un area que las contenga. Pulse el botón Dividir o las teclas `Shift+D` para dividir las figuras seleccionadas.

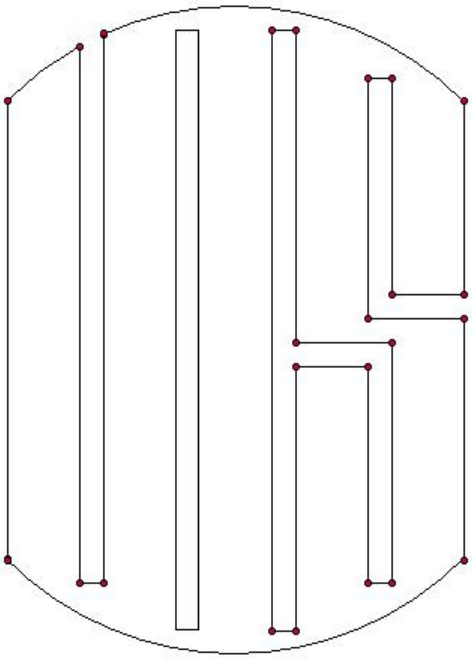

Figura B.16: Ejemplo 3. Geometría final del escudo.

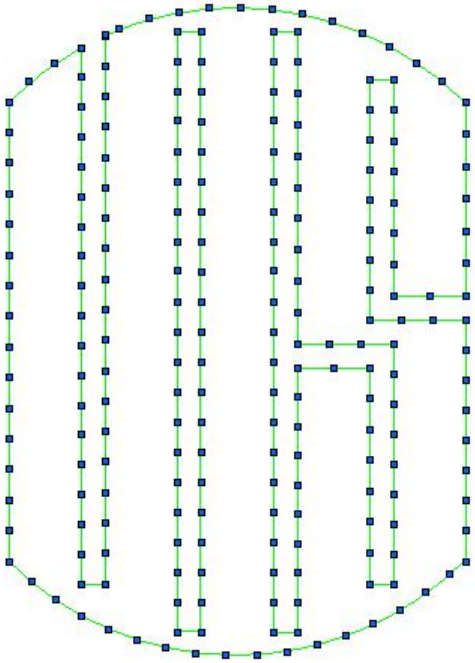

Figura B.17: Ejemplo 3. Puntos de división.

Luego, proceda a borrar todas las figuras sobrantes generadas. Utilice las teclas `Tabs` y `Shift+Tabs` para seleccionar las figuras en vez de utilizar el ratón para no mover las figuras.

Figura B.15: Una figura de tamaño cero.

Es necesario también eliminar todas las figuras de tamaño cero, las cuales se reconocen porque se muestra una "z.^{en} el tercer campo de la barra de estado (ver figura B.15). Estas se generan al dividir un objeto que ya se encuentra unido en alguno de sus extremos (una línea o un arco). Finalmente la geometría obtenida debe ser similar a la mostrada en la figura B.16.

A continuación se pasa a la etapa de enmallado con las teclas `Shift+2`, si la geometría es correcta, todos los contornos se mostraran de color verde, de lo contrario, los contornos abiertos se mostraran de color amarillo.

Abra el cuadro de diálogo Condiciones de frontera, e ingrese una, abra el cuadro de diálogo Contornos, seleccione todos los contornos de la lista y asigne la condición de frontera y 20 pixels como distancia entre puntos para todos. Los contornos con los puntos deben verse como en la figura B.17.

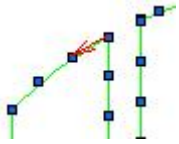

Figura B.21: Un contorno definido en sentido antihorario.

Finalmente ajuste los valores del cuadro de diálogo Región para que sean iguales a los mostrados en la figura B.18. Observe que el sentido del contorno exterior debe invertirse. Al oprimir la tecla `Enter` o el botón OK se debe dibujar la malla tal como en la figura B.19. Es posible que no obtenga la región enmallada sino un mensaje de error al cual deberá cerrar con las teclas `Ctrl+C`

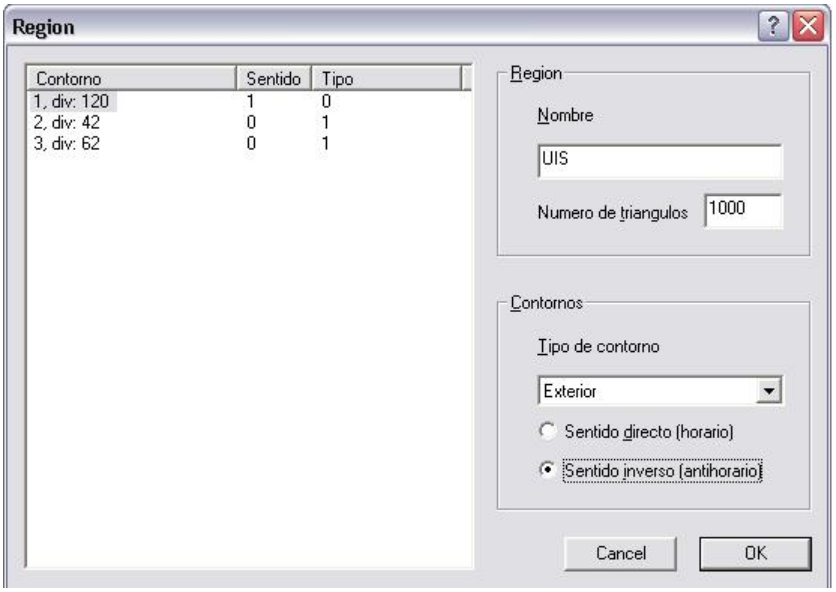

Figura B.18: Ejemplo 3. Diálogo Región.

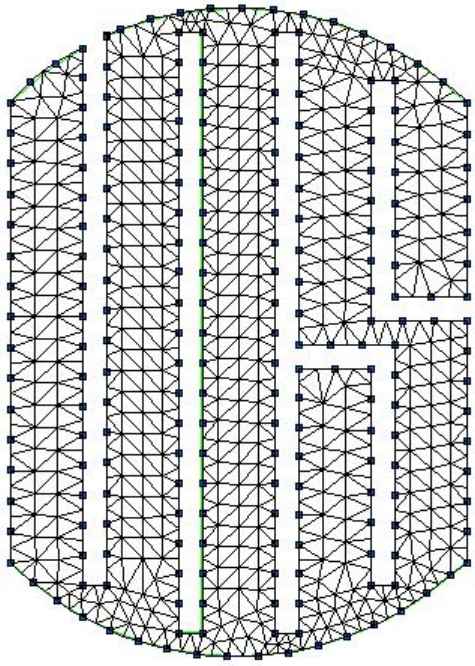

Figura B.19: Ejemplo 3. Región discretizada o malla.

(dos veces), en lugar de la malla aparecerán entonces, flechas rojas sobre los contornos mostrando el sentido de cada contorno. Por ejemplo, en la figura se muestra un contorno que tiene sentido antihorario, lo cual causa que el pre-procesador no pueda leer correctamente la geometría. Entonces es necesario que el usuario seleccione el sentido contrario del contorno en el diálogo Región.
Por otra parte si la malla se genera exitosamente, se pueden modificar los parámetros de los cuadros de diálogo Condiciones de frontera, Contornos y Región sin tener que volver a ingresar los demás datos. Por ejemplo se puede modificar solamente el numero de puntos y de elementos para generar

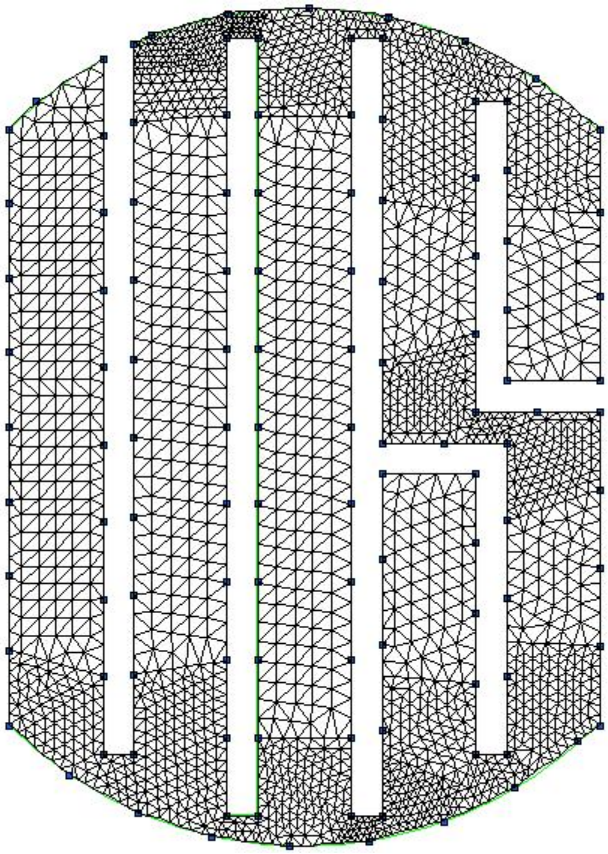

Figura B.20: Ejemplo 3. Malla con 4000 elementos.

la siguiente malla con 4000 elementos (figura B.20), observe que el numero máximo de elementos de una malla depende de los puntos utilizados.

ANEXO C

GLOSARIO

Acelerador: Combinación de teclas para ejecutar una función en un programa, por ejemplo Alt+F4.

Aplicación: Aplicación, programa, utilidad y herramienta computacional significan código ejecutable en una computadora.

Apuntador: Variable que guarda la dirección en memoria de un objeto.

Clase: Aunque a veces se sustituye el nombre "clase" donde debería utilizarse objeto, este es uno de los dos tipos de objetos existentes. Los miembros de una clase son declarados *private* por defecto.

Cuadro de Dialogo: Ventana con diversos controles como botones, listas y campos de texto. Normalmente esta contenida en una clase derivada de una ventana y es llamada desde una aplicación u otra ventana.

Cuadro de Dialogo Modal: Cuadro de dialogo que, cuando aparece, bloquea al programa que lo llamo, hasta que el usuario escoja una opción para cerrar el dialogo. Los diálogos modales contienen al menos dos botones, uno para aplicar los cambios y salir y otro para rechazar los cambios y salir.

Declaración: Sección de código fuente en C++ donde se escribe el nombre y los parámetros de una función Además de definir los marcadores public, protected, private, virtual y static.

Definición: Sección de código fuente en C++ donde se escribe el funcionamiento de una función.

Entorno Integrado de Desarrollo: Programa para desarrollar y editar programas utilizando uno o varios lenguajes de programación, ejemplo: Microsoft Visual Studio.

Estructura: El termino se utiliza desde la programación estructurada, actualmente es el tipo de objeto cuyos miembros son declarados *public* por defecto.

Funcional: En calculo variacional, variable que depende de una función.

Herencia: El concepto de herencia es uno de los pilares de la programación orientada a objetos, es la capacidad que tienen las clases derivadas (o clases hijas) de tener (o heredar) las variables y funciones miembros de la clase base (o clase padre). La herencia se aplica recursivamente es decir que las clases hijas heredan características de su clase padre e implícitamente de la clase padre de su clase padre hasta la primera de ellas.

Interfaz de programación de aplicaciones: (API), es un conjunto de bibliotecas especializadas diseñadas para servir de ayuda a los programas en un determinado tipo de operación, normalmente en conjunción con tipos específicos de *hardware*.

MDI, SDI: Arquitecturas de clases utilizadas en donde el programa, el documento y la vista son clases independientes, en el SDI o Single Document Interface por cada documento abierto se requiere una nueva instancia del programa mientras que con la arquitectura MDI, Multiple Document Interface un solo programa abierto permite manejar varios documentos a la vez. El Bloc de notas de Windows es un programa hecho con arquitectura SDI y el Microsoft Word uno con MDI.

Menú de contexto: Menú accesible con el click derecho del ratón.

MFC: *Microsoft Foundation Classes*, librería de clases disponible con Visual Studio que permite crear programas para los sistemas operativos Windows de Microsoft.

Objeto: Unidad funcional de código utilizada en los lenguajes de programación orientados a objetos, consiste en uno o mas miembros que pueden ser variables (datos) o funciones (rutinas), se utiliza de acuerdo a unas reglas definidas que pretenden encapsular la información y hacer que solo se acceda al objeto mediante algunos de sus miembros -la interfaz del objeto- mientras que otros -la implementación del objeto- son utilizados por el mismo, sus clases derivadas y clases amigas solamente. Los objetos se clasifican en clases y estructuras.

Polimorfismo: Del griego $\pi o \lambda \psi \mu o \rho \phi o \zeta$, es la capacidad de llamar a diferentes funciones mediante el mismo nombre. Según la clase utilizada, esto permite escoger entre funciones durante la ejecución del programa. El polimorfismo se consigue con funciones virtuales en la clase base que son sobrecargadas en las clases descendientes. Al utilizar un apuntador para llamar a la función se hace una llamada polimorfa, por ejemplo, apuntador->función_virtual().

Tabla de cadenas: recurso de un programa donde se guardan las cadenas de caracteres. Es es un termino utilizado en Visual Studio.

BIBLIOGRAFÍA

[1] ARCADIO MORENO AGUILAR. *Entienda La Gramática Moderna.* Ediciones Larousse 1985.

[2] ELKIN RAFAEL ARROYO NEGRETE, PEDRO JOSÈ DIAZ GUERRERO. *Creación De Objetos En C++, Para La Solución De Problemas De Ingeniería.* Universidad Industrial de Santander, 1998.

[3] The GNOME Usability Project. *Human Interface Guidelines 2.0* Calum Benson 2004.

[4] ANDREW KOENIG, BARBARA MOO. *Accelerated C++* AT&t, Inc., and Barbara Moo 2000.

[5] JON BATES & TIM TOMPKINS. *Descubre Microsoft Visual C++ 6* Prentice Hall 1999.

[6] DAVID J. KRUGLINSKI. *Programación avanzada con Visual C++ 5* Microsoft Corporation 1997.