UNIVERSIDAD NACIONAL AUTÓNOMA DE MEXICO

FACULTADO DE ESTUDIOS SUPERIORES ARAGÓN

"Interfaz Gráfica Innovadora para un Simulador Numérico de Flujos"

T E S I S
QUE PARA OBTENER EL TÍTULO DE
INGENIERO EN COMPUTACIÓN
P R E S E N T A

L I D I A A R I S T A S Á N C H E Z

DIRECTOR DE TESIS
M. EN I. ELIO VEGA MUNGUÍA

MÉXICO, D.F., NOVIEMBRE 2007

UNAM – Dirección General de Bibliotecas
Tesis Digitales
Restricciones de uso

DERECHOS RESERVADOS ©
PROHIBIDA SU REPRODUCCIÓN TOTAL O PARCIAL

Todo el material contenido en esta tesis esta protegido por la Ley Federal del Derecho de Autor (LFDA) de los Estados Unidos Mexicanos (México).

El uso de imágenes, fragmentos de videos, y demás material que sea objeto de protección de los derechos de autor, será exclusivamente para fines educativos e informativos y deberá citar la fuente donde la obtuvo mencionando el autor o autores. Cualquier uso distinto como el lucro, reproducción, edición o modificación, será perseguido y sancionado por el respectivo titular de los Derechos de Autor.

AGRADECIMIENTOS

Dios
Por la oportunidad de vivir y permanecer
a mi lado cada momento.

Papás
Les agradezco todos los sacrificios realizados
para poder ser lo que soy.

A mis hermanos y a mi abuelita
Por su cariño e infinita paciencia.

Tom y Jony
Los quiero, y ojala tengan siempre en su corazón la fortaleza
para luchar por sus propias convicciones.

Mary
Sabes que eres algo especial en mi vida;
gracias por tu apoyo incondicional,
y sobre todo por tu gran corazón.

Gustavo
Mi amor, gracias por recorrer este
camino juntos.

A la Universidad Nacional Autónoma de México
Por dejarme formar parte de ella, sintiéndome orgullosamente azul y oro.

Y a todas las personas que me apoyaron con sus conocimientos
y concejos para seguir adelante.

GRACIAS

ÍNDICE

INTRODUCCIÓN .... **I**

CAPÍTULO 1. INTERFAZ GRÁFICA DE USUARIO **1**

INTRODUCCIÓN **1**

1.1. DEFINICIÓN DE INTERFAZ GRÁFICA DE USUARIO **2**

1.2. ORÍGENES DE LAS INTERFACES GRÁFICAS DE USUARIO **5**

1.3. ALTERNATIVAS PARA LA CONSTRUCCIÓN DE INTERFACES GRÁFICAS DE USUARIO **11**

CAPÍTULO 2. ANÁLISIS **20**

2.1. DESCRIPCIÓN DEL SIMULADOR DE FLUJOS **20**

2.1.1. DINÁMICA DE FLUIDOS COMPUTACIONAL 21
2.1.2. PLANTEAMIENTO DE LAS ECUACIONES DE LA MECÁNICA DE FLUIDOS 22
2.1.3. EJEMPLOS 23
2.1.3.1. Difusión en una y dos dimensiones *24*
2.1.3.2. Convección forzada en 2D y 3D *27*

2.2. ANÁLISIS DE INTERFACES GRÁFICAS DE OTROS SIMULADORES **30**

CAPÍTULO 3. DISEÑO **39**

3.1. PAPEL DEL DISEÑO **39**

3.2. USO DE METÁFORAS VISUALES **43**

3.2.1 LA NECESIDAD DE INTERFACES MÁS INTUITIVAS **43**

3.2.2 ¿POR QUÉ USAR METÁFORAS VISUALES EN INTERFACES GRÁFICAS DE USUARIO? 43
3.2.3 LINEAMIENTOS PARA EL DISEÑO DE METÁFORAS VISUALES EFECTIVAS 44
3.2.3.1 Errores comunes en uso de metáforas visuales *44*
3.2.3.2 Recomendaciones para diseñar sistemas de metáforas para una interfaz *44*
3.2.3.3 La importancia de los paradigmas *45*
3.2.3.3.1 Modelo-Vista-Controlador (View-Controller-Model) 46

3.3. MODELO ANALÍTICO DE DESCRIPCIÓN DE TAREAS ORIENTADO A LA ESPECIFICACIÓN DE INTERFACES. **46**

3.4. PRINCIPIOS **53**

3.4.1. PRINCIPIO 1. RECONOCE LA DIVERSIDAD 53
3.4.1.1. Perfiles de usuarios *53*
3.4.1.2. Estilos de interacción *53*
3.4.2. PRINCIPIO 2. USA LAS 8 REGLAS DORADAS DEL DISEÑO DE INTERFACES 55
3.4.3. PRINCIPIO 3. PREVENIR ERRORES 55
3.4.3.1. Correcto apareamiento de pares *55*
3.4.3.2. Secuencias completas *56*
3.4.3.3. Comandos correctos *56*

3.5. GUÍA PARA EL DESPLIEGUE DE DATOS **56**

3.5.1. ORGANIZACIÓN DEL DESPLIEGUE 56
3.5.2. OBTENER LA ATENCIÓN DEL USUARIO 56

3.6. GUÍA PARA LA CAPTURA DE DATOS **57**

3.7. EVALUACIÓN DE INTERFACES **58**

3.7.1. REVISIONES POR EXPERTOS 58
3.7.2. LABORATORIOS Y PRUEBAS DE USO 59
3.7.3. CUESTIONARIOS 59

CAPÍTULO 4. DESARROLLO DE LA INTERAFAZ GRÁFICA DE USUARIO Y RESULTADOS OBTENIDOS 63

4.1. METODOLOGÍAS DE DESARROLLO 63

4.1.1. ¿QUÉ ES UNA METODOLOGÍA Y POR QUÉ NOS INTERESA? 63
4.1.2. METODOLOGÍAS TRADICIONALES CONTRA METODOLOGÍAS ÁGILES 64
4.1.3. METODOLOGÍAS TRADICIONALES 66

4.2. DESARROLLO 68

4.2.1 EL ROL DEL USUARIO 68
4.2.2 EL ROL DEL DESARROLLADOR 69
4.2.3 RELACIÓN BIPARTITA EN EL DESARROLLO DE LA INTERFAZ 70

4.3. ANÁLISIS DEL PROBLEMA 70

4.3.1 PLANTEAMIENTO DEL PROBLEMA 71
4.3.1.1 Modelo de Requisitos 71
4.3.1.2 Lista de procesos 72
4.3.2 DIAGRAMA DE CONTEXTO 73
4.3.3 DIAGRAMAS DE CASOS DE USO 74

4.4. HERRAMIENTAS DE DESARROLLO 74

4.4.1. LENGUAJE UNIFICADO DE MODELADO 74
4.4.2. HILOS (*THREADS*, EN INGLES) 75
4.4.2.1. Definición 75
4.4.2.2. Estados de un hilo 76
4.4.2.3. Cambios de estados 77
4.4.2.3.1. Implementaciones 77
4.4.2.4. Planificación de hilos 78
4.4.2.5. Ciclo de Vida de una Hilo 80
4.4.2.6. Ventaja de los hilos contra los procesos 81
4.4.2.7. Planificación de hilos 81
4.4.3. VISUALIZACIÓN DE RESULTADOS A TRAVÉS DE LA GUI 81
4.4.3.1. OpenGL 82
4.4.3.1.1. Características de OpenGL 83
4.4.3.1.2. Las librerías de OpenGL 84
4.4.3.1.3. Primitivas de OpenGL 85
4.4.3.1.4. Funciones de CallBack 88
4.4.3.1.5. El proceso de visualización en OpenGL 89
4.4.3.1.6. Sistemas de coordenadas 90
4.4.3.1.7. Proyecciones en OpenGL 91
4.4.3.1.8. Proyección Ortográfica 91
4.4.3.1.9. Proyección en perspectiva 92
4.4.3.1.10. Volúmenes de Vista 93
4.4.3.1.11. Transformación del viewport 95
4.4.3.2. OpenDX 96
4.4.3.2.1. Herramientas de OpenDX 97
4.4.3.2.2. Programación visual 99
4.4.3.2.3 Modelo de datos 100
4.4.4. PROGRAMACIÓN ORIENTADA A OBJETOS 101
4.4.4.1. Tipos de Datos Abstractos 101
4.4.4.2. Clases y objetos 101
4.4.4.3. Herencia y polimorfismo 102
4.4.4.4. Programación genérica 103
4.4.4.5. Arquitecturas para el sistema 103
4.4.4.5.1. Sistema Operativo 104
4.4.4.5.2. Software de desarrollo 104

4.5. DISEÑO, PROGRAMACIÓN E IMPLEMENTACIÓN 104

4.5.1. DISEÑO 105
4.5.1.1. Diseño del sistema 105
4.5.1.2. Adquisición de requerimientos 106
4.5.1.3. Reuniones 106

4.5.1.4. Mapa de navegación ... *107*
4.5.1.5. Diseño de pantallas ... *107*
4.5.2. IMPLEMENTACIÓN ... 114
4.5.3. PROGRAMACIÓN DE LAS PANTALLAS ... 115
4.5.4. PRUEBAS ... 125
CONCLUSIÓN ... 129
TRABAJO FUTURO ... 130
APÉNDICE A. ... 132
HISTORIA DE LAS GUI A TRAVÉS DE IMÁGENES ... 132
APÉNDICE B ... 133
CÓDIGO DEL SIMULADOR. ... 133
APÉNDICE C ... 135
ANÁLISIS DE INTERFACES GRÁFICAS DE OTROS SIMULADORES EVALUACIÓN ... 135
APÉNDICE D ... 147
DISEÑO ... 147
APÉNDICE E ... 150
CASOS DE USO ... 150
DIAGRAMA DE CLASES MODELO DE IMPLEMENTACIÓN CLASES ... 157
APÉNDICE F ... 158
INSTALACIÓN A PARTIR DE FUENTES ... 158
APÉNDICE G ... 161
DESARROLLO DE APLICACIONES EN QT ... 161
APÉNDICE H ... 163
COMANDOS BÁSICOS DE "VI" ... 163
APÉNDICE I ... 165
MAKEFILE ... 165
APÉNDICE J ... 168
COMANDOS BÁSICOS DE LINUX ... 168
ÍNDICE DE FIGURAS ... 170
GLOSARIO ... 172
BIBLIOGRAFÍA ... 178

INTRODUCCIÓN

Hoy en día los computadores están presentes en todos los escritorios y son considerados bienes de consumo. A partir de esta gran proliferación y uso de computadores personales (PC) es natural que la definición y el proceso de desarrollo de sistemas hayan adquirido otra dimensión. Consideraciones tanto en la realización, presentación del producto como a las necesidades y deseo de los usuarios, y estas ultimas pasan a tener un papel central en el desarrollo de aplicaciones con interfaces graficas.

Hace mucho que las pantallas con fondo oscuro y letras verdes fueron dejadas de lado. Los sistemas basados en caracteres, que exigían entre otras cosas memorización de comandos, no pueden ser comparados a aplicaciones con interfaces graficas intuitivas y atrayentes.

Hoy en día el diseño de Interfaces de Usuario es una polémica por diversos motivos, unos de ellos sigue siendo el mismo desde los inicios del computador, como es el temor al ser desplazado el ser humano por un computador, y otro que es el que nos interesa en este trabajo, es como el usuario se comunica con el computador, con nuestra sistema; el diseño de interfaces esta enfocado principalmente a nuestro usuario como parte primordial, sin él, la herramienta no seria útil.

El desarrollo de software ha ido creciendo rápidamente y con el debería también ir creciendo el diseño de Interfaz de Usuario, como su nombre lo indica de "usuario", pero el diseño a comenzado a olvidarse y detenerse en esta parte, en nuestros días nos encontramos con diversidad de software, pero analicemos ¿Qué diferencias o facilidades se le proporcionan al usuario para utilizar dicha herramienta?; nuestro usuario sigue necesitando de manuales, capacitación, practica, entre otras cosas para poder utilizarlo, considero que los diseños de hoy están muy enfocadas a la solución de problemas y a llamar la atención del usuario, olvidándose, de cómo esté, puede utilizar nuestra herramienta de manera simple y transparente.

No debemos olvidar que ahora los usuarios son mas exigentes, además de considerar como parte primordial su tiempo, así que pensemos en disminuir su tiempo de aprendizaje, proporcionándoles una herramienta que le apoye y con ello lograr que se olvide de que el computador es complejo de utilizar; además de que la comunicación entre el usuario y la computadora se realiza a través de una interfaz, la cual es responsable por el buen éxito de la interacción y por la fuerza del dialogo. Paul Heckel[1] sugiere que, para proyectar buenas interfaces con usuarios, se debe pensar más en comunicación que en computación.

El cómputo científico es un complemento de la teoría y la investigación que permite, a través de simulaciones complejas, acelerar el desarrollo de nuevas tecnologías. En la mayoría de los casos, la generación de software necesario para realizar dichas simulaciones requiere de un equipo de trabajo multidisciplinario para

[1] Heckel, P. Software Amigável: Técnicas de Projeto para una melhor Interface com o Usuario. Editora Campus, Rio de Janeiro, 1991.

integrar conocimientos de matemáticas, física, ingeniería y cómputo. El software resultante es bastante complejo y se requiere de enormes manuales para aprender a usarlo, además de que muchos de ellos no incorporan una Interfaz Gráfica de Usuario (GUI por sus siglas en inglés). Por otro lado, aquellos que si contienen una GUI, no son amigables y se requiere de otro manual que explique la misma GUI.

En el departamento de visualización de la DGSCA, se ha desarrollado un sistema (simulador) que resuelve numéricamente las ecuaciones de balance de flujos laminares y turbulentos. Este software se realizó en C++ y pretende ser de aprendizaje simple mediante el uso de clases y objetos bien definidos, que describen de forma natural las entidades del dominio del problema. Sin embargo, para resolver un problema se requiere de un conocimiento mínimo de la programación orientada a objetos, lo cual no es del gusto de muchos alumnos e investigadores de ciencia e ingeniería. De acuerdo con la experiencia de investigadores, es necesario tener una interfaz gráfica amigable, de fácil uso y aprendizaje, y que además presente una vista atractiva al usuario.

Por lo anterior se busca lograr un buen diseño de Interfaz de Usuario donde nuestro usuario se sienta cómodo, que el software le apoyo en la solución de sus problemas, además de solucionárselos; el éxito o el fracaso de las Interfaces de Usuario dependen de un buen análisis de nuestro usuario, requerimientos, necesidades, entre otras cosas que analizaremos en este trabajo.

Actualmente existen muchas herramientas para el desarrollo de interfaces gráficas de usuario, por ejemplo: QT, wxWidgets, GTK, Java, etc. En este trabajo se realizará una investigación de estas herramientas y se seleccionará la más adecuada para llevar a cabo el objetivo arriba planteado.

Una vez seleccionada la herramienta a utilizar, se hará un estudio de las interfaces que utilizan algunos de los programas existentes en la actualidad para encontrar sus defectos en cuanto al uso y el aprendizaje. Posteriormente, se realizará un diseño que subsane los defectos de otras interfaces. Además, se realizarán interfaces con efectos atractivos a los usuarios que le permitan recordar los pasos que se requieren para realizar una simulación. La interfaz deberá permitir adecuar su presentación (color, forma, distribución, etc.) mediante el uso de temas, los cuales podrá diseñar el usuario a su conveniencia y gusto.

Una buena interfaz gráfica solo puede ser obtenida con un trabajo consistente que tenga como objetivo conocer al usuario final considerando tanto la presentación como las necesidades y deseos del usuario, ya que estas pasan a tener un papel central en el desarrollo de aplicaciones con interfaces graficas, y esto provoca un cambio en la definición y en el proceso de desarrollo de sistemas computacionales.

Es importante mencionar que para comenzar con el diseño o con una investigación siempre se inicia con los orígenes, así como con opiniones de personalidades en el mundo de la investigación sobre el diseño de GUI´s; con ello se busca satisfacer las expectativas, de hecho la experiencia recogida muestra que, a pesar de las múltiples e innegables ventajas que las interfaces bien analizadas ofrecen, y de la importancia que esto ha tomado en algunas aplicaciones aún no es tomada como algo importante; sin embargo lograremos un mejor análisis con mejores resultados.

En conclusión el principal objetivo de este trabajo es diseñar una Interfaz que sea además de útil, agradable y fácil de manipular, se busca ayudar al usuario ya que una interfaz que no cumpla tal condición puede considerarse inútil, aunque esta pueda utilizarse habitualmente[2]. También se proporcionan algunos puntos importantes que serán sin duda, necesarios y de gran apoyo para cualquier persona que desee comenzar un diseño de GUI, ya que contará con puntos de partida, referencias importantes y considerables para el diseño.

El desarrollo del software se basa en un falsa premisa; las computadoras son fáciles de usar, por ende, el software es fácil de utilizar. Desafortunadamente, este es falso. Se trata, en efecto, de una apreciación sustentada en la creencia de que las nuevas tecnologías (Ej. Interfaces gráficas, videos, sonidos, imágenes animadas, realidad virtual, etc.), facilitan de facto el uso de las computadoras. Sin embargo, esto no es necesariamente verdadero, puesto que cada nueva tecnología trae consigo problemas que es necesario estudiar.

Así, este trabajo se basa en la idea que es necesario también analizar las herramientas que deben ser integradas en el desarrollo de la aplicación.

En el laboratorio de Visualización se comenzó a trabajar con el desarrollo de una interfaz de calidad planteándose algunas preguntas para comenzar. ¿Qué es un software de calidad?, ¿Con que se debe comenzar?, ¿Cómo integrar técnicas eficientes para desarrollar?

Por todo lo anterior, y aprovechando algunas de las técnicas de la Ingeniería en Computación se planteó la posibilidad de comenzar por la investigación de herramientas para le desarrollo de Interfaces así como de diseño.

Esta Interfaz comprende: captura de datos, de cálculos y por último de visualización. Por lo anterior se comenzó con la comprensión del problema, continuando con el diseño y desarrollo.

Las secciones realizadas se desarrollan en el presente documento de la siguiente manera:

En el capítulo 1 se expone y describe la historia, definición y alternativas para la su construcción de una Interfaz Gráfica de Usuario.

En el capítulo 2 se exponen los temas: descripción del simulador de fluidos y análisis de interfaces gráficas para fluidos.

En el capítulo 3 se describe el papel que juega el usuario en el diseño, definición, características del diseño, uso de metáforas y formas de evaluación de Interfaces Gráficas de Usuario.

En el capítulo 4 se describe la comunicación entre la GUI y el simulador, la visualización de resultados a través de la GUI, el desarrollo, la implementación de la

---

[2] En este sentido, es interesante leer artículo referente al fracaso de la transmisión y recepción de datos por teléfono escrito por Jakob Nielsen: Nielsen, Jakob; "Telephone Usability: Voice is Just Another Datatype", disponible en http://www.useit.com/papers/telephone_usability.html

interfaz, también se exponen los resultados alcanzados y por último se presentan las conclusiones obtenidas con el desarrollo de este trabajo.

CAPÍTULO 1

INTERFAZ GRÁFICA DE USUARIO

Capítulo 1. INTERFAZ GRÁFICA DE USUARIO

INTRODUCCIÓN

El término "**Interfaz**[1] **Gráfica de Usuario**" (**GUI**, por sus siglas en inglés, *Graphical User Interface*) de un programa es la parte del mismo que determina como se comunica usuario y ordenador, es el artefacto tecnológico de un sistema interactivo que posibilita, a través del uso y la representación del lenguaje visual, una interacción amigable con un sistema informático, por lo que la GUI es la parte crucial en cualquier aplicación.

Surge como evolución de la línea de comandos de los primeros sistemas operativos y es pieza fundamental en un entorno gráfico.

La historia de la informática está indisolublemente unida a las interfaces gráficas, puesto que los sistemas operativos gráficos han ocasionado grandes consecuencias en la industria del software y del hardware.

Las interfaces graficas surgen de la necesidad de hacer los ordenadores (o PCs) más accesibles para el uso de los usuarios comunes. La gran limitación que poseían las computadoras diseñadas para un uso masivo era que sólo funcionaban bajo líneas de comando, lo que requería que la persona que quisiera utilizar un PC tuviera un mínimo conocimiento sobre su funcionamiento.

Esta limitación fue salvada gracias al desarrollo de los entornos gráficos, que permitieron que las personas pudieran acceder a un PC sin tener que pasar por el tortuoso proceso de tener que aprender a manejar un entorno bajo línea de comandos.

Los principales pasos en la evolución de las GUI's comenzaron con Douglas Engelbart, además de inventor del Mouse; desarrolló la primera interfaz gráfica en los años 1960 en EE.UU. en los laboratorios de XEROX. Fue introducida posteriormente al público en las computadoras Apple Macintosh en 1984, los Commodore Amiga en 1985, y a las masas hasta 1993 con la primera versión popular del sistema operativo Windows 3.0[2].

Existen dos tipos de Interfaces:

Las de Línea de Comandos (CLI, por sus iniciales en inglés), posteriormente nombradas Interfaces de Texto (TUI)[3] por solo utilizar modo texto para su funcionamiento, en estas la comunicación entre el usuario y el ordenador es muy pobre ya que el usuario requiere de aprender la sintaxis del lenguaje empleado para la comunicación.

---

[1] Interfaz (Del ingl. *Interface*, superficie de contacto). F. *inform.* Conexión física y funcional entre dos aparatos o sistemas independientes. Real Academia Española © Todos los derechos reservados. En español es **interfaz** y su plural es **interfaces**.

[2] En el 2001 aparece KDE3, GNOME2 y Windows XP.

[3] CLI y TUI no es lo mismo. Las CLI emigraron de las shells y se transformaron en parte integral de varias aplicaciones como interfaz alternativa y/o paralela a las GUI.

Las CLI se originaron cuando se conectaron teletipos[4] a computadoras, en los años 50. En términos de acción inmediata y respuesta, supusieron un avance sobre el uso de tarjetas perforadas. Podían tener menús, ventanas y cursores del ratón, pero todo representado con texto ASCII.

Las de modo Grafico (GUI, Graphical User Interface)[5] donde la comunicación es por imágenes y elementos gráficos (iconos, ventanas, tipografía) que representan objetos del mundo real y que el usuario maneja de manera intuitiva, existe retroalimentación y dialogo con el usuario; para su construcción es necesario mencionar que se requiere de una toolkit[6], es decir, de un conjunto de funciones y procedimientos que proporciona un lenguaje de programación permitiendo así construir todos los elementos de la GUI.

La interfaz tiene que hacer "usables" para los usuarios las aplicaciones informáticas[7]. El objetivo fundamental de una GUI es básicamente hacerle comprensible y amigable el trabajo al usuario, "Si el modelo de programa corresponde al modelo de usuario, tu interfaz de usuario tendrá éxito"[8]. La mayoría de autores que saben y hablan sobre el tema, comentan que un sistema debe ser "invisible" para el usuario final.

Un ejemplo de estos dos tipos de Interfaces de Usuario la podrás apreciar en el Apéndice A junto con la evolución de las GUI´s a través de imágenes.

1.1. DEFINICIÓN DE INTERFAZ GRÁFICA DE USUARIO

Interfaz es un concepto sin una definición aceptada por todos. En general, puede entenderse que la interfaz de usuario es lo que ve el usuario del sistema. No es el sistema en sí, sino su puesta en escena[9] y como tal debe comprenderse. Puede considerarse que todo sistema que permita una interacción entre él y su usuario consta de una interfaz de usuario[10].

Por lo tanto, es necesario saber qué es una interfaz, ya que de su calidad, utilidad, usabilidad y aceptación depende el éxito de un sistema.

Definiciones hay muchas:

---

[4] Sistema de transmisión de textos, vía telegráfica, a través de un teclado que permite la emisión, recepción e impresión del mensaje.

[5] Algunas GUI´s son diseñadas para cumplir usos específicos como las ahora *Touchscreeen* o Pantalla Táctil, iniciado por Gene Mosher en la computadora del ST de Atari en 1986; su uso específico ahora famoso en la industria alimenticia, bebidas, cajeros automáticos, pantallas de monitoreo, etc.

[6] Otro nombre para los toolkits es "sistemas de desarrollo multiplataforma".

[7] Laurel, Brenda; *Computers as Theatre*. Reading, Massachusetts: Addison-Wesley, 1993; p. 2.

[8] Joel Spolsky. En la pagina titulada "Joel on Software. La Opinión de Joel sobre qué es Software" podrás encontrar información sobre el diseño de Interfaces de Usuario http://spanish.joelonsoftware.com/.

[9] Norman, Donald A., "Why interfaces don´t work". En: Laurel, Brenda (ed.); *The Art of Human-Computer Interface Design*. Reading, Massachusetts: Addison-Wesley, 1991; pp. 209-219.

[10] Borges, José A., Morales, Israel y Rodríguez, Néstor J.; "Guidelines for Designing Usable World Wide Web Pages".

- **Landauer** considera, sucintamente, que una interfaz es "el juego de mandos por el que los usuarios pueden hacer trabajar un sistema".

- **Baecker** completa la anterior definición, mostrando la interfaz de dos formas: en su explicación abreviada sostiene que la interfaz "comprende los dispositivos de entrada y salida y el *software* que los gestionan", y en la ampliada afirma que la interfaz abarca "todo aquello que permite al usuario vivir una experiencia con un ordenador, incluyendo la ayuda humana, la documental y la ofrecida por la propia máquina".

- **Mandel** opina que la interfaz es "lo que el usuario ve en la pantalla", y alarga la idea hasta abarcar la "totalidad de la experiencia que se da entre usuario y ordenador". La interfaz incluye tanto el *software* como el *hardware* que presentan información al usuario y que permiten a éste interaccionar con la propia información y con la máquina, además de la documentación *online* o impresa que acompaña al sistema.

- En este sentido, **Rew** y **Davis** opinan que una interfaz "relaciona las capacidades de presentación e interacción del sistema con el modelo cognoscitivo y perceptual del usuario".

- **Bradford** sigue en la misma línea y mantiene que la interfaz se define como "cualquier parte del sistema con la que el usuario pueda comunicarse, sea a nivel físico, conceptual o de percepción".

- **Shneiderman** va más allá y define "interfaz" como la membrana de comunicación entre los usuarios y el sistema por la que los diseñadores y desarrolladores de herramientas informáticas pueden hacer que la tecnología sea inteligible y sensible a las necesidades de los usuarios. Por lo tanto, las características del diseño de esta interfaz pueden facilitan o impedir la interacción entre hombre y máquina.

En este sentido, Shneiderman opina que desde que las interfaces gráficas de usuario han ido sustituyendo a interfaces de usuario de corte más clásico -por ejemplo, los de "modo comando"- el usuario ha ido teniendo la posibilidad de abandonar complejas sintaxis escritas utilizando el sistema mediante manipulaciones directas a objetos gráficos representados en su pantalla. El énfasis ha ido cambiando para centrarse en el medio visual del usuario final, lo que facilita el aprendizaje del sistema, sobre todo si se ajustan a ciertos estándares de comportamiento, simbolismo y visualización que eviten la desorientación y que mejoren la comprensión del funcionamiento del sistema y su manejo intuitivo. Por lo tanto, lo que el usuario ve en pantalla debe ser una metáfora, primordialmente gráfica, del mundo en el que trabaja.

Dentro de esta tendencia, Williams, Sochats y Morse opinan que la interfaz es el filtro por el que un "conjunto de datos modelados por un sistema informático" se

presentan al usuario. La utilidad de la interfaz se da en función de si la información presentada al usuario "le ayuda a conseguir sus objetivos dentro de los límites establecidos por su cultura".

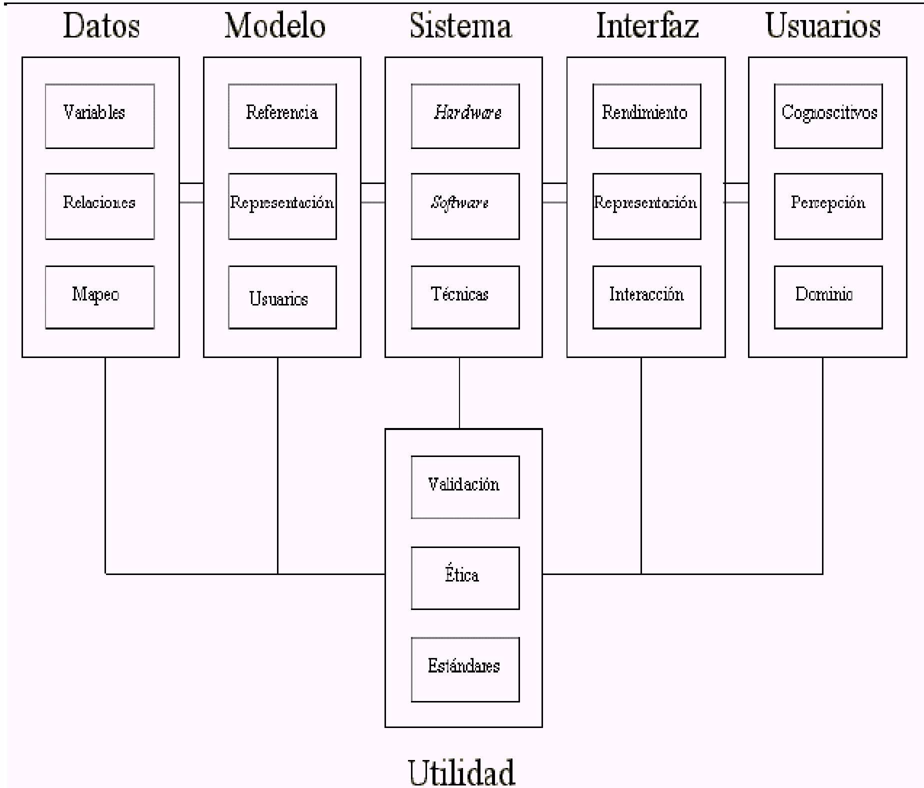


Figura *1.1.* Modelo de visualización de componentes, -según Williams, Sochats y Morse-.

Por lo tanto, la interfaz de usuario "es algo más que software" e incluye todas las interacciones que el usuario tiene con el producto. En este sentido, el concepto de interfaz de usuario recoge la totalidad de los aspectos "visibles o superficiales" de un sistema:

- Los dispositivos de entrada y salida de datos.
- La información presentada al usuario y la que se extrae de sus acciones.
- El comportamiento del sistema.

Desde este punto de vista, la interfaz de usuario divide el canal de comunicación entre hombre y máquina. Y para una interacción hombre-máquina adecuada, la interfaz debe ofrecer a cada uno de los dos agentes soluciones a sus necesidades, no sólo ser una pantalla divisoria.

Para que cada agente tenga la información que necesita, es necesario que ambos "hablen" el mismo idioma. Un primer paso fue intentar que el hombre hablase con la máquina en su idioma, pero esta idea fue fallando conforme el número de usuarios potenciales de un sistema fue aumentando y sus demandas fueron escuchadas: el hombre no puede acostumbrarse a hablar con la máquina en un "lenguaje autómata", sino que es la máquina quien debe reflejar el mundo humano y quien debe comunicarse de forma inteligible.

Rodríguez de las Heras, a modo de conclusión, argumenta que "interfaz es, habitualmente, un ente especular de comunicación entre dos agentes activos: hombre y máquina. El interfaz copia el entorno del usuario para hacer más comprensible su relación con él. Para ello, se basa en objetos del mundo real -por ejemplo, una carpeta de documentos- que tienen su representación dentro de lo virtual -la representación de esta carpeta - y la realidad de lo que se representa -la estructura de almacenamiento de los datos.

1.2. ORÍGENES DE LAS INTERFACES GRÁFICAS DE USUARIO

En 1973 Xerox inició la revolución de las interfaces gráficas, dando así el camino a muchas otras hasta obtener lo que hoy en día observamos y manejamos en la computación.

Para los usuarios, las interfaces gráficas más amigables y llamativas de los sistemas operativos han sido una constante en la computación; pero eso no siempre fue así.

Xerox Alto. El Alto fue el primer sistema en reunir todos los elementos de la Interfaz Gráfica de Usuario. El Alto se diseñó y fue construido por Xerox[11] para la investigación y, aunque se donaron varios de estos sistemas nunca fueron vendidos. El Xerox Alto fue diseñado para ser pequeño, poderoso, para trabajar en computadoras de oficina con personal que contaba con habilidades en información gráfica, y para compartir información fácilmente.

Cuenta la leyenda que un día Steve Jobs visitó Xerox y vio que desarrollaron un entorno gráfico con uso de mouse, y aprovechando que a los directivos de Xerox no le interesaba, utilizó su idea para Apple, y llevo a cabo un sistema más robusto donde agrego los menús en la parte de abajo, una barra de menú, un sistema menú Apple, un copiado y pegado moderno; por ello Xerox después solo utilizo a Alto para crear un solo procesador de documentos, el Xerox 8010 o "Star Information System" (Sistema de Información Estrella 8010) introducida en 1981. Fue el primer sistema comercial en incorporar varias tecnologías que han llegado a ser hoy en día corrientes en computadores personales, incluyendo la pantalla con bitmaps en lugar de solo texto, una interfaz gráfica de usuario basada en ventanas, iconos, carpetas, mouse, red Ethernet, servidores de archivos, servidores de impresoras y e-mail.

[11] El desarrollo y diseño de las copiadoras láser era de Xerox.

Las características sobresalientes de algunas aplicaciones de Alto son:

Aplicación	Características
Net Executive	• Manejo de nombre (archivos) largos e insensibles. • Manejo de la barra espaciadora para autocompletar. • Manejo de signo "?" para sustitución de caracteres y así desplegar todas las opciones posibles. • Manejo de la red (cargar programas).
Neptune Directory Editor	• Utiliza Mouse. • Botones Gráficos. • Lista de archivos (no con iconos ni pinturas).
Maze War[12] (el original 3D)	• Primer juego de disparador. • Conectó una red de computadoras.
Bravo	• Su caracterización fue su filosofía de: "what you see is what you get", ("lo que ves es lo que obtienes"). • Variedad de tipo de caracteres y opciones para dar formato a documentos.
Smalltalk (lenguaje de alto nivel)	• Proporcionaba su propio ambiente de GUI que incluía pop-up (muestra y oculta). • Ventanas e imágenes que después se llamaron "iconos".
Star[13].	• Sistema de información. • Basado en cuatro principios: ver y apuntar, descubrimiento progresivo (mostrar y ocultar), uniformidad en todas las aplicaciones y "what you see is what you get". • Servicio de archivos, e-mail e impresión. • Lenguajes con Unicote. • Representación de archivos con "iconos" básicos. • Utilización del Mouse para copiar, arrastrar y crea un nuevo documento. • Especificaciones de documento como tipo de texto, fondo y color de contorno. • Los bordes de las imágenes cubrían toda la pantalla. • Contaba con un icono de basura "Waste Basket".

[12] El original fue escrito para una maquina Imlac (una maquina basada en gráficos de vector).

[13] Se comenzó a desarrollar en 1977, se escribió con códigos como Dolphin y Dorado, al final se escribió en Xerox´s MESA y ya no era compatible con Alto. En 1985 fue renovado para Xerox 6085 (Sistema de computadora profesional) y reforzado en sistemas como Apple Lisa, Macintosh, GEM, y Windows.

Global View[14].	• GUI bien diseñada. • Las ventanas contaban en sus extremos con botones para cambiarla de tamaño. • Contaba con botones para guardar, aplicar, hacer o cerrar colocados en la barra de título. • En vez de utilizar cajas para menús de pop-up (mostrar y ocultar) o alarmas, contaba con una sola caja de mensajes localizada en la parte superior de la ventana y siempre presente. • Los documentos solo se abrían para solo lectura y se hacia uso del "edit" para modificar. • Los menús de contexto eran mostrados pulsando el botón de la izquierda y derecha del Mouse. • Un botón que se encontraba en la parte superior izquierda de la ventana mostraba todas las ventanas minimizadas. • Permitía elegir cualquier opción del menú aunque no fuera aplicable pero desplegaba un mensaje de error.

Tabla *1.2.1*. Caracteristicas de Alto.

VisiCorp Visi On[15]. Primer escritorio destacado por sus GUI's para la IBM PC. Se permitió su uso en estaciones de trabajo corporativas, aunque fue diseñado para ser portátil en las OS como CP/M o Unix, también con CPU´s además de los 8086, y lo hacia proporcionando un número de máquina, la "máquina virtual" (máquina de Visi), el control de Visi On (Visi Host) era una máquina específica.

Aplicación	Características
VisiCorp Visi On	• Hacia uso del Mouse. • Interfaces consistentes. • No utiliza iconos, contaba con gráficos mas orientados a texto. • Podía trabajar al mismo tiempo con múltiples aplicaciones. • Consistía de una Application Manager (GUI), Accesories, Graph (Programa de gráficos), Word (Procesador de palabras), Calc (hoja de cálculo).

Tabla *1.2.2*. Características de VisiCorp Visi On.

GEM. Es un DOS se realizó en 1985 por la Investigación digital, fue la mas popular hasta que Microsoft Windows le ganó, había tres aplicaciones para correr en GEM, el escritorio GEM después se usó en View MAX, el manejador de archivos para DR-DOS (los mostraba como texto en ves de como iconos y eran ordenados por nombre, fecha, tamaño o por tipo).

Deskmate. En sus primeras versiones sólo se utilizo en modo texto y estaba disponible para modelos TRS-80; las versiones 2 y 3 eran con gráficos y corrían en Tandy 1000 PC, después en MS/PC-DOS y estando al mismo nivel que: GEM, Visi On o Windows 2.0; su popularidad fue en 1987 aproximadamente; consistía de opciones

[14]Versión 2.1 fue realizada en 1996, descendiente de ViewPoint escritorio de Xeros Star, comercializado con la idea de que procesaba documentos aunque podía hacer más.

[15] Fue el primer sistema de escritorio destacado por sus GUI´s para la IBM PC. La leyenda dice que Bill Gates vio una demostración de esté en 1982 y quedo fascinado con la idea, Microsoft no tenía nada en ese entonces, y con está visita comenzó lo que hoy es Windows.

avanzadas como: menú para comunicaciones, colores, protector de pantalla, uso de mouse, datos, consideración del tiempo, impresoras.

DESQview/X. La Application Manager es la primer DESQview/X con interfaz de usuario, que podía correr en un shell o sin él, lanzaba aplicaciones de menú que aparecían en pantalla; la interfaz consistía básicamente en porciones de muchos botones, donde para seleccionar una aplicación debías realizar doble clic, uno para seleccionar el botón y otro para abrir la aplicación, incluía un editor de iconos.

Amiga. Se introdujo en 1985 con gráficos de buen color, resoluciones múltiples, sonido estereofónico, y multitasking del pre-emptive que la hicieron una buena máquina en las aplicaciones multimedia y de juegos. Se creo en el escritorio (desktop) llamado Workbench, sus iconos aparecían en la pantalla principal y los fólderes se abrían en nuevas ventanas, los menús estaban colocados como en la MacOS, en la parte superior de la pantalla, mostraba accesorios, bloc de notas, una calculadora, un reloj, y un editor de iconos, también incluía un programa de cliente de e-mail simple.

RISC OS[16]. Buscaba aprovechar al máximo el espacio de la pantalla debido a ello en vez de tener una barra de menú en la parte superior de la pantalla, creó un estilo, hacer aparecer menús que eran accesibles dando clic al botón de en medio del mouse, contaba con una barra de iconos al fondo de la pantalla que representaban programas para ejecutar, iconos para los discos (como disquetes), el logotipo de Acorn donde aparece un menú y se podía acceder a funciones del sistema.

El estilo de las ventanas era diferente a otras GUI's, como por ejemplo el botón de mover hacia atrás la ventana de otras se encontraba en la parte superior izquierda en vez del botón de cerrar. El símbolo de cruz "X" cierra la ventana pero no la aplicación, del lado derecho superior de la ventana se encontraba un botón para modificar el tamaño de la ventana pero no para darle el tamaño completo de la pantalla, el área de fondo es llamada "*pinboard*" (pantalla) y se utiliza para colocar los archivos, directorios u otras aplicaciones que se pueden colocar en *n* diferentes lugares del "pinboard".

BeOS[17]. Es un OS poderoso diseñado especialmente para el uso de escritorios multimedia. Está edición personal podía ser instalada en Windows 95/98 y podías acceder a ella dando clic en el icono de terminar con Windows y comenzar BeOS pero debido a problemas de licencia y acuerdos con Microsoft ya no fue posible.

Esta aplicación permitía colocar los iconos y fólderes directamente en el escritorio, la barra "Deskbar" se colocó del lado superior derecho, sus rasgos mas nombrados era el uso de etiquetas en vez de las típicas barras de título que quitaban espacio a la pantalla, estas etiquetas le daban al sistema de ventanas una apariencia única; al pulsar el botón del logotipo de BeOS en el escritorio se desplegaba un menú con programas que se ejecutaban al dar clic, al Deskbar era posible colocarlo en las esquinas, abajo o en la parte superior de la pantalla; si los programas contaban con múltiples ventanas, con solo darle clic en el icono indicado se desplegaba las lista de ventanas abiertas y era posible seleccionarlas; los archivos se podían ver como iconos

[16] Un programa popular Newlook mejora su apariencia, creando las barras de movimiento (scroll bars), la barra de título entre otros controles con apariencia 3D.

[17] Originalmente fue diseñado para sistemas de computadora comunes como BeBox que contaba con multimedia de E/S, después fue trasportado a Macintosh y finalmente a PC.

normales, pequeños o bien con detalle; ofrecía varios escritorios virtuales llamados espacios de trabajo (“Workspaces"), cada uno con su propio tipo de fondo y resolución de pantalla y además de ser posible configurar el número; contaba con protectores de pantalla, era posible configurar los menús, la apariencia de la barra (scroll bar), trucos para cambiar la apariencia de la barra de título donde puede ser decorada como BeOS, AmigaOS, MacOS 8 y Windows 9x, además de poder darle doble clic y minimizar la ventana en vez de maximizarla.

Contaba con media player el cual podía tocar diferentes tipos de formatos de archivos, la partición de discos se podía realizar gráficamente, los disquetes debían ser montados con el botón derecho del mouse ya que no los monta automáticamente, contaba con una aplicación Valet que se utilizaba para instalar software que estaba empaquetado y podía actualizarse o desinstalarse, también era posible por medio de unzipping o usando una aplicación llamada Expand-O-Matic.

OS/2. Diseñada originalmente por Microsoft e IBM (Intemational Business Machines) en 1987, después cada uno realizó su propia versión con su propio nombre; IBM realizo la IBM OS/2 Versión 2, y mientras tanto Microsoft tomo algo de la tecnología y realizo Windows NT; la MS OS/2 fue patentada por Microsoft y a su vez IBM también tenían sus derechos de propiedad literaria.

El OS/2 Workplace Shell[18] podía tener diferentes iconos de archivos, alguna aplicación, fólderes (en forma de árbol con detalles), u otros objetos; hacia un uso excesivo del botón 1 y 2 del mouse, el botón del centro del mouse se utilizaba para arrastrar iconos aunque podía ser modificada esta opción; también era excesivo el uso del despliegue de menús aunque tenia opciones interesantes como poder crear otro objeto con un atajo, es decir, se podía reproducir otro objeto teniendo la liga del original; también era posible arrastrar los iconos y colocarlos en la papelera donde eran eliminados inmediatamente y no podían ser recuperados; contaba con su editor de iconos.

El sistema de fólderes al ser ejecutado contaba con opciones y herramientas para configurar el OS/2; existían conexiones de los discos locales, impresoras, red e Internet, las ventanas minimizadas desaparecían y para ser recuperadas tenías que acudir a la lista de ventanas (Window List displays) abiertas para recuperarla, las cajas de dialogo no existían, para ello existía un botón de “aplique” (o “ok”) y la ventana simplemente se cerraba.

El Desktop Manager era amigable como Windows 3.x excepto por los programas que se abren en nuevas ventanas en vez de en la misma ventana. Los iconos del DOS y el de Print Manager eran siempre visibles en el escritorio como programas minimizados, al oprimir el icono se ejecutaban ya sea el DOS (sesión de MS-DOS) o el Print Manager (programa que corre los trabajos de impresión).

Todas estas interfaces han progresado, por lo que también se pueden clasificar en 5 generaciones de Interfaces Gráficas de Usuario:

---

[18] IBM implemento el Workplace Shell utilizando el diseño orientado a objetos ("Object-oriented"), ahora todo era llamado objetos, como los archivos, programas, discos, fólderes, etc.

Primera generación: aparición de los CRT´s que son interfaces de preguntas y órdenes, basados en el diálogo que establece la aplicación con el usuario a través de la introducción de una instrucción con formato fijo.

Segunda generación: las mejoras fueron en la introducción de menús simples, la maquina los mostraba y el usuario introducía el código asociado. Comenzaron las primeras herramientas de carácter gráfico (ratón); se hizo necesario la introducción de cadenas de acceso rápido; existió una mayor organización y diseño de las Interfaces de Usuario (IU *Interface User*).

Tercera generación: aparición de los sistemas gráficos, presentaban ventanas, iconos, menús desplegables y elementos apuntadores, visualizaban diferentes tipos de información simultanea, elementos de asociación gráfica que reducen el uso del teclado para especificar órdenes; mayor cuidado en el diseño, herramientas más complejas y con mejora progresiva que apoyan cada vez mas al usuario.

Cuarta generación: se une a la tercera el hipertexto y la multitarea.

Quinta generación: asociada a las nuevas tendencias hipermedia y como objetivo final la manipulación directa. Las GUI´s basadas en realidad virtual ahora son usadas con más frecuencia en las investigaciones. Muchos grupos de investigación en Norteamérica y Europa están trabajando actualmente en la interfaz de enfoque del usuario o ZUI (*Zooming User Interface*), que es un adelanto lógico en las GUI´s.

La siguiente Tabla 1.2.3 muestra las diferentes GUI's que han existido desde sus inicios y que han servido como referencias para progresar en esté ámbito y contar con lo que hasta el momento se tiene sobre investigación y realización de Interfaces.

GUI	Aplicaciones
Xerox	Xerox Alto(1973), Star(1977), GlobalView 2.1 (1996)
Visi On	Visi Corp (1982)
GEM	GEM 1.1 (1985), 2.0 (1986), 3.11 (1988), Atari TOS 1.0 (1985)
Deskmate	Tandy DeskMate 3.69 (1987)
GEOS	GEOS For the Commodore (1985), GEOS for the Apple II (1988), GeoWorks (PC/GEOS) Versión 1.2, NewDeal 3.2, BreadBox.
Desqview/X	Desqview/X Versión 2.1
AmigaOS	AmigaOS 3.5 (1985)
RISC OS	RISC OS (Versión 3 y 4)
BeOS	BeOS 5.0 Edición Personal
QNX	QNX 1.44 y 6.2.1
OS/2	OS/2 V1.3, 2.0, Warp 4
Apple	Apple Lisa (1983), Apple II GUI's, Apple Macintosh (1984), Apple MacOS X.
Linux/Unix	Mandrake Linux 9.0 con KDE 3.0.3, Red Hat 8.0 / con GNOME/Nautilus 2.06, IRIX Interactive Desktop 6.5, Solaris 8 CDE y OpenWindows, Wine-Windows, etc.
Windows	ReactOS 0.2, 98Lite Version 1.3, 2,4.5, Microsoft Windows 1983 pre-Versión, versiones: 1.0 ,1.x ,2.x, 3.0, 3.10 , 3.11, 3.2, 3.1,95 (1995) ,98,ME (Versión 4.9), Windows NT 3.1 (1997) ,3.51,4.0, 2000(NT 5.0), XP (NT 5.1) (2001) , Server 2003 (NT 5.2).
Windows Shells	Microsoft BOB, Central Point Desktop, Norton Desktop for Windows, PubTech File Organizer 2.11 y 3.10, HP NewWave Working Model, Packard Bell Navigator 1.1 y 3.5, Calmira II, Workplace Shell for Windows.

Tabla *1.2.3.* Esta tabla muestra las diferentes Interfaces Graficas de Usuario desde sus inicios.

1.3. ALTERNATIVAS PARA LA CONSTRUCCIÓN DE INTERFACES GRÁFICAS DE USUARIO

Hoy en día las herramientas de GUI se encuentran compitiendo en el mundo de la Web por lo que los aficionados se olvidan un poco de las GUI's de escritorio.

Sin embargo existen muchas alternativas para la creación de Interfaces Gráficas de Usuario[19], cave mencionar que se seleccionaron las más significativas -observar la Tabla 1.3.1- de acuerdo con ciertos puntos que se consideraron relevantes y que cumplen con las necesidades de nuestro objetivo, se comenzó por elegir las herramientas que fueran software libre, es decir, con licencia GNU[20], de las cuales también existen varias opciones, pero de estás, se eligieron las que hasta el momento -de la elaboración- contaban con actualizaciones, portabilidad en la mayoría de las plataformas, compatibilidad con otro software principalmente OpenGL, y aceptación de los usuarios.

Herramienta para GUI	Página	Plataformas	Licencia	Comentarios
WxWidgets	www.wxwidgets.org/	Windows (y sus variantes), Unix (y sus variantes), Linux, MacOS, OS/2.	*LGPL* ("Lesser General Public License"), Open Source.	Anteriormente wxWindows, diseñado para el lenguaje de C++.
Qt	www.trolltech.com/	Windows (y sus variantes), Unix (y sus variantes).	Qt cuenta con dos versiones de licencia: Comercial y LGPL.	Diseñada por Trolltech y escrita en C++, cuanta con diferentes módulos, entre ellos una de OpenGL.
GTK	www.gtk.org/	Windows (y sus variantes), Unix (y sus variantes), MacOS X, BeOS, en la consola de Linux.	LGPL, puedes diseñar: Open Software, Free Software y Comercial.	Realizada en lenguaje C, aunque existe la versión en C++. Esta basada en tres librerías Glip, Pango y Atk.
Java	www.java.sun.com/	Windows, Linux, Apple y Solaris.	No es software libre pero si gratuito.	Orientado a objetos. Cuenta con un enlace para la librería OpenGL.
LibUFO	www.libufo.com/	Windows, Unix.	LGPL	Es una librería escrita en C++ para diseñar principalmente Interfaces Graficas de Usuario para OpenGL.

Tabla *1.3.1.* Aquí se muestran algunas herramientas para crear GUI's, con sus respectivas ligas, plataformas donde se soporta, licencias y algunas características.

[19] En la siguiente página puedes encontrar la mayoría de las herramientas libres (Free Toolkits) para GUI's, su tipo de licencia y sus respectivas ligas, http://www.free-soft.org/guitool/ (consultada el 24/10/2005).

[20] El proyecto GNU fue iniciado por Richard Stallman con el objetivo de crear un sistema operativo completamente libre: el sistema GNU. El 27 de septiembre de 1983 se anunció públicamente el proyecto por primera vez en el grupo de noticias net.unix-wizards. La licencia completa se encuentra en: http://www.gnu.org, y la página "The Free Software Foundation" es: http://www.fsf.org.

wxWidgets

Fue diseñado en 1992 en el Instituto de Aplicaciones de Inteligencia Artificial -Artificial Intelligence Applications Institute-, en la Universidad de Edinburgh, por Julian Smart. Julian diseñaba la herramienta meta-CASE llamada Hardy que necesitaba correr en Windows, así como en estaciones de trabajo de X-Unix, y las herramientas existentes y comerciales multiplataforma eran costosas para un proyecto experimental, así que su única alternativa era crear su propia herramienta wxWidgets[21](w para Windows y la x para X) comenzando a soportar Xview, MFC 1.0, y AIAI permitió el lanzamiento al Internet.

Es un Framework parecido a MFC, especializado en el desarrollo de aplicaciones multiplataforma en lenguaje C++ aunque también existe para Python y Perl, es multiplataforma, soporta Windows, Linux, Mac OS X , Unix y sus variantes, Solaris, Plataformas Embedded (inicios de investigación[22]), también en plataformas móviles como Microsoft Pocket PC, y Palm OS; se distribuye bajo licencia Open Source[23] y GNU LGPL (Library General Public License[24]) permitiendo utilizarla para desarrollos comerciales, siempre y cuando estos desarrollo no usen código distribuido bajo alguna licencia GNU.

Cuenta con una parte denominada *wxBase* que incluye clases como *wxString*, clases para el manejo de archivos y directorios de manera independiente del sistema, funcionalidades como: gráficos 2D, 3D con OpenGL, Bases de Datos (ODBC), Redes, Impresión, Hilos, visión e impresión del HTML, un sistema de archivos virtual y cuanta con algunos IDEs[25].

La razones por las que se podría elegir wxWidgets son además de sus ya mencionadas características es que cuanta con soporte, documentación en Internet, ayuda en línea, foros, tutoriales en diversos formatos, desarrolladores en la red por lo que se percibe interés y un futuro, cuenta con un libro de 1000 páginas imprimibles de documentación y en línea, sistema flexible a eventos, llamadas a gráficos como líneas, rectángulos con esquinas redondeadas, etc. Soporte de MDI (Multiple Document Interface), puedes crear tus DLLs sobre Windows, y librerías dinámicas en Unix -Figura 1.3.1-.

---

[21]Inicialmente wxWindows que tuvo que ser cambiado al nombre wxWidgets por que Microsoft (“Bill Gates”) puso una demanda a finales del 2003 por que podría existir confusión con su Sistema Operativo.

[22]Si se desea saber acerca de este proyecto ver información sobre wxUniversal.

[23]Empezó a utilizarse en 1998 por usuarios de la comunidad del software libre, su licencia es similar a free-software, su definición se puede encontrar en: www.opensource.org/docs/definition.php/.

[24]A grandes rasgos es software libre (es recomendable que para mayor y mejor información ingreses a http://www.gnu.org), pero no tiene un copyleft, es decir, cualquiera que redistribuye el software, con o sin cambios, debe dar la libertad de copiarlo y modificarlo lo cual garantiza que cada usuario tiene libertad, también permite que el software se enlace con módulos no libres; es recomendada para circunstancias especiales.

[25]Integrated Development Environment, es decir, un entorno integrado de desarrollo.

wxWidgets API								
wxWidgets Port								
wxMSW	wxGTK	wxX11	wxMotif	wxMac	wxCocoa	wxOS2	wxPalmOS	wxMGL
Platform API								
Win32	GTK+	Xlib	Motif/ Lesstif	Carbon	Cocoa	PM	Palm OS Protein APIs	MGL
Operating System								
Windows/ Windows CE	Unix/Linux			Mac OS 9/ Mac OS X	Mac OS X	OS/2	Palm OS	Unix/ DOS

Figura 1.3.1. El alcance de wxWidgets.

Qt

El equipo de diseñadores de Trolltech desarrollo Qt, en 1992 y la primera edición comercial fue realizada en 1995; Qt es un Framework[26] multiplataforma, Windows y sus variantes, Mac OS X, Unix y sus variantes como Linux, Solaris, IRIX,etc.; está escrito en C++, es una librería de 400 clases para GUI's y para no GUI's, cuenta con diferentes módulos para una buena funcionalidad en base de datos, establecimiento de una red (operación y protocolos), XML, internacionalización (utiliza Unicode manejando por lo tanto idiomas como árabe, chino, hebreo, japonés y coreano), manejo de OpenGL. Es elegante, intuitivo y completamente orientado a objetos.

La Figura 1.3.2. muestra el stack de capas de un equipo corriendo aplicaciones de Qt, y se muestra que al correr nativamente, no existen capas de emulación ni maquinas virtuales. Para desarrollar en Qt solo es necesario aprender una sola API para escribir aplicaciones que corren casi en cualquier lado –ver Apéndice G-. Qt tiene un set completo de widgets estándar, y permite escribir controles personalizados. Encapsula cuatro diferentes APIs de plataforma-especifica, y los APIs para manejo de archivos, redes, manejo de procesos, cadenas (threading), acceso a base de datos, etc.

Las aplicaciones más populares que usan Qt, son por supuesto las de KDE. Las librerías de KDE extienden y complementan a las de Qt, así que las aplicaciones KDE dependen de Qt. Hay otras aplicaciones interesantes hechas con Qt, pero que no son KDE. Las más populares son Scribus, JahShaka, Psi, y algunas otras.

[26] En esté esta basado el escritorio de GNU/Linux KDE.

Figura *1.3.2.* Qt Stack

En el diseño de Interfaces es rápido, tiene un buen rendimiento y con la sensación de la plataforma nativa, cuanta con dos ediciones[27]:

- **Qt Commercial Editions**: para el desarrollo de software comercial y el costo esta basado en el número de plataformas en las que se desee soporte, está a su vez cuenta con tres diferentes ediciones comerciales.
  - **Qt Console Edition**: esta optimizada para el desarrollo de poderosas GUI's independientes del servidor de aplicación.
  - **Qt Desktop Light Edition**: incluye todas las funcionalidades requeridas para niveles de entrada de la GUI para el desarrollo de aplicaciones.
  - **Qt Desktop Edition**: provee accesos extensos para niveles empresariales.
- **Qt Open Source Edition**: que es utilizada para el desarrollo libre y abierto; licencia pública GNU-GPL (General Public License[28]).

GTK

Es una herramienta multiplataforma desarrollada por el equipo de GTK+, es software libre, forma parte del proyecto GNU con licencia LGPL con ciertos términos. Esta basada en tres librerías:

- **GLip**: biblioteca de bajo nivel, es la estructura básica de GTK+[29] y GNOME.
- **Pango**: biblioteca para el diseño y reenderezado texto y núcleo para manejar las fuentes, hace hincapié especialmente en la internacionalización (uso de Unicode).
- **Atk**: biblioteca que proporciona características para un interfaz con una gran accesibilidad para personas discapacitadas o minusválidos por medio de un soporte amplio de dispositivos como lupas, lectores de pantalla, entradas de datos como teclado o mouse, screen readers, etc.

[27] Para mayor información consulte la página de Qt: www.trolltech.com/

[28] Es una licencia de software libre de tipo copyleft (cualquiera que redistribuye el software, con o sin cambios, debe dar la libertad de copiarlo y modificarlo más, garantiza que cada usuario tiene libertad), es recomendada para la mayoría de los paquetes de software; es recomendable que para una mayor y mejor información acerca de esta y otras licencias visites la página: http://www.gnu.org.

[29] Disponible -08/03/2006- la versión 2.8.14.

Aunque GTK está completamente escrito en lenguaje C cuanta con una *API* (Application Programming Interface[30]) orientada a objetos la versión GTKmm[31] con un diseño estructurado pero usando clases en C++.

Esta diseñada no solo para soportar ciertos tipos de lenguajes como C/C++, si no también soporta Perl, Python y Java, entre otros que están siendo desarrollados como Ada, C#, JavaScript, Pascal, PHP, TCL, Ruby, Eiffel, etc. Inicialmente fue diseñada y usada por GIMP y construida sobre GDK (el conjunto de herramientas de dibujo del Gimp, manipulación de imágenes de GNU), que a su vez no es mas que un wrapper de las funciones de Xlib, hoy en día GTK+ es usada por un largo numero de aplicaciones y herramientas usadas en el proyecto GNU Network Object Model Environment (GNOME); el inconveniente de GTK es que solo esta diseñada para desarrollar Interfaces gráficas de usuario y no consideras a las bases de datos, a las redes, etc.

JAVA

Java es un lenguaje desarrollado por Sun, con la intención de competir con Microsoft en el mercado de la red. Sin embargo, su historia se remonta a la creación de una filial de Sun (FirstPerson) enfocada al desarrollo de aplicaciones para electrodomésticos, microondas, lavaplatos, televisión, etc. Esta filial desapareció tras un par de éxitos de laboratorio y ningún desarrollo comercial.

El primer proyecto en que se aplicó este lenguaje en 1991 recibió el nombre de proyecto "*Green*" y consistía en un sistema de control completo de los aparatos electrónicos y el entorno de un hogar. Para ello se construyó un ordenador experimental denominado *7 (Star Seven).

El sistema presentaba una interfaz basada en la representación de la casa de forma animada y el control se llevaba a cabo mediante una pantalla sensible al tacto. Posteriormente se aplicó a otro proyecto denominado VOD (Video On Demand) en el que se empleaba como interfaz para la televisión interactiva. Ninguno de estos proyectos se convirtió nunca en un sistema comercial, pero fueron desarrollados enteramente en un Java primitivo.

Bill Joy, cofundador de Sun y uno de los desarrolladores principales del Unix de Berkeley, juzgó que Internet podría llegar a ser el campo de juego adecuado para disputar a Microsoft su primacía casi absoluta en el terreno del software, y vio en Oak el instrumento idóneo para llevar a cabo estos planes. James Gosling, desarrolló un lenguaje derivado de C++ que intentaba eliminar las deficiencias del mismo. Llamó a ese lenguaje Oak. Cuando Sun abandonó el proyecto de FirstPerson, se encontró con este lenguaje y, tras varias modificaciones (entre ellas la del nombre), decidió presentar el lenguaje Java en sociedad en agosto de 1995.

Java tenía que funcionar en numerosos tipos de CPU's, y por tanto se busco que

[30]Es un conjunto de especificaciones de comunicación entre componentes de software, proporcionando un conjunto de funciones de uso general.
[31]GTKmm es la versión C++ de GTK+, su pagina http://gtkmm.sourceforge.net.

fuera “independiente de la plataforma”, esto a permitido a Java convertirse en el lenguaje para la creación de aplicaciones (y con mejor auge en el ambiente de Internet).

El éxito de Java reside en varias de sus características. Java es un lenguaje sencillo, o todo lo sencillo que puede ser un lenguaje orientado a objetos, eliminando la mayor parte de los problemas de C++, que aportó su granito (o tonelada) de arena a los problemas de C. Es un lenguaje independiente de plataforma, por lo que un programa hecho en Java se ejecutará igual en un PC con Windows que en una estación de trabajo basada en Unix. También hay que destacar su seguridad, desarrollar programas que accedan ilegalmente a la memoria o realizar caballos de Troya es una tarea propia de titanes.

Cabe mencionar también su capacidad multihilo, su robustez o lo integrado que tiene el protocolo TCP/IP, lo que lo hace un lenguaje ideal para Internet. Pero es su sencillez, portabilidad y seguridad lo que le han hecho un lenguaje de tanta importancia.

Sun describe a Java como "simple, orientado a objetos, distribuido, interpretado, robusto, seguro, de arquitectura neutra, de altas prestaciones, multitarea y dinámico". Soporta las tres características propias del paradigma de la orientación a objetos: encapsulación, herencia y polimorfismo.

La portabilidad se consigue haciendo de Java un lenguaje con un esquema que han seguido otros lenguajes, como por ejemplo Visual Basic. Sin embargo, nunca se había empleado como punto de partida a un lenguaje multiplataforma ni se había hecho de manera tan eficiente. Cuando Java apareció en el mercado se hablaba de que era entre 10 y 30 veces más lento que C++. Ahora, con los compiladores JIT (*Just in Time*) se habla de tiempos entre 2 y 5 veces más lentos. Con la potencia de las máquinas actuales, esa lentitud es un precio que se puede pagar sin problemas contemplando las ventajas de un lenguaje portable.

En simplicidad Java ofrece toda la funcionalidad de un lenguaje potente, pero sin las características menos usadas y más confusas de éstos. C++ es un lenguaje que adolece de falta de seguridad, pero C y C++ son lenguajes más difundidos, por ello Java se diseñó para ser parecido a C++ y así facilitar un rápido y fácil aprendizaje.

Java elimina muchas de las características de otros lenguajes como C++, para mantener reducidas las especificaciones del lenguaje y añadir características muy útiles como el garbage collector (reciclador de memoria dinámica). No es necesario preocuparse de liberar memoria, el reciclador se encarga de ello y como es de baja prioridad, cuando entra en acción, permite liberar bloques de memoria muy grandes, lo que reduce la fragmentación de la memoria.

En cuanto a distribución Java se ha construido con extensas capacidades de interconexión TCP/IP. Existen librerías de rutinas para acceder e interactuar con protocolos como http y ftp. Esto permite a los programadores acceder a la información a través de la red con tanta facilidad como a los ficheros locales.

Se dice que Java tiene robustez por que realiza verificaciones en busca de problemas tanto en tiempo de compilación como en tiempo de ejecución. La comprobación de tipos en Java ayuda a detectar errores, lo antes posible, en el ciclo de

desarrollo. Java obliga a la declaración explícita de métodos, reduciendo así las posibilidades de error. Maneja la memoria para eliminar las preocupaciones por parte del programador de la liberación o corrupción de memoria.

Dado que Java es un lenguaje orientado a objetos, es imprescindible entender qué es esto y en qué afecta a nuestros programas. Desde el principio, la carrera por crear lenguajes de programación ha sido una carrera para intentar realizar abstracciones sobre la máquina. Al principio no eran grandes abstracciones y el concepto de lenguajes imperativos es prueba de ello. Exigen pensar en términos del ordenador y no en términos del problema a solucionar. Esto provoca que los programas sean difíciles de crear y mantener, al no tener una relación obvia con el problema que representan No abstraer lo suficiente.

Muchos programas intentaron resolver este problema alterando la visión del mundo y adaptándola al lenguaje. Estas aproximaciones modelaban el mundo como un conjunto de objetos o de listas. Funcionaban bien para algunos problemas pero no para otros. Los lenguajes orientados a objetos, más generales, permiten realizar soluciones que, leídas, describen el problema. Permiten escribir soluciones pensando en el problema y no en el ordenador que debe solucionarlo en último extremo.

La seguridad en Java tiene dos facetas. En el lenguaje, características como los punteros o el casting implícito que hacen los compiladores de C y C++ se eliminan para prevenir el acceso ilegal a la memoria. Cuando se usa Java para crear un navegador, se combinan las características del lenguaje con protecciones de sentido común aplicadas al propio navegador.

El lenguaje C, por ejemplo, tiene lagunas de seguridad importantes, como son los errores de alineación. Los programadores de C utilizan punteros en conjunción con operaciones aritméticas. Esto le permite al programador que un puntero referencia a un lugar conocido de la memoria y pueda sumar (o restar) algún valor, para referirse a otro lugar de la memoria. Si otros programadores conocen nuestras estructuras de datos pueden extraer información confidencial de nuestro sistema. Con un lenguaje como C, se pueden tomar números enteros aleatorios y convertirlos en punteros para luego acceder a la memoria.

Básicamente un programa en Java puede ejecutarse como:

- **Stand Alone**: Aplicación independiente.
- **Applet**: Una aplicación especial que se ejecuta en el navegador del cliente.
- **Servlet**: Una aplicación especial sin Interfaz que se ejecuta en el servidor.

Java funciona en varias plataformas sin necesidad de recompilar, tiene un Framework bien diseñado, avanzado y potente, no es software libre aunque si gratuito, corre bajo una *Máquina Virtual Java* (JVM), parte dependiente del hardware, y por tanto no realiza aplicaciones rápidas, existen máquinas virtuales para sistemas operativos Windows, Linux, Apple y Solaris; se distribuye como: JDK, SDK o J2SE, incluye el API de Java, el compilador y el JRE (Java Runtime Enviroment)[32]. Es posible

[32]Principalmente es el instalador de la Maquina Virtual de Java y los plugins necesarios para el/los

con java programar páginas Web dinámicas, con accesos a bases de datos, utilizando XML, con cualquier tipo de conexión de red entre cualquier sistema, es multithreaded (multihilvanado, en mala traducción), permite actividades simultáneas en un programa, soporta threads (llamados procesos ligeros), son básicamente pequeños procesos o piezas independientes de un gran proceso, soporte de OpenGL.

LibUFO

Es una librería diseñada por Johannes F. Schmidt, realizada en C++ para el diseño de GUI, principalmente para ser usado por OpenGL, cuenta con licencia GNU LGPL V.2, cuanta con los componentes estándar como botones, etiquetas, menús, etc. Cuenta con la particularidad de manejar sus estilos con archivos CSS (Cascading Style Sheets)[33], y también puedes crear interfaces vía XUL [34](eXtensible User interface Language, es un lenguaje XML[35] eXtensible Markup Language), cuenta con controladores de video como son GLX/X11, WGL/Win32 y el mas conocido LibSDL[36], cada uno es seleccionado y cargado en tiempo de ejecución, puedes trabajar ya contando con tu dibujo en OpenGL y agregar LibUFO o bien trabajar dentro de LibUFO tu dibujo de OpenGL.

navegador/es instalados en nuestro sistema; es distribuido también de forma independiente en un paquete llamado J2RE.

[33]Hojas de estilo en cascada denominadas en inglés Cascading Style Sheets, definen el estilo en diferentes niveles, definen el formato del texto, los encabezados, subtítulos, etc. Podrás encontrar una sección sobre CSS en las páginas de World Wide Web Consortium (W3C): http://www.w3c.org/.

[34]Diseñado para crear interfaces de manera fácil y rápida, está disponible en todas las versiones Windows, Mac, Unix y sus variantes, pero no es compatible con Internet Explorer (inconveniente), XUL es XML, para mayor información consulta la página de XULPlanet: http://www.xulplanet.org/.

[35]Lenguaje concebido para mostrar información, ayuda en la organización de contenidos y portables, para mayor información consulta: http://www.w3c.org/xml/.

[36]SDL (Simple Directmedia Layer) es una librería escrita en C que proporciona acceso sencillo y de bajo nivel a los recursos del sistema tales como la tarjeta de video, de sonido, mouse, teclado, joysticks, etc.

CAPÍTULO 2

ANÁLISIS

Capítulo 2. ANÁLISIS

2.1. DESCRIPCIÓN DEL SIMULADOR DE FLUJOS

La construcción de nuevas y más poderosas arquitecturas de cómputo es, en comparación con el desarrollo de software, la parte “fácil” del proceso de simulación numérica. Esta afirmación se basa en el problema conocido como la crisis del software que es la dificultad de construir sistemas simples de entender, mantener y extender, y que además, provean soluciones realistas y con buena precisión a problemas complejos. Este problema no es particular de las simulaciones numéricas y se da en distintas áreas de la ciencia de la computación, en las que un sistema de software robusto es necesario. Diferentes paradigmas de programación han sido desarrollados con el objetivo específico de eliminar el problema de la crisis del software. No obstante, estos “nuevos” paradigmas no se han aplicado directamente a problemas de simulación numérica, debido principalmente a que se tiene la percepción de que el rendimiento se ve afectado sustancialmente.

En la dinámica de fluidos, por ejemplo, el problema de la turbulencia no ha sido resuelto completamente porque existen distintos enfoques teóricos y numéricos. Adicionalmente, para simular problemas de turbulencia se requieren recursos de cómputo que superan a la mayoría de las arquitecturas de supercómputo existentes en nuestros días. El problema anterior se ha intentado resolver a través del uso de arreglos de computadoras personales (*clusters*) y recientemente del uso de *Grids*[1]. A pesar del impresionante avance en la capacidad de procesamiento y almacenamiento, muchas veces no es posible aprovechar este recurso debido a la falta de un desarrollo complementario de software. Actualmente se siguen utilizando códigos bastante complejos, cuyo mantenimiento es complicado y sólo se utilizan para resolver problemas particulares, el rehúso es prácticamente nulo.

Los objetivos principales del simulador de flujos comenzaron siendo el construir una herramienta para resolver problemas de convección natural en flujo laminar y turbulento, que fuera fácil de usar y modificar, y que permitiera utilizar arquitecturas de cómputo multiprocesador.

Como metas planteadas y logradas fueron las siguientes:

Se realizo la construcción del sistema mediante una metodología de desarrollo de software ordenada he incremental, que permitió tomar en cuenta los principales requerimientos de una modelación computacional.

Se escribieron de manera general los modelos matemático y discreto de las ecuaciones de balance en flujo laminar y turbulento, para implantar un conjunto de componentes genéricas que pudieran reutilizarse en distintos tipos de problemas. Así como una arquitectura del sistema simple, que se asemejara a la estructura de una modelación computacional.

[1] Organizaciones virtuales que conjuntan equipos de supercómputo muy sofisticados por medio de redes de alta velocidad, como Internet 2.

Lo mas importante y quizá la parte que ayudo a la realización de esta interfaz, fue que para dar solución a los problemas es siguiendo una misma metodología, con cambios mínimos, aún cuando fueran problemas distintos.

2.1.1. Dinámica de fluidos computacional

Dinámica de Fluidos Computacional o (CFD por sus siglas en inglés Computational Fluid Dynamics), es una de las áreas de la computación científica que mayor crecimiento ha tenido en los últimos años.

Esta nueva rama de la dinámica de fluidos, complementa a la teoría y la experimentación, y provee de una alternativa a bajo costo para simular flujos reales. Al igual que en otras áreas de aplicación, la simulación de flujos involucra un proceso que en algunos textos se conoce como modelación computacional, figura 2.11.1. En un principio, se tiene el problema en el mundo real. El fenómeno de interés se puede explicar mediante la aplicación de leyes o principios físicos. En este punto se decide a que nivel de aproximación se desea estudiar el fenómeno. Mediante un modelo matemático se traduce el fenómeno del mundo real en un conjunto de ecuaciones, cuya solución permitirá, en principio, entender el fenómeno. En la mayoría de las ocasiones, no existen técnicas matemáticas para resolver analíticamente las ecuaciones que describen el fenómeno del mundo real en un conjunto de ecuaciones que describen el fenómeno bajo estudio. Sólo en casos particulares muy simples se tiene esa ventaja. Cuando no es posible obtener una solución analítica, generalmente se recurre a métodos numéricos para obtener soluciones aproximadas.

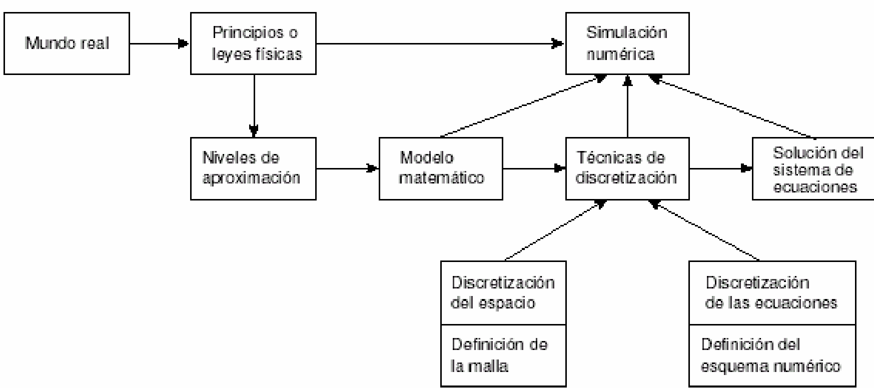


Figura *2.1.1.1* Diagrama general de una modelación computacional.

En este caso en particular, la dinámica de un flujo se puede describir mediante un conjunto de ecuaciones que incluyen a la ecuación de balance de masa o ecuación de continuidad, las ecuaciones de balance de cantidad de movimiento o ecuaciones de Navier-Stokes y la ecuación de energía. Estas ecuaciones diferenciales parciales, no lineales y acopladas, se conocen desde hace más de un siglo y es bien conocido que su solución puede presentar estructuras a diferentes escalas. La interacción entre todas las

escalas influye fuertemente en el comportamiento de la solución. En el lenguaje de Fourier, este comportamiento se describe como la interacción de diferentes frecuencias.

El sistema fue basado en clases C++, para resolver problemas de convicción natural en cavidades rectangulares. Durante la construcción se combinaron tres paradigmas de programación: Programación Orientada a Objetos (POO), Programación Genérica (PG) y Programación en Paralelo (PP). Las principales aportaciones de ese trabajo son:

Metodología de desarrollo de software para cómputo científico.

- A partir del modelo de desarrollo propuesto por Schimmel (el cual esta basado en el propuesto por Larman que a su vez utiliza el modelo unificado), se construye una nueva metodología combinándola con la programación extrema (eXtreme Programming).
- En la nueva metodología se incluyen las etapas de generalización y optimización del sistema, además, de las de construcción del modelo matemático y del modelo discreto, incluidas por Schimmel.

Sistema para solución de problemas de convicción natural en cavidades.

- Arquitectura simple, basada en un proceso general de modelación computacional.
- Clases generales polimórficas adaptables.
- Creación de un "metalenguaje" para fácil acceso a las herramientas.
- Uso de técnicas basadas en "*templates*" para la optimización.
- Paralelismo a través de descomposición de dominio, mediante el diseño de un algoritmo paralelo alternante de Schwarz.

2.1.2. Planteamiento de las ecuaciones de la mecánica de fluidos

Las ecuaciones de la mecánica de fluidos representan leyes de conservación o balance de cantidades físicas como la masa, cantidad de movimiento (momentum) y energía. Para obtener las ecuaciones, el fluido se considera continuo y se hace un balance de las diferentes cantidades sobre un elemento de fluido.

Las ecuaciones que gobiernan esté fenómeno se obtienen tomando en cuenta la aproximación de Boussinesq, es decir la densidad se considera constante excepto en el término de fuerzas de cuerpo, y las propiedades físicas restantes del material se consideran constantes siempre.

Las ecuaciones resultantes generalmente se utilizan en forma adimensional tanto para su discretización, como para su implementación. En la forma adimensional, aparecen parámetros que permiten hacer estudios de flujos en diferentes estados: laminar o turbulento. Aunque no existe una definición exacta de turbulencia, se puede decir que un flujo es turbulento cuando es irregular, consiste de un amplio rango de escalas de movimiento, se incrementa la difusividad, es completamente tridimensional, es muy disipativo. La simulación directa de flujos turbulentos o DNS (Direct Numerical Simulation), es posible sólo en algunos casos simples, pero en la mayoría de las

ocasiones se requieren recursos enormes de cómputo. Algunas técnicas se han desarrollado para evitar este problema y una de las más conocidas es la Simulación de Vórtices Grandes o LES (Large Eddy Simulation). Está técnica LES ayuda a simular flujos turbulentos en mallas gruesas. La base de esta técnica consiste en la aplicación de un filtro de convolución espacial a las ecuaciones gobernantes.

Las ecuaciones resultantes en general son bastante complejas y a pesar de que se conocen desde hace mucho tiempo, no exciten hoy en día un método analítico para encontrar sus soluciones. Por esta razón, se han desarrollado muchos métodos numéricos con los que es posible encontrar soluciones numéricas aproximadas.

Mediante el uso de las arquitecturas sofisticadas de cómputo y de las arquitecturas multiprocesador, ha sido posible resolver las ecuaciones gobernantes para diferentes tipos de flujos, con excelente exactitud y precisión. Como es sabido, es posible resolver el mismo problema usando diferentes técnicas numéricas. Entonces, la cesibilidad del código, además de la eficiencia, es importante en la solución de problemas numéricos, y es por ello que el desarrollador utilizó procesos de desarrollo ordenados y paradigmas de programación modernos.

2.1.3. Ejemplos

En esta parte veremos como usar y combinar cada una de las clases y funciones construidas en el simulador para resolver diferentes problemas.

El conjunto de clases desarrollado en el simulador puede usarse fácilmente para resolver diferentes tipos de problemas numéricos. En esta su primera versión sólo se consideraron problemas descritos en dominios rectangulares con mallas cartesianas. La ventaja del desarrollo es que para resolver la mayoría de los problemas que se presentaron se realizan los mismos pasos:

1. Incluir los encabezados de las clases que se deseen usar.
2. Declarar e inicializar las variables del problema.
3. Declarar el sistema lineal para almacenar los coeficientes de la discretización. Este sistema puede ser compartido cuando haya más de una ecuación por resolver.
4. Declarar y definir la malla del dominio.
5. Declarar y definir las variables dependiente (campos escalares y vectoriales sobre la malla) necesarias y las condiciones iniciales.
6. Declarar y definir las ecuaciones que se deben resolver.
7. Resolver la ecuación: calcular los coeficientes del sistema lineal y resolverlo.

En los ejemplos que siguen se muestra como se realiza cada uno de los pasos.

2.1.3.1. Difusión en una y dos dimensiones

El objetivo de mostrar estos ejemplos es para observar la similitud en el procedimiento a seguir para resolver los diferentes problemas en 1, 2 y 3 dimensiones, por lo que solo veremos la estructura general que se sigue para resolverlos, detallando un poco el primer ejemplo, en el Apéndice B se presentan y explican las clases implementadas; debo mencionar que no es la finalidad de este trabajo describir la manera de realización y solución del simulador; la metodología utilizada para la solución del simulador permitió a este trabajo ser realizado de una forma independiente, estructuralmente.

El primer problema es muy simple y tiene como objetivo describir el uso de las clases y módulos desarrollados en el simulador. Posteriormente, se hará referencia a esta descripción cuando se revisen problemas más complejos. En este caso se considera el problema de difusión de calor en una dimensión, el cual se describe en la figura 2.1.3.1.1[2].

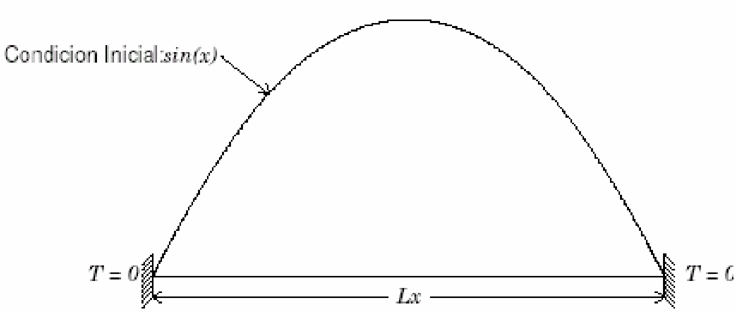


Figura *2.1.3.1.1.* Condiciones iniciales y de frontera para el problema de difusión en 1D.

El siguiente código figura 2.1.3.1.2 resuelve el problema en una dimensión y con las condiciones iniciales y de frontera mostradas en la figura 2.1.3.1.1:

```
1   #include "Utils/Common.h"
2   #include "Equations/EnergyEquation.h"
3   #include "Meshes/StructuredMesh.h"
4   #include "NumSchemes/Diffusion.h"
5   #include "Solver/Solver.h"
6
7   int main()
8   {
9       double longitud, dt, dx, tolerancia, error, Tleft, Tright;
10      int num_nodos, num_vols, iteracion, max_iter;
11  // ... inicializacion de variables ...
12
13      DiagonalMatrix<Tri>       A(num_vols);              // Matriz A
14      ScalarField1D             b(num_vols);              // Vector b
15      StructuredMesh<double, 1> mesh(longitud, num_nodos); // Malla
16      ScalarField1D             T(mesh.getExtentVolumes());// Campo escalar T
```

[2] Para una major comprensión de este y consecuentes ejemplos puedes consultar la tesis de Doctorado: De la Cruz Salas, Luis Miguel, "Cómputo Paralelo en la Solución Númerica de las Ecuaciones de Balance en Flujo Turbulento", pags. 87-118.

```
17
18        dx = mesh.getDelta(0);
19        firstIndex index;                       //
20        T = sin(PI * index * dx / longitud);  // Condiciones
21        T(T.lbound(firstDim)) = Tleft;        // iniciales
22        T(T.ubound(firstDim)) = Tright;       //
23
24        EnergyEquation<double, 1, Diffusion<double, 1> > energia(
25        energia.setLinearSystem(A, b);
26        energia.setGamma(1.0);
27        energia.setDeltas(mesh.getDeltas());
28        energia.setDeltaTime(dt);
29        energia.setDirichlet(LEFT_WALL, Tleft);
30        energia.setDirichlet(RIGHT_WALL, Tright);
31
32        while (error > tolerancia && ++iteracion < max_iter) {
33            energia.calcCoefficients();
34            Solver::TDMA(energia);
35            error = energia.calcError();
36            energia.update();
37            Output::printToFile_GP(T, iteracion, "temp.", dx);
38        }
39        return 0;
40    }
```

Figura *2.1.3.1.2.* Código del problema de difusión en 1D.

El código de arriba se explica a continuación:

Líneas *1-5*: En estas líneas se incluyen los archivos que contienen la definición de las clases que se necesitan para resolver el problema.

Líneas *13-14*: Declaración del sistema lineal. Se declara primero un objeto de la clase DiagonalMatrix. En este caso, el objeto A es una matriz tridiagonal (nótese el parámetro Tri en la declaración) que contiene el sistema lineal de ecuaciones. La matriz es de tamaño *num_vols* x *num_vols*. La declaración para problemas en dos y tres dimensiones se hace de manera similar usando como parámetro *penta* o *hepta* respectivamente.

Línea *15*: Declaración del objeto mesh que representa la malla del problema. En la declaración se usan los parámetros double y 1 que determinan la precisión y la dimensión del problema respectivamente. El objeto mesh toma como argumentos para su creación la longitud y el número de puntos en cada dirección.

Línea *16*: Declaración del objeto T que es un arreglo en donde se almacenará la solución del problema. El tamaño del arreglo se determina a partir de los datos de la malla, que se obtienen a partir del objeto mesh.

Líneas *18-22*: Definición de las condiciones iniciales. Se utilizan las bondades de la biblioteca Blitz++, para inicializar arreglos usando operadores sobrecargados.

Línea *24*: Se define el objeto energía. Se utilizan los parámetros: double para la precisión, 1 para la dimensión del problema y Diffusion<double, 1> para el esquema numérico. Este último parámetro determina la forma en que se calcularán los coeficientes de la ecuación discreta. Esta es una de las características importantes de este sistema: para cambiar de esquema numérico sólo se cambia este parámetro, pues la clase EnergyEquation<> es una clase polimórfica. En este caso el adaptador es Diffusion<double, 1>.

Líneas *25-30*: En estas líneas se define el sistema. Las condiciones de frontera se especifican en las líneas *29* y *30*, que en este caso son dos condiciones tipo Dirichlet[3] en los extremos. Cuando sea necesario resolver más de una ecuación, estás pueden compartir el sistema lineal (A y b) para ahorrar espacio en memoria.

Líneas *32-38*: Se inicia un ciclo que finaliza hasta que el error sea menor que una tolerancia especificada o se haya alcanzado el número máximo de iteraciones. Dentro de este ciclo primero se calculan los coeficientes, en donde se utiliza el esquema definido en la línea *24*.

Después se resuelve el sistema usando el algoritmo TDMA[4] que está contenido en el módulo Solver, finalmente se calcula el error y se actualiza la solución. El objeto energía tiene una copia del campo T que contiene los datos más recientes de la solución. Por otro lado, T contiene la solución en el paso anterior. Por esta razón, es necesario hacer una actualización a través del objeto energía. La línea *37* indica que se almacene la solución en un archivo y se utiliza la función printToFile_GP( ) del módulo Output que recibe el campo a almacenar, el número de interacción, la cadena base para el nombre del archivo y el tamaño de la malla.

Lo que es notable del código anterior es que, siempre se seguirá el mismo formato para resolver otro tipo de problemas. Los cambios serán mínimos y básicamente serán en la dimensión del problema, el número de ecuaciones a resolver y/o el esquema numérico a utilizar.

Para este ejemplo, se utilizó una malla unidimensional de 50 puntos y un paso en el tiempo de 10^{-2}.

En el siguiente problema se resuelve la ecuación de Laplace en dos dimensiones. En dos dimensiones el sistema lineal de ecuaciones es pentadiagonal, además se deben definir la longitud y el número de nodos en las direcciones *x* y *y*, y se deben utilizar campos escalares bidimensionales:

---

[3] La discretización de las condiciones de frontera es importante dado que definen la solución que se obtiene en el interior del dominio de estudio. Para los problemas resueltos en el simulador, existen básicamente dos tipos de condiciones de frontera: Dirichlet y Neumann.

[4] Es un método interactivo que se basa en el algoritmo directo de Thomas o TDMA para matrices tridiagonales. Método directo para resolver de manera simple y eficiente sistemas tridiagonales en una dimensión; de manera iterativa, línea por línea, para resolver problemas en 2 y 3 dimensiones.

```
DiagonalMatrix<Penta>      A(num_vols_x, num_vols_y);
ScalarField2D              b(num_vols_x, num_vols_y);
StructuredMesh<double, 2> malla(long_x, num_nodos_x, long_y, num_nodos_y);
ScalarField2D              T(malla.getExtentVolumes());
```

En este caso el tamaño de la matriz es de (*num_vols_x* x *num_vols_y*)^2

La definición de la condición inicial es muy clara dado que se está usando sobrecarga de operadores:

```
Range I(T.lbound(firstDim)+1,T.ubound(firstDim)-1);
int ej = T.ubound(firstDim);
T(I,ej) = 10 * sin (PI * dx * I);
```

La ecuación y el esquema numérico se parametrizan con dimensión 2:

```
EnergyEquation<double, 2, Diffusion<double, 2> > energia(T);
```

Las condiciones de frontera se definen en las cuatro paredes del cuadrado unitario:

```
energia.setDirichlet(TOP_WALL);
energia.setDirichlet(LEFT_WALL, 0.0);
energia.setDirichlet(RIGHT_WALL, 0.0);
energia.setDirichlet(BOTTOM_WALL, 0.0);
```

En la primera de estas condiciones no se utiliza ningún valor de frontera, por lo que la función *setDirichlet*() utiliza los valores de la condición inicial para definir la condición de frontera en la pared superior.

La ecuación se resuelve usando un algoritmo que es una extensión del TDMA a dos dimensiones:

```
Solver::solveByLines(energia, tolerancia, tdma_iter);
```

La función *solveByLines*() recibe como argumento la ecuación a resolver, la tolerancia y el número máximo de iteraciones.

2.1.3.2. Convección forzada en 2D y 3D

En los siguientes ejemplos resuelve lo mostrado en la siguiente figura 2.1.3.2.1:

Para resolver este problema necesitamos considerar lo siguiente:

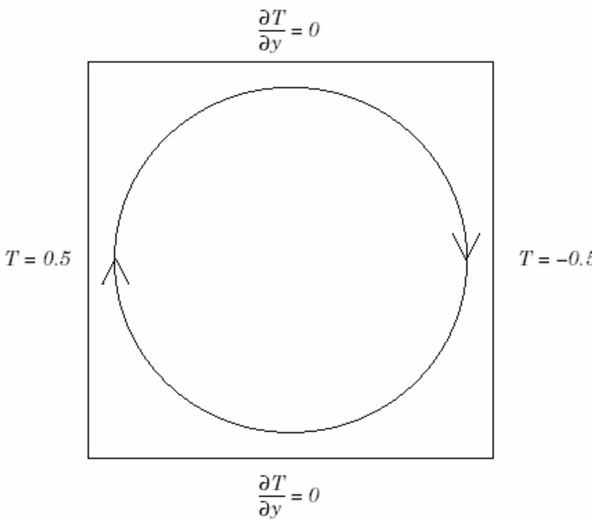


Figura *2.1.3.2.1.*Dominio de estudio y condiciones de frontera para el problema de convección forzada en 2D.

La ecuación resultante contiene un término convectivo, el cual debe tratarse con alguno de los esquemas numéricos. Las clases donde se implementan estos esquemas son adaptadores de las ecuaciones discretas y para usarlos se debe incluir alguno de los siguientes encabezados:

`#include "NumSchemes/QuickE.h"`, equema QUICK.

`#include "NumSchemes/UpwindE.h"`, esquema Upwind.

`#include "NumSchemes/CentralDiffE.h"`, esquema CDS.

Las declaraciones de la matriz del sistema, la malla y los campos escalares se hacen como sigue:

```
DiagonalMatrix<Penta>      A(num_vols_x, num_vols_y);
ScalarField2D              b(num_vols_x, num_vols_y);
StructuredMesh<double, 2> malla(longitud_x, num_nodos_x,
                                 longitud_y, num_nodos_y);

ScalarField2D T(malla.getExtentVolumes());    // Temperatura
ScalarField2D u(malla.getExtentVolumes());    // u-velocidad
ScalarField2D v(malla.getExtentVolumes());    // v-velocidad
ScalarField2D us(num_nodos_x, num_vols_y); // u-velocidad (staggered)
ScalarField2D vs(num_vols_x, num_nodos_y); // v-velocidad (staggered)
```

Donde los campos escalares us y vs se utilizan para almacenar las velocidades en mallas desplazadas. Nótese que estos campos escalares se definen de forma diferente a los casos anteriores: en la dirección del desplazamiento las variables están en las caras, que corresponden a los nodos, mientras que la otra dirección las variables están al centro de los volúmenes.

Para definir las condiciones iniciales hacemos uso de las ventajas de la biblioteca Blitz++, y las calculamos de la siguiente forma:

```
const int bi = T.lbound(firstDim) + 1, ei = T.ubound(firstDim) - 1,
          bj = T.lbound(secondDim) + 1, ej = T.ubound(secondDim) - 1;
Range I(bi,ei), J(bj,ej), all = Range::all();
double dx = malla.getDelta(0);
double dy = malla.getDelta(1);

T(bi-1, all) = left_wall;
T(ei-1, all) = right_wall;
u(I,J) = -A * cos (PI * dy * J) * sin (PI * dx * I);
v(I,J) =  A * sin (PI * dy * J) * cos (PI * dx * I);
```

Donde se utilizan rangos, I y J, para inicializar de manera simple y clara los componentes de la velocidad.

Durante el cálculo, las velocidades estarán desplazadas, por lo tanto, los valores almacenados en los campos escalares u y v se deben interpolar. Esto lo realizamos usando las funciones *staggerX*() y *staggerY*() del módulo *NumUtils*. En el código se escribe:

```
us = NumUtils::staggerX(u);
vs = NumUtils::staggerY(v);
```

La declaración y definición de la ecuación a resolver se realiza como sigue:

```
EnergyEquation<double, 2, QuickE<double,2> > energia(T);
energia.setLinearSystem(A,b);
energia.setGamma(1.0);
energia.setDeltas(malla.getDeltas());
energia.setDeltaTime(dt);
energia.setNeumann(TOP_WALL, 0.0);
energia.setNeumann(BOTTOM_WALL, 0.0);
energia.setDirichlet(LEFT_WALL, left_wall);
energia.setDirichlet(RIGHT_WALL, right_wall);
energia.setUvelocity(us);
energia.setVvelocity(vs);
```

En donde se han definido el sistema lineal, el coeficiente de difusión, el tamaño de la malla, el paso en el tiempo, dos condiciones tipo Neumann en las paredes superior (TOP) e inferior (BOTTOM) y dos condiciones tipo Dirichlet en las paredes derecha (RIGHT) e izquierda (LEFT). Además se definen las velocidades que se usarán, y que en este caso serán las desplazadas.

El ciclo para resolver el problema es el siguiente:

```
while ( (error > tolerancia) && (iteracion <= nmax) ) {
    energia.calcCoefficients();
    Solver::solveByLines(energia, tolerancia, max_iter);
    error = energia.calcError();
    residuo = energia.calcResidual();
    energia.update();
    //... actualizaciones, etc.
}
```

El problema en tres dimensiones se resuelve de forma idéntica, sólo es necesario hacer todas las declaraciones en 3D (sistema lineal, malla, campos escalares, acceso a elementos).

2.2. ANÁLISIS DE INTERFACES GRÁFICAS DE OTROS SIMULADORES

El análisis de las interfaces es con la finalidad de determinar el valor o la calidad que se desea conseguir mediante un método de evaluación, con el apoyo de estos se consiguen datos relevantes sobre la operatibilidad y usabilidad de un sistema, además de ser de gran utilidad para mejorar el diseño; se toman las características más sobresalientes de calidad, se aprende de los errores, para con ello tener una interfaz basada en la experiencia, y al final la convicción de que se obtendrá una buena interfaz.

Partiendo de que una buena interfaz debe ser transparente para quien utiliza la aplicación la representación del conocimiento y de la información, se realiza a manera que el mundo del usuario sea representado dentro del computador. Brenda Laurel[5] considera esencial que cada sistema se base en metáforas del mundo real, para construir

[5] Laurel B., *Introduction en The Art of Human-Computer Interface Design*. Addison-Wesley Publishing Company, 1990.

escenas familiares e intuitivas, que faciliten la interacción y la comunicación usuario-sistema.

El impacto de las interfaces gráficas justifica la atención que debe ser dada a su elaboración y al usuario. Debora Hix[6] dice que: “Una buena interfaz es como el teléfono o como la luz eléctrica; cuando funciona nadie la nota”. Una buena interfaz parece algo obvio, pero lo que no parece ser tan obvio es como desarrollar una interfaz que tenga un alto grado de usabilidad[7].

Cuando se habla de Usabilidad se enfoca en cuestiones relacionadas a que tan bien los usuarios pueden utilizar las funcionalidades de la aplicación y como la aplicación le ayuda a cumplir sus tareas específicas. Nielsen[8] relaciona el concepto de Usabilidad a cinco atributos que pueden ser medidos, ellos son:

- Facilidad de aprendizaje
- Eficiencia de uso
- Facilidad de memorización
- Baja tasa de errores
- Satisfacción subjetiva.

Dependiendo del proyecto y de los usuarios típicos, estas características tienen importancia variable, y es el desarrollador quien les da la prioridad a estos aspectos en base al dominio del sistema que se esta desarrollando. Pero lo que no debe olvidar es que la *Usabilidad* es el parámetro central para desarrollar buenas aplicaciones y para verificar la calidad de la interfaz. ¿Cómo saber si se desarrollo una buena interfaz?- por medio de un análisis como el de Nielsen-, podemos responder a esta pregunta; se reduce a principios que pueden ser utilizados para explicar gran parte de los problemas que se observan en una interfaz.

Los principios utilizados por Nielsen son:

- Relación entre el sistema y el mundo real
- El usuario controlando la interacción
- Simplicidad y estética de la aplicación
- Flexibilidad y eficiencia en el uso de la aplicación
- Minimización de la carga cognitiva del usuario
- Consistencia de la aplicación
- Información sobre el estado del sistema
- Buenos mensajes de error
- Prevención de errores

---

[6] Hix D. y Hartson H., Developing User Interfaces: Ensuring Usability Through Product & Process. John Wiley & Sons, Inc., 1993.

[7] Entre todas las denominaciones aparentemente posibles: “amigabilidad”, “operatibilidad”, “ergonomía” o “manejabilidad”, entre otras, se ha elegido hablar en términos de “usabilidad” por una razón principal: “usabilidad” es la traducción literal de “usability”, concepto que, según Booth-Booth, Paul; *An introduction to Human-Computer Interaction. London:* Lawrence Erlbaum Associates, 1989; p. 104-, es ampliamente aceptado por la comunidad de investigadores y por los medios de comunicación.

[8] Nielsen J., *Usability Engineering*. AP. Profesional, 1993.

Es importante comentar que la experiencia demuestra que solo una pequeña parte de los problemas de usabilidad de una interfaz pueden ser encontrados de está forma y que además es importante el número de personas que realizan esté análisis.

Esté y otros análisis siguen un ciclo de diseño[9] que ayudan a mejorar el desarrollo de la interfaz casi asegurando que a final se obtendrá una interfaz con la mayor usabilidad y satisfacción; ya que el refinamiento sucesivos, a través de la evaluación del interfaz resultante y su modificación a raíz de los comentarios sobre el mismo es lo que te lleva a la mejora; como muestra la figura 2.2.1.

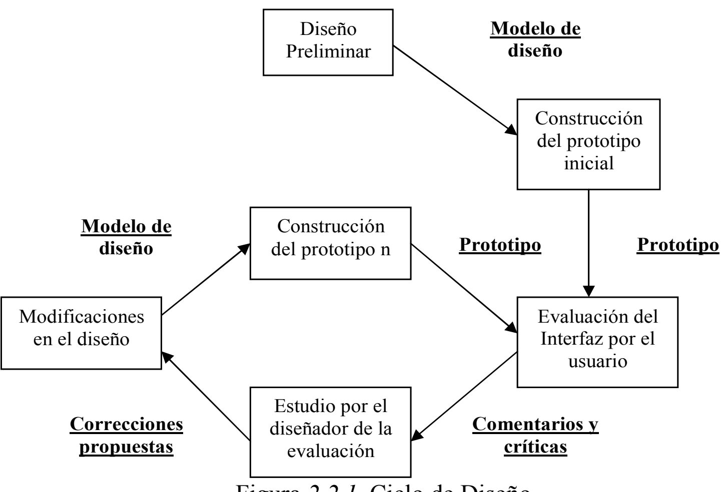


Figura *2.2.1.* Ciclo de Diseño.

Recuadros y agrupación espacial

A partir de la identificación de estos problemas se obtiene -en nuestro caso- un análisis de las interfaces de otros simuladores dando como resultado subsidios para el proceso de diseño de nuestra interfaz.

Otra parte importante para el análisis es tomar en cuenta conceptos como:

La "***eficiencia***", la forma más habitual de medirla es preguntarle directamente al usuario. Algunas preguntas posibles son las siguientes:

- ¿Se pregunta usted a menudo si utiliza el comando correcto?
- ¿Es satisfactorio trabajar con el sistema?
- ¿Responde rápidamente el sistema a sus acciones?
- ¿Responde adecuadamente el sistema a sus acciones?
- ¿Utiliza los "atajos de teclado" para llevar a cabo alguna acción?
- ¿Entiende la información que le proporciona la interfaz?
- ¿Le es útil la información que le proporciona la interfaz?

Una respuesta positiva a las preguntas planteadas supone que la interfaz es eficiente.

[9] Existen ciclos de diseño de interfaces: ciclo de vida en cascada, en espiral, entre otros, Apéndice C.

La "***consistencia***" es un factor importante para considerar que una interfaz es buena o no; según el *Usability First Glossary*[10], consistencia es el principio por el cual los elementos relacionados deben ser presentados de forma idéntica e inequívocamente. Es un concepto aplicable a lo siguiente:

- La *tipografía* utilizada en la presentación de la información -por ejemplo: si en una pantalla del sistema, la letra es destacada en negrita, no puede enfatizarse la tipografía en otra pantalla utilizando la cursiva.
- Los *iconos*, *comandos* y *menús*, que deben ser consistentes con lo que representan: el usuario debe esperar una acción del sistema de acuerdo con su elección[11].
- La aplicación, debe funcionar igual en cualquier contexto y/o plataforma[12].
- La *percepción* del sistema, debe ser igual por todos los usuarios.
- La *estructura* del sistema, mediante sus metáforas, debe representar adecuadamente el universo de trabajo del usuario.

Las interfaces inconsistentes son aquellas en las que los usuarios deben emplear diferentes métodos en contextos diversos para llevar a cabo las mismas tareas. Para evaluar esta parte podríamos realizar las siguientes preguntas:

- ¿Se presenta la información de forma clara e inequívoca?
- ¿Cree usted que la interfaz es inconsistente?
- ¿La semántica de los menús es clara?
- ¿Existe alguna opción que no sabe lo que significa?
- ¿Conoce usted la diferencia entre las opciones "aceptar" y "aplicar"?
- ¿Se ha encontrado usted con algún icono que no representaba la acción que esperaba?

La "***simplicidad***" se basa en el entendimiento del problema que se quiere transmitir, mostrando capacidad de hacerlo de una forma clara y concisa consiguiendo comunicación. Los buenos diseños son los que están ahí, pero no se ven y por ello son los más fáciles de usar.

La facilidad o simplicidad de uso está en relación directa con la eficiencia o efectividad, en el diseño depende de tres principios estrechamente relacionados: unidad, refinamiento y conveniencia.

Los beneficios de la simplicidad son:

- Accesibilidad
- Reconocimiento
- Rapidez de percepción
- Uso continuo

[10] *Usability First Glossary,* disponible en http://www.usabilityfirst.com/glossary.

[11] Tognazzini, en: Tognazzini, Bruce; "Consistency". En: Laurel, Brenda (ed.); *The art of Human-Computer Interface Design*. Reading, Massachusetts: Addison-Wesley, 1991; pp. 75-77, presenta esta idea como el concepto "inferencia de la consistencia".

[12] Shneiderman, en: Shneiderman, Ben; *Op. Cit.*; llama "recursividad de sistemas" a la compatibilidad entre diferentes versiones de un producto, la consistencia no debe ir en contra de la creatividad, sólo debe "controlarla".

Esta depende de tres principios estrechamente relacionados:

- Unidad: La meta debe ser integrar de manera armónica cada elemento de la interfaz como si fuese uno sólo.
- Refinamiento: El refinamiento sucesivo es el único camino hacia la simplicidad. Para crear una solución elegante, todo aquello que no sea esencial para la comunicación debe ser eliminado.
- Conveniencia: Las soluciones propuestas deben ser las más convenientes.

El "***contraste***" se consigue colocando elementos que no son lo mismo de manera diferente proporcionando una distinción visual ya que se realiza un énfasis en la actividad de interés, no hacerlo lleva a la confusión. Si se realiza un contraste tendremos como resultado una interfaz amena y conseguiremos que el usuario se desplace de un elemento a otro acertadamente.

Los beneficios del uso de contrastes y proporcionalidad:

- Diferenciación
- Énfasis
- Actividad
- Interés

La "***eficacia***" se logra pensando en el usuario, buscando la productividad de nuestro usuario y no del ordenador.

La "***predicción***" se logra utilizando diálogos simples y naturales, agrupar los datos lógicamente y jerarquizando la información, mostrar sólo la necesaria.

La "***retroalimentación***" es comunicar el estado, esto para el usuario es fundamental ya que con ello sabrá si se le responde apropiadamente en base a la información disponible.

La "***composición***" es la forma de ordenar y organizar los elementos en el espacio que se ofrece, en base a la unidad y claridad, hay que buscar delimitar claramente el centro de interés; se puede crear diversidad y contraste para añadir dinamismo, aunque esto ultimo complica la composición.

El "***color***" es un aspecto muy importante, el uso de colores apropiados puede ayudar a la memoria del usuario y facilitar la formación de modelos mentales efectivos, pero el uso inapropiado de los colores puede causar distracción al usuario y forzar su vista.

Algunas de las maneras en que se utiliza son: para agrupar visualmente elementos con funciones o contenidos relacionados, para separar visualmente zonas de la interfaz, para atraer la atención del usuario, para establecer jerarquías entre los elementos de la interfaz, para codificar categorías o tipologías, para codificar procesos o secuencias, para identificar el tipo de información (su origen y/o estado), etc.

Los "***controles***" son las herramientas se colocan en la pantalla y permite dar importancia a ciertas cosas, su orden de lectura -la posición- es importante ya que ayuda a enfatizar, el uso del color y los atributos del texto en dichos controles ayudan mucho a predecir y utilizar de manera adecuada cada control.

Lo importante de todo es observar si los usuarios están satisfechos y les es útil la interfaz, el que estén satisfechos permite dar un critica justificable acerca de las Interfaces de Usuario existentes; la satisfacción del usuario podría definirse como la "capacidad de la interfaz de interactuar con el usuario de forma agradable"[13]. Esta "forma agradable" será evaluada en función de la actitud del usuario frente a la interfaz[14].

La medición de la satisfacción del usuario puede realizarse de muchas formas. Bien puede hacerse preguntando directamente al usuario si recomendaría la interfaz o si trabajar con ella le ha resultado satisfactorio; se puede elaborar una pequeña encuesta con enunciados en forma de escala de Likert[15], o bien, podría utilizarse lo que muestra la Figura 2.2.2., ejemplo :

- El uso de la interfaz fue sencillo de aprender:
  - DE ACUERDO |—| NEUTRAL |—| EN DESACUERDO
- Interaccionar con la interfaz fue una experiencia frustrante:
  - DE ACUERDO |—| NEUTRAL |—| EN DESACUERDO
- Creo que la interfaz me ayuda a ser más productivo en mi trabajo:
  - DE ACUERDO |—| NEUTRAL |—| EN DESACUERDO
- Creo que la interfaz dispone de todas las potencialidades que necesito:
  - DE ACUERDO |—| NEUTRAL |—| EN DESACUERDO

Califique la interfaz

	Muy	Algo	Neutral	Algo	Muy	
Agradable						Irritante
Completa						Incompleta
Simple						Confusa
Segura						Insegura

Figura *2.2.2.* Escalas semánticas diferenciales.

Con estos puntos se evaluaron los simuladores existentes de flujos con la finalidad de subsanar los defectos al momento de diseñar la interfaz, y se obtuvieron como resultados lo siguiente -consultar Apéndice C-:

FLUENT[16]- Flow Modeling Software

Evaluando los principios generales del diseño de GUI, se encontraron dificultades en el reconocimiento de la funcionalidad de algunos controles, por lo que no resulta muy sencillo o simple utilizar la interfaz; la consistencia en cuanto el tamaño

[13] Nielsen, Jakob; *Usability Engineering*. New York: Academic Press, 1993; p. 33.

[14] Cabe decir que la actitud irá en relación directa con la sensación de control que tenga el usuario sobre la interfaz.

[15] Nielsen, Jakob; *Ibid.*; p. 35.

[16] Para mayor información acerca de este software visita: http://www.fluent.com/software/fluent/index.htm

y distribución de la información no es muy clara, el uso de varias ventanas para hacer contraste con el uso de menús es muy elevada y con ello la poca facilidad de identificación de elementos requeridos, pero sin embargo esto contribuye a que el usuario tenga el control en cuanto a la visualización o no, de ciertos menús; al respecto podemos comentar que esto podría afectar en la predicción de su uso; en cuanto su retroalimentación es muy buena, visualmente se observan cambios tanto en los controles como en lo que se esta realizando, lo cual es bueno para el usuario, ya que sabrá que es exactamente el cambio que propicia cierto control.

La composición no cuanta con un buen análisis o valoración del espacio y ubicación de los elementos; el uso del color es poco utilizado en cuestiones de menús pero si adecuado en la parte de visualización de resultados cumpliendo así con la función de informar. En cuanto su eficiencia se cumple, ya que se obtiene lo que se desea procurando utilizar las herramientas que proporciona el software.

Finalmente la interfaz estuvo entre sencilla y complicada, se obtuvo la información que se esperaba, y aunque en términos generales no es muy agradable o atractiva, se consigue con su uso las soluciones deseadas.

Algunos puntos claves para mejorar esta interfaz podrían ser los siguientes: el uso de metáforas, utilizaría iconos, menos ventanas buscando no saturar la pantalla, y por ende el área de trabajo; buscaría la solución en menús pop-up, en el área de botones es recomendable dar contraste junto con presentación, un buen alineado y jerarquización de éstos, con información mas explicita para el usuario, y en general un ambiente de trabajo menos saturado, mas ordenado, un uso de colores que le dieran un toque mas formal como representativo, y por lo tanto atractivo a la vista de nuestro usuario.

FIDAP[17]**-** Advanced Materials Physics

Se encontró que no resulta sencillo identificar la usabilidad de ciertos controles, además de ser poco semejante al mundo real, aunque resulta ordenada; es consistente en la forma de presentar la información, el manejo de la pantalla es adecuado aunque no agradable; existe un uso de contraste pero quizá no el mejor al igual que con el uso del color, estos no ayudan mucho en la predicción en cuanto el uso o funcionalidad de ciertos controles; sin embargo existe una retroalimentación por parte de la interfaz haciendo uso de colores y representación con visualización de imagen en una sección de la pantalla, con una buena composición de los elementos, la manera de estar localizados en pantalla; a niveles generales el uso de esta interfaz es utilizada más por su eficacia[18] que por su eficiencia[19].

Considero que el color utilizado no es muy atractivo, inspira a mi punto de ver, antigüedad, como si fuese una interfaz de hace ya algunos años, es decir, no moderna, el uso de iconos y con ello metáforas del mundo real, no solo le dan mejor entendimiento al usuario, logrando así identificar con facilidad la usabilidad de ciertos controles, si no

[17]Para mayor información acerca de este software visita: www.fluent.com/software/fidap/ o www.cavendishcfd.com/fd-info1.htm

[18] Virtud, activo, que produce efecto.

[19] Virtud para lograr algo, produce realmente un efecto, competente.

también le mejor visión, logrando así atraer al usuario y además da una buena presentación.

La jerarquización o contraste en los controles no esta muy definida, tiene puntos buenos como son su sencillez, ya que no satura de controles, solo tiene lo necesario para solucionar los problemas, quizá el enfoque o la solución fue que resolviera los problema, no como se vería el aspecto físico de la interfaz.

RealFlow[20]

Resulta muy ordenada e interactiva, la utilización de iconos es buena, pero no son los adecuados, es decir, no muy representativos o fáciles de identificar, no dan mucha información para identificarlos provocando utilizar mas memoria de nuestro usuario, por otro lado, si cuenta con diferentes modos de interacción, los formularios son fáciles de comprender y junto con todo la interfaz es constante, cuanta con contrastes, logra una buena predicción -aunque poco en el manejo de iconos- pero en cambio cuanta con buena retroalimentación y eficacia; quizá en el manejo de colores no es bueno en la representación general de la interfaz pero si en la visualización de resultados. La composición y distribución de los elementos es muy adecuada, agradable y muy familiar para los que han trabajado con software similar, y no tan desagradable y complicado para los que no.

Al final cumple con su funcionalidad, es atractiva, eficaz y eficiente, por lo que simplemente le mejoraría la utilización de colores y un mejor uso de metáforas en uso de iconos.

PHOENICS[21] -Parabolic Hyperbolic Or Elliptic Numerical Integration Code Series

Como resultado de su valoración se puede percibir que cuanta con pocos controles por lo que la hace sencilla, pero el uso saturado de iconos es para la vista poco atractivo y confuso, provocando que no haya predicción del uso de los elementos, aunque existiera una buena retroalimentación, llevando a errores constantes y la utilización un poco excesiva de la mente del usuario.

El uso del contraste existe pero no con gran éxito; el uso del color resulta inadecuado, además de en ocasiones excesivo, difícil para la vista y no general, para todos los elementos haciendo a la interfaz no constante. Considero que no cuenta con el potencial de una interfaz moderna, atractiva, sobresaliente, aunque sea eficaz.

[20] Para mayor información acerca de este software visita: www.nextlimit.com/, o visita www.4bytes.com/store/3dshopware/moreinfo/realflow.htm

[21] Para mayor información acerca de este software visita: http://www.cham.co.uk

CAPÍTULO 3

DISEÑO

Capítulo 3. DISEÑO

3.1. PAPEL DEL DISEÑO

Etimológicamente la palabra Diseño tiene varias acepciones del término *design* -referente al signo, señalar, señal (indicación gráfica de sentido o dirección) representada mediante cualquier medio y sobre cualquier soporte analógico, digital, virtual en dos o más dimensiones-.

En el diccionario se define como el bosquejo o dibujo de alguna cosa y también como la actividad creativa consistente en determinar las propiedades formales o características exteriores de los objetos que se van a producir. Aunque en la informática no se crea un objeto sólido y palpable en sí, el diseño es una parte esencial del producto final, que permite tener una referencia (generalmente gráfica) de la forma que tendrá el producto al ser terminado.

Diseño como verbo -"diseñar"- se refiere al proceso de creación y desarrollo para producir un nuevo objeto o medio de comunicación (máquina, producto, edificio, grafismo, etc.) para uso humano. Como sustantivo, el diseño se refiere al plan final o proposición determinada fruto del proceso de diseñar (dibujo, proyecto, maqueta, plano, o descripción técnica), o (más popularmente) al resultado de poner ese plan final en práctica (la imagen o el objeto producido).

El proceso de diseñar, suele implicar las siguientes fases:

Observar y analizar: el medio en el cual se desenvuelve el ser humano, descubriendo alguna necesidad.

Planear y proyectar: proponiendo un modo de solucionar esta necesidad, por medio de planos y maquetas, tratando de descubrir la posibilidad y viabilidad de la(s) solución(es).

Construir y ejecutar: llevando a la vida real la idea inicial, por medio de materiales y procesos productivos.

Hoy por hoy, y debido al mejoramiento del trabajo del diseñador (gracias a mejores procesos de producción y recursos informáticos), podemos destacar otra fase fundamental en el proceso:

Evaluar: ya que es necesario saber cuando el diseño está finalizado.

Diseñar como acto cultural implica conocer criterios de diseño como: presentación, producción, significación, socialización, costos, mercadeo, entre otros. Estos criterios son innumerables, pero son contables a medida que el encargo aparece y se define.

El diseño de interfaces es una disciplina que estudia y trata de poner en práctica procesos orientados a construir la interfaz más usable posible. La interfaz deberá anticiparse a las necesidades del usuario.

El entorno dentro del cual se inscribe el diseño de una interfaz y la medida de su usabilidad, está dado por tres factores: una persona, una tarea, un contexto, figura 3.1.1:

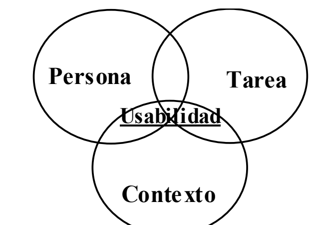


Figura *3.1.1*. Factores de usabilidad.

La utilidad es la capacidad que tiene una herramienta para ayudar a cumplir tareas específicas; aunque esta afirmación parece obvia, es importante observar que una herramienta que es muy usable para una tarea, puede ser muy poco usable para otra, aún incluso si se trata de una tarea similar pero no idéntica. La facilidad de uso está en relación directa con la eficiencia o efectividad, medida como velocidad o cantidad de posibles errores. La facilidad de aprendizaje es una medida del tiempo requerido para trabajar con cierto grado de retención de estos conocimientos luego de cierto tiempo de no usar la herramienta o sistema.

El diseño de interfaces pertenece a un campo mayor del conocimiento humano, de origen altamente interdisciplinario, llamado Interacción Hombre-Computador (Human Computer Interaction[1]); la comunicación entre el usuario y la computadora se realiza a través de una interfaz -figura 3.1.2-, la cual es responsable por el buen éxito de la interacción y por la fuerza del dialogo. Paul Heckel sugiere que, para proyectar buenas interfaces con usuarios, se debe pensar mas en comunicación que en computación; y es precisamente lo que se trata de lograr en el diseño de está interfaz; construir una interfaz usable, así como unificar armónicamente los elementos como si fuesen uno solo.

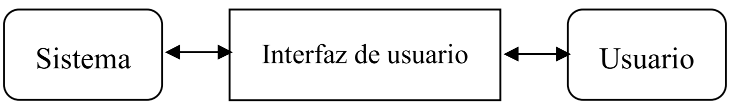


Figura *3.1.2* Interacción Hombre-Computador.

En el diseño se busca que los elementos estén unificados para producir un todo coherente, las partes (al igual que todo) fueran refinadas para captar la atención del usuario sobre sus aspectos esenciales, y en cada nivel nos aseguramos que la solución fuese la más conveniente.

[1] Human.Computer Interaction (HCI), área multidisciplinaria que estudia el Diseño, Evaluación e Implementación de sistemas interactivos, abordando toda la problemática en torno a ellos.

El crear una interfaz, no solo lo hace un programador de sistemas, también una interfaz es creada al idear cualquier tipo de objeto usado por el ser humano, ayudado por la ergonomía, psicología, y diseño, con la finalidad de que el objeto ideado sea más fácil de usar, y más rápido de aprender.

Se hizo uso del diseño gráfico "Holistic", el cual juega con el modelo conceptual del sistema a desarrollar, comienza con una imagen informal de cómo el sistema podría aparecer finalmente ante el usuario; esta metodología aporta una versión creativa e idealista como primera estancia al diseño, ya que se parte de un problema, en el que se tienen ya como mínimo, el conjunto de tareas que se han de llevar a cabo en el sistema.

Es así como se emprende la realización del modelo con papel y lápiz, antes de haber definido nada del software o hardware a utilizar, todo esto es un proceso iterativo, que en sus inicio incluso se excluye de comprender elementos como iconos, menús, etc., conforme se va avanzado se van considerando hasta obtener el primer prototipo real; a medida que se obtienen modelos y que se llevan acabo, es como se incorporan elementos que se acopla perfectamente a los mecanismos que se desean conseguir, otros quizá, de los elementos abstractos que se tienen se crearán elementos nuevos o incluso podrían desaparecer.

Existen planteamientos que se utilizan para descubrir partes de la interfase que podrían ser fuertes o débiles; una técnica llamada "cognitive walkthrough" es utilizada precisamente para distinguir partes del diseño donde los usuarios podrían más frecuentemente cometer errores; este método analiza las interacciones de los usuarios con la interfase y su rendimiento en tareas específicas.

Esto se realizó construyendo un prototipo que se pudiera mostrar a los usuarios y que fuera de manera muy detallada, para obtener la información que requeríamos[2], así como también el uso del ciclo de vida de cascada (figura 3.1.2) –Apéndice D-, ya que posibilitó el refinamiento sucesivo y correcciones de problemas que iban surgiendo. El refinamiento sucesivo es el camino hacia la simplicidad, por que eliminas todo aquello que no sea esencial para la comunicación.

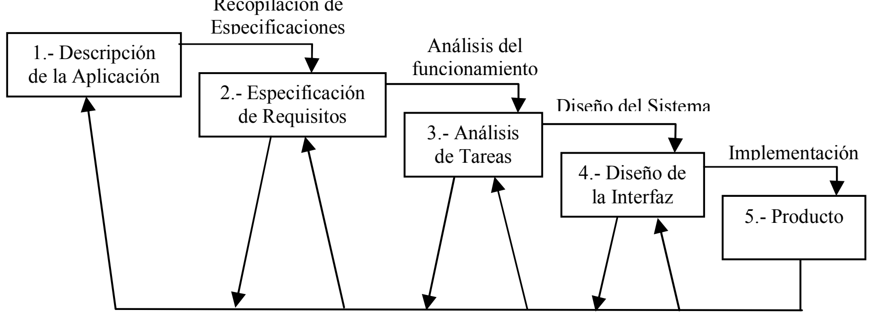


Figura *3.1.2*. Ciclo de vida de cascada.

[2] Existen herramientas para realizar prototipos como HyperCard, Dan Bricklin´s, Demo Package, etc. No requieres de realizar el diseño completo, los esfuerzos iniciales deben ser concentrados en las partes de la interfaz que representan tareas principales.

Pero quizá te preguntaras ¿Cuándo debes parar?, si te haz definido objetivos de usabilidad, entonces deberás parar cuando los hayas conseguido.

Otras técnicas que ayudan a obtener una solución para reducir el diseño con cualidades esenciales, elementos formales como etiquetas, controles, colores, texturas, patrones o imágenes, es examinando de manera crítica cada elemento en el diseño y cuestionarse: ¿por qué es necesario?, ¿cómo está relacionado a la esencia del diseño? y ¿qué tanto le hace falta al diseño si no estuviera?, si no existe una respuesta para alguna de estas preguntas, el elemento debe ser eliminado, pero si al hacerlo, el diseño se colapsa de manera funcional o estéticamente, entonces debe permanecer.

También es importante verificar que los elementos en una misma región se comporten como una unidad, pero que el grupo mismo pueda ser separado visualmente del resto de la interfase, así como las rutas que el ojo necesita seguir en el diseño.

Antes que nada se definieron los objetivos principales como son que al final el usuario tuviera facilidad de aprendizaje con la fácil identificación de los elementos, así como la organización de la pantalla, la información y una buena interacción, todo esto para una usabilidad de nuestra interfaz; no se olvidaron parámetros que se analizaron en las interfaces existentes como el uso del color donde se procuro obtener excelentes combinaciones como usos del mismo, sin abusar de este recurso de comunicación, en el uso de la tipografía se pensó en no mezclar en pantalla más de dos tipos y tres medidas diferentes de letra, entre otros.

El uso de nemotécnicos para subrayar palabras (Alt + Letra), se utilizó la primera letra de la palabra, si entraba en conflicto se hizo uso de una consonante prominente (T, X, Z, etc.), si aún esta entraba en conflicto se hizo uso de una vocal prominente (estas fueron recomendaciones de diseño). Uso de atajos (Shortcuts) que fueran consistentes, manejo de errores y visualización de los mismos, buscar retroalimentar a nuestro usuario mostrándole lo que ocurre cuando realiza determinada acción, manejo de ventanas y campos buscando que fueran representativos y que mostraran su relación entre ellos, se busco mostrar información estrictamente necesaria para cada vista de la interfaz así como mostrar los objetos homólogos con la misma representación.

Los beneficios de las acciones y recomendaciones también de lo mencionado anteriormente son que obtienes una Interfaz Gráfica con contrastes, proporcionalidad y con ello tu usuario tendrá una buena diferenciación detectando el énfasis, la actividad y su interés en cierto proceso.

El uso de imágenes es muy importante en tres áreas: te sirven para identificar objetos concretos, en la expresión ya que tienen un poder expresivo muy grande y en la comunicación por que no tienen límites, no se debe olvidar que es preciso buscar reducir la cantidad de información que debe memorizar entre acciones nuestro usuario, así como la eficiencia en el diálogo, perdonar los errores y categorización de los procesos ha realizar.

3.2. USO DE METÁFORAS VISUALES

3.2.1 La necesidad de interfaces más intuitivas

Una de las cosas que resultan obvias cuando uno hace un aprueba de usabilidad -ya sea que esté uno probando sitios web, software o electrodomésticos- es la gran cantidad de gente que usa las cosas sin saber en realidad como funcionan o con ideas completamente erróneas acera de su funcionamiento.

De cara a cualquier tipo de tecnología, muy poca gente se toma el tiempo de leer instrucciones. En su lugar, avanzamos experimentando, haciendo historias, vagamente plausibles acerca de qué estamos haciendo y por qué funciona.

Para el usuario *no es relevante saber como funcionan las cosas*, en tanto que pueda usarla. No es un problema de inteligencia sino de falta de interés. Si logran hacer que algo funcione, no importa que tan mal lo haga, no buscarán otra manera de hacerlo. Si dieran con una mejor opción, seguro la usarían pero no la buscaran por ellos mismos.

La pregunta es, ¿Si de cualquier forma los usuarios *intuyen* como usar las cosas, que más da si entienden o no? y la respuesta es que importa y mucho. Porque mientras puede estar cada día adivinando como usar una interfaz, este uso tiende a ser ineficiente y propenso a errores además de que no se logrará extraer todo el potencial de la aplicación.

Si los usuarios entienden, hay entonces una mejor oportunidad de que encuentren lo que buscan y cumplan sus objetivos, lo que es bueno tanto para el usuario como para el proveedor del servicio. Además, el usuario se sentirá más inteligente y en control cuando use nuestro sitio, lo que garantizará su regreso.

3.2.2 ¿Por qué usar metáforas visuales en Interfaces Gráficas de Usuario?

Para poder interactuar efectivamente con una aplicación es importante para los usuarios obtener un modelo cognitivo apropiado y relacionado a esa aplicación. Una manera simple de presentarle al usuario una función abstracta dentro de la funcionalidad de una aplicación es estableciendo una comparación entre esa funcionalidad y un objeto tangible del *mundo real* con el que esté familiarizado. Esto se logra mediante el uso de metáforas. Las metáforas contenidas en la interfase de usuario final de una aplicación son herramientas poderosas para el desarrollo de modelos cognitivos y conceptuales.

Muchas de los lenguajes de las culturas antiguas eran visuales o ideográficas al menos en parte. Tomemos por ejemplo los jeroglíficos egipcios, los pictogramas mayas y la escritura china son todos pro necesidad un tipo de atajos visuales. La clave de estos lenguajes era representar visualmente solo la cantidad necesaria para *sugerir* el objeto en cuestión. Lo que significa que mucha de la responsabilidad de completar los detalles faltantes reside en el lector, aunque este esfuerzo de descifrado se vuelve innecesario con el tiempo al acostumbrarse éste a saltar (mentalmente) de la imagen a la idea.

En este contexto, una metáfora es el uso de una *idea* u objeto en lugar de otro para sugerir una similitud[3]. Una metáfora visual es un ideograma que representa visualmente esa idea abstracta. La importancia de las metáforas reside en su habilidad para iniciar una transferencia cognitiva entre un dominio del conocimiento familiar hacia otro menos familiar.

3.2.3 Lineamientos para el diseño de metáforas visuales efectivas

3.2.3.1 Errores comunes en uso de metáforas visuales

Si los pictogramas o las metáforas requieren, para ser comprendidas la inclusión de un texto, que generalmente es un palabra o frase perfectamente monosémico e inequívoca, hay que pensar si no es preferible utilizar esta palabra en lugar de una metáfora ambigua y dotarle mejor de condiciones optimas de legibilidad.

Otro error común es la falta de consistencia en el diseño de las metáforas a través de todo el sistema icónico, puesto que se toman metáforas existentes de diversa fuentes que no siguen una misma línea estética en su diseño. Esto genera confusión y falta de identidad.

3.2.3.2 Recomendaciones para diseñar sistemas de metáforas para una interfaz

Cuando existan metáforas estandarizadas, úselas. Si bien, no existen manuales internacionales con estándares establecidos, algunos estándares están surgiendo. Es bueno estar informado de como avanza la comunidad de diseño en éste sentido.

Cuando una metáfora es usada en varias aplicaciones, impleméntelas en la manera estándar. Es bueno tener conjuntos de íconos probados y examinados que se relacionen con determinadas metáforas particulares listos para ser adaptados y reutilizados.

Use una metáfora en toda su extensión. Una implementación incompleta puede llevarnos a confusión, frustración y el desarrollo de estructuras pobremente formadas. Esto es claro sobre todo al usar sistemas metafóricos complejos como en el caso del escritorio, un elemento perteneciente a otro campo semántico — como una sartén, por ejemplo— creará confusión en cuanto a su uso y le restará seriedad a la aplicación.

Evita metáforas similares cuando uses múltiples metáforas. Obviamente, metáforas similares están propensas a provocar confusión en situaciones donde se producen íconos que podrían ser igualmente aplicables a una u otra metáfora.

No use referencias alusivas a un lenguaje específico. Esto es especialmente importante al diseñar aplicaciones con alcance internacional como puede ser un sitio web. Por ejemplo, en inglés se utiliza la palabra home (casa) para designar a la página principal de un sitio, pero en otros idiomas se utiliza otra palabra para describirlas, en

[3] http://www.intellectbooks.com/iconic/metaphor/metaphor.htm

español por ejemplo, decimos portada, índice o simplemente página principal. Así que el uso de un icono que representa una casa será confuso para todo aquel que no domine el idioma inglés.

Cuando un usuario se enfrenta a una nueva aplicación, sea un programa de escritorio o un sitio web puede que no intuya inmediatamente cómo debe operarlo para hacerlo funcionar. Una manera de remediarlo es diseñar cuidadosamente un sistema de metáforas visuales que hagan más intuitiva la interfaz de la aplicación. Esta técnica ha sido utilizada con éxito durante mucho tiempo pero conforme avanza el alcance de la tecnología y las aplicaciones computacionales forman una parte cada vez mas importante de nuestra vida diaria es urgente que los diseñadores visuales aprendan a aplicar los fundamentos de la comunicación visual para diseñar interfaces que le den al usuario una experiencia más enriquecedora[4].

3.2.3.3 La importancia de los paradigmas

Existe el uso de paradigmas de desarrollo de sistemas, que desde luego tienen que ver con la parte visual, con nuestra interfaz de usuario, el paradigma Modelo Vista Controlador (View-Controller-Model[5]) -con el que continuaremos un poco más adelante-.

A la hora de construir aplicaciones, en general, es muy común partir de una simple lista de requerimientos y comenzar directamente a programar si detenerse previamente a pensar un poco en la estructura que tendrá la aplicación. Aún en casos en que se intenta aplicar un diseño, es habitual que los desarrolladores partan desde cero creando todo tipo de diseños extremadamente complejos que luego los programadote tienen dificultad para comprender y ejecutar.

Provee un diseño mundialmente reconocido de manera que será más fácil que un programador lo conozca y comprenda, aún cuando acabe de incorporarse al proyecto.

Permite tener una estructura de código común a todos lo proyecto que implemente la funcionalidad genérica.

Concretamente a lo que muchos piensan, permite ahorrar grandes cantidades de tiempo en la construcción de software. Para implementar una arquitectura adecuada, deberemos dedicar un poco más de tiempo al principio del primer proyecto pero este precio es despreciable si se comprara con el tiempo que nos ahorra en mantenimiento y en desarrollo de los siguiente proyectos.

El software construido es más fácil de comprender, mantener y extender. También facilita la estabilidad, esto es, la adaptación del software a una mayor carga de trabajo con el paso del tiempo.

[4] http://nolimit-studio.com/tesis/
[5] Esté paradigma es el mas utilizado en el diseño de GUIs.

3.2.3.3.1 Modelo-Vista-Controlador (View-Controller-Model)

Este patrón de diseño se basa en separar la lógica de negocio, la presentación al usuario y el control del flujo de la aplicación de manera que variaciones en uno de estos componentes afecten en la menor medida posible al resto. Esté se conciben tres capas de implementación:

Modelo

- Destinada a resolver el problema (cálculos).
- Encapsula el estado de los datos.
- Implementa la funcionalidad de la aplicación.
- Expone la funcionalidad (no su implementación) por medio de una interfaz pública.

Vista

- Destinada únicamente a la representación de la interfaz de usuario.
- Recibe datos del modelo.
- Da formato a los datos
- Presenta los datos al cliente de la aplicación, ya sea una persona u otra aplicación.

Controlador

- Destinada a servir de comunicación bidireccional entre las capas anteriores.
- Controla el comportamiento de la aplicación.
- Coordina qué paso se ejecuta después de otro.
- Elige la vista a mostrar al usuario.

Puede haber un controlador por cada funcionalidad de la aplicación o un único controlador. Esta última estrategia de implementación permite centralizar el control de la aplicación de forma de facilitar su gestión y será la estrategia que utilizaremos en nuestro ejemplo.

El objetivo de esté paradigma es poder crear una GUI sin dependencia, y con ello tener las ventajas de un sistema extensible, manipulable e independiente entre modelos con sus propias funcionalidades y así te preocupas solo por una capa al momento de diseñar obteniendo como resultado un sistema funcional. Donde el usuario se enfrentará a algo fácil, pero poderoso.

3.3. Modelo Analítico de Descripción de Tareas Orientado a la Especificación de Interfaces.

-(MAD* (MAD STAR): Modèle Analytique de Description de tâcheS orienTé spécificAtion d'inteRfaces)-.

Gramática y semántica -Unidad-Tarea-

Cuerpo: <Núcleo>, <Pre / post-condiciones>, <Estado del mundo>

Descomposición: <Tiempo>, <Constructores de orden>, <Constructores de sincronización>, <Constructores auxiliares>

Cuerpo de la Unidad-Tarea

El *Cuerpo de la Unidad-Tarea* representa el núcleo del modelo MAD*. Es ahí donde se definen las informaciones que caracterizan cada elemento, las condiciones necesarias para su ejecución, los objetos utilizados, etc. Desde un punto de vista informático, el *cuerpo de la unidad-tarea* puede ser estudiado como una entidad formada por tres estructuras diferentes figura 3.3.1.

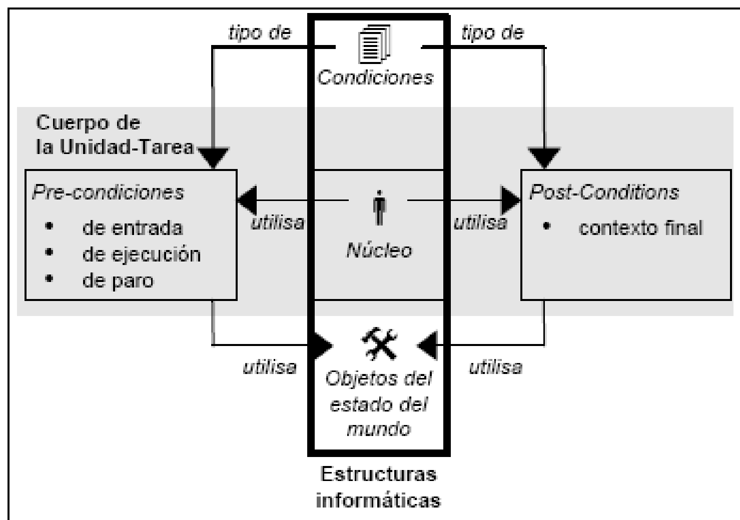


Figura *3.3.1.* Cuerpo de la Unidad-Tarea.

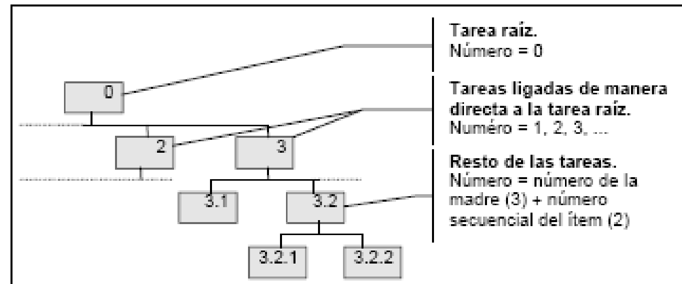
Se puede observar una estructura central (el *Núcleo*), con ligas hacia instancias del objeto *Condición* (las pre y post-condiciones). A su vez, las pre y post-condiciones tienen ligas hacia los Objetos del estado del mundo (es decir, los objetos que modelan las informaciones necesarias para la tarea). Estas tres estructuras son detalladas en este documento.

Núcleo

Nombre (*Tipo alfanumérico):* Designa el o los términos con los que el operador nombra su tarea. Ej. Nombre de la tarea: « Verificar la entrada de un automóvil »

Objetivo (*Tipo alfanumérico):* Atributo aclaratorio en lenguaje natural, que busca facilitar la comprensión de la información modelada. Ej. Objetivo de la tarea: « Confirmar que el auto está en la plataforma »

Número (*Tipo alfanumérico):* Cada elemento de la descripción se identifica por un número. Dicho número es una cadena que se forma a partir del número de la tarea "madre" (inmediata superior), y el número secuencial que le corresponde. Ej. Número de la tarea: « 3.2.3 »



Esta numeración presenta dos excepciones: la tarea raíz, y las tareas directamente ligadas a ella. En efecto, la tarea raíz recibe siempre el número cero; las tareas directamente ligadas a ella reciben únicamente el número secuencial que les corresponde (1, 2...).

Prioridad (*Tipo entero natural, corto {0 - 255}*): Indica la prioridad del elemento con respecto a los otros elementos de la descripción: entre más elevado sea su valor, más rápidamente será tratado. Este atributo permite resolver conflictos cuando dos elementos diferentes desean tomar el control. Ej. Prioridad de la tarea: 10.

Facultativa (*Tipo Booleano):* Indica si la tarea es facultativa u obligatoria. En un primer momento, todos los elementos son considerados como indispensables para la ejecución de la tarea, y por lo tanto, obligatorios. Cuando el usuario puede elegir entre realizar o no una tarea, el elemento correspondiente deberá ser señalado como facultativo. Ej. Tarea facultativa: Falso.

Relaciones entre elementos obligatorios y elementos facultativos:

- Une tarea con sub-tareas facultativas y obligatorias, será considerada como terminada cuando todas las sub-tareas obligatorias sean terminadas;
- Una tarea que sólo tenga sub-tareas facultativas, será considerada como facultativa;
- Una tarea que sólo tenga tareas obligatorias, no será considerada forzosamente como obligatoria; las sub-tareas de una tarea facultativa no son obligatoriamente facultativas.

Interrumpir (*Tipo Booleano):* Indica si una tarea puede ser interrumpida durante su ejecución, o no. Este atributo es importante, puesto que permite asegurar que las tareas críticas no serán interrumpidas por otras tareas durante su ejecución. Ej. tarea interrumpir: Verdadero.

Relaciones entre elementos interrumpir y no interrumpir:

- Una tarea interrumpir puede ser descompuesta en sub-tareas interrumpir o no-interrumpir;
- Una tarea no-interrumpir no puede contener otras sub-tareas no interrumpir.

Tipo (*{sensori-motriz,cognitiva}):* Indica si se trata de una tarea cognitiva o sensori-motriz. El objetivo es identificar de entre las tareas de alto nivel de abstracción, aquellas que requieren de habilidades cognitivas particulares, y esto, con la finalidad de poder identificar los objetivos fundamentales del usuario. Ej. Tipo de la tarea: Sensori-motriz (verificar la llegada del auto).

Modalidad (*{manual, automática,interactiva}):* Indica de que manera, y por quién, es realizada una tarea (una tarea manual es realizada por el operador, mientras que una tarea automática es realizada por el sistema). Ej. Modalidad de la tarea: interactiva (teclear los datos del auto en el sistema).

Relaciones entre elementos interactivos, automáticos y manuales:

- Una tarea interactiva puede ser descompuesta en sub-tareas interactivas, manuales o automáticas;
- Una tarea automática no puede contener más que sub-tareas automáticas;
- Una tarea manual no puede contener más que sub-tareas manuales.

Importancia (*{importante, relativamente importante, secundaria}):* Atributo que indica la importancia de la tarea. Su objetivo es establecer una jerarquía de tareas en función de su importancia, diferente a la obtenida en base a su división en otras tareas más simples. El atributo encierra otros dos sub-atributos:

- La **frecuencia** (elevada, media, baja) que es el número de veces que la tarea es efectuada, o llamada por otras tareas;
- Las **entidades importantes** entre los objetos de la tarea, es decir, si la tarea utiliza objetos que son utilizados con frecuencia por el resto de las tareas. Ej. Importancia de la tarea: relativa [frecuencia: media; entidades importantes: auto, rampa].

Papel del usuario *(Tipo alfanumérico):* Este atributo permite describir al usuario, así como la actividad que realiza, no sólo en términos de las acciones que efectúa, sino considerando su experiencia, sus competencias y conocimientos. Se define en base a cuatro sub-atributos.

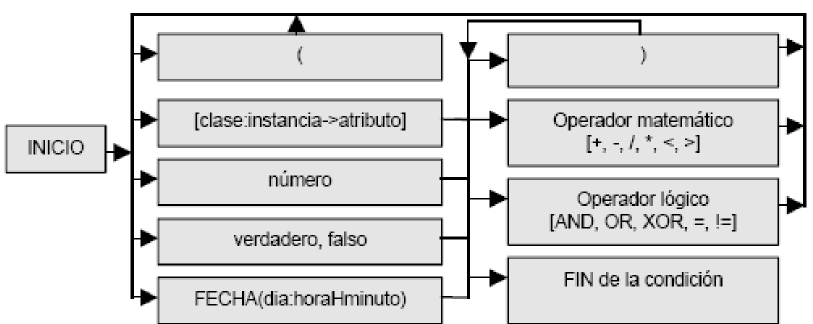
- **papel**: función que desarrolla el usuario (ej. secretaria, administrativo, etc.);
- **experiencia**: indica la experiencia del usuario en la tarea que realiza (novato, principiante, ocasional, especialista, experto);
- **competencias**: atributo que representa los conocimientos que el usuario tiene de la tarea, y que el conceptor no puede representar con los operadores y los atributos MAD* ;
- **preferencias**: permite al conceptor dejar una traza sobre las preferencias del usuario, en términos de sistemas de diálogo (lenguajes de comandos, formularios, menús, manipulación directa).

Ejem. Papel del usuario: Jefe de circulación [**papel**: verificar las entradas y las salidas de los trenes en la estación; **experiencia**: experto; **competencias**: identificación

de retrasos, replanificación de la circulación; **preferencias**: lenguajes de comandos, formularios].

Condición

Gramática General



Tipo condición *(Tipo entero corto{0 - 3}):* Este atributo señala si se trata de una pre-condición de ejecución, de arranque, o de paro; o una post-condición.

Texto de la Condición (*Tipo alfanumérico):* El texto de la condición se almacena aquí.

Resultado de la verificación (*Tipo Booleano*): Indica el resultado de la última verificación.

Ej. de condiciones:

1. Tipo de la condición: 0 (condición de ejecución).
Texto de la condición:

[tren:trenLlegada->viaAsignada] = Verdadero AND [tren:trenLlegada->horaLlegada] = [estación:estaciónMéxico->horaPrevista].

Resultado: Falso (la condición no fue satisfecha en la última verificación. La tarea no puede ser ejecutada)

2. Tipo de la condición: 2 (condición de paro).
Texto de la condición:

[niño:Pepito->edad] > 12 OR ([niño:Pepito->peso>35] AND [niño:Pepito->estatura] > 130)

Resultado: Falso (la condición no fue satisfecha en la última verificación, la tarea continua iterando)

3. Tipo de la condición: 1 (condición de arranque).
Texto de la condición:

[madera:maderaVentana->area] > 15 AND (([madera:maderaVentana->precio] * 5) < 1000)

Resultado: Verdadero (la condición fue satisfecha. La tarea pide ser ejecutada) 4. etc.

Pre-condiciones

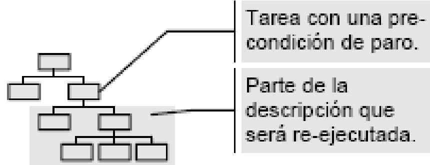
Estado Inicial del Mundo: Lista de objetos ([clase:instancia->atributo]), utilizados en las pre-condiciones de la tarea.

Condición de Ejecución: Indica el contexto necesario para la ejecución de una tarea. Esta condición debe ser **obligatoriamente** satisfecha para que la tarea sea ejecutada.

Condición de Arranque: La verificación de esta condición señala que un evento externo demanda la ejecución de la tarea. Aún cuando la condición de arranque es verificada, la ejecución de la tarea depende de otras condiciones:

1. la condición de ejecución también fue verificada;
2. la tarea que actualmente se ejecuta, se puede interrumpir;
3. la prioridad de la tarea es más elevada que la de la tarea en ejecución;

Condición de Paro: El objetivo de esta condición es definir de manera precisa cuantas veces, o hasta que condición, se deberá repetir la ejecución de una parte de la descripción (la que se encuentra bajo la tarea que integra este tipo de condiciones, ver figura abajo).



Post-condición

Estado final del Mundo: Lista de objetos (*[clase: instancia->atributo]*), incluidos en la post-condición.

Condición de Salida: Especifica los objetos que fueron modificados por la tarea, así como las modificaciones realizadas.

Descomposición de la unidad-tarea: En esta parte del modelo se definen los diferentes constructores que permiten subdividir de manera jerárquica una tarea en otras más simples. Dos categorías de operadores son definidos: los operadores de sincronización, (SECuencial, PARalelo, SIMultaneo), y los operadores de ordenamiento (Y, O, ALTernativo), que son especializaciones del constructor paralelo. Un tercer sub-grupo de atributos permite señalar el momento en el que la tarea debe iniciar, terminar, o el tiempo que debe permanecer activa.

Tiempo

Inicio: Instante en el que la tarea debe ser ejecutada.
Fin: Instante en el que la tarea debe ser detenida.
Duración: *Intervalo de tiempo durante el cual la tarea debe permanecer activa.*

Constructores de sincronización

Secuencial: Las sub-tareas son ejecutadas en el orden. Las tareas facultativas pueden ser ignoradas. La ejecución de la tarea se termina una vez que la última sub-tarea termina su ejecución, o es ignorada.

Paralelo: Las sub-tareas son ejecutadas en cualquier orden, y no comparten datos. La ejecución de la tarea se termina una vez que todas las tareas obligatorias fueron ejecutadas.

Simultaneo: n sub-tareas son realizadas al mismo tiempo por m usuarios. La ejecución de la tarea se termina una vez que las n sub-tareas obligatorias fueron ejecutadas.

Constructores de ordenamiento

Y**:** Las subs-tareas son ejecutadas en cualquier orden, y comparten los datos. La ejecución de la tarea se termina una vez que todas las sub-tareas obligatorias fueron ejecutadas.
O**:** Las sub-tareas son ejecutadas en cualquier orden, y comparten datos. La ejecución de la tarea se termina cuando al menos una de las sub-tareas obligatorias es ejecutada.

Alternativo: Una y solamente una de las sub-tareas es ejecutada.

Constructores auxiliares

Elemental: Constructor que señala que una tarea no tiene más sub-tareas. Aún cuando se le puede considerar redundante, este constructor *permite señalar una tarea como el final formal de una rama del árbol, y a partir de esto, hacer las pruebas de verificación correspondientes.*

Desconocido: Se utiliza cuando se ignora que constructor (de sincronización o de ordenamiento), es necesario utilizar. Este constructor permite señalar un elemento como incompleto, generar reportes de las partes de la descripción que aún deben ser detalladas, y sobre todo, continuar con la descripción a pesar de la falta de información.

3.4. PRINCIPIOS

3.4.1. Principio 1. Reconoce la diversidad

Cuando la diversidad humana es multiplicada por el amplio rango de situaciones, tareas, frecuencias de uso, el conjunto de posibilidades de diseño se hace enorme. El diseñador puede responder al escoger de un espectro de estilos de interacción.

3.4.1.1. Perfiles de usuarios

"Conoce al usuario" Todos los diseños deben comenzar con el entendimiento de los usuarios, incluyendo perfiles de población, género, habilidades físicas, educación, cultura, entrenamiento, motivación, metas y personalidad.

El proceso de conocer al usuario nunca termina. Cada paso en entender al usuario y en reconocerlo como individuo cuyo punto de vista es diferente al del diseñador es probablemente un paso más cercano a un diseño exitoso.

Por ejemplo, una separación genérica puede llevar a estas siguientes metas:

- *Usuarios novatos o primerizos*: En los novatos se asume que saben poco de la tarea o de conceptos de interfaces. En contraste, lo primerizos son profesionales que saben los conceptos de la tarea, pero poco de conceptos de interfaces.

Por tanto el número de acciones deben ser pequeñas, una retroalimentación acerca de cada tarea es de mucha ayuda y al cometer errores deben ser proveídos mensajes de errores específicos y constructivos.

- *Usuarios conocedores no frecuentes:* Pueden ser conocedores pero usuarios frecuentes de una variedad de sistemas. Ellos tienen conceptos estables acerca de la tarea y conocimiento de conceptos de interfaces, pero tendrán problemas en retener la estructura de menús o la localización de características especiales. Les ayudará tener secuencias consistentes de acciones, y mensajes con significados.

- *Usuarios expertos y frecuentes:* Están bien familiarizados con la tarea y con conceptos de interfaces y buscan terminar su trabajo rápidamente. Ellos demandan rápidos tiempos de respuesta, retroalimentaciones breves y sin distracciones, y la capacidad de llevar acabo acciones con unos pocos teclazos o selecciones. Para ellos son requeridos cadenas de comandos, accesos rápidos en menús, abreviaciones, y otros aceleradores.

3.4.1.2. Estilos de interacción

Cuando el análisis de la tarea esta completo, el diseñador puede escoger de estos estilos primarios de interacción:

Manipulación directa, selección de menús, llenado de formas, lenguaje de comandos y lenguaje natural.

Selección de menús: Selecciona la más apropiada para su tarea, y observa el efecto. Requiere de poco aprendizaje y memorización.

Llenado de formas: Es apropiado cuando es necesario capturar datos, deben entender las etiquetas de los campos, saber los valores permitidos y el método para capturar los datos. Apropiado para usuarios con conocimientos pero que no son frecuentes.

Lenguaje de comandos: Para usuarios frecuentes, le da control e iniciativa. Los usuarios aprenden la sintaxis y pueden expresar posibilidades complejas muy rápidamente. Sin embargo los rangos de error son muy altos, un entrenamiento es necesario, y la retención puede ser pobre.

Lenguaje natural: La esperanza de que las computadoras respondan a frases o enunciados arbitrarios ha atraído mucho, pero su éxito ha sido limitado. Este estilo de interacción puede ser más lento, pero se necesita poco entendimiento.

Mezclar varios estilos de interacción puede ser apropiado, cuado la tarea requerida y los usuarios sean diversos.

Estilo	Ventajas	Desventajas
Manipulación directa	Presenta los conceptos de la tarea. Permite un aprendizaje rápido. Permite fácil retención. Da paso a la exploración.	Difícil de programar. Requiere de un poderoso manejo de gráficos y dispositivos para señalar.
Selección en menús	Acorta el aprendizaje. Reduce el número de tecleos. Estructuras de decisión. Permiten herramientas de manejos de diálogos. Permiten un soporte fácil de manejo de errores.	Se puede llenar de menús. Pueden alentar a los usuarios frecuentes. Consumen espacio de pantalla.
Llenado de formas	Simplifican la entrada de datos. Requiere de poco entrenamiento. Da ayuda conveniente.	Consume espacio de pantalla. Requiere buena terminología en las etiquetas. El usuario debe saber los valores permitidos.
Lenguajes de comandos	Es flexible. Soporta la iniciativa del usuario. Permite creación de macros.	Tiene un pobre manejo de errores. Requiere de un alto entrenamiento y memorización.

Tabla *3.4.1.2.1.* Estilos de Interacción.

3.4.2. Principio 2. Usa las 8 reglas doradas del diseño de interfaces

Los principios subyacentes del diseño que son aplicables en la mayor parte de los sistemas de interacción. Estos principios, derivados heurísticamente de la experiencia, deben ser validos y refinados.

1. *Esfuérzate en la Consistencia:* Es la que mas frecuentemente se viola; seguirla es un poco difícil debido a que hay muchas formas de consistencia. Secuencias consistentes de acciones deben ser requeridas en situaciones similares; la misma terminología debe ser usada en avisos, menús, y pantallas de ayuda; y el color consistente, la forma de acomodar las cosas, fuentes de letras.
2. *Permite a usuarios frecuentes usar atajos:* Mientras la frecuencia de uso se incrementa, el usuario desea reducir el número de interacciones e incrementar el paso de interacción. Abreviaciones, teclas especiales, etc.
3. *Ofrece respuestas informativas:* A toda acción del usuario, debe existir una respuesta del sistema.
4. *Diseña diálogos que lleven a un fin:* Secuencias de acciones deben ser organizadas dentro de grupos con un principio, una parte intermedia, y un fin. La respuesta informativa al completar un grupo de acciones, le da a los operadores el sentido de realización.
5. *Ofrece prevención de errores y manejo de errores simple:* Tanto como se pueda diseña el sistema de tal forma que el usuario no pueda cometer grandes errores; por ejemplo, preferir menús envés de llenando libre y no permitir caracteres alfabéticos en campos numéricos.
6. *Permite fáciles revocaciones para las acciones:* Tanto como se pueda las acciones deben ser reversibles. Esto calma al usuario.
7. *Da soporte de locación interna de control:* Operadores experimentados desean sentirse que están a cargo del sistema y que el sistema responde a sus acciones. Gaines en 1981 capturó parte de este principio con su regla evita la no casualidad y su estimulo por hacer a los usuarios los iniciadores de acciones envés de los que responden a acciones.
8. *Reduce la carga de memoria de corto plazo:* La limitación del procedimiento de información en la memoria de corto plazo del humano requiere que los despliegues se mantengan simples, despliegue de múltiples paginas sean consolidados, sea reducida la frecuencia del movimiento de ventanas, y suficiente tiempo de entrenamiento sea asignado para códigos y secuencia de acciones. Donde apropiados, acceso en línea a formas de sintaxis de comandos, abreviaciones, códigos, y otra información deben ser proveídos.

3.4.3. Principio 3. Prevenir errores

3.4.3.1. Correcto apareamiento de pares

Un ejemplo de este es la falla de proveer el paréntesis derecho para cerrar un paréntesis izquierdo abierto, o la falta de cerrar marcas abiertas. Esto puede prevenirse, cuando se teclea el paréntesis izquierdo se genere un paréntesis izquierdo y un derecho y dejar al cursor en medio de ellos, o bien que cuando el usuario teclea uno izquierdo, la pantalla muestra en la esquina inferior derecha un mensaje indicando la necesidad de un paréntesis derecho, hasta que este es tecleado.

3.4.3.2. Secuencias completas

Algunas veces, una acción requiere de varios pasos o comando para completar la tarea. Debido a que la gente puede olvidar completar cada paso de una acción, diseñadores intentan ofrecer una secuencia de pasos como una sola acción.

3.4.3.3. Comandos correctos

Consideran estos típicos errores en lenguajes de comandos: invocan un comando que no esta disponible, pide un archivo que no existe o captura valores de datos que no son aceptables. Algunos sistemas ofrecen un auto completar, el cual permite al usuario presionar algunas letras de un comando con significado para el sistema. Este puede ahorrar teclear algunas letras, pero también puede distraer debido a que el usuario debe considerar cuantos caracteres debe teclear para que el comando sea reconocido, y debe verificar que la computadora ha hecho la auto completación que se quería.

3.5. GUÍA PARA EL DESPLIEGUE DE DATOS

3.5.1. Organización del despliegue

Smith y Moiser en 1986 ofrecieron 5 objetivos para el despliegue de datos los cuales hasta ahora siguen siendo vitales.

Consistencia de despliegue de datos: Durante el proceso de diseño, la terminología, abreviaciones, formatos, colores, deben ser estandarizados y controlados por un diccionario.

Eficiente asimilación de información por el usuario: El formato debe ser familiar para el operador y debe ser relacionado a las tareas requeridas a ser realizadas.

Mínima carga de memoria en el usuario: Los usuarios no se les debe requerir que recuerden información de una pantalla para usar en otra pantalla.

Compatibilidad entre el despliegue de datos y la captura de datos: El formato de información desplegada debe estar claramente ligado con el formato de la captura de datos.

Flexibilidad para el usuario de controlar el despliegue de datos: Los usuarios deben poder poner la información en el despliegue en la forma en que más le conviene.

3.5.2. Obtener la atención del usuario

Como información substancial puede ser presentada a los usuarios, condiciones excepcionales o información dependiente del tiempo debe ser presentada de tal forma que atraiga la atención.

Intensidad: Utiliza solo dos niveles, con un uso limitado de intensidad alta para llamar la atención.

Marcar: Subraya, encierra en una caja, apunta con una flecha, o usa un indicador como el asterisco.

Tamaño: Utiliza hasta 4 tamaños, con tamaños grandes para atraer la atención.

Escoge el tipo de letra: Usa hasta 3 tipos distintos.

Video inverso: Utiliza un coloreado inverso.

Parpadeo: Úsalos con mucho cuidado y en áreas limitadas.

Color: Utiliza hasta 4 colores, con colores adicionales reservados para usos ocasionales.

Color cambiantes: Utiliza cambios de color (parpadeo entre color y otro) con cuidado y en áreas limitadas.

Audio: Utiliza tonos suaves para una retroalimentación regular y positiva y sonidos fuertes para raras condiciones de emergencia.

3.6. GUÍA PARA LA CAPTURA DE DATOS

La captura de datos puede ocupar una fracción substancial de tiempo del operador. Smith y Moiser en 1986 ofrecieron cinco objetivos de alto nivel para la captura de datos:

Consistencia de transacciones de la captura de datos. Secuencias similares de acciones deben ser usadas bajo todas las condiciones; delimitadas similares, abreviaciones deben ser usados.

Acciones mínimas de entrada por el usuario. Menos número de acciones de entrada significan mayor productividad y -a veces- menor posibilidad de cometer errores.

Mínima carga de memoria en los usuarios. Los usuarios no deben de recordar largas listas de código y comandos de sintaxis compleja.

Compatibilidad de la captura de datos con el despliegue de datos. El formato de entrada de códigos y comandos de sintaxis compleja.

Flexibilidad para el usuario controle la entrada de datos. Usuarios experimentados pueden preferir meter información en una secuencia que ellos controlen. Flexibilidad debe ser usada con cuidado, ya que va en contra del principio de consistencia.

Recomendación para el diseño de GUI´s. La proporcionalidad debe ser consistente especialmente para cajas de diálogos que tienen un diseño visual y una funcionalidad similar. El área libre en una caja de diálogo debe ser de al menos el 20% del área total Los márgenes deben ser consistentes. Los Widgets dentro de una caja de diálogo deben estar alineados horizontal y verticalmente. Deben evitarse diseños muy densos buscar la consistencia en colores (foreground y background) y tipografías. La localización y tamaño de Widgets frecuentemente usados debe ser consistente. Ser consistente en terminología buscar la consistencia en las etiquetas de botones esto es, sinónimos como “Abort”, “Cancel”, “Close” y “Exit” no deben ser usadas para tareas similares.

3.7. EVALUACIÓN DE INTERFACES

Factores a tomar en cuanta: etapa del diseño (primera, media, última); que tan nuevo es el proyecto (bien definido contra exploración), novedad; número de usuarios esperados, que tan crítico es la interfaz (por ejemplo, sistemas médicos contra un soporte a exhibición de museos); costos de producto y finanzas asignadas para evaluar; tiempo disponible; experiencia del grupo de diseño y evaluación.

El rango de la evaluación puede ser desde una evaluación ambiciosa de dos años con múltiples fases hasta una evaluación de tres días con seis usuarios.

Para evaluar una interfaz se tienen varias formas de hacerlo: Revisiones por expertos, Revisiones en laboratorios, cuestionarios y entrevistas, y experimentos orientados a psicología.

3.7.1. Revisiones por expertos

Estos métodos dependen de expertos con experiencia ya sea en la aplicación o en dominios de interfaces de usuario. Hay una gran variedad de métodos de revisiones de los que podemos escoger:

Evaluación Heurística. Los expertos critican la interfaz con una lista de heurística como las 8 reglar de oro.

Guías. La interfaz es evaluada para que cumplan algunas guías predefinidas para hacer interfaces.

Inspección de consistencia. Los expertos verifican consistencia sobre una familia de interfaces, evaluando consistencia de terminología, color, la forma en que esta todo acomodado, formatos de entrada y salida, etc.

Paseo cognitivo. Los expertos simulan usuarios usando la interfaz y hacer las típicas tareas de los usuarios finales.

Inspección formal de uso. Los expertos llaman a una junta con un juez moderador, para presentar la interfaz y discutir sus meritos y sus debilidades. La gente del grupo de diseñadores pueden refutar la evidencia acerca de los problemas en un formato adversario.

3.7.2. Laboratorios y pruebas de uso

Esta emergió desde los 80´s y fue por la necesidad de los usuarios, dando como sorpresa en pruebas de uso no solo aceleró los proyectos sino que también redujo los costos dramáticamente.

El movimiento hacia las pruebas de uso estimuló la construcción de laboratorios con dos áreas de 3x3m., una para los participantes para hacer su trabajo y otro, separado por un vidrio polarizado para los observadores (diseñadores, jefes y compradores).

Los participantes deben representar la comunidad de usuarios, con conocimientos en cómputo, voluntarios y que tengan toda la libertad de poder no seguir si así lo desea.

Por ejemplo los diseñadores de juegos hacen pruebas de tipo "¿Puede tronar esto?".

Las desventajas pueden ser que se enfatiza en usuarios que usan el sistema por primera vez y limita la cobertura de todas las características de la interfaz. Este y otros problemas obligan a suplementarse con una revisión del experto.

3.7.3. Cuestionarios

Los cuestionario escrito son familiares, baratos y generalmente aceptables, acompañantes de pruebas de uso y revisiones de expertos. Sus metas pueden ser atadas a los modelos creados; otras metas a ser constatadas son: los usuario (edad, sexo, orígenes, educación, ingresos), experiencia con computadoras, responsabilidades laborales, personalidad, el sentimiento que les dejo después de usar la interfaz (confusa contra clara).

A manera de conclusión diremos que una interfaz debe estar diseñada en función del usuario. Rubin define la idea como la renovación conceptual de términos tipo "ergonomía" -o lo que él llama "*human factors engineering*"- señalando que el concepto no sólo representa las técnicas, procesos o métodos para diseñar sistemas y productos "usables" sino que el usuario es el centro del proceso de diseño.

El no seguir las necesidades del usuario lleva a crear productos no sólo no usables sino incluso inútiles. Debe darse un proceso de comunicación con el usuario y éste debe tomar parte activa dentro del desarrollo del sistema.

Por lo tanto, las tendencias de diseño de nuevas herramientas deben cambiar si se quiere ofrecer un mejor producto.

Sin embargo la visión del usuario, es lo más difícil de reflejar ya que cada usuario tiene su idea de lo que debe ser un sistema, organización, presentación, o terminología, en función de sus experiencias y expectativas –figura 3.7.3.1-

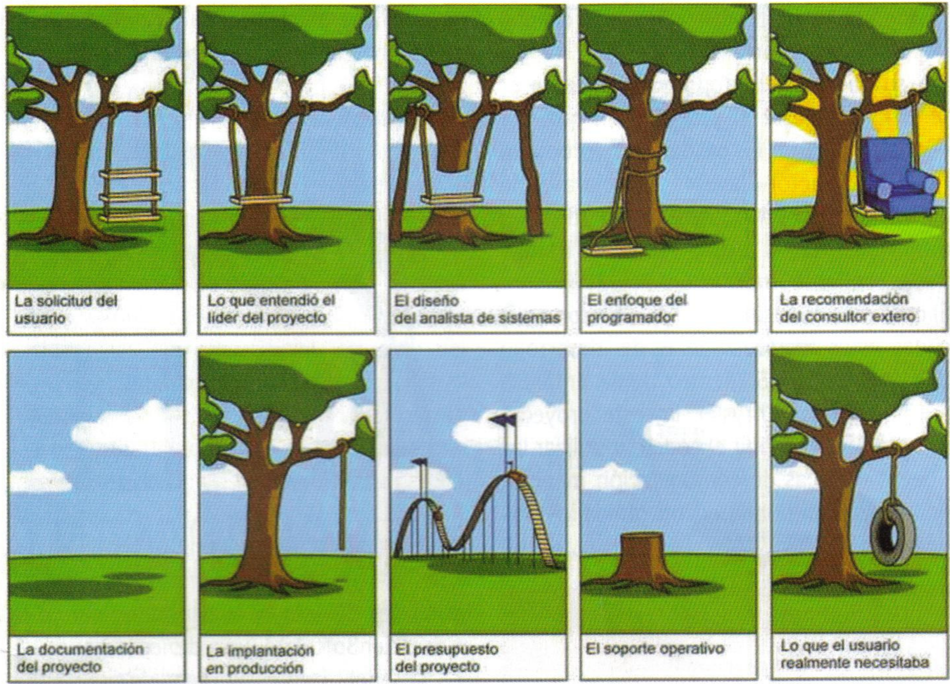


Figura *3.7.3.1.* Al desarrollar.

Así, un sistema bien valorado será aquel que le permita realizar sus tareas de forma adecuada frente a otro que no se lo permita, e, incluso, aquel que le permita "disfrutar" del proceso frente a otro que no.

Para llegar a conocer esta visión es necesario comunicarse con el usuario y mantener un proceso de alimentación del sistema derivado de su evaluación, teniendo siempre presente que la interfaz diseñada no gustará a todos por igual. La visión del diseñador es básica para el éxito de la interfaz, ya que debe funcionar como un arquitecto que transforme las necesidades del usuario en procedimientos realizables por el desarrollador del sistema.

La visión del desarrollador es la que finalmente se convierte explícitamente en la interfaz. Debe tenerse en cuenta que el conocimiento en sistemas operativos, herramientas de desarrollo o recomendaciones de programación, no conlleva, necesariamente, el conocimiento de las necesidades y preferencias del usuario. Es más, en la mayoría de los casos, parece que su visión va más hacia la utilidad o eficiencia del producto que hacia su usabilidad incluso en la fase de redesarrollo: el programador debe escuchar tanto al diseñador como al usuario para desarrollar un producto de calidad.

Dentro de esta división, el diseñador del sistema debe ser el vínculo de unión entre desarrollador y usuario, actuando mediante equipos interactivos de trabajo cuyos

integrantes sean especialistas en ciencias cognitivas y que tengan al usuario como meta final, ya que el objetivo de una aplicación es permitir al usuario que realice sus tareas de la mejor manera posible.

Por lo tanto, el diseño centrado en el usuario deviene una necesidad: todas las actuaciones de desarrollo deben tener al usuario como foco central, siendo básico conocer sus ideas y su modo de actuación. En consecuencia, los objetivos, el plan de desarrollo y el contexto de funcionamiento de una interfaz deben extrapolarse del punto de vista del usuario.

Averiguar este punto de vista tiene sus dificultades, ya que incluir en todo proceso un elemento tan imprevisible como es el comportamiento humano es complejo: aconsejar obviedades como “conoce a tu usuario” es sencillo; pero conseguirlo no lo es.

Además, el diseñar interfaces mediante esta línea de pensamiento requiere no sólo identificar y categorizar a los usuarios finales, sino estar en permanente contacto con ellos. Este contacto debe estar pensado, programado y estructurado para no dar una impresión de precipitación e inexperiencia: un vehículo de comunicación con el usuario resultará de gran interés si se sabe llevar a cabo. Este contacto debe medirse a través de los procedimientos de evaluación de sistemas en los que el usuario tenga un papel más activo.

CAPÍTULO 4

DESARROLLO DE LA INTERAFAZ GRÁFICA DE USUARIO Y RESULTADOS OBTENIDOS

Capítulo 4. DESARROLLO DE LA INTERAFAZ GRÁFICA DE USUARIO Y RESULTADOS OBTENIDOS

4.1. METODOLOGÍAS DE DESARROLLO

Existe un tópico sobre la labor del programador en la gran empresa de desarrollo, inmortalizado por cientos de chistes e historias, entre las que probablemente la más conocida sea la tira de comics de Dilbert[1]. Según éste, el programador es un tipo aislado encerrado en su cubículo, un genio solitario con una vida completamente gris y aburrida. Y hasta cierto punto hay algo de verdad en este tópico, especialmente en las grandes empresas de desarrollo americanas. Aparte de esto, todos conocemos lo complicado que resulta llegar al final de un proyecto, la inmensa cantidad de horas exactas que hay que trabajar, y que la jornada laboral del informático no es precisamente de nueve a cinco, sino más bien de siete a nueve. Está claro que esta situación es indeseable, pero ¿quién tiene la culpa?, el problema está esencialmente en nuestro método de trabajo.

Ahora imaginemos una situación completamente distinta. El programador ya no trabaja solo, sino que siempre tiene al lado un compañero que trabaja en el mismo proyecto y sobre el mismo código. No trabajan aislados sobre unas especificaciones en papel, sino que tienen al cliente ahí mismo, en su propio edificio, listo para contestar a sus preguntas y a sentirse involucrado en el proyecto. Mejor todavía, se respeta escrupulosamente su horario. ¿Esto es un sueño? No, en realidad no. Se trata simplemente de que la empresa ha decidido adoptar los principios de la Programación Extrema (XP), la más conocida de las llamadas metodologías ágiles –observar el Apéndice D-.

Pero ¿de donde sale todo esto? Como condiciones de trabajo parecen ideales, pero ¿cómo es posible que la empresa considere que pueden ser rentables? ¿Qué es una metodología ágil y dónde puedo comprarme una?

Muy buena pregunta, pero la respuesta no es sencilla. Antes de hablar de las metodologías ágiles, debemos plantearnos qué es una metodología. “Usamos metodologías porque tenemos miedo. Miedo a desarrollar un mal producto, miedo a desarrollar un producto de mala calidad, miedo a los retrasos, [...]”[2].

4.1.1. ¿Qué es una metodología y por qué nos interesa?

Las metodologías aparecen en la construcción de software como un conjunto de métodos y técnicas fundamentados en una filosofía de trabajo, preestablecida como guía en la consecución del objetivo esencial, la construcción de un producto software, y en el proceso necesario para construirlo. Las experiencias que nos han suministrado otros campos, como el de la ingeniería, han hecho que la construcción de software haya pasado de un proceso artesanal a un proceso de ingeniería. Cualquier ingeniería trabaja

[1] Scott McKenzie, http://www.dilbert.com
[2] Según Robert Martin.

de una forma sistemática, controlando sus procesos y su producto, evaluando la calidad, escogiendo lo mejor, las mejores prácticas, “lo que da resultado”.

A nadie se le pasaría por la cabeza que un ingeniero mecánico no se molestase en comprobar el calibre de las tuercas y tornillos al construir un motor, o que un ingeniero de caminos construyese un puente colocando piedras en un río y las fuese encajando casi al azar, tal como le dicte su experiencia y su imaginación. Sin embargo, no esperamos que un ingeniero informático siga un proceso sistemático en un desarrollo; nos parece normal escribir código sin un plan, basados sólo en nuestra experiencia, y ni siquiera nos molestamos en evaluar la calidad del código o los componentes que introducimos en nuestros sistemas, lo hayamos escrito nosotros o no. ¿Para qué hacerlo, si somos lo bastante buenos? No necesitamos perder tiempo siguiendo un método.

Tal vez podamos pensar que eso es adecuado para los grandes sistemas que hacen las grandes empresas, pero que en desarrollos pequeños no es necesario. Esto es, por supuesto, un error. Un sistema más pequeño necesita, por supuesto, un plan más pequeño, pero sigue siendo necesario un plan.

El beneficio de usar una metodología no se muestra necesariamente en el desarrollo inicial, sino que resulta mucho más evidente en futuros trabajos sobre el mismo sistema. Hemos de tener en cuenta que, como es bien conocido, la fase de mantenimiento agrupa entre el 60 y el 80% de los costes de un proyecto de software[3]. No es extraño el caso de proyectos enteros tirados a la basura a causa del aumento de la complejidad del mantenimiento, muy superior a la complejidad resultante de reiniciar el proyecto desde cero.

Pero, ¿es posible programar sin usar una metodología? Por supuesto, todos sabemos que sí, lo hemos hecho cuando estábamos aprendiendo. A esta táctica se le suele llamar “programar a lo vaquero” (cowboy coding). Aunque a muchos les gusta pensar que esto hace referencia al hecho de que este programador cowboy es un intrépido, que programa “a pelo”, sin ninguna red de seguridad, en realidad el término es totalmente peyorativo.

4.1.2. Metodologías tradicionales contra Metodologías ágiles

Las metodologías surgen como una técnica de ingeniería de software que intentará aportar el orden y la estabilidad de la que carecían los desarrollos de software iniciales. Así proporcionan un conjunto de técnicas, métodos y normas que en ocasiones pueden resultar muy prolijos.

Sin embargo, las estrictas reglas impuestas pueden aumentar significativamente la carga de trabajo aparente de un desarrollo software; hablamos de cientos de páginas de documentación de análisis y diseño, imprescindibles en un desarrollo sistemático, pero que son percibidas a menudo como una carga excesiva en el contexto de determinados proyectos.

[3] Roger S. Pressman, *Ingeniería de software: Un enfoque práctico,* McGrawHill, 2002.

Y es que, en ocasiones, la prioridad de un proyecto es realizar un desarrollo lo más rápido y adaptable posible sin preocuparnos en principio por aspectos como el costo o la mantenibilidad. Es en este contexto en el que surgen las metodologías ágiles, entre las que el ejemplo clásico es sin duda la Programación Extrema (XP) de Kent Beck, uno de los padres de los patrones de diseño.

Estamos hablando de mediados de los años 90, una época en la que la industria comenzaba a moverse demasiado rápido para los estrictos modelos impuestos por las metodologías vigentes. Quizá demasiado estrictas, si, pero como quedó demostrado el no utilizarlas podía conllevar resultados catastróficos.

Las metodologías ágiles suponen la esperada respuesta, loa alternativa a la propuesta original de las metodologías tradicionales, llamadas por contraposición metodologías "pesadas". En febrero del 2001 destacaría un evento importante para la historia de las metodologías. En esa fecha os representantes de las diferentes corrientes ágiles se reúnen con el objetivo de encontrar un nexo común entre todas ellas. De ahí salió, entre otras cosas, el Manifiesto Ágil[4].

Por lo general podemos tender a relacionar ágil con rápido. Sin embargo en nuestro contexto algo ágil no es necesariamente algo rápido. Para entender las diferencias entre ambos mundos nada mejor que partir de los supuestos beneficios que aportan las metodologías ágiles. Así, en el manifiesto ágil encontramos:

Originalidad e interacción frente a procesos y herramientas.
Software en producción frente a documentación exhaustiva.
Colaboración con el cliente frente a negociación de contratos.
Respuesta ante los cambios frente a estricto seguimiento de un plan.

Como vemos, se trata de toda una declaración de intenciones. Las metodologías ágiles, principalmente, pretenden ser flexibles como respuesta a la aparente dificultad al cambio de las metodologías tradicionales.

En una metodología ágil deberíamos encontrar, entre otros, los siguientes atributos:

Velocidad: Desarrollos más rápidos, siempre en comparación con las metodologías tradicionales.
Agilidad: Capacidad de improvisar y desarrollar nuevas soluciones al vuelo.
Adaptabilidad: Desarrollos dinámicos, capaces de reaccionar entre entornos cambiantes.
Ingenio: Capaz de tomar decisiones meditadas o bajo cierta disciplina.

En definitiva las metodologías tradicionales aportaban la organización y el orden del que carecía de desarrollo de software. Estas metodologías pretenden mejorar la calidad del producto desarrollado.

[4] Kent Beck y otros, Manifesto for Agile software development, http://agilemanifesto.org, 2001.

Por su parte, loas metodologías ágiles pretenden flexibilizar el proceso (sin que por ellos se resienta la calidad del producto final). Un factor clave es la mejora de la comunicación entre los diversos actores del proyecto, desde el propio equipo de trabajo hasta el cliente final.

4.1.3. Metodologías tradicionales

El término ingeniería del software fue utilizado por primera vez a finales de los años 50 y principios de los 60. Podemos encontrar sus orígenes en dos conferencias patrocinadas por la OTAN en 1968 y 1969 en las que se sentaron las bases de lo que hoy conocemos como ingeniería del software.

Muchos proyectos fracasaban entonces (y, aunque en menor grado, siguen fracasando hoy en día) por incumplimiento de plazos, subestimación de presupuestos, o motivos similares. A esto se le denominó como la Crisis del Software, uno de los desencadenantes del interés en esta disciplina por parte de la industria. Hay quien dice que esta crisis, a pesar de todos los avances, continúa en la actualidad, e incluso hay quien manifiesta que la crisis es crónica[5].

El software, un producto intangible y ubicado en un mundo virtual, mostró que podía ocasionar grandes pérdidas materiales por culpa de fallos en sistemas de gran importancia. Eso sin contar con la consiguiente pérdida de prestigio por parte de la empresa que sufría sus consecuencias. En OS/360 de IBM, por ejemplo, fue un sistema en el que llegaron a trabajar más de 1000 programadores, y suele utilizarse como un ejemplo clásico de mala planificación. No obstante, este proyecto fue el primero en su categoría, y de sus errores se extrajeron múltiples enseñanzas. Así el proyecto del OS/370 fue un éxito rotundo. Pero lo más grave es que no estamos hablando sólo de daños materiales. Sistemas embebidos responsables de aparatos de radioterapia provocaron la desgracia pérdida de vidas humanas a causa de dosis letales de radiación.

Ágiles	Tradicionales
Basadas en heurísticas provinentes de prácticas de producción de código.	Basadas en normas de estándares.
Preparadas para cambios en cualquier momento del proyecto.	Relativamente resistentes a cambios.
Bajo grado de control.	Alto grado de control, con muchas políticas y normas.
Si contrato tradicional.	Sujetas a acuerdos contractuales.
El cliente forma parte del equipo de desarrollo.	Comunicación con el cliente mediante reuniones.
Grupos de trabajo pequeños.	Grupos de trabajo muy grandes y con posibilidad de estar distribuidos.
Poco énfasis en la arquitectura.	La arquitectura es esencial.

Tabla *4.1.3.1.* Comparativa entre metodologías ágiles y tradicionales

[5] Tom Gilb, Software´s Chronic Crisis, IEEE Software.

Estas técnicas, métodos y herramientas comenzaron a agruparse en especificaciones sistemáticas del proceso de desarrollo del software, intentando reunir las mejores prácticas de este proceso de una manera coherente y unificada. De este modo, nacieron las metodologías de software propiamente dichas.

En la primera fase (años 70 y 80) las metodologías siguieron el paradigma del Análisis y Diseño Estructurado. El pionero fue Tom DeMarco, que con su SA/SD definió la que podría considerarse como la primera metodología importante. Siguiendo su estela surgieron muchas otras, como la de Gane/Sarson, la de Yourdon/Constantine, la de Ward/Mellos o el YSM del propio Yourdon.

En este momento los gobiernos reconocen la importancia de los desarrollos informáticos dentro de su funcionamiento, y deciden proponer, ellos mismos, unas metodologías sistemáticas basada en éstas, de uso obligado para todo contratista del gobierno. Las más célebres de esta época son la SSADM inglesa, la MERISE francesa, la Dafne italiana o el Euromethod europeo, que aglutinaba a estas anteriores y a otras más, y que fue la base tanto del estándar METRICA español como el PRINCE inglés.

A principios de los 90, la pujanza de las tecnologías Orientadas a Objetos provocó un cambio de paradigma que llevó a la definición de toda una nueva serie de metodologías. Entre las grandes metodologías Orientadas a Objetos de esta época se pueden destacar la de Shalaer/Mellor, considerada la primera metodología de este tipo, la de Coad/Yourdon, la Fusión de Coleman, el marco OPEN de Henderson/Sellers y Younessi, o la famosísima terna formada por el método Booch´93 de Booch, la OOSE de Jacobson y la OMT de Rumbaugh.

Precisamente la fusión de estas tres últimas nacen tanto el lenguaje UML como el Proceso Unificado, en cualquiera de sus diversas variantes.

Probablemente resulte necesario señalar que el uso de la expresión "metodologías tradicionales" es relativo, esto es, les damos este nombre por contraposición a las metodologías ágiles, señalando que el enfoque ágil es relativamente reciente; en cambio el enfoque de la demás se enmarca en una tradición que nos lleva hasta los mismos orígenes de la ingeniería del software.

Pero ha de quedar claro que el hecho de que el enfoque sea tradicional sólo indica que se continúa con una tradición, no que se trate de un enfoque "antiguo". De hecho, cuando en este contexto hablamos de metodologías tradicionales, no estamos pensando en los clásicos ejemplos del paradigma estructurado como SA/SD, MERISE o YSM.

Generalmente estamos hablando de las grandes metodologías Orientadas a Objetos, y en particular del Proceso Unificado, incluso más que de otras metodologías anteriores como OMT.

Las metodologías tradicionales fueron concebidas como metodologías universales. Esto es aún más cierto si nos referimos al proceso unificado; en tal caso, se las supone adecuadas para todo proceso de desarrollo. Sin embargo la aparición de las metodologías ágiles demuestra que puede haber circunstancias para las que no sean

adecuadas. Por lo tanto cabe preguntarse, ¿cuáles son estas circunstancias?, o mejor planteada la pregunta, ¿cuándo conviene utilizar metodologías ágiles en lugar de metodologías tradicionales?

Una metodología ágil es ante todo una forma de pensar; si se permite adaptar el plan de proyecto de manera flexible y rápida dejando un amplio margen de maniobra. Por tanto, la conveniencia de utilizar una metodología ágil no depende tanto del tipo de proyecto o del dominio de desarrollo como de las circunstancias del mismo, una serie de factores que incluyen tanto el contexto externo como el equipo de desarrollo o los recursos disponibles. Se trata ante todo de una cuestión de prioridades.

4.2. DESARROLLO

Para la realización del presente trabajo y como en todo es necesario apoyase en fuentes de carácter documental, esto es, en documentos de cualquier especie que contengan información sobre el diseño, creación de interfaces gráficas de usuario, así como programación de las mismas en diferentes lenguajes que facilitará su realización, se tomo como referencia nuestras necesidades en cuanto el lenguaje, de primera instancia, de hardware y software, etc.

Durante el desarrollo se utilizo información que fue obtenida tanto de entrevistas, cuestionarios y observaciones. Siendo compatible desarrollar este tipo de investigación junto a la investigación de carácter documental.

Del conocimiento obtenido de las fuentes primarias se realizo una sistematización para lograr con ello un nuevo conocimiento.

El siguiente diagrama muestra las etapas de la metodología, que se amplía más a detalle durante la realización de este trabajo.

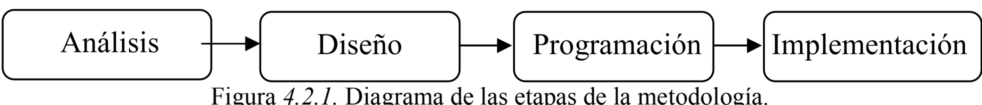


Figura *4.2.1.* Diagrama de las etapas de la metodología.

Para cubrir las deficiencias es indispensable que tanto los usuarios como las personas que participan en el desarrollo de los módulos se comuniquen durante las fases de producción de software (pre-producción, producción, post-producción) para alcanzar su objetivo. Estas personas son: el usuario –actor- y el desarrollador; a continuación veremos los roles y con ello la importancia de cada uno.

4.2.1 El rol del usuario

Durante la pre-producción el usuario es el encargado de definir las necesidades iniciales y la administración del proceso, en donde se seleccionan, estructuran, identifican, y se aportan los conocimientos, dudas y estrategias de las necesidades. La experiencia es adquirida a través del tiempo y conforme a las necesidades que se van requiriendo continuamente.

El usuario debe ser ante todo un bien facilitador de las necesidades y especificaciones de las mismas, una vez seleccionados y definidos dichas necesidades y modos de empleo se decide la manera adecuada e interesante para el desarrollo de nuestra interfaz. El usuario puede aportar estrategias que hagan de nuestra Interfaz algo atractivo e interesante, además de útil.

En la fase de **producción** el usuario participa activamente, interactuando con el desarrollador, ya que en muchas ocasiones el experto no sabe expresar sus necesidades, y el desarrollador tiene que interactuar muchas veces para concretar ideas, por lo que se debe de dar al usuario la opción de participar en la creación de sus propias necesidades.

En la fase de **post-producción** el usuario debe ser capaz de **validar** que los objetivos de uso planteados en las etapas anteriores se llevan a cabo de manera satisfactoria. Proporcionar sugerencias para mejorar el producto final.

4.2.2 El rol del desarrollador

En la fase de **pre-producción** el desarrollador es el encargado de realizar metodologías de análisis, y diseño.

En esta etapa se especifica la función del software, así como la descripción de la interfaz y el establecimiento de las restricciones (definir alcances del sistema) de diseño que debe considerar el software.

El desarrollador debe establecer contacto con el usuario para establecer las características de la interfaz del sistema y descubrir las restricciones de diseño. Asimismo, debe evaluar el flujo y la estructura de la información, definir y elaborar las funciones del software, entender cual será el comportamiento del programa.

Tanto el usuario como el desarrollador identifican las necesidades del sistema reconociendo la importancia de cada paso y punto en algún proceso, ejemplos, y situaciones provocadoras que le permiten al desarrollador la comprensión del problema planteado.

Confrontar los conocimientos previos (visión del mundo) con respecto a los conceptos que se presentan, de tal manera que sea capaz de reflejar y probar el nuevo conocimiento adquirido.

En la fase de **producción** el desarrollador genera la programación y el desarrollo de software a través de una especificación, metodologías de análisis y desarrollo de software y técnicas de pruebas.

En la fase de **post-producción** el desarrollador es el encargado de presentar al usuario el producto final con el objetivo de recabar sugerencias que le permitan mejorar el software educativo, de manera que satisfaga las necesidades reales. Además, en esta fase es posible descubrir errores y deficiencias que pudiera presentar dicho sistema. Asimismo validar que los objetivos planteados se cubran satisfactoriamente.

4.2.3 Relación bipartita en el desarrollo de la interfaz

Por lo anterior al hacer referencia al desarrollo de la interfaz, es evidente la necesidad de contar con personas altamente capacitadas, tanto en el desarrollo de interfases como de problemas numéricos como los tratados.

El proceso de desarrollo siempre existirá una gran dependencia entre el usuario y el desarrollador. Esta dependencia se presenta en la siguiente figura 4.2.3.1.

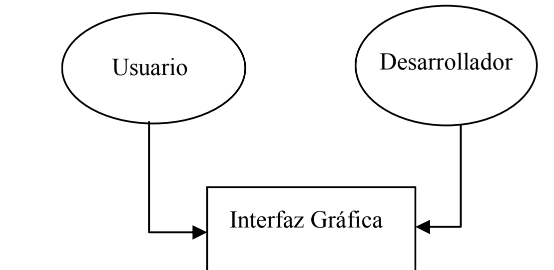


Figura *4.2.3.1* Dependencia entre el usuario y el desarrollador(es).

En la figura 4.2.3.1 se ve la dependencia entre el usuario y el desarrollador. El problema de este enfoque es: El tiempo de producción es muy lento por la excesiva interactividad entre el desarrollador y el usuario, pues no se logran concretar los objetivos rápidamente.

Con base en lo anterior, es necesario buscar una ruta alternativa donde queden satisfechas ambas partes para lograr así un objetivo común y satisfactorio; se hace uso UML, para especificar cada proceso y cada necesidad el usuario, llegando a un acuerdo común, así como lograr un entendimiento de cada proceso.

4.3. ANÁLISIS DEL PROBLEMA

Empezaremos el desarrollo del sistema con el análisis, siendo una de las etapas más críticas, pues será la base de toda recopilación de la información que dará forma a nuestro sistema.

Se busca obtener las necesidades del usuario que serán traducidas en requerimientos de software, estos requerimientos transformados en diseño y el diseño implementado en código, el código es probado, documentado y certificado para su uso operativo. Define quién está haciendo qué, cuándo hacerlo y cómo alcanzar un cierto objetivo[6]

[6] Booch G., Rumbaugh J., Jacoboson I. “El proceso Unificado de Desarrollo de Software”, Adison Wesley, Madrid, 2000.

Apoyados en los casos de uso –ver Apéndice E- tendremos un conocimiento total de los casos que se pueden presentar, los procesos que se van a llevar acabo en cada una de las áreas estudiadas.

En lo que respecta al factor de análisis de nuestro problema, se realizó principalmente con cuestionamientos al usuario partiendo sobre sus necesidades generales y compromisos del programador, pero sin olvidar el análisis de las herramientas necesarias; se llevo acabo un plan de trabajo con la finalidad de desarrollar el sistema buscando no desviarse del objetivo.

Partiendo con el análisis de necesidades del problema se comenzó con la investigación de herramientas –ventajas, desventajas-, así como de nuestro tema en particular que es el diseño, que nos fueran de utilidad en la realización del sistema de la manera mas objetiva, se continuo realizando casos de uso que nos apoyaran en la comprensión y especificación del problema –comprensión de las reglas del negocio del sistema- entre el usuario y el programador para obtener las especificaciones del problema mucho mas claras y comprensible.

El sistema esta enfocado para usuarios con conocimientos de problemas físicos y con necesidades de solución de los mismos. Por lo que nuestro universo de usuarios es variado en cuanto el conocimiento de sistemas computacionales, ya que pueden o no saber utilizar una herramienta de computo, por lo que nuestra interfaz debe contemplar este punto necesariamente.

Por esta razón, el objetivo principal de este trabajo es contribuir a los esfuerzos que se realizaron, de tal modo que se cumpla de una manera adecuada con las necesidades y satisfacción del usuario.

Durante toda la realización y como regla en el diseño de Interfaces así como en todos los sistemas, las necesidades del usuario será nuestro punto de partida.

4.3.1 Planteamiento del problema

4.3.1.1 Modelo de Requisitos

Es importante siempre describir en forma clara el problema a resolver. Aquí se plantean los objetivos y metas que se desean alcanzar durante el desarrollo del sistema.

Problema: Realizar una Interfaz Gráfica de Usuario que se comunica con un sistemas que simula numéricamente las ecuaciones de balance de diferentes tipos de flujos en un régimen laminar y turbulento.

Objetivo: Desarrollar una Interfaz Gráfica de Usuario amigable, agradable y eficiente para simplificar el uso de un sistema que simula numéricamente las ecuaciones de balance de diferentes tipos de flujos.

Descripción del problema: Con el objetivo de limitar el alcance del problema, nos referimos a la solución de problemas de convección natural en prismas rectangulares. Entonces, la interfaz deberá proporcionar al usuario una herramienta para resolver este tipo de flujos. El simulador resuelve estos problemas de manera numérica,

en donde se describe el dominio de solución mediante una malla cartesiana. El usuario contará con espacios para colocar información sobre ciertos parámetros de acuerdo al problema que esté resolviendo; podrá seleccionar entre diferentes métodos o ecuaciones matemáticas; tendrá una sección de visualización de resultados. Todos los pasos que debe seguir el usuario para resolver su problema deberán ser simples y fáciles de recordar.

El sistema contará con una sección donde el usuario podrá adecuar su interfaz cambiando colores, distribución de las ventanas y menús, de manera que se sienta cómodo y considere que le será aún más sencillo recordar -durante el desarrollo y por cuestiones de tiempo y objetivos de esta primera parte, lo antes mencionado se propone como una extensión o mejora del sistema-.

Está GUI deberá estar compuesta por *widgets* que le permitirán al usuario interactuar con el computador para obtener resultados favorables y útiles en la solución de sus simulaciones y de la visualización de los resultados. Se requiere que tenga la mayor interacción posible, que pueda personalizarse -como segunda versión-, que ofrezca un conjunto de valores razonables por omisión -como segunda versión-, que sea una interfaz sencilla (es decir que no distraiga la atención del usuario innecesariamente), con widgets significativas y relacionadas con el uso que tengan (uso de metáforas) y una retroalimentación constante.

Informe de Investigación Preliminar: Debemos definir antes que nada los limites del sistema, sus funciones que realiza y requisitos iniciales.

De primera instancia nuestra interfaz deberá comunicarse con el sistema de simulación numérica de una manera limpia –modular-, ver tema Comunicación entre la GUI y el simulador; los resultados deberán ser visualizados en OpenDX –ver Herramientas de desarrollo y Visualización de resultados a través de la GUI-, ya que los resultados que arroja el simulador son en archivos que puede leer e interpretar este software.

Los requisitos que no debemos hacer a un lado y con los que nos regiremos son con buscar la innovación, sencillez, buena interacción, una retroalimentación al usuario, visualización de resultados, y como ampliación la personalización.

4.3.1.2 Lista de procesos

Los procesos obtenidos por las sesiones y estudiados para la automatización, desglosan la inserción de datos con condiciones iniciales a considerar como puntos importantes la tolerancia e interacción; la tolerancia nos ayudara a definir puntos limites de estudio de nuestro problema a nivel de especificación, el número de interacciones es para poder tener un grado de exactitud para la solución, así como la validación de campos solo numéricos con limites de caracteres; elección de ecuación que son las diferentes opciones que nos proporciona el simulador, con opción a poder modificar en cualquier punto de la inserción de datos; y como condición necesaria la separación del proceso de compilación que genera el simulador con la muestra visual de nuestra interfaz. Observar la figura 4.3.1.2.1 para la compresión de este punto.

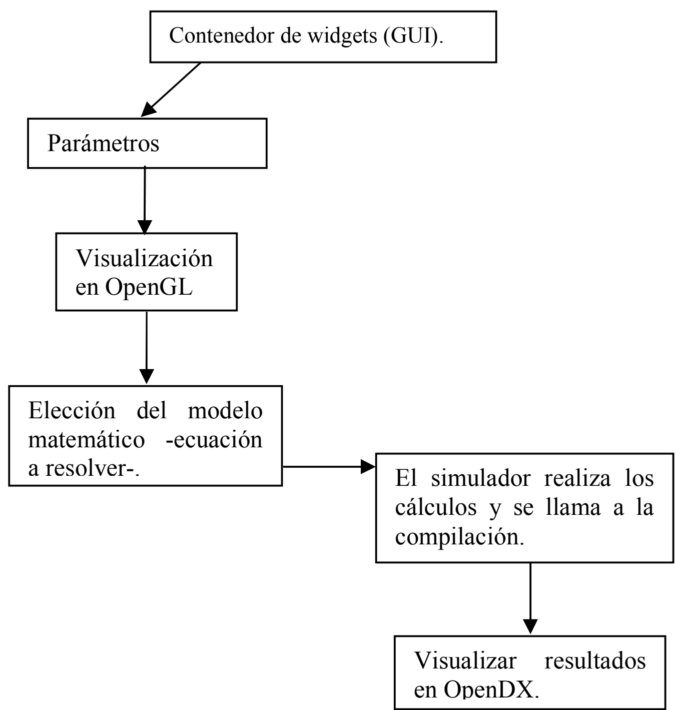


Figura *4.3.1.2.1*. Proceso de desarrollo ideado para construir la GUI.

4.3.2 Diagrama de Contexto

El diagrama de contexto tiene una gran importancia puesto que muestra las diversas entidades que van interactuar –ver figura 4.3.2.1-.

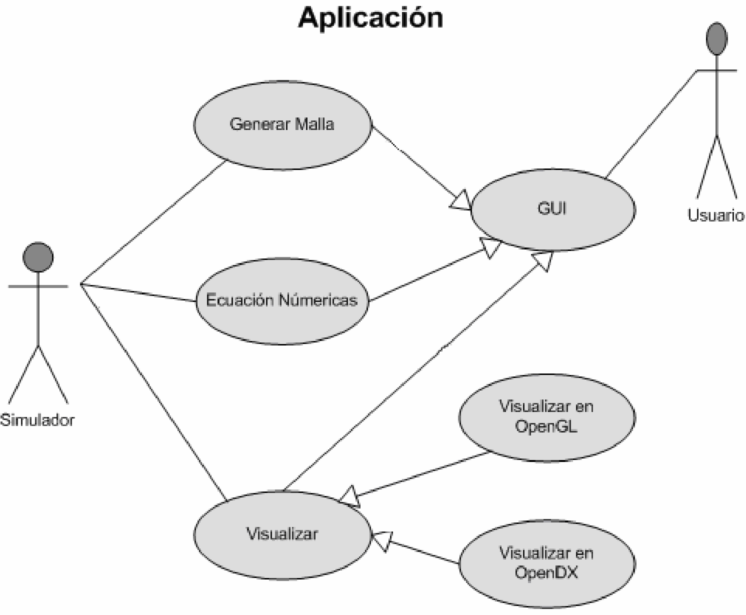


Figura *4.3.2.1* Diagrama de Contexto.

4.3.3 Diagramas de casos de uso

Es una representación parte o total de los actores y casos de uso del sistema, incluyendo sus interacciones y su funcionalidad principal. Un diagrama de casos de uso muestra, por tanto, los distintos requisitos funcionales que se esperan de una aplicación o sistema y cómo se relaciona con usuarios u otras aplicaciones. Los casos de uso, son problemas particulares, los cuales se complican conforme se avanza en la construcción. Para completar este apartado ver el -Apéndice E-.

Un actor es una entidad que utiliza alguno de los casos de uso del sistema. Se representa mediante un símbolo. Un actor en un caso de uso representa un rol que alguien o algo podría desempeñar.

Actor	Descripción
Usuario	Es el encargado de proporcionar, ingresar al sistema sus parámetros iniciales para especificar las características de la malla, o delimitación del problema, así como elegir el modelo matemático con el que desea resolver su problema antes planteado.
Visualizador de inicialización	Es el encargado de obtener los parámetros iniciales y visualizarlos en forma de una malla utilizando propiedades de OpenGL.
Simulador	Es el encargado de obtener los valores iniciales así como el modelo matemático, para con ello realizar los cálculos numéricos. Dar como resultado un archivo con los datos obtenidos del cálculo, que será materia prima para el visualizador de resultados (OpenGL).
Visualizador de resultados	Es el encargado de mostrar el resultado de los cálculos de una manera gráfica por omisión y comprensible para el usuario, y que podrá manipular para visualizarlo según sus necesidades.

Tabla *4.3.3.1.* Descripción de los Actores.

4.4. HERRAMIENTAS DE DESARROLLO

4.4.1. Lenguaje Unificado de Modelado[7]

Para el análisis y desarrollo el UML, prescribe un conjunto de notaciones y diagramas estándar para modelar sistemas y describe la semántica esencial de lo que estos diagramas y símbolos significan. UML es un lenguaje gráfico para visualizar, especificar, construir y documentar distintos tipos de sistema de software, de hardware, y organizaciones del mundo real.

Lenguaje Unificado de Modelado (**UML**, por sus siglas en inglés, *Unified Modeling Language*) es el lenguaje de modelado de sistemas de software más conocido y utilizado en la actualidad; aún cuando todavía no es un estándar oficial, está apoyado en gran manera por el OMG (Object Management Group). UML ofrece un estándar para describir un "plano" del sistema (modelo), incluyendo aspectos conceptuales tales como procesos de negocios y funciones del sistema, aspectos concretos como expresiones de lenguajes de programación, esquemas de bases de datos y componentes de software reutilizables.

[7] Booch G., Rumbaugh J., Jacobson. “El Proceso Unificado del Desarrollo del Software”, Addison Wesley, Madrid, 2000 y http://es.wikipedia.org

El punto importante para notar aquí es que UML es un "lenguaje" para especificar y no un método o un proceso. UML se usa para definir un sistema de software; para detallar los artefactos en el sistema; para documentar y construir -es el lenguaje en el que está descrito el modelo-. UML se puede usar en una gran variedad de formas para soportar una metodología de desarrollo de software (tal como el Proceso Unificado de Rational) -pero no especifica en sí mismo qué metodología o proceso usar.

UML cuenta con varios tipos de diagramas, los cuales muestran diferentes aspectos de las entidades representadas.

- *Diagramas de Casos de Uso*: permite el modelado de una vista 'business' del escenario y con ello determinar qué objetos son necesarios para la implementación del escenario. Si tienes modelada la descripción de cada caso de uso como una secuencia de varios pasos, entonces puedes "caminar sobre" esos pasos para descubrir qué objetos son necesarios para que se puedan seguir los pasos.
- *Diagramas de Secuencia*: para modelar el paso de mensajes entre objetos. Es uno de los diagramas más efectivos para modelar interacción entre objetos en un sistema. Muestra la interacción de un conjunto de objetos en una aplicación a través del tiempo y se modela para cada caso de uso.
- *Diagramas de Colaboración*: para modelar interacciones entre objetos
- *Diagramas de Estado*: para modelar el comportamiento de los objetos en el sistema
- *Diagramas de Actividad*: para modelar el comportamiento de los Casos de Uso, objetos u operaciones.
- *Diagramas de Clases*: son utilizados durante el proceso de Análisis y Diseño de los sistemas informáticos, donde se crea el diseño conceptual de la información que se manejara en el sistema, los componentes que se encargaran del funcionamiento y la relación entre uno y otro.
- *Diagramas de Objetos*: para modelar la estructura estática de los objetos en el sistema. Los diagramas de objetos utilizan un subconjunto de los elementos de un diagrama de clase, y no muestran la multiplicidad ni los roles.
- *Diagramas de Componentes*: para modelar los componentes del sistema.
- *Diagramas de Implementación*: para modelar la distribución del sistema.

4.4.2. Hilos (*Threads*, en ingles)

4.4.2.1. Definición

Un hilo de ejecución, en Sistemas Operativos, es similar a un proceso en que ambos representan una secuencia simple de instrucciones ejecutada en paralelo con otras secuencias, dicho de otra manera un hilo es la traza de ejecución de un proceso, refiriéndose esto a la parte dinámica del proceso, la ejecución el código representada por la pila del usuario y la parte del bloque de control del proceso que hacer referencia al estado del procesador y del proceso, frente a la parte más estática, como es la del resto del bloque de control del proceso y el código mismo.

La aparición de los hilos viene justificada desde dos pilares básicos: facilitar la sincronización entre procesos y mejorar la eficiencia en la alternancia de procesos en el procesador. Los hilos permiten dividir un programa en dos o más tareas que corren

simultáneamente, por medio de la multiprogramación. Este método permite incrementar el rendimiento de un procesador de manera considerable.

Todos los hilos de un proceso comparten los recursos del proceso. Residen en el mismo espacio de direcciones y tienen acceso a los mismos datos. Cuando un hilo modifica un dato en la memoria, los otros hilos utilizan el resultado cuando acceden al dato. Cada hilo tiene su propio estado, su propio contador, su propia pila y su propia copia de los registros de la CPU. Los valores comunes se guardan en el bloque de control de proceso (PCB), y los valores propios en el bloque de control de hilo (TCB).

Un ejemplo es precisamente lo que realizamos, es utilizar hilos, teniendo un hilo atento a la interfaz gráfica (iconos, botones, ventanas), mientras otro hilo hace una larga operación internamente. De esta manera el programa responde más ágilmente a la interacción con el usuario.

Quizá alguno piensa que un nuestra aplicación bastaría con separar procesos, pero por razones de rendimiento, flexibilidad, crecimiento, poder manipular una u otra aplicación, y por ello que es importante mencionar la diferencia entre un hilo y un proceso.

Los hilos se distinguen de los tradicionales procesos en que los procesos son generalmente independientes, llevan bastante información de estados, e interactúan sólo a través de mecanismos de comunicación dados por el sistema. Por otra parte, muchos hilos generalmente comparten otros recursos directamente. En sistemas operativos que proveen facilidades para los hilos, es más rápido cambiar de un hilo a otro dentro del mismo proceso, que cambiar de un proceso a otro. Este fenómeno se debe a que los hilos comparten datos y espacios de direcciones, mientras que los procesos al ser independientes no lo hacen. Al cambiar de un proceso a otro el sistema operativo (mediante el *dispatcher*[8]) genera lo que se conoce como *overhead*, que es tiempo desperdiciado por el procesador para realizar un cambio de modo (mode switch), en este caso pasar del estado de Running al estado de Waiting o Bloqueado y colocar el nuevo proceso en Running. En los hilos como pertenecen a un mismo proceso al realizar un cambio de hilo éste overhead es casi despreciable.

La gran funcionalidad de los hilos, y es otra de las razones por las que se eligieron; la continuación de esté proyecto deberá contemplar el manejo de los cálculos numérico, de mallas cuando la cantidad de puntos y longitud es demasiado grande, etc. que requieren de un tiempo de cálculo considerable, deberán hacerse por este medio; esta condición fue considerada desde el inicio del simulador, ya que fue diseñado pensando en que funcionara en diferentes procesadores con el uso de hilos.

Al igual que los procesos, los hilos poseen un estado de ejecución y pueden sincronizarse entre ellos para evitar problemas de compartimiento de recursos. Generalmente, cada hilo tiene una tarea específica y determinada, como forma de aumentar la eficiencia del uso del procesador.

4.4.2.2. Estados de un hilo

[8] Un dispatcher o gestionador a bajo nivel de la CPU conmuta los procesadores centrales entre los distintos procesos.

Los principales estados de ejecución de los hilos son: Ejecución, Listo y Bloqueado. No tiene sentido asociar estados de suspensión de hilos ya que es un concepto de proceso. En todo caso, si un proceso está expulsado de la memoria principal (RAM), todos sus hilos deberán estarlo ya que todos comparten el espacio de direcciones del proceso.

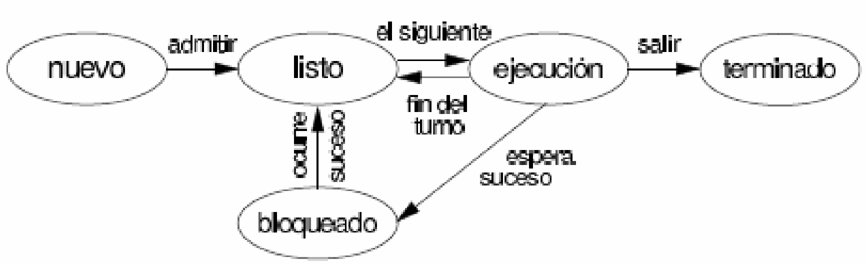


Figura *44.2.2.1.*Estados de un hilo.

4.4.2.3. Cambios de estados

Creación: Cuando se crea un proceso se crea un hilo para ese proceso. Luego, este hilo puede crear otros hilos dentro del mismo proceso. El hilo tendrá su propio contexto y su propio espacio de pila, y pasara a la cola de listos.

Bloqueo: Cuando un hilo necesita esperar por un suceso, se bloquea (salvando sus registros). Ahora el procesador podrá pasar a ejecutar otro hilo que esté en la cola de Listos mientras el anterior permanece bloqueado.

Desbloqueo: Cuando el suceso por el que el hilo se bloqueó se produce, el mismo pasa a la cola de Listos.
Terminación: Cuando un hilo finaliza se liberan tanto su contexto como sus pilas.

4.4.2.3.1. Implementaciones

Hay dos grandes categorías en la implementación de hilos:

- Hilos a nivel de usuario
- Hilos a nivel de Kernel

También conocidos como **ULT** (*User Level Thread*) y **KLT** (*Kernel Level Thread*).

Hilos a nivel de usuario (ULT): En una aplicación ULT pura, todo el trabajo de gestión de hilos lo realiza la aplicación y el núcleo o kernel no es consciente de la existencia de hilos. Es posible programar una aplicación como multihilo mediante una biblioteca de hilos. La misma contiene el código para crear y destruir hilos, intercambiar mensajes y datos entre hilos, para planificar la ejecución de hilos y para salvar y restaurar el contexto de los hilos.

Todas las operaciones descritas se llevan a cabo en el espacio de usuario de un mismo proceso. El kernel continua planificando el proceso como una unidad y asignándole un único estado (Listo, bloqueado, etc.).

Hilos a nivel de núcleo (KLT): En una aplicación KLT pura, todo el trabajo de gestión de hilos lo realiza el kernel. En el área de la aplicación no hay código de gestión de hilos, únicamente un API (interfaz de programas de aplicación) para la gestión de hilos en el núcleo. Windows 2000, Linux y OS/2 utilizan este método.

Combinaciones ULT y KLT: Algunos sistemas operativos ofrecen la combinación de ULT y KLT, como Solaris. La creación de hilos, así como la mayor parte de la planificación y sincronización de los hilos de una aplicación se realiza por completo en el espacio de usuario. Los múltiples ULT de una sola aplicación se asocian con varios KLT. El programador puede ajustar el número de KLT para cada aplicación y máquina para obtener el mejor resultado global.

¿Qué es un thread (hilo de control o simplemente hilo)? Un hilo -algunas veces llamado *contexto de ejecución* o *proceso ligero*- es un flujo de control secuencial dentro de un programa. Un *único* hilo es similar a un programa secuencial; es decir, tiene un comienzo, una secuencia y un final, además en cualquier momento durante la ejecución existe un sólo punto de ejecución. Sin embargo, un hilo no es un programa; no puede correr por sí mismo, corre *dentro* de un programa. Un hilo por si mismo no nos ofrece nada nuevo. Es la habilidad de ejecutar varios hilos dentro de un programa lo que ofrece algo nuevo y útil; ya que cada uno de estos hilos puede ejecutar tareas distintas.

4.4.2.4. Planificación de hilos

Para solucionar el problema de comparición de recursos se puede asignar distintas prioridades a los hilos. Se puede modificar la prioridad de un hilo en cualquier momento después de su creación usando el método: setPriority() getPriority().

La prioridad de un hilo tiene el efecto de:

- Si hay 2 hilos para ejecutarse, se ejecuta primero el de mayor prioridad.
- Si hay 1 hilo ejecutándose y entra un hilo de mayor prioridad, el primer hilo sale de ejecución y entra este ultimo.

Hilos demonio: Son hilos de muy baja prioridad (llamados servicios) que normalmente se ejecutan en periodos muy largos usando pocos recurso. Para crear un demonio se usa setDaemon():true o false.

Clase Thread: Maneja hilos de ejecución. Hay métodos que controlan si el hilo esta ejecutado, durmiendo, en suspenso o detenido.

Métodos básicos:

- Sleep(milis): Pone en suspenso un hilo en ejecución durante un cierto tiempo.
- start(): Pone a ejecutar un hilo. Crea un hilo de sistema y ejecuta. Luego llama al metodo run().
- run(): Lleva el cuerpo del hilo. Es llamado por el método start().

Normalmente es un bucle. Hay que redefinirlo.
- suspend(): Detiene el hilo, pero no lo destruye. Puede ejecutarse de nuevo. Met.de instancia.
- resume(): Reanuda el hilo de ejecución detenido. Met.de instancia.
- interrupt(): Detiene el hilo de ejecución, normalmente no se usa, se deja que el hilo termine su correcta ejecución.
- join(): Fuerza al hilo a esperar la finalización de los hilos asociados.

Sincronización: Cuando dos hilos necesitan usar el mismo objeto aparecen las operaciones entrelazadas que pueden corromper (dañar) los datos.

La solución es sincronizar el acceso a esa región. Cuando hay múltiples hilos que pueden acceder a una misma región se utiliza el "bloqueo" de la misma, que consiste en darle el acceso y control total solo a un hilo.

El bloqueo comienza cuando se ejecuta el método (synchronized) y finaliza cuando termina el método ya sea por return o por una terminación anormal (excepciones).

Comunicación entre hilos: Aparte de la sincronización, se usa la comunicación entre hilos para solucionar los problemas de comparación de datos.

Los hilos se notifican entre si la finalización o comienzo de las acciones.

- Métodos utilizados para la comunicación:
  - Wait(): Deja al hilo esperando hasta que ocurra alguna condición. Hasta que otro hilo le notifique que ocurrió algo.
  - Notify(): Notifica al hilo que espera que hay algún cambio que podría satisfacer esa condición.
  - Notifyall(): Notifica a todos los hilos en espera (wait).

- Modelo productor – consumidor: Un hilo produce una salida que otro hilo usa o consume.

En la programación secuencial no hay problema ya que si el productor se ejecuta primero, luego el consumidor tiene lo que el productor le envía.

En la programación concurrente si hay problema ya que los dos corren al mismo tiempo ¿cuál es el que corre primero? ¿el productor o el consumidor?

Si el productor corre mas rápido que el consumidor se pueden perder datos, ya que el consumidor no los puede tomar a la misma velocidad.

Si el consumidor es mas rápido querrá tomar los datos que el productor todavía no ha producido.
La solución a este problema es contar con una clase que controle y supervise y sincronice al productor y consumidor, esta clase la denominaremos "monitor".

Finalización de hilos:

- Que el método run() retorne normalmente
- Que el método run() finalice bruscamente
- Que el método destroy() se invoque.
- Que el programa termine normalmente.

Hace algún tiempo la única forma de realizar varias tareas a la vez era usar varios procesos simultáneos, pero esto tiene una desventaja a la vez que una ventaja. Los procesos son completamente independientes entre si y uno de ellos no puede acceder a la memoria usada por otro. El sistema operativo intercambia todos los punteros al ceder el control a un proceso u otro. Todo este intercambio, aunque hace que un proceso un pueda interferir a otro consume recursos. Por otra parte dificulta la comunicación entre varios procesos distintos.

QT facilita muchas cosas. En el modelo de hilos de QT simplemente tienes que heredar de QThread y redefinir el método run(). El hilo se inicia con start().

```
connect (MiBoton, SIGNAL(released()), this, SLOT(empezarHilo()))
void empezarHilo(){
      miSuperHilo.start(); }
```

4.4.2.5. Ciclo de Vida de una Hilo

Cada hilo, después de su creación y antes de su destrucción, estará en uno de cuatro estados: creación, "corrible", bloqueada, o muerta.

Creación (*New thread*): entra aquí inmediatamente después de su creación. Es decir luego del llamado a new. En este estado los datos locales son ubicados e iniciados. Luego de la invocación a start(), el hilo pasa al estado "corrible".

Corrible (*Runnable*): aquí el contexto de ejecución existe y el hilo puede ocupar la CPU en cualquier momento. Este estado puede subdividirse en dos: Corriendo y encolado. La transición entre estos dos estados es manejado por el itinerador de la máquina virtual. Nota: Un hilo que invoca al método () voluntariamente se mueve así mismo al estado encolado desde el estado corriendo.

Bloqueada (*not Runnable*): se ingresa cuando: se invoca suspend(), el hilo invoca el método wait() de algún objeto, el hilo invoca sleep(), el hilo espera por alguna operación de I/O, o el hilo invoca join() de otro hilo para espera por su término. El hilo vuelve al estado corrible cuando el evento por que espera ocurre.

Muerta (*Dead*): se llega a este estado cuando el hilo termia su ejecución (concluye el método run) o es detenida por otro hilo llamando al su método stop(). Esta última acción no es recomendada.

4.4.2.6. Ventaja de los hilos contra los procesos

Si bien los hilos son generados a partir de la creación de un proceso, podemos decir que un proceso es un hilo de ejecución, conocido como Monohilo. Pero las ventajas de los hilos se dan cuando hablamos de Multihilos, que es cuando un proceso tiene múltiples hilos de ejecución los cuales realizan actividades distintas, que pueden o no ser cooperativas entre sí. Los beneficios de los hilos se derivan de las implicaciones de rendimiento.

Se tarda mucho menos tiempo en crear un hilo nuevo en un proceso existente que en crear un proceso. Algunas investigaciones llevan al resultado que esto es así en un factor de 10.

Se tarda mucho menos en terminar un hilo que un proceso, ya que cuando se elimina un proceso se debe eliminar el PCB del mismo, mientras que un hilo se elimina su contexto y pila.

Los hilos aumentan la eficiencia de la comunicación entre programas en ejecución. En la mayoría de los sistemas en la comunicación entre procesos debe intervenir el núcleo para ofrecer protección de los recursos y realizar la comunicación misma. En cambio, entre hilos pueden comunicarse entre sí sin la invocación al núcleo. Por lo tanto, si hay una aplicación que debe implementarse como un conjunto de unidades de ejecución relacionadas, es más eficiente hacerlo con una colección de hilos que con una colección de procesos separados.

4.4.2.7. Planificación de hilos

Para solucionar el problema de comparición de recursos se puede asignar distintas prioridades a los hilos. Se puede modificar la prioridad de un hilo en cualquier momento después de su creación usando el método: setPriority() getPriority().

La prioridad de un hilo tiene el efecto de: Si hay 2 hilos para ejecutarse, se ejecuta primero el de mayor prioridad. Si hay 1 hilo ejecutándose y entra un hilo de mayor prioridad, el primer hilo sale de ejecución y entra este ultimo.

4.4.3. Visualización de resultados a través de la GUI

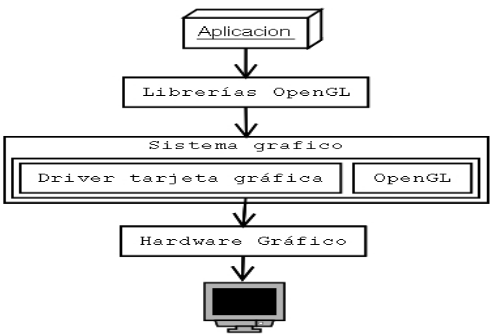
¿Qué es la visualización científica? es la generación de imágenes a partir de datos, para transformarlos en información y ganar entendimiento; así como para adecuar las limitaciones ópticas del ser humano y para comunicarlo a otros; como definición es el proceso de mapear valores numéricos en dimensiones conceptuales[9] permite explorar simulaciones y cálculos a través de imágenes y geometrías para ver incluso aquello que no es visible y así descubrir relaciones y crear conocimiento, fin último de los cálculos científicos.

[9] B.E. Rogowitz, IEEE Visualization, 1993.

La visualización de los resultados dentro de nuestra Interfase será mediante OpenGL y OpenDX, ya que te ofrecen bastantes herramientas para su implementación y posibilidades de crecimiento:

4.4.3.1. OpenGL

OpenGL es un Interfaz de programación de aplicaciones (API) estándar desarrollada por Silicon Graphics Inc. (SGI) en 1992 como evolución de la antigua Iris GL, en pro de hacer un estándar en la representación 3D gratuito y con código abierto (open source). Esta basado en sus propios OS y lenguajes IRIS, de forma que es perfectamente portable a otros lenguajes. Entre ellos C (originalmente desarrolladas con este lenguaje por ser estándar), C++, C++, Python, Perl, etc. y las librerías dinámicas permiten usarlo sin problema en Visual Basic, Visual Fortran, Java, etc.



En principio Silicon Graphics desarrollo una librería para sus estaciones graficas Iris llamada "Iris GL". Estas máquinas disponen de un hardware especialmente optimizado para visualización de gráficos, transformadas de matrices, soporte para Z-Buffer, etc. Con el uso de esta librería conseguían la independencia del hardware entre sus distintas estaciones Irix.

Al tratarse de una tecnología abierta, su especificación no debe estar controlada por un solo fabricante, sino que esta dirigida por un consorcio independiente, la plataforma de revisión de la arquitectura OpenGL (OpenGL Architecture Review Board) cuyos miembros fundadores son SGI, Digital Equipament Corporation, IBM, Intel y Microsoft. Actualmente 3Dfx, 3DLabs, ATI, Evans & Sutherland, Hewlett-Packard, NVidia y Sun también son miembros.

Durante años OpenGL se ha consolidado como la librería por excelencia para desarrollar aplicaciones 2D y 3D con independencia de la plataforma o el hardware gráfico.

Una de las principales ventajas que aporta OpenGL es que se trata de un estándar industrial. Gracias a la OpenGL ARB es realmente una tecnología abierta, lo que supone una ventaja inestimable frente a otras tecnologías.

Por otra parte cuenta con más de nueve años de vida, por lo que existe una extensa base de conocimientos a su alrededor. Durante todo este tiempo se han producido cambios en la librería, pero OpenGL siempre asegura una compatibilidad

"marcha atrás". De esta forma, una aplicación que se desarrolló usando la primera implementación de la librería, compilaría y funcionaria con la ultima versión de la misma.

Gracias a la portabilidad de OpenGL nuestras aplicaciones podrán ejecutarse en una amplia variedad de arquitecturas y de soportes gráficos, sin que el resultado se vuelva inconsistente. Por ejemplo, nuestro código fuente va a ser el mismo para una máquina Sparc corriendo Linux, que para un PC corriendo Windows y usando una aceleradora gráfica. Este punto dota a las aplicaciones de una gran escalabilidad. Actualmente OpenGL esta disponible para una gran variedad de sistemas operativos, tales como Unix, Windows, Mac OS, BeOS, etc.

Debido a los cambios en el hardware gráfico, y a que OpenGL es básicamente un interfaz de abstracción del hardware, parece muy probable que se quede anticuado pronto, es decir, podríamos suponer que en cuanto aparezca una nueva prestación (por ejemplo en las aceleradores) OpenGL no sería capaz de proporcionar unas funciones al programador para que este pudiese hacer uso de estas nuevas características en sus programas. Pues bien, OpenGL se diseñó desde el principio para ser capaz de hacer frente a este problema, y gracias a sus mecanismos de extensión, las nuevas funcionalidades se pueden ir introduciendo sin problemas mientras que se respeta la compatibilidad con las versiones anteriores.

4.4.3.1.1. Características de OpenGL

OpenGL (ogl) es la interfaz software de hardware gráfico. Es un motor 3D cuyas rutinas están integradas en tarjetas gráficas 3D. Ogl posee todas las características necesarias para la representación mediante computadoras de escenas 3D modeladas con polígonos, desde el pintado más básico de triángulos, hasta el mapeado de texturas, iluminación o NURBS[10].

Ofrece:

- Primitivas geométricas básicas como: puntos, líneas, polígonos y *raster*: bitmap e imágenes.
- Splines que sirven para dibujar líneas curvas.
- Transformación de vista modelo, gracias a esto puedes trasladar, rotar y escalar objetos dentro de la escena y a su vez mover la cámara fácilmente.
- Trabajar con color, permitiéndote trabajar con modo RGBA (*Red Green Blue Alpha*) o utilizando modo indexado, donde los colores son seleccionados desde una paleta. Eliminación de líneas y superficies ocultas.
- Permite utilizar uno o dos buffer; el doble buffer es utilizado para eliminar el párpado de las animaciones.
- Mapeado de texturas algo vital para cualquier API gráfica 3D.
- *Antialiasing* permite el suavizado de los bordes de polígonos y líneas, realizándolo mediante el cambio de la intensidad de los píxeles adyacentes a la línea que procesamos.
- Luces que permiten estableces la fuente de la luz, su posición, su intensidad, color, etc.

[10] Que es NURBS

- Efectos atmosféricos como son la niebla o humo.
- Transparencia, entre otras más.

Ogl soporta hardware 3D, y es altamente recomendable poseer este tipo de hardware grafico. Si no se tiene disposición de el, las rutinas de representación correrán por software, en vez de hardware, decrementando en gran medida su velocidad.

Ogl es una maquina de estados. Cuando se activan o configuran varios estados de la maquina, sus efectos perduraran hasta que sean desactivados. Por ejemplo, si el color para pintar polígonos se pone a blanco, todos los polígonos se pintaran de este color hasta cambiar el estado de esa variable. Existen otros estados que funcionan como booleanos (on o off, 0 o 1). Estos se activa mediante las funciones glEnable y glDisable.

Ogl contiene un conjunto de poderosos pero primitivos comandos, a muy bajo nivel. Además la apertura de una ventana en el sistema grafico que utilicemos (win32, X11, etc.) donde pintar no entra en el ámbito de Ogl. Por eso las siguientes librerías[11] son muy utilizadas en la programación de aplicaciones de ogl:

OpenGL Utility Library (GLU): está librería incluye funciones como definición de un cilindro o un disco con un solo comando, contiene funciones para trabajar con splines y operaciones con matrices para tener una orientación especifica, subdivisión de polígonos, etc. Está es la implementada por QT.

GLX y WGL: GLX da soporte para maquinas que utilicen X Windows System, para inicializar una ventana, permite no sólo renderizar en la máquina local, sino también a través de una red. WGL seria el equivalente para sistemas Microsoft.

OpenGL Utility Toolkit (GLUT)[12]: esta librería es independiente de la librería Ogl de cada plataforma, esta librería no incluye funciones adicionales para Ogl, pero nos permite utilizar funciones para el tratamiento de ventanas, teclado y ratón.

Para trabajar con Ogl en cualquier plataforma, primero tenemos que inicializar una ventana (y esto es diferente entre Windos y Linux por ejemplo), GLUT nos permite crear ventanas y controlar la entrada independientemente de la plataforma utilizada, aunque también contiene comandos para crear conos, tazas de té, etc.

4.4.3.1.2. Las librerías de OpenGL

OpenGL está dividido en varias librerías, cada una de ellas con unas funcionalidades claras. Cada una de ellas dispone de un prefijo en cada uno de los nombres de sus funciones.

GL: En ella se encuentran las funciones básicas de OpenGL. En sistemas Unix tiene el nombre de libgl.*, y en sistemas Windows opengl32.lib. En prefijo es "gl".

[11] También existen otras librerías más específicas para el control de entrada, sonido, red, etc., por ejemplo: OpenAL (Audio), OpenNL (Red), OpenIN (Entrada), etc.

[12] Escrita por Mark Kilgard.

GLU: Esta siglas tienen el significado de Utilidades para OpenGL. En ella se encuentran una gran variedad de funciones que pueden simplificar mucho el desarrollo con OpenGL. Está escrita usando funciones de OpenGL, por que lo su porte entre distintas plataformas es prácticamente inmediato. El prefijo de esta librería es "glu".

GLUT: Esta librería sustituye a la antigua AUX. En este caso es altamente dependiente de la plataforma, ya que es la encargada de proveer un interfaz común de programación en temas como las ventanas, el uso del ratón, los eventos de teclado, etc. Gracias a ella cuando se desarrolla una aplicación OpenGL no hay que tener en cuenta detalles sobre el sistema.

El programador siempre va a tener la posibilidad de programar sin GLUT usando el interfaz de un sistema concreto, pero en principio no gana nada, y pierde toda la portabilidad de su programa. Normalmente no hay ningún problema en compilar un programa que use GLUT en GNU/Linux, Windows, BSD, etc. sin modificar ni una sola línea de código.

4.4.3.1.3. Primitivas de OpenGL

Dentro de las primitivas que permite utilizar OpenGL se encuentran las siguientes:

- GL_POINTS Dibuja puntos.
- GL_LINES Dibuja líneas no conectadas.
- GL_POLYGON Dibuja un polígono de *n* vértices donde *n* es como mínimo 3.
- GL_TRIANGLES Dibuja una serie de triángulos.
- GL_LINE_STRIP Dibuja una serie de líneas interconectadas.
- GL_LINE_LOOP Igual que el anterior pero el primer y último están interconectados.
- GL_QUADS Dibuja una serie de cuadriláteros.
- GL_QUAD_STRIP Dibuja cuadriláteros pegados unos con otros.
- GL_TRIANGLE_STRIP Dibuja triángulos pegados unos con otros (uno frente a otro).
- GL_TRIANGLE_FAN Dibuja triángulos pegados unos con otros en forma de abanico.

La información más importante acerca de los vértices son sus coordenadas, que se especifican con el comando *glVertex*()*. Por cada uno de los vértices se puede especificar un color, un vector normal y coordenadas de textura. Al querer definir un polígono, debemos hacerlo las llamadas a *glVertex*()* exclusivamente dentro de los limites que fijan las instrucciones: *glBegin()* y *glEnd()*. Para la asignación de colores usaremos el comando *glColor*()*. Es también necesario saber que un polígono tiene dos caras, se define la cara anterior como aquella para la cual los vértices van en el sentido contrario al de las manecillas del reloj; ejemplo caras del cubo:

```
glBegin(GL_POLYGON);
    glColor3f(0.0,0.0,0.0);
      glVertex3f(1.0,1.0,1.0);
```

```
        glVertex3f(1.0,1.0,-1.0);
        glVertex3f(-1.0,1.0,-1.0);
        glVertex3f(-1.0,1.0,1.0);
glEnd();
```

Existen más variables que controlan cosas como los modos de dibujo de los polígonos, el posicionamiento de las luces, y las propiedades materiales de los objetos que están siendo dibujados. Se activan y desactivan mediante llamadas a *glEnable()* y *glDisable(),* aunque todas las variables de estado tienen un valor predefinido.

Limpiando la pantalla. Las facetas de un objetos tridimensional, antes de comenzar a dibujar cualquier cosa en la pantalla, debemos primero limpiarla, en el caso de OpenGL, lo que vamos a hacer es limpiar los buffers principales.

Entre los diversos buffers que existen, debemos conocer principalmente dos de ellos: el buffer de color que contiene el valor del color de los pixeles que son desplegados, y el buffer de profundidad. Este último es muy importante pues contiene el valor en el eje Z de cada uno de los pixeles que hay en la pantalla, así que si se desea dibujar un nuevo objeto, el valor en el eje Z de sus puntos se compara con el que existe en el buffer de profundidad, y si sus puntos se hallan más próximos a la pantalla que los que existían anteriormente, entonces se dibujan los puntos del objeto, de lo contrario, esto no se realiza.

El "limpiado" de los buffers se realiza de maneras diferentes para cada uno de los mencionados, pues para el de color, basta hacer una llamada a *glClearColor()*, dando el color con el cual se desea limpiar la pantalla y después a *glClear()* para hacer el limpiado. El buffer de profundidad realmente no se "limpia" sino que más bien se llena con un valor de profundidad así que una vez que esto se realiza, si los siguientes objetos están igual o más cerca de la pantalla que los que se llenaron con el valor dado, siempre serán dibujados. Normalmente se llena con el valor máximo de profundidad. Para limpiar los buffers de una sola llamada se puede hacer lo siguiente:

```
glClearColor(0.0,0.0,0.0,0.0); /* El color de limpiado será cero */
glClearDepth(0.0);
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```

El empleo de dos buffers nos permite dibujar una imagen mientras se despliega la otra, así mientras una vista del objeto se muestra, la siguiente se calcula, y así al intercambiar las imágenes, tenemos la impresión de que estamos viendo una animación.

Visualizando: Una vez que hemos hablado de las primitivas gráficas, así como de los buffers en los cuales estas se presentarán, debemos estudiar la visualización que es el otro aspecto esencial a considerar. Para visualizar un objeto en 3 dimensiones, podemos hacer una analogía con la actividad de tomar una fotografía, pues básicamente se siguen los mismos pasos. En el primero se pone la cámara apuntando a la escena que queremos tomar, después posiblemente modifiquemos algunas cosas de la escena a ser fotografiada, luego ajustamos el acercamiento, y finalmente se escoge de qué tamaño se quiere la impresión final. Estos mismos pasos tienen nombres específicos en el campo de la visualización 3D, pero aquí se habla de transformaciones. Siguiendo el mismo orden que citamos anteriormente, vendrían siendo: la transformación de vista (viewing),

la de modelado (modeling), la de proyección (projection) y por último la del puerto de vista (viewport).

Las transformaciones de vista deben preceder las de modelado forzosamente, las demás pueden ser especificadas en cualquier momento, antes de que se haga el dibujo. Los vértices se componen de 4 coordenadas, estas son (x,y,z,w). Si la coordenada w es diferente de cero, entonces las coordenadas corresponden al punto tridimensional (x/w,y/w,z/w). La coordenada w se usa poco, y si no se especifica, toma el valor 1 por omisión.

Cada vértice de la escena es multiplicado por una matriz de tamaño 4x4 que se construye a partir de las transformaciones de vista, modelado y proyección. Las normales se ven afectadas automáticamente por esta operación. Las coordenadas de recortado crean un volumen de vista, en el cual sólo los objetos que se hallan en su interior se toman en cuenta. Y finalmente las coordenadas ya transformadas se convierten en coordenadas de ventana, el puerto de vista puede ser manipulado causando modificaciones a la imagen final tales como agrandamiento, reducción o estiramiento.

Vamos a estudiar cada una de estas transformaciones individualmente ya que es necesario aplicarlas correctamente para obtener una imagen aceptable en la pantalla. Es importante limpiar la matriz actual o de trabajo, antes de hacer modificaciones, de lo contrario pueden obtenerse resultados equivocados si la matriz de trabajo contiene operaciones realizadas anteriormente. Para esto se utiliza el comando glLoadIdentity().

Antes de realizar transformaciones, debemos especificar con cual de las matrices vamos a estar trabajando, para esto, se usa el comando *glMatrixMode()* con uno de los argumentos siguientes: *GL_PROJECTION* o bien *GL_MODELVIEW*, dependiendo de la que necesitemos.

La transformación de vista: Como hemos visto anteriormente esta transformación es análoga al posicionamiento y orientación de la cámara. Es importante notar que en OpenGL, la cámara así como los demás objetos de la escena aparecen en el origen, y la cámara apunta hacia el eje z negativo. Se pueden usar funciones como *glTranslate*()* o *glRotate*()* para hacer modificaciones. Esta transformación se aplica a la proyección, usaremos entonces: *glMatrixMode(GL_PROJECTION).*

La transformación de modelado: Esta transformación se utiliza para posicionar y orientar el modelo u objeto que vamos a visualizar, se puede rotar, trasladar o cambiar la escala del modelo (scaling), para las dos primeras operaciones se usan los comandos mencionados en la transformación de vista, para el cambio de escala se usa *glScale*().* Es necesario especificar que trabajamos con el modelo, de ahí que usaremos: *glMatrixMode(GL_MODELVIEW).*

La transformación de proyección: Esta transformación equivale a poner lentes diferentes a la cámara, que cambiarán el campo de vista (es diferente usar un telefoto que un gran angular) además de esto, la transformación determina de que forma se proyectarán los objetos en la pantalla, existen dos maneras de realizarlo, la primera es usando perspectiva que hace cambiar el tamaño de los objetos dependiendo de la distancia a la cual están con respecto al observador, para esto se usa el comando

glFrustum(). El otro tipo de proyección es ortográfica en la cual el tamaño de los objetos no cambia (esta proyección se usa comúnmente en aplicaciones CAD).

La transformación de puerto de vista: Esta transformación define el tamaño y la posición de la imagen final (como la impresión de una fotografía), se usa el comando *glViewPort()* y es necesario modificar el puerto de vista cada que la ventana se modifica.

Una vez que todas estas transformaciones son aplicadas a cada uno de los vértices, la imagen puede ser desplegada dentro de la ventana.

La sintaxis de los comandos de transformación es la siguiente:

glTranslate {fd} (TYPE x, TYPE y TYPE z);
glRotate {fd} (TYPE ángulo,TYPE x, TYPE y, TYPE z);
glScale {fd} (TYPE x, TYPE y, TYPE z);

En donde TYPE es float o int, dependiendo del comando que se escogió.

4.4.3.1.4. Funciones de CallBack

Hay cuatro funciones de callback de debemos conocer:

Display: Esta función es invocada cuando el sistema determina que el contenido de la ventana tiene que ser redibujado, por ejemplo cuando la ventana se abre, o arrastramos otra ventana por delante de esta. La función de glut encargada de registrar este evento es glutDisplayFunc(). La función que registramos no tiene ningún argumento de entrada.

Reshape: La ventada de la aplicación puede cambiar de tamaño, normalmente porque el usuario arrastre alguno de los bordes de la ventana con el ratón. La función que registra este evento es glutReshapeFunc(). Y la función que nosotros registremos para manejar el evento debe tener dos parámetros enteros. Cuando se ejecute esta función de redimensión estos dos parámetros contendrán los nuevos valores del tamaño de la ventana.

Mouse: Esta función va a manejar los eventos que sean producidos por el ratón. Por ejemplo, que uno de los botones se presione o se suelte. La función que registra este evento es glutMouseFunc(). Al igual que en el caso anterior, esta función, al ser invocada pasa argumentos a la función de callback. En este caso estos argumentos describen la posición del ratón y el evento que se acaba de producir.

Keyboard: Esta función es muy parecida a la anterior. En este caso en lugar del ratón se trata de un evento de teclado. La función de glut que registra el callback es glutKeyboardFunc(). Y como viene siendo normal, a la función de callback también se le pasa información sobre la tecla en la que se ha producido el evento. Como caso excepcional, al callback también se le suministra información sobre la posición del ratón en el momento que se pulsó la tecla.

Estas son las cuatro funciones de callback más importantes, y las que más se usan, pero glut incluye muchas otras. Si bien no se suelen usar mucho merece la pena conocer: Idle, Timer, Special y Joystick.

Por otra parte hay que comentar que no todos los callbacks están disponibles en todas las versiones de glut. Según esta ha ido avanzando se han introducido nuevas funciones. Por ejemplo glutJoystickFunc(), - esta función se añadió en la versión 4 de glut-. Si escribimos un programa que haga uso de ella, va a ser necesario que se ejecute siempre sobre glut 4 o superior.

4.4.3.1.5. El proceso de visualización en OpenGL

El proceso de obtener una imagen sintética de un modelo en un ordenador es similar al proceso de hacer una fotografía de un objeto real. Tal y como se muestra en la figura adjunta, los pasos que sería necesario realizar son los siguientes:

1. Fijar trípode y apuntar la cámara a la escena (Transformación de la vista).
2. Situar los objetos en su disposición adecuada (Transformación del modelo).
3. Elegir el "zoom" o la lente de la cámara (Proyección).
4. Determinar el tamaño final de la fotografía (Transformación del viewport).

Estos pasos se corresponden con el orden en que se especifican las transformaciones en el programa, y no necesariamente con el orden en que son realizadas las operaciones matemáticas asociadas. En la figura 4.4.3.1.5.1 se muestra el orden en que estas operaciones se llevan a cabo en el ordenador:

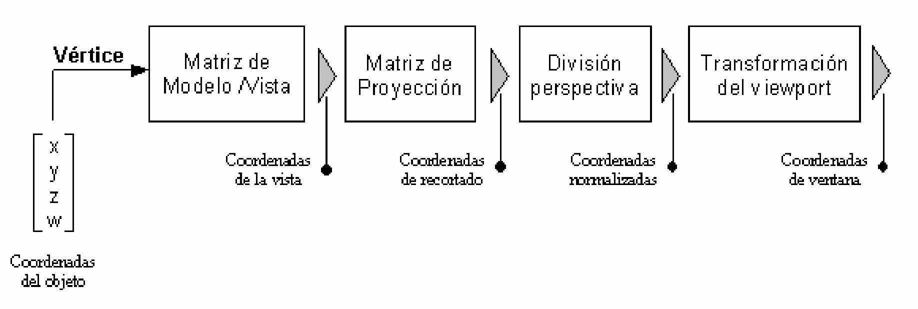


Figura *4.4.3.1.5.1*: Proceso de visualización.

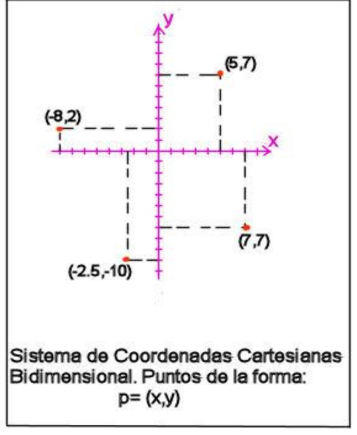
Las transformaciones del modelo, de la vista y la proyección se especifican mediante matrices 4x4 que multiplican a los vectores columna representativos de los vértices.

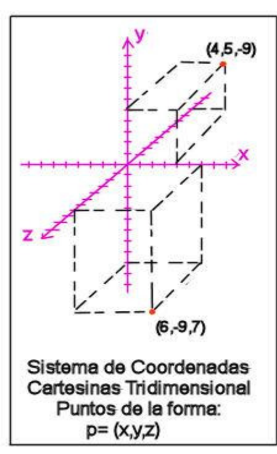
La transformación del modelo y de la vista especificadas se combinan en OpenGL para formar una única matriz denominada *matriz de Modelo / Vista*. Esta matriz se aplica a las coordenadas de los vértices para convertirlos al *sistema de coordenadas de la vista*. Posteriormente, OpenGL aplica la *matriz de proyección* para obtener las *coordenadas de recortado*. Esta transformación define un *volumen de la vista*; los objetos fuera de este volumen se recortan y no aparecerán en la imagen final.

Tras esta operación, se realiza la *división perspectiva* para producir *coordenadas normalizadas*, que finalmente se convierten a *coordenadas de ventana* aplicando la *transformación del viewport*.

4.4.3.1.6. Sistemas de coordenadas

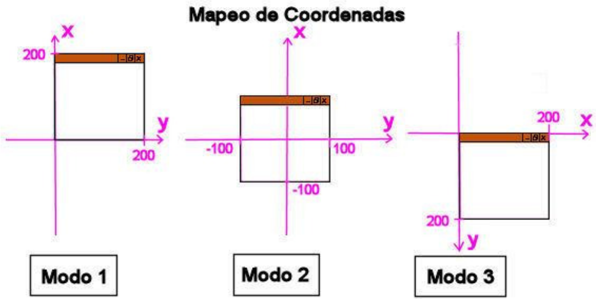
Para trabajar gráficos bien sea 2d o 3d es necesario tener una forma de describir la posición de un objeto cualquiera en el espacio. Para esto es necesario establecer un patrón de referencia, lo cual se hace generalmente a través de un sistema de coordenadas y el sistema al que mas estamos acostumbrados es el de coordenadas cartesianas, en el cual un punto puede ser expresado como una distancia a través uno (x), dos (x,y) o tres ejes (x,y,z) mutuamente perpendiculares que se intersecan en el punto (0,0) en el caso de dos dimensiones y (0,0,0) en el caso de tres.





OpenGL maneja los sistemas 2d y 3d, aunque el sistema de 2d sobre píxeles en pantalla es bastante ineficiente, por lo que en la mayoría de los casos mejor utilizar 3d y simular el 2d con una proyección ortográfica de la escena.

Si bien la forma en que se representan las coordenadas cartesianas en matemáticas es bastante estándar (‘x’ positivo a la derecha y ‘y’ positivo arriba), en los sistemas de dibujo en el computador hay variaciones dependiendo del sistema de dibujo sobre el cual se esté trabajando y el caso del 3d es totalmente dependiente de desde donde, hacia donde y que es “arriba”. Esto quiere decir que las coordenadas del sistema abstracto no tienen que ser concordantes con el sistema de coordenadas sobre la pantalla, por lo que se deber realizar un proceso llamado mapeo de coordenadas.



El grafico muestra tres de las posibles formas de realizar el mapeo las coordenadas. En el Modo 1, el punto (0,0) en coordenadas de dibujo es el correspondiente al extremo inferior izquierdo de la ventana y el punto (200,200) estará en el extremo superior derecho. Tal vez es uno de los modos que parece más "natural".

En el Modo 2, el punto (0,0) en coordenadas de dibujo es el centro de la ventana y los puntos extremos estarán situados en los cuatro cuadrantes, en (100,100) superior derecha, (-100,100) superior izquierda, (-100.-100) inferior izquierda y (100,-100) inferior derecha. Este modo es similar a la forma en que lo muestra la cámara por defecto de OpenGL, con el agregado del eje 'z' positivo "saliendo" de la pantalla.

En el Modo 3, el punto (0,0) en coordenadas de dibujo se encuentra en el extremo superior izquierdo de la ventana y el punto (200,200) corresponde al extremo inferior derecho. Hay que notar, que en este modo el eje 'y' positivo se encuentra "cabeza abajo", es decir, cuando un punto aumenta su coordenada 'y' va hacia abajo, no hacia arriba. Este modo es usado generalmente por los dispositivos de hardware y es la forma por defecto de GDI.

Estas son solo algunas formas de mapeo de coordenadas y en cualquier caso y con un poco de matemáticas (matrices de transformación) se puede pasar de cualquier modo de mapeo a otro.

4.4.3.1.7. Proyecciones en OpenGL

Los monitores actuales de PC, los televisores, las pantallas de PDA, de celulares y consolas portátiles son todos dispositivos bidimensionales, la percepción 3d sobre estos dispositivos se debe a un conjunto de "trucos" para engañar al ojo humano y hacerle ver objetos "tridimensionales" sobre un dispositivo bidimensional, el primero de estos trucos es la proyección.

La matriz de proyección también afecta a los vértices de la escena. Es importante recordar que de forma previa a la especificación de la matriz de proyección es necesario activar dicha matriz e inicializarla mediante las ordenes OpenGL: glMatrixMode (GL_PROJECTION) y glLoadIdentity().

La proyección consiste en generar una imagen bidimensional de una escena tridimensional, algo así como sacarle una foto, o pintar un cuadro de la escena. Tal vez las dos mas conocidas y las que permite trabajar OpenGL directamente son la ortográfica y la perspectiva.

4.4.3.1.8. Proyección Ortográfica

En este tipo de proyección se trazan rayos paralelos al plano de proyección (son paralelos porque se considera la fuente de "luz" o centro de proyección en el infinito) y la imagen se forma con aquellos rayos que intersequen al objeto –figura 4.4.3.1.8.1-.

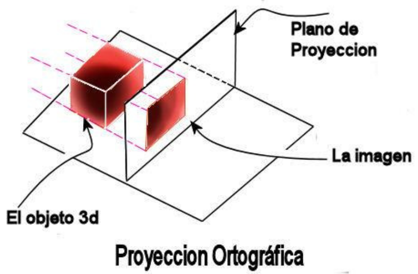


Figura *4.4.3.1.8.1*. Proyección Ortográfica.

Las imágenes producidas por este tipo de proyección se ven como si el objeto se hubiera “aplanado” sobre el plano de proyección (que puede ser tomado como la pantalla), es decir, no existe sensación de profundidad, ya que variando la distancia del objeto al plano de proyección, no variará el tamaño de la imagen producida por este. Este tipo de proyección es usada en aplicaciones CAD porque permite ver el modelo desde diferentes partes (bien sea rotando el objeto o la cámara) sin que se deformen las dimensiones en la proyección.

Una forma de dibujo 2d basada en esta proyección es utilizada en los videojuegos, se trata de los juegos con perspectiva isométrica donde se conservan parte de las medidas, al igual que las líneas paralelas.

También es posible realizar juegos 2d utilizando esta perspectiva, ya que los gráficos se ven planos sobre la pantalla porque su dimensión es independiente de su profundidad con respecto al plano de proyección.

Las imágenes producidas por este tipo de proyección se ven como si el objeto se hubiera “aplanado” sobre el plano de proyección (que puede ser tomado como la pantalla), es decir, no existe sensación de profundidad, ya que variando la distancia del objeto al plano de proyección, no variará el tamaño de la imagen producida por este. Este tipo de proyección es usada en aplicaciones CAD porque permite ver el modelo desde diferentes partes (bien sea rotando el objeto o la cámara) sin que se deformen las dimensiones en la proyección.

4.4.3.1.9. Proyección en perspectiva

En está proyección los objetos sufren una deformación por la distancia, a medida que los objetos se alejan del plano de proyección “disminuyen” su tamaño justo de la forma en que ocurre cuando un objetos se aleja de nuestros ojos (en la figura se muestra como los dos objetos a pesar de ser de diferentes tamaños producen una imagen casi del mismo tamaño sobre el plano de proyección, debido a la diferencia de distancias), es por eso, que esta perspectiva es la mas utilizada para dar una sensación de realismo, como la que se necesitan en los juegos, simulaciones y animaciones.

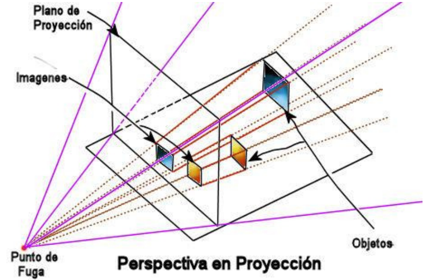


Figura *4.4.3.1.9.1*. Proyección en Perspectiva

4.4.3.1.10. Volúmenes de Vista

En el caso de OpenGL, tanto en el modo de perspectiva como en el ortográfico se limita la región del espacio sobre la que se realizan los cálculos, a esta región se le llama volumen de vista o volumen de recorte (clipping volume) y en el caso de la perspectiva también se la llama frustrum por la forma que tiene. Dicha región se especifica, ya que los objetos por fuera de este campo a derecha o izquierda no aparecerán en pantalla y aquellos muy lejanos generarán pocos rasgos, por lo que de calcularse consumirían recursos innecesariamente (la consideración de "lejano" depende del programador).

En el caso de la proyección ortográfica el volumen de vista tiene forma de caja rectangular, por lo que se determina fácilmente mediante 6 medidas: near, far, top, bottom, left, right (cerca, lejos, arriba, abajo, izquierda, derecha) y en el caso de proyección, tiene forma de pirámide truncada de base rectangular, que puede ser determinada por el field of view (campo de visión), el aspect ratio (el alto del plano mas cercano dividido el ancho), near y far.

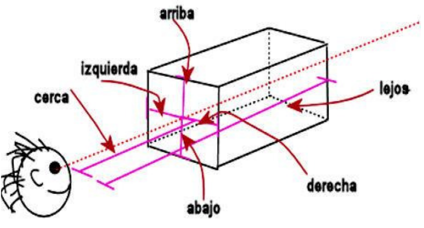


Figura *4.4.3.1.10.1*. Volumen de vista en la Proyección Ortográfica.

Esta define un volumen de la vista cuya geometría es la de un paralelepípedo rectangular (o informalmente, una "caja"). A diferencia de la proyección perspectiva, la distancia de un objeto a la cámara no influye en el tamaño final del mismo en la imagen. Entre las proyecciones ortográficas más comunes en este tipo de sistemas cabe destacar la *proyección ortográfica* (planta, alzado y perfil) y la *proyección isométrica*. La caja de visualización se define en OpenGL de la forma siguiente:

```
void glOrtho (GLdouble left, GLdouble right, GLdouble bottom, GLdouble top,
GLdouble near, GLdouble far)
```

Dónde los parámetros *left*, *right*, *bottom*, *top*, *near* y *far* definen la caja tal y como se muestra en la figura siguiente:

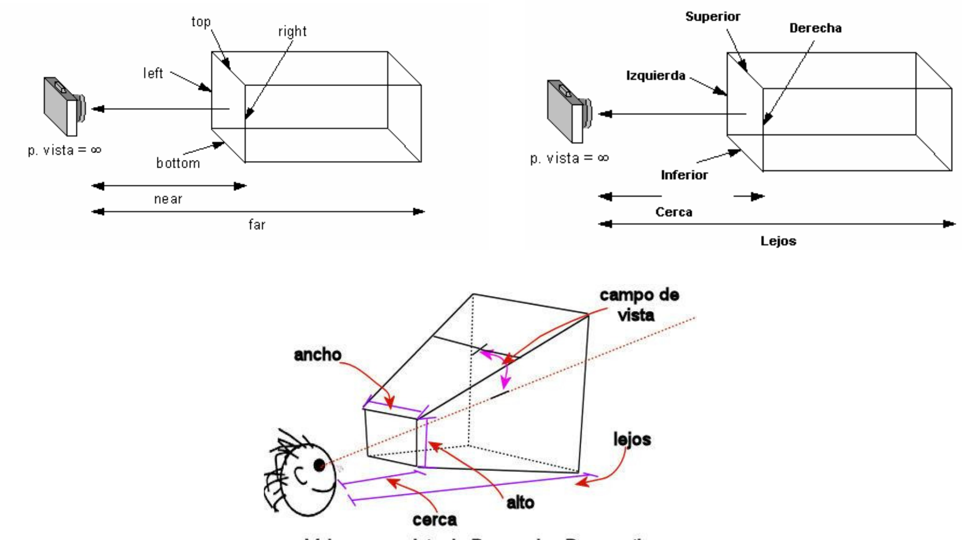
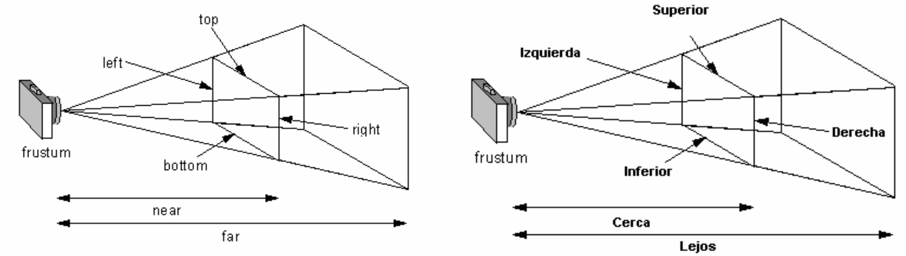


Figura *4.4.3.1.10.2*. Volumen de vista en la Proyección Perspectiva.

Este volumen es una pirámide truncada o *frustum*. Por lo tanto, en esta proyección se produce un efecto tamaño - distancia (los objetos aparecen más pequeños cuanto más alejados están del punto de vista). El frustum de visualización se define en OpenGL de la forma siguiente:

void glFrustum (GLdouble left, GLdouble right, GLdouble bottom, GLdouble top,
GLdouble near, GLdouble far)

Dónde los parámetros *left*, *right*, *bottom*, *top*, *near* y *far* definen el frustum tal y como se muestra en la figura siguiente:

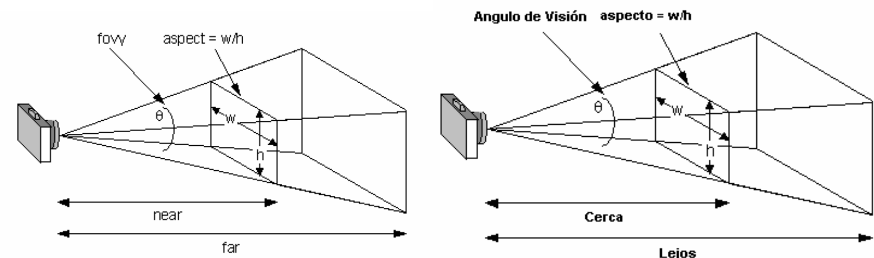


Tal y como se muestra el frustum no tiene porqué ser simétrico respecto al eje Z, ya que es posible utilizar valores distintos para *left* y *right*, o para *bottom* y *top*.

La especificación de una proyección perspectiva mediante *glFrustum* puede resultar complicada debido a que la forma de definición no resulta intuitiva.

En lugar de esta orden, podemos utilizar la rutina de la *librería de utilidades* de OpenGL *gluPerspective*. Esta rutina permite especificar el volumen de la vista de forma

diferente, utilizando el *ángulo de visión* sobre el plano XZ (*fovy)* y el *ratio* de la anchura respecto a la altura (*aspect*). Mediante estos parámetros es posible determinar el volumen de la vista, tal y como se muestra en la figura siguiente:



Esta forma de definir la proyección resulta más intuitiva. La sintaxis de la rutina *gluPerspective* es la siguiente:

void gluPerspective (GLdouble fovy, GLdouble aspect, GLdouble near, GLdouble far)

Cuando utilicemos la rutina *gluPerspective* tendremos que tomar precauciones con respecto a los valores de *fovy* y *near*. Una mala elección de dichos valores puede producir deformaciones en la imagen similares a las aberraciones ópticas habituales en las lentes fotográficas (v.g. efecto "ojo de pez").

Otra forma de definir la manera en que una cámara sintética observa una escena es mediante dos posiciones en el espacio: la del *punto de interés* o punto al que la cámara está enfocando, y la del *punto de vista* o punto dónde se encuentra situada la cámara. Esta forma de definir la transformación de la vista es la que utiliza la orden de la librería de utilidades de OpenGL *gluLookAt*. La sintaxis de la orden es la siguiente:

gluLookAt(GLdouble eyex, GLdouble eyey, GLdouble eyez,
GLdouble centerx, GLdouble centery, GLdouble centerz,
GLdouble upx, GLdouble upy, GLdouble upz);

Donde los parámetros *eyex*, *eyey, eyez* indican la posición del punto de vista; *centerx, centery, centerz* especifican el punto de interés y *upx, upy, upz* la dirección tomada como "arriba", o *inclinación* de la cámara.

Hay que notar que ambos volúmenes se calculan a partir de la posición del observador, que en el caso de OpenGL es la cámara. También hay que observar que las dimensiones del plano más cercano a la cámara pueden ser mayores o menores a las de la ventana, por lo el mapeo de las condenadas no será con relación uno a uno, adicionalmente OpenGL permite definir el viewport (vista), es decir la región sobre la ventana en la que se va a realizar el dibujado, que puede ser menor al tamaño de la ventana.

4.4.3.1.11. Transformación del viewport

Volviendo a la analogía con la cámara fotográfica, la *transformación del viewport* se correspondería con el proceso en que se decide el tamaño final de la

fotografía revelada. Definiremos un viewport como la porción de la ventana de la aplicación en la que se mostrará la imagen. El hecho de que el viewport sea una porción de una ventana implica que el encargado de crearlo y manejarlo será el *sistema gestor de ventanas* sobre el que se ejecute la aplicación (X-11, Windows, etc.) y no OpenGL.

Por defecto, al crear una ventana se inicializa un viewport que ocupa toda el área gráfica disponible. Es posible cambiar este valor mediante la rutina OpenGL:

```
void glViewport (GLint x, GLint y, GLsizei width, GLsizei height);
```

El ratio del viewport (width / height) debería ser normalmente equivalente al definido en la proyección utilizada. Existen aplicaciones en las que se divide la ventana en distintos viewports para mostrar una vista distinta en cada uno de ellos (aplicaciones de diseño asistido por ordenador). Si queremos mantener la ventana de proyección y el viewport proporcionales, será necesario detectar los eventos de ventana (fundamentalmente el cambio de tamaño) y redefinir la proyección y el viewport de forma adecuada.

Definimos primero un viewport es un área rectangular de la ventana de visualización. Por defecto es la ventana entera pero podemos variarlo a gusto. Se utiliza la función:

```
void glViewport(GLint x, GLint y, GLsizei w, GLsizei h)
```

Dónde (x,y) es la esquina inferior izquierda del rectángulo o viewport. Esta coordenada debe especificarse con relación a la esquina inferior izquierda de la ventana. Claro está que w, h son la anchura y altura de nuestro viewport dentro de la ventana. Todos los valores son enteros ya que se tratan de pixels.

```
glViewport(0, 0, (GLsizei) w, (GLsizei) h);
```

En primer lugar tenemos decirle a OpenGL como debe proyectar nuestros gráficos en pantalla. Por ello y para empezar le decimos que active la matriz de proyección:

```
glMatrixMode (GL_PROJECTION);
```

Procedemos a "limpiarla" para que no contenga ningún valor que pueda falsear los cálculos y por tanto nos haga obtener resultados inexactos. Para ello cargamos a la matriz activa (proyección) con la matriz identidad:

```
glLoadIdentity ();
```

4.4.3.2. OpenDX

OpenDX apareció en mayo de 1999 cuando IBM convirtió su sistema comercial de visualización IBM Visualization Data Explorer (conocido como “DX”) en un software de código abierto –Apéndice F-. En la actualidad, científicos, ingenieros y analistas de negocios, han utilizado DX para generar imágenes y animaciones de sus

investigaciones. Lo que es más importante, han ganado un mejor entendimiento de datos complejos a través de la visualización.

OpenDX es el resultado de más de diez años de trabajo de programadores e investigadores del T.J. Watson Research Center in New York State. El Cornell Theory Center ha sido uno de los primeros usuarios de OpenDX desde 1991. A partir de la liberación del código, OpenDX se ha portado a distintos tipos de plataformas que incluyen SGI, HP, Sun, Aix, Linux y FreeBSD. De igual manera, existe una versión comercial para plataformas Windows aunque requiere la instalación de software especial.

OpenDX siempre será un trabajo en progreso, en donde todos los usuarios tienen la oportunidad de hacer contribuciones y reportar problemas del código.

OpenDX, es un sistema de visualización de propósito general, que permite leer datos de distintas fuentes de una manera flexible y amigable. Aun cuando la organización de los datos puede ser muy diferente para cada fuente, OpenDX ofrece una manera de leer casi cualquier tipo de datos a través su herramienta conocida como Data Prompter.

Por otro lado, OpenDX contiene un lenguaje de programación visual mediante el cual es muy simple realizar operaciones sobre conjuntos de datos para extraer y visualizar características importantes. Debido a la generalidad de su modelo de datos y a la flexibilidad de la programación visual, OpenDX no está optimizado para alguna rama de la ciencia en particular. Por lo tanto, mientras que es posible utilizar efectivamente OpenDX para visualizar la salida de simulaciones, mediciones, experimentos o cualquier dato generado por computadora, no tiene funcionalidades optimizadas para un área de estudio particular. Por ejemplo, no es posible realizar directamente operaciones visuales dinámica molecular (aunque, se pueden adicionar estas operaciones a través de la construcción módulos especializados).

4.4.3.2.1. Herramientas de OpenDX

OpenDX contiene un conjunto de herramientas e interfaces gráficas para visualizar datos. En términos generales la visualización de datos se puede considerar como un proceso de 3 etapas:

1. Describir e importar datos.
2. Procesar los datos a través de un programa de visualización.
3. Desplegar los resultados en una imagen.

El diagrama de la figura 4.4.3.2.1.1 muestra como y donde se realiza cada etapa de este proceso dentro de OpenDX.

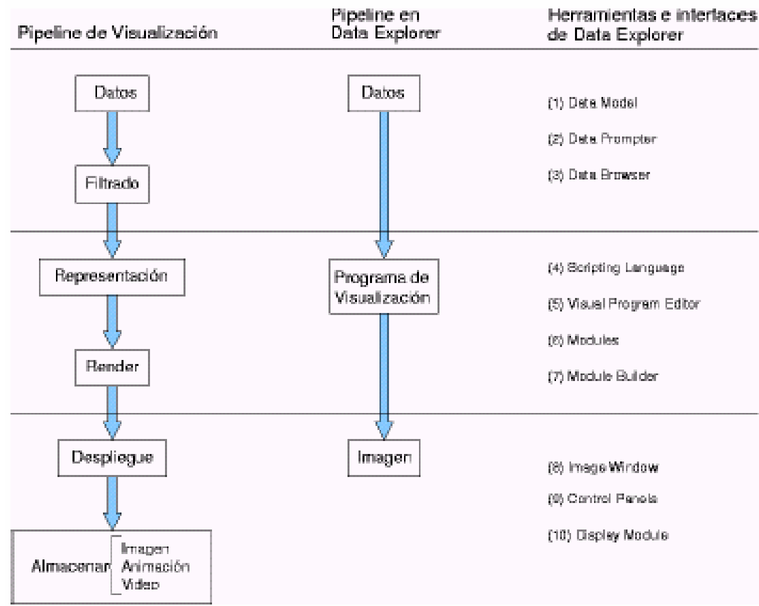


Figura *4.4.3.2.1.1*: Características principales de OpenDX comparadas con un pipeline de visualización general.

En la figura 4.4.3.2.1.1 se enlistan las características contenidas en OpenDX y se hace una comparación con un pipeline de visualización general para obtener una visión global del objetivo de cada una de sus herramientas cuyo propósito se describe a continuación:

(1) Data Model. Conjunto de definiciones, reglas y convenciones usadas para describir las entidades de OpenDX, que incluyen campos de datos, objetos geométricos e imágenes.

(2) Data Prompter. Interfaz de usuario para describir el formato de los datos que serán visualizados en OpenDX.

(3) Data Browser. Interfaz de usuario para revisar datos y determinar el formato y la organización de los mismos. Permite transferir esta información al Data Prompter.

Scripting Language. Lenguaje de alto nivel para crear programas visuales.

Puede también ser usado en modo de comandos. Cualquier programa visual desarrollado en el Visual Program Editor (VPE) es transformado en este lenguaje.

(5) Visual Programa Editor (VPE). Interfaz gráfica de usuario para crear y modificar programas (redes) visuales. Los programas creados en esta interfaz son traducidos al scripting language de OpenDX y se almacenan en este formato.

(6) Modules. Data explorer contiene más de un centenar de módulos, los cuales encapsulan algoritmos y procesos que se realizan dentro de un programa visual.

Los módulos son los bloques de construcción de los programas visuales y puede accederse a ellos por medio del VPE.

(7) Module Builder. Esta es una interfaz que permite personalizar y crear nuevos módulos dentro de OpenDX. Posteriormente, estos nuevos módulos pueden utilizarse en un programa visual.

(8) Image Window. Ventana interactiva para ver y modificar la presentación de la imagen producida por un programa visual. Es posible interactuar con la imagen (rotación, traslación, vistas, etc.).

(9) Control Panels. Interfaces de usuario para cambiar los valores de los parámetros usados por un programa visual.

(10) Display Module. Es una alternativa a Image Window, con la desventaja de que no es posible interactuar con la imagen.

4.4.3.2.2. Programación visual

El concepto de programación visual se deriva, de lo que se denomina flujo de datos, en donde conceptualmente se pone un conjunto de datos en la cima de un pipeline y conforme aquellos van “bajando” a través de los componentes del pipeline, se realizan operaciones hasta obtener un resultado final, como se muestra en la figura 4.4.3.2.1.1. En la programación visual, el programador “dibuja” el programa en la pantalla y proporciona información a cada uno de sus elementos para que funcione adecuadamente, véase por ejemplo figura 4.4.3.2.2.1.

En OpenDX los elementos se conocen como módulos y los datos se leen con el módulo de Import. Un programa visual es una “red” de módulos conectados, los cuales realizan ciertos procesos sobre los datos que se desean visualizar. Se pueden realizar de manera interactiva, modificaciones a los parámetros de los módulos, permitiendo una mejor manipulación de las visualizaciones de los datos. En la figura 4.4.3.2.2.1 se muestra un ejemplo en donde se pueden identificar las 3 etapas descritas en el tema 4.4.3.2.1.

Los programas visuales nos se compilan y son portables a cualquier plataforma donde esté instalada la misma versión de OpenDX y que contenga todos los módulos a los que acceda el programa visual.

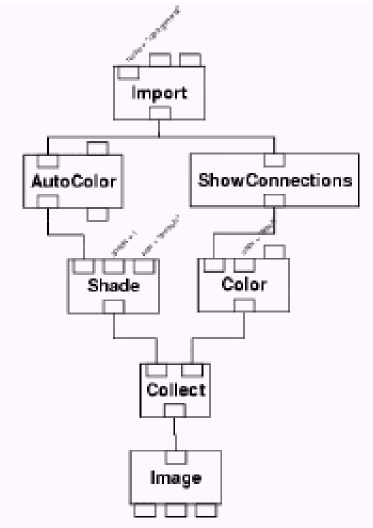


Figura *4.4.3.2.2.1*: Programa visual en DX. El módulo Import lee los datos; los 5 módulos intermedios realizan algunos procesos sobre los datos; el módulo Image despliega la imagen resultante.

4.4.3.2.3 Modelo de datos

La importación de los datos dentro de OpenDX es el primer paso en toda visualización y requiere de conocer algunos conceptos importantes, los cuales se describen a continuación.

Posiciones, conectividad y dependencia de los datos.

OpenDX grafica datos sobre dominios geométricos representados mediante puntos (posiciones en 1D, 2D o 3D) y conectividades. Las conectividades definen elementos, por ejemplo, conectando cuatro puntos se es posible definir un elemento cuadrilátero.

En OpenDX se tienen los siguientes tipos de elementos:

Tipo	Nombre en ODX	No. puntos	Caras/Aristas	Dimensión
Líneas	lines	2	0/1	1D, 2D y 3D
Triángulos	triangles	3	1/3	2D y 3D
Cuadrángulos	quads	4	1/4	2D y 3D
Paralelepípedos	cubes	8	6/8	3D
Tetrahedros	tetrahedra	4	4/6	3D

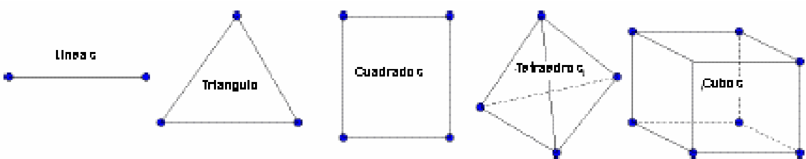


Figura *4.4.4.3.1*: Tipos de elementos definidos en OpenDX.

El dominio geométrico se particiona en un número finito de elementos, a esta partición se le conoce como la "malla" del dominio. Los elementos que conforman la malla pueden ser de varios tipos y dependiendo de esto último se clasifican en mallas estructuradas o no estructuradas. Las mallas pueden ser regulares o irregulares, uniformes o no uniformes.

Los datos que se definen sobre la malla pueden depender de las posiciones o de las conexiones. En el primer caso (position-dependent data) el valor de las variables (los datos) se conoce en cada punto de la malla. OpenDX es capaz de realizar una interpolación al interior de los elementos siempre y cuando exista una conectividad, en otro caso no se podrá tener información de la variable dentro de los elementos.

En el segundo caso (connection-dependent data) la variable es constante dentro de cada elemento y aunque aquí no se realiza ninguna interpolación, es necesario también definir.

4.4.4. Programación orientada a objetos

El paradigma de programación orientada a objetos (POO) tiene como principal objetivo el manejo de la complejidad del software. Los conceptos básicos de abstracción, encapsulación, mensajes, polimorfismo y herencia, permiten desarrollar programas de fácil manejo y control, tanto para el desarrollador como para el usuario final. La descripción y funcionalidad de estos conceptos se da a continuación usando el lenguaje C++ para ejemplificar.

4.4.4.1. Tipos de Datos Abstractos

Algunos autores describen a la POO como programación con Tipos de Datos Abstractos (TDA) y sus relaciones. Un TDA es una abstracción que modela alguna entidad de la vida real y encapsula los datos y las operaciones que se necesitarían para construir dicho modelo. En la POO un problema de la vida real se resuelve mediante la construcción de TDA's que representan entidades bien definidas dentro del dominio del problema. Un TDA consiste de dos partes:

Tipos de Datos Abstractos (TDAs):

Datos o Atributos: En esta parte se describe la estructura de los datos que serán usados para almacenar información.

Operaciones o Métodos: En esta parte se describen las operaciones que serán válidas para el TDA. Un conjunto de estas operaciones serán la interfaz del TDA con las que el usuario podrá interactuar.

4.4.4.2. Clases y objetos

Conceptualmente un objeto es una entidad con identidad y con la que se puede interactuar: a un objeto le podemos enviar mensajes y éste reaccionará de alguna manera determinada a mensajes particulares. La reacción a cada mensaje depende del estado interno del objeto, que puede cambiar como consecuencia de la recepción de un mensaje.

El objeto con el cual se interactúa tiene un nombre, esto es, tiene una identidad que lo distingue de los demás. Por lo tanto, un objeto tiene comportamiento, estado interno e identidad. Esta definición se debe a Grady Booch[13].

Un objeto es una entidad relacionada con el dominio del problema que se desea resolver: es la abstracción de una entidad del mundo real. Una abstracción significa, en este contexto, enfocar la atención en lo que es fundamental, eliminando o posponiendo las particularidades de la entidad en cuestión. Un objeto puede ser un botón, una gráfica, una pantalla, etc., o simplemente algo conceptual que es útil en el desarrollo del sistema.

[13] G. Booch. Object-Oriented Analysis and Design with Applications. Benjamin-Cummings, Redwood City, CA, 1994.

En la implementación, un objeto contendrá algunas características de objetos del mundo real, y características de construcciones conceptuales o computacionales.

La tarea primordial en el desarrollo de un sistema orientado a objetos es la identificación acertada de los objetos que deberán ser implementadas para resolver un problema particular. La interacción de los objetos a través del intercambio de mensajes dará como resultado la solución a nuestro problema.

Cada objeto pertenece a una clase de objetos. La clase define las características de los objetos, en este sentido se dice que un objeto es un ejemplo concreto de una clase. En términos de programación, la clase se relaciona con la definición de un TDA, mientras que el objeto con la declaración de una variable:

```
class TipoNuevo { }; \\ definicion de la clase (TDA)
TipoNuevo objeto1;   \\ declaracion de un objeto de la clase A
```

El estado interno de un objeto es el valor de los datos que encapsula. Estos datos se conocen como atributos y pueden cambiar durante el tiempo de vida del objeto.

Un objeto reacciona cuando recibe un mensaje. El mensaje debe estar definido dentro de su clase, de otra manera dicho mensaje no será válido. Los mensajes son peticiones para que el objeto realice una o varias acciones, las cuales se definen en los métodos. Los métodos, desde el punto de vista de la implementación, no son otra cosa que funciones o subrutinas definidas dentro de la clase. Finalmente se dice que un objeto encapsula sus atributos (datos) y sus métodos (operaciones).

4.4.4.3. Herencia y polimorfismo

La herencia es una de las principales herramientas de rehúso de la POO. Una clase puede derivarse de otra a través de la herencia. La nueva clase se conoce como: clase derivada, clase hija o subclase. La clase de la cual se deriva se conoce como clase base, clase padre o superclase. Las clases derivadas se comportan como un subtipo de la clase base y un objeto de la clase derivada puede ser asignado a una variable del tipo de la clase base a través de apuntadores y referencias.

Las clases derivadas pueden ser usadas a su vez como clases base para derivar otras clases más especializadas, de tal manera que es posible construir una jerarquía de clases tan larga y compleja como se quiera. En C++ es posible también derivar una clase de dos o más clases a la vez, esto se conoce como herencia múltiple. La complejidad de las jerarquías, así como la herencia múltiple provocan algunos problemas que afectan principalmente el desempeño del código.

El polimorfismo dinámico se realiza mediante la herencia y las funciones virtuales. Consiste en que un objeto de una clase derivada se puede comportar como objeto de la clase base. No obstante la utilidad del polimorfismo dinámico, en aplicaciones numéricas puede causar un desempeño muy bajo comparado con otros códigos, esto se debe a que el compilador no es capaz de determinar la clase que se está usando en la llamada a la función, pues esta última recibe como argumento una referencia a una clase general. El polimorfismo estático soluciona el problema de

rendimiento ocasionado por el polimorfismo dinámico. Este tipo de polimorfismo se puede realizar mediante las plantillas de (templates) de C++. Las plantillas conducen directamente a la programación genérica.

4.4.4.4. Programación genérica

El objetivo principal de la Programación Genérica (PG) es desarrollar programas o componentes genéricas, es decir que se puedan reutilizar en un amplio rango de aplicaciones, manteniendo la eficiencia de los códigos. Mediante la construcción de componentes genéricas se reduce el número de líneas de código que se tienen que implementar.

La programación genérica permite construir programas, que en principio, se pueden adaptar a cualquier tipo de dato abstracto. Los programas genéricos involucran un tipo de polimorfismo no tradicional, en el que códigos especializados para diferentes TDA's se obtienen a partir de la parametrización de un programa genérico, el cual utiliza operaciones comunes (requerimientos) a todos los TDA's para realizar las operaciones del algoritmo que implementa. En contraste con programas normales, los parámetros de un programa genérico son estructuras más ricas en contenido, que pueden ser desde tipos básicos (int, char, oat, etc.) hasta TDA's muy complejos.

La ventaja de este tipo de programación es que se reduce drásticamente el tamaño del código. Por otro lado, debido a la generalidad de los códigos, es probable que algunas operaciones no estén optimizadas para tipos de datos importantes. En este último caso se puede realizar lo que se conoce como una especialización del programa genérico e implementar un caso excepcional para mejorar la eficiencia. En el lenguaje C++, la herramienta de plantillas o templates permite construir código genérico. Las plantillas o templates de C++ son un mecanismo mediante el cual es posible programar usando TDA's como parámetros. De esta manera, es posible que un TDA sea un parámetro en la definición de clases o funciones. Para las funciones el uso de los templates es similar al de las clases, se recomienda consultar Stroustrup[14] para ver más detalles al respecto.

4.4.4.5. Arquitecturas para el sistema

El proceso unificado toma en cuenta los riesgos posibles que se tienen en el desarrollo de software, y permite realizar iteraciones como un medio para controlar ese riesgo.

Además, se basa en el diseño y descripción de arquitecturas sobre las cuales se elaboran las diferentes componentes del sistema. Es importante entonces, muy al inicio del desarrollo, diseñar y describir una arquitectura central a partir de la cual se determinen las componentes o módulos que deben ser desarrollados.

Los módulos deben tener baja dependencia entre ellos para permitir un desarrollo y mantenimiento simple. Un sistema con muchas dependencias tiene un alto

[14] B. Stroustrup. The C++ Programming Language. Addison-Wesley, third edition 1997.

acoplamiento que lo hace difícil de elaborar y mantener. Un buen sistema tiene entonces bajo acoplamiento y los cambios en un módulo no se propagan fácilmente a otros módulos.

La encapsulación permite el bajo acoplamiento entre los módulos debido a que la mayor parte de la implementación está escondida y sólo se puede interactuar mediante una interfaz bien definida que no cambia. Un módulo bien diseñado tiene la propiedad de que sus interfaces proveen una abstracción de alguna entidad bien entendida, pero que sin embargo, su implementación puede ser muy compleja. Los módulos que cumplen con esta propiedad se dice que tienen alta cohesión.

La arquitectura de un sistema es la estructura global que determina la manera en que se desarrollarán las diferentes componentes del software. Un desarrollo basado en componentes permite que éstas puedan ser reutilizadas, modificadas o sustituidas en el futuro sin afectar otras partes. Por supuesto que la reusabilidad de una componente depende también de factores técnicos y del contexto donde fue desarrollada. El contexto lo determina la arquitectura del sistema. Por ejemplo, si dos componentes son desarrolladas usando una misma arquitectura, éstas podrán ser intercambiadas fácilmente, mientras que si se desarrollan con arquitecturas diferentes posiblemente no podrán intercambiarse.

4.4.4.5.1. Sistema Operativo

La plataforma en que se desarrollo fue el Sistema Operativo Linux X11. Puesto que es la plataforma mejor vista en cuanto a nivel de cálculos numéricos, de desarrollo de libre.

4.4.4.5.2. Software de desarrollo

- QT
- OpenGL
- OpenDX

4.5. DISEÑO, PROGRAMACIÓN E IMPLEMENTACIÓN

Actualmente existen muchas herramientas para el desarrollo de interfaces gráficas de usuario, por ejemplo: QT –Apéndice G-, wxWidgets, GTK, Java, etc. En este trabajo se realizó una investigación de estas herramientas y se seleccionó la más adecuada para llevar a cabo el objetivo planteado.

Una vez seleccionada la herramienta a utilizar, se hizo un estudio de las interfaces que utilizan algunos de los programas existentes en la actualidad para encontrar sus defectos en cuanto al uso y el aprendizaje. Posteriormente, se realizó un diseño que subsane los defectos de otras interfaces. Además, se realizarán interfaces con efectos atractivos a los usuarios que le permitan recordar los pasos que se requieren para realizar una simulación. La parte donde la interfaz deberá permitir adecuar su

presentación (color, forma, distribución, etc.) en esta versión se subsana mediante el uso de mostrar la interfaz como una vista de Windows, Plastique, Motif y CDE.

4.5.1. Diseño

Después de realizar el análisis y establecer los procesos y quienes los utilizarán o ejecutarán, se puede empezar el diseño del sistema.

4.5.1.1. Diseño del sistema

Esta parte exige tomar en cuenta las técnicas de diseño más adecuadas, esta actividad es fundamental y la razón de ser de nuestro desarrollo, ya que los usuarios tendrán contacto con el sistema a través de la interfaz.

Al abrir una ventana los primeros segundos en los que una persona contempla la interfaz son cruciales, y de ellos depende que esta continúe con el sistema. El objetivo de usar estos principios es alcanzar la claridad visual reforzando las relaciones lógicas. Poner la información de tal manera que minimice los movimientos del ojo para que la persona adquiera las diferentes unidades de información necesarias para la tarea y minimizar los movimientos de la mano requeridos en la navegación del cursor en la pantalla.

Toda interfaz debe ser amigable, ya que será un elemento de trabajo, los procesos deben de ser los establecidos en el análisis, el orden y la secuencia en que se interactué entre los elementos de la interfaz.

Debe existir una respuesta del sistema para los eventos que requieran de una retroalimentación, en ocasiones los procesos pueden tomar más tiempo de lo esperado y el usuario puede pensar que no esta funcionando, por lo que es conveniente el uso de mensajes para informarle la situación o tener módulos de ayuda.

Una vez recaudada la información necesaria se realizó un modelo de datos que es una serie de conceptos que puede utilizarse para describir un conjunto de datos y las operaciones para manipularlos. Buscando alternativas desde el punto de vista lógico.

Todas las observaciones y especificaciones deben ser desde un punto de vista funcional, se debe diseñar el modelo lógico de procesos describiendo lo que hará el sistema sin definir cómo lo hará, considerando todos aquellos agentes que puedan intercambiar información, definición de los flujos de datos tomando en cuenta los contenidos, frecuencia y sucesos que originen.

Se crearon modelos lógicos de datos con el apoyo de diagramas del Lenguaje Unificado de Modelado (UML); teniendo que especificar qué módulos del sistema le corresponde interactuar con un perfil de usuario en específico tratando de obtener los volúmenes de información y la frecuencia con que se da.

Se realizó durante el desarrollo, diferentes pruebas de aceptación, mostrando los bosquejos del sistema al usuario, teniendo la aprobación que sirvió para verificar el cumplimiento y funcionalidad del sistema.

4.5.1.2. Adquisición de requerimientos

Para el desarrollo de la interfaz, sabemos que es necesario el proceso de adquisición de las necesidades y requerimientos operacionales, en el cuál es vital la participación del usuario y el desarrollador. Así pues, en esta etapa, se realizó una discusión con expertos en el área sobre los aspectos necesarios para realizar el control del sistema de manera eficiente y funcional.

4.5.1.3. Reuniones

Durante las sesiones con los expertos en el área de Física se discutió sobre la forma en que obtienen sus resultados, los pasos a seguir para lograr o averiguar la(s) dirección hacia su investigación. Así como cuáles eran los pasos que seguían para mejorar o llegar a un resultado visual e ilustrativo.

Previo a la reunión se hace una investigación, para comprender a nuestro usuario enfocándose a sus necesidades computacionales, representativas y de cálculo.

En cada reunión se busca recolectar conceptos más importantes, experiencias de un tema que el experto domina y como exponerlo de manera interactiva y clara. Recolectar las dudas más frecuentes observadas al utilizar el usuario un ordenador, si es que ha tenido experiencia en ello. Analizar las demandas del usuario en cuanto a sencillez se refiere.

Como resultado de algunas reuniones, y siguiendo con las recomendaciones generales para la adquisición del conocimiento de las necesidades tanto operacionales como visuales (parte importante de este trabajo), se logró obtener parte de la información valiosa que permitió, posteriormente, modelar dicho conocimiento y convertirlo en parte visual y operativa de la importación de este estudio, desarrollo de una interfaz gráfica.

Esta información obtenida a grandes rasgos fue el saber limites de campos, que campos requerían que fueran de selección así como qué y de que manera debería estar estructurada la información.

Es necesario reunirse constante mente, pero conforme paso el tiempo se observo que era complicado tener reuniones por lo en este caso, las reuniones no resultaron ser el método mas efectivo en el proceso de adquisición del conocimiento, debido a que el usuario, dadas sus múltiples ocupaciones, no siempre estaba disponible en el momento en que se le necesitaba, observando con lo anterior la dependencia existente entre el desarrollador y el usuario, originando el aumento significativo del tiempo de desarrollo, repercutiendo en el avance del proyecto. Por lo tanto se decidió utilizar el método de adquisición de requerimientos y conocimiento del problema, así como de visualización,

economizar el tiempo, tanto del desarrollador como del usuario y había reuniones poco frecuentes.

Subsecuentemente se trabajó en cuál sería la forma adecuada de presentar al usuario sus procedimientos para llegar a una representación visual de su problema numérico. Se eligieron entre varios software para el desarrollo de interfases así como diseños, que nos diera la facilidad de representar de manera sencilla y agradable nuestros campos para inserción y obtención de datos.

Más adelante se retomaron más en detalle como se interactuaría con la interfaz, que lo haría más sencillo y que se requería y/o ya no será necesario colocar o simplemente modificar para su vista.

Posteriormente se inició la implementación de una primera versión de la interfaz y durante, con apoyo de personas familiarizadas con la programación de interfases gráficas e incluso con el recurso utilizado para la realización de la interfaz (software, librería QT), se fue realizando la parte tanto operacional como visual de la interfaz gráfica. Una vez desarrollada la primera versión, se mostró al usuario, donde fueron realizadas las modificaciones pertinentes, llegando al primer punto satisfactorio de la interfaz.

4.5.1.4. Mapa de navegación

El mapa de navegación para el sistema mantiene un equilibrio entre las estructuras y las relaciones entre sus procesos. Debe establecer una navegación lógica, deberá llevar un orden una secuencia para cuando se navegue entre los elementos de la interfaz y sus pantallas.

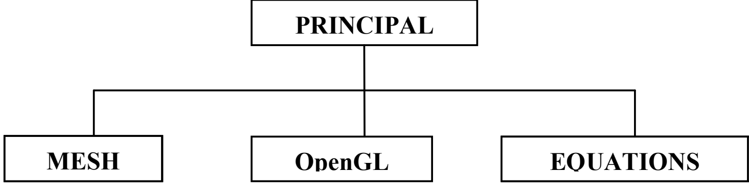


Figura *4.5.4.1.* Mapa General.

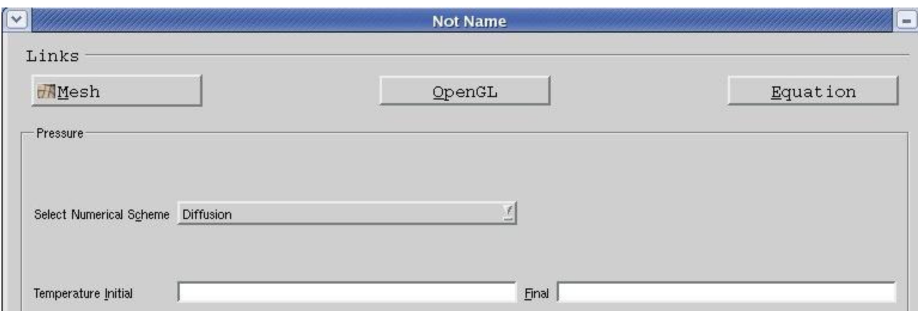
4.5.1.5. Diseño de pantallas

Las pantallas del sistema se presentan a continuación describiendo brevemente cada una de las partes que las componen.

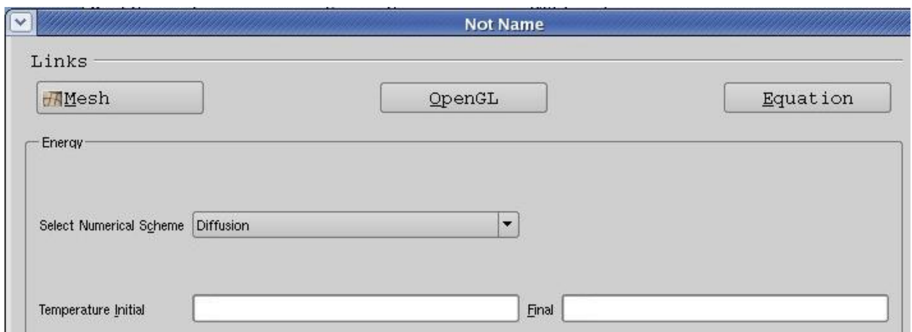
Con la sección de links, es posible moverse de una pantalla a otra sin necesidad de ir en orden aunque para fines de cálculos es necesario insertar datos para obtener el resultado deseado.

En está parte es posible elegir estilo (Windows, Plastique, Motif y CDE) de representación de alguna plataforma en especial; por el momento se introdujeron algunas opciones.

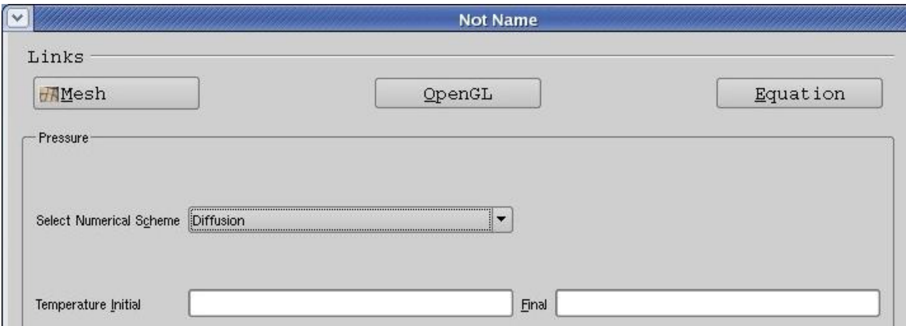
Motif



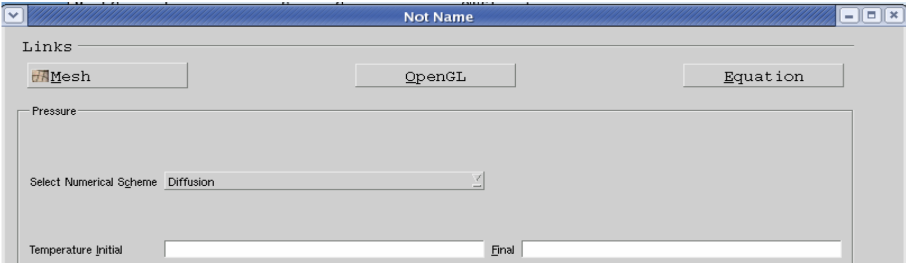
Plastique



Windows

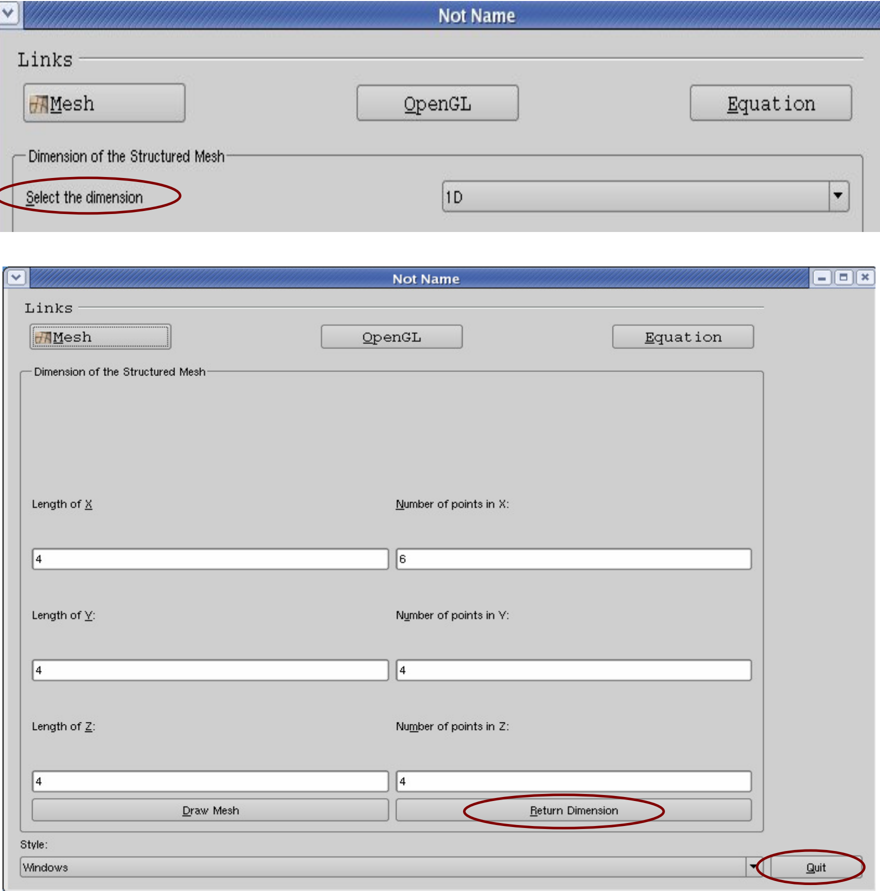


CDE



En general esta pantalla permite al usuario:

- Elegir tres tipos de mallas a crear, 1, 2 y 3 dimensiones –"Select the dimension", así como ingresar los valores del tamaño a nivel longitud y número de puntos.
- Puede cambiar de dimensión si no es la deseada dando clic en el botón de "Retorn Dimension".
- "Draw Mesh" llama al visualizador de la malla.
- "Quit" le da la opción al usuario de salir del sistema.



Figuras 4.5.1.5.1. Pantalla "MESH" Malla.

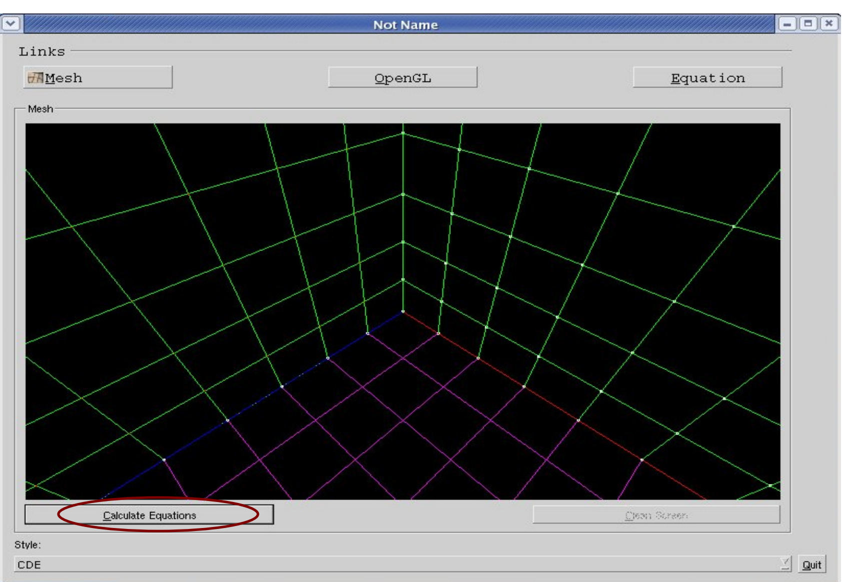
Podemos apreciar que no cuenta con menús comunes, ya que de lo que se trata es de hacer una interfaz amigable, fácil de utilizar, donde solo se muestre lo necesario para realizar la tarea deseada, no bombardearlo de herramientas y maneras de hacer una misma tarea.

El usuario puede generar tantas veces lo desee un malla en diferentes dimensiones, pero considerando que solo trabajara con una para la especificación de su entorno o mundo a estudiar.

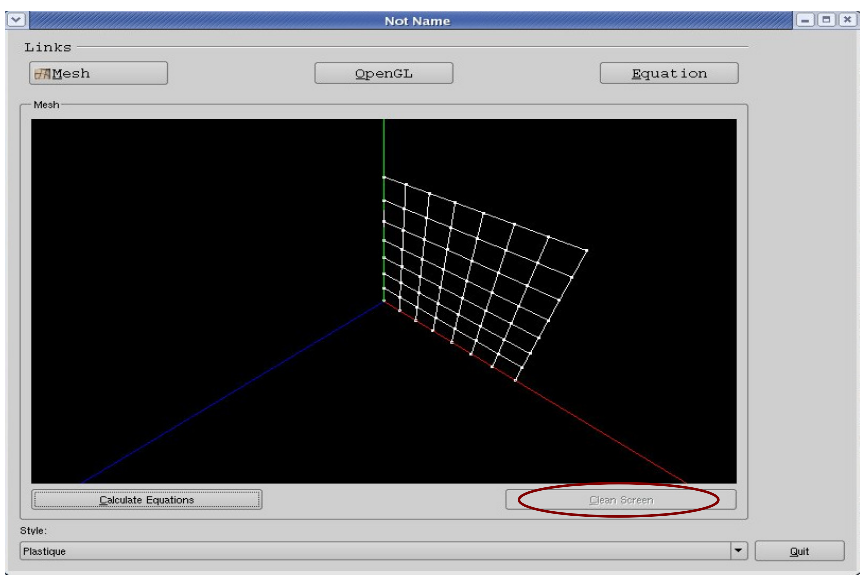
Pantalla de visualización de la malla (mesh) "OpenGL". Observa los límites establecidos a su problema; las teclas de flecha le permiten hacer un zoom a la imagen.

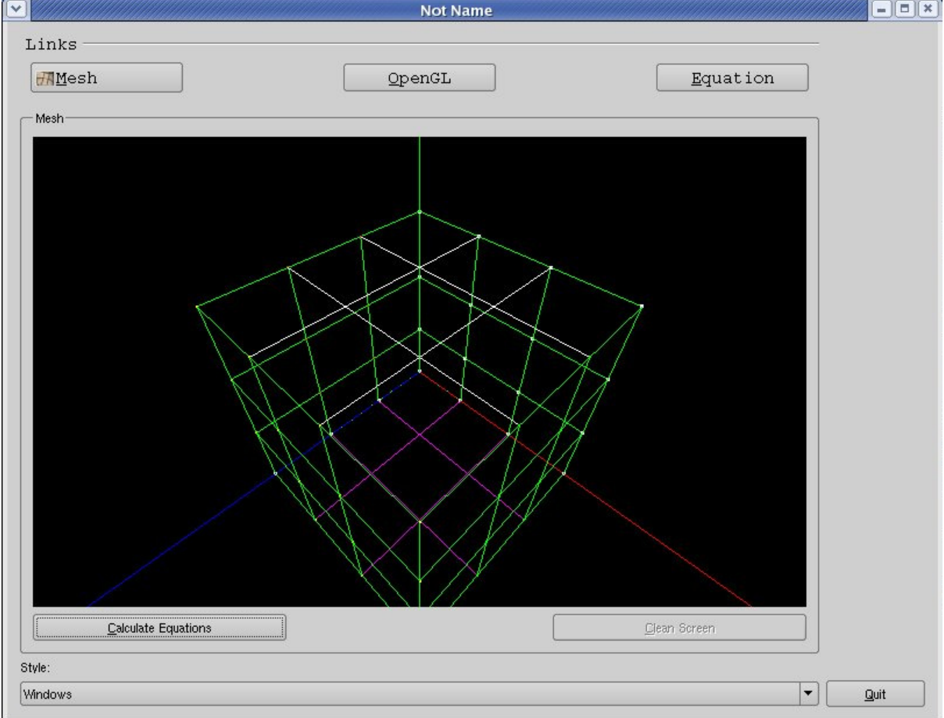
A continuación se describen cada uno de los componentes:

- "Calculate Equation" muestra el formulario donde puedes elegir el tipo de ecuación a calcular para tu problema en particular.
- "Clean Screen" limpia la pantalla para que puedas retroceder y cambiar de dimensión o datos de inicialización.



Figuras *4.5.1.5.2.* Pantalla "OpenGL" visualizador de la malla.





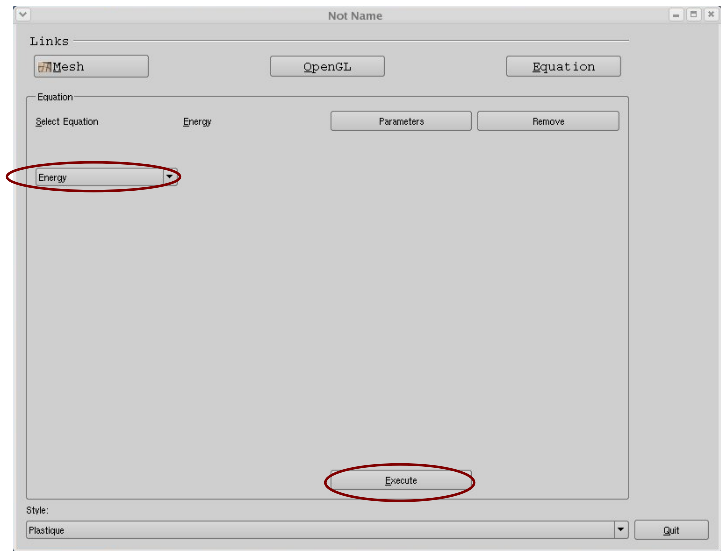
Figuras *4.5.1.5.2.* Pantalla “OpenGL” visualizador de la malla.

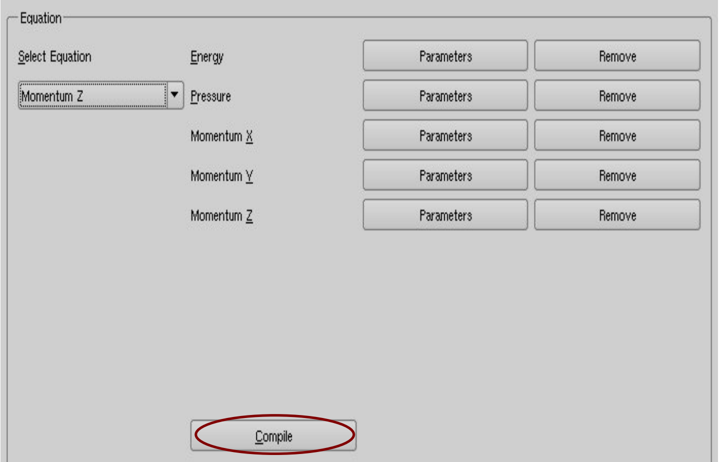
Pantalla de ecuaciones numéricas “Equation”. A continuación se describen cada uno de los componentes:

- “Select Equation” permite elegir entre 5 diferentes ecuaciones (Energy, Pressure, Momentum X, Y y Z), es posible elegir uno o varias de ellas y posteriormente ingresar los parámetros correspondientes.
- “Parameters” para ingresar cada uno de los parámetros necesarios para cada ecuación seleccionada
- “Remove” remueve la ecuación para no ser calculada.
- “Compile” una vez ingresados los datos correspondientes, procedes a

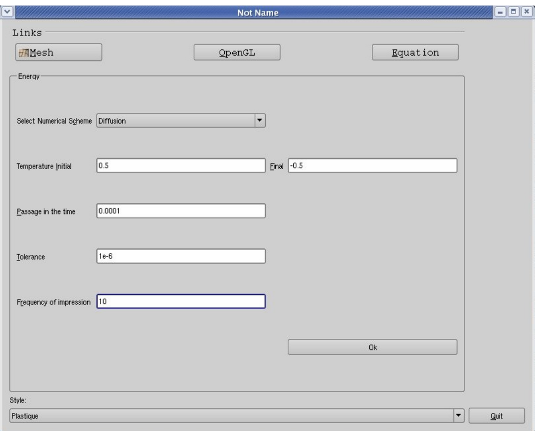
compilar el proceso.

- "Execute" ejecución del proceso de compilación, obtención de resultados. Puedes cambiar de pantalla y volver a compilar con los nuevos parámetros que hayas modificado.

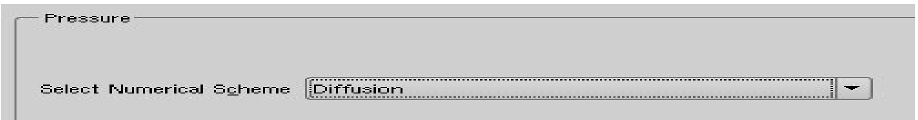




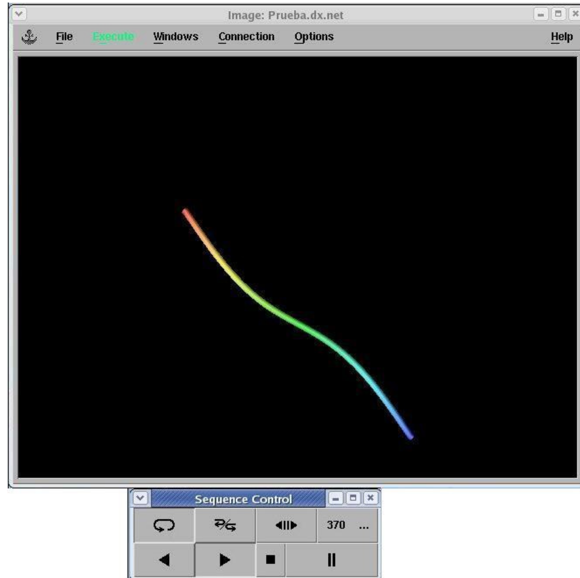
Inserción de parámetros como ejemplo para el cálculo de la ecuación de energía. El botón de "Ok" es para aceptar que los valores ingresados son correctos y se desea continuar.

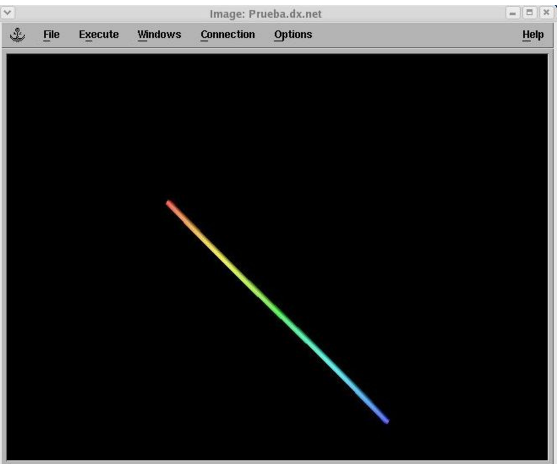


Para cada ecuación también es posible seleccionar el tipo de esquema numérico con el que se desean realizar los cálculos (Diffusion, etc.).



Visualización de resultados después de dar ejecutar "Execute".





4.5.2. Implementación

Para la descripción utilizaremos diagramas así como capturas de pantalla o simplemente a una descripción textual; todo esto para intentar aportar una visión lo más completa posible, que ilustre de manera conveniente el sistema.

Descripción, el programa con las siguientes características:

- El usuario puede generar una malla, proporcionando los valores para la representación de mallas en 1D, 2D y 3D, visualizarla y hacer cambios.
- La visualización de la(s) malla(s) utiliza el modulo de QT, QTOpenGL (OpenGL support classes)[15].
- El usuario puede proporcionar los valores de las variables necesarias de acuerdo a su problema a resolver; visualizando sus resultados con OpenDX.
- Interfaz gráfica amigable.

El uso general de este programa sería como sigue:

- Al principio se le pregunta al usuario de que dimensión desea visualizar la malla.
- Malla 1D, 2D ó 3D:
    - El usuario proporciona longitud y numero de puntos.
    - Se visualiza como se menciono con OpenGL (donde se dibuja).
    - El usuario podrá regresar a redefinir los valores y volver a visualizar los cambios.
    - Posteriormente el usuario podrá definir sus problema proporcionando los parámetros requeridos para dicho problema, resolver las ecuaciones que desea.
- Generar el resultado, que será visualizado con OpenDX.
- Para ello el usuario tendrá la posibilidad de rectificar sus valores y volver a visualizar dichos cambios.

[15] Visita la página http://doc.trolltech.com/4.0/modules.html.

- Dentro de OpenDX podrá utilizar este sistema de visualización.
- OpenDX te permite leer datos de distintas fuentes de una manera flexible aún cuando la organización de los datos es diferente para cada fuente; ofrece una manera de leer casi cualquier tipo de datos a través de Data Prometer.
- Se puede obtener distintos aspectos de apariencia de los objetos que se visualizan en la ventana de imagen (Image Window IW). Permite rotaciones, acercamientos, etc.
- Guardas los datos obtenidos y el usuario regresa al programa donde podrá comenzar a resolver otro problema.

4.5.3. Programación de las pantallas

En la etapa de programación debe existir un estricto apego con lo que se describe en el diseño, contando con la disponibilidad del equipo y software que se requirió para el desarrollo, realizar revisiones informales y formales con el usuario, al encontrar detalles o errores es necesario realizar acciones correctivas.

Se debe contemplar como una parte primordial los perfiles de usuarios, qué tipos de usuarios tendrá acceso a él y los recursos que se necesitan; definir todos los procesos y describiendo qué realiza cada uno de estos.

Para fines prácticos sólo se mostrará fragmentos de código relevante que se utilizó en las diferentes pantallas. Comenzaremos conforme al diagrama de clases –figura 4.5.3.1-.

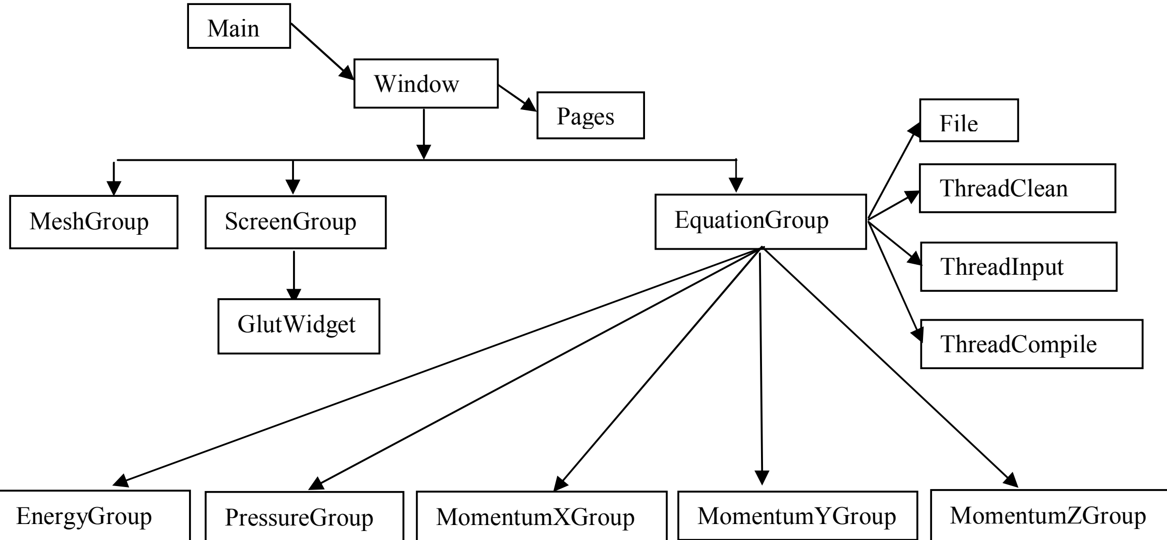


Figura *4.5.3.1*. Diagrama de clases.

En main siempre deberá existir como mínimo un QApplication el cual recibirá los argumentos argc y argv, a partir de esto se creamos nuestra ventana que nombramos Window para ser mostrada y ejecutada.

```
/***********************************************************************
 * Arista Sánchez Lidia
 * main.cpp
 ***********************************************************************/

#include <QApplication>
#include "Window.h"

int main(int argc, char **argv)
{
 QApplication a(argc, argv);

 Window w;
 w.show();
 return a.exec();
}
```

Definición de Window, así como la inserción de los estilos para las diferentes vistas de las pantallas:

```
Window::Window(QWidget* parent, Qt::WFlags f)
      : QWidget(parent,f)
{
    auxiliary = "OD";

    file = new File();

    /*Style*/
    styleLabel = new QLabel(tr("S&style:"));
    styleComboBox = new QComboBox;
    /*QStyleFactory::keys(): add "windows", "motif", "cde", and "plastique"*/
    styleComboBox->addItems(QStyleFactory::keys());
    styleLabel->setBuddy(styleComboBox);

    connect(styleComboBox, SIGNAL(activated(const QString &)),
            this, SLOT(changeStyle(const QString &)));
```

Definición de los diferentes Group que forman las pantallas:

```
private:
    QStackedWidget *stackedWidget;

    Pages *controlsPages;
    MeshGroup *controlsMesh;
    ScreenGroup *controlsScreen;
    EquationGroup *controlsEquation;

    /*Equation*/
    EnergyGroup *controlsEnergy;
    PressureGroup *controlsPressure;
    MomentumXGroup *controlsMomentumX;
    MomentumYGroup *controlsMomentumY;
    MomentumZGroup *controlsMomentumZ;
```

Declaración de los group que estarán agrupados en la Window.

```
controlsPages = new Pages(tr("Links"));
controlsMesh = new MeshGroup(tr("Dimension of the Structured Mesh"));
controlsScreen = new ScreenGroup(tr("Mesh"));
controlsEquation = new EquationGroup(tr("Equation"));

/*Equation*/
controlsEnergy = new EnergyGroup(tr("Energy"));
controlsPressure = new PressureGroup(tr("Pressure"));
controlsMomentumX = new MomentumXGroup(tr("MomentumX"));
controlsMomentumY = new MomentumYGroup(tr("MomentumY"));
controlsMomentumZ = new MomentumZGroup(tr("MomentumZ"));
```

Inserción a la ventana principal (Window) y control de las señales para cada acción que se realiza, es decir, cuando existe un cambio en los valores de algún parámetro en alguna de las diferentes pantallas (divididas o manipuladas por pages), serán obtenidos por la pantalla principal con el motivo que sean conocidos por todas las demás pantallas (pages) que los utilizan, esto es por que aunque cambies de pantalla todas sobran los nuevos datos ingresados en cada una de las demás.

```
stackedWidget = new QStackedWidget();
stackedWidget->addWidget(controlsMesh);
stackedWidget->addWidget(controlsScreen);
stackedWidget->addWidget(controlsEquation);
stackedWidget->addWidget(controlsEnergy);
stackedWidget->addWidget(controlsPressure);
stackedWidget->addWidget(controlsMomentumX);
stackedWidget->addWidget(controlsMomentumY);
stackedWidget->addWidget(controlsMomentumZ);

/*Connection of Pages*/
connect(controlsPages,SIGNAL(changedMesh(int)),this,SLOT(setIndex(int)));
connect(controlsPages,SIGNAL(changedScreen(int)),this,SLOT(setIndex(int)));
connect(controlsPages,SIGNAL(changedEquation(int)),this,SLOT(setIndex(int)));

/*Connection of MeshGroup*/
connect(controlsMesh,SIGNAL(valueChangedNX(int)),this,SLOT(setValueNumPointX(int)))
connect(controlsMesh,SIGNAL(valueChangedNY(int)),this,SLOT(setValueNumPointY(int)))
```

Inserción completa de Window:

```
gridPrincipal = new QGridLayout(this);
gridPrincipal->addWidget(controlsPages,0,0);
gridPrincipal->addWidget(stackedWidget,1,0);
gridPrincipal->addWidget(styleLabel,2,0);
gridPrincipal->addWidget(styleComboBox,3,0);
gridPrincipal->addWidget(quit,3,2);

setWindowTitle(tr("Not Name"));
changeStyle("plastique");
setLayout(gridPrincipal);
```

Función para el cambio entre las diferentes pantallas:

```
void Window::changedGroupScreen(bool checked)
{
 if(checked != false){
    controlsScreen->setValueNX(iNumPointX); controlsScreen->setValueLX(fLengthX);
    controlsScreen->setValueNY(iNumPointY); controlsScreen->setValueLY(fLengthY);
    controlsScreen->setValueNZ(iNumPointZ); controlsScreen->setValueLZ(fLengthZ);
    controlsScreen->setValueAux(auxiliary); controlsScreen->calculateMesh();
    setIndex(1);
 }
}

void Window::changedGroupEquation(bool checked)
{
 if(checked != false){
    setIndex(2);
 }
}
```

Manejo de las diferentes pantallas por medio de Pages.cpp:

```
 pbMesh = new QPushButton("&Mesh");
 connect(pbMesh, SIGNAL(clicked()), this, SLOT(setIndexMesh()));

 pbScreen = new QPushButton("&OpenGL");
 connect(pbScreen, SIGNAL(clicked()), this, SLOT(setIndexScreen()));

 pbEquation = new QPushButton("&Equation");
 connect(pbEquation, SIGNAL(clicked()), this, SLOT(setIndexEquation()));

 gridPages = new QGridLayout(this);

        gridPages->addWidget(pbMesh,0,0);
        gridPages->addWidget(pbScreen,0,2);
        gridPages->addWidget(pbEquation,0,4);

 setLayout(gridPages);
}
```

Clase MeshGroup

```
 * Arista Sánchez Lidia
 * Sep/06
 * MeshGroup.cpp
 ******************************************************************/

#include <QtGui>

#include "MeshGroup.h"

#include <iostream>
using namespace std;

MeshGroup::MeshGroup(const QString &title, QWidget* parent)
        : QGroupBox(title,parent)
{
    auxiliary = "0D";

    intValidator = new QIntValidator(1, 9999, this);
    doubleValidator = new QDoubleValidator(0.0001, 9.9999, 4, this);

    /*X*/
    /*Como si fuera HTML, cambio o estilo de letra*/
    lengthX = new QLabel("<h2><i>Length </i><font color = red> of &X: </font></h2>",0);
    leLengthX = new QLineEdit();
    leLengthX->setValidator(doubleValidator);
    lengthX->setBuddy(leLengthX);
```

Clase ScreenGroup la cual contiene a GlutWidget la pantalla de para visualizar los gráficos con OpenGL:

```
ScreenGroup::ScreenGroup(const QString &title, QWidget* parent)
            : QGroupBox(title, parent)
{
 /*Window of OpenGL*/
 openglWidget = new GlutWidget();

 /*Buttons*/
 pbCalEquation = new QPushButton("&Calculate Equations");
 pbClearGLWidget = new QPushButton("&Clean Screen");
 pbClearGLWidget->setEnabled(false);

 /*Connection with QGroupBox*/
 connect(pbCalEquation,SIGNAL(clicked()),this,SLOT(showEquationGroup()));
 connect(pbClearGLWidget,SIGNAL(clicked()),this,SLOT(cleanScreen()));

 gridOpenGL = new QGridLayout(this);

    gridOpenGL->addWidget(openglWidget,0,0, 1,3);
    gridOpenGL->addWidget(pbCalEquation,1,0);
    gridOpenGL->addWidget(pbClearGLWidget,1,2);

 setLayout(gridOpenGL);
}
```

Métodos principales que deben existir en una ventana para visualizar gráficos en OpenGL.

- initializeGL()

```
/*Convoca al render de contextos, define las listas de despliegue.
  Es llamada solo una vez antes de ser llamada resizeGL() o paintGL().*/
void GlutWidget::initializeGL()
{
 /*Limpia la pantalla con el color de limpieza actual*/
 qglClearColor(clearColor);
 glShadeModel( GL_FLAT );
 glEnable( GL_DEPTH_TEST );
}
```

- painGL()

```
/*Esta funcion paintGL es llamada siempre que necesite
  la widget ser actualizada (redibujar)*/
void GlutWidget::paintGL()
{
 /*Limpia la pantalla antes de dibujar y se le indica que borre el bufer de color
   (es decir, la imagen) y el buffer de profundidadi (depth), por si han sucedido
   cambios*/
 glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
 drawScene();
}
```

➢ resizeGL(width, height)

```
/*Convoca al viewport,la proyeccion,etc.Es llamada cada
  que la widget modifica su tamano.*/
void GlutWidget::resizeGL(int width, int height)
{
  GLfloat aspectRatio;
  /*Prevent of division on zero*/
  if( height == 0){ height = 1; }

  /*Set del viewport to screen complete*/
  glViewport(0, 0, (GLint)width,(GLint)height);

  aspectRatio = (GLfloat) width / (GLfloat)height;

  /*Reset de coordenadas*/
  glMatrixMode( GL_PROJECTION );
  glLoadIdentity();
  /*Set de perspectiva correcta*/
  gluPerspective(60.0f, aspectRatio, 1.0, 1000.0); //near +=0.2 fear += 10.0
  glMatrixMode( GL_MODELVIEW );
  glLoadIdentity();
  /*set camara*/
  gluLookAt(0.0f,0.0f,10.0f, //posicion de la cámara
       0.0f,0.0f,0.0f,  //mira hacia
       0.0f,1.0f,0.0f); //vector (perpendicular a las anteriores)



 /*Podria cambiar el radio con las flechas izq y dere y en near y fear con arria abajo
 recordar sistema solar*/
 glTranslatef(0.0f, 0.0f, 0.0f);

 glRotatef(45.0,1.0f,0.0f,0.0f);
 glRotatef(45.0,0.0f,-1.0f,0.0f);

 /*vacia el buffer de dibujo, haciando que todo lo que este en espera se ejecute*/
 glFlush();
}
```

Función para dibujar la escena:

```
void GlutWidget::drawScene()
{
 /*Draw axex*/
 glBegin(GL_LINES);
  /*X*/
  qglColor(Qt::red);
  glVertex3f(0.0f,0.0f,0.0f);
  glVertex3f(13.0f,0.0f,0.0f);
  /*Y*/
  qglColor(Qt::green);
  glVertex3f(0.0f,0.0f,0.0f);
  glVertex3f(0.0f,13.0f,0.0f);
  /*Z*/
  qglColor(Qt::blue);
  glVertex3f(0.0f,0.0f,0.0f);
  glVertex3f(0.0f,0.0f,13.0f);
 glEnd();

 /*Characteristic to draw*/
 glLineWidth(1.2);
 glPointSize(3.1f);
 qglColor(Qt::white);

 if(option == "1D"){ Meshe1D(); }
 if(option == "2D"){ Meshe2D(); }
 if(option == "3D"){ Meshe3D(); }
}
```

Función para realizar el cálculo de la malla en Window (cálculos de los puntos y líneas de unión).

```
void Window::compileEquation(bool checked)
{
 int value = 0;

 if(auxiliary == "1D"){ value = 1; }
 if(auxiliary == "2D"){ value = 2; }
 if(auxiliary == "3D"){ value = 3; }

    file->setDimension(value);
    file->setFrecuency(iFrecuency);
    file->setEquationScheme(indexEquation,indexScheme);
    file->setValues(fTemLeft,fTemRight,fDt,fTolerance);
    file->setValuesLength(fLengthX,fLengthY,fLengthZ);
    file->setValuesNumPoints(iNumPointX,iNumPointY,iNumPointZ);

 checked = file->generateInput("input");

 if(checked == true){
    checked = file->generateFile("heatcond1D.cpp");
      if(checked == true){
           threadC.start();
      }
      else{ criticalMessage(); }
 }
 else{ criticalMessage(); }
}
```

La imagen siguiente muestra un fragmento de código de GlutWidget para la realización de la escena completa.

```
/********
 ***1D***
 ********/
void GlutWidget::Meshe1D()
{
 GLfloat x;

 /***Genera los puntos***/
 glBegin(GL_POINTS);
    for(x = 0.0; x <= numPtosX; x +=delta_X)
        glVertex3f(x, 0.0, 0.0);
 glEnd();

 /***Conectar los puntos***/
 glBegin(GL_LINE_STRIP);
   for(x = 0.0; x <= numPtosX; x +=delta_X)
        glVertex3f(x, 0.0, 0.0);
 glEnd();

 /*Limpia los comandos de dibujo Forza para dibujar*/
 glFlush();
}
```

```
/********
 ***2D***
 ********/
void GlutWidget::Meshe2D()
{
 GLfloat x,y;

 /***Genera los puntos***/
 glBegin(GL_POINTS);
 /*Inferiores y superiores*/
   for(x = 0.0; x <= numPtosX; x +=delta_X){
       glVertex3f(x, 0.0, 0.0);
       glVertex3f(x, numPtosY, 0.0);}
 /*Izquierda y Derecha*/
   for(y = 0.0; y <= numPtosY; y +=delta_Y){
       glVertex3f(0.0, y, 0.0);
       glVertex3f(numPtosX, y, 0.0);}
 /*Centricos*/
   for(x = 0.0; x <= numPtosX; x +=delta_X){
    for(y = 0.0; y <= numPtosY; y +=delta_Y){
        glVertex3f(x, y, 0.0);}}
  glEnd();

 /***Conectar los puntos***/
```

La conexión de puntos en general es así:

```
/***Conectar los puntos***/
/*Parte superior e inferior*/
glBegin(GL_LINE_STRIP);
  for(x = 0.0; x <= numPtosX; x +=delta_X)
      glVertex3f(x, numPtosY, 0.0);
glEnd();

glBegin(GL_LINE_STRIP);
  for(x = 0.0; x <= numPtosX; x +=delta_X)
      glVertex3f(x, 0.0, 0.0);
glEnd();
/*Lado izquierdo y derecho*/
glBegin(GL_LINE_STRIP);
  for(y = 0.0; y <= numPtosY; y +=delta_Y)
      glVertex3f(0.0, y, 0.0);
glEnd();
```

```
/********
 ***3D***
 ********/
void GlutWidget::Meshe3D()

{
 GLfloat x,y,z;

 /***Genera los puntos***/
 /*Frente*/
 glBegin(GL_POINTS);
 /*Inferiores y superiores*/
    for(x = 0.0; x <= numPtosX; x +=delta_X){
        glVertex3f(x, 0.0, 0.0);
        glVertex3f(x, numPtosY, 0.0);}
    for(z = 0.0; z <= numPtosZ; z +=delta_Z)
        glVertex3f(0.0, 0.0, z);
 /*Izquierda y Derecha*/
    for(y = 0.0; y <= numPtosY; y +=delta_Y){
        glVertex3f(0.0, y, 0.0);
        glVertex3f(numPtosX, y, 0.0);}
 /*Centricos*/
    for(x = 0.0; x <= numPtosX; x +=delta_X){
      for(y = 0.0; y <= numPtosY; y +=delta_Y){
        glVertex3f(x, y, 0.0);}}
 glEnd();
```

Clase EquationGroup, en general está contiene a las diferentes pantallas, de cada una de las ecuaciones que se pueden calcular:

```
/*****************************************************************
 * Arista Sánchez Lidia
 * Sep/06
 * EquationGroup.cpp
 *****************************************************************/

#include <QtGui>

#include "EquationGroup.h"

EquationGroup::EquationGroup(const QString &title, QWidget* parent)
             : QGroupBox(title, parent)
{
    /*Equation*/
    qlEnergy = new QLabel(tr("&Energy"));
    qlEnergy->setVisible(false);
    pbParametersE = new QPushButton("Parameters");
    pbParametersE->setVisible(false);
    qlEnergy->setBuddy(pbParametersE);

    pbEnergyRemove = new QPushButton("Remove");
    pbEnergyRemove->setVisible(false);
    connect(pbEnergyRemove, SIGNAL(clicked()), this, SLOT(removeE()));

    qlPressure = new QLabel(tr("&Pressure"));
    qlPressure->setVisible(false);
    pbParametersP = new QPushButton("Parameters");
```

Función para la ejecutar la(s) ecuación(es):

```
/*Buttons*/
void EquationGroup::executeEquation()
{
 emit bChangedExecute(true);
}

void EquationGroup::compileEquation()
{
 if(indexEquation == 0){
   criticalMessage();
 }
 else{
    pbCompile->setVisible(false);
    pbExecute->setVisible(true);
    emit bChangedCompile(true);
 }
}
```

En cada una de las pantallas se hizo manejo de parámetros vacíos y sólo numéricos mostrando el siguiente mensaje:

```
void Window::criticalMessage()
{
 QMessageBox::critical(this,tr("Error"),QString("You didn't insert values"));
}
```

Código la clase que generará el archivo que será compilado, el cual crea el problema a resolver, como primera parte de esta aplicación (una dimensión):

```
/*******************************************************************
 * Arista Sánchez Lidia
 * Sep/06
 * File.cpp
 *******************************************************************/

#include "File.h"

File::File()
{
 sScheme.assign("Diffusion");
 sEquation.assign("Energy");
}

bool File::generateFile(char *nameFile)
{
 char route[sizeof(nameFile) + 9] = "Archivo/";

 /*It writes within the file*/
 ofstream file;
 file.open(strcat(route,nameFile), ios::out);

 /*It verifies if file opened correctly*/
 if(!file){
   std::cout << "\nThe file can't be opened or be created\n"<< endl;
        exit(0);
   return false;
```

Fragmento del código que se generara:

```
 file << "#include \"Storage/DiagonalMatrix.h\"" << endl;
 file << "#include <Meshes/StructuredMesh.h>" << endl;
 file << "#include \"Equations/EnergyEquation.h\"" << endl;
 file << "#include \"NumSchemes/Diffusion.h\"" << endl << endl;

 file << "using namespace std;" << endl << endl;

 file << "int main() " << endl << "{" << endl;

 file << "  double length, dt, tolerancy;" << endl;
 file << "  double Tleft, Tright;" << endl;
 file << "  int numNodos, frecuency, numVols;" << endl;

 file << "///" << endl;
 file << "///     Get values" << endl;
 file << "///" << endl;
```

Generación del hilo a ejecutar que toma el archivo que se genero de File:

```
/*******************************************************************
 * Arista Sánchez Lidia
 * Oct/06
 * ThreadCompile.h
 *******************************************************************/

#ifndef THREADCOMPILE_H
#define THREADCOMPILE_H

#include <QThread>

class ThreadCompile : public QThread
{
 public:
     void run();
};

#endif
```

```
/*****************************************************************************
 * Arista Sánchez Lidia
 * Oct/06
 * ThreadCompile.cpp
 *****************************************************************************/

#include <QtGui>

#include "ThreadCompile.h"

void ThreadCompile::run()
{
 emit started();
 system("./script.sh");
}
```

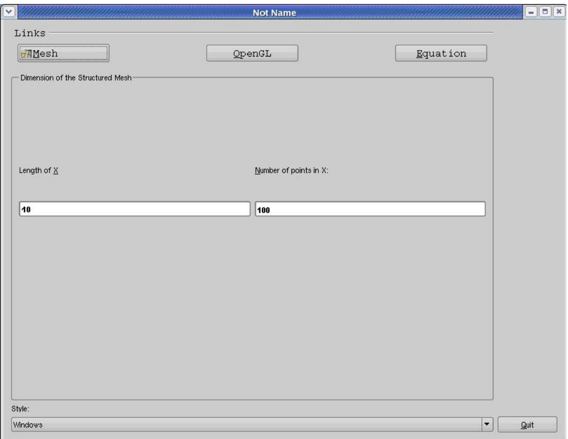
4.5.4. Pruebas

Teniendo las pantallas o algunas de ellas se acudió con el usuario para probar mostrarle el avance de la aplicación, si el objetivo se estaba cumpliendo, si era amigable al usuario, sin olvidar la funcionalidad en cuanto a cálculos numéricos se refiere.

A continuación se describe un problema en particular que el usuario eligió resolver, en está parte de la pantalla el usuario eligió comenzar con la creación de un problema en 1 dimensión (malla en 1 dimensión), la interfaz le permite elegir comenzar con cualquier dimensión, en caso de no ser la deseada, le permite el botón de "Regresar..." modificar su elección de tamaño de dimensión.

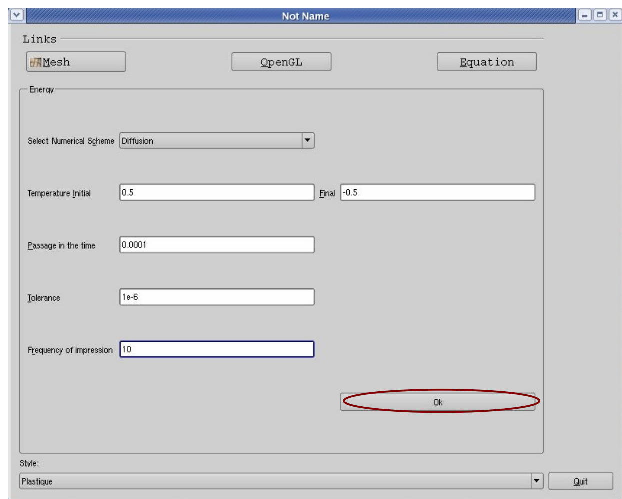
Inserción de datos inicialmente; probó el botón de regresar a elegir dimensión.

Capturo lo siguientes datos:



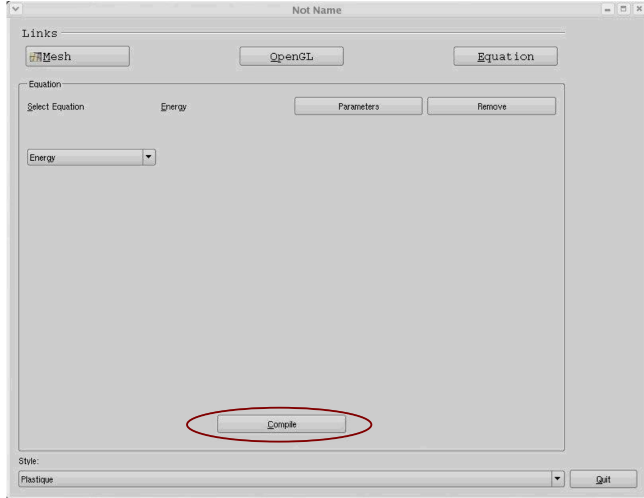
Paso a la pantalla de ecuaciones, en la que eligió únicamente una, ingreso sus p parámetros e hizo el cálculo, obteniendo la imagen en OpenDX la cual muestra los datos por default, pero puede modificar la representación en OpenDX y visualizar los resultados como mejor le satisfaga.

Los valores de los campos de la ecuación con los siguientes valores:

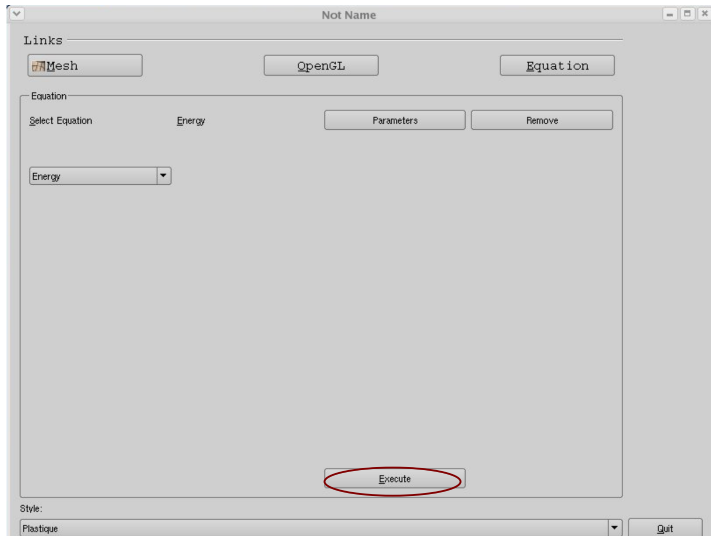


Para cada ecuación también es posible seleccionar el tipo de esquema numérico con el que se desean realizar los cálculos eligió (Diffusion, etc.).

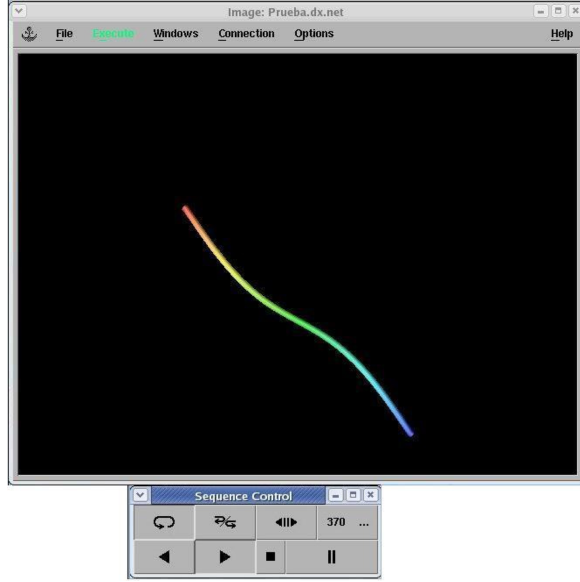
Al clic en Ok, y se muestra:



Después de dar clic en compiler aparece Ejecutar:



Y para visualizar los resultados se da clic en “Execute”.



Con los botones superiores se cambio entre pantallas.

Una vez terminadas las pruebas se continúa con modificar las cosas que el usuario sugirió detallar; recuerda que siempre se deberá buscar la aprobación formal de

los usuarios y directivos. Revisar que cumplan las especificaciones y que pueda interactuar con el entorno.

También con lleva una capacitación de tal manera que al ponerse el sistema en producción deberá lograr su objetivo que será ver o evitar que el usuario no abandone el sistema.

CONCLUSIÓN

En este trabajo se construyo una Interfaz gráfica para resolver el problema en el que se encuentra un físico al enfrentarse a código que por muy simple que esté sea, requiere de conocimientos mínimos en programación orientada a objetos, cosa complicada para un sector de investigadores; y aunque el simulador tiene como objetivo principal ser lo mas simple posible requiere de dichos conocimientos.

En el área de desarrollo de sistemas es necesario contar con una metodología para cada problemática, sin embargo el uso de una metodología no te asegura cubrir todas las deficiencias presentadas, pero si aproximarte al proceso adecuado. Cuando se desarrolla un sistema los procesos tienden adecuarse y/o mejorarse siempre y cuando exista un buen análisis; incluso en la vida cotidiana, en la toma decisiones y no podemos descartar el desarrollo de software, interfaces y/o aplicaciones donde debe existir la mejora continua; sin olvidar la calidad total -hacer las cosas bien y a la primera-.

El Software libre es una excelente opción para el desarrollo de aplicaciones, pero no por ello el software privado no lo es, sin embargo, para dicho objetivo fue necesario buscar opciones que se adaptaran a las necesidades; ya que el software libre por su filosofía, se puede adecuar y cubrir cualquier necesidad, siempre y cuando se tenga conocimiento sobre el funcionamiento y el código en el cual fue desarrollado.

La interfaz es la parte crucial de comunicación entre el usuario y el sistema, por tanto, su característica puede facilitar o impedir dicha comunicación y como resultado el éxito o fracaso del sistema.

Por lo antes mencionado no debemos olvidar que en toda aplicación, independientemente del tipo que sea, por el hecho de involucrar a un usuario, esté último se debe tomar como la parte crucial y más importante de la aplicación, por lo que debe formar parte en cada una de las etapas del proceso de creación de la aplicación. Esto nos traerá como resultado, además de lograr su aceptación, hacer que esté se siente la parte fundamental del desarrollo.

Es importante comentar que la convivencia con el usuario requiere de mucha habilidad y/o experiencia para entender sus necesidades, problemas, inquietudes, etc. Siendo esto notorio en cada una de las etapas, pero con ello se asegura el éxito.

Respecto al desarrollo, se procuro que fuera lo más independiente posible; clase por clase, logrando con ello, la fácil adaptación de nuevos cambios, extensiones y/o adaptaciones, colocando el mayor número de comentarios en código para hacer comprensible su funcionalidad, para facilitar con esto al siguiente programador.

Por último, cabe mencionar que aunque no se ha resulto el problema con todas sus necesidades y soluciones, en la construcción de este prototipo, por así decirlo, constituye una pare importante de inicialización y reduce de manera considerable el entendimiento del problema, ya que esta bien ordenado, y es de fácil uso y extensión

TRABAJO FUTURO

La aplicación desarrollada en este trabajo permite resolver problemas de una dimensión y la estructura general para futuras dimensiones. Debido a la forma en que se construyó, es posible adicionar nuevas características de manera muy simple.

Unas de las características que se pueden agregar en corto plazo son:

Resolver problemas en 2D y 3D.
Uso de iconos.
Mayor uso de eventos.
Agregar esquemas numéricos.

A mediano y largo plazo podrían ser:

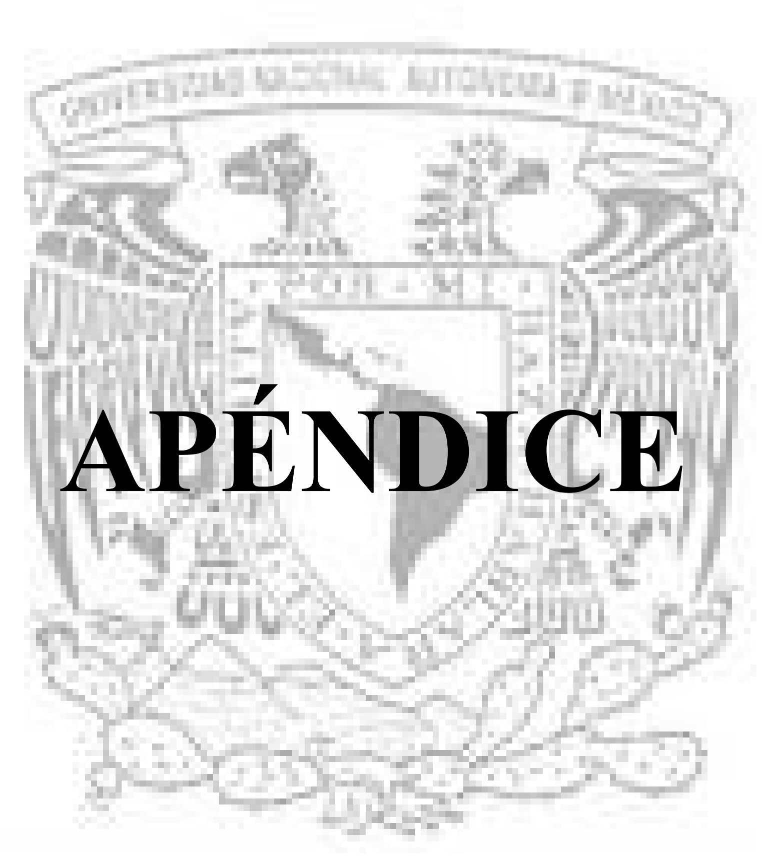
Creación de mallas con el uso del mouse, para la limitación del problema.
Manejo y creación de mallas estructurales no uniformes.
Uso de otros modelos o presentaciones en OpenDX.

Todo lo antes mencionado es posible incluirlo si se sigue el proceso de desarrollo y si se toma en cuenta la arquitectura de la aplicación actual.

Una buena práctica es utilizar los pasos de ingeniería de software, metodología y paradigmas así como llevar un control con un calendario de las actividades a realizar y realizadas.

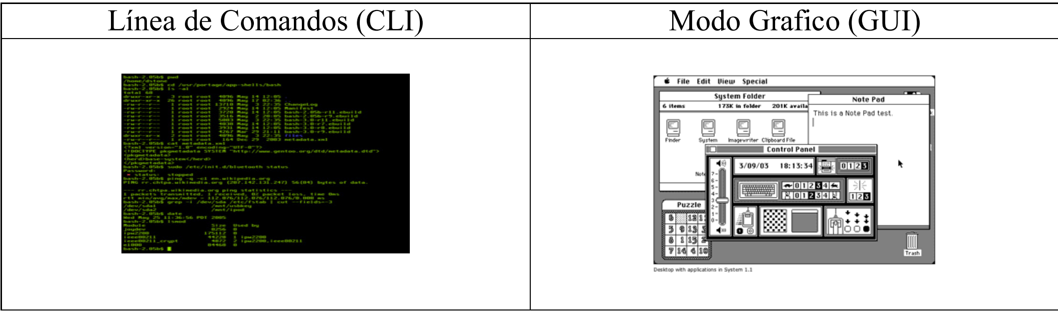
Como resultado del sistema desarrollado, cumple con una parte inicial de las necesidades y de los objetivos planteados sobre el proyecto. Se pretendió ser un tanto futuristas tomando en cuenta los cambios importantes que pudieran existir; no olvidando los requerimientos iniciales a los que por cuestiones de tiempo no era posible llegar, nos enfocamos objetivos y prioridades: logrando con ello dejar una interfaz con lo necesario para poder se continuado y lograr la satisfacción completa del usuario final.

Opino que el presente proyecto es de gran interés para la comunidad universitaria, sobre todo en el área de investigación y propiamente de físicos, ya que cualquier otra persona puede seguir con el proyecto o implantarlo en beneficio de está población y así talvez dar nuevos planeamientos y/o mejoras al desarrollo presente.



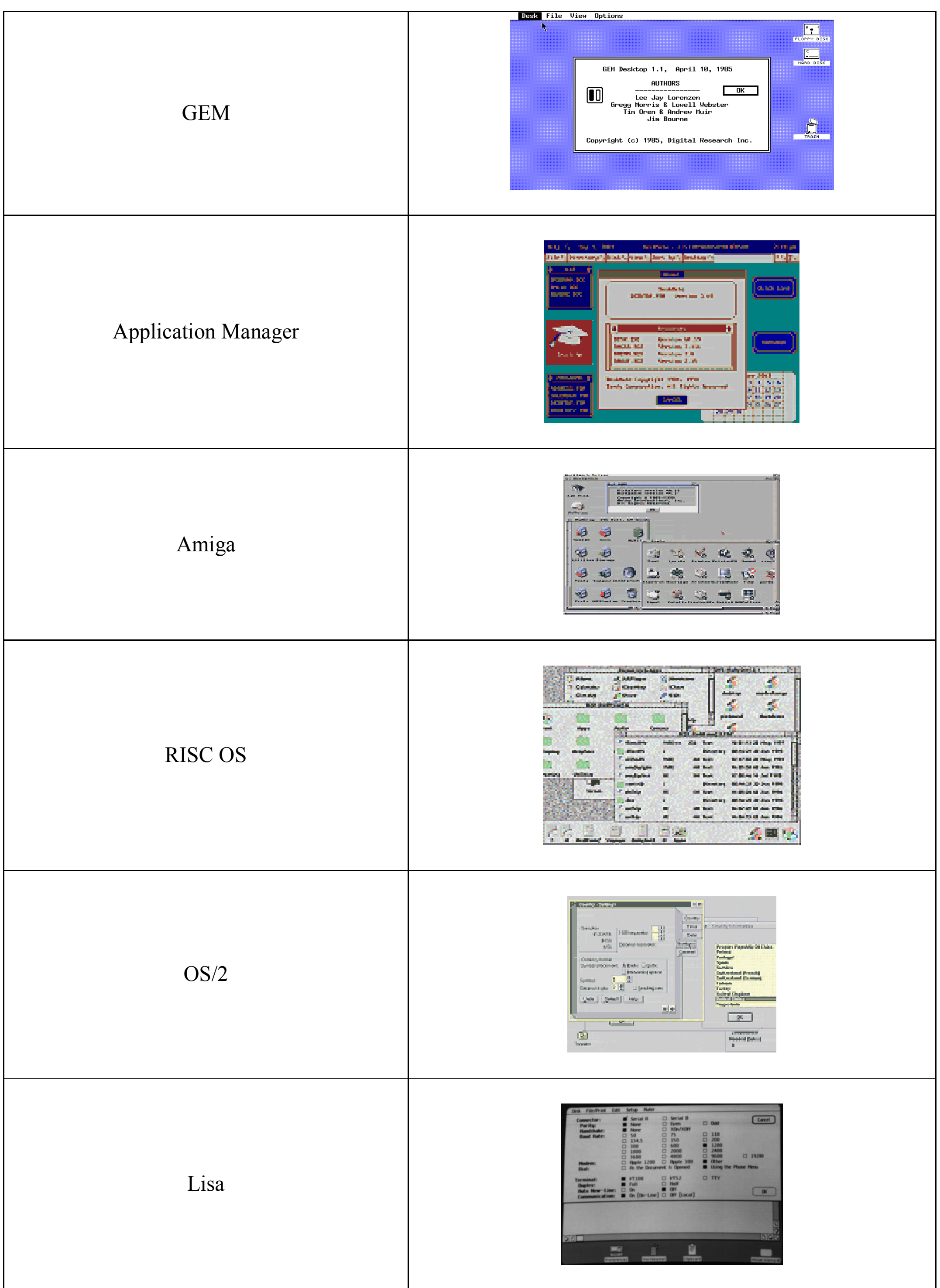
APÉNDICE

APÉNDICE A.
Historia de las GUI a través de imágenes

Línea de Comandos (CLI)	Modo Grafico (GUI)
	

Historia de las GUI a través de imágenes.

Interfaz	Imagen
Xerox Alto	
Bravo -"what you see is what you get"-.	
VisiCorp Visi On	



APÉNDICE B
Código del simulador.

Difusión en una y dos dimensiones

```
1 #include "Utils/Common.h"
2 #include "Equations/EnergyEquation.h"
3 #include "Meshes/StructuredMesh.h"
```

```
4 #include "NumSchemes/Diffusion.h"
5 #include "Solver/Solver.h"
6
7 int main()
8 {
9 double longitud, dt, dx, tolerancia, error, Tleft, Tright;
10 int num_nodos, num_vols, iteracion, max_iter;
11 //... inicializacion de variables ...
12
13 DiagonalMatrix<Tri> A(num_vols); // Matriz A
14 ScalarField1D b(num_vols); // Vector b
15 StructuredMesh<double, 1> mesh(longitud, num_nodos); // Malla
16 ScalarField1D T(mesh.getExtentVolumes());
    // Campo escalar T 6.1 Resultados preliminares 77
17
18 dx = mesh.getDelta(0);
19 firstIndex index; //
20 T = sin(PI * index * dx / longitud); // Condiciones
21 T(T.lbound(firstDim)) = Tleft; // iniciales
22 T(T.ubound(firstDim)) = Tright; //
23
24 EnergyEquation<double, 1, Diffusion<double, 1> > energia(T);
25 energia.setLinearSystem(A, b);
26 energia.setGamma(1.0);
27 energia.setDeltas(mesh.getDeltas());
28 energia.setDeltaTime(dt);
29 energia.setDirichlet(LEFT_WALL, Tleft);
30 energia.setDirichlet(RIGHT_WALL, Tright);
31
32 while (error > tolerancia && ++iteracion < max_iter) {
33 energia.calcCoefficients();
34 Solver::TDMA(energia);
35 error = energia.calcError();
36 energia.update();
37 Output::printToFile_GP(T, iteracion, "temp.", dx);
38 }
39 return 0;
40 }
```

Convección forzada en 2D y 3D

```
#include "NumSchemes/QuickE.h", equema QUICK.
#include "NumSchemes/UpwindE.h", esquema Upwind.
#include "NumSchemes/CentralDiffE.h", esquema CDS.

DiagonalMatrix<Penta> A(num_vols_x, num_vols_y);
ScalarField2D b(num_vols_x, num_vols_y);
StructuredMesh<double, 2>malla(longitud_x, num_nodos_x, longitud_y, num_nodos_y);

ScalarField2D T(malla.getExtentVolumes()); // Temperatura
ScalarField2D u(malla.getExtentVolumes()); // u-velocidad
ScalarField2D v(malla.getExtentVolumes()); // v-velocidad
ScalarField2D us(num_nodos_x, num_vols_y); // u-velocidad (staggered)
ScalarField2D vs(num_vols_x, num_nodos_y); // v-velocidad (staggered)

const int bi = T.lbound(firstDim) + 1, ei = T.ubound(firstDim) - 1,
bj = T.lbound(secondDim) + 1, ej = T.ubound(secondDim) - 1;
Range I(bi,ei), J(bj,ej), all = Range::all();
double dx = malla.getDelta(0);
double dy = malla.getDelta(1);
```

```
T(bi-1, all) = left_wall;
T(ei-1, all) = right_wall;
u(I,J) = -A * cos (PI * dy * J) * sin (PI * dx * I);
v(I,J) = A * sin (PI * dy * J) * cos (PI * dx * I);

us = NumUtils::staggerX(u);
vs = NumUtils::staggerY(v);

EnergyEquation<double, 2, QuickE<double,2> > energia(T);
energia.setLinearSystem(A,b);
energia.setGamma(1.0);
energia.setDeltas(malla.getDeltas());
energia.setDeltaTime(dt);
energia.setNeumann(TOP_WALL, 0.0);
energia.setNeumann(BOTTOM_WALL, 0.0);
energia.setDirichlet(LEFT_WALL, left_wall);
energia.setDirichlet(RIGHT_WALL, right_wall);
energia.setUvelocity(us);
energia.setVvelocity(vs);

while ( (error > tolerancia) && (iteracion <= nmax) ) {
energia.calcCoefficients();
Solver::solveByLines(energia, tolerancia, max_iter);
error = energia.calcError();
residuo = energia.calcResidual();
energia.update();
//... actualizaciones, etc.
}
```

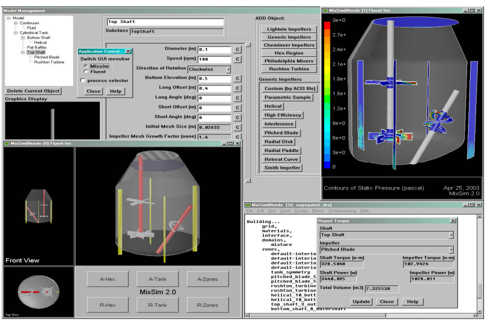
APÉNDICE C

Análisis de interfaces gráficas de otros simuladores Evaluación

Objetivo General

Hacer un análisis de las interfaces existentes para el manejo de flujos para encontrar posibles deficiencias en cuanto al aprendizaje y uso de dicha interfaz. Se hará uso de la metáfora visual -“normalmente es una imagen que nos permite representar alguna cosa y que el usuario puede reconocer lo que representa y por extensión puede comprender el significado de la funcionalidad que recubre”-.

FLUENT[1]



Principios generales del diseño de la GUI

Objetivo

Valorar la usabilidad de la interfaz en relación con los seis principios que rigen el diseño de Interfaces Gráficas de Usuario.

S: Siempre **CS**: Casi Siempre **O**: Ocasionalmente **N**: Nunca **NP**: No Procede

Simplicidad					
Puntos	**S**	**CS**	**O**	**N**	**NP**
La composición de las pantallas resulta sencilla y ordenada			X		
La apariencia de los controles es sencilla			X		
Semejanza del sistema al mundo real, ya que resulta sencillo de interpretar			X		
El reconocimiento de acciones y opciones es sencillo				X	
El número de colores diferentes utilizados de forma simultánea es igual o inferior a cuatro			X		
El color utilizado para codificar categorías o tipologías es consecuente con la simbología asociada		X			

La consistencia se manifiesta en los siguientes aspectos:

Consistencia					
Puntos	**S**	**CS**	**O**	**N**	**NP**
El tamaño de los elementos de la interfaz				X	
La distribución de los elementos de la interfaz en la pantalla				X	
Las zonas en que se divide la pantalla				X	

[1] Para consultar información acerca de esta aplicación consulta la siguiente página: www.fluent.com

Consistencia					
El uso del color			X		
El tamaño y las fuentes del texto			X		

Existe un contrasté suficiente entre:

Contrasté					
Puntos	**S**	**CS**	**O**	**N**	**NP**
Los elementos de la interfaz con distinta función				X	
Los elementos de la interfaz con distinta importancia				X	
Las diferentes zonas de la pantalla				X	

Eficacia					
Puntos	**S**	**CS**	**O**	**N**	**NP**
La disposición de los elementos facilita su rápida identificación				X	
Los controles favorece la comodidad en su manejo				X	
Control y libertad por parte del usuario			X		

La funcionalidad de los controles es predecible:

Predicción					
Puntos	**S**	**CS**	**O**	**N**	**NP**
En los menús			X		
En los botones				X	
En los iconos				X	
Cada control tiene asignada una única función que es siempre la misma	X				

Retroalimentación					
Puntos	**S**	**CS**	**O**	**N**	**NP**
Las opciones de los menús ofrecen un aspecto distinto según su estado		X			
Los botones ofrecen un aspecto distinto -efecto visible- según su estado		X			
Los iconos ofrecen un aspecto distinto -efecto visible- según su estado		X			

Retroalimentación					
Existe un cambio evidente al realizar alguna acción		X			
Conoce el usuario la información necesaria para llevara a cabo una tarea			X		

Composición
Objetivo

Valorar la eficacia en el tamaño y ubicación de los elementos, así como el modo en el que son agrupados y las zonas de la pantalla en las que son situados.

Composición					
Puntos	**S**	**CS**	**O**	**N**	**NP**
El tamaño de los elementos corresponde con su importancia				X	
El tamaño de los elementos contribuye al equilibrio visual				X	
El tamaño de los elementos se adecua al usuario				X	
Los elementos se agrupan visualmente en relación con sus funciones			X		
La posición de los elementos contribuye al equilibrio visual y de uso				X	

Color
Objetivo

Valorar si el color cumple una función informativa y sirve como recurso.

Color					
Puntos	**S**	**CS**	**O**	**N**	**NP**
El color se utiliza para atraer la atención del usuario		X			
El color se utiliza para separar visualmente las zonas de la interfaz			X		
El color se utiliza para resaltar información importante para el usuario		X			

Controles
Objetivo

Valorar el diseño formal de los controles (menús, botones, iconos, cuadros donde insertar información, están en relación con el espacio que ocupan, su permanencia en pantalla y su representación.

Controles					
Puntos	**S**	**CS**	**O**	**N**	**NP**
El espacio que ocupan los controles es adecuado al tamaño de la pantalla (no resulta excesivo)				X	
El orden de las opciones de los menús sigue un criterio lógico			X		

Controles					
El orden de la secuencia de botones o iconos sigue un criterio lógico		X			
Los controles mas utilizados están visibles de forma permanente		X			
Los iconos y botones de carácter icónico se utilizan para la representación de objetos tangibles y concretos					X
Para la representación de conceptos abstractos, procesos y/o acciones se utilizan controles textuales		X			
Se emplean símbolos convencionales en los controles que son utilizados por el usuario con mayor frecuencia				X	

Otros

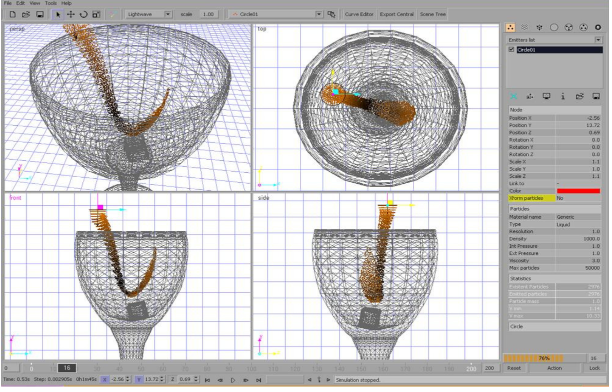
Objetivo

Valorar que tan sencillo resulta para el usuario aprender a utilizar la interfaz, así como si se logro el objetivo de haber tomado o no en consideración al usuario como parte esencial, si es aceptada o simplemente utilizada, si resulta eficaz y eficiente utilizarla.

B: Bajo **M**: Medio **A**: Alto **NP**: No Procede

	Manipulación directa	**Menú**	**Llenado de formas**	**Lenguaje natural**
Tiempo de aprendizaje	B	M	B	B
Velocidad de uso	M	B	M	M
Propenso a errores	M	B	B	B
Tecleo (habilidad)	M	M	A	M
Mouse (habilidad)	M	M	M	M

RealFlow[2]



[2] Para consultar información acerca de esta aplicación consulta la siguiente página: www.nextlimit.com/realflow/index.html

Principios generales del diseño de la GUI

Objetivo

Valorar la usabilidad de la interfaz en relación con los seis principios que rigen el diseño de Interfaces Gráficas de Usuario.

S: Siempre **CS**: Casi Siempre **O**: Ocasionalmente **N**: Nunca **NP**: No Procede

Simplicidad					
Puntos	**S**	**CS**	**O**	**N**	**NP**
La composición de las pantallas resulta sencilla y ordenada		X			
La apariencia de los controles es sencilla		X			
Semejanza del sistema al mundo real, ya que resulta sencillo de interpretar			X		
El reconocimiento de acciones y opciones es sencillo		X			
El numero de colores diferentes utilizados de forma simultánea es igual o inferior a cuatro	X				
El color utilizado para codificar categorías o tipologías es consecuente con la simbología asociada			X		

La consistencia se manifiesta en los siguientes aspectos:

Consistencia					
Puntos	**S**	**CS**	**O**	**N**	**NP**
El tamaño de los elementos de la interfaz	X				
La distribución de los elementos de la interfaz en la pantalla	X				
Las zonas en que se divide la pantalla	X				
El uso del color		X			
El tamaño y las fuentes del texto	X				

Existe un contraste suficiente entre:

Contraste					
Puntos	**S**	**CS**	**O**	**N**	**NP**
Los elementos de la interfaz con distinta función			X		
Los elementos de la interfaz con distinta importancia		X			

Contraste					
Las diferentes zonas de la pantalla	X				

Eficacia					
Puntos	**S**	**CS**	**O**	**N**	**NP**
La disposición de los elementos facilita su rápida identificación		X			
Los controles favorece la comodidad en su manejo		X			
Control y libertad por parte del usuario		X			

La funcionalidad de los controles es predecible:

Predicción					
Puntos	**S**	**CS**	**O**	**N**	**NP**
En los menús		X			
En los botones			X		
En los iconos			X		
Cada control tiene asignada una única función que es siempre la misma	X				

Retroalimentación					
Puntos	**S**	**CS**	**O**	**N**	**NP**
Las opciones de los menús ofrecen un aspecto distinto según su estado		X			
Los botones ofrecen un aspecto distinto (efecto visible) según su estado	X				
Los iconos ofrecen un aspecto distinto (efecto visible) según su estado	X				
Existe un cambio evidente al realizar alguna acción		X			
Conoce el usuario la información necesaria para llevara a cabo una tarea			X		

Composición

Objetivo

Valorar la eficacia en el tamaño y ubicación de los elementos, así como el modo en el que son agrupados y las zonas de la pantalla en las que son situados.

Composición

Composición					
Puntos	**S**	**CS**	**O**	**N**	**NP**
El tamaño de los elementos corresponde con su importancia		X			
El tamaño de los elementos contribuye al equilibrio visual		X			
El tamaño de los elementos se adecua al usuario	X				
Los elementos se agrupan visualmente en relación con sus funciones	X				
La posición de los elementos contribuye al equilibrio visual y de uso		X			

Color

Objetivo

Valorar si el color cumple una función informativa y sirve como recurso.

Color					
Puntos	**S**	**CS**	**O**	**N**	**NP**
El color se utiliza para atraer la atención del usuario			X		
El color se utiliza para separar visualmente las zonas de la interfaz			X		
El color se utiliza para resaltar información importante para el usuario		X			

Controles

Objetivo

Valorar el diseño formal de los controles (menús, botones, iconos, cuadros donde insertar información, están en relación con el espacio que ocupan, su permanencia en pantalla y su representación.

Controles					
Puntos	**S**	**CS**	**O**	**N**	**NP**
El espacio que ocupan los controles es adecuado al tamaño de la pantalla (no resulta excesivo)		X			
El orden de las opciones de los menús sigue un criterio lógico	X				
El orden de la secuencia de botones o iconos sigue un criterio lógico		X			
Los controles mas utilizados están visibles de forma permanente	X				
Los iconos y botones de carácter icónico se utilizan para la representación de objetos tangibles y concretos		X			
Para la representación de conceptos abstractos, procesos y/o acciones se utilizan controles textuales	X				

Controles				
Se emplean símbolos convencionales en los controles que son utilizados por el usuario con mayor frecuencia		X		

Otros

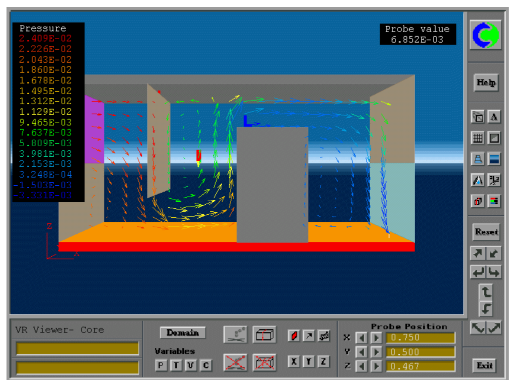
Objetivo

Valorar que tan sencillo resulta para el usuario aprender a utilizar la interfaz, así como si se logro el objetivo de haber tomado o no en consideración al usuario como parte esencial, si es aceptada o simplemente utilizada, si resulta eficaz y eficiente utilizarla.

B: Bajo **M**: Medio **A**: Alto **NP**: No Procede

	Manipulación directa	**Menú**	**Llenado de formas**	**Lenguaje natural**
Tiempo de aprendizaje	A	M	M	A
Velocidad de uso	A	M	M	M
Propenso a errores	A	B	B	B
Tecleo (habilidad)	M	M	A	M
Mouse (habilidad)	A	M	B	M

PHOENICS[3]



Principios generales del diseño de la GUI

Objetivo

Valorar la usabilidad de la interfaz en relación con los seis principios que rigen el diseño de Interfaces Gráficas de Usuario.

S: Siempre **CS**: Casi Siempre **O**: Ocasionalmente **N**: Nunca **NP**: No Procede

[3] Para consultar información acerca de esta aplicación consulta la siguiente página: www.cham.co.uk

Simplicidad					
Puntos	**S**	**CS**	**O**	**N**	**NP**
La composición de las pantallas resulta sencilla y ordenada	X				
La apariencia de los controles es sencilla		X			
Semejanza del sistema al mundo real, ya que resulta sencillo de interpretar			X		
El reconocimiento de acciones y opciones es sencillo			X		
El numero de colores diferentes utilizados de forma simultánea es igual o inferior a cuatro			X		
El color utilizado para codificar categorías o tipologías es consecuente con la simbología asociada			X		

La consistencia se manifiesta en los siguientes aspectos:

Consistencia					
Puntos	**S**	**CS**	**O**	**N**	**NP**
El tamaño de los elementos de la interfaz		X			
La distribución de los elementos de la interfaz en la pantalla		X			
Las zonas en que se divide la pantalla		X			
El uso del color			X		
El tamaño y las fuentes del texto			X		

Existe un contraste suficiente entre:

Contraste					
Puntos	**S**	**CS**	**O**	**N**	**NP**
Los elementos de la interfaz con distinta función			X		
Los elementos de la interfaz con distinta importancia			X		
Las diferentes zonas de la pantalla		X			

Eficacia					
Puntos	**S**	**CS**	**O**	**N**	**NP**
La disposición de los elementos facilita su rápida identificación		X			
Los controles favorece la comodidad en su manejo			X		

Control y libertad por parte del usuario			X		

La funcionalidad de los controles es predecible:

Predicción					
Puntos	**S**	**CS**	**O**	**N**	**NP**
En los menús			X		
En los botones			X		
En los iconos			X		
Cada control tiene asignada una única función que es siempre la misma		X			

Retroalimentación					
Puntos	**S**	**CS**	**O**	**N**	**NP**
Las opciones de los menús ofrecen un aspecto distinto según su estado			X		
Los botones ofrecen un aspecto distinto (efecto visible) según su estado		X			
Los iconos ofrecen un aspecto distinto (efecto visible) según su estado			X		
Existe un cambio evidente al realizar alguna acción		X			
Conoce el usuario la información necesaria para llevara a cabo una tarea			X		

Composición

Objetivo

Valorar la eficacia en el tamaño y ubicación de los elementos, así como el modo en el que son agrupados y las zonas de la pantalla en las que son situados.

Composición					
Puntos	**S**	**CS**	**O**	**N**	**NP**
El tamaño de los elementos corresponde con su importancia		X			
El tamaño de los elementos contribuye al equilibrio visual		X			
El tamaño de los elementos se adecua al usuario		X			
Los elementos se agrupan visualmente en relación con sus funciones			X		
La posición de los elementos contribuye al equilibrio visual y de uso			X		

Color
Objetivo

Valorar si el color cumple una función informativa y sirve como recurso.

Color					
Puntos	**S**	**CS**	**O**	**N**	**NP**
El color se utiliza para atraer la atención del usuario		X			
El color se utiliza para separar visualmente las zonas de la interfaz		X			
El color se utiliza para resaltar información importante para el usuario		X			

Controles
Objetivo

Valorar el diseño formal de los controles (menús, botones, iconos, cuadros donde insertar información, están en relación con el espacio que ocupan, su permanencia en pantalla y su representación.

Controles					
Puntos	**S**	**CS**	**O**	**N**	**NP**
El espacio que ocupan los controles es adecuado al tamaño de la pantalla (no resulta excesivo)		X			
El orden de las opciones de los menús sigue un criterio lógico			X		
El orden de la secuencia de botones o iconos sigue un criterio lógico			X		
Los controles mas utilizados están visibles de forma permanente		X			
Los iconos y botones de carácter icónico se utilizan para la representación de objetos tangibles y concretos		X			
Para la representación de conceptos abstractos, procesos y/o acciones se utilizan controles textuales			X		
Se emplean símbolos convencionales en los controles que son utilizados por el usuario con mayor frecuencia			X		

Otros
Objetivo

Valorar que tan sencillo resulta para el usuario aprender a utilizar la interfaz, así como si se logro el objetivo de haber tomado o no en consideración al usuario como parte esencial, si es aceptada o simplemente utilizada, si resulta eficaz y eficiente utilizarla.

B: Bajo **M**: Medio **A**: Alto **NP**: No Procede

	Manipulación directa	**Menú**	**Llenado de formas**	**Lenguaje natural**
Tiempo de aprendizaje	B	M	M	A

Velocidad de uso	M	M	M	M
Propenso a errores	A	M	B	A
Tecleo (habilidad)	B	M	B	B
Mouse (habilidad)	M	M	M	M

APÉNDICE D
Diseño

El diseño de interfaces es una disciplina que estudia y trata de poner en práctica procesos orientados a construir la interfaz más usable posible, dadas ciertas condiciones de entorno. El entorno dentro del cual se inscribe el diseño de una interfaz y la medida de su usabilidad, está dado por tres factores: Una **persona**, una **tarea** y un **contexto**.

El diseño de interfaces pertenece a un campo mayor del conocimiento humano, de origen altamente interdisciplinario, llamado **Human Computer**.

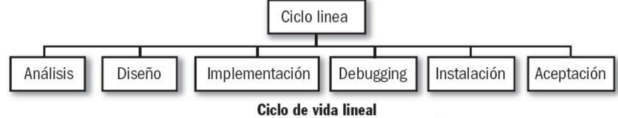
El diseño iterativo de interfaces es un **proceso** independiente de la/s técnica/s utilizada/s para llevarlo a cabo. Actualmente, el proceso del desarrollo de una interfaz se concibe como un ciclo que consta de 4 etapas, en varios niveles:

1. **Diseño**
2. **Implementación**
3. **Medición**
4. **Evaluación**

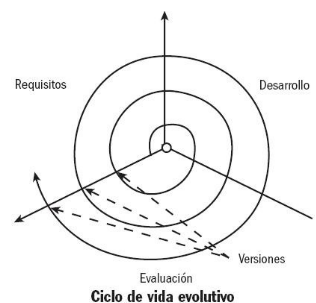
El resultado (o output) de cada etapa es la alimentación (o input) de la que sigue, incluso el de la última. Los resultados de la etapa de evaluación se toman para rediseñar la interfaz, implementarla nuevamente, medir, y así sucesivamente. Debido a esa repetición o auto-alimentación se lo llama **diseño iterativo**.

Es importante comprender que este ciclo no sólo se cumple dentro del ciclo de vida de un producto, sino también *entre productos* y dentro de cada etapa misma. Mientras tengamos tiempo, trataremos de hacer tantos ciclos de mejoramiento como nos sea posible, hasta la fecha límite. La siguiente versión, tomará al producto existente como su comienzo y otra vez comenzará el ciclo.

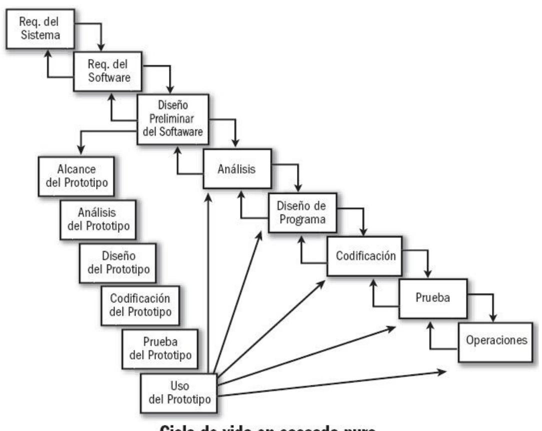
Modelos de Ciclos de Vida



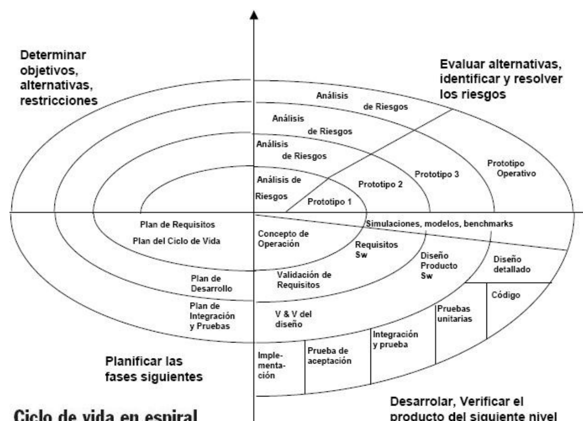
Ciclo de vida lineal



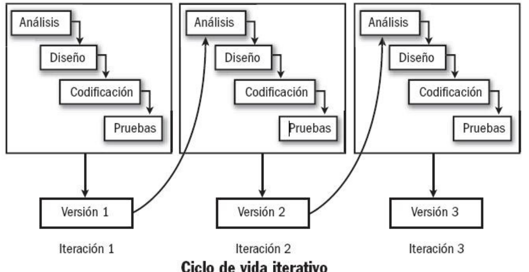
Ciclo de vida evolutivo



Ciclo de vida en cascada puro

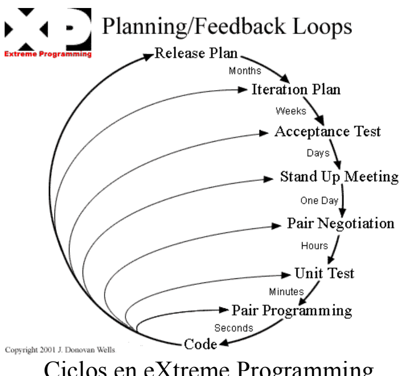


Ciclo de vida en espiral



Ciclo de vida iterativo

El ciclo de vida ideal consiste de seis fases (XP):



Ciclos en eXtreme Programming.

Exploración: En esta fase, los clientes plantean a grandes rasgos las necesidades del usuario que son de interés para la primera entrega del producto. Al mismo tiempo el equipo de desarrollo se familiariza con las herramientas, tecnologías y prácticas que se utilizarán en el proyecto. Se prueba la tecnología y se exploran las posibilidades de la arquitectura del sistema construyendo un prototipo. La fase de exploración toma de pocas semanas a pocos meses, dependiendo del tamaño y familiaridad que tengan los programadores con la tecnología.

Planificación de la Entrega (*Release*): En esta fase el cliente establece la prioridad de cada necesidad del usuario, y correspondientemente, los programadores realizan una estimación del esfuerzo necesario de cada una de ellas. Se toman acuerdos sobre el contenido de la primera entrega y se determina un cronograma en conjunto con el usuario (cliente). Una entrega debería obtenerse en no más de tres meses. Esta fase dura unos pocos días. Las estimaciones de esfuerzo asociado a la implementación de las historias la establecen los programadores utilizando como medida el punto. Un punto, equivale a una semana ideal de programación. Las prioridades generalmente valen de 1 a 3 puntos. Por otra parte, el equipo de desarrollo mantiene un registro de la "velocidad" de desarrollo, establecida en puntos por iteración, basándose principalmente en la suma de puntos correspondientes a las prioridades del usuario que fueron terminadas en la última iteración. La planificación se puede realizar basándose en el tiempo o el alcance. La velocidad del proyecto es utilizada para establecer cuántas prioridades se pueden implementar antes de una fecha determinada o cuánto tiempo tomará implementar un conjunto de requerimientos. Al planificar por tiempo, se multiplica el número de iteraciones por la velocidad del proyecto, determinándose cuántos puntos se pueden completar. Al planificar según alcance del sistema, se divide la suma de puntos de las historias de usuario seleccionadas entre la velocidad del proyecto, obteniendo el número de iteraciones necesarias para su implementación.

Iteraciones: Esta fase incluye varias iteraciones sobre el sistema antes de ser entregado. El Plan de Entrega está compuesto por iteraciones de no más de tres semanas. En la primera iteración se puede intentar establecer una arquitectura del sistema que pueda ser utilizada durante el resto del proyecto. Esto se logra escogiendo las prioridades que fuercen la creación de esta arquitectura, sin embargo, esto no siempre es posible ya que es el cliente quien decide qué requerimientos se implementarán en cada iteración (para maximizar el valor del negocio). Al final de la última iteración el sistema estará listo para entrar en producción. Los elementos que deben tomarse en cuenta durante la elaboración del Plan de la Iteración son: requerimientos del usuario no abordadas, velocidad del proyecto, pruebas de aceptación no superadas en la iteración anterior y tareas no terminadas en la iteración anterior. Todo el trabajo de la iteración es expresado en tareas de programación, cada una de ellas es asignada a un programador como responsable, pero llevadas a cabo por parejas de programadores.

Producción: La fase de producción requiere de pruebas adicionales y revisiones de rendimiento antes de que el sistema sea trasladado al entorno del cliente. Al mismo tiempo, se deben tomar decisiones sobre la inclusión de nuevas características a la versión actual, debido a cambios durante esta fase. Es posible que se rebaje el tiempo que toma cada iteración, de tres a una semana. Las ideas que han sido

propuestas y las sugerencias son documentadas para su posterior implementación (por ejemplo, durante la fase de mantenimiento).

Mantenimiento: Mientras la primera versión se encuentra en producción, el sistema se mantiene en funcionamiento al mismo tiempo que se desarrollan nuevas iteraciones. Para realizar esto se requiere de tareas de soporte para el cliente. De esta forma, la velocidad de desarrollo puede bajar después de la puesta del sistema en producción. La fase de mantenimiento puede requerir nuevo personal dentro del equipo y cambios en su estructura.

Muerte del Proyecto: Es cuando el cliente no tiene más requerimientos para ser incluidas en el sistema. Esto requiere que se satisfagan las necesidades del cliente en otros aspectos como rendimiento y confiabilidad del sistema. Se genera la documentación final del sistema y no se realizan más cambios en la arquitectura. La muerte del proyecto también ocurre cuando el sistema no genera los beneficios esperados por el cliente o cuando no hay presupuesto para mantenerlo.

APÉNDICE E
Casos de Uso

Los Casos de Uso no son parte del diseño (cómo), sino parte del análisis (qué). De forma que al ser parte del análisis ayudan a describir qué es lo que el sistema debe hacer. Los Casos de Uso son qué hace el sistema desde el punto de vista del usuario. Es decir, describen un uso del sistema y cómo este interactúa con el usuario.

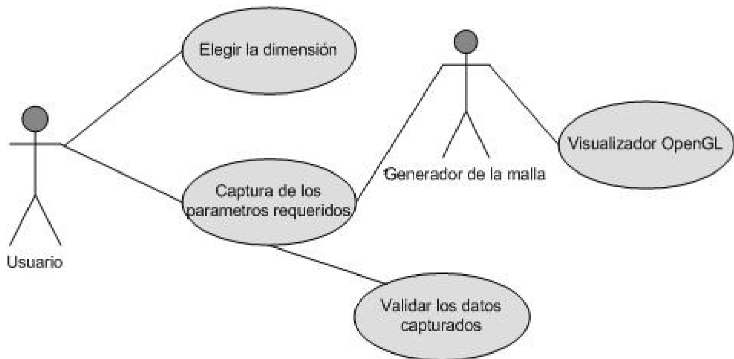
El documento que describe el caso de uso (use case), explica la forma de interactuar entre el sistema y el usuario. Cada caso de uso proporciona uno o más escenarios que indican cómo debería interactuar el sistema con el usuario o con otro sistema para conseguir un objetivo específico. Normalmente, en los casos de usos se evita el empleo de palabras técnicas, prefiriendo en su lugar un lenguaje más cercano al usuario final.

En otras palabras, un caso de uso es una secuencia de transacciones que son desarrolladas por un sistema en respuesta a un evento que inicia un actor sobre el propio sistema.

Un diagrama de casos de uso (*Use Case Diagram*) es una representación gráfica de parte o el total de los actores y casos de uso del sistema, incluyendo sus interacciones. Todo sistema tiene como mínimo un diagrama *Main Use Case*, que es una representación gráfica del entorno del sistema (actores) y su funcionalidad principal (casos de uso).

Caso de uso	Captura dimensión de una malla
Descripción	El sistema requiere tener la funcionalidad de permitir elegir la dimensión y capturar los parámetros de una malla y visualizarla.
Complejidad	Alta

Diagrama del Caso de Uso



Flujo Normal de Eventos

Flujo principal

El Usuario elige la dimensión

El Sistema muestra tres opciones:

1 Dimensión
2 Dimensión
3 Dimensión

Capturar de parámetros

El usuario selecciona una opción.

Inicia la captura de los datos de carácter obligatorio y numérico que se requieren dependiendo de la dimensión: para la generación de la malla; loa campos se muestran en la siguiente lista:

1 Dimensión
- Longitud en X
- Numero de puntos en X

2 Dimensión
- Longitud en X
- Numero de puntos en X
- Longitud en Y
- Numero de puntos en Y

3 Dimensión
- Longitud en X
- Numero de puntos en X
- Longitud en Y
- Numero de puntos en Y

Validación de los valores capturados

Creación de la malla

El Generador de la malla realiza los cálculos y creación de líneas, puntos y unión de los mismos, pasando los datos a formato (fundones) de OpenGL y visualiza.

Flujos alternos

Si el usuario no introduce algún valor.
El sistema pide al usuario que introduzca el valor correcto o que no debe haber campos vacíos.

El usuario puede cambiarse de pantallas, OpenGL, Equation y regresar a Mesh, si aún no ha realizado acciones vera las pantallas vacías, es decir sin ninguna información ni visual ni de parámetros capturados, de lo contrario visualizara la información ya generada o capturada.

Si la respuesta visual no fue la deseada el usuario pondrá tener la posibilidad de volver a proporcionar valores desde cualquier punto y visualizar.

Requerimientos Especiales

Primer Requerimiento

N/A

Precondiciones

Precondición

El usuario debe saber el problema que desea resolver y por que ecuación lo desea realizar. Análisis previo para comenzar.

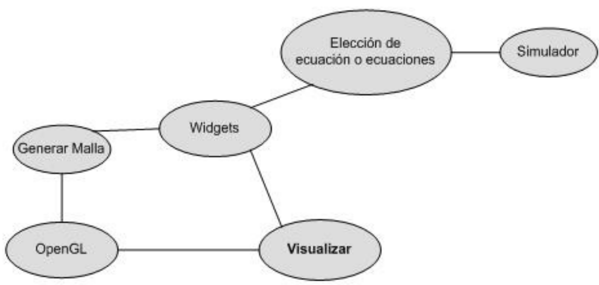
Post-condiciones

Post-condición

La información capturada permanece visible hasta que cambia de valores o mientras este en la aplicación. Si se generaron valores para visualizarlos en OpenGL, es decir, resultados podrán ser guardados en esté.

Caso de Uso	Cálculos de ecuaciones
Descripción	El sistema requiere tener la funcionalidad de permitir capturar valores de inicialización, para diferentes tipos de ecuaciones, y posibilidad de elegir entre una o varias.
Complejidad	Alta

Diagrama del Caso de Uso



Flujo Normal de Eventos

Flujo principal

El usuario captura los valores de inicialización de la malla, límites del problema.
El sistema muestra la malla en el visualizador con parámetros de OpenGL.
El Usuario elige una ecuación o varias a resolver, para continuar con la captura de los parámetros requeridos para dicha ecuación o ecuaciones, una por una. Tipo de ecuaciones:

- **Energy**
- **Presure**
- **Momentum X**
- **Momentum Y**
- **Momentum Z**

El sistema valida los campos capturados.
El sistema genera un archivo con los valores.
El sistema los envía al simulador numérico, realiza los cálculos, actualiza el archivo generado.
El usuario oprime el botón de compilar.
El sistema llama al script de compilación.
El usuario oprime el botón de ejecutar.
El sistema envía el resultado de la compilación un script para ser abierto el archivo generado en OpenDX y ejecutarse el visualizador de resultados que se genero por default.
Una vez visualizando los resultados en OpenDX el usuario puede manipularlos para visualizarlos según lo requiera, y trabajar con OpenDX de manera independiente a la aplicación.

Flujos Alternos

Si el usuario no introduce algún valor. El sistema pide al usuario que introduzca el valor correcto o que no debe haber campos vacíos. El usuario puede cambiarse de pantallas, OpenGL, Mesh y regresar a Equation, visualizara la información ya generada o capturada. Puede cambiar entre ecuaciones, cambiar los valores de cada parámetro, incluso los valores de la malla, así como dimensión y solo cuando este convencido de los parámetros de entrada, oprimir el botón de compilar y por último de ejecutar. Puede manipular la aplicación y generar otros cálculos, aún cuando tenga abierto visualizador de OpenGL. Si la respuesta visual no fue la deseada el usuario pondrá tener la posibilidad de volver a proporcionar valores desde cualquier punto y visualizar.

Requerimientos Especiales

Primer requerimiento Especial	N/A

Precondiciones

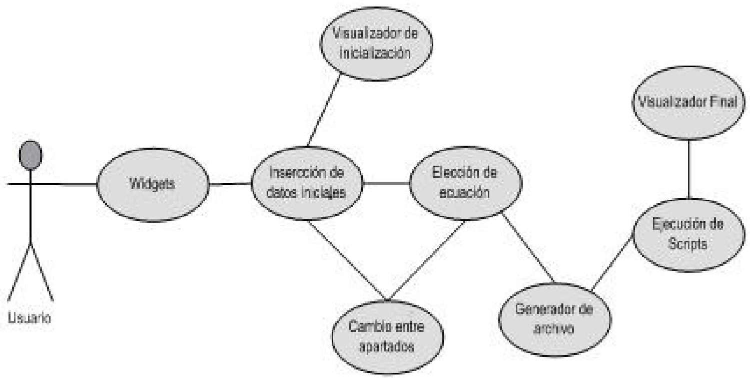
Precondición Uno	Debe haberse generado una malla, valores de inicialización.

Post-condiciones

Post-condición Uno	La información capturada permanece visible hasta que cambia de valores o mientras este en la aplicación. Si se generaron valores para visualizarlos en OpenGL, es decir, resultados podrán ser guardados en esté.

Caso de Uso	Creación de archivo y compilación
Descripción	El sistema requiere generar un archivo con los datos finales, el cual deberá ser compilado y visualizar los resultados en OpenDX, con la ventaja de manipular el software, OpenDX deberá ejecutarse de manera independiente a la aplicación.
Complejidad	Alta

Diagrama del Caso de Uso



Flujo Normal de Eventos

Flujo Principal

El usuario captura los parámetros correspondientes a cada ecuación que eligió.
El sistema muestra para cada ecuación sus parámetros correspondientes en diferentes paginas (pantallas), con la posibilidad de moverse entre ellas.

Ecuaciones:
- **Energy**
- **Presure**
- **Momentum X**
- **Momentum Y**
- **Momentum Z**

El sistema valida los campos capturados.
El sistema genera un archivo con los valores, toma los valores de inicialización (la malla generada), los coloca en un File, lo mismo con los parámetros capturados de la o las ecuaciones.
El usuario oprime el botón de compilar.
El sistema ejecuta un script el cual compila el File generado, con el uso de las clases del simulado, se generan los archivos que podrán ser leídos en OpenDX.
El usuario da clic en el botón de Ejecutar.
El sistema llama al siguiente script que envía el File y genera la estructura por default en OpenDX
OpenDX recibe el conjunto de Files generados de la compilación, crea la estructura de visualización por default y muestra la respuesta, visualización final.
El usuario manipula la respuesta, la modifica, cambia la visualización, trabaja con OpenDX, de manera independiente a la aplicación.
Puede regresar a la aplicación y modificar los valores, cambiar de dimensión y volver a generar otros datos para obtener otro resultado.

Flujos Alternos

Si el usuario no introduce algún valor.
El sistema valida que los campos sean correctos y que no se pase de los rangos requeridos, que no haya campos vacíos.

El usuario puede cambiarse entre OpenDX y la aplicación.
Si la respuesta visual no fue la deseada el usuario pondrá tener la posibilidad de volver a proporcionar valores desde cualquier punto y visualizar.

Diagramas de Estado

Los diagramas de estados muestran el comportamiento de los objetos, es decir, el conjunto de estados por los cuales pasa un objeto durante su vida, junto con los cambios que permiten pasar de un estado a otro; engloban todos los mensajes que un objeto puede enviar o recibir. En un diagrama de estados, un escenario representa un camino dentro del diagrama. Dado que generalmente el intervalo entre dos envíos de mensajes representa un estado, se pueden utilizar los diagramas de secuencia para buscar los diferentes estados de un objeto.

En todo diagrama de estados existen por lo menos dos estados especiales inicial y final: start y stop. Cada diagrama debe tener uno y sólo un estado start para que el objeto se encuentre en estado consistente. Por contra, un diagrama puede tener varios estados stop.

Un estado es una condición durante la vida de un objeto, de forma que cuando dicha condición se satisface se lleva a cabo alguna acción o se espera por un evento. El estado de un objeto se puede caracterizar por el valor de uno o varios de los atributos de su clase, además, el estado de un objeto también se puede caracterizar por la existencia de un enlace con otro objeto.

Los diagramas de estado resultan adecuados para describir el comportamiento de un objeto a través de diferentes casos de uso, sin embargo, no resultan del todo adecuados para describir el comportamiento que incluye a una serie de objetos colaborando entre sí. Por lo tanto, resulta útil combinar los diagramas de estado con otras técnicas como con los diagramas de secuencia.

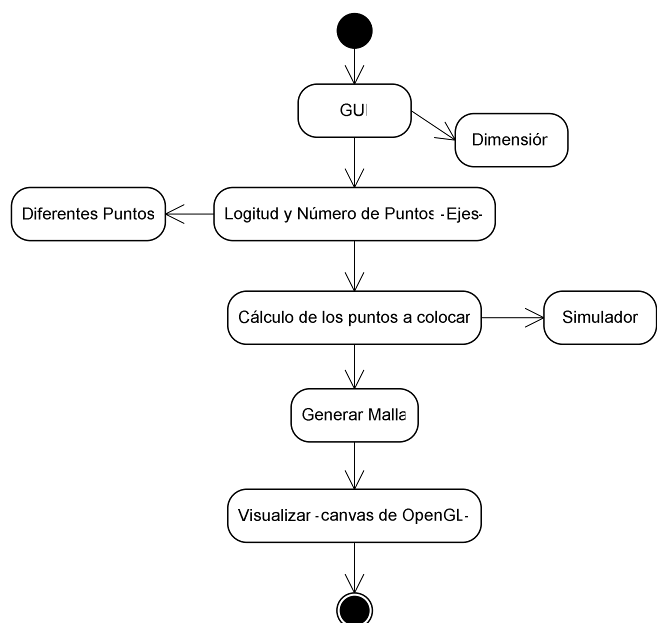


Diagrama de estado. Visualizar en OpenGL.

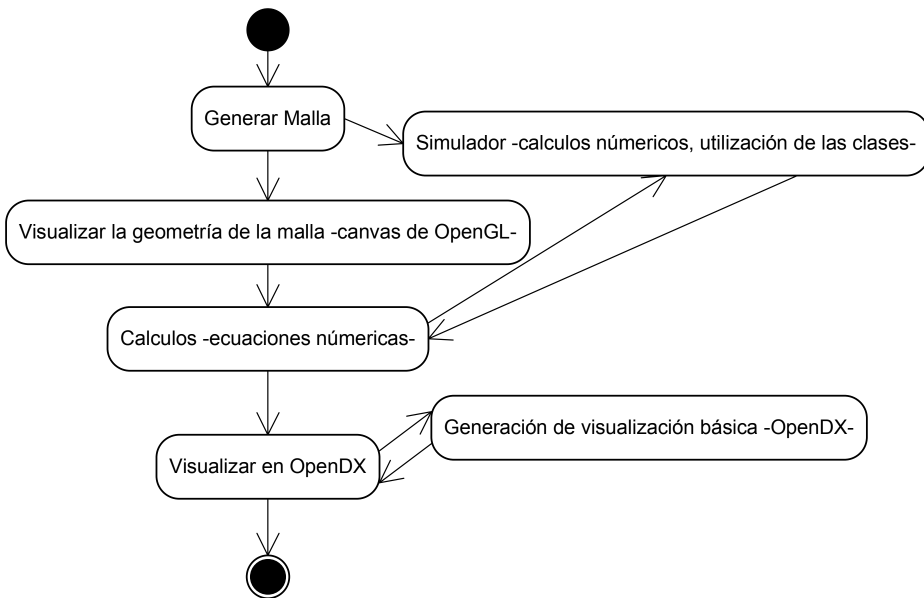


Diagrama de estado. Visualizar en OpenDX.

Diagramas de secuencia

Es uno de los diagramas más efectivos para modelar interacción entre objetos en un sistema. Típicamente uno examina la descripción de un caso de uso para determinar qué objetos son necesarios para la implementación del escenario. Muestra los objetos que intervienen en el escenario con líneas discontinuas verticales, y los mensajes pasados entre los objetos como vectores horizontales. Los mensajes se dibujan cronológicamente desde la parte superior del diagrama a la parte inferior; la distribución horizontal de los objetos es arbitraria.

A continuación se muestran el diagrama de secuencia del proceso general.

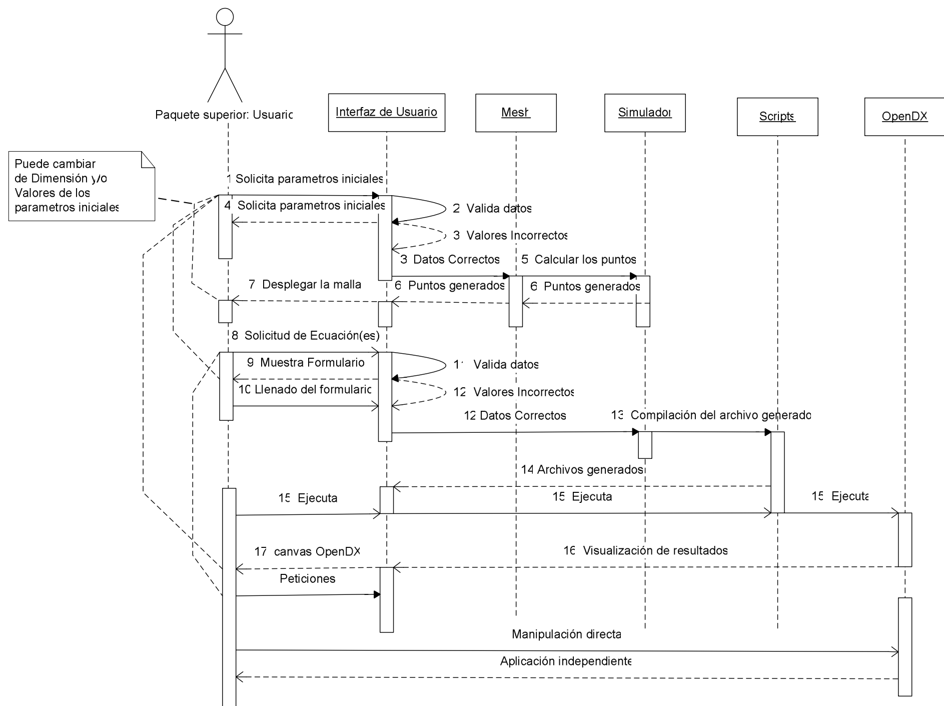
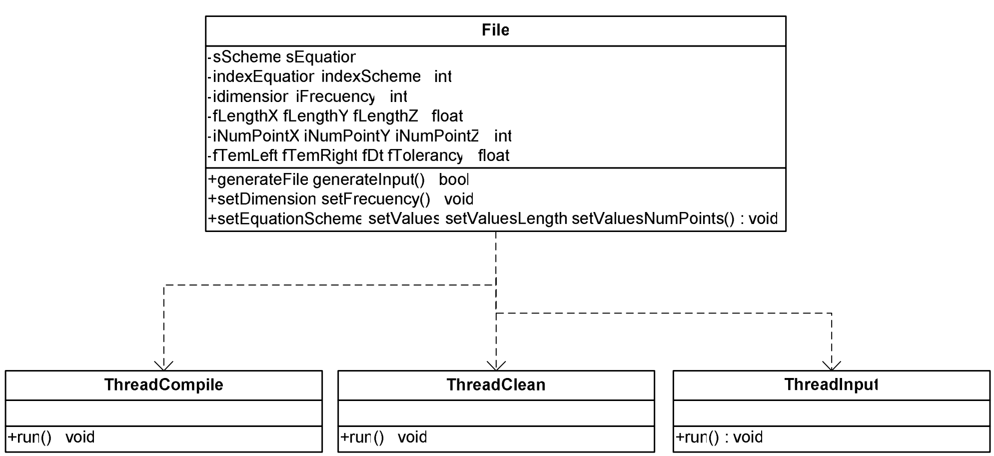


Diagrama de clases Modelo de implementación clases



Diagramas de clases.

APÉNDICE F
Instalación a partir de fuentes

OpenDX es un software de código abierto que se distribuye gratuitamente a través de su página http://www.opendx.org. En ese sitio es posible obtener versiones compiladas para varias de plataformas. Aunque es más simple instalar una versión compilada de OpenDX, muchas veces no se aprovecha al 100% las capacidades del equipo donde se realice la instalación. Por esta razón, es preferible hacer la instalación a partir de los archivos fuente. En la dirección http://www.opendx.org/compiling.html se pueden encontrar las instrucciones generales para configurar, compilar e instalar OpenDX en sistemas operativos tipo Unix. De cualquier forma es recomendable echarle un vistazo a los README para cualquier instalación que se desee hacer, en estos archivos encontraras mejor información.

Los pasos principales son:

Acción	Comando
Desempaquetar	tar -zxvf dx-4.3.2.tar.gz
Configurar	./configure
Compilar	Make
Instalar	make install

La instalación se realiza por omisión en el directorio /usr/local, de tal manera que el paso de instalación se debe hacer con privilegios de root. Sin embargo, cualquier usuario sin privilegios pues hacer la instalación en algún directorio donde tenga permisos de escritura.

Lo anterior se logra agregando la opción: --prefix=/path/to/install al comando de configuración, donde /path/to/install es cualquier directorio donde el usuario tenga permiso de escribir. El proceso de compilación es un poco tardado y depende de la velocidad de la máquina en la que se haga la instalación. En algunos casos puede tomar varias horas.

Es conveniente instalar también los ejemplos, que se obtienen del mismo sitio de donde se baja el código fuente de OpenDX. La instalación se hace siguiendo los pasos de configuración (./configure) e instalación (make install). En caso de que en la instalación de OpenDX se haya usado la opción --prefix, aquí también se debe usar, con el mismo directorio.

Una vez instalado OpenDX, la manera de ejecutarlo es mediante el comando: *dx* el cual despliega una ventana de inicio como la que se muestra en la **figura F.1**. La opción *Help* mostrará una descripción de cada uno de los botones del menú y como acceder a ellos directamente a través de la línea de comandos. Una ayuda del comando *dx* en línea se obtiene con las opciones *-h* o *-morehelp*, donde esta última muestra todas las opciones descritas en detalle a manera de manual de Unix.

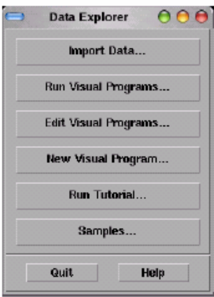


Figura *F.1.*: Ventana de inicio de OpenDX.

Instalación de bibliotecas auxiliares

Existen bibliotecas de uso general que se combinan con OpenDX para facilitar el manejo de distintos formatos de archivos. Algunos sistemas de simulación escriben su salida en estos formatos, por lo que es conveniente tener instaladas dichas bibliotecas.

Para utilizar las bibliotecas de manejo de formatos, éstas se deben instalar antes que OpenDX, para que éste último las reconozca en el momento de su configuración.

Algunas bibliotecas

NetCDF: NetCDF (Network Common Data Form) es una interfaz a una biblioteca para acceder a datos almacenados en forma de arreglos. NetCDF implementa un tipo de datos abstracto, lo cual significa que todas las operaciones para acceder y manipular datos, se deben realizar a través de un conjunto de funciones provistas por la interfaz. La biblioteca netCDF define un formato que es independiente del tipo de máquina para representar datos científicos. El software incluye interfaces para C y FORTRAN, y está disponible de manera gratuita en http://www.unidata.ucar.edu/.

La forma de instalar netCDF es similar ha como se instala OpenDX: desempacar *tar -zxvf netcdf-3.6.0-p1.tar.gz*; configurar *./configure*; compilar *make* y finalmente instalar *make install*.

Observaciones importantes: Aunque originalmente sólo se puede acceder a las funciones de netCDF a través de C y FORTRAN, existen algunas extensiones para C++ y FORTRAN 90 las cuales no son oficiales pero resultan útiles en algunos casos. Es conveniente entonces, tener instalado un compilador de C++ y FORTRAN 90 antes de instalar netCDF para que en la configuración de este último se tomen en cuenta dichas extensiones.

HDF: El proyecto HDF (Hierarchical Data Format) involucra el desarrollo y soporte de software y formatos de archivos para manejo de datos científicos. HDF incluye bibliotecas de entrada y salida de archivos, y herramientas para analizar, visualizar y convertir datos. Existen dos versiones: HDF4.x y HDF5. Estos formatos son completamente diferentes y NO compatibles.

El software HDF es desarrollado y soportado por la NCSA (National Center for Supercomputing Applications) y está disponible de forma gratuita en http://hdf.ncsa.uiuc.edu/. Es utilizado en todo tipo de aplicaciones.

OpenDX utiliza el formato HDF en su configuración inicial, es decir las versiones 4.x y anteriores, aunque existen módulos que se pueden adicionar para importar formatos HDF5, http://www-beams.colorado.edu/dxhdf5/.

La instalación de HDF se realiza similarmente que netCDF, siguiendo los tres pasos principales (configurar, compilar e instalar). Se recomienda utilizar *../configure* --prefix=/usr/local/hdf, dado que HDF utiliza cientos de encabezados y sus propias bibliotecas jpeg y zlib.

CDF: CDF (Common Data Format) es un formato auto descriptivo parar almacenar y manipular datos escalares y multidimensionales, independiente del área de aplicación y de la plataforma. CDF contiene, además de sus propias reglas para formatear datos, un conjunto de herramientas para manejar

datos científicos que se pueden utilizar desde lenguajes como C, FORTRAN y JAVA. Estas herramientas se conocen como la interfaz de CDF, y hacen transparente al usuario el formato interno de CDF. Por lo tanto, los programadores no se deben preocupar por realizar las acciones de entrada y salida de bajo nivel, esto lo hace CDF automáticamente.

La biblioteca CDF y la documentación para su instalación y uso se puede obtener en http://nssdc.gsfc.nasa.gov/cdf/cdfhome.html

Interfaz de OpenDX

El menú principal, que se abre con el comando **dx -full,** muestra opciones para ejecutar las siguientes interfaces.

- El **data prompter**, que provee de una GUI sencilla, permite importar una variedad de formatos de datos diferentes, incluyendo algunos específicos a ciertos programas. Abre con el comando **dx –prompter**
- El visualizador, con el comando **dx -image**, utilizado para ejecutar los programas previamente construidos, para observar los resultados finales
- El editor. Con el comando **dx -edit** se obtienen un sin fin de herramientas fáciles de manipular para elaborar un programa, importar datos y diseñar la red que conecta las entradas y las salidas deseadas.
- El constructor. Para diseñar módulos se utiliza **dx -builder**. Crea plantillas en C correspondientes a un programa básico para módulos.
- Tampoco falta un pequeño tutorial básico, que abre con **dx -tutor**.

El Proceso de Visualización con OpenDX

El proceso de visualización consiste en cinco pasos:

1. Elaborar, recoger o producir los datos a ser visualizados.

2. Investigar sobre la mejor manera de visualizar los datos. Buscar ejemplos. Formular un modelo visual que represente la visión de la presentación.
3. Preparar los datos para el modelo de datos de OpenDX y los detalles de su importación.
4. Diseñar el análisis visual y las transformaciones visuales requeridas para obtener la visión planeada.
5. Determinar y entender los requerimientos de la salida, desde el punto de vista externo.

Datos

Los datos pueden provenir de diferentes fuentes, como la recolección tomada en el campo o la simulación en computadora, o ambos.

Para describir el conjunto de datos, es necesario entender su forma básica. Esto se determina generalmente en las interrelaciones entre sus elementos.

Los datos pueden ser:

Dispersos: Cada componente tiene una localización y un valor pero no están conectados particularmente entre sí. Es decir, es meramente un conjunto de puntos en el mismo espacio de coordenadas.

Mayado regular: Consiste en un conjunto de puntos de intersección generado por líneas, planos, etc. regulares.

Mayado deforme regular: Consiste en un conjunto de puntos conectados de una manera regular pero siguiendo un esquema más complejo a linear.

Mayado irregular: Representa un conjunto de puntos interconectados pero sin asumir un patrón específico.

Dependencia de datos

La dependencia de datos tiene que ver con la relación entre la localización de los datos y su valor asociado. Los valores asociados con una localización particular o por un espacio particular definidos por un conjunto de puntos, se dice que es dependiente de la posición, o dependiente de las conexiones.

Esto es importante de definir, especialmente por la forma en que OpenDX asigna valores en localizaciones donde no se ha especificado ninguno.

Importar datos

La parte más difícil de usar un paquete de visualización es insertar los datos en el sistema. A diferencia de otros paquetes, que soportan sólo formatos estándares específicos, OpenDX tiene además facilidades programables por el usuario para ajustar diferentes formatos de datos.

Las opciones predefinidas por OpenDX son:

- Usar el Importador de Vectores General para crear una descripción de datos apropiada (con el data prompter).
- Usar el módulo ImportSpreadsheet de openDX.
- Usar el módulo ReadImage para leer en los formatos TIFF, MIFF, GIF y RGB.
- Usar el módulo Import de openDX para leer los datos con una estructura específica.
- Leer directamente los datos en el formato nativo de openDX.
- Leer directamente utilizando archivos del Importador de Vectores General.
- Leer directamente archivos NetCDF, CDF, HDF o CM (colormap).

Diseñar el análisis visual

Una vez importados los datos se procede a diseñar el análisis, de cualquiera de las siguientes maneras:

Usar la opción Visualizar datos del data prompter. Puede no producir la imagen deseada, pero es rápido y efectivo para la verificación inicial.

Crear un programa visual utilizando el Editor de Programas Visuales (VPE). La interfaz gráfica, que se revisará más adelante, permite crear y modificar programas visuales nativos a openDX.

Re-utilizar ejemplos de programas como punto de partida, para adecuarlo a las nuevas necesidades.

APÉNDICE G
Desarrollo de aplicaciones en QT

QT es una biblioteca en C++ para el desarrollo de interfaces gráficas de usuario. Adicionalmente a la librería de clases en C++, Qt incluye herramientas para crear aplicaciones rápidamente. El soporte de multiplataformas y de internacionalización de Qt asegura que las aplicaciones alcancen el más amplio mercado posible.

Como programar en Qt y C++ están fuera del alcance de esta tesis. Como referencia al respecto se puede consultar la bibliografía, ver *Qt programming*.

Se muestra el famoso “Hola Mundo” para familiarizarse con QT -solo para inicializarse-.

Hola Mundo

Este programa contiene apenas lo mínimo para tener una aplicación en Qt corriendo. La siguiente imagen muestra como se ve el programa.

```
/*****************************************************************
 * Programa con la librería QT
 * Ejercicio de Hello World con tres botones (uno para cerrar)
 * Arista Sánchez Lidia
 * 03/10/05
 * Tutorial QT
******************************************************************/

/* Tiene lo mínimo que se necesita para correr una aplicación en QT*/

/* Esta librería tiene la definición de la clase QApplication que aparecerá en todas las aplicaciones que se
realicen en QT*/
#include <qapplication.h>
/* Tiene la definición de la clase QPushButton*/
#include <qpushbutton.h>
/* Se utiliza clase QFont por ellos se agrega esta librería*/
#include <qfont.h>
/* Se utiliza la clase QVBox y se debe agregar esta librería*/
#include <qvbox.h>

int main( int argc, char **argv )
{

        /*Es esencial que el objeto QApplication sea creado antes de usar cualquier parte del sistema
de ventanas de Qt.*/
        QApplication a( argc, argv );

        /* En esta caja se agregaran los botones*/
        QVBox box;
        box.resize( 200, 120 );

        /* Esté botón es puesto para mostrar el texto "Hello world" y pertenece al QVBox (padre). En
un botón puede haber texto o un QPixmap*/
        QPushButton hello( "Hello world!", &box );
        /* Tipo de letra*/
        hello.setFont( QFont("Times", 18, QFont::Bold));

        QPushButton botton( "Botton 1", &box );
        botton.setFont(QFont( "Times", 12, QFont::Bold) );

        QPushButton close( "Close", &box );
        close.setFont( QFont( "Times", 12, QFont::Bold) );
        /* Se le da una acción (señal) al botón*/
        QObject::connect( &close, SIGNAL(clicked()), &a, SLOT(quit()) );

        /* Se agregan la caja a la ventana*/
        a.setMainWidget( &box);
        /* Se hace visible la caja. Un widget no es visible cuando es creado. Debes llamar la función
show( ) para hacerlo visible. */
        box.show();
       /* Aquí es donde main( ) pasa el control a Qt, y exec( ) regresará cuando la aplicación termine.
En exec( ), Qt recibe y procesa los eventos del usuario y del sistema y los pasa al widget apropiado. Se
ejecuta la ventana*/
       return exec(); }
```

Compilación

Para compilar una aplicación C++ necesitas crear un makefile. La manera más fácil de crearlo para Qt es usar la herramienta qmake proveída por Qt. Si guardaste el programa en un archivo llamado helloworld.cpp en su propio directorio, todo lo que tienes que hacer es:

```
qmake –project
qmake
```

El primer comando le dice a qmake que cree un archivo .pro (un archivo de proyecto). El segundo comando le dice que cree un makefile (dependiente de plataforma) basado en tu archivo de proyecto. Ahora debes poder teclear make y luego ejecuta tu primera aplicación hecha en QT.

Ejemplo 2

```
#include <qapplication.h>
#include <qpushbutton.h>
#include <qfont.h>

int main( int argc, char **argv )
{

QApplication a( argc, argv );

QPushButton quit( "Salir", 0);
quit.resize(75, 30)
/*Es possible cambiar la fuente de texto por omisión (usando QApplication::setFont()) para toda la aplicación.*/
quit.setFont(QFont( "Times", 18, QFont::Bold) );

QObject::connect( &quit, SIGNAL(clicked()), &a, SLOT(quit() ) );

a.setMainWidget(&quit);
quit.show();

return a.exec( );
}
```

Connect( ) es quizá la característica central de Qt. Observa que connect ( ) es una función estática en QObject. No confundirla con la función connect de la biblioteca de socket.

Esta línea establece una conexión de una dirección entre dos objetos de Qt (Objetos que heredan directa o indirectamente de QObject). Todos los widgets son objetos de Qt. Ellos heredan de QWidget que a su vez hereda de QObject. Cada objeto de Qt puede tener señales (para mandar mensajes) y slots (para recibir mensajes).

Aquí, la señal clicked( ) de quit es conectada al spot quit( ) de a, así cuando el botón es presionado, la aplicación termina.

APÉNDICE H
Comandos básicos de “vi”

Vi es un editor de texto para consola, sus siglas se derivan de “visual”. Es el editor de texto tradicional de UNIX, y en muchos sistemas es el único disponible, de manera que es importante saber usarlo, aunque sea básicamente.

En vi existen dos modos de trabajo:

- un modo de edición: todo lo que ingresemos será texto del archivo.

- un modo de comandos: todo lo que ingresemos no será texto del archivo. A veces lo que escribamos no mostrará ninguna salida inmediata en la pantalla.

Al iniciar el programa, estamos en el modo de comandos. Para ingresar al modo de edición debemos apretar `i`, o bien, Insert. Para volver al modo de comandos, utilizamos la tecla ESC.

Insertar texto	
vi archivo	Abre el archivo para capturar más información
Vi	Capturar en archivo nuevo
I	Antes del cursor
I	Al inicio de la línea
A	Al final de la línea

Cambiar texto	
Cw	Cambia una palabra
Cc	Cambia una línea
C	Cambia desde el cursor hasta el final de la línea
R	Cambia el carácter donde se encuentra el cursor
R	Cambia el texto desde donde se encuentra el cursor

Teclas de movimiento	
0	Inicio de línea
B	Palabra anterior
H	Izquierda
K	Arriba
G	Fin de archivo
$	Fin de línea
W	Próxima palabra
L	Derecha
J	Abajo

Borrar	
Dw	Borra una palabra
Dd	Borrar 1-n líneas. Ej. 10dd
X	Borra el carácter donde se encuentra el cursor
X	Borra el carácter a la izquierda del cursor
D	Borra desde el cursor hasta el final de la línea
Dw	Corta la próxima palabra
d$	Corta hasta el final de la línea
P	Pega lo que se haya cortado o copiado después del cursor
P	Pega lo que se haya cortado o copiado antes del cursor
U	Deshacer la última acción
Yy	Copia la línea
X	Corta el carácter

Última línea	
w nom_archivo	Graba el archivo con ese nombre
W	Graba el archivo
Q	Sale del vi sólo si ya se guardaron los cambios
q!	Sale del vi sin guardar los cambios
Wq	Guarda el archivo y sale del vi

R	Leer un archivo
E	Editar un archivo
A	Se coloca en el carácter siguiente al actual, y en modo inserción

Otras funciones	
?	Busca una cadena hacia atrás
nY o yy	Copia "n" líneas a partir del cursor
nG	Se mueve a la línea n
ZZ	Graba el archivo y sale del editor
?texto o /texto	Nos lleva una por una a la palabra texto que exista en el archivo (movimientos n a la siguiente aparición y N a la anterior)
<desde>,<hasta>s/<buscar>/<reemplazar>/g Ejemplo: :1,$s/texto/Texto/g	`Significa: <desde>, <hasta>` indican líneas en el archivo; `<buscar>` y `<reemplazar>` son cadenas de caracteres o expresiones regulares; `/` es un separador, `s` (sustituir) y `g` (global) son letras de comando para el manejo de expresiones regulares.

APÉNDICE I
Makefile

Este es un buen momento para presentar la idea de **make** y **Makefile**. Probablemente haya oído hablar de **scripts** o **lenguajes de scripting**. En general estos son lenguajes en los que tecleas unos comandos en un fichero y entonces ejecutas el fichero sin ninguna compilación. Por ejemplo, en lugar de tres comandos para compilar, podríamos escribir los mismos tres comandos en un fichero, y entonces ejecutar el fichero en su lugar. Esto sería básicamente un script.

Pero ¿y si hacemos algún cambio mínimo, como el nombre de un fichero fuente o incluso el nombre del compilador? Tendríamos que cambiar la mitad de ese script para hacer que las cosas funcionaran otra vez. Bien, con un **Makefile** las cosas son un poco diferentes.

Un archivo Makefile es un archivo de texto en el cual se distinguen cuatro tipos básicos de declaraciones:

- Comentarios.
- Variables.
- Reglas explícitas.
- Reglas implícitas.

Comentarios

Al igual que en los programas, contribuyen a un mejor entendimiento de las reglas definidas en el archivo. Los comentarios se inician con el carácter #, y se ignora todo lo que continúe después de ella, hasta el final de línea.

Este es un comentario

Variables

Se definen utilizando el siguiente formato:

Nombre = dato

De esta forma, se simplifica el uso de los archivos `Makefile`. Para obtener el valor se emplea la variable encerrada entre paréntesis y con el carácter *`$`* al inicio, en este caso todas las instancias de *`$(nombre)`* serán reemplazadas por *`dato`*. Por ejemplo, la siguiente definición.

SRC = main.c

Origina la siguiente línea:

gcc $(SRC)

y será interpretada como:

gcc main.c

Sin embargo, pueden contener más de un elemento *dato*. Por ejemplo:

```
objects = programa_1.o programa_2.o programa_3.o \
          programa_4.o programa_5.o

  programa: $(objects)
  gcc -o programa $(objects)
```

Hay que notar que `make` hace distinción entre mayúsculas y minúsculas.

Reglas explícitas

Estas le indican a `make` qué archivos dependen de otros archivos, así como los comandos requeridos para compilar un archivo en particular. Su formato es:

```
archivoDestino: archivosOrigen
comandos  # Existe una caracter TAB (tabulador) antes de cada comando.
```

Esta regla indica que, para crear *archivoDestino*, `make` debe ejecutar *comandos* sobre los archivos *archivosOrigen*. Por ejemplo:

```
main: main.c funciones.h
gcc -o main main.c funciones.h
```

Significa que, para crear el archivo de destino *main*, deben existir los archivos *main.c* y *funciones.h* y que, para crearlo, debe ejecutar el comando:

gcc -o main main.c funciones.h

Reglas implícitas

Son similares a las reglas explícitas, pero no indican los comandos a ejecutar, sino que `make` utiliza los sufijos (extensiones de los archivos) para determinar que comandos ejecutar. Por ejemplo:

funciones.o: funciones.c funciones.h

origina la siguiente línea:

$(CC) $(CFLAGS) -c funciones.c funciones.h

Existe un conjunto de variables que se emplean para las reglas implícitas, y existen dos categorías: aquellas que son nombres de programas (como CC) y aquellas que tienen los argumentos para los programas (como CFLAGS). Estas variables son provistas y contienen valores predeterminados, sin embargo, pueden ser modificadas, como se muestra a continuación:

```
CC = gcc
CFLAGS = -Wall -O2
```

En el primer caso, se ha indicado que el compilador que se empleará es `gcc` y sus parámetros son *`-Wall -O2`*.

Un ejemplo de un archivo makefile completo, donde se incluyen todas los tipos de declaraciones. En este ejemplo, se utiliza el programa `make` para compilar los programas *`funciones.c`* y *`main.c`* para crear un ejecutable llamado *`main`*. Tanto *`funciones.c`* como *`main.c`* utilizan el archivo *`funciones.h`*.

```
# La siguiente no es necesariamente requerida, se agrega para
# mostrar como funciona.

.SUFFIXES: .o .c
.c.o:
   $(CC) -c $(CFLAGS) $<

# Macros

CC = gcc
CFLAGS = -g -Wall -O2
SRC = main.c funciones.c funciones.h
OBJ = main.o funciones.o

# Reglas explícitas

all: $(OBJ)
   $(CC) $(CFLAGS) -o main $(OBJ)

clean:
   $(RM) $(OBJ) main

# Reglas implícitas

funciones.o: funciones.c funciones.h
main.o: main.c funciones.h
```

El compilador de C++ creará el archivo compilado, enlazado y listo para usar **a.out**. Pero, ¿Y si tenemos múltiples proyectos? ¿Se llamarán todos **a.out**? Bueno, afortunadamente tenemos la opción del compilador **-o** que especifica el nombre del fichero de salida.

Algunos comandos de línea de órdenes; Make se puede invocar de la siguiente forma:

make [-f makefile] [opciones] [nombres]

Estas son algunas de las opciones que se pueden usar con make

Opción	Descripción
-f fichero	Usar "fichero" en lugar del Makefile por defecto
-p	Imprime un conjunto de macros y definiciones de destinos
-s	Modo silencioso. No imprime las líneas antes de ejecutarlas
-u	Incondicional. Construye todos los destinos

APÉNDICE J
Comandos básicos de Linux

Los comandos son esencialmente los mismos que cualquier sistema UNIX. En la tablas 1 y 2 se muestran una lista de comandos mas frecuentes. En la tabla 3 se tiene una lista de equivalencias entre comandos Unix/Linux y comandos DOS.

Comando/Sintaxis	Descripción	Ejemplos
cd [*dir*]	Cambia de directorio	cd /tmp
chmod *permisos fich*	Cambia los permisos de un archivo	chmod +x miscript
chown *usuario:grupo fich*	Cambia el dueño un archivo	chown nobody miscript
cp *fich1...fichN dir*	Copia archivos	cp foo foo.backup
file *arch*	Muestra el tipo de un archivo	file arc_desconocido
find *dir test acción*	Encuentra archivos.	find . -name ``.bak" – print
grep [-*cilnv*] *expr archivos*	Busca patrones en archivos	grep mike /etc/passwd
mkdir *dir*	Crea un directorio.	mkdir temp
mv *fich1 ...fichN dir*	Mueve un archivo(s) a un directorio	mv a.out prog1
mv *fich1 fich2*	Renombra un archivo.	mv .c prog_dir
less / more *fich(s)*	Visualiza página a página un archivo.	more muy_largo.c
ln [-*s*] *fich acceso*	Crea un acceso directo a un archivo	ln -s /users/mike/.profile .
Ls	Lista el contenido del directorio	ls -l /usr/bin
Pwd	Muestra la ruta del directorio actual	Pwd
rm *fich*	Borra un fichero.	rm foo.c
rm -r *dir*	Borra un todo un directorio	rm -rf prog_dir
rmdir *dir*	Borra un directorio vacío	rmdir prog_dir
vi *fich*	Edita un archivo.	vi .profile

Tabla 1. Comandos Linux/Unix de manipulación de archivos y directorios.

Comando/Sintaxis	Descripción	Ejemplos
cal [[*mes*] *año*]	Muestra un calendario del mes/año	cal 1 2025
date [*mmddhhmm*] [+*form*]	Muestra la hora y la fecha	Date
echo *string*	Escribe mensaje en la salida estándar	echo ``Hola mundo"
finger *usuario*	Muestra información general sobre	finger nn@maquina.aca.com.co

Id	Número id de un usuario	id usuario
kill [-*señal*] *PID*	Matar un proceso	kill 1234
man *comando*	Ayuda del comando especificado	Man gcc
		Man -k printer
Passwd	Cambia la contraseña.	Passwd
ps [*axiu*]	Muestra información sobre los procesos	ps –ux
	que se están ejecutando en el sistema	ps –ef
who / rwho	Muestra información de los usuarios	Who
	conectados al sistema.	

Tabla 2. Comandos Linux/Unix más frecuentes.

Linux	DOS	Significado
Cat	type	Ver contenido de un archivo.
cd, chdir	cd, chdir	Cambio el directorio en curso.
chmod	attrib	Cambia los atributos.
clear	cls	Borra la pantalla.
ls	dir	Ver contenido de directorio.
mkdir	md, mkdir	Creación de subdirectorio.
more	more	Muestra un archivo pantalla por pantalla.
mv	move	Mover un archivo o directorio.
rmdir	rd, rmdir	Eliminación de subdirectorio.
rm -r	deltree	Eliminación de subdirectorio y todo su contenido.

Tabla 3. Equivalencia de comandos Linux/Unix y DOS.

GLOSARIO

Comando: Medio por el cual se le ordena una acción determinada al sistema operativo a través de un intérprete.

Compilar: Proceso por el cual se "traduce" un programa escrito en un lenguaje de programación a lo que realmente entiende el ordenador.

Copyleft: es la forma general de hacer un programa software libre y requiere que todas las modificaciones y versiones extendidas del programa sean también software libre.

CSS: Las hojas de estilo en cascada (*Cascading Style Sheets*,) son un lenguaje formal usado para definir la presentación de un documento estructurado escrito en HTML o XML (y por extensión en XHTML).

Distribución: Un sistema operativo (en general Linux), que se ha empaquetado para facilitar su instalación.

Eficiencia: Capacidad administrativa de producir el máximo de resultados con el mínimo de recursos, el mínimo de energía y en el mínimo de tiempo posible.

FIDAP (Advanced Materials Physics): es una herramienta de modelación para modelar fluidos no newtonianos. Basado en el método de elemento finito, ofrece modelos numéricos y físicos para un amplio rango de flujos, ya sean laminares o turbulentos; esta disponible en todas las plataformas Unix/Linux y Windows tanto en serie como paralelo.

FLUENT (Flow Modeling Software): es un CFD (Computational fluid dynamics), software para sim ular problemas de flujo de fluidos, tiene la capacidad de utilizar modelos físicos comprensibles e incompresible, no viscosos o viscosos, laminares o turbulento, etc.

Flujo: fluido desplazándose en una superficie.

Framework: En el desarrollo de software, un framework es una estructura de soporte efinida en la cual otro proyecto de software puede ser organizado y desarrollado. Típicamente, puede incluir soporte de programas, bibliotecas y un lenguaje de scripting entre otras herramientas para ayudar a desarrollar y unir los diferentes componentes de un proyecto.

GNOME: Es un entorno de escritorio para sistemas operativos de tipo Unix bajo tecnología XWindows.

GNU: Gnu is Not Unix. Fue iniciado por Richard Stallman con el objetivo de crear un sistema operativo completamente libre. El 27 de septiembre de 1983 se anunció públicamente en el grupo de noticias net.unix-wizards.

GPL: General Public License. Una de las mejores aportaciones de la FSF. En la Licencia Pública General el autor conserva los derechos de autor (copyright) y permite la redistribución y modificación bajo términos diseñados para asegurarse de que todas las versiones modificadas del software permanecen bajo los términos más restrictivos de la propia GNU GPL. Esto hace que no sea imposible crear un producto con partes no licenciadas GPL: el conjunto tiene que ser GPL.

Se permite la copia y distribución de copias literales de esta licencia, pero no está permitido modificarla. **Preámbulo** La Licencia Pública General GNU es una licencia libre, de copia permitida (copyleft) para software y otros tipos de trabajos.

Implementar: Poner en funcionamiento, aplicar métodos, medidas, etc., para llevar algo a cabo.

Información: Se refiere a los datos a los que se les ha dado una forma que tiene sentido y es útil para los humanos.

Interfaz de usuario: es la manera en que los usuarios pueden interaccionar o comunicarse con la computadora. Existen dos tipos: interfaces alfanuméricas e interfaces gráficas de usuario (IGU).

Interfaz Gráfica de Usuario: es un método para poder interaccionar el usuario con el ordenador a través de imágenes y objetos como iconos, menús, ventanas, etc. además de texto. **GUI** (por sus siglas en inglés, *Graphical User Interface)*.

Interfaz: medio que permite la interconexión de dos procesos diferenciados con un único propósito común.

LGPL: La Licencia Pública General Reducida de GNU, o GNU LGPL para abreviar. Es una licencia de software libre, pero no tiene un *copyleft* fuerte, porque permite que el software se enlace con módulos no libres. Sólo la recomendamos para circunstancias especiales. Entre la versión 2 y la 2.1, la GNU LGPL cambió su nombre de "Licencia Pública General para Bibliotecas de GNU" a "Licencia Pública General Reducida de GNU", pues no es sólo para bibliotecas. Además la GNU GPL es habitualmente más apropiada para las bibliotecas (N. del T.: en inglés ambas expresiones tienen las mismas siglas: LGPL).

Librerías: Se refiere al conjunto de rutinas que realizan las operaciones usualmente requeridas por los programas.

Linux: Sistema operativo compuesto de las herramientas GNU de la FSF y el núcleo desarrollado por Linus Torvalds (Linus Benedict Torvalds nacido el 28 de Diciembre de 1969 en Helsinki, es un ingeniero de software finlandés), y sus colaboradores.

Makefile: Es un archivo de texto en el cual se distinguen cuatro tipos básicos de declaraciones: Comentarios, Variables, Reglas explícitas y Reglas implícitas.

Mesh (Malla): Las mallas (en inglés *mesh* o *grid*) son usadas en diferentes ámbitos. Es posible encontrar mallas en aplicaciones de realidad virtual, sistemas de información geográfica, sistemas físicos, etc. Existen básicamente dos tipos de mallas: *estructuradas* y *no-estructuradas*. Dependiendo del tipo de aplicación se usa uno u otro tipo de malla. Una malla es, a grandes rasgos, un conjunto de puntos conectados de alguna manera. Para definir una malla en el espacio (2D y 3D) se debe tener:

- Los puntos: coordenadas en 2D o 3D
- Los elementos: conectividad entre los puntos

La conectividad describe los elementos que conforman la malla

Descripción: En el caso del estudio de algunos sistemas físicos, y particularmente en dinámica de fluidos, la construcción de mallas sobre el dominio de estudio es un problema complicado. La construcción de dichas mallas es usando un mapeo entre la región irregular (dominio físico) y un cuadrado unitario (dominio computacional).

Metodología estructurada: la orientación de esta metodología se dirige hacia los procesos que intervienen en el sistema a desarrollar, es decir, cada función a realizar por el sistema se descompone en pequeños módulos individuales. Es más fácil resolver problemas pequeños, y luego unir cada una de las soluciones, que abandonar un problema grande.

Metodología orientada a objetos: a diferencia de la metodología mencionada anteriormente, ésta no comprende los procesos como funciones sino que arma módulos basados en componentes, es decir, cada componente es independiente del otro. Esto nos permite que el código sea reutilizable. Es más fácil de mantener porque los cambios están localizados en cada uno de estos componentes.

Metodología: Se refiere al conjunto de los pasos y métodos sistemáticos de investigación en una ciencia, que guían o deberían guiar una investigación

Modelo Vista Controlador (MVC): es un patrón de arquitectura de software (de tres niveles) que separa los datos de una aplicación, la interfaz de usuario, y la lógica de control en tres componentes distintos.

Modelo: Componente del MVC. Esta es la representación específica del dominio de la información sobre la cual funciona la aplicación, es otra forma de llamar a la capa de dominio.

Multitarea: Capacidad de un sistema para el trabajo con varias aplicaciones al mismo tiempo.

Multiusuario: Capacidad de algunos sistemas para ofrecer sus recursos a diversos usuarios conectados a través de terminales.

OpenDX: es un sistema de visualización de propósito general, que permite leer datos de distintas fuentes de una manera flexible y amigable. Aun cuando la organización de los datos puede ser muy diferente para cada fuente, OpenDX ofrece una manera de leer casi cualquier tipo de datos a través su herramienta conocida como Data Prompter.

OpenGL: biblioteca gráfica; para generar gráficos 2D y 3D por hardware.

Patrón de diseño: Es una solución a un problema de diseño no trivial que es efectiva y reusable, es decir, se puede aplicar a diferentes problemas de diseño en distintas circunstancias.

PHOENICS: es un software de propósito general que predice cuantitativamente cómo se comportan los fluidos (aire, agua, vapor, aceite, sangre, etc.) en o alrededor de motores, quipos de proceso, edificios, seres humanos, lagos, ríos, océanos, etc.

Planificación: ideamos un planeamiento detallado que guíe la gestión del proyecto, temporal y económicamente.

Programación orientada a objetos: La POO es un paradigma de programación que define los programas en términos de "clases de objetos". Se trata de una filosofía de programación donde la realidad se modela mediante objetos y la interacción entre ellos.

Puesta en producción: nuestro proyecto entra en la etapa de definición, allí donde se lo presentamos al cliente o usuario final, sabiendo que funciona correctamente y responde a los requerimientos solicitados en su momento. Esta etapa es muy importante no sólo por representar la aceptación o no del proyecto por pare del cliente o usuario final sino por las múltiples dificultades que suele presentar en la práctica, alargándose excesivamente y provocando costos no previstos.

RealFlow: es una herramienta de simulación de partículas en un entorno 3D basado en técnicas dinámicas de fluidos, es capaz de importar/exportar información de software de animación 3D como LightWave 3D, 3D Studio Max y Softimage. Se puede conseguir simular movimientos de líquidos, gases, lodo, sustancias viscosas, lava y otras sustancias pegajosas.

Reglas del negocio: Es el conjunto de restricciones, políticas, validaciones, fórmulas de cálculo, maneras implícitas de trabajo, suposiciones y declaraciones que forman parte de una organización.

Slots y Signals: Son mecanismos de comunicación entre objetos, esta es la principal característica de Qt y es el rasgo que hace distintas las librerías Qt del resto de herramientas para la elaboración de GUI, es un mecanismo de comunicación seguro, flexible y totalmente orientado a objetos y por supuesto implementado en C++.

La forma genérica de esta función es la siguiente:

```
connect( &a, SIGNAL (signal()), &b, SLOT(slot()));
```

El primer miembro &a es un puntero al objeto que emite la señal, el segundo indica la señal (la función signal()) que emite el objeto, el tercer miembro es un puntero al objeto que recibe la señal y que tendrá como función miembro el slot asociado a la señal (slot()).

Software libre es un asunto de libertad, no de precio. Se refiere a la libertad de los usuarios para ejecutar, copiar, distribuir, estudiar, cambiar y mejorar el software. De modo más preciso, se refiere a cuatro libertades de los usuarios del software:

- La libertad de usar el programa, con cualquier propósito (libertad 0).
- La libertad de estudiar cómo funciona el programa, y adaptarlo a tus necesidades (libertad 1). El acceso al código fuente es una condición previa para esto.
- La libertad de distribuir copias, con lo que puedes ayudar a tu vecino (libertad 2).
- La libertad de mejorar el programa y hacer públicas las mejoras a los demás, de modo que toda la comunidad se beneficie. (libertad 3). El acceso al código fuente es un requisito previo para esto.

Software libre no significa no comercial. Un programa libre debe estar disponible para uso comercial, desarrollo comercial y distribución comercial.

GNU copylef: para proteger de modo legal las libertades del software libre para todos, aunque existe software libre sin Copyleft.

Open source [código abierto] para designar algo parecido (pero no idéntico) a Software Libre.

Software de código fuente abierto describe una categoría de licencias de software casi, *pero no completamente,* igual que "software libre". La gente quien decide el significado de "software de código fuente abierto" ha aceptado una licencia que tiene restricciones inaceptables.

Toolkit: Sistemas de desarrollo multiplataforma.

UML: Lenguaje Unificado de Modelado, es un lenguaje gráfico para visualizar, especificar, construir y documentar un sistema de software.

Vista: Componente del MVC que presenta el modelo en un formato adecuado para interactuar, usualmente un elemento de interfaz de usuario.

Visualización científica: Es la generación de imágenes a partir de datos, para transformarlos en información y ganar entendimiento; así como para adecuar las limitaciones ópticas del ser humano y para comunicarlo a otros.

Visualización: Es el proceso de mapear valores numéricos en dimensiones conceptuales.

Widget: Es un objeto de interfaces de usuario que puede procesar entradas del usuario y dibujar gráficos. El programador puede cambiar la apariencia y el comportamiento y muchas propiedades menores como el color, hasta el contenido del widget.

“Lo maravilloso de aprender algo, es que nadie puede arrebatárnoslo.”
B.B. King

BIBLIOGRAFÍA

BIBLIOGRAFÍA

LIBROS

- Dalheimer, Kalle Matthias. *Programming with QT 2ª Ed*; O'Reilly, 2002.
- Blanchette J.; Summerfield M.. C++ GUI Programming with QT 3; Prentice Hall PTR
- Kempf Renate. *OpenGL Reference Manual: The Official Reference Document to OpenGL*, Version 1.1; Addison-Wesley, 1997.
- Booch G., Rumbaugh J., Jacobson I. *El Proceso Unificado de Desarrollo de Software.* Addison-Wesley, Madrid, 2000.
- Shneiderma, Ben. *Designing the usr interface: strategies for effective human-computer intraction.* 4thed. Madrid Pearson Education, 2006.
- Laurel, Brenda.*The art of human-computer interface design*. Reading, Massachesetts: Addison-wesley, 1990.
- Spolsky, Joel. *User interface design for programmers.* Berkeley, California: Apress, 2001.

TESIS

- Sosa, Armando. *Semiótica y Retórica Visual Aplicada al diseño de Interfases: La Metáfora como elemento de Navegación.*
- De la Cruz S., Luis Miguel. *Cómputo Paralelo en la Solución Numérica de las Ecuaciones de Balance en Flujo Turbulento.* Julio 2005; Capítulos: 2-6.
- Catalán Vega Marcos. *Metodologías de evaluación de Interfaces Gráficas de Usuario*.

DOCUMENTOS

- MARTÍN SANDE. *Programación en C++ con Qt bajo Entorno GNU/Linux* [En línea]. Linux User. <http://www.gluca.com.ar> [Consulta: Julio 2006].
- Greenberg, Ilan. "*acing Up to New Intrfaces. IEEE*". Computer, Abril 1999. P. 14-16.
- *"Estándares y guías de estilo; Interacción persona-ordenador"*. PDF.
- Del Corte Vladimir. "*Informe sobre librerias QT*". GVA-ELAI-UPM.
- Dorado de la Calle, Julian; Jaspe Villanueva, Alberto. *"Una aproximación a OpenGL"*.
- *"Graphics Programming with OpenGL*". Tutorial and Reference Manual Toby Howard Alan Murta Department of Computer Science University of Manchester V1.0, Jnuary 28, 1999

DOCUMENTACIÓN ELECTRÓNICA (WEB).

Diseño de Interfaces Gráficas de Usuario [En línea].

- <http://www.cs.cinvestav.mx/CursoVis/prinvisual.html> [Consultado: Diciembre 2005]
- <http://www.hipertexto.info/documentos/interfaz.htm> [Consultado: Enero 2006]
- <http://www.albertolacalle.com> [Consultado: Mayo 2006]

- <http://www.alzado.org> [Consultado: Mayo 2006]
- <http://www.ilustrados.com/publicaciones/EpZVVZkAVFDnjetyop.php> [Consultado: Septiembre 2006]
- <http://peterpan.uc3m.es/docencia/p_s_ciclo/iu/trasparencias.html> [Consultado: Febrero 2006]
- <http://www.uag.mx/66/proceso2.htm> [Consultado: Febrero 2006]
- <http://www.monografias.com/trabajos10/diusuar/diusuar.shtml> [Consultado: Febrero 2006]
- <http://griho.udl.es/ipo/libroe.html>[Consultado: Febrero 2006]
- <http://www.comminit.com/la/modelosdeplaneacion/modelos2004/modelosplaneacion-18.html> [Consultado: Febrero 2006]
- <http://melody.syr.edu/pzhang/t_hci_bk97.cgi>[Consultado: Febrero 2006]
- <http://www.cdli.ca/~elmurphy/emurphy/laurel.html>[Consultado: Febrero 2006]
- <http://sigchi.org/cdg/cdg2.html#2_1> [Consultado: Febrero 2006]
- <http://www.albertolacalle.com/diseno-visual.htm> [Consultado: Febrero 2006]

UML

- <http://es.wikipedia.org/wiki/Lenguaje_Unificado_de_Modelado> [Consulado: Febrero 2006]
- <http://oness.sourceforge.net/docbook/oness.html#metodologia> [Consulado: Febrero 2006]

Diseño de Iconos [En línea].

- <http://www.icon-maker.com/> [Consultado: Febrero 2006]
- <http://programas.navegalis.com/programas/4-0.php> [Consultado: Febrero 2006]
- <http://platea.cnice.mecd.es/~jmas/manual/html/textos_impecables.html> [Consultado: Febrero 2006]

Documentación de las diferentes API para la construcción de GUI´s.

- <http://www.trolltech.com> [Consultado: Septiembre 2005]
- <http://www.wxwidgets.org/> [Consultado: Septiembre 2005]
- <http://www.gtk.org> [Consultado: Septiembre 2005]
- <http://libufo.sourceforge.net> [Consultado: Noviembre 2005]
- <http://w3.org/Style/CSS> [Consultado: Noviembre 2005]
- <http://www.xulplanet.com/> [Consultado: Diciembre 2005]
- <http://del.icio.us/etiqueta/xml> [Consultado: 2005]
- <http://www.cs.cinvestav.mx/CursoVis/prinvisual.html> [Consultado: 2005]

Programación en QT [En línea].

- <http://www.trolltech.com> [Consultado: Enero 2006]
- <http://doc.trolltech.com/4.1/index.html>[Consultado: Enero 2006]
- <http://trolltech.com/products/qt> [Consultado: Enero 2006]
- <http://doc.trolltech.com/> [Consultado: Enero-Noviembre 2006]
- <http://www.qt-es.org> [Consultado: Enero-Noviembre 2006]
- <http://es.wikibooks.org/wiki/Programaci%C3%B3n:Qt4> [Consultado: Enero-Noviembre 2006]
- <http://www.escomposlinux.org/lfs-es/blfs-es-6.0/x/lib.html> [Consultado: Febrero 2006]

Documentación de programación en C++.

- <http://www.conclase.net/> [Consultado: Enero y en Agosto 2006]
- <http://es.tldp.org/Manuales-LuCAS/doc-tutorial-c++/html/> [Consultado: Agosto 2006]
- <http://www.cplusplus.com> [Consultado: Marzo 2006]
- <http://www.infosys.tuwien.ac.at/Research/Component/tutorial/prw231.htm> [Consultado: Junio 2006]
- <http://www.conclase.net/c/librerias/libreria.php?lib=string> [Consultado: Marzo-Junio

2006]

- <http://www.4p8.com/eric.brasseur/cppcen.html> [Consultado: Febrero 2006]
- <http://www.conclase.net/c/ficheros/index.php?cap=002b#FIC_FUNCCPP> [Consultado: Febrero 2006]
- <http://www.modelo.edu.mx/univ/virtech/prograc/cplus4.htm> [Consultado: Febrero 2006]
- <http://www.zator.com/Cpp/E_Ce.htm> [Consultado: Febrero 2006]

Documentación de programación en OpenGL.

- <http://www.opengl.org>[Consultado: Marzo-Septiembre 2006]
- <http://articulos.conclase.net/glut/glut001.html>[Consultado: Agosto-Septiembre 2006]
- <http://www.humbertocervantes.net/homepage/itzamna/TUTORIAL/tutorial.html>[Consultado: Septiembre 2006]

Documentación sobre el manejador OpenDX.

- <http://www.opendx.org/>[Consultado: Septiembre-Noviembre 2006]
- <http://www.facyt.uc.edu.ve/opendx/tutorial.html>[Consultado: Septiembre 2006]

Documentación sobre Linux

- <http://es.tldp.org/Manuales-LuCAS/LIPP2/lipp-2.0-beta-html/node1.html> [Consultado: Noviembre 2006]
- <http://iie.fing.edu.uy/~vagonbar/gcc-make/make.htm#UsoVariables> [Consultado: Diciembre 2005
- <http://www.it.uc3m.es/mario/enlaces/> [Consulado: Octubre 2006]
- <http://www.gnu.org/software/make/make.html>[Consultado Agosto 2005]
- <http://www.gnu.org/philosophy/philosophy.es.html>[Consultado: 2006]

Documentación sobre fluidos

- <http://www.cavendishcfd.com/fd-info1.htm> [Consulado: Julio 2006]
- <http://www.labvis.unam.mx/luiggi/> [Consulado: Febrero, Noviembre, Diciembre 2006]
- <http://www.labvis.unam.mx/~lmd/documentos/mallas/megema.html> [Consulado: Mayo 2006]

REVISTAS

- Santos, Dávila. “Programación Java EE desde cero [III] *Modelo Vista Controlador*”. En: *Sólo programadores*, [No.143], Distribución en México DIMSA-C/Mariano Escobedo, 218 Col. Anáhuac. 11320 México, D.F. Revistas profesionales S.L. Agustín. Asociación Española de Editoriales de Publicaciones Periódicas. P. 30-35.

- Moren, Alejandro; Romay, Pilar. “Metodologías de desarrollo [I]”. En: *Sólo programadores*, [No.144], Distribución en México DIMSA-C/Mariano Escobedo, 218 Col. Anáhuac. 11320 México, D.F. Revistas profesionales S.L. Agustín. Asociación Española de Editoriales de Publicaciones Periódicas. P. 46-51.