

UNIVERSIDADE DE BRASÍLIA
FACULDADE DE TECNOLOGIA
DEPARTAMENTO DE ENGENHARIA CIVIL E AMBIENTAL

PROGRAMA DE INTERFACE DE PRÉ E PÓS
PROCESSAMENTO E LINK COM EXECUTÁVEL PARA
ANÁLISE DE INSTABILIDADE DE TUBULAÇÕES

FÁBIO PESSOA DA SILVA NUNES

ORIENTADOR: LUCIANO MENDES BEZERRA

CO-ORIENTADOR: WILLIAM TAYLOR MATIAS SILVA

DISSERTAÇÃO DE MESTRADO EM ESTRUTURAS E CONSTRUÇÃO
CIVIL

PUBLICAÇÃO: E.DM-007A/07

BRASÍLIA/DF: JULHO 2007

**UNIVERSIDADE DE BRASÍLIA
FACULDADE DE TECNOLOGIA
DEPARTAMENTO DE ENGENHARIA CIVIL**

**PROGRAMA DE INTERFACE DE PRÉ E PÓS PROCESSAMENTO E
LINK COM EXECUTÁVEL PARA ANÁLISE DE INSTABILIDADE
DE TUBULAÇÕES**

FÁBIO PESSOA DA SILVA NUNES

**DISSERTAÇÃO SUBMETIDA AO DEPARTAMENTO DE
ENGENHARIA CIVIL E AMBIENTAL DA FACULDADE DE
TECNOLOGIA DA UNIVERSIDADE DE BRASÍLIA COMO PARTE
DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU
DE MESTRE EM ESTRUTURA E CONSTRUÇÃO CIVIL.**

APROVADA POR:

**Prof^o Luciano Mendes Bezerra, PhD
(Orientador)**

**Prof^a. Aura Conci, DSc
(Examinador Externo)**

**Prof^o Manoel Porfirio Cordão Neto, DSc
(Examinador Interno)**

BRASÍLIA, 09 DE JULHO DE 2007

FICHA CATALOGRÁFICA

NUNES, FÁBIO PESSOA DA SILVA

Programa de Interface de Pré e Pós Processamento e Link com o Executável para Análise de Instabilidade de Tubulações [Distrito Federal] 2007.

xvii, 157p., 297 mm (ENC/FT/UnB, Mestre, Estruturas e Construção Civil, 2007).

Dissertação de Mestrado – Universidade de Brasília. Faculdade de Tecnologia.

Departamento de Engenharia Civil e Ambiental.

1.Tubulações

2.Interface

3.C++

4.*OpenGL*

I. ENC/FT/UnB

II. Título (série)

REFERÊNCIA BIBLIOGRÁFICA

NUNES, F. P. S. (2007). Programa de Interface de Pré e Pós Processamento e Link com o Executável para Análise de Instabilidade de Tubulações. Dissertação de Mestrado em Estrutura e Construção Civil, Publicação E.DM-007A/07, Departamento de Engenharia Civil e Ambiental, Universidade de Brasília, Brasília, DF, 157p.

CESSÃO DE DIREITOS

AUTOR: Fábio Pessoa da Silva Nunes.

TÍTULO: Programa de Interface de Pré e Pós Processamento e Link com o Executável para Análise de Instabilidade de Tubulações.

GRAU: Mestre

ANO: 2007

É concedida à Universidade de Brasília permissão para reproduzir cópias desta dissertação de mestrado e para emprestar ou vender tais cópias somente para propósitos acadêmicos e científicos. O autor reserva outros direitos de publicação e nenhuma parte dessa dissertação de mestrado pode ser reproduzida sem autorização por escrito do autor.

Fábio Pessoa da Silva Nunes
SQN 406 Bl. O, Asa Norte.
70855-050 Brasília – DF – Brasil.
fabionunes@atarde.com.br

AGRADECIMENTOS

A meus pais, Marília e Audi, a minha irmã Lorena.

Aos meus familiares em especial, minha madrinha Anete, meu padrinho Fred, e meus avós Sonia e Wilson.

Aos meus orientadores, Prof. Luciano Mendes Bezerra e Prof. William Taylor Matias Silva pela amizade, paciência, pelos ensinamentos e por confiarem em mim a responsabilidade deste trabalho.

A todos os professores do Programa de Pós-Graduação em Estruturas e Construção Civil, pelos ensinamentos e conhecimentos que foram transmitidos de forma muito competente nesses dois anos de trabalho.

Ao professor, David W. Murray pela ajuda e suporte para o desenvolvimento do trabalho.

A minha namorada Iviane Cunha.

Aos amigos do mestrado, João Uchôa, André, Aline, Marcelo, Wellington, Marcus, Luiz e Gustavo.

Aos amigos, Ary e Thania.

Aos amigos Normando, Genésio, Garcia, Rogério Lobo, Guilherme, Luiz Heleno, Fabrício, Jeffrey, Jorge Zidde, Glacy, Claudio e Murilo.

E finalmente a Deus, por mais essa oportunidade de crescimento.

A minha Mãe Marília, o meu maior exemplo.

RESUMO

INSTABILIDADE DE TUBULAÇÕES: PROGRAMA DE INTEFACE DE PRÉ E PÓS PROCESSAMENTO E LINK COM EXECUTÁVEL

Autor: Fábio Pessoa da Silva Nunes

Orientador: Luciano Mendes Bezerra

Programa de Pós-graduação em Estruturas e Construção Civil

Brasília, julho de 2007

A interface gráfica ou VUI (Visual User Interface) é um módulo muito importante para qualquer programa de análise, tanto na parte de visualizar os dados de entrada, pré-processamento, quanto na interpretação dos dados de saída, pós-processamento. Muitos programas utilizam este recurso gráfico para facilitar o uso por parte do usuário, de programas em nível operacional com *Windows*, ambientes gráfico de desenho como AutoCAD e programas de análise de estruturas de modo geral.

Este trabalho visa o desenvolvimento de uma interface gráfica em C++ para o programa ABP (Analysis of Buried Pipelines), o qual foi desenvolvido em Fortran 77. O compilador escolhido para o desenvolvimento da interface gráfica foi o *Dev-C++*.

Além da escolha do compilador outros dois elementos foram importantes para a construção da VUI, as bibliotecas *OpenGL* e *Win32*, a primeira forneceu todo o suporte necessário para a criação dos elementos gráficos assim como todas as operações envolvidas, já a segunda permitiu que fossem criadas telas no padrão dos sistemas operacionais baseados no *Windows*.

O processo de desenvolvimento da interface se baseou na criação de telas a partir dos cabeçalhos presentes no manual do ABP, gerando assim o arquivo de dados utilizado com entrada do ABP, por fim interpretando e modelando graficamente o arquivo de saída.

ABSTRACT

INSTABILITY OF PIPING: PROGRAM OF INTEFACE OF PRÉ AND PÓS PROCESS AND LINK WITH EXECUTABLE

Author: Fabio Pessoa da Silva Nunes

Supervisor: Luciano Mendes Bezerra

Programa de Pós-graduação em Estruturas e Construção Civil

Brasília, July of 2007

An interface graph is a very important module for any program of analyzes, so much in the part of visualizing the entrance data, pré-processing, as in the interpretation of the exit data, powder-processing. Many programs use this resource to facilitate the use for part do user, from operational programs like *Windows*, drawing as AutoCAD and programs of analyzes of structures in general like Ansys.

This work seek the development of a interface graph in C++ for ABP (ANALYSIS OF BURIED PIPELINES) program, which was developed in Fortran 77. The chosen compiler for the development of the interface was *Dev-C++*.

Besides the choice of the compiler other two elements were very important, as libraries *OpenGL* and *Win32*, the first supplied all necessary support for the creation the graphic elements as well as all the involved operations, already the second allowed the creation of the screens based in the *Windows* systems operation pattern.

The process development of the interface was based on the creation of screens to leave of the headers presents in ABP manual, generating the data file entrance used in ABP, interpreting and modeling the exit file graphically.

SUMÁRIO

1 – INTRODUÇÃO	1
1.1 - MOTIVAÇÃO	2
1.2 – OBJETIVO GERAL.....	3
1.3 – OBJETIVOS ESPECÍFICOS.....	3
1.4 – ESTRUTURA DO TRABALHO.....	4
1.5 – O QUE HÁ DE NOVO NESTE TRABALHO	4
2 – REVISÃO BIBLIOGRÁFICA	5
2.1 – A COMPUTAÇÃO GRÁFICA	5
2.1.1 - Origens	5
2.1.2 – Percepção Tridimensional.....	6
2.1.3 – Informações Monoculares.....	7
2.1.4 – Informações Visuais Óculo-motoras	7
2.1.5 – Informações Visuais Estereoscópicas.....	7
2.4 - BIBLIOTECAS.....	8
2.4.1 – <i>OpenGL</i>	8
2.4.1.1 – Introdução	8
2.4.1.2 – Padronização das Funções e Tipos de Dados	10
2.4.1.3 – GLUT Utility Toolkit	12
2.4.1.4 – Variáveis de Estado e Espaço de Trabalho.....	14
2.4.1.5 – Primitivas Gráficas	18
2.4.1.6 – Transformações Geométricas	20
2.4.1.7 – Zoom e Pan	22
2.4.2 – <i>Win32 API</i>.....	23
2.4.2.1 – Uma Breve História do <i>Windows</i>	23
2.4.2.2 – Como funciona a <i>Win32 API</i>	24
2.4.2.3 – Início e Terminação de Programas	25
2.4.2.4 – <i>Windows Messages</i>	28
2.4.2.5 – Criação de uma Janela	29
2.4.2.6 – Arquivos de Recursos	37
2.4.2.7 – Elementos Gráficos.....	38
2.5 – O COMPILADOR <i>DEV-C++</i>	46

2.6 – ABP (ANALYSIS OF BURIED PIPELINES)	47
3 – METODOLOGIA.....	52
3.1 – VISÃO GERAL DO ABP SEM A INTERFACE GRÁFICA.....	52
3.2 – VISÃO GERAL DO ABP COM A INTERFACE GRÁFICA.....	53
3.3 – PRINCIPAIS ELEMENTOS DA INTERFACE GRÁFICA.....	54
3.3.1 – Processo de Criação dos Elementos Gráficos.....	55
3.3.1.1 – Apoios.....	56
3.3.1.2 – Tubulação na Geração de Malhas.....	59
3.3.1.3 – Tubulação na Seção Transversal	63
3.3.1.4 – Gráficos.....	66
3.3.1.5 – Deformada	70
3.3.2 – Tela de Saudação	72
3.3.3 – ABP Interface.....	73
3.3.4 – New.....	74
3.3.5 – General Analisys	74
3.3.6 – Nodal Informations.....	79
3.3.7 – Material Information.....	81
3.3.8 – Pipe Size Information	82
3.3.9 – Element Attributes Information.....	84
3.3.10 – Element Information	84
3.3.11 – Load Information	85
3.3.12 – Graphics.....	88
3.3.13 – Strain.....	90
3.3.14 – Run	92
3.3.15 – Reports IN e OUT	93
3.3.16 – About.....	94
4 – EXEMPLOS.....	95
4.1 – Exemplo com carga concentrada.....	95
4.2 – Exemplo com pressão interna.....	107
5 – CONCLUSÕES E SUGESTÕES PARA TRABALHOS FUTUROS.....	122
REFERÊNCIAS BIBLIOGRÁFICAS	124

LISTA DE TABELAS

Tabela 2.1 – Padronização dos nomes das funções	10
Tabela 2.2 – Tipos de argumentos em funções <i>OpenGL</i>	11
Tabela 2.3 – Tipos de dados <i>OpenGL</i>	12
Tabela 2.4 – Primitivas Gráficas em <i>OpenGL</i>	18
Tabela 2.5 – Primitivas Gráficas <i>GLUT</i> em <i>OpenGL</i>	19

LISTA DE FIGURAS

Figura 2.1 – Erupção vulcânica no espaço na lua Joviam.....	6
Figura 2.2 – Toróide	7
Figura 2.3 – Funcionamento do pipeline <i>OpenGL</i>	9
Figura 2.4 – Objetos pré-definidos da biblioteca GLUT.....	12
Figura 2.5 – Sistema de Referência do Universo (SRU).....	14
Figura 2.6 – Sistema de Referência da Tela (SRT).....	15
Figura 2.7 – Mapeamento da Imagem do SRU para SRT.....	15
Figura 2.8 – SRU Tridimensional e a Regra da Mão Direita	15
Figura 2.9 – (a) Projeção Paralela, (b) Projeção Perspectiva.	16
Figura 2.10 – Rotação da câmera sintética devido ao vetor up.....	17
Figura 2.11 – Primitivas Gráficas 2D em <i>OpenGL</i>	19
Figura 2.12 – Primitivas Gráficas 3D em <i>OpenGL</i>	20
Figura 2.13 – (a) Pontos de iluminação, (b) Ponto vermelho translacionado.	20
Figura 2.14 – Transformação geométrica de escala.....	21
Figura 2.15 – Transformação geométrica de rotação.....	21
Figura 2.16 – (a) Cenário 3D, (b) Zoom In.....	22
Figura 2.17 – Exemplo de Pan.....	22
Figura 2.18 – Tela do <i>MS-DOS</i>	24
Figura 2.19 – Passagem de parâmetros para programas externos.....	26
Figura 2.20 – Fluxo de informações dentro do programa.....	28
Figura 2.21 – Janela Principal criada a partir do código acima.....	31
Figura 2.22 – Funcionamento dos arquivos de recurso.....	38
Figura 2.23 – <i>Static</i> indicado pela seta vermelha.....	39
Figura 2.24 – <i>Edit</i> indicado pela seta vermelha.....	41
Figura 2.25 – <i>Push Button</i> indicado pela seta vermelha.....	42
Figura 2.26 – <i>Radio Button</i> indicado pela seta vermelha.....	43
Figura 2.27 – <i>Group Box</i> indicado pela seta vermelha.....	44
Figura 2.28 – <i>Combo Box</i> indicado pela seta vermelha.....	45
Figura 2.29 – Logotipo do Dev-C++.....	46
Figura 2.30 – Deformação no regime elasto-plástico.....	48
Figura 2.31 – Tubulação Instrumentada.....	49

Figura 2.32 – Mecanismo de Flambagem: representação do upheaval.....	50
Figura 2.33 – Não pressurizada: modo diamond whinkle.....	50
Figura 2.34 – Pressurizada: modo bulge-wrinkle.....	51
Figura 3.1 – Visão geral do ABP.....	52
Figura 3.2 – Visão geral do ABP, após o advento da interface gráfica.....	53
Figura 3.3 – Menu da interface gráfica do ABP.....	54
Figura 3.4 – Geração da Malha em 2D.....	55
Figura 3.5 – Diâmetro e Espessura da Tubulação	56
Figura 3.6 – Apoio com Restrição de Movimento	57
Figura 3.7 – Apoio Flexível e o Zoom do Detalhe.....	59
Figura 3.8 – Tubulação e o Zoom mostrando com Detalhe	63
Figura 3.9 – Tubulação e o Zoom mostrando com Detalhe	66
Figura 3.10 – Gráfico	70
Figura 3.11 – Animação da estrutura deformada	71
Figura 3.12 – Tela de Saudação	72
Figura 3.13 – Tela inicial da Interface Gráfica do ABP.....	73
Figura 3.14 – Tela <i>File Information</i>	74
Figura 3.15 – Tela <i>General Analysis</i>	75
Figura 3.16 – Convenção de sinal	76
Figura 3.17 – Tela <i>Nodal Information</i>	79
Figura 3.18 – Tela <i>Nodal Information</i> com <i>zoom</i> e <i>pan</i>	80
Figura 3.19 – Tela <i>Nodal Information</i> com <i>zoom</i> e <i>pan</i>	80
Figura 3.20 – Tela <i>Material Information</i>	81
Figura 3.21 – Tela <i>Pipe Size Information</i>	83
Figura 3.22 – Tela <i>Pipe Size Information</i> com <i>zoom</i> e <i>pan</i>	83
Figura 3.23 – Tela <i>Element Attributes Information</i>	84
Figura 3.24 – Tela <i>Element Information</i>	85
Figura 3.25 – Tela <i>Load Information</i>	86
Figura 3.26 – Exemplo de gráfico capturando apenas uma coordenada	88
Figura 3.27 – Aplicação de <i>Zoom</i> e <i>Pan</i>	89
Figura 3.28 – Exemplo de gráfico capturando vinte e seis coordenadas.....	89
Figura 3.29 – Aplicação de <i>Zoom</i> e <i>Pan</i>	90
Figura 3.30 – Tela <i>Strain</i>	90
Figura 3.31 – Tela <i>Strain</i> com <i>Zoom</i> , <i>Rotação</i> e <i>Pan</i>	91

Figura 3.32 – Tela <i>Strain</i> com <i>Zoom</i> , <i>Rotação</i> e <i>Pan</i>	91
Figura 3.33 – Execução do ABP através da Interface	92
Figura 3.34 – Seta indicando o ícone <i>Reports In</i>	93
Figura 3.35 – Seta indicando o ícone <i>Reports Out</i>	93
Figura 3.36 – Seta indicando o ícone <i>About</i>	94
Figura 4.1 – Situação esquemática do 1° exemplo.....	95
Figura 4.2 – Nome do arquivo.....	95
Figura 4.3 – Criação do arquivo <i>freespan30Mhb.inp</i>	96
Figura 4.4 – Informações Gerais	96
Figura 4.5 – Arquivo <i>freespan30Mhb.inp</i> com as informações gerais.....	97
Figura 4.6 – Malha da tubulação.	97
Figura 4.7 – Detalhe do apoio da tubulação	98
Figura 4.8 – Detalhe do apoio visto de fora da tubulação	98
Figura 4.9 – Arquivo <i>freespan30Mhb.inp</i> com as informações nodais.....	98
Figura 4.10 – Propriedades dos Materiais	99
Figura 4.11 – Arquivo <i>freespan30Mhb.inp</i> com as informações nodais.....	99
Figura 4.12 – Diâmetro e Espessura da Tubulação	100
Figura 4.13 – Detalhe com zoom da seção transversal.	100
Figura 4.14 – Detalhe interno da tubulação.....	101
Figura 4.15 – Arquivo <i>freespan30Mhb.inp</i> com as informações da seção transversal	101
Figura 4.16 – Parâmetros de integração	101
Figura 4.17 – Arquivo <i>freespan30Mhb.inp</i> com as informações de integração.....	102
Figura 4.18 – Informações dos elementos	102
Figura 4.19 – Arquivo <i>freespan30Mhb.inp</i> com as informações dos elementos	103
Figura 4.20 – Informações do Carregamento.	103
Figura 4.21 – Arquivo <i>freespan30Mhb.inp</i> com as informações dos carregamentos	103
Figura 4.22 – Gráfico Coordenada X - Cortante.	104
Figura 4.23 – Detalhe da captura da Coordenada X.....	104
Figura 4.24 – Gráfico Coordenada X - Deformada.	105
Figura 4.25 – Detalhe da captura da Coordenada X.....	105
Figura 4.26 – Deformada.....	106
Figura 4.27 – Situação esquemática do 2° exemplo.....	107
Figura 4.28 – Nome do arquivo.....	107
Figura 4.29 – Criação do arquivo <i>freespan15Mhb.inp</i>	108

Figura 4.30 – Informações Gerais	108
Figura 4.31 – Arquivo <i>fre span15Mhb.inp</i> com as informações gerais.....	109
Figura 4.32 – Malha da tubulação.	109
Figura 4.33 – Detalhe do segundo apoio da tubulação.....	110
Figura 4.34 – Detalhe do primeiro apoio visto de dentro da tubulação	110
Figura 4.35 – Arquivo <i>fre span15Mhb.inp</i> com as informações nodais.....	110
Figura 4.36 – Propriedades dos Materiais	111
Figura 4.37 – Arquivo <i>fre span15Mhb.inp</i> com as informações do material.....	111
Figura 4.38 – Diâmetro e Espessura da Tubulação	112
Figura 4.39 – Detalhe com zoom da seção transversal.	113
Figura 4.40 – Detalhe da parede da tubulação	113
Figura 4.41 – Arquivo <i>fre span15Mhb.inp</i> com as informações da seção transversal	114
Figura 4.42 – Parâmetros de integração	114
Figura 4.43 – Arquivo <i>fre span15Mhb.inp</i> com as informações de integração.....	114
Figura 4.44 – Informações dos elementos.....	115
Figura 4.45 –Arquivo <i>fre span15Mhb.inp</i> com as informações dos elementos	115
Figura 4.46 – Informações do Carregamento.....	116
Figura 4.47 –Arquivo <i>fre span15Mhb.inp</i> com as informações dos carregamentos	116
Figura 4.48 – Gráfico Coordenada X - Momento.	117
Figura 4.49 – Gráfico Coordenada X - Momento.	118
Figura 4.50 – Gráfico Coordenada X - Curvatura.....	118
Figura 4.51 – Gráfico Coordenada X - Curvatura.....	119
Figura 4.52 – Gráfico Coordenada X – Força Axial.	119
Figura 4.53 – Gráfico Coordenada X – Força Axial.	120
Figura 4.54 – Gráfico Coordenada X – Top Strain & Bottom Strain.....	120
Figura 4.55 – Deformada.....	121

LISTA DE SÍMBOLOS, NOMENCLATURA E ABREVIACÕES

2D	- Duas Dimensões
3D	- Três Dimensões
3DLABS	- Fabricante de placas de vídeo e desenvolvedora de tecnologias para processadores gráficos
ABP	- Analysys of Buried Pipeline
ALGOL 68	- É uma família de linguagens de programação de alto nível voltadas principalmente para aplicações científicas. Seu nome provém das palavras "ALGOarithmic Language"
ANSI	- American National Standards Institute
API	- Application Programming Interface
APPLE	- Empresa multinacional norte-americana que atua no ramo de eletrônicos e informática
ASCII	- American Standard Code for Information Interchange
Assenbly	- É uma notação legível por humanos para o código de máquina que uma arquitetura de computador específica usa
AT&T	- American Telephone and Telegraph
BCPL	- Basic Combined Programming Language
Buffers	- É uma região de memória temporária utilizada para escrita e leitura de dados
C	- Linguagem de programação estruturada e padronizada criada na década de 1970 por Dennis Ritchie e Ken Thompson

C++	- É uma linguagem de programação orientada a objeto de alto nível com facilidades para o uso em baixo nível, multiparadigma e de uso geral
GLU	- Graphics Utility Library
IBM	- International Business Machines
Linux	- Sistema operacional do tipo Unix que utiliza o núcleo Linux e também os programas de sistema GNU
LISA	- Computador pessoal (PC) revolucionário lançado pela Apple Computer em 1983
MSDOS	- MicroSoft Disk Operating System
NURBS	- Nonuniform Rational B-Splines
NVIDIA	- Empresa norte-americana que fabrica peças de computador, e é mais popularmente conhecida por sua série de placas de vídeo GeForce
OpenGL	- Open Graphics Library
PARC	- Palo Alto Research Center
PC	- Personal Computer
POO	- Programação Orientada a Objeto
SGI	- Silicon Graphics
SIMULA 67	- Linguagens de programação denominadas Simula, extensões de ALGOL 60, projetada para apoiar a simulação de eventos discretos.
SUN	- Empresa fabricante de computadores, semicondutores e software
Unix	- Sistema operacional portátil multitarefa e multiutilizador
Win32	- Biblioteca padrão do sistema operacional Windows para 32 bits
Windows	- Popular família de sistemas operacionais criados pela Microsoft

1 – INTRODUÇÃO

A utilização de uma interface amigável para o usuário é uma etapa do processo de criação de *software*, independente do ramo do conhecimento, porém é uma etapa que por muitas vezes, não é desenvolvida ao ponto de promover uma maior facilidade para o usuário.

É comum o uso de arquivos no modo texto, como forma de entrada e saída de dados, entretanto muitas vezes, estes arquivos são gerados por outros *softwares* ou de maneira manual, o que demanda, na maior parte dos casos, um aumento no tempo da análise.

A diminuição deste tempo pode ser alcançada, através de uma boa interatividade com o programa, onde o usuário tenha acesso somente nos pontos de tomada de decisão, não sendo necessário participar diretamente dos processos repetitivos e manuais, como a criação de arquivos texto, que deve ser realizada automaticamente pelo programa de interface. Além disto, a interface deve possibilitar em sua modelagem gráfica, clareza e fidelidade aos dados.

Um dos problemas no desenvolvimento de interfaces se encontra no programa fonte utilizado como objeto, como na maioria dos casos se desenvolve a parte gráfica a partir de um programa já existente, o objeto, é imprescindível que o formato dos dados de entrada esteja igual ao do programa objeto, para que não ocorra incompatibilidade no momento da análise.

A utilização da linguagem “C++” para o desenvolvimento do programa ocorreu pelos seguintes motivos: a larga utilização desta linguagem de programação que começou a ser idealizada desde 1966 conseqüentemente uma vasta quantidade de material desde bibliografia até *sites* de desenvolvimento e fóruns de discussão, o fato da linguagem permitir o uso de diversas bibliotecas e permitir o acesso aos dispositivos de *hardware*.

A escolha do compilador foi difícil devido ao fato deste ter que proporcionar o desenvolvimento do *software* e também ser de domínio público para que assim não ocasionasse problemas de direitos autorais. Após a procura por diversos compiladores tanto em plataformas baseadas em *Windows* como em Linux chegou-se no *Dev-C++*, que

tem versões nos dois sistemas operacionais, o que facilita uma possível migração para o Linux, e é de domínio público.

Da mesma maneira a biblioteca gráfica *OpenGL* foi selecionada, ela tem versões tanto para *Windows* quanto pra Linux e também é de domínio público, um dos pontos importantes da *OpenGL* é permitir que o usuário tenha acesso a uma grande quantidade de recursos gráficos e de inúmeras comunidades de desenvolvimento na *internet*.

Como a biblioteca *OpenGL* não fornece suporte para criação de janelas, *menus* e botões, por exemplo, era preciso escolher alguma biblioteca que pudesse fornecer estes elementos tão necessários quanto os demais, a escolha da *Win32* foi natural após a decisão de se adotar o *Windows* como sistema operacional, onde podemos encontrar nela todos os elementos gráficos pertinentes nos sistemas operacionais baseados no *Windows*.

1.1 - MOTIVAÇÃO

Muitos programas de análise estrutural utilizam arquivos de texto como entrada de dados, porém a montagem desses arquivos em muitas das vezes demandam uma considerável quantidade de tempo para serem montados, o mesmo problema ocorre com os arquivos de saída que também estão em formato de texto e para serem interpretados são necessários a ajuda de programas auxiliares.

Conseqüentemente é comum o uso de programas auxiliares que permitem visualizar estes dados de maneira gráfica tanto na entrada, no caso de uma malha, quanto na saída como gráficos que mostram a variação da tensão ao longo do comprimento ou o comportamento da deformada de uma estrutura.

O desenvolvimento de uma interface gráfica adaptada ao tipo de estrutura em estudo e que prive o usuário de tarefas repetitivas, como montar o arquivo de entrada, foram os motivos que levaram a elaboração deste trabalho.

1.2 – OBJETIVO GERAL

Neste trabalho, um software de uso na indústria petrolífera e desenvolvido desde a década de 80 (Zhou, 1980) pela UofA (*University of Alberta*), Canadá, chamado de ABP (*Analysis of Buried Pipeline*) será objeto de estudo. O ABP faz análises de instabilidade de tubulações enterradas, inclusive tubulações submarinas para condução de petróleo, porém utiliza o ambiente DOS. O objetivo geral deste trabalho consiste em desenvolver uma interface gráfica VUI (*Visual User Interface*), tanto de pré-processamento quanto de pós-processamento, para o programa executável do ABP (*Analysis of Buried Pipelines*).

Os arquivos de entrada serão gerados por um processamento automático, assim como a própria execução do programa ABP, que ficará por conta de um *link* entre o programa de interface e o executável, vale ressaltar que a VUI aqui desenvolvida utilizará apenas o executável, pois o código fonte do programa está protegido por leis de direitos autorais, finalizando a VUI desenvolvida interpreta e modela de forma gráfica os dados contidos nos arquivos de entrada e saída. Com isso, a interface ficará encarregada de proporcionar ao usuário tomar decisões com maior clareza e rapidez.

1.3 – OBJETIVOS ESPECÍFICOS

- 1) Desenvolver o programa de interface utilizando a linguagem de programação “C++”.
- 2) Utilizar o compilador de domínio público *DEV-C++*.
- 3) Utilizar a API (Application Programming Interface) *Win32*, para viabilizar a execução nos sistemas operacionais da “família” *Windows*.
- 4) Utilizar a API (Application Programming Interface) gráfica de domínio público *OpenGL*.

1.4 – ESTRUTURA DO TRABALHO

No Capítulo 1 será mostrado uma visão geral do trabalho apresentando os principais pontos que serão abordados, assim como a motivação que levou ao desenvolvimento da interface.

No Capítulo 2 serão apresentados todos os elementos utilizados para o desenvolvimento do *software*, como as bibliotecas *Win32* e *OpenGL*, a linguagem utilizada, o C++ e o programa que servirá de objeto da interface gráfica o ABP.

No Capítulo 3 é demonstrado a metodologia usada para a construção do *software*, explicando como foram concebidas as telas de cada etapa do programa e como estas foram interligadas.

No Capítulo 4 se encontram os exemplos utilizados para demonstrar o uso da VUI e como esta pode facilitar a análise por parte do usuário.

No Capítulo 5 contém as conclusões e sugestões para trabalhos futuros.

1.5 – O QUE HÁ DE NOVO NESTE TRABALHO

O desenvolvimento de uma interface gráfica de pré e pós processamento para análise de instabilidade de tubulações enterradas. Esta interface preserva a inviolabilidade do código fonte do ABP o qual se encontra em Fortran, fazendo o *link* deste executável com a interface gráfica para ajudar o usuário a preparar os dados e a interpretar resultados.

2 – REVISÃO BIBLIOGRÁFICA

2.1 – A COMPUTAÇÃO GRÁFICA

2.1.1 - Origens

O primeiro computador a possuir recursos gráficos de visualização de dados numéricos foi o *Whirlwind I*, desenvolvido pelo MIT com finalidades acadêmicas e militares, posteriormente usado para o desenvolvimento de um sistema de monitoramento de vôos chamado SAGE (*Semi-Automatic Ground Environment*). Com o passar do tempo, o termo *Computer Graphics*, criado por Verne Hudson, enquanto coordenava um projeto para a *Boeing*, passou a fazer parte do vocabulário computacional. Finalmente, em 1962, surgiu a mais importante publicação da computação gráfica, que foi a tese de Ivan Sutherland chamada de *Sketchpad – (A Man-Machine Graphical Communication System)*, que incorporou todos os principais conceitos da computação gráfica existentes. Destaca-se que essa teoria é a base dos programas que utilizam a tecnologia *CAD (Computer Aided Design)*, largamente empregada nos *softwares* gráficos.

Novas técnicas e algoritmos surgiram na década 70 e são utilizados até hoje, como os métodos de sombreamento.

Pouco tempo depois, em 1975, surgiu o primeiro hardware com interface visual que seria o predecessor do *Macintosh*.

Nesta mesma década a computação gráfica foi reconhecida como área específica da ciência da computação e a publicação do livro *Fractals: Form, Chance and Dimension*.

Outro marco da computação gráfica foi em 1980 com a publicação da imagem de uma erupção vulcânica no espaço na lua Jovian tirada pela nave espacial *Voyager*, isto devido ao processamento da imagem pelo telescópio que permitiu a filtragem de ruído e imperfeições, destacando os aspectos relevantes.

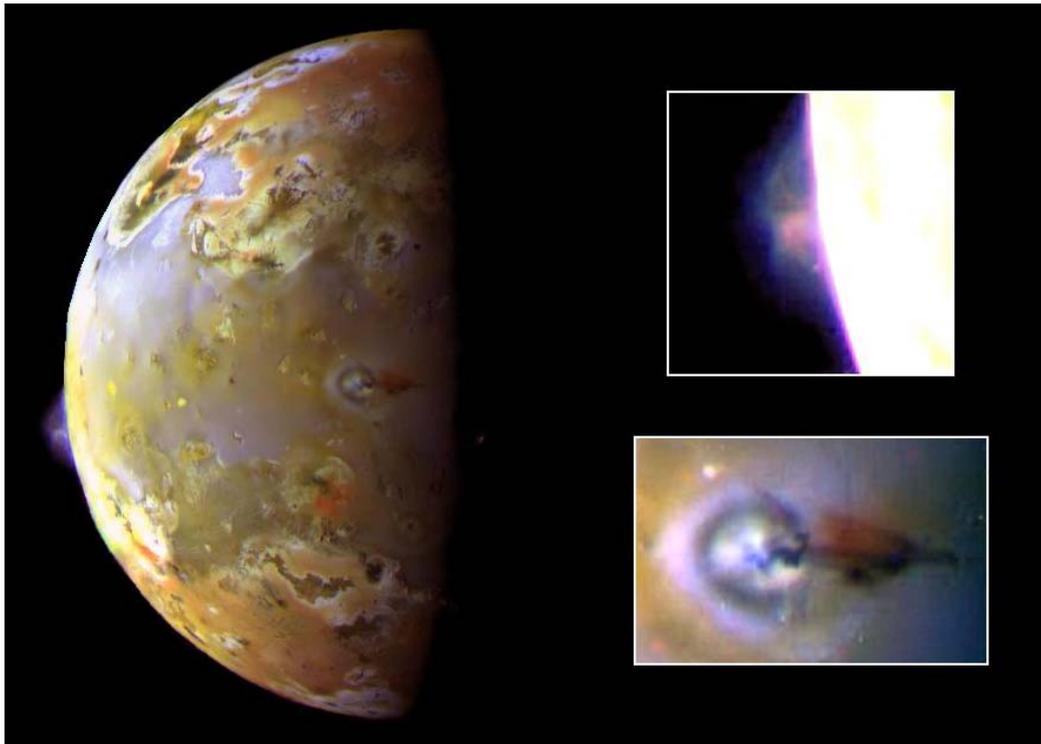


Figura 2.1 – Erupção vulcânica no espaço na lua Joviam
(<http://apod.nasa.gov>, Acesso 15 jun 2006)

A partir deste momento surgiram técnicas mais avançadas, como o *ray-tracing* e a radiosidade utilizadas para efeitos de iluminação global, que puderam aproximar essas imagens do realismo, introduzindo assim o conceito do fotorrealismo a computação gráfica.

2.1.2 – Percepção Tridimensional

Toda essa tecnologia está apoiada em dois conceitos. O primeiro refere-se à percepção tridimensional, que é a maneira como o usuário percebe a profundidade do objeto, diferenciando se este é 2D ou 3D; se apresenta uma diminuição em seu tamanho ao longo da distância, dando assim, a noção de perspectiva; ou se possui fenômenos visuais, como a presença de sombras, de variação da luz ou de uma diferença na densidade das texturas (Azevedo & Conci, 2003), como pode ser visto na Figura 2.2.

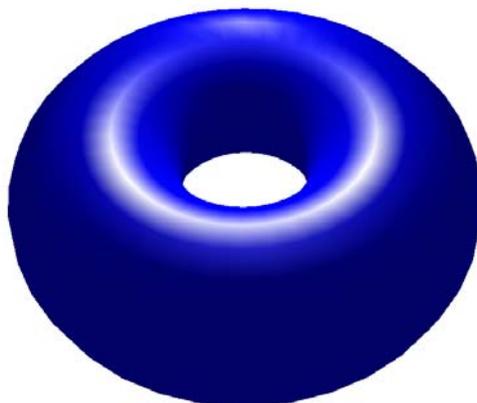


Figura 2.2 – Toróide

2.1.3 – Informações Monoculares

Podem ser chamadas também de *static depth cues* (informações estáticas de profundidade) ou *pictorial depth cues* (informações de profundidade na imagem) e tratam de informações como: perspectiva linear, conhecimento prévio do objeto, oclusão, densidade de texturas, variação da reflexão da luz e as sombras, (Azevedo & Conci, 2003).

2.1.4 – Informações Visuais Óculo-motoras

Este tipo de informação ocorre devido a movimentação dos olhos através de dois conjuntos de músculos do globo ocular, onde um desses conjuntos informa ao cérebro o grau de contração e o segundo conjunto tem a tarefa de focar os raios luminosos na retina, alterando assim a curvatura do cristalino, (Azevedo & Conci, 2003).

2.1.5 – Informações Visuais Estereoscópicas

Essa informação é fruto do posicionamento dos olhos, já que estes estão posicionados de forma diferente. Esta diferença também chamada de disparidade binocular é usada para obter a distância relativa dos objetos a uma distância de até 10 metros do observador, contudo não está bem definido quanto a nossa percepção esta relacionada a disparidade e quanto a familiaridade com os objetos, (Azevedo & Conci, 2003).

2.4 - BIBLIOTECAS

2.4.1 – *OpenGL*

2.4.1.1 – Introdução

Esta biblioteca pode ser definida como uma especificação aberta multiplataforma de um conjunto de rotinas gráficas. *OpenGL*, pode ser compreendida também como uma “interface para o hardware gráfico”, é largamente utilizada para este tipo de aplicação, devido conter rotinas capazes de modelar objetos tanto bidimensionais (2D) quanto tridimensionais (3D) e devido a biblioteca ser extremamente portátil e rápida.

Uma das vantagens da *OpenGL (Open Graphics Library)*, esta no fato dela ser um *free-software*, baseado em uma licença de domínio público GNU (*General Public License*). Permitindo que qualquer usuário possa utilizar o software sem a necessidade de pagar por isso.

A especificação *OpenGL* é gerenciada por um consórcio independente formado em 1992, o ARB (*Architecture Review Board*). O grupo, constituído por muitas empresas líderes da área, entre as quais estão a 3Dlabs, Apple, NVIDIA, SGI, e SUN, é responsável pela aprovação de novas funcionalidades, versões e extensões de *OpenGL*. As freqüentes revisões da especificação permitem a sua constante evolução e aproveitamento da capacidade do hardware gráfico.

Resumindo, *OpenGL* é uma API (*Application Programming Interface*) para criação de programas gráficos 2D e 3D para diversas plataformas, que variam de potentes estações de trabalho a simples computadores pessoais, (Cohen & Manssour, 2006).

As aplicações desta biblioteca têm uma grande abrangência, passando por ferramentas CAD até poderosos programas de modelagem.

Ao se utilizar a biblioteca *OpenGL*, não é necessário descrever detalhadamente o objeto desejado seja ele 2D ou 3D, basta especificar o conjunto de passos que devem ser seguidos para se obter o aspecto ou o efeito desejado. Esses passos envolvem chamadas desta API

que inclui em torno de 250 funções: 200 da própria biblioteca *OpenGL* e 50 da biblioteca *GLU* (*OpenGL Utility Library*), que faz parte da implementação *OpenGL*.

Devido a sua portabilidade, a biblioteca *OpenGL* não possui funções para gerenciamento de janelas, tratamento de eventos ou manipulação de arquivos. Neste caso, são utilizadas as funções específicas de cada plataforma para tal propósito, como, por exemplo, a própria API do ambiente Microsoft *Windows* que também será utilizada no desenvolvimento da interface gráfica, como também não existe um formato de arquivo *OpenGL*, ela oferece um conjunto de primitivas gráficas, como pontos, linhas e polígonos, para exibição de modelos. Já a biblioteca *GLU* fornece algumas funções para modelagem de superfícies quadráticas, assim como curvas e superfícies *NURBS* (*Non Uniform Rational B-Splines*) são algumas delas (Schreiner, 2004; Wright, 2000).

A palavra *pipeline* é usada para descrever um processo composto de duas ou mais etapas para a geração de uma imagem. A Figura 2.4, adaptada de (Schreiner, 2004; Wright, 2000), mostra uma versão simplificada do *pipeline OpenGL*. Como uma aplicação faz chamadas a várias funções da API *OpenGL*, os comandos são armazenados em uma memória específica (buffer de comandos). Depois de ser preenchida com comandos de desenho de primitivas, iluminação e dados de textura, entre outros, ela é “esvaziada”, ou seja, os comandos e dados são passados para o próximo estágio do *pipeline* (Wright, 2000).



Figura 2.3 – Funcionamento do pipeline *OpenGL*.

O processamento dos dados, através dos estágios, ocorre de maneira diferente. Dados geométricos, por exemplo, vértices são processados de uma forma diferente dos dados de imagens, tais como *pixels*. Porém todos os dados passam pelo estágio de rasterização, o

qual converte os dados em fragmentos, o qual em *OpenGL* se traduz em uma posição na tela, que além da cor possui outras informações associadas como profundidade e coordenadas de textura. Cada um dos fragmentos atualiza os *pixels* do *frame buffer*, que nada mais é que à memória do dispositivo gráfico. Apo esta etapa é que são executadas operações como oclusão e transparência, antes da exibição final da imagem.

2.4.1.2 – Padronização das Funções e Tipos de Dados

Como a *OpenGL* é uma biblioteca de domínio público tornou-se necessário uma padronização das funções existentes, para que seja possível a inclusão de novas classes e subrotinas, de tal forma que qualquer pessoa possa utilizá-la.

Portanto, os nomes das funções são divididos em quatro partes. A primeira é um prefixo que informa qual é a biblioteca, a segunda é a raiz, ou seja, representa o comando *OpenGL* que corresponde a função. E por fim aparecem a quantidade e o tipo dos argumentos.

<Prefixo_Biblioteca> <Comando_Raiz> <Contador_Argumentos_Opcional>
<Tipo_Argumentos_Opcional>

Na função:

glColor3f(GLfloat red, GLfloat green, GLfloat blue)

Tabela 2.1 – Padronização dos nomes das funções

Argumento	Descrição
gl	É o prefixo que representa a biblioteca gl.
Color	É o comando raiz que indica o objetivo da função.
3	É o contador para o número de argumentos que a função possui.
f	Indica que os argumentos são valores de ponto flutuante.

Percebe-se que os dois últimos argumentos possibilitam a criação de inúmeras funções variando-se apenas a quantidade e o tipo dos dados a serem recebidos, como as funções

glColor3i (*GLint red*, *GLint green*, *GLint blue*) e *glColor3d* (*GLdouble red*, *GLdouble green*, *GLdouble blue*), onde houve apenas uma variação no tipo de argumento recebido, a tabela abaixo mostra os tipos de argumentos das funções *OpenGL*.

Tabela 2.2 – Tipos de argumentos em funções *OpenGL*

Argumento	Descrição
b	Para argumentos do tipo <i>signed char</i> .
s	Para argumentos do tipo <i>short</i> .
i	Para argumentos do tipo <i>integer</i> .
f	Para argumentos do tipo <i>float</i> .
d	Para argumentos do tipo <i>double</i> .
ub	Para argumentos do tipo <i>unsigned char</i> .
us	Para argumentos do tipo <i>unsigned short</i> .
ui	Para argumentos do tipo <i>unsigned int</i> ou <i>insigned long</i> .
v	Pode ser qualquer um dos tipos anteriores, e <i>v</i> indica que é um argumento do tipo ponteiro para um vetor do tipo especificado no lugar de * (por exemplo, <i>fv</i> é um vetor do tipo <i>float</i> e <i>iv</i> é um vetor do tipo <i>int</i>).

Porém existem funções que não recebem as duas ultimas informações, isto se deve ao fato de que muitas destas não possuem argumentos, ou então trabalham com variações de quantidade ou tipo, por exemplo, as funções *glLoadIdentity* (*void*), função da biblioteca GL que troca a matriz de transformação corrente pela matriz identidade, e a função *glTranslatef* (*GLfloat x*, *GLfloat y*, *GLfloat z*), função também da biblioteca GL, que multiplica a matriz de transformação pela matriz de translação.

Existe também a padronização dos tipos de dados da *OpenGL*, para que assim seja possível garantir a portabilidade da biblioteca, logo a declaração das variáveis passa a não depender do compilador que esta sendo utilizado, a tabela 2.9 mostra os tipos de dados da *OpenGL*.

Tabela 2.3 – Tipos de dados *OpenGL*

Tipos de dado <i>OpenGL</i>	Representação Interna	Tipo de dado C Equivalente	Sufixo
GLbyte	Inteiro de 8 bits	Signed char	b
GLshort	Inteiro de 16 bits	Short	s
GLint, GLsizei	Inteiro de 32 bits	Int ou long	i
GLfloat, GLclampf	Ponto flutuante de 32 bits	Float	f
GLdouble, GLclampd	Ponto flutuante de 64 bits	Double	d
GLubyte, GLboolean	Inteiro de 8 bits sem sinal	Unsigned char	ub
GLushort	Inteiro de 16 bits sem sinal	Unsigned short	us
GLuint, GLenum, GLbitfield	Inteiro de 32 bits sem sinal	Unsigned long ou unsigned int	ui

2.4.1.3 – GLUT Utility Toolkit

Projetada por Mark Kilgard, a biblioteca GLUT é capaz de realizar operações como: Criar e gerenciar janelas, menus; desenhar textos e objetos pré-definidos, e por fim tratar de eventos provenientes de hardwares como teclados, *mouses* e joysticks. Para que ela seja corretamente utilizada são necessários três arquivos, o *header* ou cabeçalho *glut.h*, a biblioteca de referência *glut32.lib* e a DLL *glut32.dll* (Kilgard, 1994), exemplificado na Figura 2.5.

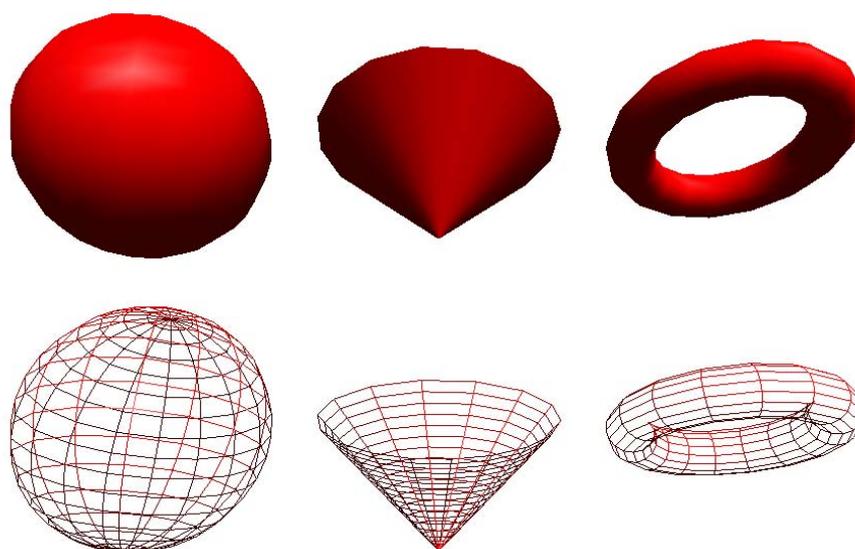


Figura 2.4 – Objetos pré-definidos da biblioteca GLUT

Basicamente a estrutura da GLUT se divide em duas partes, a inicialização e o tratamento de eventos.

Na inicialização são definidos alguns parâmetros necessários para a execução e tem como principais funções: *glutInitDisplayMode* que informa quantos e quais *buffers* serão utilizados, através dos seguintes parâmetros: GLUT_DOUBLE para dois *buffers*, GLUT_SINGLE para um *buffer*, a necessidade ou não de um *buffer* de profundidade com o GLUT_DEPTH, os esquemas de cores com o GLUT_RGB ou GLUT_RGBA e por fim o GLUT_INDEX que define que as cores serão especificadas através de uma tabela, além disto é necessário algumas informações adicionais para a correta utilização da biblioteca GLUT, como a posição inicial da janela através da função *glutInitWindowPosition (int x, int y)* onde x e y indicam a posição através de um plano cartesiano medido em *pixels*, o tamanho da janela em *pixels* através da função *glutInitWindowSize (int width, int height)* onde *width* e *height* são respectivamente os valores da largura e altura da janela, o nome a ser exibido na barra de título da janela GLUT pela função *glutCreateWindow (char *string)* através da variável **string* e por fim a função *glutDestroyWindow (int win)* utilizada para destruir a janela GLUT, (Cohen & Manssour, 2006).

Já as funções de tratamentos de eventos também denominadas de *callbacks*, não necessitam da intervenção direta do programador para realizar tarefas, por exemplo, a interface com certos *hardwares* como teclados e *mouses*, porém é necessário que estas sejam “chamadas” no início do programa. Algumas destas funções merecem um destaque especial como: *glutDisplayFunc* que é utilizada para chamar a função responsável em executar os programas da *OpenGL*, a *glutReshapeFunc* chamada sempre que o programador altera o tamanho da janela durante a execução do programa, a *glutKeyboardFunc* que permite tratar de eventos referentes ao teclado desde que ele esteja configurado com os algarismos no padrão *ASCII*, assim como as funções *glutMouseFunc*, *glutMotionFunc* e *glutPassiveMotionFunc* que definem respectivamente, o tratamento dos eventos referentes aos botões, movimento quando alguma tecla estiver pressionada e do movimento do *mouse* quando nenhuma tecla estiver pressionada.

2.4.1.4 – Variáveis de Estado e Espaço de Trabalho

A *OpenGL* é uma máquina de estados, pois é possível colocá-la em vários estados (ou modos) que não são alterados, a menos que uma função seja chamada para isto. Esta máquina de estado é composta de muitas variáveis de estado que armazenam um determinado valor e podem ser alteradas pela chamada a uma função (Schereiner 2004).

Existem muitas variáveis de estado contidas na biblioteca *OpenGL* que definem diversas características das chamadas primitivas gráficas, que serão melhor explicadas no decorrer do trabalho, entre essas variáveis temos as que guardam o estilo e espessura da linha, posição da luz, propriedades do material e a matriz de transformação corrente.

A definição do espaço de trabalho será subdividida em dois tipos, a visualização bidimensional e a tridimensional. Porém antes disso teremos que entender o conceito de universo e como interpreta-lo, o Sistema de Referência do Universo (SRU) em *OpenGL* trabalha com um plano cartesiano, onde “x” é o eixo horizontal orientado de forma crescente e positiva da esquerda para direita, e o eixo “y” é o eixo vertical orientado de forma crescente e positiva de baixo para cima, estes dois eixos são perpendiculares e tem sua origem no centro geométrico do espaço de trabalho.

Entretanto o monitor trabalha com o chamado Sistema de Referência da Tela (SRT) que não trabalha exatamente da mesma forma que o SRU, no SRT a origem se localiza no canto superior esquerdo do espaço de trabalho e o eixo “y” é vertical, contudo este cresce positivamente de cima para baixo, os dois sistemas ficam bem exemplificados nas Figuras 2.6 e 2.7.

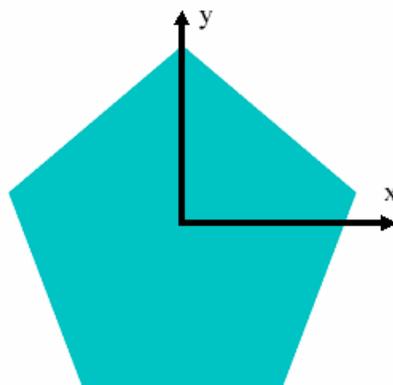


Figura 2.5 – Sistema de Referência do Universo (SRU).

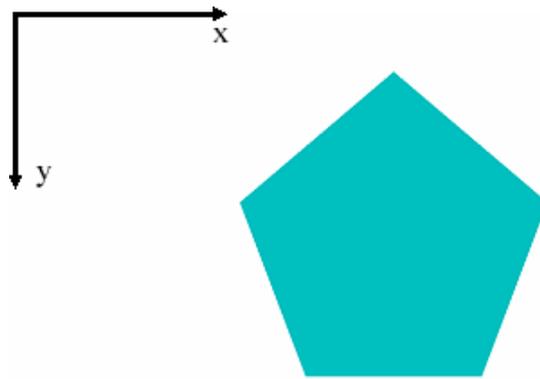


Figura 2.6 – Sistema de Referência da Tela (SRT).

Para que seja exibida levando-se em conta o sistema SRU e não o SRT é necessário realizar uma conversão entre a imagem no SRU para o SRT, o que é denominado de mapeamento e mostrado na Figura 2.8.

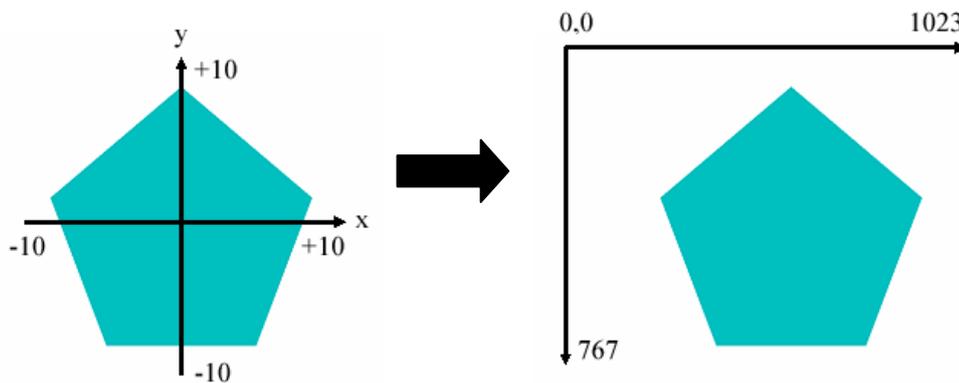


Figura 2.7 – Mapeamento da Imagem do SRU para SRT

No caso da visualização tridimensional o SRU corresponde a três eixos ortogonais (x, y, z), com a origem em (0, 0, 0), como pode ser visualizado na Figura 2.7, Para localizar o eixo z, basta recorrer a regra da mão direita ou da mão esquerda, no caso da *OpenGL* é utilizado a da mão direita como mostra a Figura 2.9.

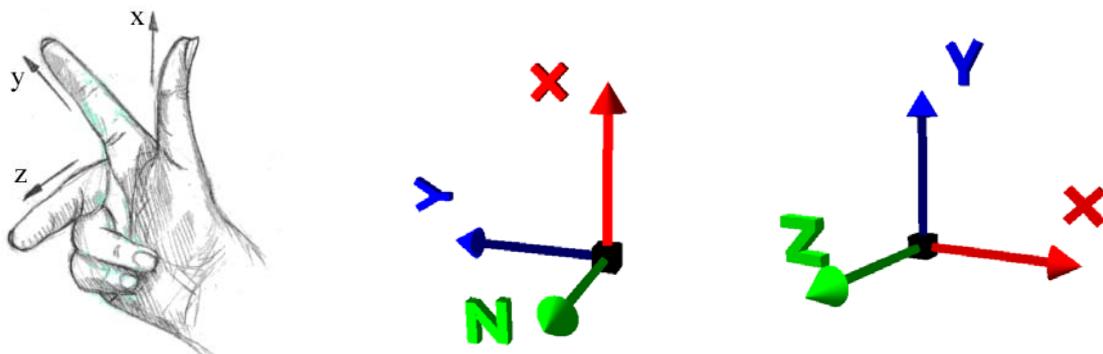


Figura 2.8 – SRU Tridimensional e a Regra da Mão Direita

A formação de imagens 3D em *OpenGL*, não é tão simples como o processo 2D e passa basicamente por três processos: definição da cena, a câmera sintética e a projeção.

A definição da cena consiste na criação e posicionamento de cada objeto em suas respectivas coordenadas estabelecidas previamente no corpo do programa, onde tudo é feito através de operações de escala, translação e rotação, os quais serão mais detalhados no decorrer deste trabalho.

O passo seguinte, a câmera sintética, consiste na criação de um observador virtual, o qual perceberá a imagem de um ponto fictício, pois o mesmo objeto 3D pode ser visto de diferente ângulos e infinitas distâncias em cada um de seus eixos (x, y ou z), o nome câmera vem do fato de ao se observar o objeto de um determinado ângulo com uma determinada distância seria como se estivesse tirando um foto.

Por fim teremos o processo de projeção que é o mapeamento dos objetos 3D para 2D, como os objetos 3D são feitos por inúmeros pontos, a projeção é realizada através de retas que passam pelos vértices do objeto e interceptam um plano, chamado de plano de projeção, este tipo de procedimento é conhecido como projeções geométricas planares e são subdivididas em duas: a Projeção paralela ortográfica que intercepta o plano de projeção a 90° e a projeção em perspectiva que tem como origem um único ponto, como ilustrado na Figura 2.10.

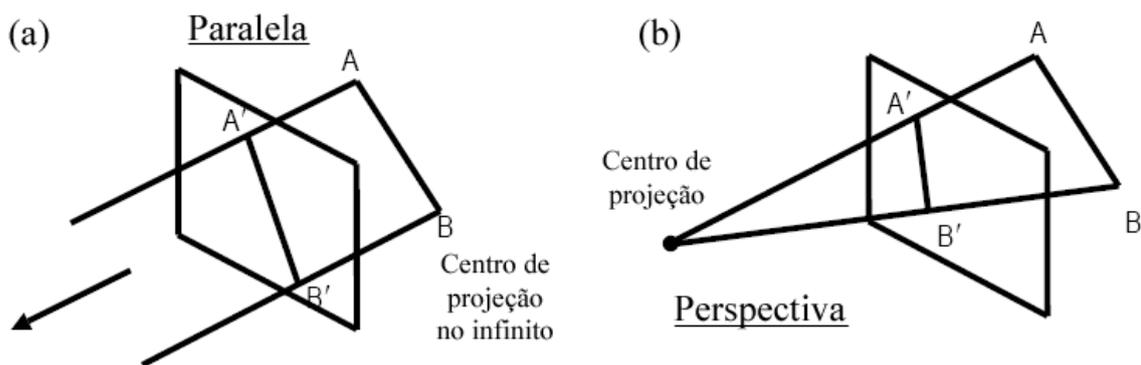


Figura 2.9 – (a) Projeção Paralela, (b) Projeção Perspectiva.

Outro ponto importante é conhecer quais comandos dentro da *OpenGL* podem fornecer tanto a câmera sintética quanto as perspectivas, a função *gluLookAt* (*GLdouble obsx*, *GLdouble obsy*, *GLdouble obsz*, *GLdouble alvox*, *GLdouble alvoy*, *GLdouble alvoz*,

$GLdouble\ upx$, $GLdouble\ upy$, $GLdouble\ upz$), os parâmetros $obsx$, $obsy$ e $obsz$ são utilizados para posicionar a câmera, os valores $alvox$, $alvoy$ e $alvoz$ são utilizados para informar onde está o objeto e por fim os valores upx , upy e upz são denominados de vetor up , que podem ocasionar o giro da imagem, como mostra a Figura 2.11

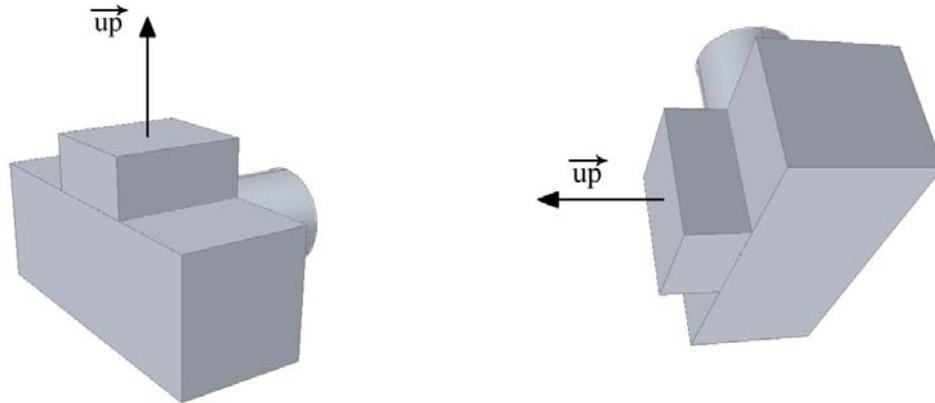


Figura 2.10 – Rotação da câmera sintética devido ao vetor up .

Já para a perspectiva utiliza-se a função $gluPerspective$ ($GLdouble\ fovy$, $GLdouble\ aspect$, $GLdouble\ zNear$, $GLdouble\ zFar$), o parâmetro $fovy$ determina o ângulo de abertura da visualização, o $aspect$ define a área de visualização na direção x, o $zNear$ que deve ter valor positivo determina a distância frontal do observador e por fim o $zFar$ que também deve ter valor positivo é a distância do observador até o plano de corte traseiro em z.

Após de todo o processo de criação e posicionamento dos objetos 3D é necessário realizar o mapeamento destes, dentro da *OpenGL* isto é realizado basicamente através de duas funções, a função $glLoadIdentity$ ($void$) utilizada para inicializar o sistema de coordenadas antes da execução de qualquer operação de manipulação de matrizes. Faz com que seja carregada a matriz identidade (Schreiner, 2004; Wright, 2000), e a função $glMatrixMode$ ($GLenum\ mode$) que permite identificar com qual matriz se vai trabalhar.

A seleção da matriz é feita por intermédio do parâmetro $mode$, que pode receber uma das seguintes constantes $GL_PROJECTION$, para selecionar a matriz de projeção; ou $GL_TEXTURE$ para selecionar a matriz de textura. Neste caso, todas as futuras alterações, tais como operações de escala, rotação e translação, irão afetar a matriz selecionada. Sem este comando, chamadas sucessivas da função de $gluPerspective$ ou da função responsável por desenhar os elementos poderiam resultar em uma corrupção do volume de visualização ou alteração indesejada do aspecto do(s) objeto(s) da cena (Schreiner, 2004; Wright, 2000).

2.4.1.5 – Primitivas Gráficas

Todo e qualquer desenho em *OpenGL* é na verdade um conglomerado das chamadas primitivas gráficas, que são na verdade pontos, retas e círculos, no caso de desenhos bidimensionais, essas primitivas gráficas são criadas utilizando um ou vários pares de vértices (Wright 2000), objetos e cenas criados em *OpenGL* consistem em um conjunto de primitivas gráficas simples que são combinadas para formar os modelos.

Para que seja possível criar esta lista de vértices utilizada na construção das primitivas gráficas, são necessários certos comandos os quais proporcionam ao compilador ler e interpretar estes valores. Os comandos em questão são o *glBegin(<argumento>)* o qual irá determinar qual objeto será desenhado e conseqüentemente quantos vértices serão necessários, no caso de uma reta, por exemplo, o valor do *<argumento>* recebe *GL_LINES*, logo deverão ser especificados dois vértices. A tabela 2.20 e a Figura 2.12 descrevem as primitivas gráficas.

Tabela 2.4 – Primitivas Gráficas em *OpenGL*

Valor	Descrição
<i>GL_POINTS</i>	Para desenhar pontos
<i>GL_LINES</i>	Para desenhar segmentos de linha
<i>GL_LINE_STRIP</i>	Para desenhar segmentos de linha conectados
<i>GL_LINE_LOOP</i>	Para desenhar segmentos de linha conectados, unindo o primeiro ao ultimo ponto
<i>GL_POLYGON</i>	Para desenhar um polígono convexo
<i>GL_TRIANGLES</i>	Para desenhar triângulos
<i>GL_TRIANGLE_STRIP</i>	Para desenhar triângulos conectados
<i>GL_TRIANGLE_FAN</i>	Para desenhar triângulos conectados a partir de um ponto central
<i>GL_QUADS</i>	Para desenhar quadriláteros
<i>GL_QUAD_STRIP</i>	Para desenhar quadriláteros conectados

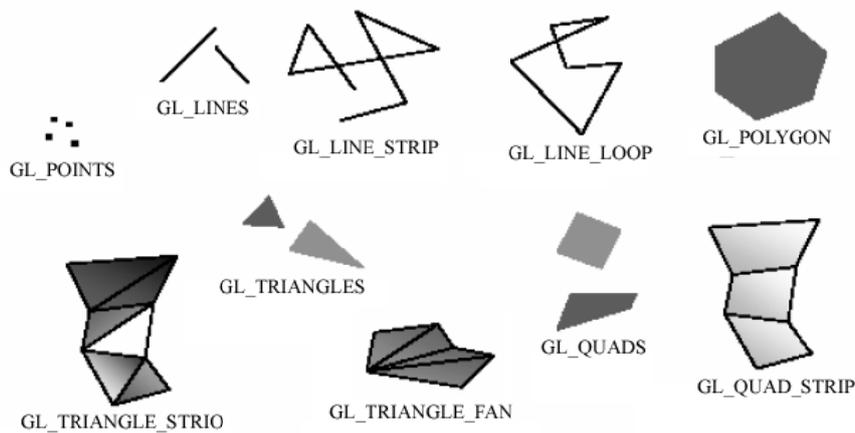


Figura 2.11 – Primitivas Gráficas 2D em *OpenGL*.

Existem ainda as primitivas responsáveis pela aparência, uma das mais importantes é a função *glColor3f* (*GLfloat red, GLfloat green, GLfloat blue*) trabalha com o modelo denominado de *RGB* (*Red, Green, Blue*) ou *RGBA* (*Red, Green, Blue, Alpha*), o processo de formação das cores se dá por adição de cada um dos seus componentes, variando da ausência de cor, o preto onde todas as suas variáveis seriam nulas, ou na intensidade máxima, branco onde a intensidade de cada cor é igual.

Mesmo passando para um plano tridimensional, as primitivas gráficas trabalham de forma semelhante as do plano bidimensional, a única diferença esta que ao se atribuir os parâmetros nas funções deve ser adicionada à coordenada *z*, como pode ser visto na Figura 2.13.

Tabela 2.5 – Primitivas Gráficas *GLUT* em *OpenGL*

Funções	Objeto
<i>glutWireCube</i> / <i>glutSolidCube</i>	Cubo
<i>glutWireCone</i> / <i>glutSolidCone</i>	Cone
<i>glutWireDodecahedron</i> / <i>glutSolidDodecahedron</i>	Dodecaedro
<i>glutWireIcosahedron</i> / <i>glutSolidIcosahedron</i>	Icosaedro
<i>glutWireOctahedron</i> / <i>glutSolidOctahedron</i>	Octaedro
<i>glutWireSphere</i> / <i>glutSolidSphere</i>	Esfera
<i>glutWireTeapot</i> / <i>glutSolidTeapot</i>	Bule
<i>glutWireTetrahedron</i> / <i>glutSolidTetrahedron</i>	Tetraedro
<i>glutWireTorus</i> / <i>glutSolidTorus</i>	Toroide

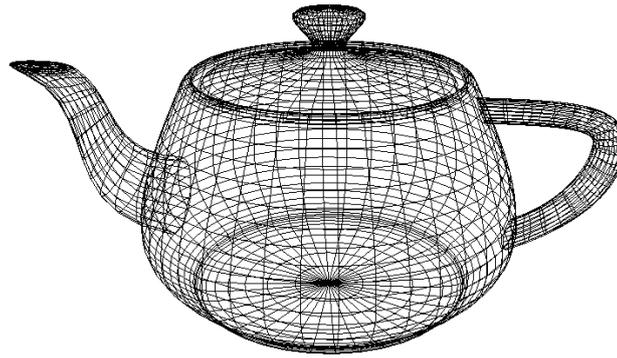


Figura 2.12 – Primitivas Gráficas 3D em *OpenGL*.

2.4.1.6 – Transformações Geométricas

Tendo visto todo o processo de criação, posicionamento e as funções gráficas da *OpenGL* é necessário entender como podemos “alterar” estes objetos dentro da *OpenGL* ou seja realizar transformações geométricas como translação, rotação e escala.

A translação é o resultado da adição de variáveis de deslocamento em todas as coordenadas do objeto original, como cada vértice corresponde a um ponto no SRU (Sistema de Coordenada do Universo), logo para que este tipo de transformação seja de fácil entendimento ele é sempre aplicado a partir da origem do sistema de referência, a função *glTranslated* ou *glTranlatef* (*GLfloat tx*, *GLfloat ty*, *GLfloat tz*) é a responsável por essa operação, onde cada um dos parâmetros é um valor de translação em sua respectiva coordenada, a diferença persiste apenas no tipo de variável utilizada *double* ou *float*, (Cohen & Manssour, 2006), conforme Figura 2.14.

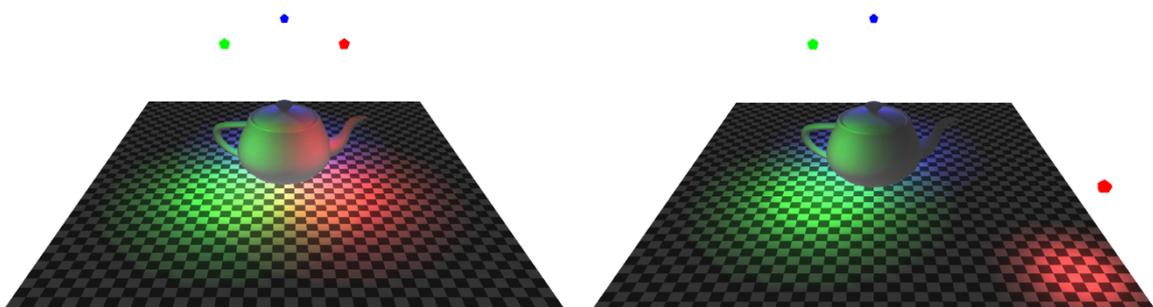


Figura 2.13 – (a) Pontos de iluminação, (b) Ponto vermelho translacionado.

Já a transformação geométrica de escala corresponde a um parâmetro que será multiplicado em uma das coordenadas do objeto, possibilitando assim o seu aumento ou diminuição, a função *glScalef* ou *glScaled* (*GLdouble ex*, *GLdouble ey*, *GLdouble ez*) é a responsável por essa operação, e trabalha de maneira semelhante a de translação, porém se aplicado valores negativos ocasionará o espelhamento do objeto, a Figura 2.15 demonstra como funciona o fator de escala, (Cohen & Manssour, 2006).

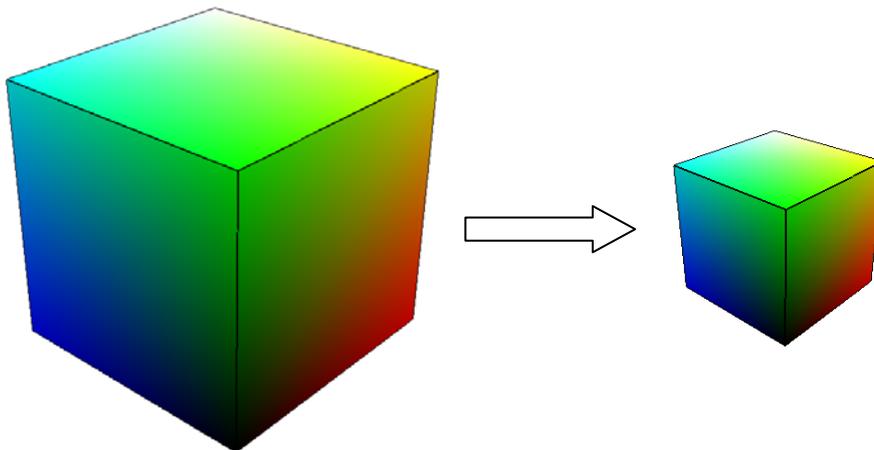


Figura 2.14 – Transformação geométrica de escala.

Por ultimo temos a rotação usada para definir o valor de um ângulo utilizado para girar o objeto, a função *glRotatef* ou *glRotated* (*GLdouble ângulo*, *GLdouble x*, *GLdouble y*, *GLdouble z*) é a responsável por essa operação, o parâmetro *ângulo* recebe o valor do ângulo a ser aplicado e os valores *x*, *y* e *z* são utilizados para indicar em qual dos eixos se dará a rotação, a Figura 2.16 mostra a rotação em um dos eixos, (Cohen & Manssour, 2006)..

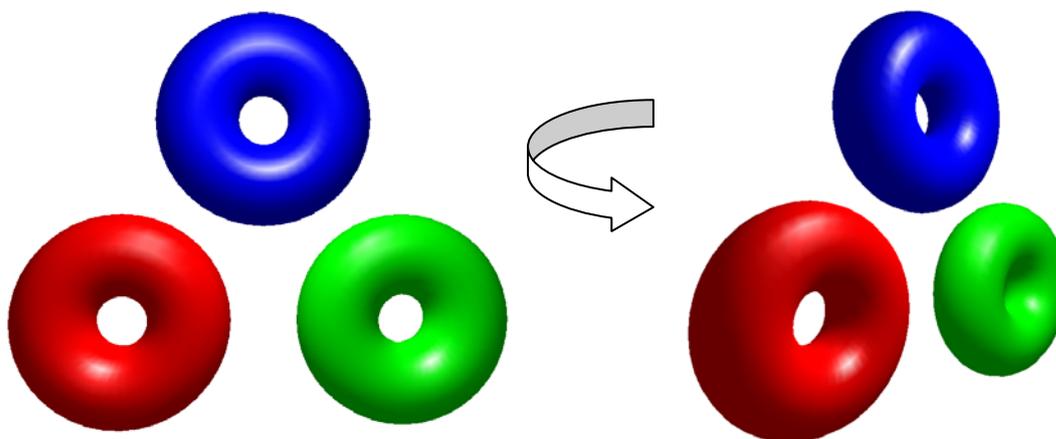


Figura 2.15 – Transformação geométrica de rotação.

2.4.1.7 – Zoom e Pan

Esta operação trabalha com a visualização de um ou vários objetos variando a distância entre eles e um observador virtual, no caso de uma aproximação (*zoom in*) ou de um distanciamento (*zoom out*). Para objetos em 2D não é suficiente aumentar ou diminuir o tamanho da *viewport* é necessário também alterar o tamanho da janela (*window*). No caso em 3D o processo é um pouco diferente, pois depende do centro de projeções das imagens o que leva a formação de um ângulo de visualização, logo quanto menor for este ângulo menor será a visualização do universo, *zoom in*, e quanto maior o ângulo maior será a visualização do universo, *zoom out*. Outra operação muito utilizada em programas gráficos é a operação de *pan*, que consiste em mover a *viewport* de modo que o usuário possa navegar pelo universo em que esta trabalhando, (Cohen & Manssour, 2006), exemplificado nas Figuras 2.17 e 2.18.

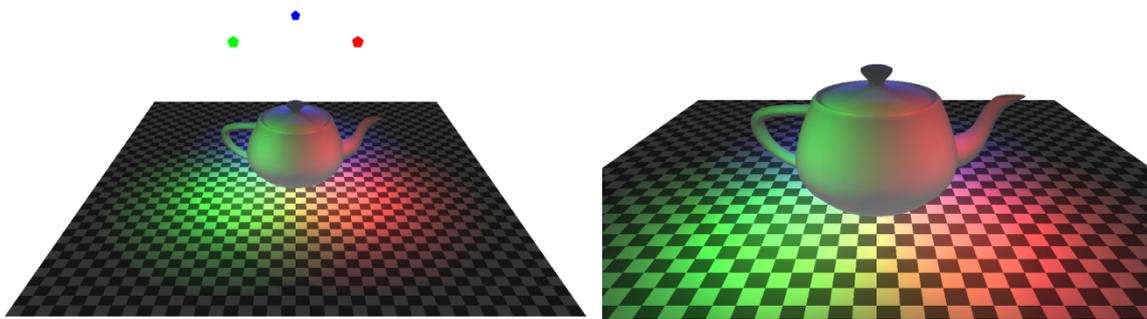


Figura 2.16 – (a) Cenário 3D, (b) Zoom In.

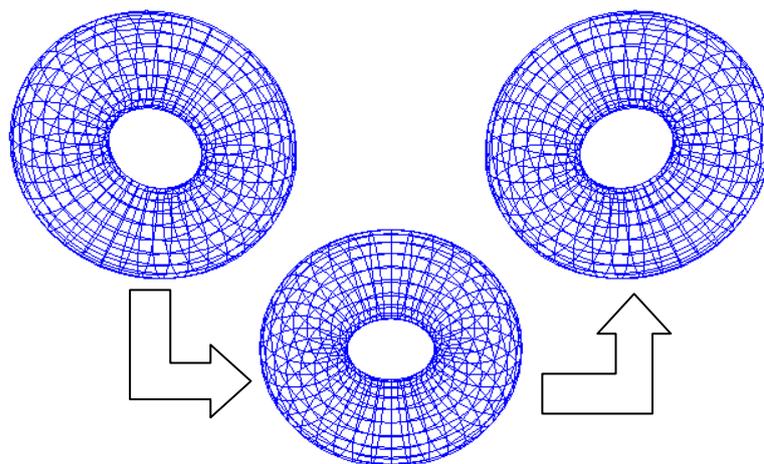


Figura 2.17 – Exemplo de Pan.

2.4.2 – Win32 API

2.4.2.1 – Uma Breve História do *Windows*

Após o surgimento do IBM PC em 1981, ficou claro que sistema operacional para os computadores seria *MS-DOS* que era o principal produto da Microsoft. O MS.DOS era um sistema operacional básico comparado com os sistemas atuais. Permitia que o usuário utiliza-se certos comandos como DIR e executasse algumas aplicações carregadas na memória.

Devido a restrições de hardware e na capacidade de memória, ambientes gráficos e sofisticados estavam muito longes de se tornarem uma realidade até que em 1983 a Macintosh com o Lisa estipulou um novo padrão de caracteres para computadores e em 1984 apresentou um novo padrão gráfico que se tornou um marco na história da informática e da computação gráfica. Porém os pioneiros no desenvolvimento de interfaces gráficas e percussores tanto do Mac quanto do *Windows* foi o trabalho da Xerox Palo Alto Research Center (PARC) em meados de 1970.

O *Windows* foi apresentado pela Microsoft em novembro de 1983 (posterior ao Lisa mas antes do Macintosh) e foi liberado dois anos depois em novembro de 1985. Durante os dois anos seguintes o Microsoft *Windows* 1.0 foi seguido por várias atualizações para suportar o mercado internacional de dispositivos gráficos como impressoras e placas de vídeo, posteriormente em 1987 uma nova versão foi liberada, *Windows* 2.0, a qual trouxe algumas inovações que para o usuário atual são muito comuns como o de sobrepor janelas e interfaces com hardwares, como *mouse* e teclado, mais elaboradas.

Até este ponto o *Windows* precisava apenas de um processador Intel 8086 ou 8088, 1 megabyte (MB) de memória para trabalhar, poucas mudanças ocorreram nas versões posteriores até a versão 3.0 lançada em 22 de maio de 1990, que foi capaz de trabalhar com 16 megabytes de memória e foi a primeira versão do *Windows* a ganhar tantos os mercados de usuários comuns como os de escritórios.

A partir deste momento o *Windows* se popularizou de maneira exponencial em todo o mundo com suas versões 3.1, 3.11, 95, 98, Milenium , XP e mais recentemente o *Windows*

Vista, Somando todas as versões o *Windows* abrange cerca de 95% dos computadores em todo mundo, (Petzold, 1996).

2.4.2.2 – Como funciona a *Win32* API

Esta biblioteca possibilita a utilização de todas as funções pertinentes aos sistemas operacionais da família *Windows* (sistemas baseados no *Win32*). Possibilitando assim o desenvolvimento de aplicações que poderão ser executadas em qualquer versão do *Windows*.

A criação de programas para *Windows* é um pouco diferente da programação de aplicativos console para *MS-DOS*, sendo à primeira vista algo mais complexo, pois o próprio *Windows* em si é um sistema mais complexo e avançado que o *MS-DOS*, na Figura 2.19 a tela do *MS-DOS*.



Figura 2.18 – Tela do *MS-DOS*

O *Windows* executa os programas através do processamento de mensagens, que ocorre da seguinte maneira: o programa que está sendo executado aguarda o recebimento de uma mensagem do *Windows* e quando o mesmo recebe a mensagem (por uma função especial do *Windows*), espera-se que o programa execute alguma ação para processar essa mensagem.

Existem diversas mensagens que o *Windows* pode enviar ao programa; por exemplo, quando o usuário modifica o tamanho da janela do programa, um clique no botão do *mouse*, a finalização do programa, enfim, para cada ação realizada pelo usuário (e conseqüentemente pelo *Windows*), uma mensagem é enviada para o programa processar. Obviamente, não é toda mensagem que deverá ser processada; se o seu programa utilizar

somente o teclado, podemos descartar todas as mensagens enviadas sobre as ações do *mouse*.

O *Win32* API é o acesso de mais baixo nível que se pode ter ao sistema (considerando o modo usuário), logo ela deve possibilitar que o usuário possa fazer tudo que é possível no *Windows*. Por isso, muitas vezes, chama-se uma função de 8 parâmetros mais só 2 são utilizados, os outros 6 parâmetros só serão usados em situações não muito comuns, mas que devem ser cobertas pela API.

Na programação utilizando a *Win32*, é o sistema quem inicia a interação com seu programa (ao contrário do que ocorre em programas *MS-DOS*) e apesar de ser possível fazer chamadas de funções da *Win32* API em resposta a uma mensagem, ainda é o *Windows* quem inicia a atividade.

2.4.2.3 – Início e Terminação de Programas

Essa API de programação é constituída por um elevado número de serviços ou funções, definidos utilizando a linguagem de programação “C”. No decorrer deste capítulo serão discutidos alguns destes aspectos.

Quando se solicita ao Sistema Operacional, no caso o *Windows*, a execução de um novo programa (serviço *CreateProcess()* em *Win32*), este começa por executar uma rotina (no caso de programas em “C”) designada por “C” *startup*. Esta rotina é a responsável por chamar a função *main()* do programa, passando-lhe alguns parâmetros, se for caso disso, e por abrir e disponibilizar três arquivos ao programa: os chamados *standard input*, *standard output* e *standard error*. O *standard input* fica normalmente associado ao teclado (exceto no caso de redirecionamento), enquanto que o *standard output* e o *standard error* ficam normalmente associados ao monitor.

A função *main()* pode ser definida num programa em “C” de muitas formas:

int ou *void main(void)*

int ou *void main(int argc)*

int ou *void main(int argc, char *argv[])*

int ou *void main(int argc, char *argv[], char *envp[])*

Assim pode ser definida como procedimento (*void*) ou como função que retorna um inteiro (*int*). Neste último caso o inteiro é passado a quem chamou a função, ou seja, à rotina de “C” *startup*, que por sua vez o transmite ao Sistema Operacional.

Quando se “chama” um programa é possível passar-lhe parâmetros, que são um conjunto de 0 ou mais *strings*, separadas por espaços. Esses parâmetros podem depois ser adicionados na função *main()* através dos seus argumentos *argc* e *argv*.

argc, número de argumentos passados, incluindo o próprio nome do programa. *argv*, *array* de ponteiros para *string*, apontando para os parâmetros passados ao programa. O array contém um número de elementos igual à “*argc+1*”. O primeiro elemento de *argv[]* aponta sempre para o nome do programa (podendo incluir todo o *path*). O último elemento de *argv[]* contém sempre o apontador nulo (valor *NULL*), exemplificado na Figura 2.20.

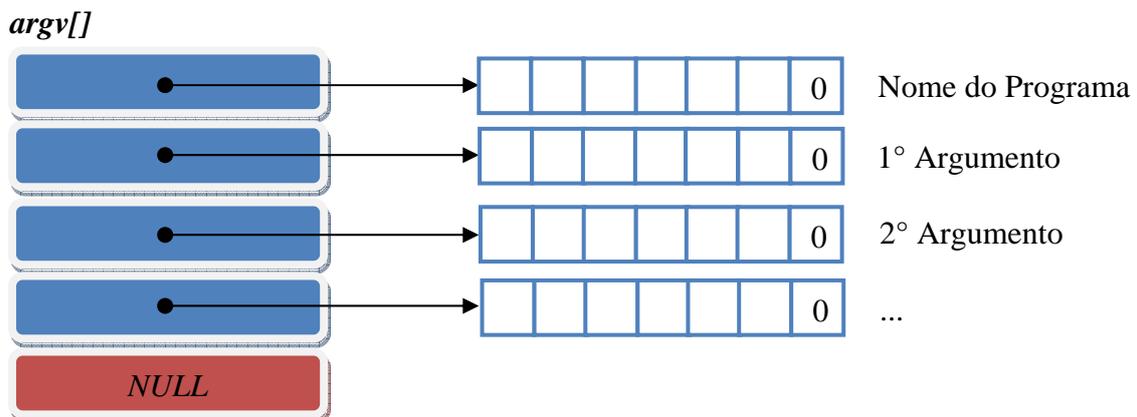


Figura 2.19 – Passagem de parâmetros para programas externos.

Um programa em “C” termina quando a função *main()* retorna (usando “*return expressão*”, no caso de ter sido definida como *int*, e usando simplesmente “*return*”, ou deixando chegar ao fim das instruções, no caso de ter sido definida como *void*). Outra possibilidade é chamar diretamente as funções que terminam o programa, definidas na biblioteca *standard* do “C”:

```
#include <stdlib.h>  
void exit(int status);
```

A instrução ***void exit(int status)*** termina imediatamente o programa retornando para o sistema operacional o código de terminação ***status***. Além disso, executa a liberação de todos os recursos alocados ao programa, fechando todos os arquivos e armazenando os dados que ainda não tivessem sido transferidos para o disco.

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

```
void _exit(int status);
```

A instrução ***void _exit(int status)*** termina imediatamente o programa retornando para o sistema operacional o código de terminação ***status***. Além disso, executa a liberação de todos os recursos alocados ao programa de forma rápida, podendo perder dados que ainda não tivessem sido transferidos para o disco.

Quando o programa é terminado pelo retorno da função *main()* o controle passa para a rotina de “C” *startup* (que “chamou” *main()*); esta por sua vez acaba por “chamar” *exit()* e esta chama por fim *_exit()*.

A função *exit()* pode executar, antes de terminar, uma série de rotinas (*handlers*) que tenham sido previamente registradas para execução no final do programa. Estas rotinas são executadas na ordem inversa.

O registro destes *handlers* de terminação é feito por outra função da biblioteca *standard* da linguagem:

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

```
int atexit(void (*func)(void));
```

A função ***int atexit(void (*func)(void))*** registra o *handler* de terminação ***func*** (função *void* sem parâmetros). Retorna 0 em caso de sucesso e um valor diferente de 0 em caso de erro. Na figura 2.21 percebe-se um esquema dos processos de início e terminação de um programa em “C”.

As funções *exit()* e *_exit()* podem ser vistas como serviços do sistema operacional. De fato correspondem as chamadas diretas Win32 (*ExitProcess()* e *TerminateProcess()*). O Caminho de 1 até o 9 é o mais comum de ocorrer.

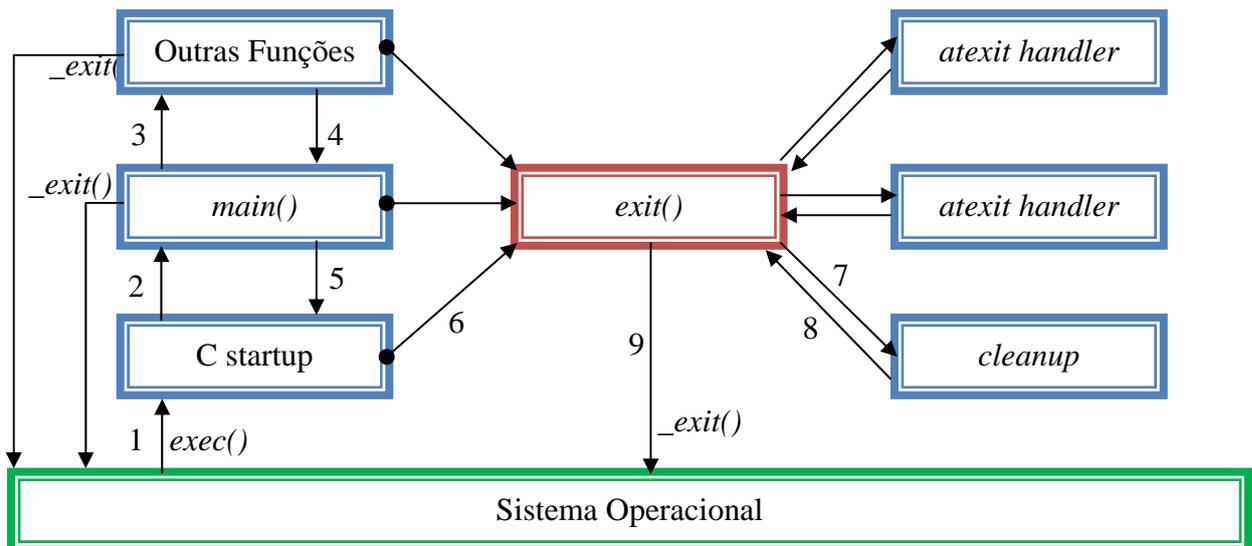


Figura 2.20 – Fluxo de informações dentro do programa.

2.4.2.4 – Windows Messages

Toda a arquitetura do *Windows* é baseada em objetos gráficos que o sistema operacional define como janela, e todo o fluxo de informações entre estes objetos são chamados de mensagens, onde cada uma tem parâmetros em particular.

Vale ressaltar que objetos como *Button*, *RadioButon*, *Edit*, *Static*, *GroupBox* e *ComboBox* são interpretados pelo *Windows* como janelas, onde todos eles são criados pela mesma função, a *CreateWindow*, variando apenas os parâmetros de classe, toda a janela de um programa *Windows* tem um procedimento associado. Esse procedimento é uma função que poderia estar dentro do programa. O *Windows* envia uma mensagem para uma janela chamando o seu procedimento de janela, o qual executa uma tarefa, baseando-se na mensagem e em seguida retorna o controle ao *Windows*.

O procedimento de janela é o código que define o processamento de mensagens para que a janela possa executar tarefas baseadas nas mensagens enviadas pelo *Windows*.

Os programas *Windows* possuem vários procedimentos de janelas, em que várias mensagens são declaradas para a comunicação com o *Windows*. Na maioria das vezes, essas mensagens informam a janela que contém um botão de seleção, por exemplo, sabe que ele foi pressionado. Outras mensagens informam à janela quando ela teve seu tamanho

alterado ou quando precisa ser redesenhada. Abaixo são apresentadas algumas das mensagens utilizadas no desenvolvimento do programa de interface:

- a) WM_CREATE – Esta mensagem é enviada quando a aplicação requisitar que uma janela seja criada através da função *CreateWindowEx* ou *CreateWindow*. O procedimento de janela recebe esta mensagem depois que a janela foi criada, mas antes que ela se torne visível.
- b) WM_DESTROY – Esta mensagem é enviada quando a aplicação requisitar que uma janela seja destruída. O procedimento de janela recebe esta mensagem depois que a janela foi removida da tela.
- c) WM_COMMAND – Esta mensagem é enviada quando o usuário seleciona algum item de comando referente a um *menu*.
- d) WM_SIZE – Esta mensagem é enviada para a janela depois que o seu tamanho é alterado.

2.4.2.5 – Criação de uma Janela

O principal objetivo do uso da biblioteca *Win32* é justamente o uso de janelas, na maioria dos programas existe uma janela principal, onde são mostrados recursos como botões, textos, mensagens, menus e até mesmo outras janelas secundárias, por exemplo, é necessário levantar alguns aspectos importantes do código por traz da biblioteca para que o seu entendimento seja mais fácil.

Além dos elementos citados anteriormente, que podem ser adicionados pelo usuário a janela principal pode conter botões de maximização, minimização e de finalização, logo ela pode ser redimensionada utilizando o *mouse*, assim como pode conter um nome estipulado pelo usuário em sua barra de título. Logo abaixo será apresentado o código para criação de uma janela simples utilizando a *Win32* API (Costa, 2005).

```
#include <Windows.h>
```

```
LRESULT CALLBACK ProcJan (HWND, UINT, WPARAM, LPARAM);
```

```
char szNomeAplic[] = "Principal";
```

```
int WINAPI WinMain (HINSTANCE hCopia, HINSTANCE hCopiaAnt, LPSTR  
szLinhaCmd, int iCmdMostrar)
```

```
{  
    HWND hjan;  
    MSG msg;  
    WNDCLASS classejan;
```

```
    classejan.style = CS_HREDRAW | CS_VREDRAW;  
    classejan.lpfnWndProc = ProcJan;  
    classejan.cbClsExtra = 0;  
    classejan.cbWndExtra = 0;  
    classejan.hInstance = hCopia;  
    classejan.hIcon = LoadIcon (NULL,  
IDI_APPLICATION);  
    classejan.hCursor = LoadCursor (NULL, IDC_ARROW);  
    classejan.hbrBackground = GetStockObject (WHITE_BRUSH);  
    classejan.lpszMenuName = NULL;  
    classejan.lpszClassName = szNomeAplic;
```

```
    if (!RegisterClass (&classejan)) return 0;
```

```
    hjan = CreateWindow (szNomeAplic, "Janela Principal",  
WS_OVERLAPPEDWINDOW, CW_USEDEFAULT, CW_USEDEFAULT,  
CW_USEDEFAULT, CW_USEDEFAULT, HWND_DESKTOP, NULL, hCopia, NULL);
```

```
    ShowWindow (hjan, iCmdMostrar);
```

```
    UpdateWindow (hjan);
```

```
    while (GetMessage (&msg, NULL, 0, 0))
```

```
    {  
        TranslateMessage (&msg);  
        DispatchMessage (&msg);  
    }
```

```
    return msg.wParam;
```

```
}
```

```
LRESULT CALLBACK ProcJan (HWND hjan, UINT iMsg, WPARAM wParam, LPARAM  
lParam)
```

```
{  
    switch (iMsg)  
    {  
        case WM_DESTROY: PostQuitMessage (0); return 0;}
```

```
    return DefWindowProc (hjan, iMsg, wParam, lParam);}
```

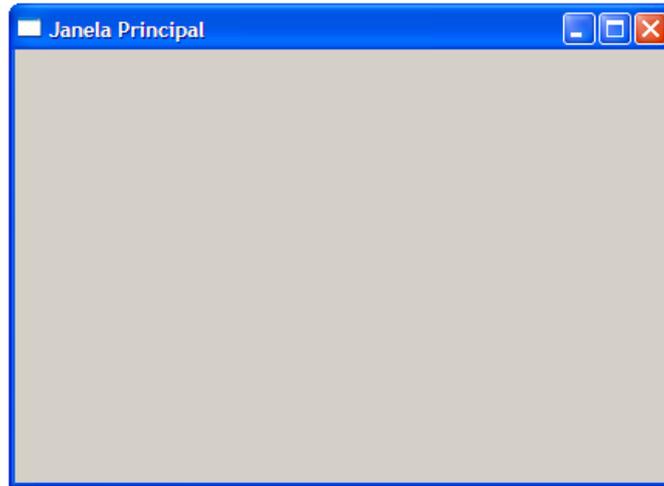


Figura 2.21 – Janela Principal criada a partir do código acima.

Percebe-se que o programa *WINPRIN.C* tem basicamente duas funções, *WinMain* e *ProcJan*, entre elas a função *WinMain* é a principal e todo programa em *Windows* deve ter esta função, já a função *ProcJan* é o procedimento que contém as mensagens enviadas ao *Windows* que definem a janela principal, seu tamanho e posição entre dezenas de outros parâmetros possíveis.

Nota-se logo na primeira linha do programa o arquivo ***WINDOWS.H***, que é o arquivo de cabeçalho do *Windows*, neste arquivo de cabeçalho encontram-se muitas declarações de constantes, funções utilizadas pelo programa principal, declarações de variáveis, estruturas.

Logo após a declaração do cabeçalho *Windows.h* temos a declaração antecipada da função *ProcJan*:

```
LRESULT CALLBACK ProcJan (HWND, UNIT, WPARAM, LPARAM);
```

Isto é um padrão dos programas que utilizam a *Win32*, onde as funções devem ser declaradas logo no começo do programa. A seguir temos:

```
char szNomeAplic[ ] = "Principal";
```

Este comando serve para definir o nome da janela que será utilizada no programa, e definir quais itens serão relacionadas a esta janela como botões e menus. Logo após temos:

```
Int WINAPI WinMain (HINSTANCE hCopia, HINSTANCE hCopiaAnt, LPSTR szLinhaCmd, int iCmdMostrar)
```

Esta função utiliza uma seqüência de chamadas *WINAPI* e retorna um inteiro ao *WINDOWS* quando termina, esta função deve se chamar *WinMain*.

- a) O parâmetro *hCopia* é o “identificador da cópia” é por este número que o *Windows* controla qual programa esta em execução.
- b) O parâmetro *hCopiaAnt* é um comando obsoleto, utilizado em versões anteriores ao *Windows 95* ele referenciava o identificador da ocorrência mais recente do programa que ainda estava ativa, nas versões mais recentes do cabeçalho *WINDOWS.H* este parâmetro é sempre nulo.
- c) O parâmetro *szLinhaCmd* é um ponteiro para uma variável de texto (string) terminada em zero (finalizada pelo caractere “\0”), que contém os parâmetros da linha de comando passados ao programa. É possível executar um programa *Windows* a partir do prompt do *MS-DOS*, ou digitando o nome do programa e o parâmetro na caixa de dialogo executar, invocada a partir do menu Iniciar.
- d) O parâmetro *iCmdMostrar* é um numero que indica como a janela deve ser inicialmente exibida no *Windows*, que pode ser um valor de 1 a 7 (valores declarados em *WINDOWS.H* por identificadores *SW (ShowWindow)* em que *SW_SHOWNORMAL = 1*- janela normal e *SW_SHOWMINNOACTIVE = 7* – inicialmente minimizado.

A partir deste ponto é necessário registrar as classes da janela, a janela é sempre criada com base em uma classe, que identifica o procedimento e processa as mensagens, portanto mais de uma janela pode ser criada a partir de uma única classe. Primeiramente é necessário declarar os identificadores da janela, das mensagens e da estrutura *WNDCLASS*, esta estrutura faz parte do cabeçalho *WINDOWS.H* e é utilizada para registrar a classe da janela através da função *RegisterClass*, (Costa, 2005).

```
HWND      hjan;  
MSG      msg;  
WNDCLASS classejan;
```

A estrutura *WNDCLASS* deve ser definida da seguinte maneira:

```
classejan.style = CS_HREDRAW | CS_VREDRAW;  
classejan.lpfWndProc      = ProcJan;  
classejan.cbClsExtra = 0;  
classejan.cbWndExtra      = 0;  
classejan.hInstance = hCopia;  
classejan.hIcon      = LoadIcon (NULL, IDI_APPLICATION);  
classejan.hCursor    = LoadCursor (NULL, IDC_ARROW);  
classejan.hbrBackground = GetStockObject (WHITE_BRUSH);  
classejan.lpszMenuName = NULL;  
classejan.lpszClassName = szNomeAplic;
```

O primeiro parâmetro da estrutura *classejan.style* define o estilo da janela, onde os dois identificadores *CS_HREDRAW* e *CS_VREDRAW* separados pelo operador *or* (|) da linguagem “C”, indicam que todas as janelas criadas com base nessa classe devem ser completamente repintadas sempre que o tamanho horizontal da janela (*CS_HREDRAW*) ou o tamanho vertical (*CS_VREDRAW*) for alterado.

O segundo parâmetro indica que para essa janela o procedimento é o *ProcJan*, e este irá processar todas as mensagens para todas as janelas criadas com base nessa classe. O terceiro e quarto parâmetro definem um espaço extra na estrutura *WNDCLASS* e que podem ser usados com propósitos específicos como isto não acontece neste caso então eles recebem zero, o quinto parâmetro determina o identificador da cópia do programa, o sexto identifica um ícone para todas as janelas baseadas nessa classe, o sétimo especifica um cursor para o *mouse*, caso o valor *IDC_ARROW* seja usado o próprio *Windows* define o cursor a ser utilizado, o oitavo especifica a cor do fundo da tela a ser usada, o nono parâmetro indica o *menu* da classe a ser utilizada como no exemplo acima não contém

menu logo esse valor recebe *NULL* e por fim o décimo parâmetro da estrutura especifica o nome da classe da janela, onde este nome é armazenado em *szNomeAplic*.

Após a declaração destes parâmetros é necessário registrar a classe através da linha de comando mostrada abaixo.

```
if(!RegisterClass (&classejan)) return 0;
```

Nota-se que apenas após o registro da janela é possível utilizar o comando *CreateWindow*, o qual necessita que todos os parâmetros definidos em *RegisterClass*, porém a criação da janela é feita da seguinte forma.

```
hjan = CreateWindow (szNomeAplic, "Janela Principal",  
WS_OVERLAPPEDWINDOW, CW_USEDEFAULT, CW_USEDEFAULT,  
CW_USEDEFAULT, CW_USEDEFAULT, HWND_DESKTOP, NULL, hCopia,  
NULL);
```

onde:

- a) *szNomeAplic* – nome da classe da janela. Por ele a janela é associada à classe.
- b) “*Janela Principal*” – nome que aparece na barra de título da janela.
- c) *WS_OVERLAPPEDWINDOW* – estilo normal da janela, cujo valor define que o programa deve ter o padrão básico: barra de título, caixa de menu do sistema à esquerda da barra título, botões de maximização, minimização e de fechamento à direita da barra de título, e uma moldura de alteração de tamanho. Este é o valor-padrão. Contudo, podem ser definidos valores próprios, para a janela só conter o que for desejado.
- d) *CW_USERDEFALT* – este valor define a posição inicial e o tamanho da janela, neste caso o *Windows* define aleatoriamente estes valores, mas podem ser especificados através de quatro valores numéricos.
- e) *HWND_DESKTOP* – identificador da janela mãe a que pertence a janela que está sendo criada. Esse valor definido determina que a janela pertence ao *desktop* do *Windows*. Esse parâmetro pode ser definido apenas como *NULL*, identificando o valor-padrão, sendo igual à *HWND_DESKTOP*.

- f) *NULL* – identificador do menu da janela é definido com este valor, desde que o programa não tenha um menu.
- g) *hCopia* – identificador da cópia do programa. Está associado ao identificador da cópia passado ao programa como um parâmetro de *WinMain*.
- h) *NULL* – parâmetro de criação. Esse parâmetro é definido como um ponteiro para a janela criada através da estrutura *CREATESTRUCT*. Esse ponteiro é referenciado ao parâmetro *lParam* de mensagem *WM_CREATE*.

Logo definida a função *CrteWindow*, exibe-se a janela por meio das funções:

```
ShowWindow (hjan, iCmdMostrar);
```

```
UpdateWindow (hjan);
```

A função *ShowWindow* tem dois parâmetros: o identificador da janela e o valor *iCmdMostrar* passado à *WinMain* que define como a janela será exibida inicialmente: normalmente (*SW_SHOWNORMAL*) ou minimizada na barra de tarefas (*SW_SHOWMINNOACTIVE*). A segunda função faz com que a área do cliente seja pintada.

Com a janela já criada, o procedimento desta deve ser ativado para que o programa possa receber entradas do usuário pelo teclado *mouse*. Isso é feito pelo laço de mensagens abaixo:

```
while (GetMessage (&msg, NULL, 0, 0))
    {
        TranslateMessage (&msg);
        DispatchMessage (&msg);
    }
return msg.wParam;
}
```

O laço de mensagens é criado para que o *Windows* possa traduzir os eventos de entrada do programa, como posicionamento dos botões do *mouse* ou do teclado. Para cada programa em execução, o *Windows* mantém uma fila de mensagens. Cada evento de entrada o

Windows traduz para uma mensagem, colocando-a na fila de mensagens para ser executada.

Este laço fica por conta da função *while {...}*. O parâmetro de teste do laço de mensagens é a chamada à função *GetMessage*, que retira uma mensagem da fila. A função *GetMessage* utiliza quatro parâmetros: ponteiro para o *Windows* da variável *msg* (que é uma estrutura) e três parâmetros que estão definidos como *NULL*, 0 e 0, indicando que são retornadas todas as mensagens de todas as janelas criadas pelo programa.

Todas as mensagens tem um identificador no arquivo de cabeçalho *Windows.h* que inicia com o prefixo *WM_* de *Windows Message*.

As declarações internas ao laço determinam respectivamente: passagem da estrutura *msg* de volta ao *Windows* para tradução de algum evento e passagem da estrutura *msg* de volta ao *Windows* para transmissão de mensagens ao procedimento de janela apropriado, para seu processamento (*Windows* chama o procedimento da janela).

O procedimento de janela sempre é definido como:

LRESULT CALLBACK ProcJan (HWND hjan, UINT iMsg, WPARAM wParam, LPARAM lParam)

Os quatro parâmetros do procedimento de janela são:

- a) *Hjan* – identificador para a janela que está recebendo a mensagem.
- b) *iMsg* – numero que identifica a mensagem.
- c) *wParam* – fornece uma informação sobre a mensagem.
- d) *lParam* – fornece outra informação sobre a mensagem.

Os dois últimos parâmetros são conhecidos como parâmetros da mensagem e contêm informações específicas para cada tipo de mensagem. Por exemplo: as posições x e y do *mouse* podem ser informações contidas nesses dois parâmetros.

O procedimento de janela recebe mensagens que são identificadas por um número que é um parâmetro para *iMsg*. Para o procedimento de janela processar uma mensagem recebida precisa determinar que mensagem foi recebida. Geralmente isso é feito através das funções *switch* e *case*. Assim, a função *switch* avalia *iMsg* e, dependendo de qual identificador ela contém (detectado pela função *case*), processa a devida mensagem. No programa *Winprinc.C*, o procedimento de janela é o seguinte:

```
switch (iMsg)
{
    case WM_DESTROY: PostQuitMessage (0); return 0;}
}
```

Nota-se que o único identificador de mensagem que o procedimento de janela processa é o que define o término do programa *WM_DESTROY* : pressão sobre o botão de término do programa localizado ao lado direito da barra de título (X); pressionamento das teclas *Alt+F4* conjuntamente; duplo no menu do sistema ao lado esquerdo da barra de título onde se encontra o ícone do programa ou abrindo o menu do sistema e selecionando *Fechar*.

Finalmente, todas as mensagens que o procedimento de janela não processar precisam ser passadas à função *DefWindowProc* da seguinte forma:

```
return DefWindowProc (hJan, iMsg, wParam, lParam);}
```

O valor devolvido por essa função precisa ser retornado dentro do procedimento de janela. É essencial chamar a função para efetuar o procedimento-padrão de todas as mensagens que o procedimento de janela não processar, (Costa, 2004).

2.4.2.6 – Arquivos de Recursos

A maioria dos programas para *Windows* possui menus, ícones personalizados (alguns incluem até cursores modificados) e caixas de diálogo (janelas como a da formatação de fonte em um editor de textos). Todos esses componentes (e mais alguns) do programa podem ser utilizados através do uso de arquivos de recursos (também conhecidos como *scripts* de recursos).

Arquivos de recursos podem ser criados em qualquer editor de textos no formato *ASCII* (como o Bloco de Notas), dentro da *IDE* do seu compilador, quando o mesmo possuir um editor de recursos (caso do *Microsoft Visual C++* e *Dev-C++*) ou através de um programa editor de recursos. No caso de editores de recursos com interface visual, o conteúdo do arquivo é gerado automaticamente pelo editor.

Esses arquivos têm extensão *.rc* e o conteúdo é definido com palavras chave em inglês, indicando os recursos que serão utilizados pelo programa. Os arquivos são então compilados, criando-se arquivos binários com extensão *.res*, que são anexados ao final do programa executável, podendo ser carregados em tempo de execução, conforme o esquema da Figura 2.22.

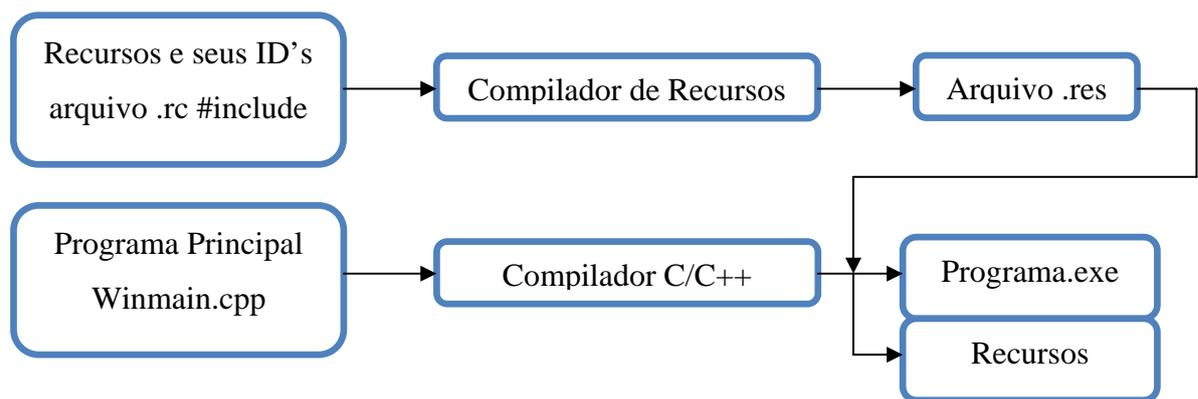


Figura 2.22 – Funcionamento dos arquivos de recurso.

2.4.2.7 – Elementos Gráficos

a) *Static*

Para criar um *static* é necessário usar a classe “*static*” na função *Create window*, este tipo de elemento não aceita a intervenções de *mouse* ou teclado e não podem receber comandos enviados através de mensagem pelo *WM_COMMAND*.

Quando se move ou clica com o *mouse* em cima de um *static*, este envia a mensagem *WM_NCHITTEST* e retorna o valor *HTTRANSPARENT* para o *Windows*, causando assim que o *Windows* envie a mesma mensagem, *WM_NCHITTEST*, a qual é convertida em uma

mensagem de localização para o *mouse*. A seguir é mostrado o código necessário para criar o *static* e a Figura 2.24 ilustra a criação de um *static*.

```
S_Arc_Lentgh_Initial = CreateWindowEx (  
    0,  
    "STATIC",  
    "Arc Lentgh Initial",  
    WS_VISIBLE/WS_CHILD,  
    122, 80, 80, 18,  
    hwnd,  
    NULL,  
    g_inst,  
    NULL  
);
```

```
SendMessage((HWND) S_Arc_Lentgh_Initial,(UINT) WM_SETFONT, (LPARAM)  
GetStockObject(DEFAULT_GUI_FONT),(LPARAM) lParam);
```

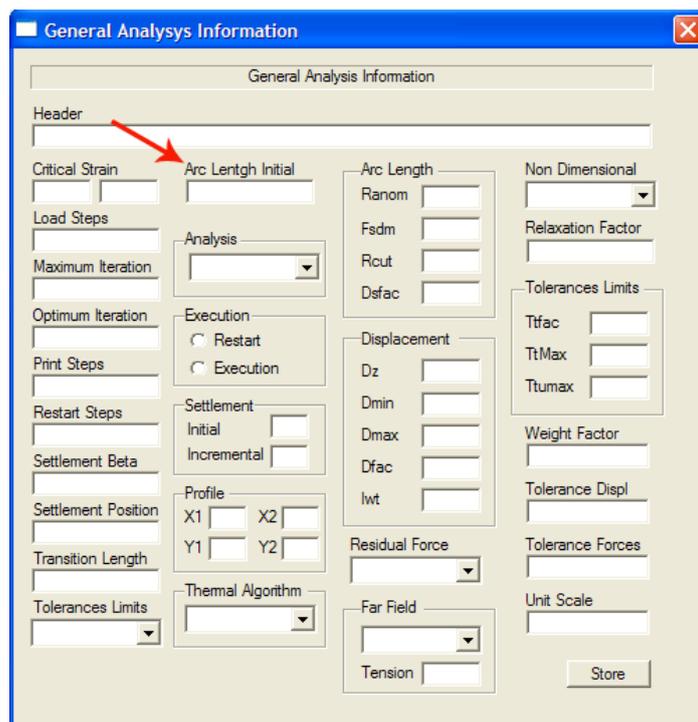


Figura 2.23 – *Static* indicado pela seta vermelha.

Verifica-se que os parâmetros que diferem na Função *CreateWindow* do exemplo anterior para a criação de uma janela, são os valores “*Static*”, “*Arc Lentgh Inicial*” e 122, 80, 80, 18, que são respectivamente de que se trata o elemento, a string que este elemento irá receber e os valores da posição x, posição y, largura e altura.

b) *Edit*

O *Edit* é em alguns casos o mais simples dos elementos da biblioteca *Win32*, em outros pode se tornar o mais complexo. Ao criar este elemento utilizando a função *CreateWindow* deve se adicionar no parâmetro de classe a string "*Edit*", conseqüentemente irá ser criado um retângulo baseados nas posições em x e y e em sua largura e altura, parâmetros estes pertencentes a função *CreateWindow*, onde este retângulo permite ao usuário escrever e editar textos, ao receber o foco permite o usuário digitar o texto e habilita o cursor a se mover para qualquer ponto do texto assim como a utilização da tecla *Shift* para selecionar parte do texto e as teclas *Ctrl+C* para copiar e *Ctrl+X* para recortar. A seguir é mostrado o código necessário para criar o *edit* e a Figura 25 ilustra a criação de um *edit*.

```
E_Arc_Length_Initial= CreateWindowEx (  
    WS_EX_CLIENTEDGE,  
    TEXT("EDIT"),  
    "",  
    WS_VISIBLE/WS_CHILD/WS_BORDER/ES_LEFT/ES_AUTOHSCROLL,  
    122, 93, 94, 20,  
    hwnd,  
    NULL,  
    g_inst,  
    NULL  
);  
  
SendMessage((HWND) E_Arc_Length_Initial,(UINT) WM_SETFONT,(WPARAM)  
GetStockObject(DEFAULT_GUI_FONT),(LPARAM) lParam);
```

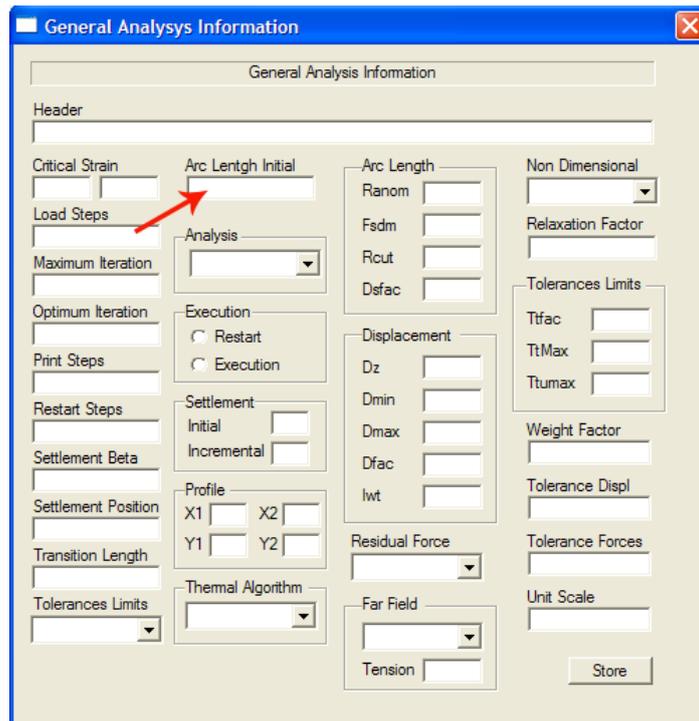


Figura 2.24 – *Edit* indicado pela seta vermelha.

c) *Push Buttons*

Um *Push Button* é um retângulo com um texto especificado pelo usuário, ele é criado adicionando a string “*Button*” no parâmetro de classe da função *CreateWindow*, os parâmetros de localização, altura e largura funcionam e forma semelhante ao do elemento *edit*.

Quando o cursor do *mouse* estiver localizado em cima do *push button* e o botão do *mouse* for pressionado, provocará um operação de redefinição no *push button* através do estilo 3D deste elemento, este evento por sua vez enviará uma mensagem ao *WM_COMMAND* através da notificação *BN_CLICKED*. A seguir é mostrado o código necessário para criar o *push button* e a Figura 26 ilustra a criação de um *push button*.

```
Store = CreateWindowEx (
    0, "BUTTON", "Store",
    WS_VISIBLE|WS_CHILD,
    395, 440, 60, 20,
    hwnd, (HMENU)ID_Store, g_inst, NULL);
```

```
SendMessage( (HWND) Store, (UINT) WM_SETFONT, (LPARAM)
GetStockObject(DEFAULT_GUI_FONT), (LPARAM) lParam);
```

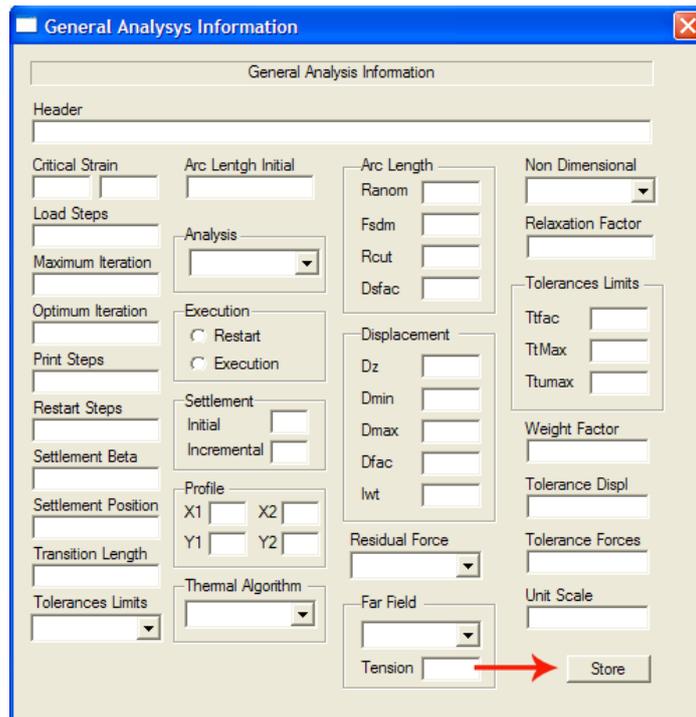


Figura 2.25 – *Push Button* indicado pela seta vermelha.

d) *Radio Buttons*

O *Radio Button* recebe este nome devido aos rádios de carros usados para sintonizar a estação, onde estes só permitirem que um entre todos eles esteja ativado. Os *radio buttons* são comumente usados dentro de caixas de diálogos para indicar opções mutuamente exclusivas. Este elemento é criado exatamente como um botão, porém o botão recebe o estilo *BS_RADIOBUTTON* ou *BS_AUTORADIOBUTTON*, para que o *Windows* altere automaticamente o estado do botão, a mensagem recebida pelo *WM_COMMAND* é realizada através da mensagem *BM_SETCHECK*. O *Radio Button* já é um estilo do elemento *Button*. A seguir é mostrado o código necessário para criar o *radio button* e a Figura 27 ilustra a criação de um *radio button*.

```
R_Execution_Restart = CreateWindowEx (
    0,
    "BUTTON",
    "Restart",
    WS_VISIBLE/WS_CHILD/BS_AUTORADIOBUTTON,
    125, 205, 70, 10,
    hwnd,
    NULL,
    g_inst,
```

```

NULL
);

SendMessage((HWND) R_Execution_Restart,(UINT) WM_SETFONT,(WPARAM)
GetStockObject(DEFAULT_GUI_FONT),(LPARAM) lParam);

//-----*/

R_Execution_Execution = CreateWindowEx (
0,
"BUTTON",
"Execution",
WS_VISIBLE/WS_CHILD/BS_AUTORADIOBUTTON ,
125, 225, 70, 10,
hwnd,
NULL,
g_inst,
NULL
);

SendMessage((HWND) R_Execution_Execution,(UINT) WM_SETFONT,(WPARAM)
GetStockObject(DEFAULT_GUI_FONT),(LPARAM) lParam);

```

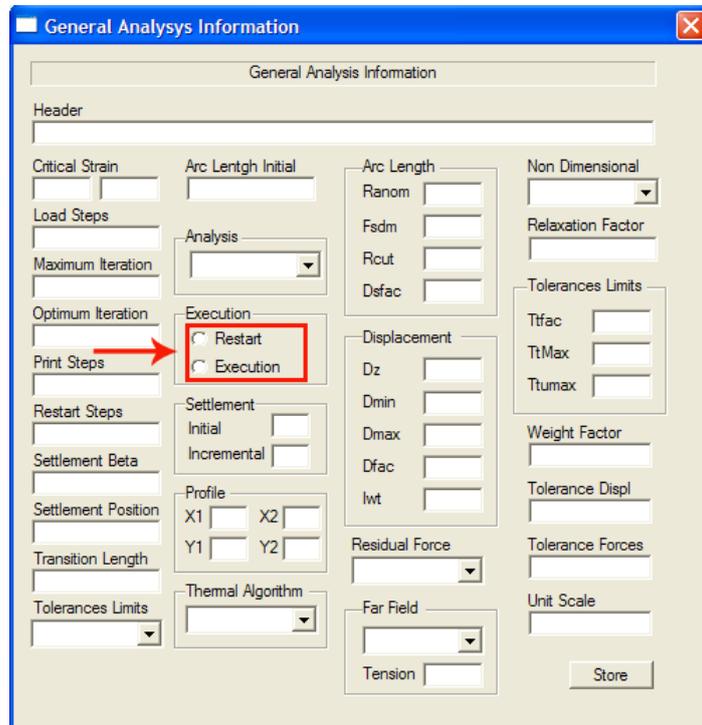


Figura 2.26 – Radio Button indicado pela seta vermelha.

e) *Group Box*

O *Group Box* é também um estilo do elemento *Button*, adicionando o estilo *BS_GROUPBOX*. Esse elemento compreende um retângulo com linhas em baixo relevo com um texto no topo do retângulo é muito utilizado para agrupar outros elementos como botões de controle. O *Group Box* já é um estilo do elemento *Button*. A seguir é mostrado o código necessário para criar o *group box* e a Figura 28 ilustra a criação de um *group box*.

```
G_Profile = CreateWindowEx (  
    0,  
    "BUTTON",  
    "",  
    WS_VISIBLE|WS_CHILD|BS_GROUPBOX,  
    114, 312, 110, 66,  
    hwnd,  
    NULL,  
    g_inst,  
    NULL  
);
```

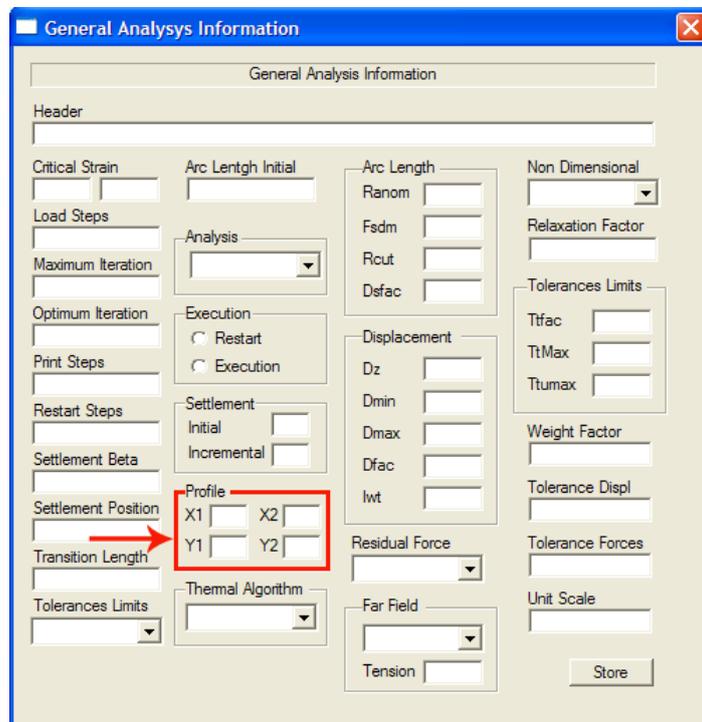


Figura 2.27 – *Group Box* indicado pela seta vermelha.

f) *Combo Box*

Um *Combo Box* é criado adicionando a string “*Combobox*” no parâmetro de classe da função *CreateWindow*, os parâmetros de localização, altura e largura funcionam e forma semelhante ao do elemento *Edit*. Este elemento é botão para valores pré selecionados relacionados em uma lista. A seguir é mostrado o código necessário para criar o *combo box* e a Figura 29 ilustra a criação de um *combo box*.

```
C_Residual_Force = CreateWindow("COMBOBOX",  
    NULL,  
    WS_CHILD|WS_TABSTOP|  
    WS_VISIBLE|CBS_DROPDOWNLIST|CBS_SORT|WS_VSCROLL,  
    240, 365, 95, 100,  
    hwnd, NULL,  
    g_inst, NULL);
```

```
SendMessage((HWND) C_Residual_Force,(UINT) WM_SETFONT, (WPARAM)  
GetStockObject(DEFAULT_GUI_FONT),(LPARAM) lParam);
```

```
SendMessage(C_Residual_Force,CB_ADDSTRING,0,(LPARAM)"X");  
SendMessage(C_Residual_Force,CB_ADDSTRING,0,(LPARAM)"Y");  
SendMessage(C_Residual_Force,CB_ADDSTRING,0,(LPARAM)"XY");  
SendMessage(C_Residual_Force,CB_ADDSTRING,0,(LPARAM)"YZ");  
SendMessage(C_Residual_Force,CB_ADDSTRING,0,(LPARAM)"XYZ");
```

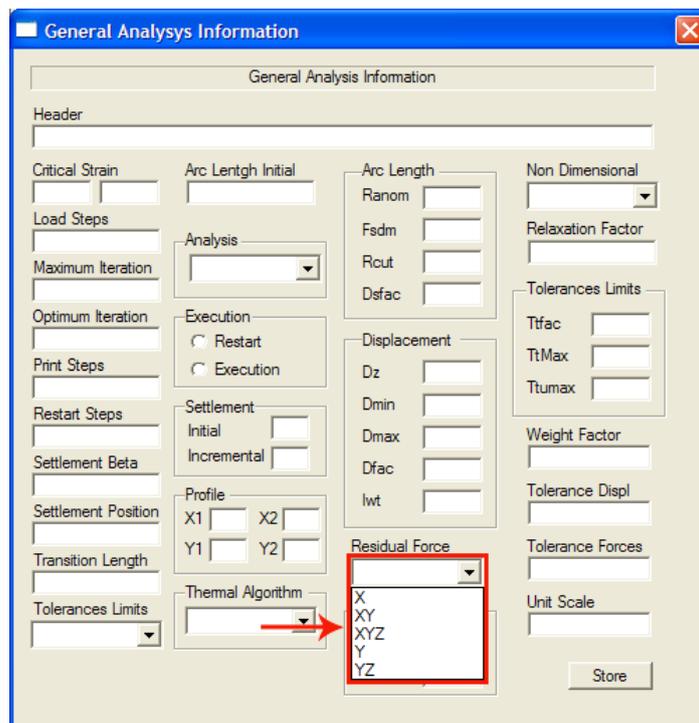


Figura 2.28 – *Combo Box* indicado pela seta vermelha.

Para adicionar uma string ao elemento *Combo Box* basta enviar uma mensagem para o elemento utilizando o parâmetro `CB_ADDSTRING`.

2.5 – O COMPILADOR *DEV-C++*

O *Dev-C++* é um *free-software*, baseado em uma licença de domínio público (*GNU General Public License*). Permitindo que qualquer usuário possa utilizar o software sem a necessidade de pagar por isso. Devido essa facilidade, o compilador se popularizou rapidamente, e também pelo fato de ser multiplataforma, ou seja, pode ser instalado em diversos sistemas operacionais, como por exemplo: *Windows*, *Linux* e *UNIX*.

O *Dev-C++* foi desenvolvido por Colin Laplace, Mike Berg e Hongli Lai, e consiste em um compilador tanto do “C” quanto do “C++”, que oferece um ambiente integrado, contendo um editor de textos e um compilador para linguagem “C”. Portanto, o usuário não precisa de um processador de textos para digitar o **código fonte**. O que facilita todo o processo desde a construção do programa até a sua execução.

O sistema operacional utilizado no decorrer do trabalho será a plataforma do Microsoft *Windows*, sendo importante destacar os dois tipos de aplicações utilizadas:

- 1) Para o console (*Console Applications*), que são executadas numa janela de texto (também conhecida como Prompt do *MS-DOS*).
- 2) Aplicações Gráficas (*GUI Applications*), que usam janelas, menus e outros elementos visuais como parte de sua interface com o programador.

Resumindo o *Dev-C++* foi o software escolhido devido ser um compilador do “C++”, já englobar as funções de editor de textos, *linkeditor* e compilador, suportar as bibliotecas, *OpenGL* e *Win32*, necessárias para o desenvolvimento do trabalho, e por ser um *software* de licença pública.

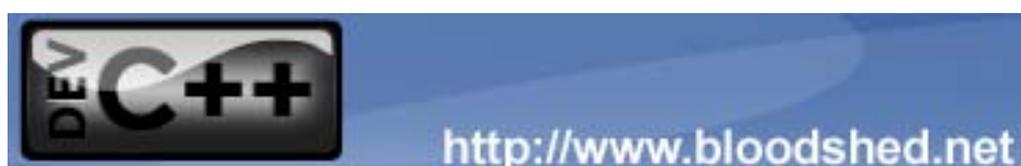


Figura 2.29 – Logotipo do Dev-C++.

2.6 – ABP (ANALYSIS OF BURIED PIPELINES)

Nas últimas décadas a utilização de tubulações enterradas tem se tornado cada vez mais utilizadas como meio de transporte de óleo e gás. Eles estão substituindo outros meios de transportes utilizados pela indústria principalmente por serem a alternativa mais econômica. Como a maioria das ocorrências de novas fontes de gás e óleo são exploradas em regiões mais remotas como no ártico, por exemplo, as tubulações estão se estendendo e sendo submetidas a condições naturais mais adversas. Todos esses motivos implicam no melhor entendimento do comportamento das tubulações e na busca de ferramentas mais poderosas de análise, simulando todas as possíveis condições adversas.

O ABP (*Analysis of Buried Pipelines*) é um programa de análise de tubulações enterradas, desenvolvido pelo grupo denominado AltaPipe. Este grupo é formado por um pequeno número de estudantes e de pós-doutorandos do Departamento de Engenharia Civil da University of Alberta, Canadá. Este grupo tem produzido freqüentes dissertações de mestrado e teses de doutorado, entre outros trabalhos acadêmicos, e várias consultorias para empresas de petróleo canadenses e internacionais (entre elas a Petrobrás).

Os trabalhos acadêmicos e de consultoria na UofA estão em grande parte associados ao desenvolvimento e aplicação da capacidade de análise do ABP. Os primórdios deste software datam de 1988 (Zhou, 1988). O Prof. David W. Murray, Professor Emérito da UofA, tem sido o principal líder no desenvolvimento do ABP.

A pesquisa sobre o comportamento de tubulações começou no Departamento de Engenharia Civil e Ambiental da Universidade de Alberta, sob supervisão do D. W. Murray e o patrocínio da *Interprovincial Pipeline Inc.* quando Zhilong Zhou (Zhou, 1988) trouxe um desenvolvimento teórico, que fazia parte do código numérico para a solução do problema de flambagem em tubulações. Magdhi Mohareb em 1991 (Mohareb, 1991) começou com a primeira série de ensaio em escala real, completando o seu trabalho em 1993, Nader Yoosef Ghodsi contribuiu com uma série de testes numéricos computacionais similares em 1993 (Ghodsi 1993). Tarcisio Souza em 1995 implementou maior capacidade ao ABP simulando cargas críticas e configurações deformadas, entre outras contribuições, algumas dessas aplicações estão ilustradas nas Figuras 2.31, 2.32 e 2.33.

O ABP foi elaborado basicamente para determinar o comportamento estrutural de tubulações enterradas, mas isso não o exclui também de fazer análises de tubulações submarinas (geralmente não enterradas e suportadas de forma intermitente em bancos de areia do terreno irregular no fundo do mar). O programa desenvolvido em etapas múltiplas por diversos pesquisadores sempre se utilizou da linguagem *FORTRAN* e roda em ambiente *MS-DOS*.

Dois tipos de carregamentos naturais para tubulações são as deformações impostas e a variação de temperatura. As deformações impostas podem ser causadas por fatores como movimentos de inclinação, empuxos devido ao congelamento e recalques por causa do degelo. O efeito mais importante das deformações impostas é a combinação da curvatura devido à deformação e a própria curvatura da tubulação. Tubulações de aço sujeitas a este tipo de deformação podem deformar dentro do regime elasto-plástico e formar “rugas”.

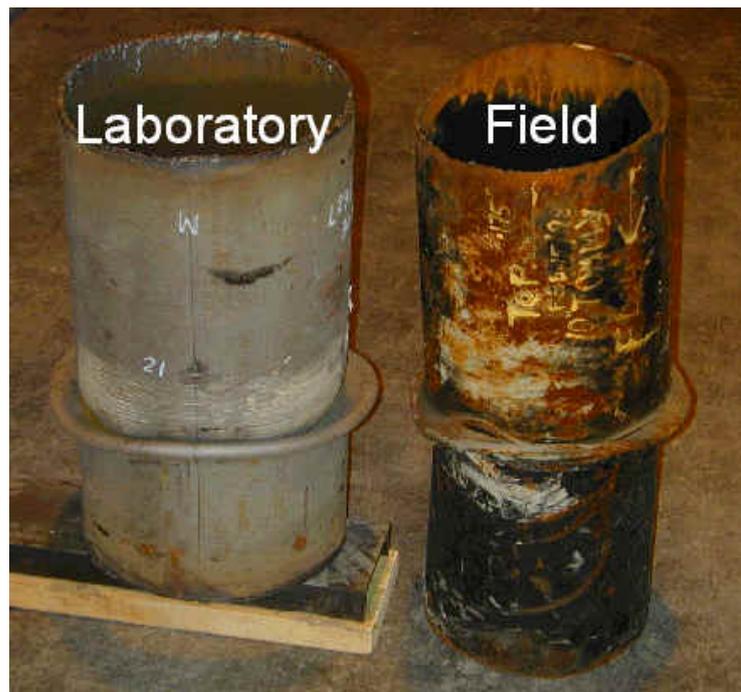


Figura 2.30 – Deformação no regime elasto-plástico
(<http://www.altapipe.ualberta.ca>, Acesso 23 jan 2005)

A diferença entre a temperatura do fluido em que a tubulação está emersa e a temperatura do fluido que se encontra dentro da tubulação provocam forças axiais de compressão na tubulação. Se a diferença de temperatura é ampla o suficiente, esta pode causar uma resultante axial de compressão a qual ocasionará uma deformação considerável, seja ela na

vertical, lateral ou as duas combinadas. Isto pode envolver deformações elasto-plásticas e a formação de “rugos” na tubulação. Muitas causas de deformações térmicas têm sido registradas tanto para tubulações *onshore* e *offshore*.

O ABP é um projeto de pesquisa que tenta desenvolver um modelo numérico capaz de analisar tubulações em diferentes condições de carregamento e em particular o carregamento térmico. O elemento finito tem características elasto-plásticas, elementos de barra isoparamétricos capazes de capturar grandes deslocamentos e deformações finitas usando uma atualização da formulação Lagrangiana.



Figura 2.31 – Tubulação Instrumentada

(<http://www.altapipe.ualberta.ca>, Acesso 23 jan 2005)

O programa tem a capacidade de fazer análises lineares e não lineares (geométrica e material) de tubulações de seções transversais variadas.

Os materiais, geralmente o aço, podem ser representados pela curva tensão-deformação de *Ramberg-Osgood*. As instabilidades estudadas pelo ABP incluem, entre outras, o *snapping* de tubulações gerando o *upheaval* ou o *snaking* (serpenteamento) da tubulação.

O *upheaval* ou levantamento da tubulação é bastante comum de acontecer em tubulações enterradas, mas que estejam na vizinhança de superfícies livres (ex. em parques abertos) ver Figura 2.31.

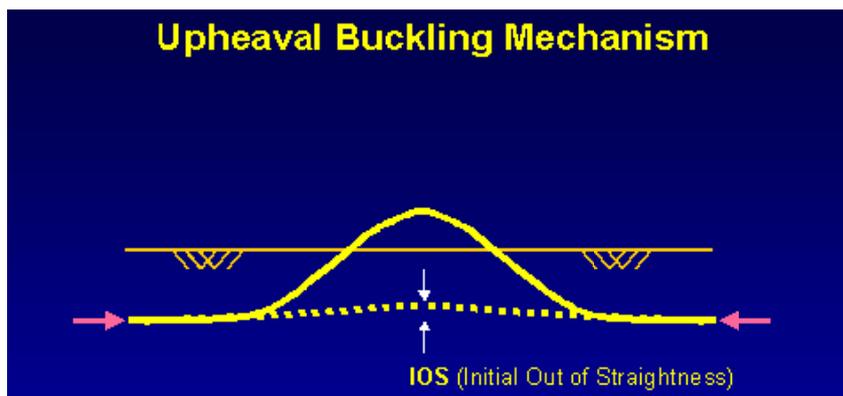


Figura 2.32 – Mecanismo de Flambagem: representação do upheaval
(<http://www.altapipe.ualberta.ca>, Acesso 23 jan 2005)

Já o *snaking* ou serpenteamento da tubulação é um fenômeno muito comum em tubulações submarinas para o transporte de petróleo – nesta instabilidade a tubulação assume a forma de uma serpente.

Para a determinação da carga crítica da tubulação enterrada é de fundamental importância a presença ou não da pressão interna. A presença da pressão interna gera modos de colapso que podem ser da forma de *Bulge wrinkle* ou *diamond* – ver Figuras 2.32 e 2.33. Estas instabilidades podem ser capturadas pelo programa ABP.



Figura 2.33 – Não pressurizada: modo diamond whinkle
(<http://www.altapipe.ualberta.ca>, Acesso 23 jan 2005)



Figura 2.34 – Pressurizada: modo bulge-wrinkle
(<http://www.altapipe.ualberta.ca>, Acesso 23 jan 2005)

O ABP é um programa com elementos finitos que suporta grandes rotações e deformações, apesar de usar elementos unidimensionais, bem como incorpora ferramentas que “discretizam” também a seção transversal da tubulação, capturando as tensões máximas nas fibras mais distorcidas e admitindo a “ovalização” da seção transversal.

O ABP estuda o acomodamento das tubulações, *settlement analysis*, bem como outras iterações entre a tubulação e o solo. O solo no ABP pode ser concebido de forma linear ou não linear – com o uso de mecanismos de apoio em molas – e este comportamento do solo leva a cargas de colapso específicas em tubulações estudadas pelo ABP.

Em fim, o ABP apesar de excelente software para tubulações ainda é executado em ambiente *MS-DOS*. Este trabalho tenta atualizar (para o usuário) interfaces que facilitaram o pré e o pós processamento do ABP – como já referenciado antes nesta monografia.

3 – METODOLOGIA

3.1 – VISÃO GERAL DO ABP SEM A INTERFACE GRÁFICA

A metodologia aplicada envolve a programação através de Aplicações Gráficas (*GUI Applications*), que usam janelas, menus e outros elementos visuais como parte da interface com o programador. O ABP está escrito em Fortran 77, e seu executável necessita de um arquivo no modo de texto com a extensão **.INP** como arquivo de entrada. O qual é inserido através de uma tela de *prompt*, após o processamento dos dados, o ABP gera um arquivo no formato **.OUT**, com as informações da análise. A Figura 3.1 mostra através de uma visão global como funciona o ABP.

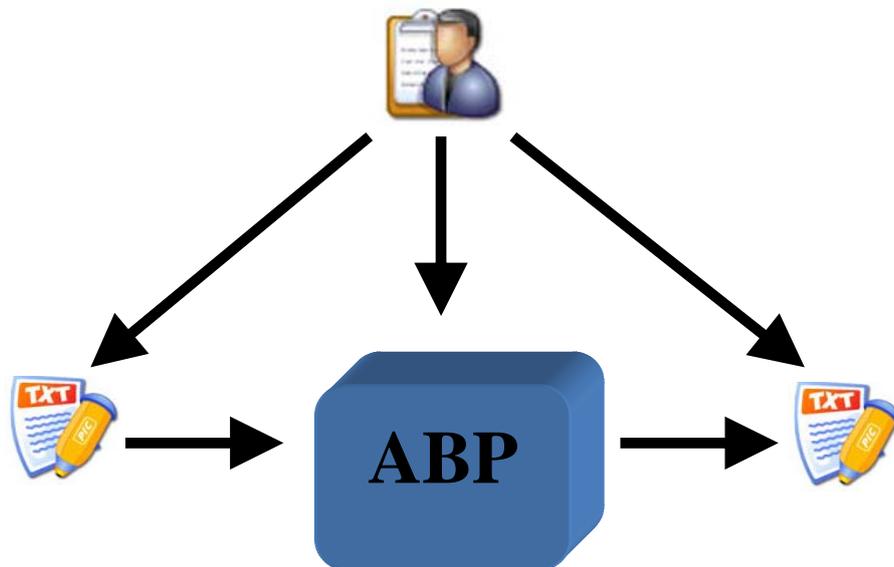


Figura 3.1 – Visão geral do ABP.

3.2 – VISÃO GERAL DO ABP COM A INTERFACE GRÁFICA

Com o advento da interface gráfica esta ficará encarregada de:

- 1) Avaliar a coerência das tomadas de decisões realizadas pelo usuário;
- 2) Interpretar as tomadas de decisões e gerar o arquivo de entrada de dados;
- 3) Processo de conectar (“linkar”), a interface gráfica, os arquivos de entrada e saída de dados;
- 4) Avaliar a coerência das informações contidas no arquivo de saída de dados;
- 5) Confeccionar os relatórios contendo os resultados da análise e modelagem gráfica dos dados.

A Figura 3.2 mostra de forma global como funciona o ABP com a nova interface.

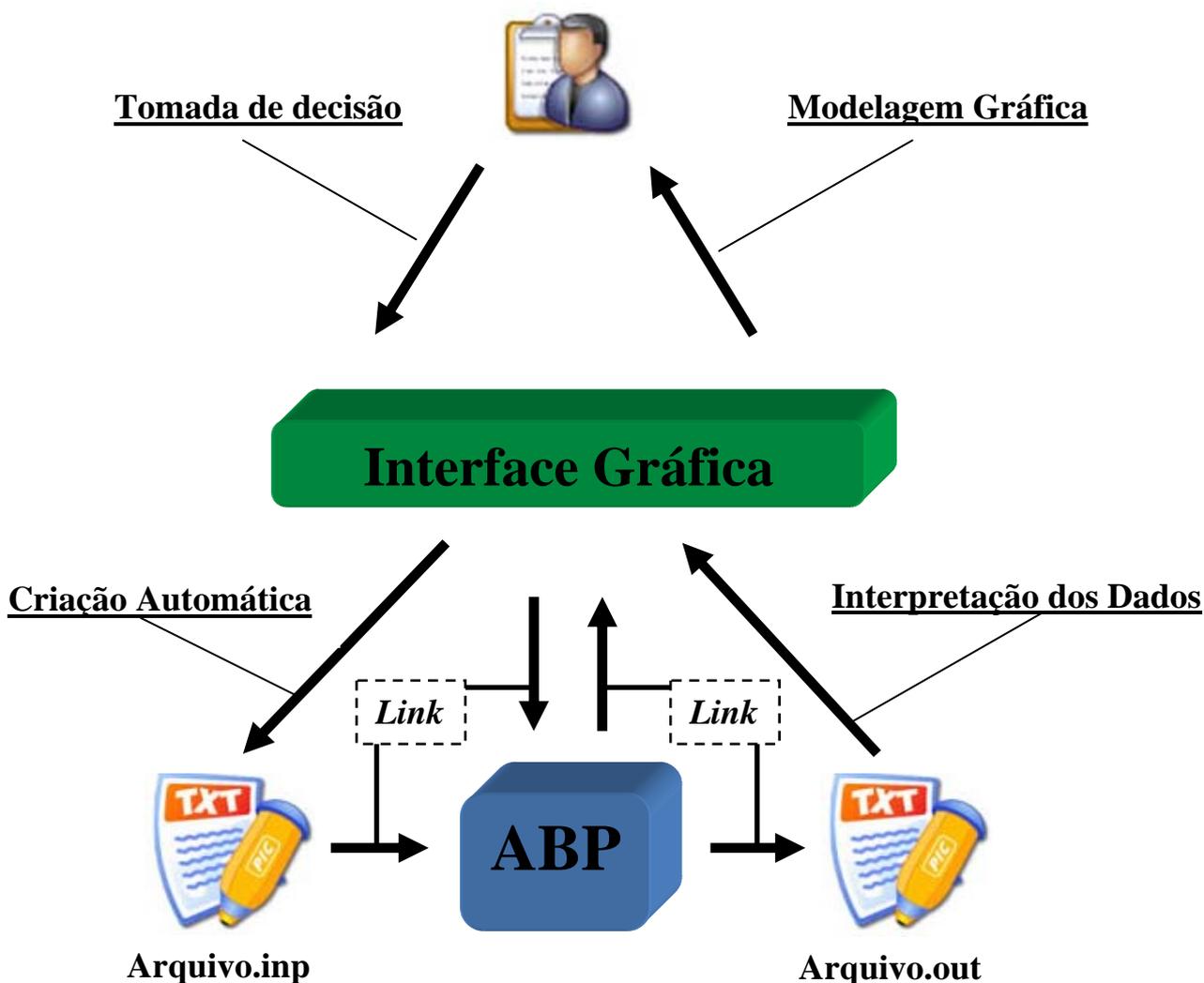


Figura 3.2 – Visão geral do ABP, após o advento da interface gráfica.

3.3 – PRINCIPAIS ELEMENTOS DA INTERFACE GRÁFICA

A figura abaixo demonstra todos os elementos da interface de forma hierárquica, no decorrer deste capítulo serão mostradas as telas, o fluxo de informações e como estas interagem.

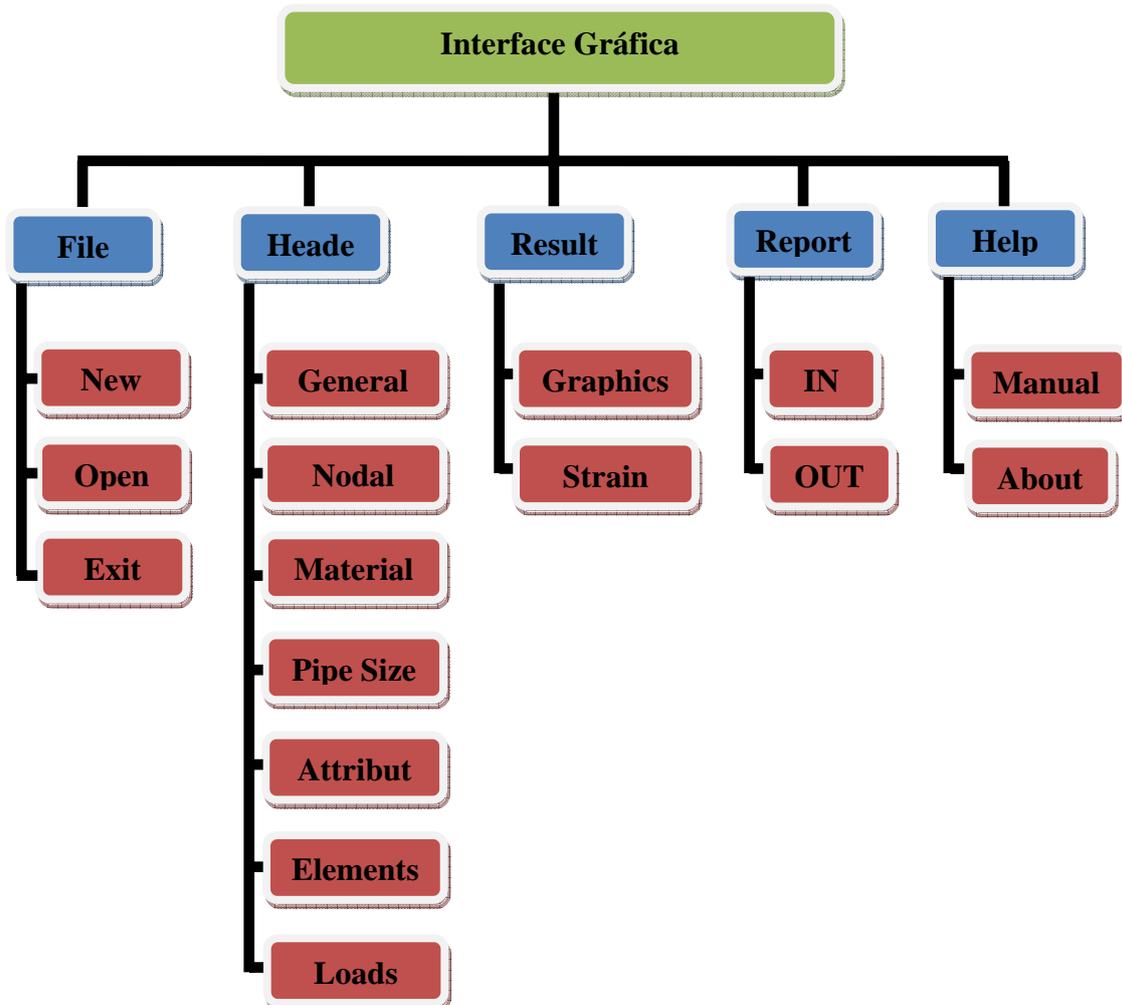


Figura 3.3 – Menu da interface gráfica do ABP

3.3.1 – Processo de Criação dos Elementos Gráficos

Neste tópico será descrito como foram criados os elementos gráficos presentes na interface do *ABP*. Partindo das primitivas gráficas pertinentes na biblioteca da *OpenGL* como a da biblioteca *GLUT*. Será descrito de maneira breve a “primeira versão” da interface gráfica com elementos 2D e posteriormente e em detalhes os elementos gráficos em 3D na sua “versão final”.

Em uma primeira etapa todos os elementos foram criados em 2D, na tela de geração da malha os apoios foram simulados como triângulos vermelhos com a notação usualmente adotada na teoria da resistência dos materiais, a tubulação foi descrita como uma barra subdividida em elementos menores usando quadrados azuis para definir os elementos, como pode ser visto na Figura 3.4.

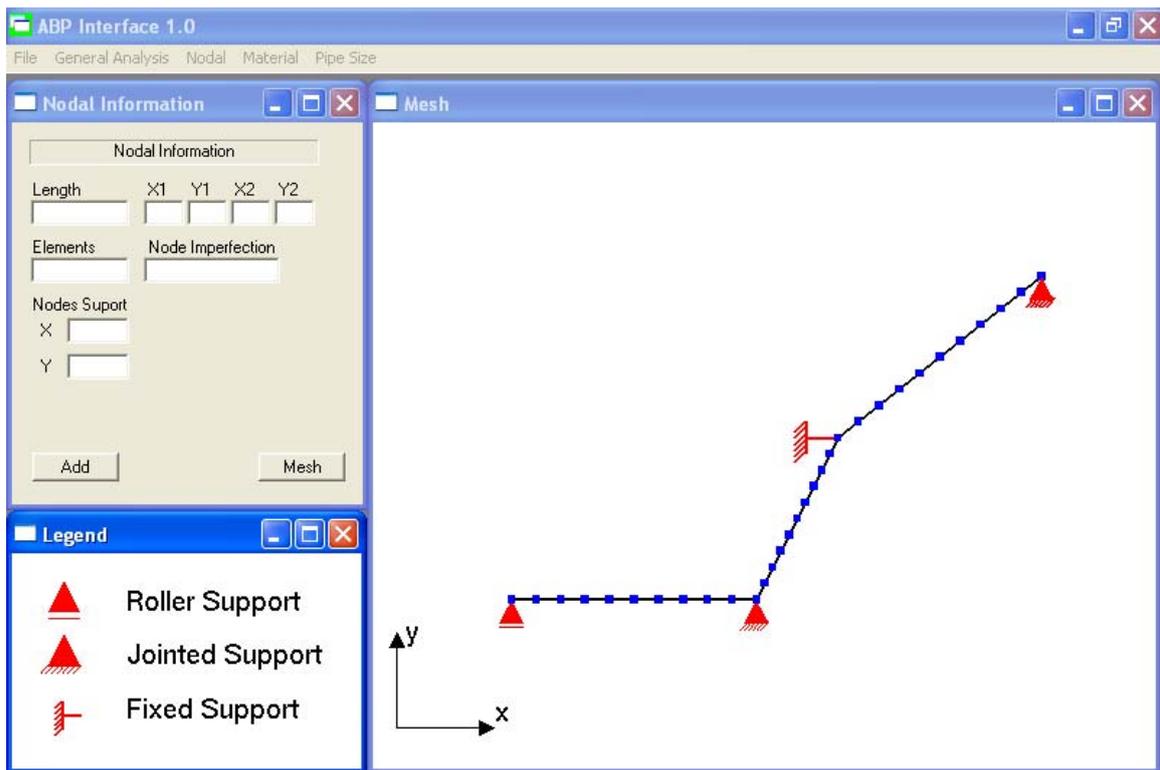


Figura 3.4 – Geração da Malha em 2D

Na tela de definição do diâmetro interno e espessura da parede da tubulação foram utilizados círculos, um interno e outro externo com contorno azul e um preenchimento laranja entre eles que denota a parede da tubulação, como pode ser visto na Figura 3.5.

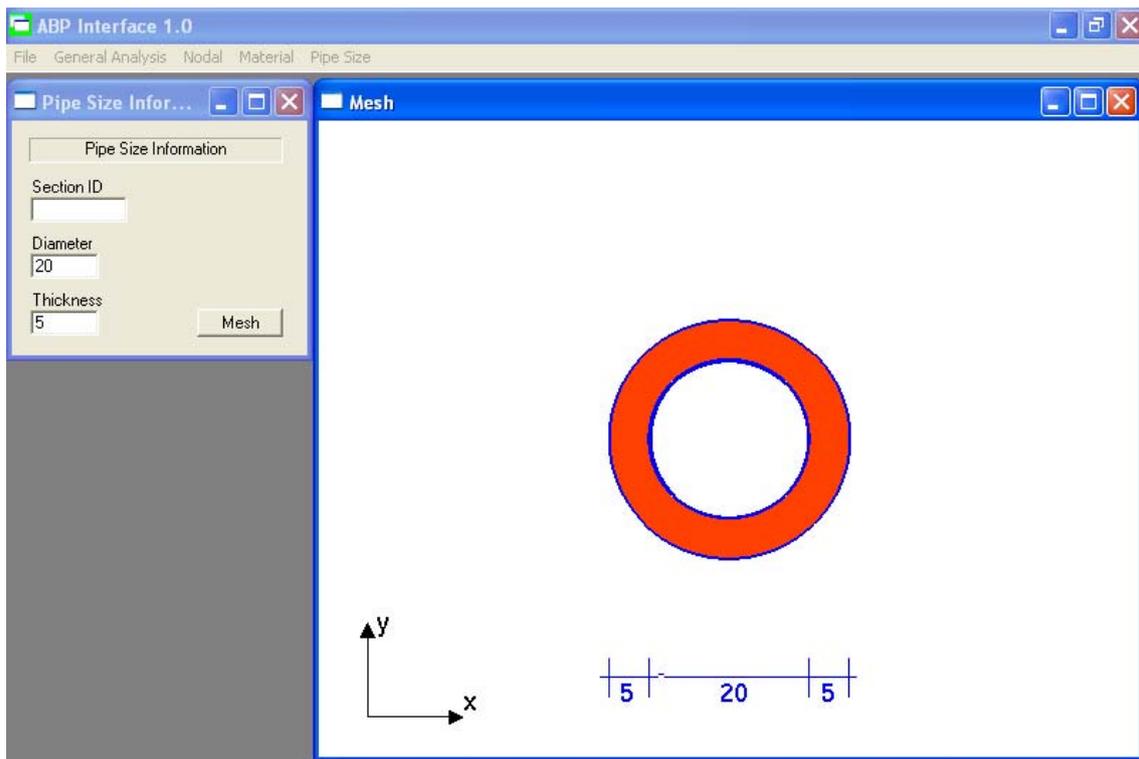


Figura 3.5 – Diâmetro e Espessura da Tubulação

Na versão 2D apenas a geração da malha e as informações sobre a seção transversal foram desenvolvidas utilizando elementos gráficos. Nos itens a seguir serão detalhados os elementos gráficos em 3D.

3.3.1.1 – Apoios

O elemento escolhido para a simulação da restrição de movimento, no eixo onde será aplicado, foi uma pirâmide, mas não existe na biblioteca nem da *OpenGL* nem da *GLUT* uma primitiva gráfica que crie diretamente uma pirâmide, porém a primitiva *glutSolidCone* tem como um de seus parâmetros o número de subdivisões em torno do eixo z, ao se reduzir este numero para quatro se chega no objeto escolhido.

```
// Desenha uma piramide em 3D translacionada de acordo com o usuário
glPushMatrix();
glTranslatef(elements[NE][1],elements[NE][2],elements[NE][3]-5);
glColor3f(0.0f, 0.0f, 2.0f);
glutSolidCone(2.0, 5.0, 4, 4);
glPopMatrix();
```

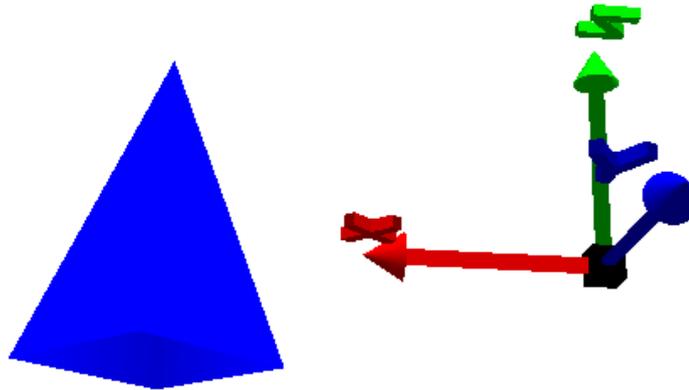


Figura 3.6 – Apoio com Restrição de Movimento

O comando `glTranslatef(elements[NE][1],elements[NE][2],elements[NE][3]-5);` define que o elemento será deslocado nos valores inseridos no vetor `elements` que recebe os valores inseridos pelo usuário e os comandos `glPushMatrix();` e `glPopMatrix();` restringem a translação apenas para os objetos criados entre esses dois comandos, não deixando que o valor da translação seja acumulado para outros objetos no decorrer do código. A maior dificuldade encontrada na criação desse objeto foi justamente o posicionamento dele no espaço e não deixar que a translação fosse acumulativa.

Para a simulação dos apoios flexíveis, molas, a escolha do elemento não foi o principal problema, mas como seria feita a criação da mola e que tipos de primitivas gráficas seriam utilizadas, já que não existe nenhuma primitiva gráfica nas bibliotecas tanto da *OpenGL* quanto da *GLUT*, após um longo processo chegou-se a utilização do comando `GL_LINE_STRIP`, que gera um número $N-1$ segmentos de linha conectando N vértices, dentro de um laço `for` utilizado para desenhar um círculo através de senos e cossenos, onde a medida que o laço completava um ciclo a variável do eixo Z , com o nome de `esp` pois denota a ideia de espessura da mola, recebia um acréscimo. Os trechos de código a seguir demonstram como foi feita os principais pontos da programação e a Figura 3.7 mostra em detalhes o resultado.

- ✓ Desenha uma mola em 3D translacionada de acordo com o usuário

```
glPushMatrix();
```

```
glRotatef(-90.0f,0.0f,0.0f,1.0f);
```

```
glTranslatef(elements[NE][1],elements[NE][2],elements[NE][3]-6.5);
```

```

    esp=0;
    glBegin(GL_LINE_STRIP);

for(ang=0;ang<2*M_PI;ang+=M_PI/100.0)
{
    glVertex3f(0.8*sin(ang),0.8*cos(ang),esp);
    glVertex3f(0.8*sin(ang),0.8*cos(ang),esp+0.3);
    esp=esp+0.007;
}
glEnd();

    glPopMatrix();

```

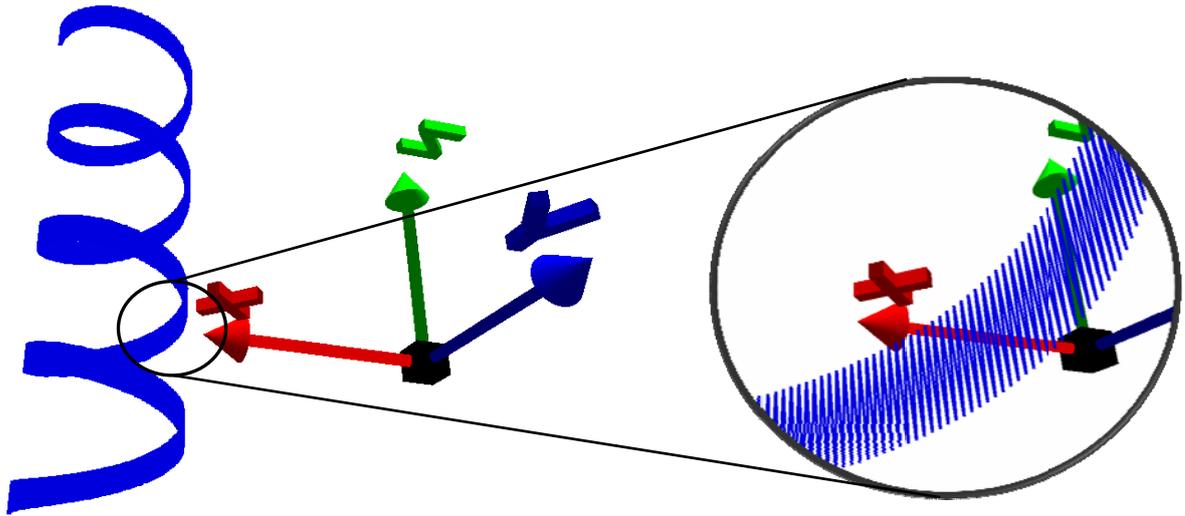


Figura 3.7 – Apoio Flexível e o Zoom do Detalhe

3.3.1.2 – Tubulação na Geração de Malhas

Na modelagem 2D a tubulação era representada apenas por uma linha, e o comprimento era facilmente calculado através da hipotenusa do triângulo formado pelas coordenadas dos pontos, na primeira tentativa da modelagem em 3D foram utilizadas linhas próximas uma das outras e paralelas a uma linha principal que conectava os pontos fornecidos pelo usuário formando a tubulação, porém foi constatado que tubo deformava quando uma coordenada era muito menor que as outras duas, para evitar esse tipo de problema o tubo passou a ser construído no eixo Z com o comprimento calculado através da hipotenusa dos triângulo e posteriormente eram aplicados duas rotações pra chegar na posição desejada, para a subdivisão da tubulação formando assim os elementos o procedimento foi praticamente o mesmo, a única diferença foi que em cada subdivisão eram criados círculos vermelhos indicando o tamanho dos elementos. Os trechos de código a seguir demonstram como foi feita os principais pontos da programação e a Figura 3.8 mostra em detalhes o resultado.

- ✓ Calcula as distancias entre as coordenadas X, Y e Z

```
glPushMatrix();
```

```
glTranslatef(elements[NE][1],elements[NE][2],elements[NE][3]);
```

```
aux1=elements[NE][4]-elements[NE][1];
```

```
aux2=elements[NE][5]-elements[NE][2];
```

```

aux3=elements[NE][6]-elements[NE][3];
aux4=(aux1/aux2);

```

✓ Calcula o comprimento do trecho.

```

comp=sqrt(pow(aux1,2)+pow(aux2,2)+pow(aux3,2));

```

```

if (aux1>0 && aux2<0)
{
    teta = (180*atan(-(aux1/aux2))/(3.141592654));
    hipot= sqrt(pow((elements[NE][4]-elements[NE][1]),2)+pow((elements[NE][5]-
elements[NE][2]),2));
    beta= -(180*atan(aux3/hipot)/(3.141592654));

    // Desenha um círculo de linhas
    glRotatef(90.00f, 1.0f, 0.0f, 0.0f);
    glRotatef(teta, 0.0f, 1.0f, 0.0f);
    glRotatef(beta, 1.0f, 0.0f, 0.0f);
}

```

```

if (aux1==0 && aux2<0)
{
    teta = (180*atan(-(aux1/aux2))/(3.141592654));
    hipot= sqrt(pow((elements[NE][4]-elements[NE][1]),2)+pow((elements[NE][5]-
elements[NE][2]),2));
    beta= -(180*atan(aux3/hipot)/(3.141592654));

    // Desenha um círculo de linhas
    glRotatef(90.00f, 1.0f, 0.0f, 0.0f);
    glRotatef(teta, 0.0f, 1.0f, 0.0f);
    glRotatef(beta, 1.0f, 0.0f, 0.0f);
}

```

```

if (aux1==0 && aux2>0)
{
    teta = (180*atan((aux1/aux2))/(3.141592654));
    hipot= sqrt(pow((elements[NE][4]-elements[NE][1]),2)+pow((elements[NE][5]-
elements[NE][2]),2));
    beta= (180*atan(aux3/hipot)/(3.141592654));

    // Desenha um círculo de linhas
    glRotatef(90.00f, -1.0f, 0.0f, 0.0f);
    glRotatef(teta, 0.0f, 1.0f, 0.0f);
    glRotatef(beta, 1.0f, 0.0f, 0.0f);
}

```

```

    if (aux1>0 && aux2>0)
    {
        teta = (180*atan((aux1/aux2))/(3.141592654));
        hipot=      sqrt(pow((elements[NE][4]-elements[NE][1]),2)+pow((elements[NE][5]-
elements[NE][2]),2));
        beta= (180*atan(aux3/hipot)/(3.141592654));

        // Desenha um círculo de linhas
        glRotatef(90.00f, -1.0f, 0.0f, 0.0f);
        glRotatef(teta, 0.0f, 1.0f, 0.0f);
        glRotatef(beta, 1.0f, 0.0f, 0.0f);
    }

    if (aux1>0 && aux2==0)
    {
        teta = (180*atan((aux1/aux2))/(3.141592654));
        hipot=      sqrt(pow((elements[NE][4]-elements[NE][1]),2)+pow((elements[NE][5]-
elements[NE][2]),2));
        beta= (180*atan(aux3/hipot)/(3.141592654));

        // Desenha um círculo de linhas
        glRotatef(90.00f, -1.0f, 0.0f, 0.0f);
        glRotatef(teta, 0.0f, 1.0f, 0.0f);
        glRotatef(beta, 1.0f, 0.0f, 0.0f);
    }

    if (aux1<0 && aux2<0)
    {
        teta = (180*atan(-(aux1/aux2))/(3.141592654));
        hipot=      sqrt(pow((elements[NE][4]-elements[NE][1]),2)+pow((elements[NE][5]-
elements[NE][2]),2));
        beta= -(180*atan(aux3/hipot)/(3.141592654));

        // Desenha um círculo de linhas
        glRotatef(90.00f, 1.0f, 0.0f, 0.0f);
        glRotatef(teta, 0.0f, 1.0f, 0.0f);
        glRotatef(beta, 1.0f, 0.0f, 0.0f);
    }

    if (aux1<0 && aux2>0)
    {
        teta = (180*atan((aux1/aux2))/(3.141592654));
        hipot=      sqrt(pow((elements[NE][4]-elements[NE][1]),2)+pow((elements[1][5]-
elements[1][2]),2));
        beta= (180*atan(aux3/hipot)/(3.141592654));

        // Desenha um círculo de linhas

```

```

    glRotatef(90.00f, -1.0f, 0.0f, 0.0f);
    glRotatef(teta, 0.0f, 1.0f, 0.0f);
    glRotatef(beta, 1.0f, 0.0f, 0.0f);
}

if (aux1 < 0 && aux2 == 0)
{
    teta = (180*atan((aux1/aux2))/(3.141592654));
    hipot=      sqrt(pow((elements[NE][4]-elements[NE][1]),2)+pow((elements[NE][5]-
elements[NE][2]),2));
    beta= (180*atan(aux3/hipot)/(3.141592654));

    // Desenha um círculo de linhas
    glRotatef(90.00f, -1.0f, 0.0f, 0.0f);
    glRotatef(teta, 0.0f, 1.0f, 0.0f);
    glRotatef(beta, 1.0f, 0.0f, 0.0f);
}

```

- ✓ Desenha as linhas que vão formar a tubulação.

```

    glBegin(GL_LINES);

for(ang=0;ang<2*M_PI;ang+=M_PI/100.0)
{
    glVertex3f(1*cos(ang),1*sin(ang),0);
    glVertex3f(1*cos(ang),1*sin(ang),comp);
}

    glEnd();

```

- ✓ Desenha os círculos que vão definir os elementos.

```

glPushMatrix();

glLineWidth(5.0f);

auxcomp=comp/elements[NE][7];
number=0;

for(I=1; I<elements[NE][7]+1 ; I++)
{
    glColor3f(2.0f, 0.0f, 0.0f);

    glBegin(GL_LINES);

```

```

for(ang=0;ang<2*M_PI;ang+=M_PI/100.0)
{
    glVertex3f(1.05*sin(ang),1.05*cos(ang),number);
}

for(ang=0;ang<2*M_PI;ang+=M_PI/100.0)
{
    glVertex3f(1.05*cos(ang),1.05*sin(ang),number);
}

glEnd();

number=number+auxcomp;
}

glPopMatrix();

glPopMatrix();

```

As variáveis *aux1*, *aux2*, *aux3* e *aux4* recebem a variação das coordenadas *X*, *Y* e *Z* entre os pontos definidos pelo usuário. Já as variáveis *beta*, *teta* e *hipot* recebem respectivamente os valores dos ângulos entre os planos e a hipotenusa do triângulo, assim como a variável *comp* que também recebe o valor da hipotenusa e a variável *number* que recebe o número de círculos a serem criados para definir os elementos.

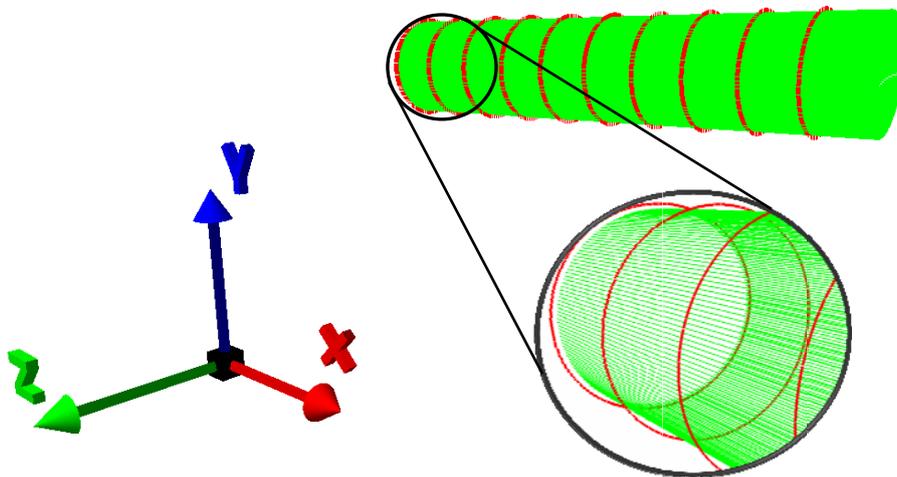


Figura 3.8 – Tubulação e o Zoom mostrando com Detalhe

3.3.1.3 – Tubulação na Seção Transversal

Uma modelagem gráfica semelhante também é usada na tela onde é definida a seção transversal, porém neste ponto são definidas duas superfícies para a tubulação com o

intuito de mostrar a espessura da tubulação, neste caso foi utilizada primitiva gráfica *GL_LINE_STRIP* assim como na modelagem da mola. Os trechos de código a seguir demonstram como foi feita os principais pontos da programação e a Figura 3.9 mostra em detalhes o resultado.

- ✓ Desenho do círculo que define o diâmetro interno

```

// Desenha um círculo de linhas
glLineWidth(5);
glBegin(GL_LINE_STRIP);
glClear(GL_COLOR_BUFFER_BIT);
glColor3f(0.0, 0.0, 5.0);

for(ang=0;ang<((2*M_PI)+M_PI);ang+=M_PI/100.0)
{
    glVertex3f((DX/2)*cos(ang),(DX/2)*sin(ang),0);
    glVertex3f((DX/2)*cos(ang),(DX/2)*sin(ang),(DX+ND+20));
}
glEnd();

glBegin(GL_LINE_STRIP);

for(ang=0;ang<((2*M_PI)+M_PI);ang+=M_PI/100.0)
{
    glVertex3f((DX/2)*cos(ang),(DX/2)*sin(ang),(DX+ND+20));
}
glEnd();

glBegin(GL_LINE_STRIP);

for(ang=0;ang<((2*M_PI)+M_PI);ang+=M_PI/100.0)
{
    glVertex3f((DX/2)*cos(ang),(DX/2)*sin(ang),0);
}
glEnd();

```

- ✓ Desenho do círculo que define a espessura

```

glLineWidth(5);
PASSO = DX+ND;

glBegin(GL_LINE_STRIP);
glColor3f(1),(0.25),(0));

```

```

for(ang=0;ang<((2*M_PI)+M_PI);ang+=M_PI/100.0)
{
    glVertex3f((PASSO/2)*cos(ang),(PASSO/2)*sin(ang),0);
    glVertex3f((PASSO/2)*cos(ang),(PASSO/2)*sin(ang),(DX+ND+20));
}

glEnd();

    glBegin(GL_LINE_STRIP);
glColor3f(1,(0.25),(0));

    for(ang=0;ang<((2*M_PI)+M_PI);ang+=M_PI/100.0)
{
    glVertex3f((PASSO/2)*sin(ang),(PASSO/2)*cos(ang),0);
    glVertex3f((PASSO/2)*sin(ang),(PASSO/2)*cos(ang),(DX+ND+20));
}

glEnd();

    glBegin(GL_LINE_STRIP);
glColor3f(1,(0.25),(0));

for(ang=0;ang<((2*M_PI)+M_PI);ang+=M_PI/100.0)
{
    glVertex3f((PASSO/2)*cos(ang),(PASSO/2)*sin(ang),0);
}

glEnd();

PASSOZ=PASSO/2;

for (I=1; I<(ND*10); I++)
{

    glBegin(GL_LINE_STRIP);
glColor3f(1,(0.25),(0));

for(ang=0;ang<((2*M_PI)+M_PI);ang+=M_PI/100.0)
{
    glVertex3f((PASSOZ)*cos(ang),(PASSOZ)*sin(ang),0);
}

glEnd();
PASSOZ=PASSOZ-(ND/(ND*20));
}

PASSOZ=PASSO/2;

for (I=1; I<(ND*10); I++)

```

```

{
    glBegin(GL_LINE_STRIP);
    glColor3f(1,(0.25),(0));

    for(ang=0;ang<((2*M_PI)+M_PI);ang+=M_PI/100.0)
    {
        glVertex3f((PASSOZ)*cos(ang),(PASSOZ)*sin(ang),(DX+ND+20));
    }

    glEnd();
    PASSOZ=PASSOZ-(ND/(ND*20));
}

    glBegin(GL_LINE_STRIP);
    glColor3f(1,(0.25),(0));

    for(ang=0;ang<((2*M_PI)+M_PI);ang+=M_PI/100.0)
    {
        glVertex3f((PASSO/2)*cos(ang),(PASSO/2)*sin(ang),(DX+ND+20));
    }

    glEnd();

```

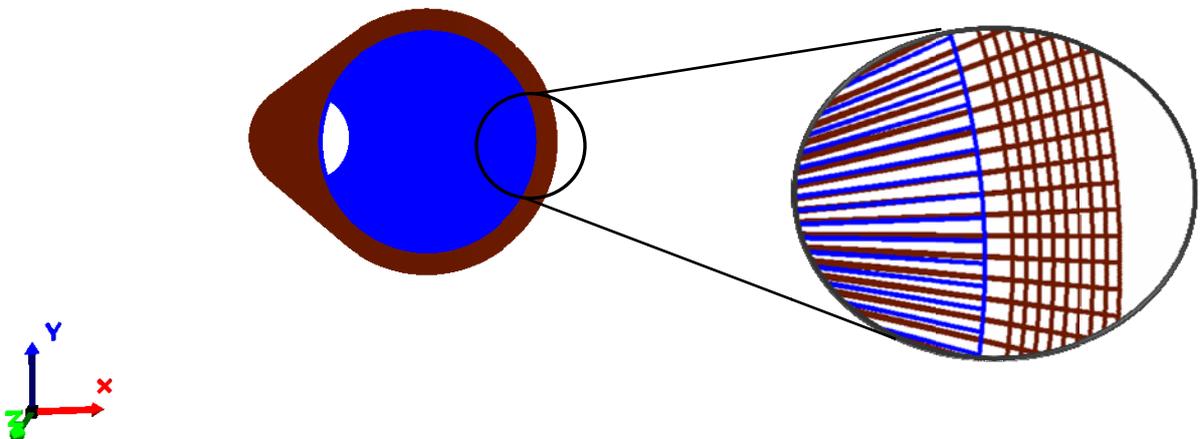


Figura 3.9 – Tubulação e o Zoom mostrando com Detalhe

3.3.1.4 – Gráficos

Mesmo este sendo o único elemento gráfico 2D com certeza foi um dos mais difíceis para serem modelados, principalmente devido ao fato da interpretação dos dados provenientes da saída do ABP, onde a interface teria que ler os arquivos de dados em uma certa posição do arquivo de saída, interpretá-los no mesmo formato em que estes estavam escrito e por fim armazená-los em um vetor. Além deste problema, outro que demandou uma boa parte

de tempo foi justamente o fator de escala do gráfico, pois os dados poderiam ter uma variação de cerca 10^{20} como no caso dos dados de provenientes da curvatura e do momento.

Na modelagem gráfica foram utilizadas as primitivas gráficas *GL_LINE_STRIP* para ler os pontos do arquivo de dados e desenhar o gráfico, além da primitiva *GL_LINES* que foi utilizada para o desenho dos eixos, da tabela e das linhas de grade do gráfico tanto na horizontal quanto na vertical e a primitiva *GL_TRIANGLES* para desenhar o triângulo no final das linhas dos eixos definindo assim a sua orientação, outras duas primitivas gráficas utilizadas foram *glutBitmapCharacter* e *glutStrokeCharacter* usadas na manipulação de textos. Os trechos de código a seguir demonstram como foi feita os principais pontos da programação e a Figura 3.10 mostra em detalhes o resultado.

- ✓ Leitura dos dados de saída, armazenamento nos vetores e processo de escrita dos dados.

```
j=1;
glLineWidth(3);

for (i=1; (fscanf(pFile, "\n%E %E", &z[i], &x[i]) != EOF); i++)
{
    fprintf(pFile2, "\n%f %f", z[i], x[i]);
    j=j+1;
}
```

- ✓ Determinação do fator de escala do gráfico.

```
maxx=x[1];
minx=x[1];

for (i=1; i<(j); i++)
{
    if (maxx<x[i]){ maxx=x[i];}
    if (minx>x[i]){ minx=x[i];}

    if (maxz<z[i]){ maxz=z[i];}
    if (minz>z[i]){ minz=z[i];}
}
```

```
}
```

```
fx=8/(maxx-minx);  
fminx=(fx*minx);  
fmaxx=(fx*maxx);  
if (fx<0) {fx=-fx;}  
fz=20/(maxz-minz);  
fminz=(fz*minz);  
fmaxz=(fz*maxz);  
if (fz<0) {fz=-fz;}
```

✓ Confecção do gráfico utilizando o *GL_LINE_STRIP*.

```
for (i=1; i<(j-1); i++)  
{  
  
glBegin(GL_LINE_STRIP);  
  
glVertex2f(fz*z[i],fx*x[i]);  
glVertex2f(fz*z[i+1],fx*x[i+1]);  
glEnd();
```

✓ Desenho dos eixos vertical e horizontal.

```
glLineWidth(2);  
  
glColor3f(0.0f, 0.0f, 0.0f);  
  
glBegin(GL_LINES);  
  
glVertex2f(fminz,fminx);  
glVertex2f(fmaxz,fminx);  
  
glEnd();  
  
glBegin(GL_LINES);  
  
glVertex2f(0,fminx);  
glVertex2f(0,fmaxx);  
  
glEnd();
```

✓ Desenho das linhas de grade vertical e horizontal.

```
for(n=1; n<(lz+1); n++)
{
    glLineWidth(2);
    glColor3f(0.5f, 0.5f, 0.5f);
    glLineWidth(1);

    glBegin(GL_LINES);

        glVertex2f(fminz,fminx+(n*(8/lz)));
        glVertex2f(fmaxz,fminx+(n*(8/lz)));

    glEnd();

}

for(n=1; n<(lx+1); n++)
{
    glLineWidth(2);
    glColor3f(0.5f, 0.5f, 0.5f);
    glLineWidth(1);

    glBegin(GL_LINES);

        glVertex2f(fminz+(n*(20/lx)),fminx);
        glVertex2f(fminz+(n*(20/lx)),fminx+8);

    glEnd();

}
```

✓ Desenho do texto.

```
glPushMatrix();

if (fmaxx>0){glTranslatef(0.0,(-(fmaxx-6.5)),0.0f);}
if (fmaxx<0){glTranslatef(0.0,((fmaxx-6.5)),0.0f);}

// Posição no universo onde o texto bitmap será colocado
glTranslatef(fz*z[1]+.6,-7.2,0);
sx[10]= ecvt(z[1],3,&decimal,&sign);

glScalef(0.004f,0.004f,0.004f);
glLineWidth(1.4f);
    DesenhaTextoStroke(GLUT_BITMAP_HELVETICA_10,sx[10]);
glPopMatrix();
```

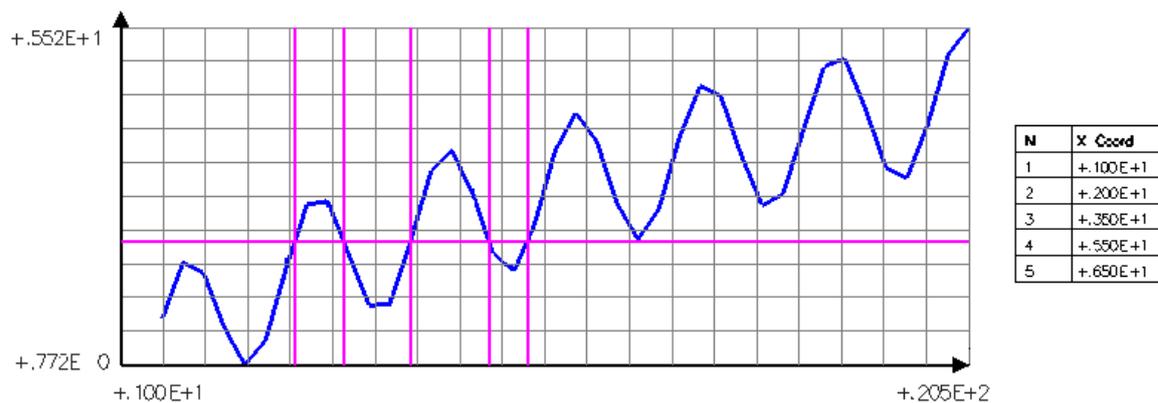


Figura 3.10 – Gráfico

3.3.1.5 – Deformada

Na criação da deformada também foi necessário a leitura do arquivo de saída de dados e o armazenamento destes dados em vetores, processo muito semelhante ao de criação dos gráficos, porém o desenho da tubulação deformada consistiu na criação de trechos da tubulação independentes em cada nó da estrutura, logo a criação se deu da mesma maneira a apresentada na criação da malha, como em cada nó tem seu próprio deslocamento em cada uma das coordenadas X, Y e Z, a não ser claro nas condições de apoio onde estes valores já eram pré-definidos, bastava à leitura das deformações em cada nó e somá-las a suas coordenadas respectivamente.

A animação da deformada foi feita através da função *glutTimerFunc* esta função é responsável pela associação de uma função de *callback* a ser executada de tempos em tempos. O primeiro parâmetro define o tempo em milissegundos que deve haver entre as chamadas da função de *callback* que será chamada, já o segundo parâmetro deve ser o nome da função *callback*. Sendo assim, a função de *callback* definida pelo usuário deve receber um valor inteiro como parâmetro (Schreiner, 2004; Wright, 2000). Os trechos de código a seguir demonstram como foi feita os principais pontos da programação e a Figura 3.11 mostra em detalhes o resultado.

```

if (scy >= 1.0f && cx == 1)
{
    scy += 0.1f;
    cx = 1;
    //if (scy <= 0.0f) { scy -= 0.1f; }
    //if (scy >= 1.0f) { scy += 0.1f; }
    if (scy >= 5.0f) { cx = 0; }
    i = i + 1;
}

```

```

if (scy >= 1.0f && cx == 0)
{
    scy -= 0.1f;
    cx = 0;
    //if (scy <= 0.0f) { scy += 0.1f; }
    //if (scy >= 1.0f) { scy -= 0.1f; }
    if (scy <= 1.1f) { cx = 1; }
    i = i + 1;
}

```

```

glutPostRedisplay();
glutTimerFunc(33, Timerx, 1);

```

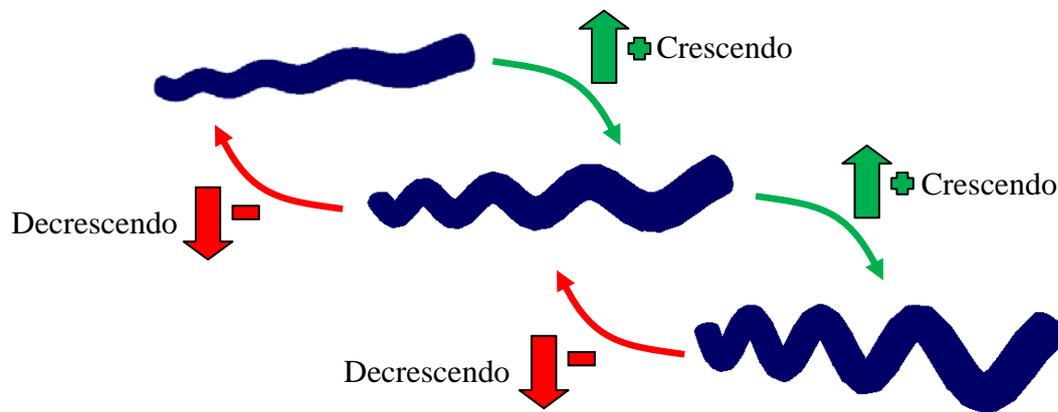


Figura 3.11 – Animação da estrutura deformada

3.3.2 – Tela de Saudação

Logo após a execução do programa é mostrado ao usuário uma tela de saudação, onde consta o nome da Universidade de Brasília, o nome do Departamento de Engenharia Civil e Ambiental e do Programa de Pós-Graduação em Estruturas e Construção Civil assim como o tema da dissertação de mestrado, o nome do programa que foi utilizado como “objeto” da interface, os nome dos orientadores e do mestrando, por fim os logotipos do Programa de Pós Graduação em Estruturas e Construção Civil, da Universidade de Alberta e do *AltaPipe Group*, como pode ser visto na Figura 3.12.

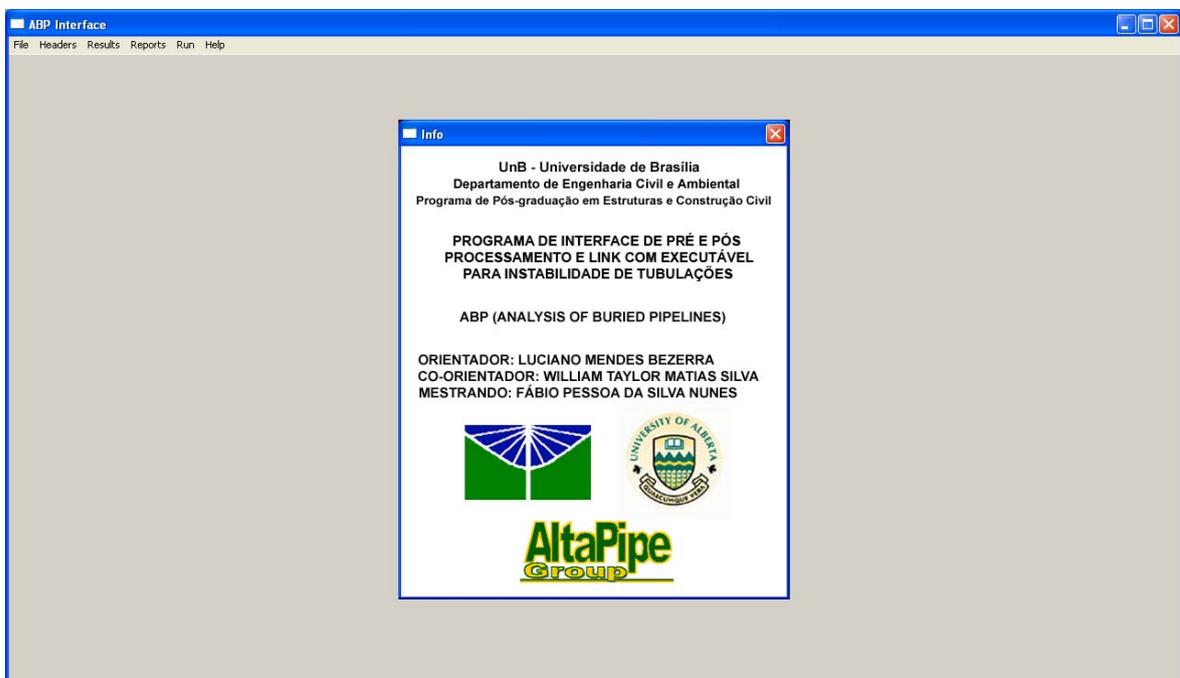


Figura 3.12 – Tela de Saudação

3.3.3 – ABP Interface

O programa de Interface do ABP foi estruturado com cada executável independente, ou seja, cada tela que é visualizada no ABP é na verdade um programa executável do tipo *exe*, onde um programa central, mostrado na Figura 3.13, tem a função de chamar todos os demais programas e controlar o fluxo de dados entre eles, o código abaixo mostra como é feita a chamada de um dos executáveis, no total foram **17634** linhas de código.

```
newfile[0] = "New.exe";
newfile[1] = NULL;
spawnv( P_NOWAITO, "New.exe", newfile);
```

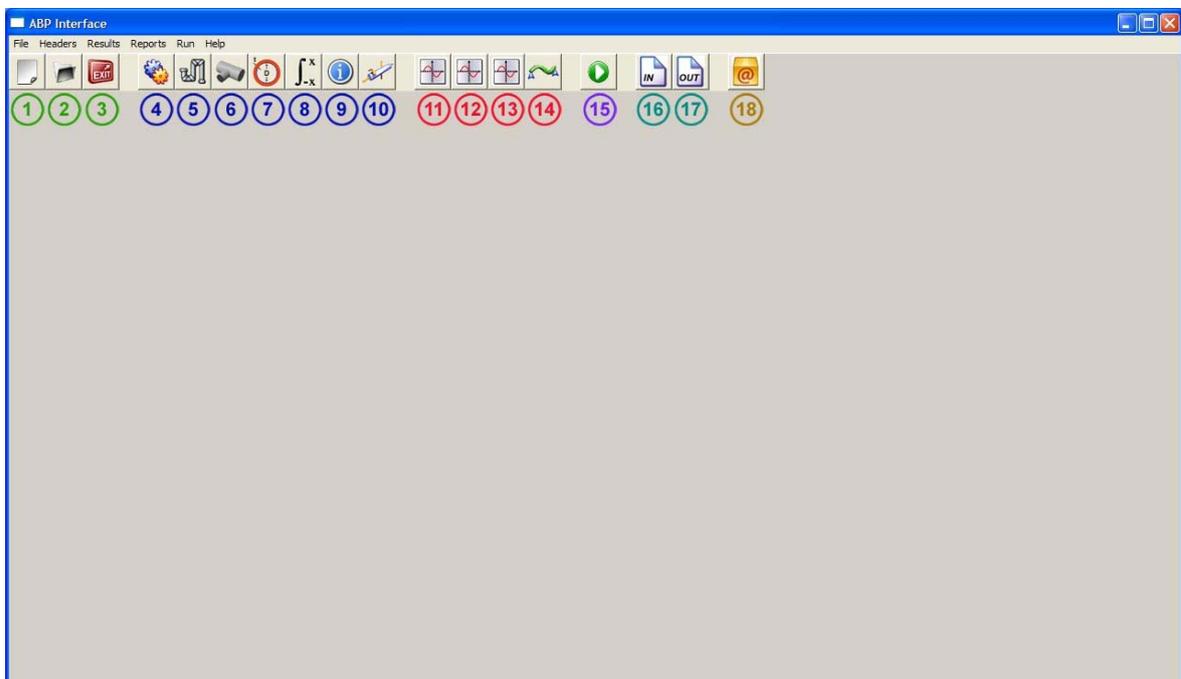


Figura 3.13 – Tela inicial da Interface Gráfica do ABP

1	New	4	General	11	Graphics1	16	Rep. IN
2	Open	5	Nodal	12	Graphics2	17	Rep. OUT
3	Exit	6	Material	13	Graphics3	18	About
		7	Pipe Size	14	Strain		
		8	Attribute	15	Run		
		9	Elements				
		10	Loads				

3.3.4 – New

Nesta tela é definido o nome do arquivo que será usado durante toda a execução do programa, primeiramente é necessário definir duas variáveis do tipo FILE, após isto é atribuído uma string informando um nome que será utilizado para criar um arquivo auxiliar, *filename.aux* e criar a partir da variável *buf_file_name_NW*, variável esta que recebe o nome digitado pelo usuário em *File Name*, este arquivo auxiliar será armazenado com o valor digitado pelo usuário, este arquivo auxiliar irá ser usado por todos os demais programas, pra que estes possam entender qual o arquivo que esta sendo usado, e assim armazenar os parâmetros necessários para a execução do ABP. Este procedimento é realizado ao clicar no botão *Store*, como pode ser visto na Figura 3.14.

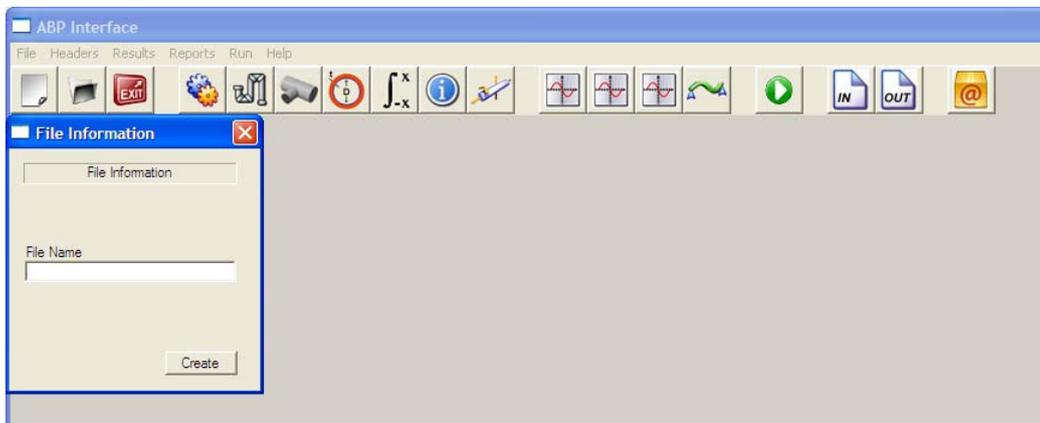


Figura 3.14 – Tela *File Information*

3.3.5 – General Analysis

Esta é a primeira tela que efetivamente vai escrever parâmetros no arquivo de entrada, neste ponto serão definidos parâmetros de entrada os quais definiram entre outras coisas o tipo de análise que será realizada e o comprimento de arco inicial, por exemplo, posteriormente serão explicados cada um desses elementos, o processo de criação dessa tela se deu utilizando diversos dos componentes explicados no tópico referente a *Win32*, onde cada um desses elementos, dependendo da interação do usuário, definirá que tipo de dado será escrito no arquivo de entrada, dependendo do elemento este pode ser arbitrário como no caso dos elementos *edits*, ou com valores pré-definidos como no caso de *como boxes*. Após definido todos os parâmetros basta ao usuário clicar no botão *Store*, e todos os valores serão escritos no arquivo de entrada. A Figura 3.15 mostra a tela de *General Analysis* e logo após uma descrição de cada parâmetro.

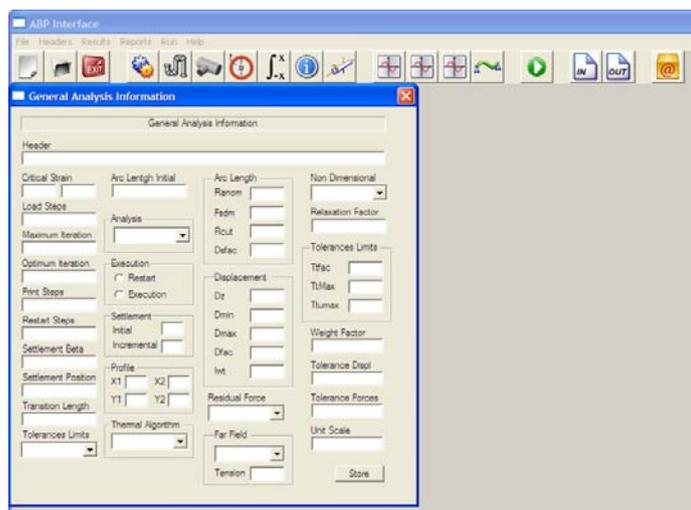


Figura 3.15 – Tela *General Analysis*

HEADER – Nome do projeto ou descrição da análise que será realizada

ANALYSIS – Tipo de análise, que tem como opções: THERMAL, SETTLEMENT ou APPLIED LOADS

ARC LENGTH INITIAL LENGTH – Comprimento inicial de arco, esta variável é usada para todos os tipos de análise. Para análise de recalque, este parâmetro irá estabelecer a norma inicial de deslocamentos.

Se o programa falhar na convergência, o valor do comprimento inicial de arco será incrementado (de uma ou mais ordens de magnitude) antes da análise começar a convergir. Além disso, se isto levar muitos passos para convergir para um típico passo de carga, o valor do comprimento inicial de arco deverá ser incrementado também. O usuário pode começar com o valor zero para o comprimento inicial de arco, então se o procedimento padrão do programa não funcionar, o usuário deverá começar a análise com um valor mais alto.

CRITICAL STRAIN – Deformação Crítica, valor n da equação de Ramberg Osgood

ANALYSIS EXECUTION – Modo de execução RESTART ou EXECUTION. EXECUTION é usado para todas as análises que comecem do zero. Para análises que forem continuar de algum ponto mais adiante deverá ser usado a opção RESTART.

LOAD STEPS – Número de passos de carga

MAXIMUM ITERATION – Esta variável especifica o numero Maximo de interações em cada passo de carga.

OPTIMUM ITERATION – Este valor não é usado para uma análise de recalque, mais usado para soluções de comprimento de arco, se o valor for zero, nenhuma modificação será tomada para o comprimento inicial de arco isto devido ao número de interações. Para um número diferente de zero, o comprimento inicial de arco será ajustado tanto quanto o valor for se aproximando.

PRINT STEPS – Esta variável especifica o intervalo da solução que será escrito no arquivo de saída, nos arquivos *out* e *plt*

RESTART STEP – O arquivo de recomeço (arquivo do tipo *res*) sempre é gerado junto com os arquivos de saída. Contudo o arquivo de recomeço será atualizado a cada passo de carga após o passo inicial. Isto é muito útil quando o usuário quer continuar a análise do passo no qual o programa foi interrompido

SETTLEMENT – Recalque inicial e incremento de recalque em cada passo.

SETTLEMENT BETA – Ângulo de recalque em graus, na Figura 3.16 é mostrado à convenção de sinal adotada pelo programa

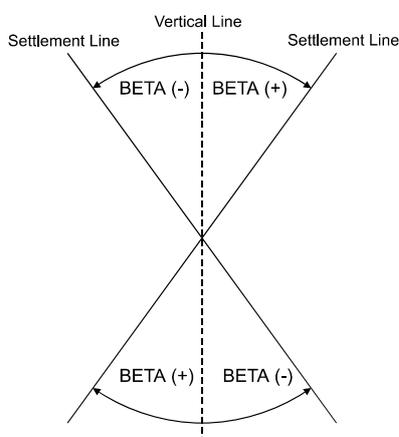


Figura 3.16 – Convenção de sinal

SETTLEMENT POSITION – Esta variável é necessária apenas para um recalque brusco ou uma transição suave da estrutura. Para um recalque brusco, esta variável especifica a coordenada axial do ponto de começo do recalque. Para uma transição mais suave, esta variável especifica a coordenada axial o ponto médio da zona de transição

SETTLEMENT PROFILE – Esta opção especifica o perfil de recalque caso o usuário especifique.

SETTLEMENT TRANSITION LENGTH – Tamanho da zona de transição para um perfil de transição suave. A presença deste cabeçalho especifica a opção de transição suave. Então para um recalque brusco, ou um perfil de recalque definido pelo usuário, este cabeçalho não deve ser incluído.

THERMAL ALGORITHM – Algoritmo de solução, QUADRATIC, LINEAR ou DISPL-CONTROL

THERMAL ALGORITHM ARC.LENGTH – Se divide nos seguintes parâmetros:

- a) *Ranom* - Razão não dimensional no método de comprimento inicial de arco
- b) *Fdsm* - Fator mínimo de redução do comprimento inicial de arco
- c) *Rcut* - Fator de redução do comprimento inicial de arco
- d) *Dsfac* - Fator de acréscimo do comprimento inicial de arco

THERMAL ALGORITHM DISPLACEMENT CONTROL – Se divide nos seguintes parâmetros:

- a) *Dz* - Incremento inicial de deslocamento no método de controle de deslocamento
- b) *Dmin* - Incremento mínimo de deslocamento no método de controle de deslocamento
- c) *Dmax* - Incremento máximo de deslocamento no método de controle de deslocamento
- d) *Dfac* - Acréscimo no fator de tolerância (TOLT) no método de controle de deslocamento

e) *Iwt* - Código para os fatores de ponderação

THERMAL DOF RESIDUAL FORCE – Índice de seleção de DOFs (Graus de Liberdade) para ser usado no cálculo da força residual

THERMAL FAR FIELD – Condição térmica de campo

THERMAL FAR FIELD TENSION – Tensão inicial na tubulação para uma condição específica

THERMAL NON DIMENSIONAL – Temperatura adimensional, opção *YES* ou *NO*

THERMAL RELAXATION FACTOR – Fator de relaxação

THERMAL TOLERANCES LIMITS – Se divide nos seguintes parâmetros:

- a) *Tfac* - Fator de incremento para a tolerância da força residual
- b) *Ttmax* - Limite superior para a tolerância da força residual
- c) *Ttymax* - Limite superior para a tolerância do deslocamento

THERMAL WEIGHT FACTOR – Fator de ponderação de temperatura no cálculo do carregamento térmico

TOLERANCE DISPL – Tolerância para a norma de incrementos de deslocamentos

TOLERANCE FORCES – Tolerância para a norma de forças residuais

UNIT SCALE – Fator de escala

3.3.6 – Nodal Informations

Após a definição dos parâmetros gerais da análise, a próxima etapa fica por conta das informações relativas à malha, esta tela foi a primeira aplicação que utilizou a biblioteca gráfica *OpenGL*, o ABP interpreta a malha da tubulação usando elementos de barra, logo a interpretação gráfica da malha foi definida como um tubo utilizando um conjunto de linhas rotacionadas de acordo com as coordenadas definidas pelo usuário, o processo de geração da malha ocorre quando o usuário após definir dois pontos nas coordenadas x, y e z, informa em quantos elementos serão divididos aquele segmento, posteriormente o usuário deve informar as condições de suporte entre esses dois pontos e se estes tem alguma imperfeição nodal, além disto existe as opções de limpar a malha, *redo e undo*, o processo de geração ocorre ao clicar no botão *mesh* e para armazenar os valores com o botão *Save*, as Figuras 3.17, 3.18 e 3.19 ilustram a modelagem e as possibilidades de *zoom* e *pan* aplicadas nos elementos gráficos.

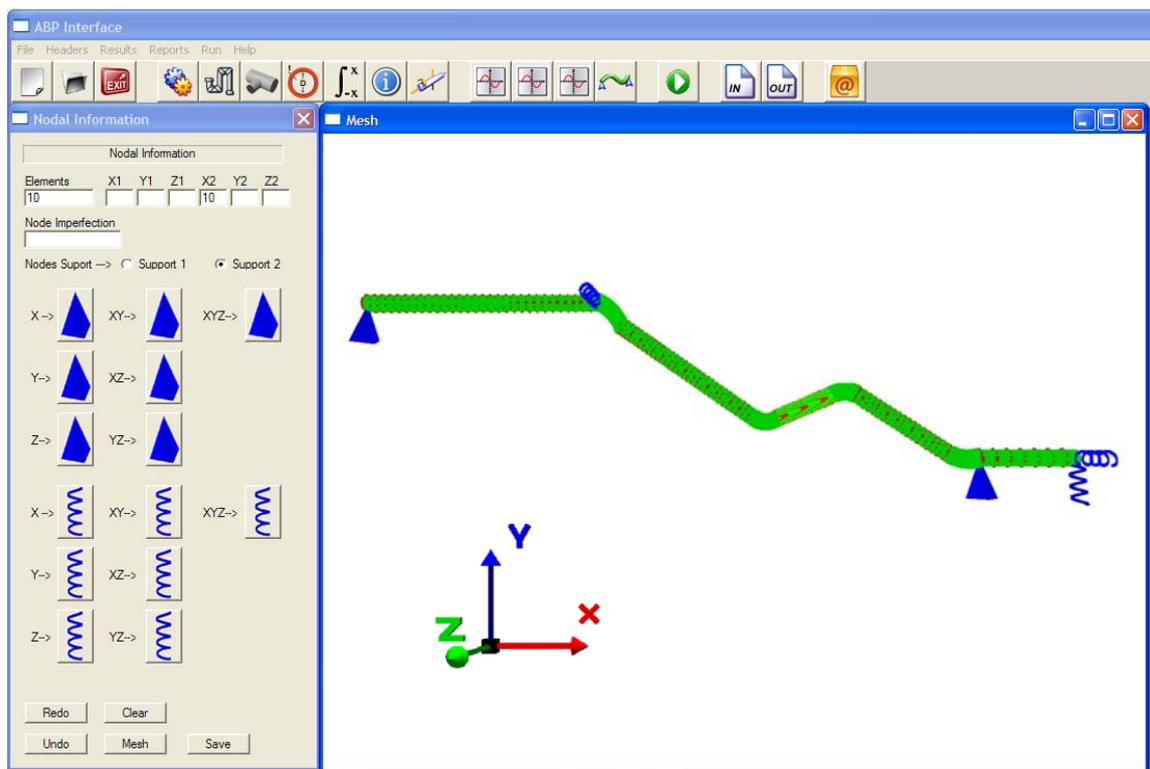


Figura 3.17 – Tela *Nodal Information*

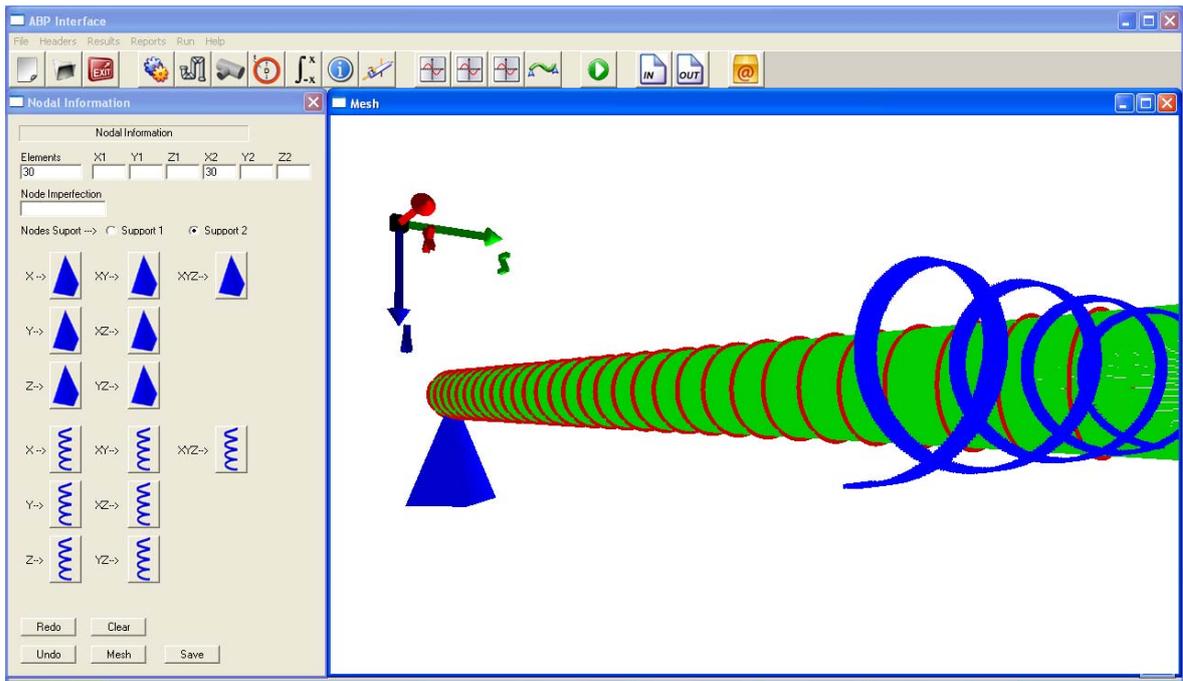


Figura 3.18 – Tela *Nodal Information* com *zoom* e *pan*.

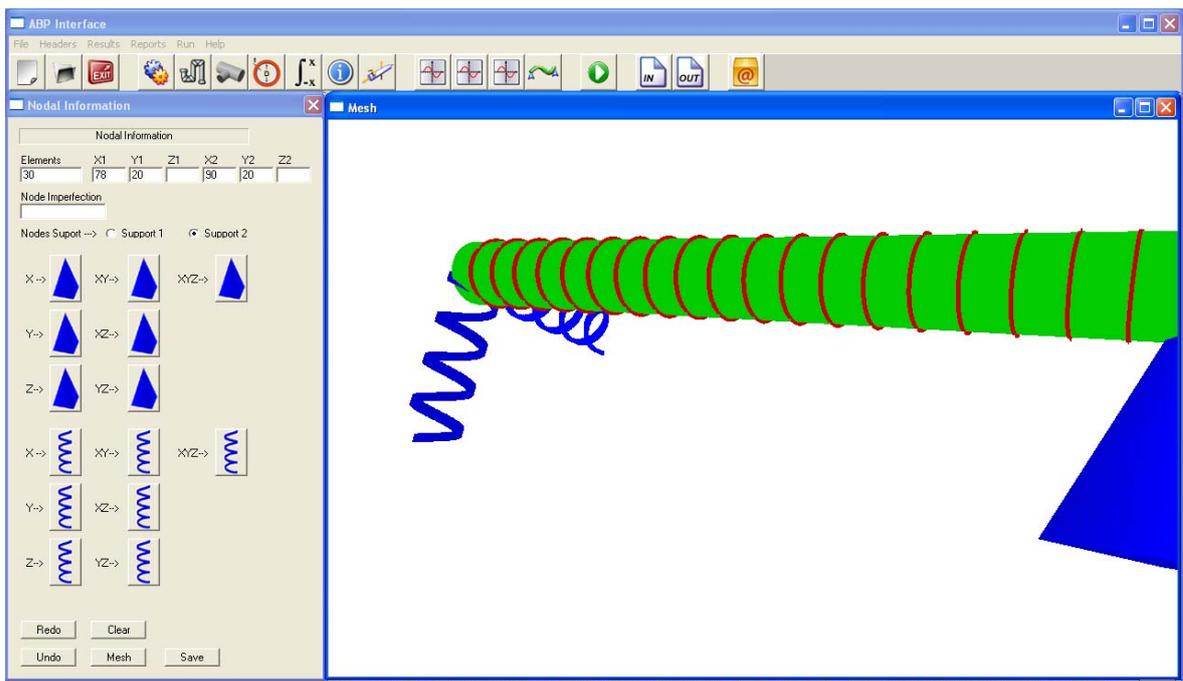


Figura 3.19 – Tela *Nodal Information* com *zoom* e *pan*.

ELEMENTS – Número de elementos em que será subdividido o segmento

X, Y e Z – Coordenadas de início e fim do segmento.

NODE IMPERFECTION – Imperfeição nodal, se as imperfeições forem globais, elas não precisam ser especificadas para cada nó, nenhum tipo de geração será necessário. Todos os valores de dx e dy são multiplicados pelo fator de amplificação, e este será adicionado para cada coordenada correspondente

NODE SUPPORT – Número do nó e em que direções ele esta restringido.

3.3.7 – Material Information

Esta tela definirá as propriedades dos materiais utilizados no programa, o processo de construção da tela se deu de maneira semelhante ao das telas anteriores. Utilizando elementos da biblioteca *Win32*, os parâmetros são adicionados no arquivo gerado na tela *New*, e armazenados através de uma subrotina associada aos botões *Add* e *Store*, como pode ser visto na Figura 3.20.

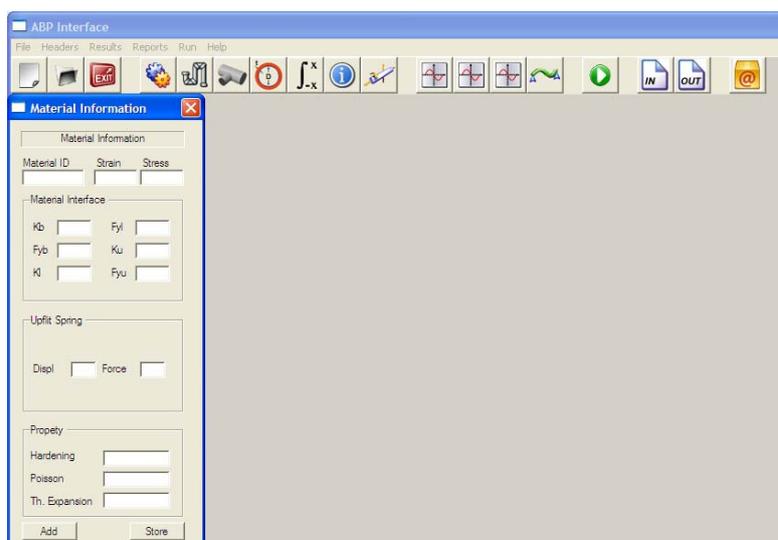


Figura 3.20 – Tela *Material Information*

GENERIC – Para materiais genéricos, informando o numero identificador do material e os valores de tensão e deformação. Este cabeçalho é usado para definir a curva de tensão x deformação associada com a seção da tubulação de aço. Cada par de pontos representa um ponto da curva de tensão x deformação. O primeiro parâmetro do par é o valor da deformação e o segundo o valor da tensão.

INTERFACE – Interface do material, onde:

- a) K_b - Rigidez das molas de situação
- b) F_{yb} - Força efetiva para as molas de sustentação
- c) K_l - Rigidez longitudinal das molas
- d) F_{yl} - Força efetiva longitudinal para as molas de sustentação
- e) K_u - Rigidez de elevação das molas
- f) F_{yu} - Força efetiva para as molas de elevação

INTERFACE UPLIFT SPRING – Este cabeçalho é usado para definir a curva de carregamento x deslocamento associada com a mola de elevação da interface do material. Cada par de pontos representa um ponto da curva de carregamento x deslocamento. O primeiro parâmetro do par é o valor de deslocamento e o segundo é o valor da força.

PROPERTY HARDENING – Propriedade de dureza. Os valores devem estar entre 0.0 e 1.0. O valor zero especifica uma regra cinemática. Valores dentro do intervalo implicam em uma regra mista.

PROPERTY POISSON – Coeficiente de Poisson.

PROPERTY THERMAL EXPANSION – Coeficiente de expansão térmica, este valor vai ser usado para todos os elementos.

3.3.8 – Pipe Size Information

Neste ponto da análise é definida a seção da tubulação que será utilizada, informando qual o diâmetro interno da tubulação e a espessura, onde ao clicar no *Add* se tem uma visualização de como ficará a seção da tubulação, e ao clicar no botão *Store* estes valores serão armazenados no arquivo de entrada, abaixo temos a Figura 3.21 que ilustra a tela *Pipe Size Information* e a Figura 3.22 mostra a mesma figura com *zoom* e *pan*.

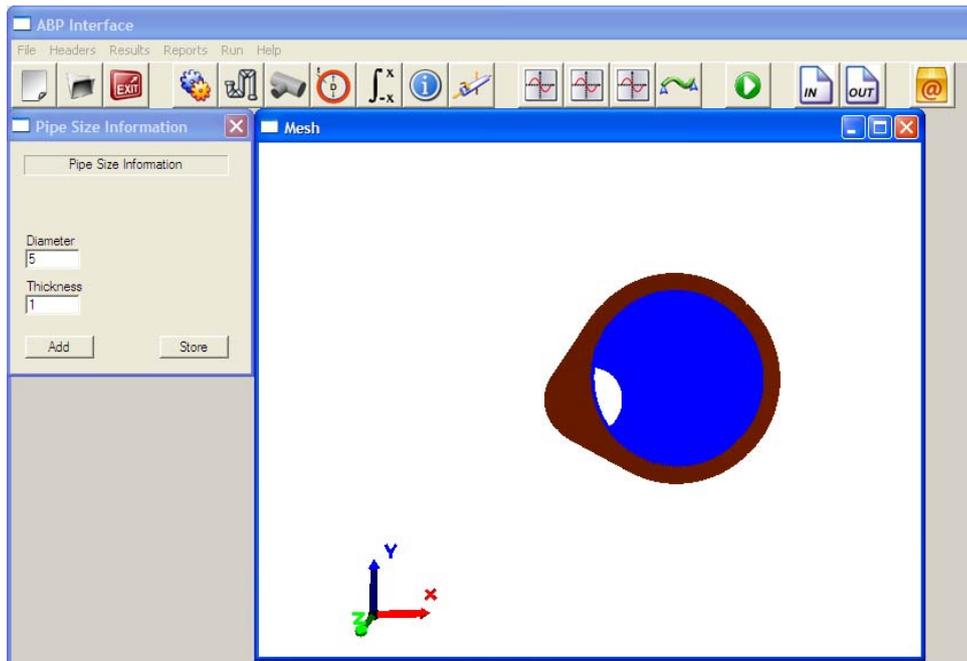


Figura 3.21 – Tela *Pipe Size Information*.

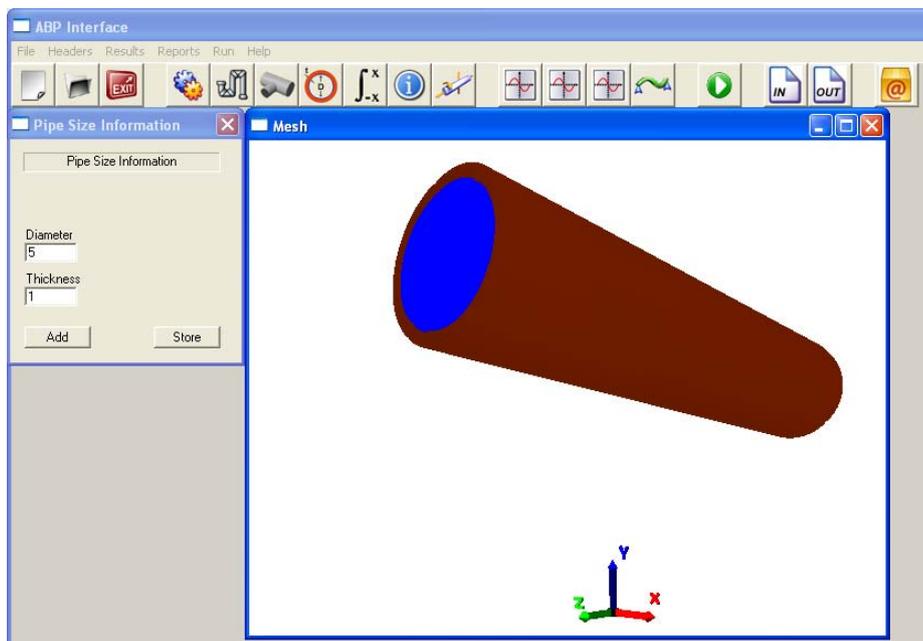


Figura 3.22 – Tela *Pipe Size Information* com *zoom* e *pan*.

SECTION TUBULAR – Diâmetro e Espessura

3.3.9 – Element Attributes Information

Aqui serão especificados os parâmetros de integração da tubulação tanto no sentido transversal quanto longitudinal, onde o processo de criação foi semelhante ao das outras telas, como pode ser visto na Figura 3.23.

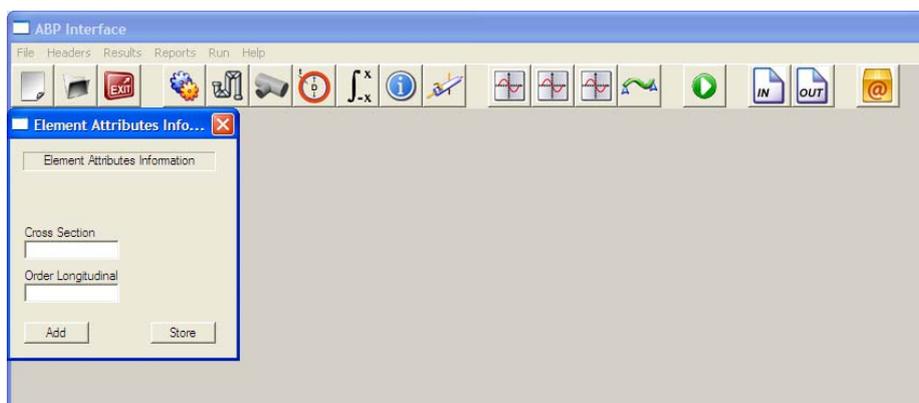


Figura 3.23 – Tela Element Attributes Information

CROSS SECTION – Número de pontos de integração da seção transversal para cada ponto na direção longitudinal. Um número entre 10 e 30 é recomendado.

ORDER LONGITUDINAL – Esta variável especifica o número de pontos de integração para cada elemento na direção longitudinal. Um numero máximo de 6 (seis) pontos de integração precisa ser especificado para cada elemento. Um numero maior ou igual a 5 (cinco) é recomendado.

3.3.10 – Element Information

Até este ponto foram definidos todos os parâmetros relativos ao tipo de análise e dos elementos, aqui serão definidos quais elementos serão associados a cada parâmetro estabelecido anteriormente, como o tipo de material e a seção transversal, por exemplo, o processo de criação foi semelhante ao das demais telas, como pode ser visto na Figura 3.24.

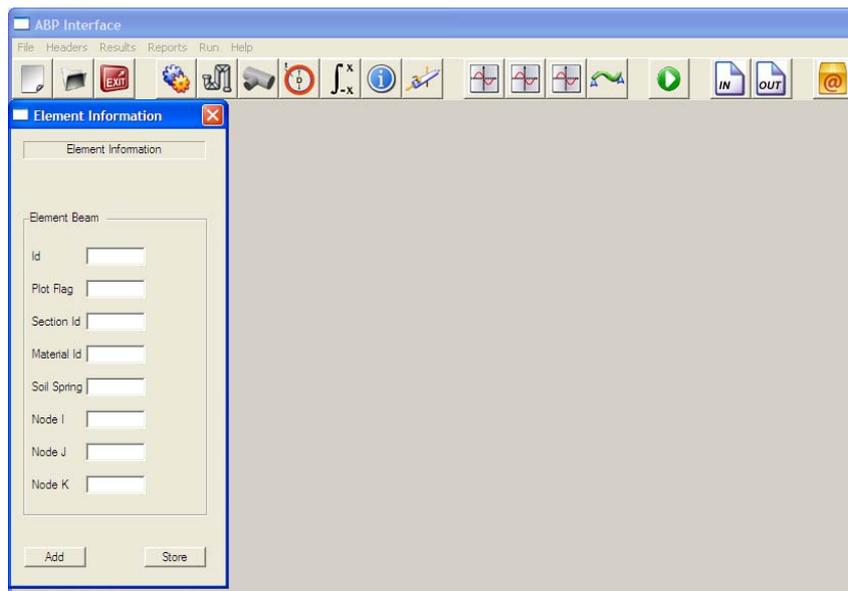


Figura 3.24 – Tela Element Information

ELEMENT – Número de elementos.

ELEMENT BEAM.PIPELINE – Informa ao programa quais das características existentes estão associadas ao elemento. O sinalizador de plotagem especifica se aquele elemento deve ser escrito no arquivo de saída do programa, vale ressaltar que o valor da mola relativa ao comportamento do solo deve ser igual a zero se o elemento não tiver suporte com molas.

3.3.11 – Load Information

Esta é a última tela referente aos parâmetros de entrada dos dados, aqui serão definidos os tipos do carregamento que serão aplicados na tubulação, entre os tipos de carregamento temos: Forças nodais, distribuídas e forças devido a temperatura, o processo de criação também foi semelhante as demais telas, onde os valores são lidos e armazenados através de uma subrotina executada ao clicar no botão *Add* e depois escritos no arquivo de saída ao clicar no botão *Store*, como pode ser visto na Figura 3.25.

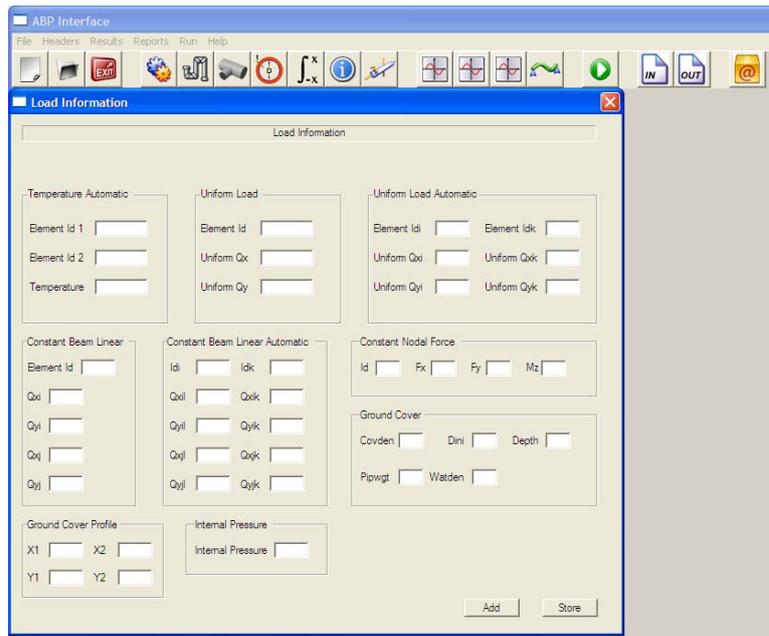


Figura 3.25 – Tela Load Information

TEMPERATURE AUTOMATIC – Para carregamentos devido à temperatura no modo automático.

CONSTANT BEAM LINEAR e REFERENCE BEAM LINEAR – Carregamentos lineares, onde:

- a) qx_i - Componente x de um carregamento linearmente no primeiro nó
- b) qy_i - Componente y de um carregamento linearmente no primeiro nó
- c) qx_j - Componente x de um carregamento linearmente no terceiro nó
- d) qy_j - Componente y de um carregamento linearmente no terceiro nó

CONSTANT BEAM LINEAR AUTOMATIC e REFERENCE BEAM.LINEAR AUTOMATIC - Carregamentos lineares no modo automático.

- a) qx_i - Componente x de um carregamento linearmente no primeiro nó para o elemento m
- b) qy_i - Componente y de um carregamento linearmente no primeiro nó para o elemento m
- c) qx_j - Componente x de um carregamento linearmente no terceiro nó para o elemento m

- d) qy_j - Componente y de um carregamento linearmente no terceiro nó para o elemento m

CONSTANT BEAM UNIFORM e REFERENCE BEAM UNIFORM – Carregamento uniformemente distribuído, onde:

- a) qx - Componente x de um carregamento distribuído
- b) qy - Componente y de um carregamento distribuído

CONSTANT BEAM UNIFORM AUTOMATIC e REFERENCE BEAM UNIFORM AUTOMATIC – Carregamentos uniformemente distribuídos no modo automático, onde:

- a) qx - Componente x de um carregamento distribuído para o elemento m
- b) qy - Componente y de um carregamento distribuído para o elemento m

CONSTANT NODAL FORCE e REFERENCE NODAL FORCE – Forças Nodais, onde:

- a) fx - Força concentrada na direção x
- b) fy - Força concentrada na direção y
- c) mz - Momento concentrado atuando no plano xy

GROUND COVER – Este cabeçalho é usado para modelar o solo da maneira mais precisa usando uma constante uniforme ou um carregamento distribuído trapezoidal. Quando este cabeçalho é incluído, um dos dois tipos deve ser informado, onde:

- a) $covden$ - Densidade efetiva do solo, para solos saturados a densidade da água deve ser subtraída da densidade total do solo.
- b) $Pipwgt$ - Peso próprio da tubulação por unidade de comprimento
- c) $dini$ - Deslocamento mínimo dentro do qual o carregamento permanece constante.
- d) $watden$ - Densidade da água para cálculo de forças de empuxo, zero caso estas forças sejam consideradas

GROUND COVER DEPTH – Profundidade do solo.

GROUND COVER PROFILE – Definição da superfície do solo.

INTERNAL PRESSURE – Pressão interna.

3.3.12 – Graphics

Esta é a primeira tela do pós-processamento, onde os valores resultantes da análise que se encontram no arquivo de saída são lidos e interpretados pelo gerador de gráfico. Esta tela tem as opções de *zoom*, *pan*, alterar a cor do gráfico (verde, vermelho ou azul), aumentar o numero de linhas de grade tanto na direção x quanto na direção y e por fim a opção de procura de valores no gráfico, onde através de uma coordenada de y é informado todas as coordenadas de x nas quais a reta toca o gráfico, o processo de procura pode ser percebido nas Figuras 3.26 e 3.27.

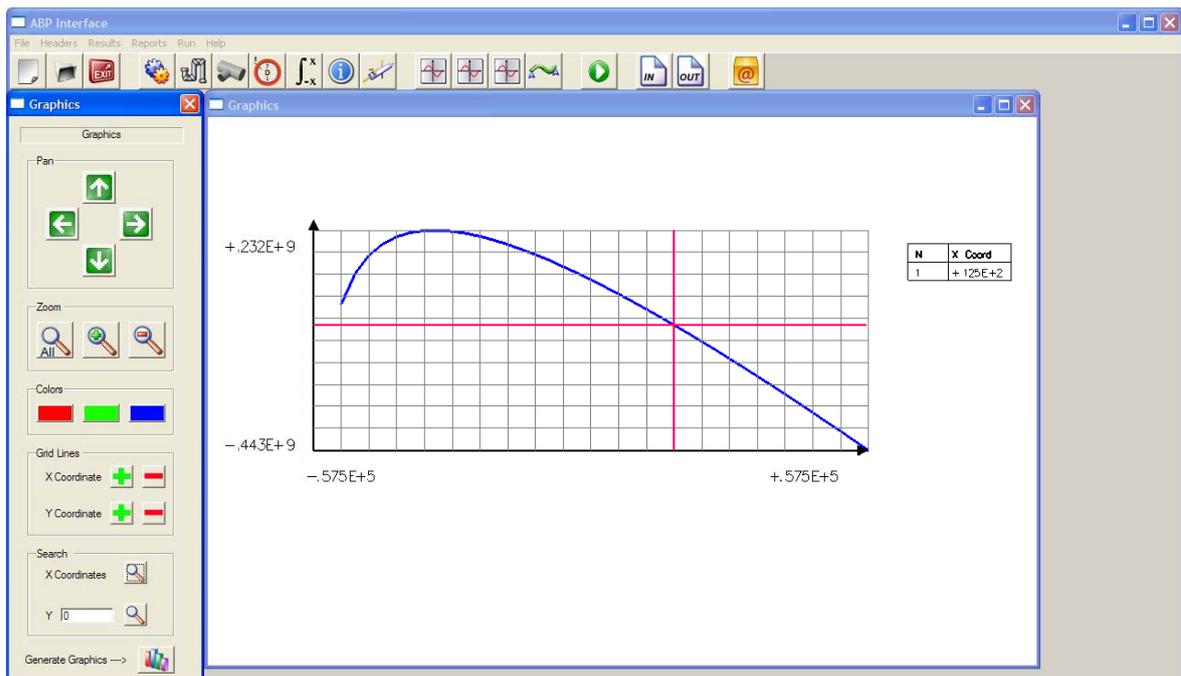


Figura 3.26 – Exemplo de gráfico capturando apenas uma coordenada

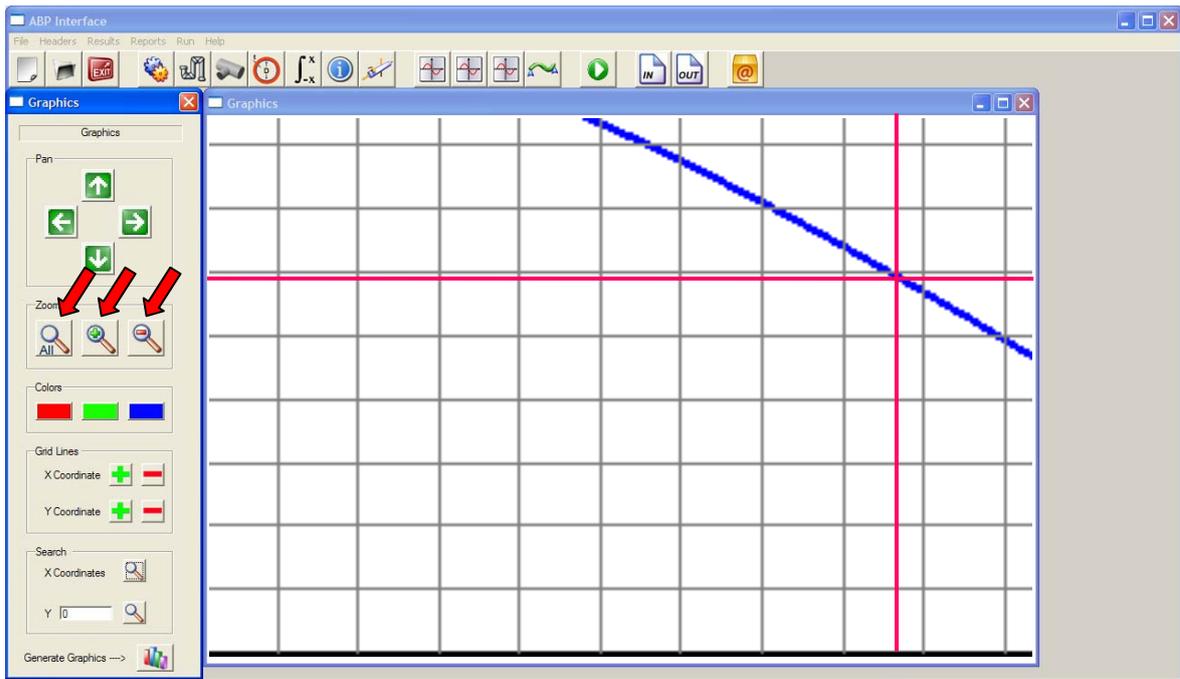


Figura 3.27 – Aplicação de *Zoom* e *Pan*.

No exemplo da Figura 3.28 e 3.29, o qual tem uma complexidade maior do que o exemplo anterior, o usuário pode perceber a captura de diversos pontos em todo o gráfico.

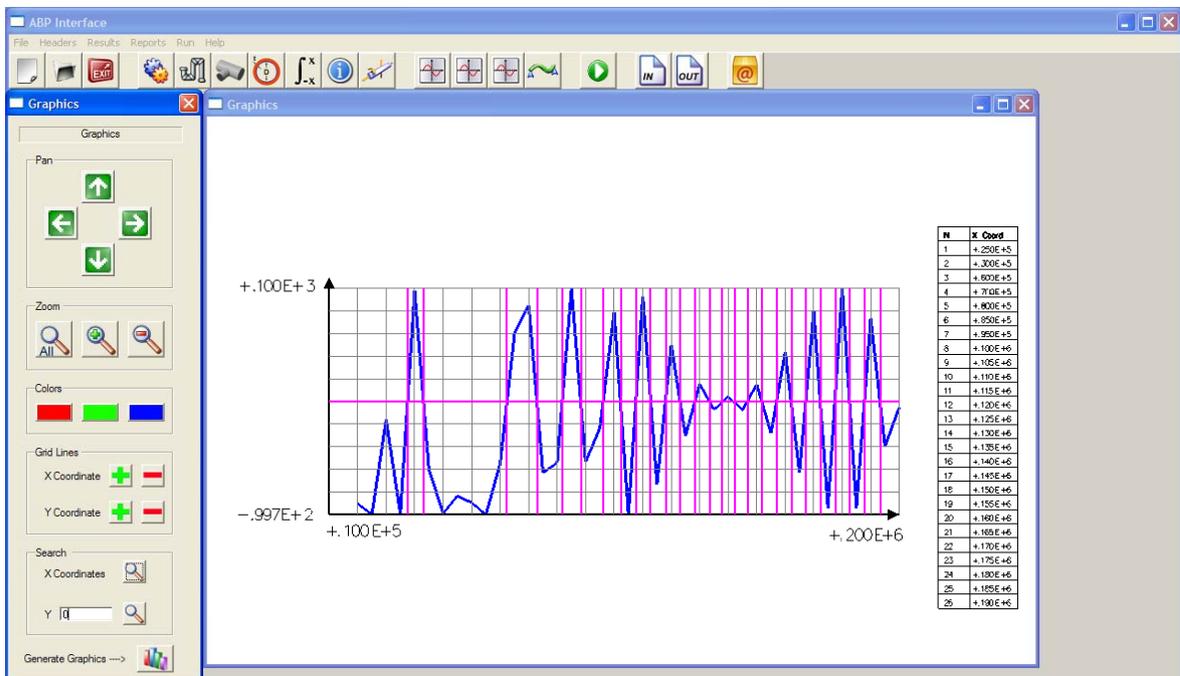


Figura 3.28 – Exemplo de gráfico capturando vinte e seis coordenadas

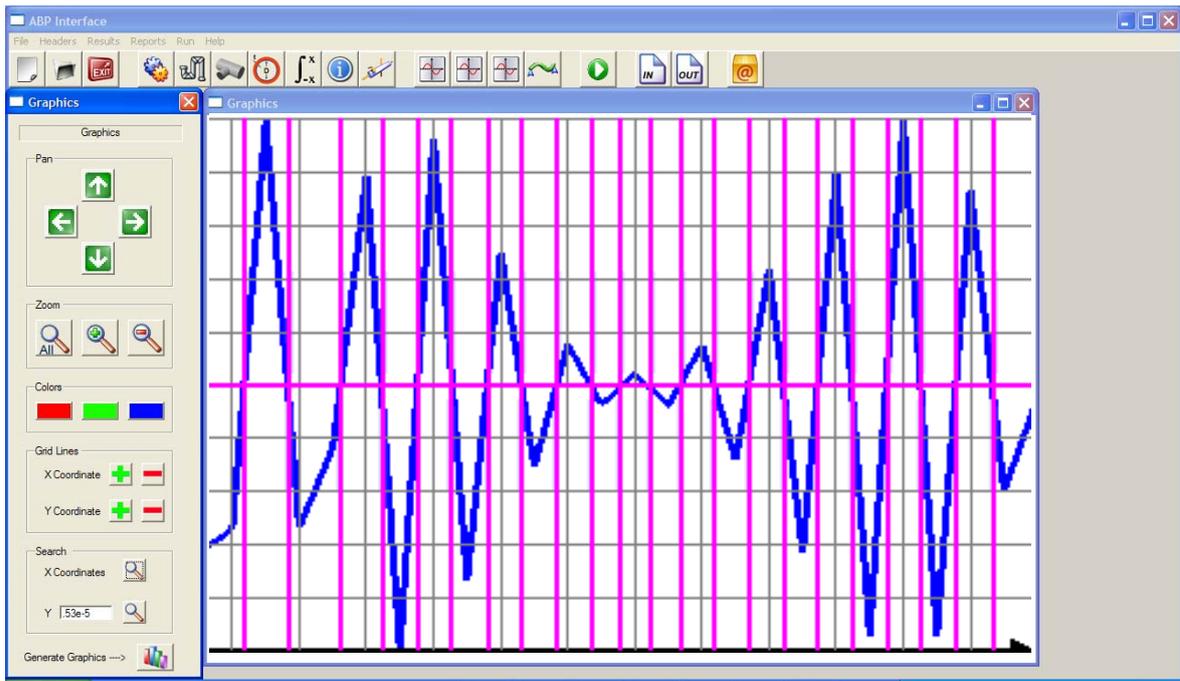


Figura 3.29 – Aplicação de *Zoom* e *Pan*.

3.3.13 – Strain

Esta é a segunda tela do pós-processamento, aqui os dados lidos do arquivo de saída são interpretados por um gerador gráfico semelhante ao de geração da malha, porém poderá ser visualizada a deformada da tubulação, onde o usuário poderá aumentar ou diminuir o fator de escala e até animar a tubulação, como pode ser visto nas Figuras 3.30, 3.31 e 3.32.

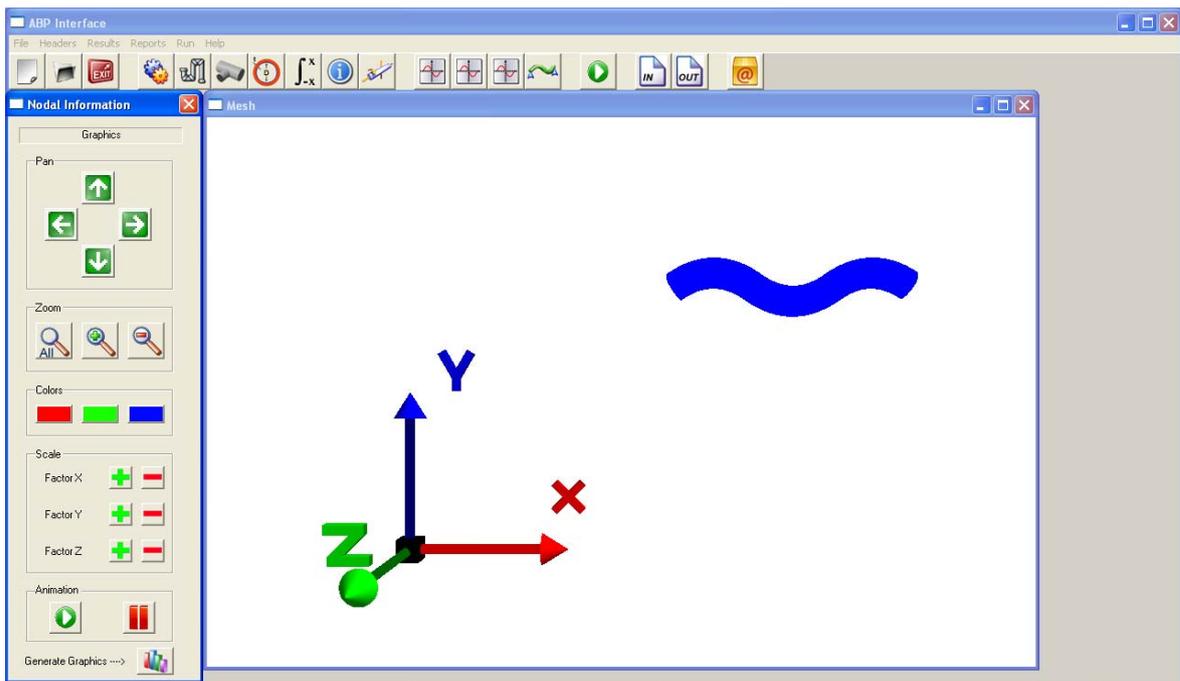


Figura 3.30 – Tela *Strain*

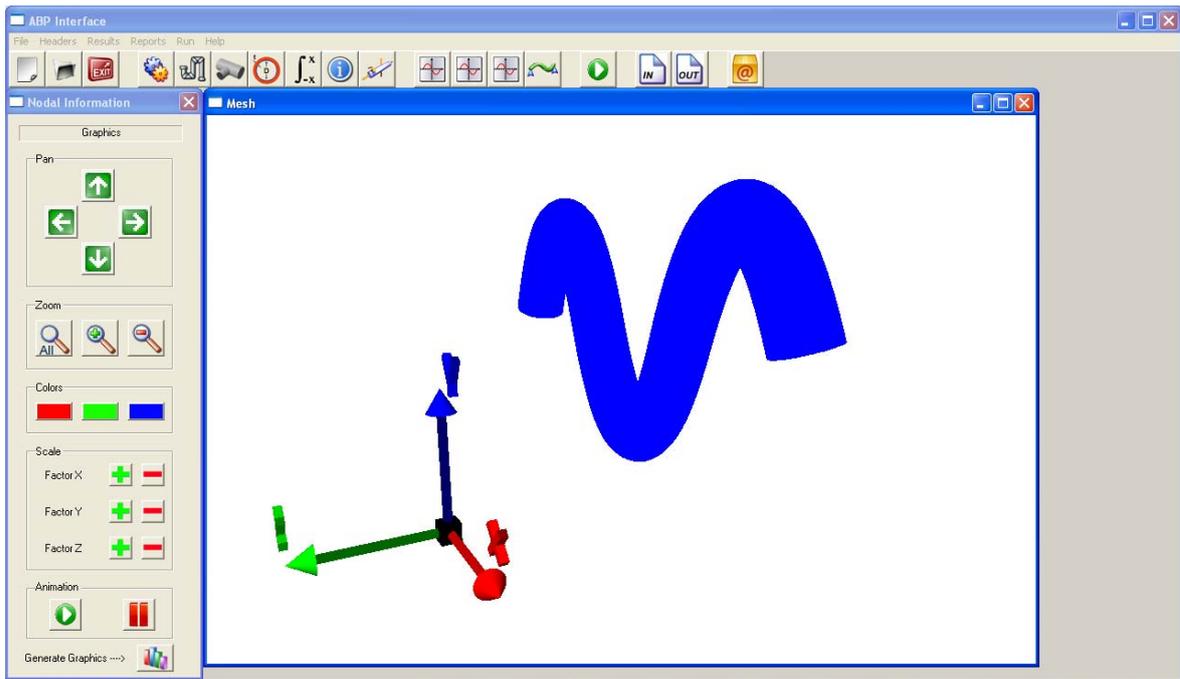


Figura 3.31 – Tela *Strain* com *Zoom*, *Rotação* e *Pan*

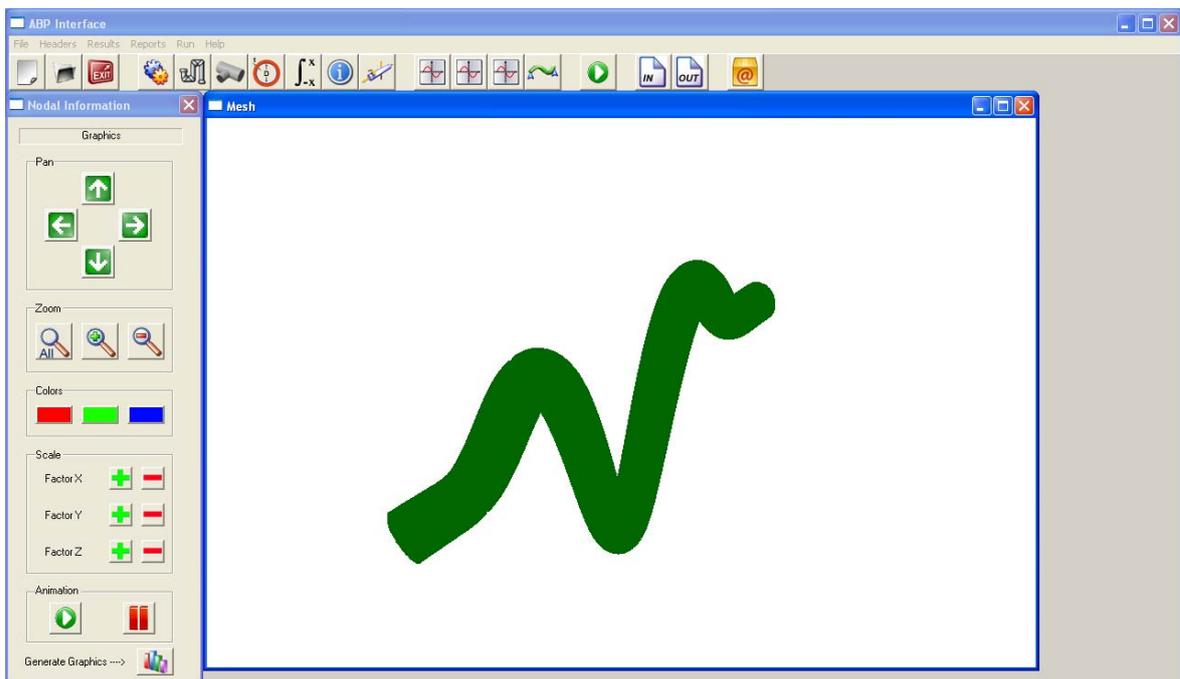


Figura 3.32 – Tela *Strain* com *Zoom*, *Rotação* e *Pan*

3.3.14 – Run

Neste ponto do programa que é feita a ligação com o programa de análise de tubulações enterradas o ABP, o link é feito através da chamada do executável do programa que deve estar na mesma pasta do programa de interface gráfica, após isto o usuário deve digitar o nome do arquivo de entrada *inp* e apertar a tecla *enter*, a Figura 3.33 mostra o ícone que “chama” o ABP indicado pela seta.

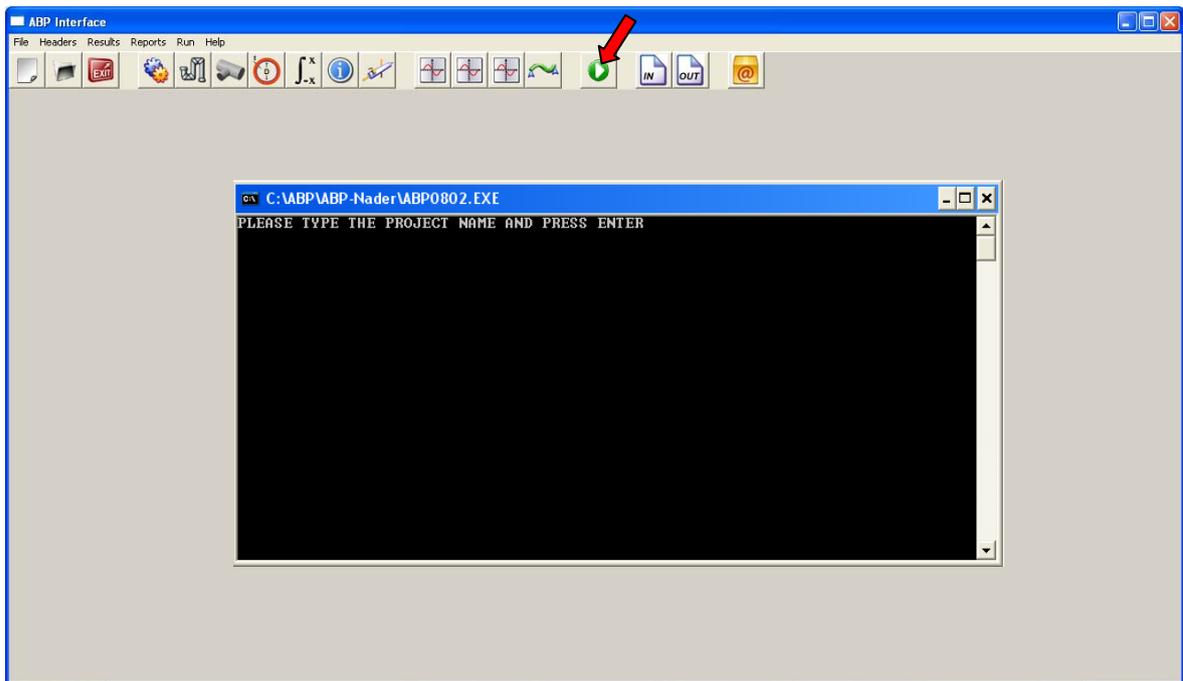


Figura 3.33 – Execução do ABP através da Interface

3.3.16 – About

Esta tela tem a função de informar ao usuário dados sobre o autor como a formação acadêmica e o nome da universidade, UnB (Universidade de Brasília), o departamento, Departamento de Engenharia Civil e Ambiental e o programa, Programa de Pós Graduação em Estruturas e Construção Civil, como pode ser visto na Figura 3.36 acionado pelo ícone indicado pela seta.

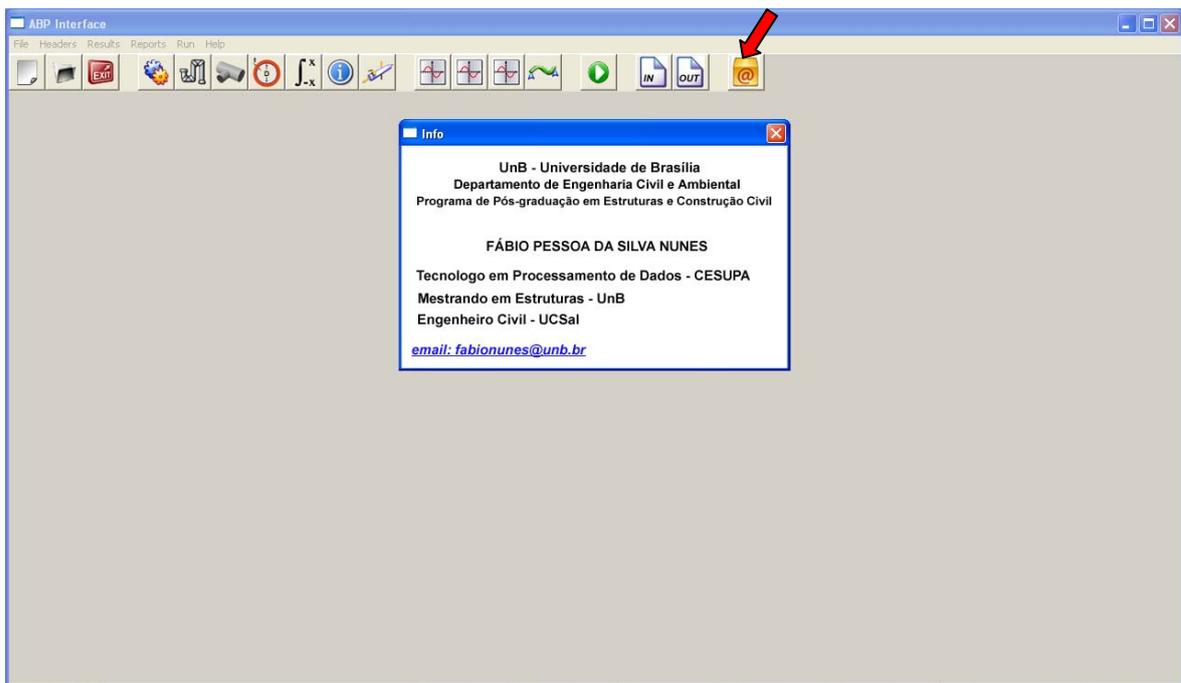


Figura 3.36 – Seta indicando o ícone *About*.

4 – EXEMPLOS

4.1 – EXEMPLO COM CARGA CONCENTRADA

Neste exemplo temos um trecho de tubulação de 3000 cm, impedido de deslocamento em x, y e z conforme figura abaixo, submetido a uma carga concentrada, como mostra a Figura 4.1.

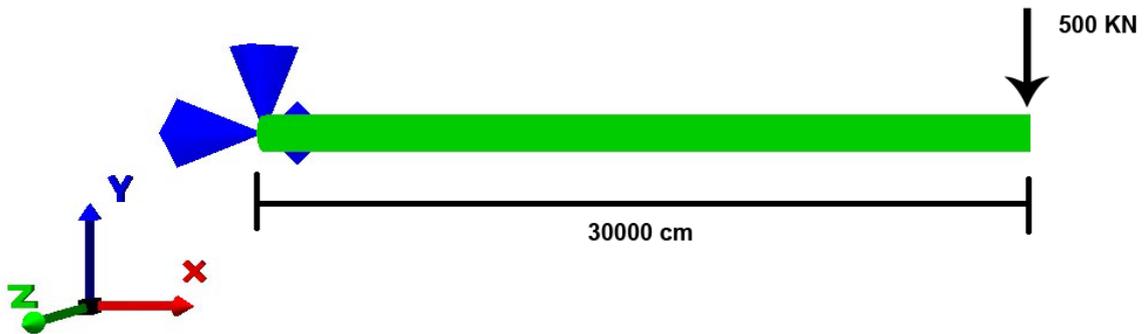


Figura 4.1 – Situação esquemática do 1º exemplo

Nas telas abaixo serão demonstrados todas as telas que fazem parte da análise desde a definição do nome do arquivo passando por todas as telas de entrada de dados: informações gerais, malha, propriedades do material, seção transversal, tipo de integração, informações dos elementos e carregamento, como também as telas do pós-processamento como os gráficos e a deformada.

O primeiro passo é definir o nome do arquivo onde serão armazenadas as informações da análise, acionando botão novo arquivo indicado pela seta na Figura 4.2 irá abrir a tela com o título *File Information*, com o campo *File Name* a ser preenchido com o nome escolhido pelo usuário com a extensão *.inp*, logo após é só acionar o botão *create* e o arquivo será criado como mostra a figura 4.3.

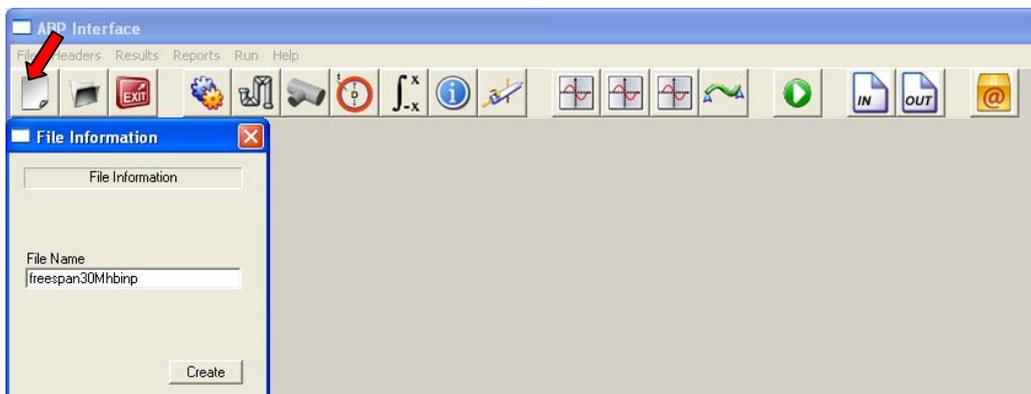


Figura 4.2 – Nome do arquivo

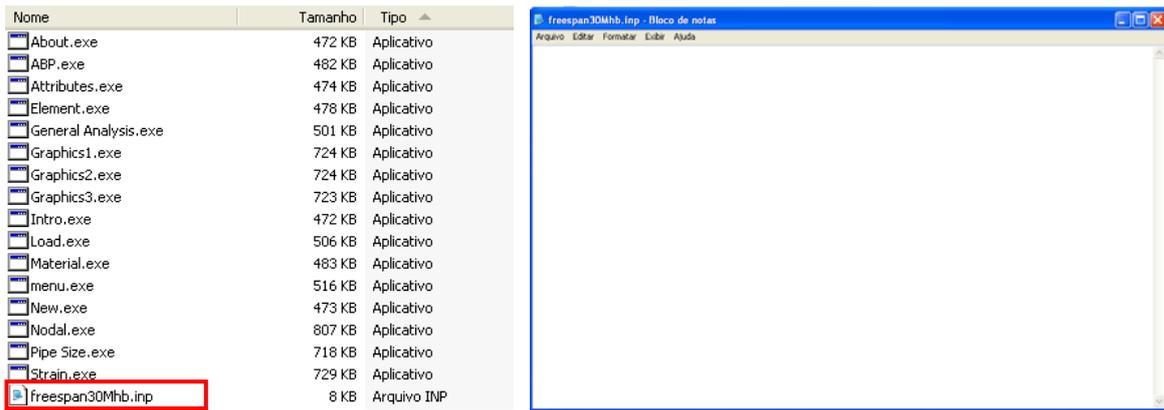


Figura 4.3 – Criação do arquivo freespan30Mhb.inp

Passando agora para o preenchimento dos dados relativos à análise do exemplo em questão, para isso deve-se clicar no botão referente as informações gerais da análise, indicado pela seta na Figura 4.4, o primeiro valor a ser preenchido é justamente o cabeçalho do arquivo que não necessariamente é igual ao nome do arquivo de entrada, neste caso Aplicação de Cargas, a seguir os dados de *Critical Strain* com os valores de 65 e 20, *Load Steps* com o valor de 100, *Maximum Iteration* de interações também com o valor de 100, *Print Steps* com o valor de 1, *Analysis* com *Applied Loads*, *Execution* com a opção *Execution* marcada, *Tolerance Displacement* e *Tolerance Forces* com o valor de 0.05. por fim é só acionar o botão *Store* e todas essas informações serão armazenadas no arquivo *freespan30Mhb.inp* como mostra a Figura 4.5.

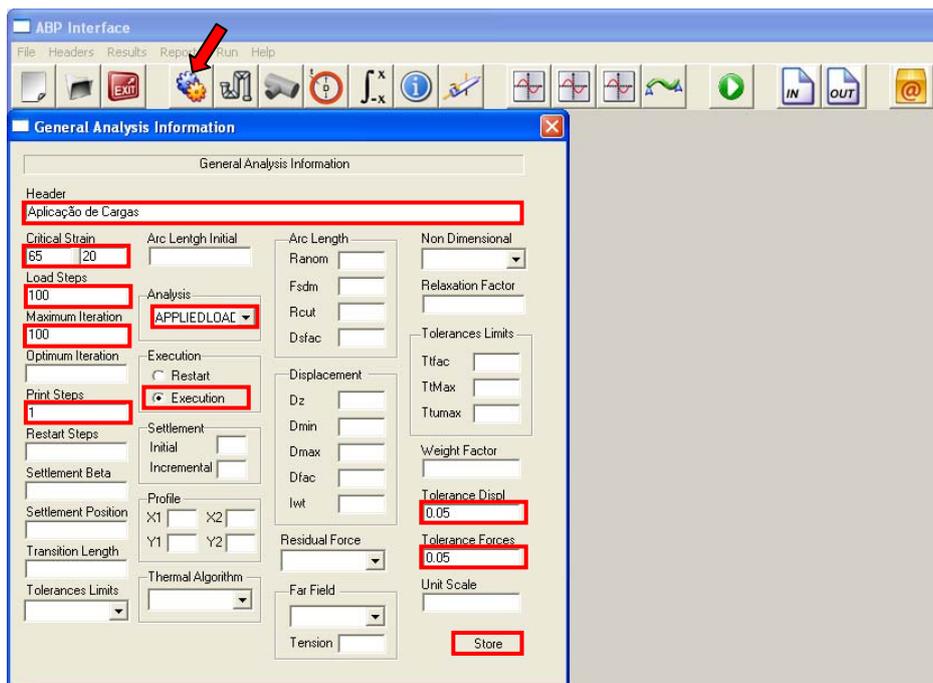


Figura 4.4 – Informações Gerais

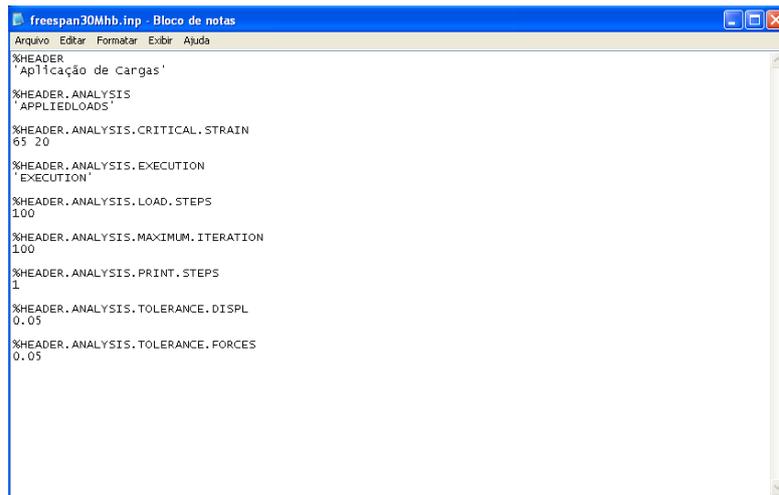


Figura 4.5 – Arquivo *freespan30Mhb.inp* com as informações gerais

O próximo passo é definir a malha de elementos da tubulação, para isso deve se acionar o botão informações nodais indicado pela seta na figura 4.6, o qual irá abrir a tela para que sejam inseridos os parâmetros da malha e mostrada em mais detalhes na Figura 4.7 e 4.8. O primeiro a ser preenchido é o numero de elementos no caso 100 elementos, logo após devem ser informados as coordenadas em X, Y e Z com isto a tubulação já pode ser criada, porém falta as informações relativas as condições de suporte que neste exemplo são restringidas nas três direções no nó 1. Para finalizar temos os botões *Mesh* o qual gera a visualização 3D da malha e o botão *Save* que escreve os dados, já formatados, referentes à malha no arquivo *freespan30Mhb.inp* como mostra a Figura 4.9

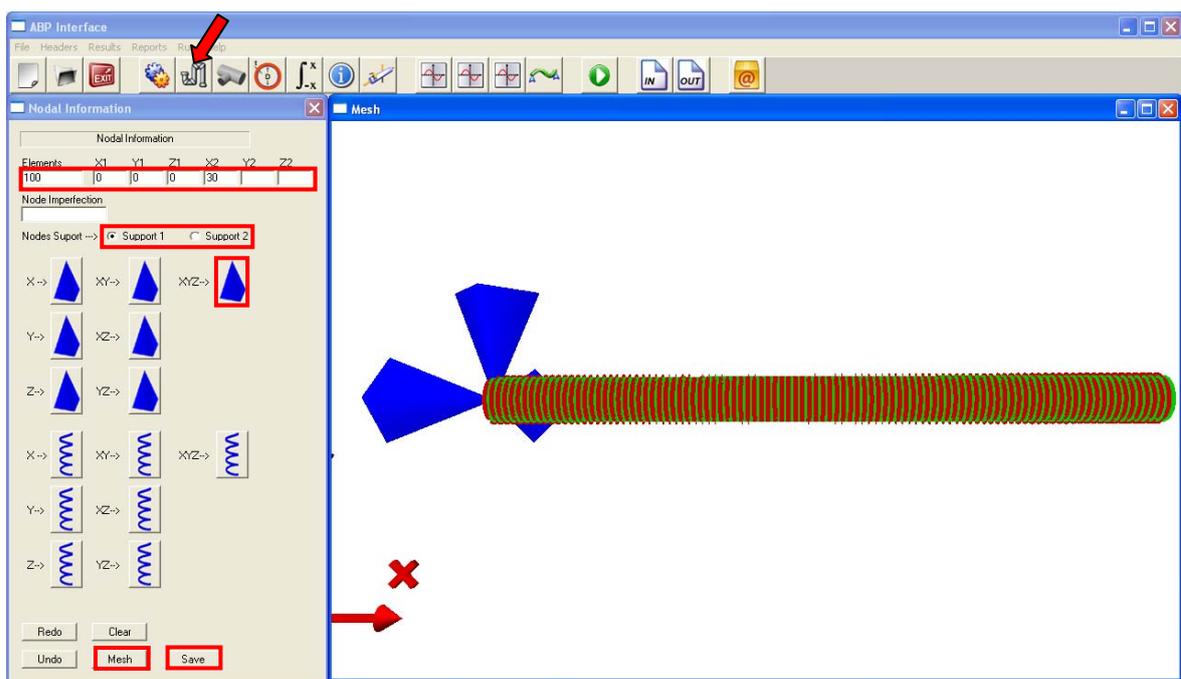


Figura 4.6 – Malha da tubulação.

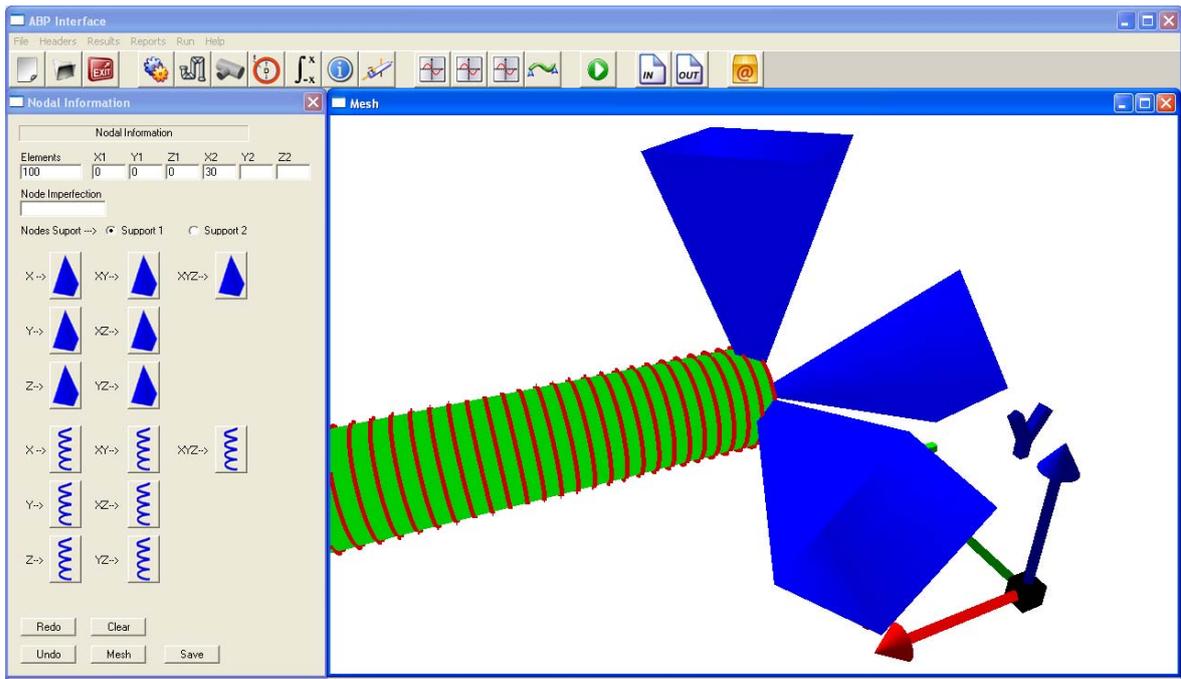


Figura 4.7 – Detalhe do apoio da tubulação

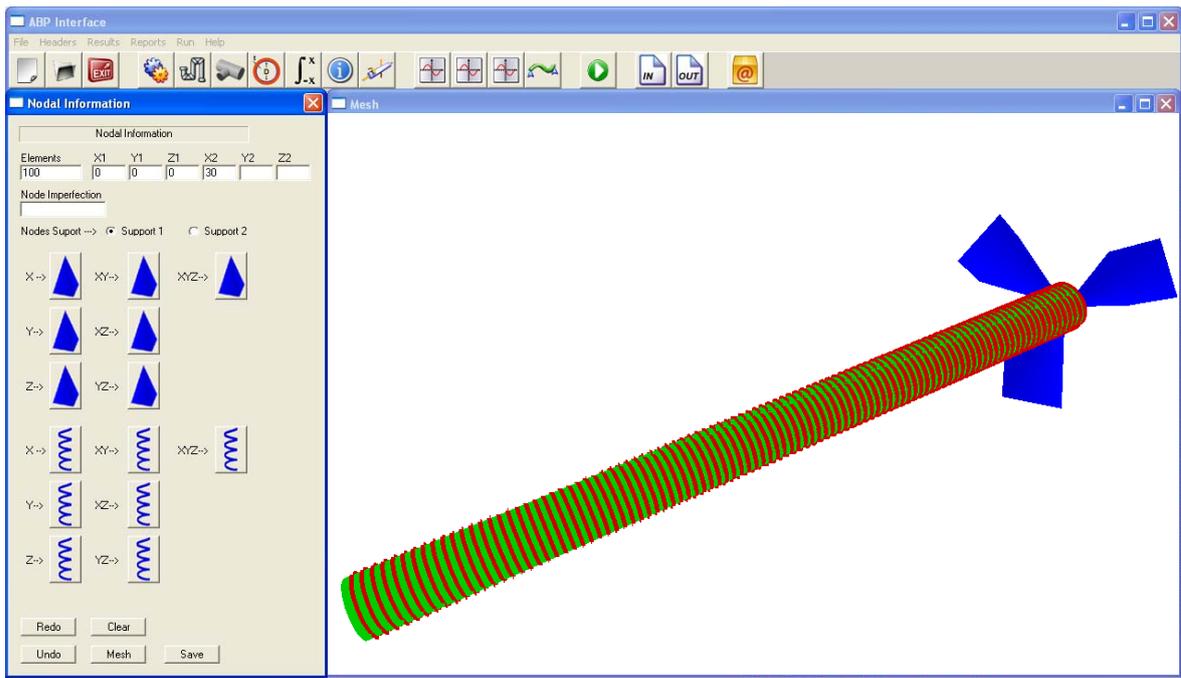


Figura 4.8 – Detalhe do apoio visto de fora da tubulação



Figura 4.9 – Arquivo *freespan30Mhb.inp* com as informações nodais

Definido a malha e as condições de apoio da tubulação o próximo passo consiste nas informações sobre o tipo de material, para inserir essa informação basta acionar o botão de informações do material como indicado pela seta na Figura 4.10, onde para o *Material ID* que recebe o valor 1 temos os valores de *Strain* igual a 0.0042 e *Stress* de 465, já para os parâmetros de *Material Interface* temos *Kb* igual a 2.0834, *Fyb* igual a 6.653, *Kl* igual a 0.8815, *Fyl* igual a 1.763, *Ku* e *Fyu* iguais a 0, já no que se refere ao *Upflit Spring* temos *Displacement* igual a 1 e *Force* igual a 3 e para finalizar temos os parâmetros *Property* com os valores de *Hardening* igual a 1 e *Poisson* igual a 0.3. após esse procedimento é só acionar o botão *Add* e caso não tenha outro material a ser cadastrado é só acionar o botão *Save*, os valores podem ser conferidos posteriormente no arquivo *freespan30Mhb.inp*, como mostra a figura 4.11.

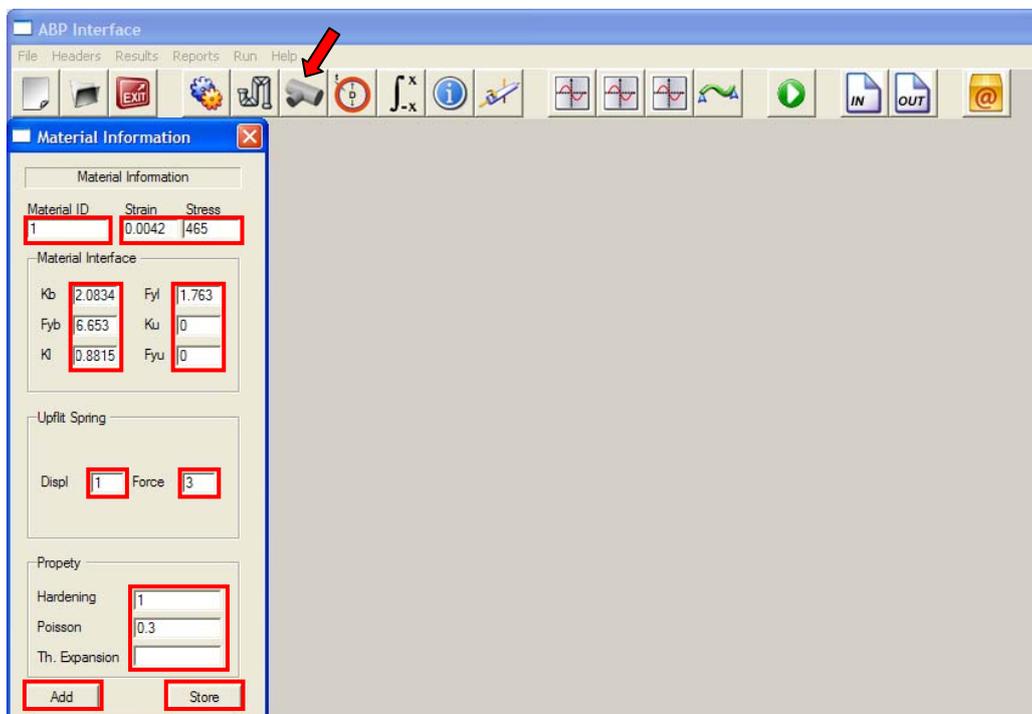


Figura 4.10 – Propriedades dos Materiais

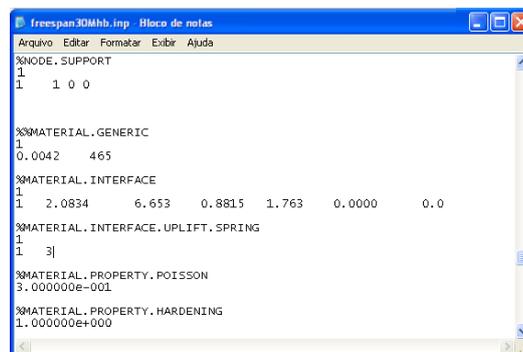


Figura 4.11 – Arquivo *freespan30Mhb.inp* com as informações nodais

O próximo passo é a definição da sessão transversal da tubulação, para inserir essa informação basta acionar o botão indicado pela seta na Figura 4.12 e mostrada em detalhes nas Figuras 4.13 e 4.14, esta etapa requer apenas que o usuário informe o diâmetro interno da tubulação e a espessura com os valores de 1000 e 100 respectivamente, nos campos *Diameter* e *Thickness*, após esta etapa basta acionar os botões *Add* para visualizar como ficou a seção transversal da tubulação e o botão *Store* para armazenar os dados, os valores podem ser conferidos posteriormente no arquivo *freespan30Mhb.inp*, como mostra a figura 4.13.

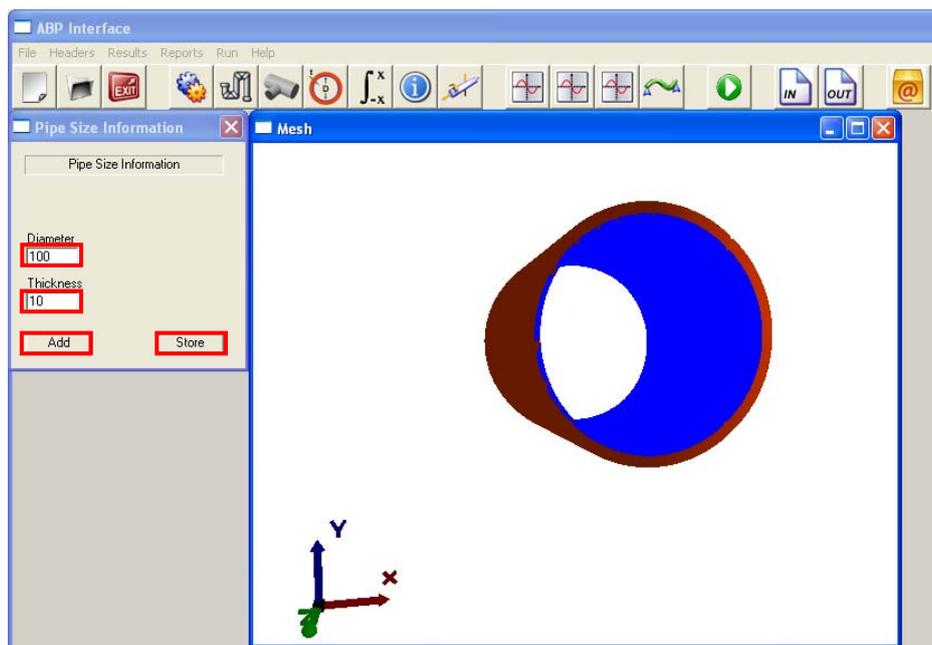


Figura 4.12 – Diâmetro e Espessura da Tubulação

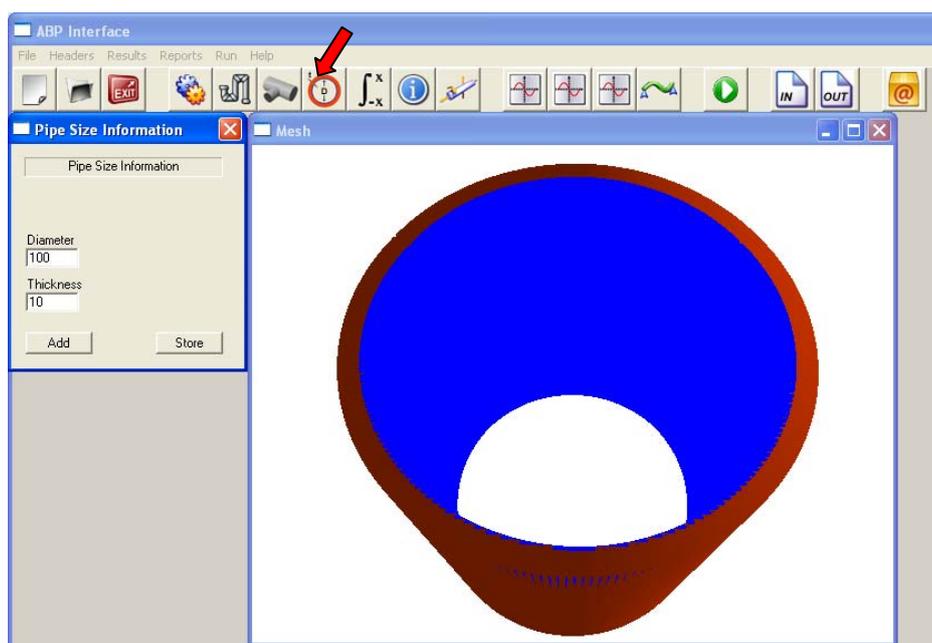


Figura 4.13 – Detalhe com zoom da seção transversal.

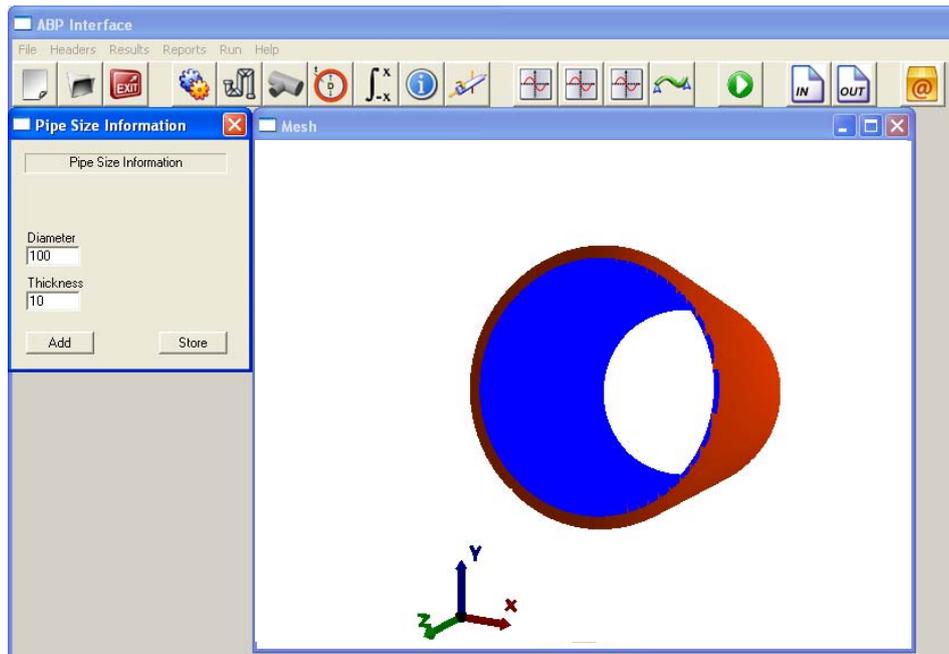


Figura 4.14 – Detalhe interno da tubulação

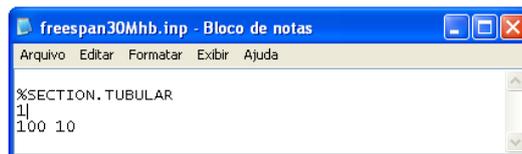


Figura 4.15 – Arquivo *freespan30Mhb.inp* com as informações da seção transversal

Seguindo com a montagem do arquivo de entrada temos a tela de informações dos atributos do elemento, nesta tela é definido como será feita a integração tanto na seção transversal quanto na longitudinal, para inserir esta informações basta acionar o botão indicado pela seta na Figura 4.16, neste caso os campos *Cross Section* e *Order Longitudinal* receberam os valores 1 e 6 respectivamente, após isto é só acionar o botão *Add* e para salvar as informação o botão *Store*, os valores podem ser conferidos posteriormente no arquivo *freespan30Mhb.inp*, como mostra a figura 4.17.

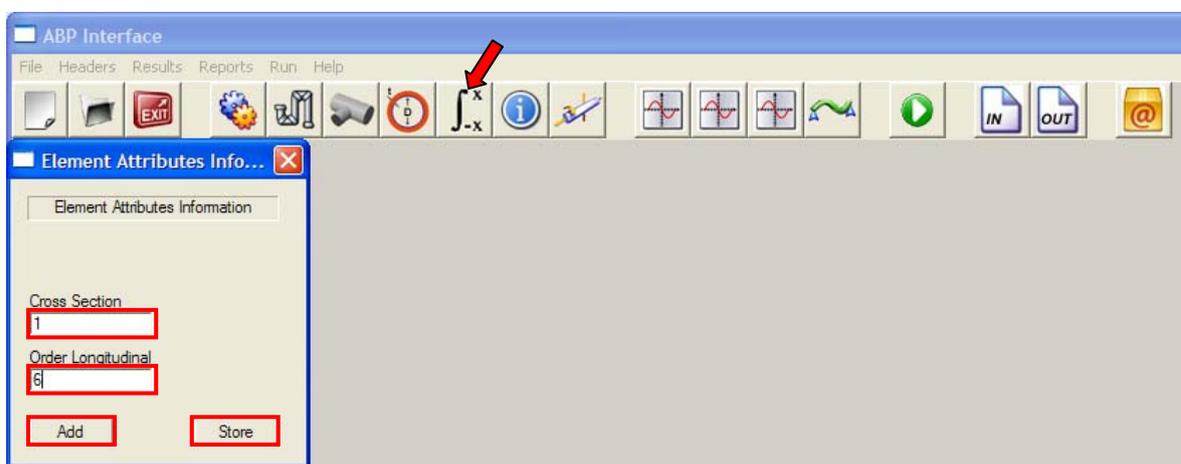


Figura 4.16 – Parâmetros de integração

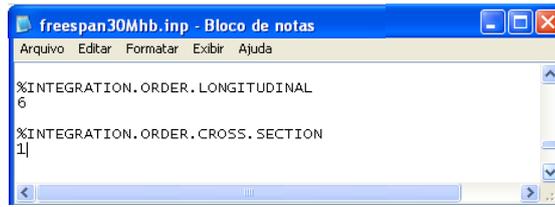


Figura 4.17 – Arquivo *freespan30Mhb.inp* com as informações de integração

O próximo passo nada mais é do que a associação dos dados anteriormente inserido com a identificação do elemento, ou seja é informar para o programa que o referido elemento tem uma determinada seção transversal, de um determinado material, com um determinado coeficiente de mola para o solo entre os nós I, J e K, para inserir essas informações é só acionar o botão indicado pela seta na Figura 4.18, neste caso com os valores 50 para *Id*, 1 para *Plot Flag*, *Section Id*, após isto é só acionar o botão *Add* e para salvar as informação o botão *Store*, os valores podem ser conferidos posteriormente no arquivo *freespan30Mhb.inp*, como mostra a figura 4.18.

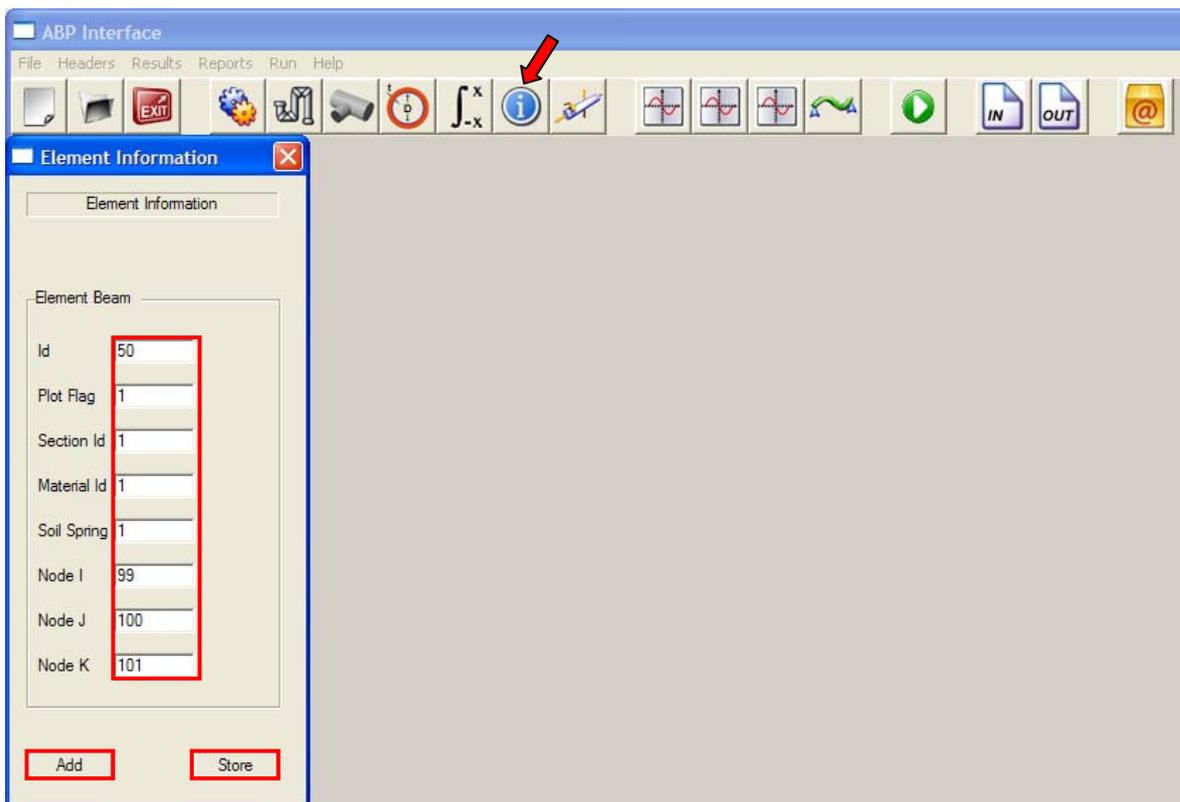


Figura 4.18 – Informações dos elementos

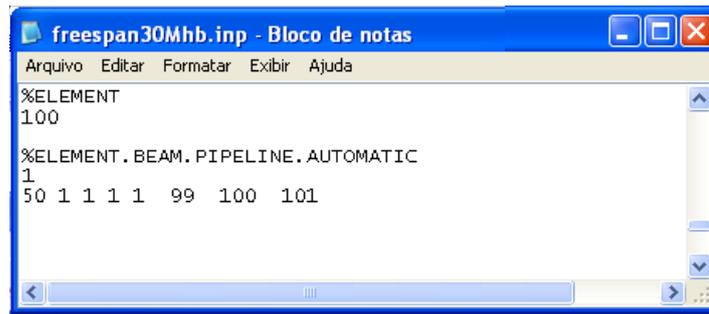


Figura 4.19 –Arquivo *freespan30Mhb.inp* com as informações dos elementos

Definido todos os parâmetros relativos aos elementos só resta definir o tipo de carregamento em que a tubulação será submetida, para inserir essas informações é só acionar o botão indicado pela seta na Figura 4.20, neste exemplo a tubulação esta solicitada apenas por uma força nodal com valor de -500, após isto é só acionar o botão *Add* e para salvar as informação o botão *Store*, os valores podem ser conferidos posteriormente no arquivo *freespan30Mhb.inp*, como mostra a figura 4.21.

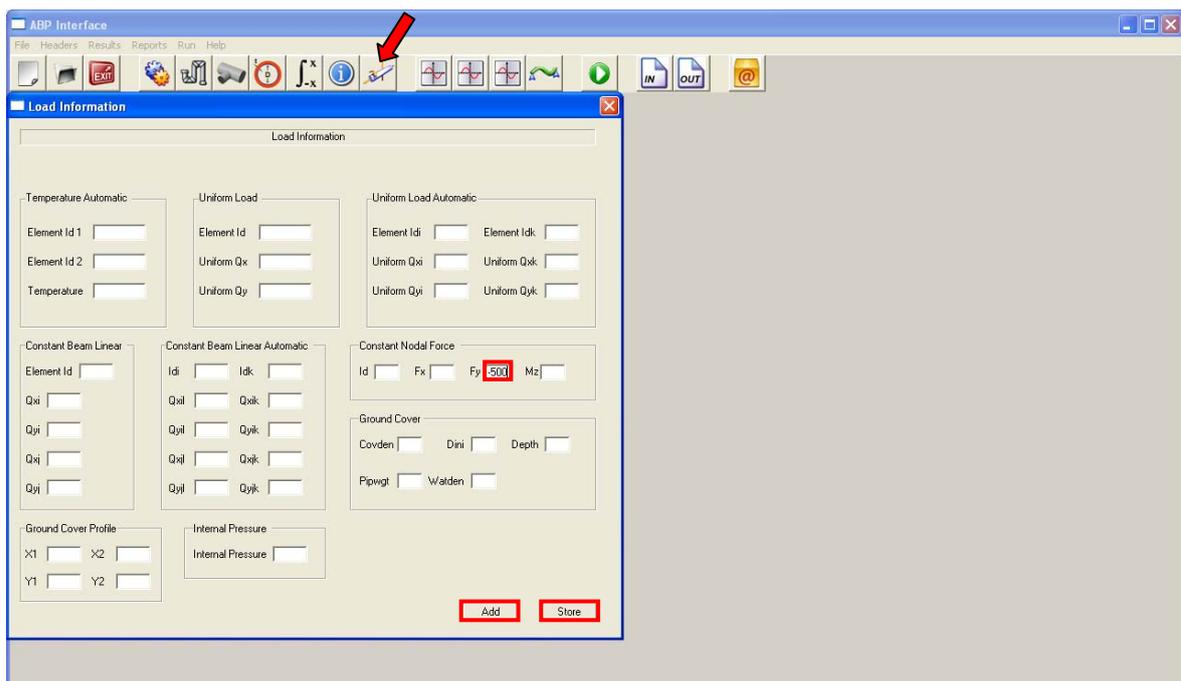


Figura 4.20 – Informações do Carregamento.

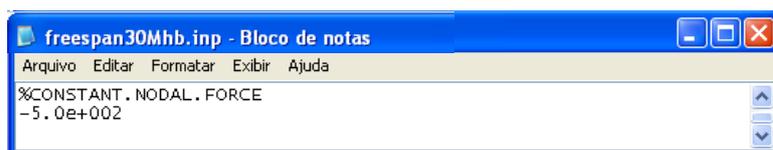


Figura 4.21 – Arquivo *freespan30Mhb.inp* com as informações dos carregamentos

Terminado o processo de entrada de dados e de execução do *ABP* passa-se para a etapa de pós-processamento dos dados, com os gráficos de cortante e deformação em relação ao eixo X, aqui o usuário tem a opção de procurar por valores de cortante e identificar em quais coordenadas do eixo X estes valores se repetem, nas Figuras 4.22 e 4.23, por exemplo, são identificados os valores de momento máximo dado pelo valor -100000 e mostrados na tabela ao lado.

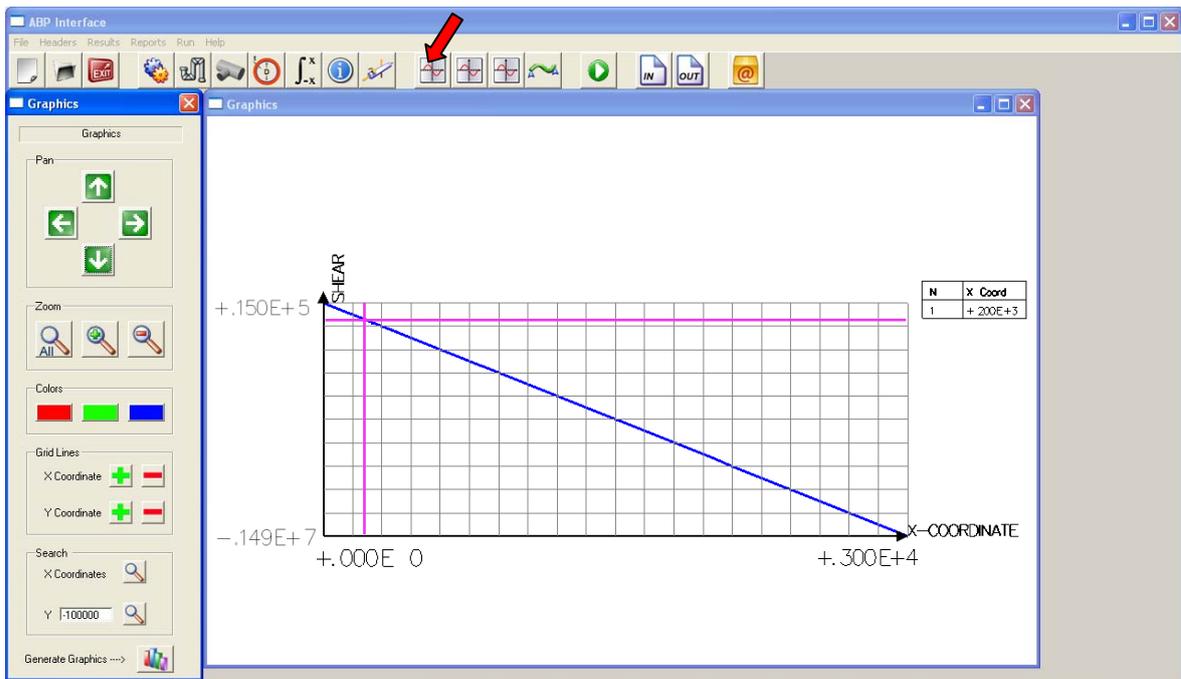


Figura 4.22 – Gráfico Coordenada X - Cortante.

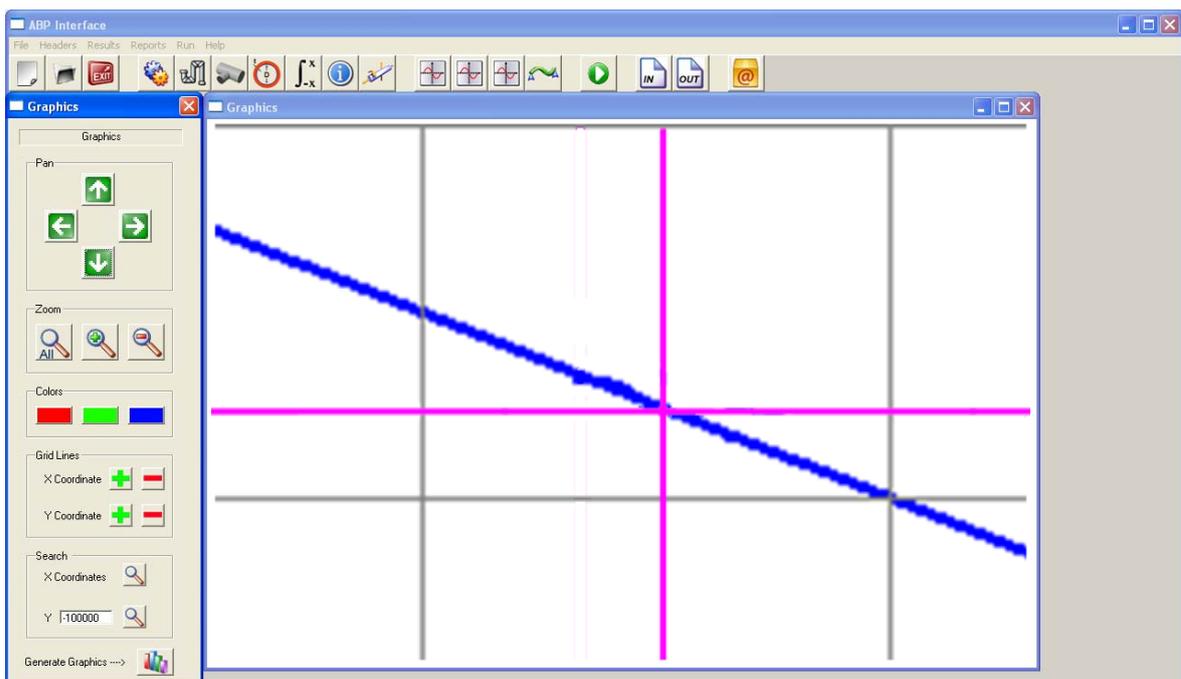


Figura 4.23 – Detalhe da captura da Coordenada X

O mesmo ocorre para a deformação, igual a 2.86×10^{-1} , o procedimento acontece da mesma maneira como no gráfico de cortante, como pode ser visto nas Figuras 4.24 e 4.25.

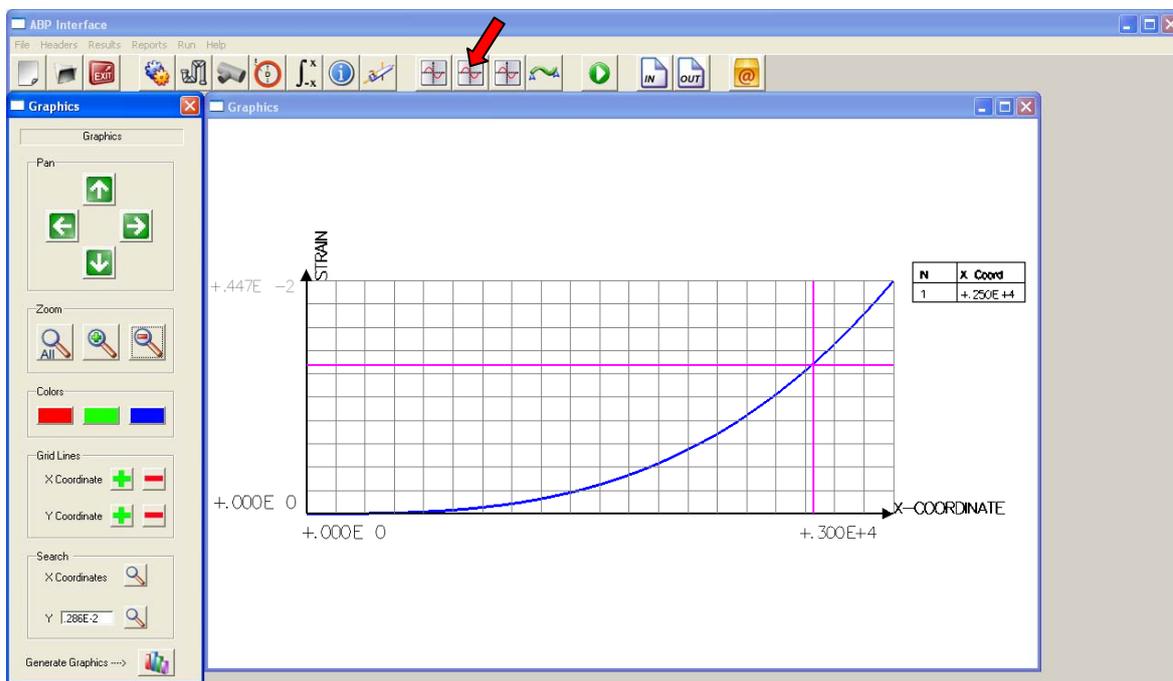


Figura 4.24 – Gráfico Coordenada X - Deformada.

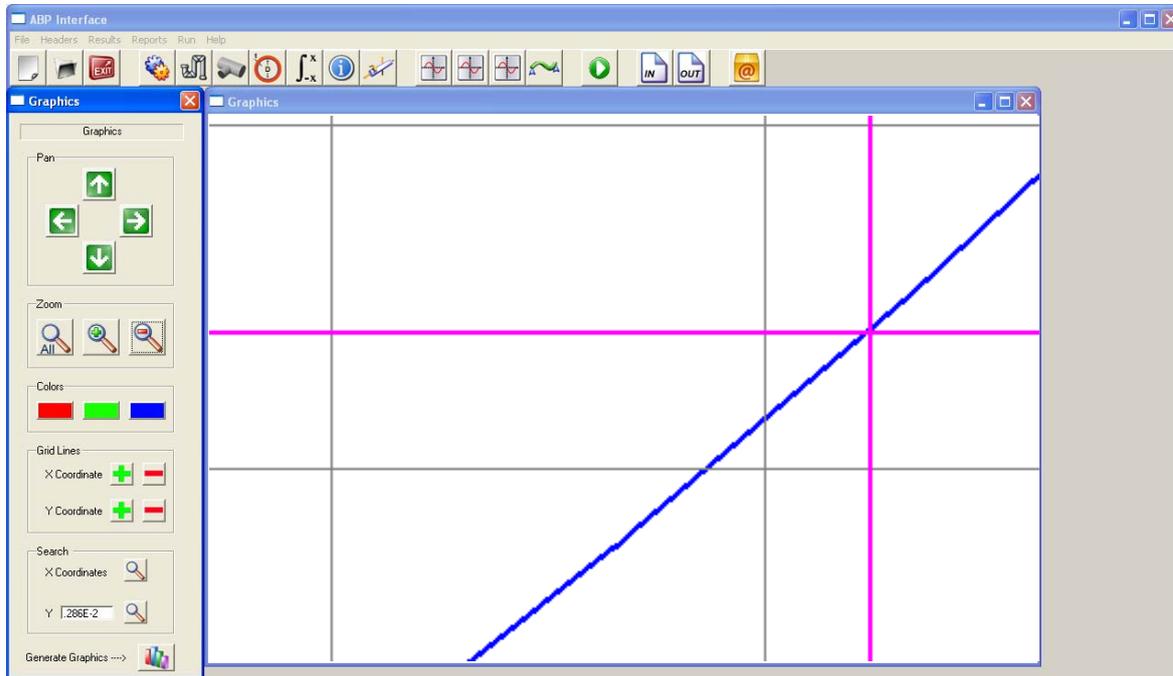


Figura 4.25 – Detalhe da captura da Coordenada X

Por fim temos a última tela do pós-processamento, a qual simula a deformação da tubulação através de uma animação, para abrir a tela da deformada é só acionar o botão indicado pela seta na Figura 4.26, aqui o usuário pode ter opção de aumentar ou diminuir a velocidade da animação assim com o fator de escala ou até pausar a animação.

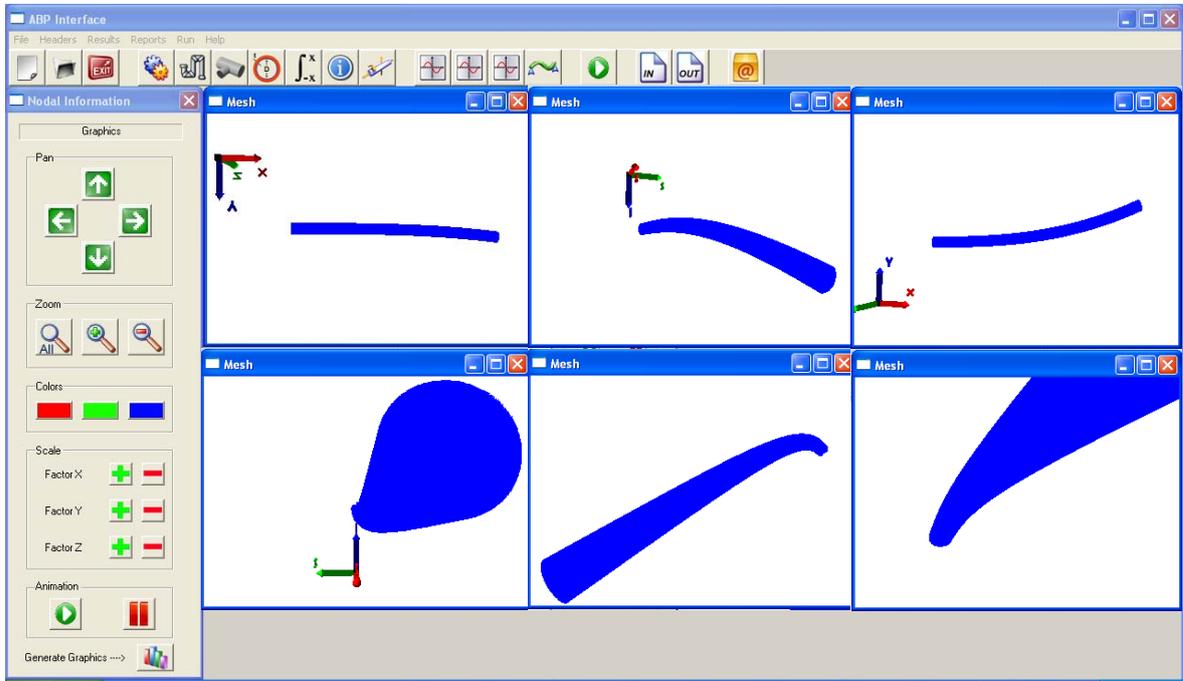


Figura 4.26 – Deformada.

4.2 – EXEMPLO COM PRESSÃO INTERNA

Neste exemplo temos um trecho de tubulação de 115000 cm, impedido de deslocamento em x, y e z conforme figura abaixo, submetido a uma pressão interna e ao peso do terreno sobre a tubulação, como mostra a Figura 4.27.

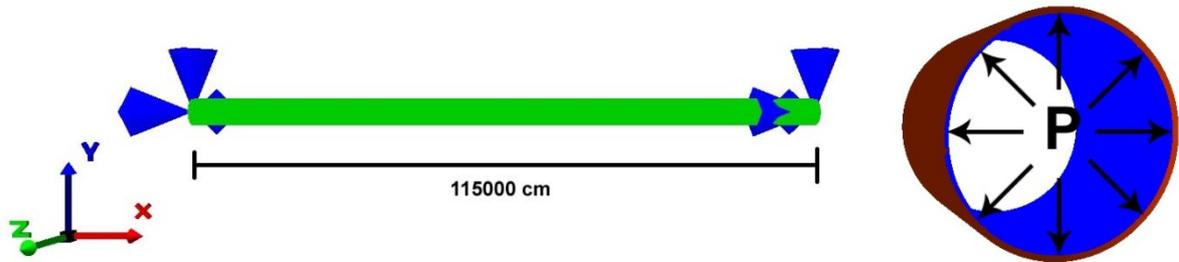


Figura 4.27 – Situação esquemática do 2º exemplo

Nas telas abaixo serão demonstrados todas as telas que fazem parte da análise desde a definição do nome do arquivo passando por todas as telas de entrada de dados: informações gerais, malha, propriedades do material, seção transversal, tipo de integração, informações dos elementos e carregamento, como também as telas do pós-processamento como os gráficos e a deformada.

O primeiro passo é definir o nome do arquivo onde serão armazenadas as informações da análise, acionando botão novo arquivo indicado pela seta na Figura 4.28 irá abrir a tela com o título *File Information*, com o campo *File Name* a ser preenchido com o nome escolhido pelo usuário com a extensão *.inp*, logo após é só acionar o botão *create* e o arquivo será criado como mostra a figura 4.29.

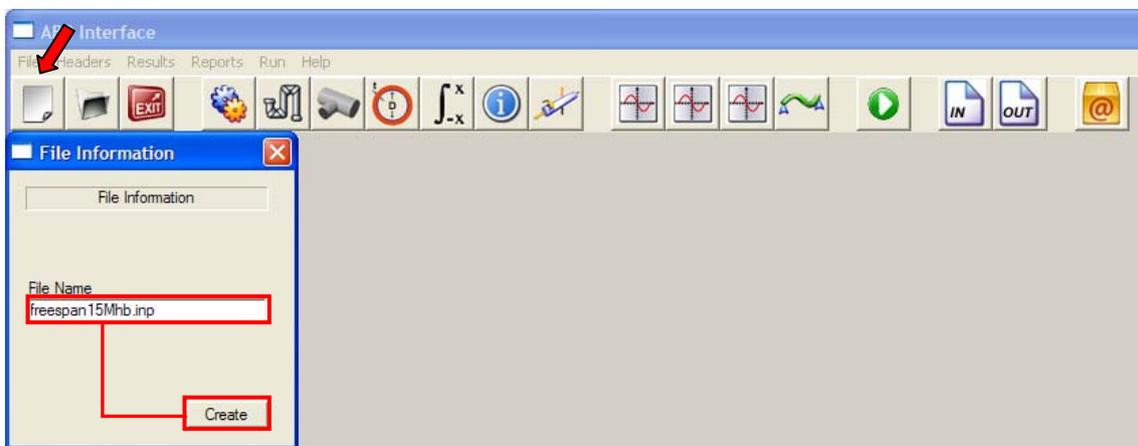


Figura 4.28 – Nome do arquivo

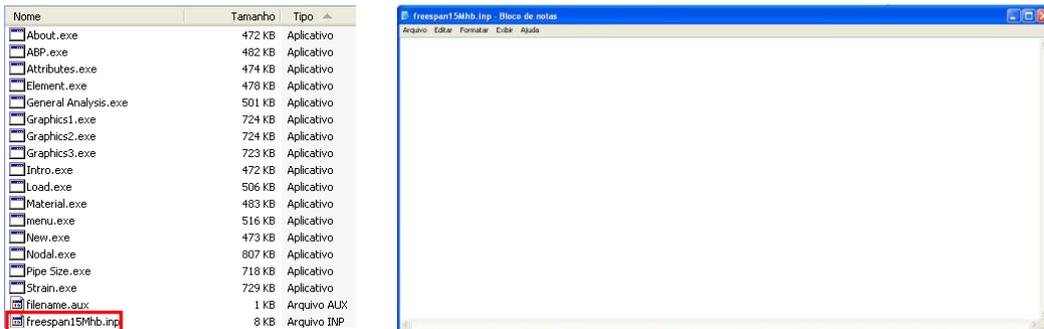


Figura 4.29 – Criação do arquivo *freespan15Mhb.inp*

Passando agora para o preenchimento dos dados relativos à análise do exemplo em questão, para isso deve-se clicar no botão referente as informações gerais da análise, indicado pela seta na Figura 4.30, o primeiro valor a ser preenchido é justamente o cabeçalho do arquivo que não necessariamente é igual ao nome do arquivo de entrada, neste caso Análise térmica, a seguir os dados de *Critical Strain* com os valores de 65 e 20, *Load Steps* com o valor de 100, *Maximum Iteration* de interações também com o valor de 100, *Print Steps* com o valor de 1, *Analysis* com *Thermal*, *Execution* com a opção *Execution* marcada, *Tolerance Displacement* e *Tolerance Forces* com o valor de 0.05. por fim é só acionar o botão *Store* e todas essas informações serão armazenadas no arquivo *freespan15Mhb.inp* como mostra a Figura 4.31.

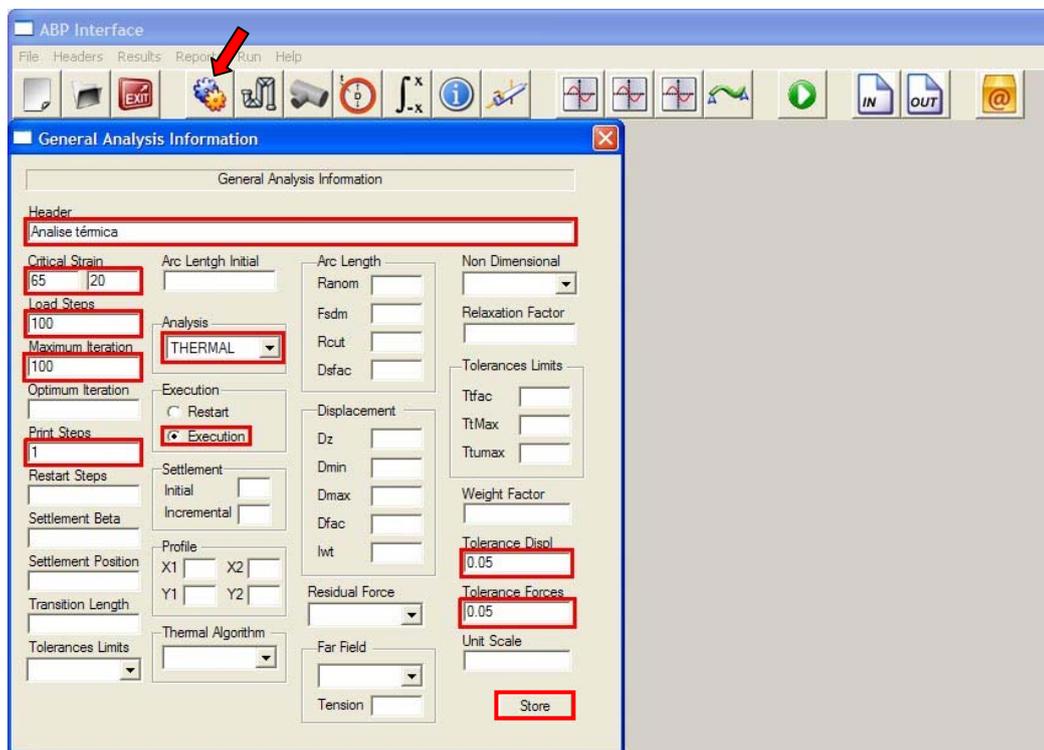


Figura 4.30 – Informações Gerais

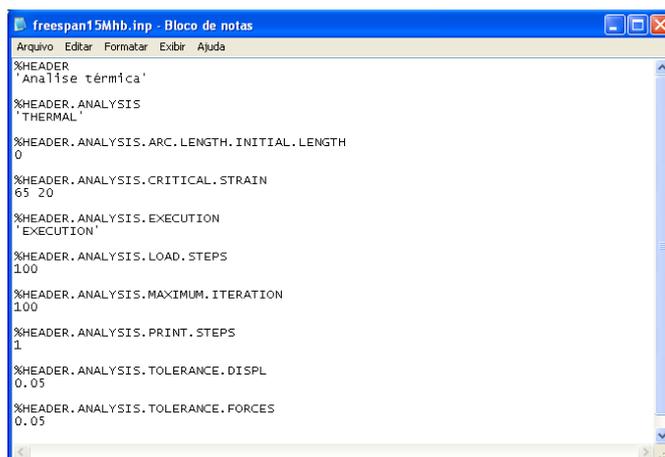


Figura 4.31 – Arquivo *freespan15Mhb.inp* com as informações gerais

O próximo passo é definir a malha de elementos da tubulação, para isso deve se acionar o botão informações nodais indicado pela seta na figura 4.32 e com mais detalhes nas Figuras 4.33 e 4.34, o qual irá abrir a tela para que sejam inseridos os parâmetros da malha. O primeiro a ser preenchido é o número de elementos no caso 232 elementos, logo após devem ser informados as coordenadas em X, Y e Z com isto a tubulação já pode ser criada, porém falta as informações relativas as condições de suporte que neste exemplo são restringidas nas três direções nos nós 1e 232. Para finalizar temos os botões *Mesh* o qual gera a visualização 3D da malha e o botão *Save* que escreve os dados, já formatados, referentes à malha no arquivo *freespan15Mhb.inp* como mostra a Figura 4.35.

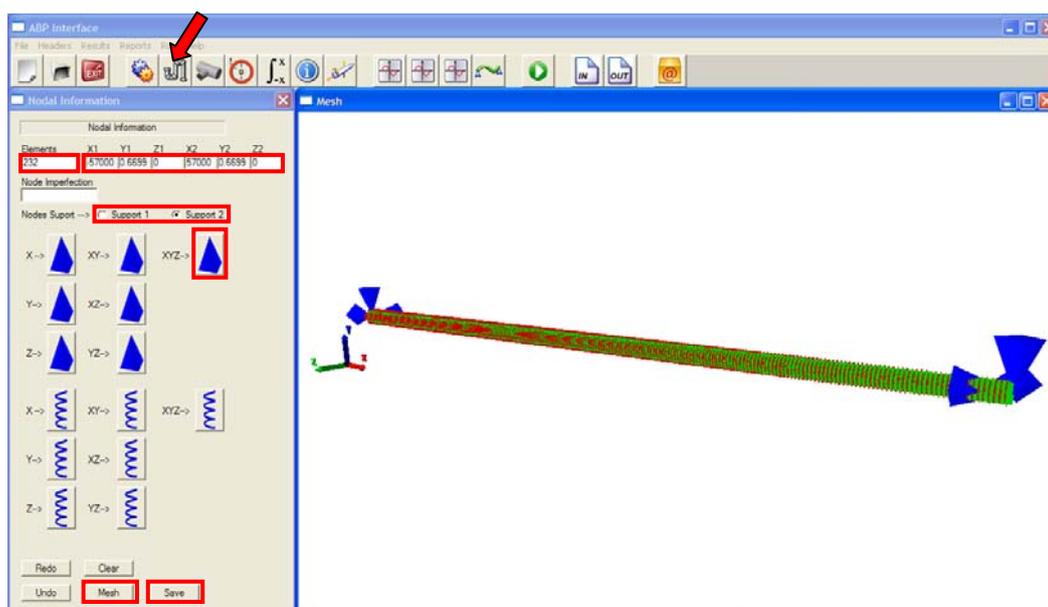


Figura 4.32 – Malha da tubulação.

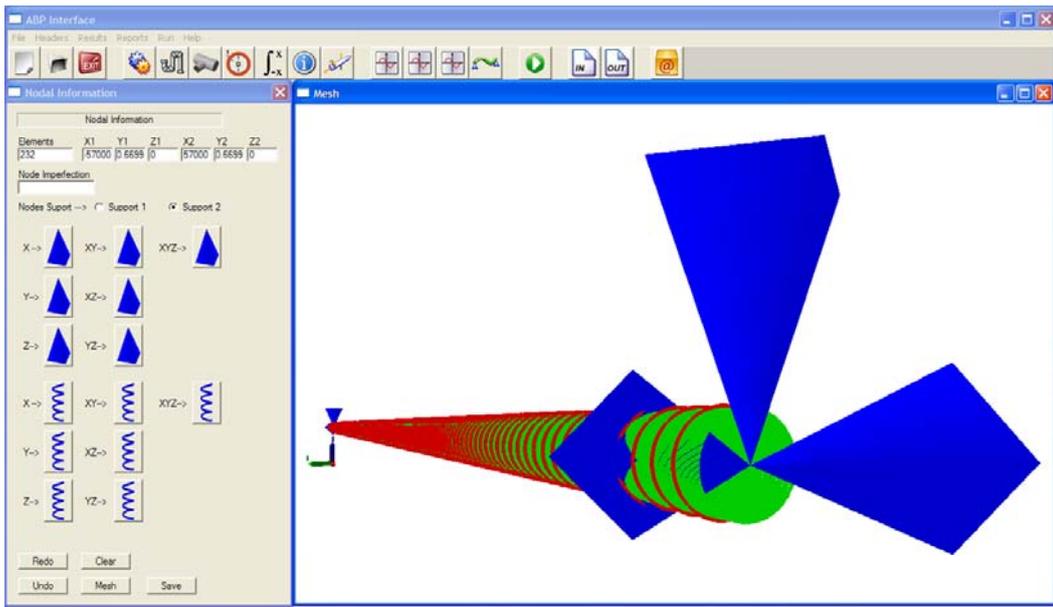


Figura 4.33 – Detalhe do segundo apoio da tubulação

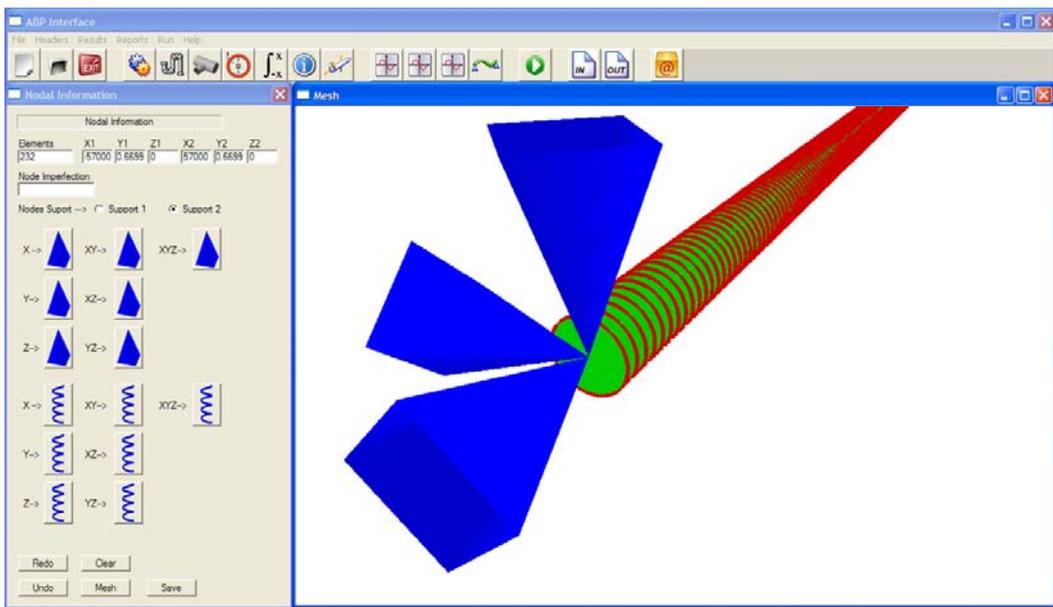


Figura 4.34 – Detalhe do primeiro apoio visto de dentro da tubulação

```

freespan15Mhb.inp - Bloco de notas
Arquivo  Editar  Formatar  Exibir  Ajuda
%NODE
233
%NODE,COORD
233
1 -57500.0000 -0.6699 0.0000
2 -57004.3103 -0.6699 0.0000
3 -56508.6207 -0.6699 0.0000
4 -56012.9310 -0.6699 0.0000
5 -55517.2413 -0.6699 0.0000
6 -55021.5518 -0.6699 0.0000
7 -54525.8621 -0.6699 0.0000
8 -54030.1724 -0.6699 0.0000
9 -53534.4827 -0.6699 0.0000
10 -53038.7930 -0.6699 0.0000
11 -52543.1035 -0.6699 0.0000
12 -52047.4141 -0.6699 0.0000
13 -51551.7246 -0.6699 0.0000
14 -51056.0352 -0.6699 0.0000
15 -50560.3457 -0.6699 0.0000
16 -30064.6563 -0.6699 0.0000
17 -49568.9668 -0.6699 0.0000
18 -49073.2773 -0.6699 0.0000
19 -48577.5879 -0.6699 0.0000
20 -48081.8984 -0.6699 0.0000

```

Figura 4.35 – Arquivo *freespan15Mhb.inp* com as informações nodais

Definido a malha e as condições de apoio da tubulação o próximo passo consiste nas informações sobre o tipo de material, para inserir essas informações basta acionar o botão de informações do material como indicado pela seta na Figura 4.36, onde para o *Material ID* que recebe o valor 1 temos os valores de *Strain* igual a 0.0042 e *Stress* de 465, já para os parâmetros de *Material Interface* temos *Kb* igual a 2.0834, *Fyb* igual a 6.653, *Kl* igual a 0.8815, *Fyl* igual a 1.763, *Ku* e *Fyu* iguais a 0, já no que se refere ao *Upflit Spring* temos *Displacement* igual a 1 e *Force* igual a 3 e para finalizar temos os parâmetros *Property* com os valores de *Hardening* igual a 1 e *Poisson* igual a 0.3. após esse procedimento é só acionar o botão *Add* e caso não tenha outro material a ser cadastrado é só acionar o botão *Save*, os valores podem ser conferidos posteriormente no arquivo *freespan15Mhb.inp*, como mostra a figura 4.37.

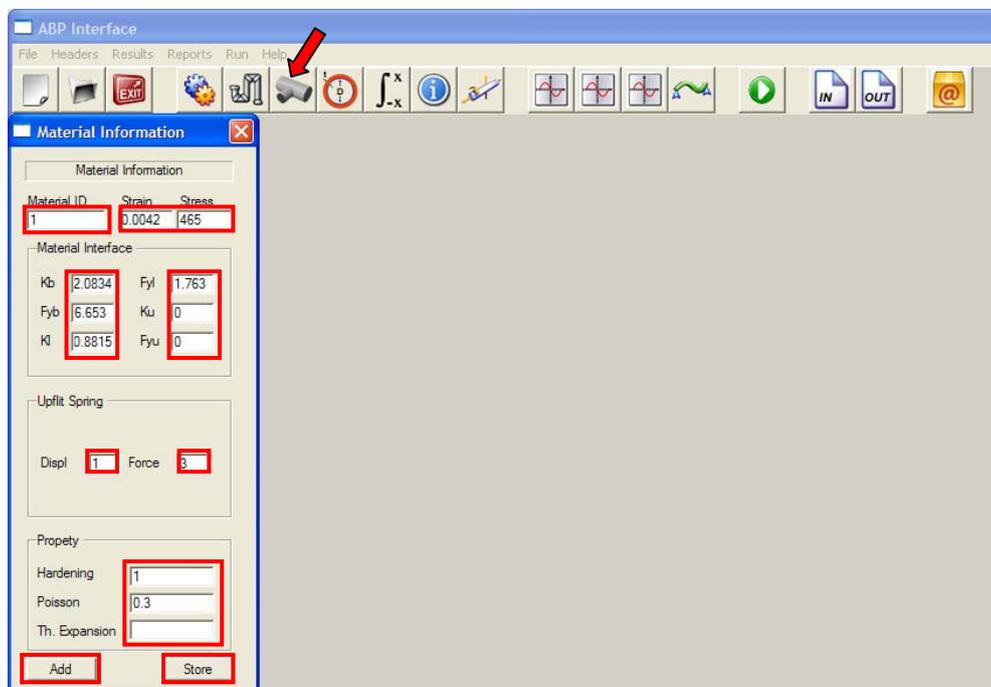


Figura 4.36 – Propriedades dos Materiais



Figura 4.37 – Arquivo *freespan15Mhb.inp* com as informações do material

O próximo passo é a definição da seção transversal da tubulação, para inserir essas informações basta acionar o botão indicado pela seta na Figura 4.38 e com mais detalhes nas Figura 4.39 e 4.40, esta etapa requer apenas que o usuário informe o diâmetro interno da tubulação e a espessura com os valores de 323.85 e 19.05 respectivamente, nos campos *Diameter* e *Thickness*, após esta etapa basta acionar os botões *Add* para visualizar como ficou a seção transversal da tubulação e o botão *Store* para armazenar os dados, os valores podem ser conferidos posteriormente no arquivo *freespan15Mhb.inp*, como mostra a figura 4.41.

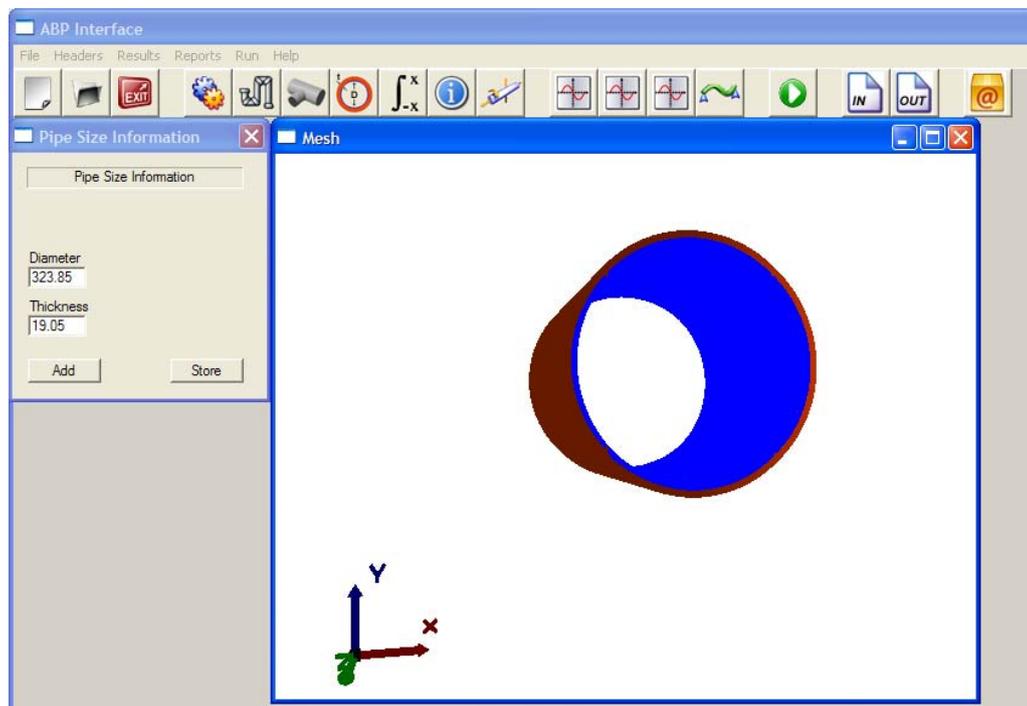


Figura 4.38 – Diâmetro e Espessura da Tubulação

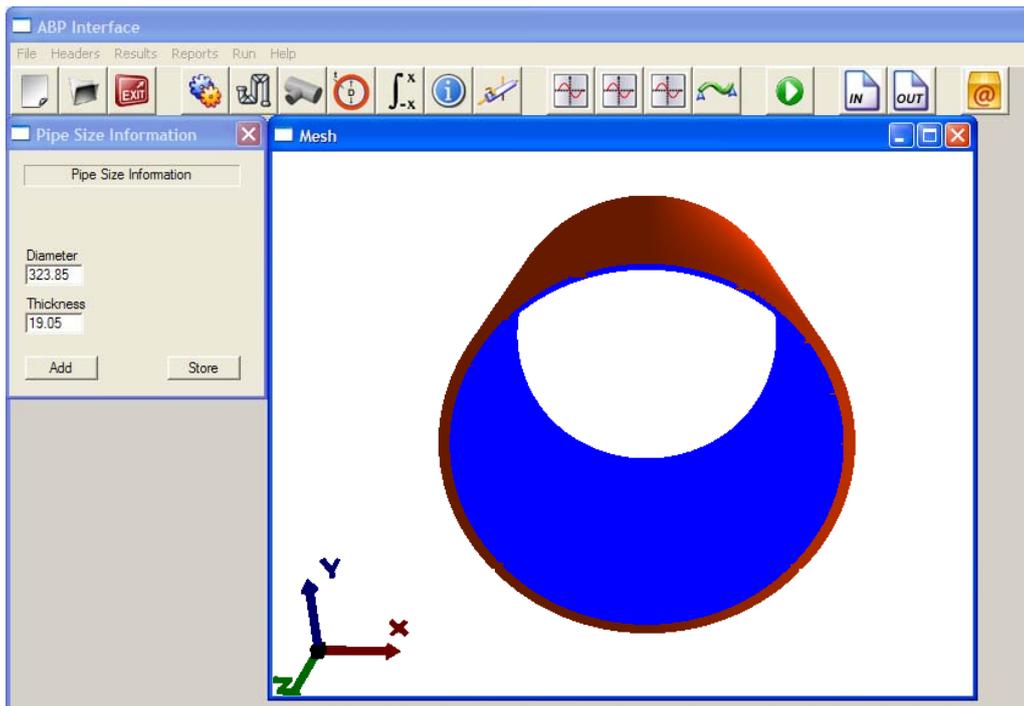


Figura 4.39 – Detalhe com zoom da seção transversal.

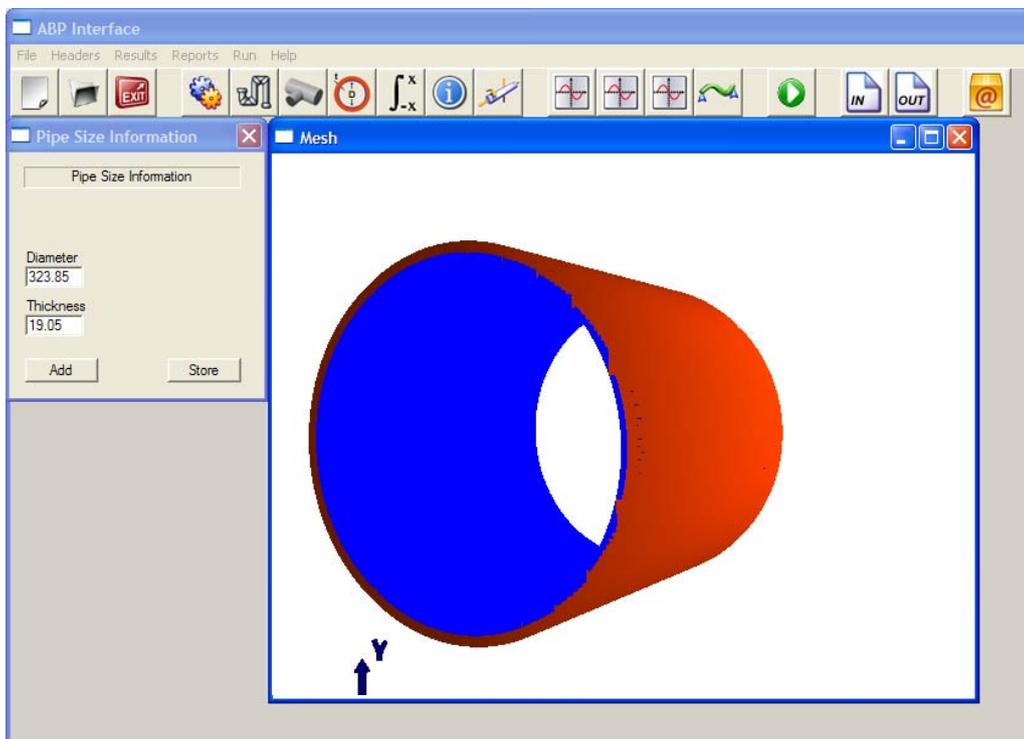


Figura 4.40 – Detalhe da parede da tubulação

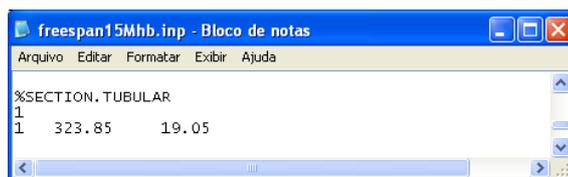


Figura 4.41 – Arquivo *freespan15Mhb.inp* com as informações da seção transversal

Seguindo com a montagem do arquivo de entrada temos a tela de informações dos atributos do elemento, nesta tela é definido como será feita a integração tanto na seção transversal quanto na longitudinal, para inserir esta informações basta acionar o botão indicado pela seta na Figura 4.42, neste caso os campos *Cross Section* e *Order Longitudinal* receberam os valores 1 e 6 respectivamente, após isto é só acionar o botão *Add* e para salvar as informação o botão *Store*, os valores podem ser conferidos posteriormente no arquivo *freespan15Mhb.inp*, como mostra a figura 4.43.

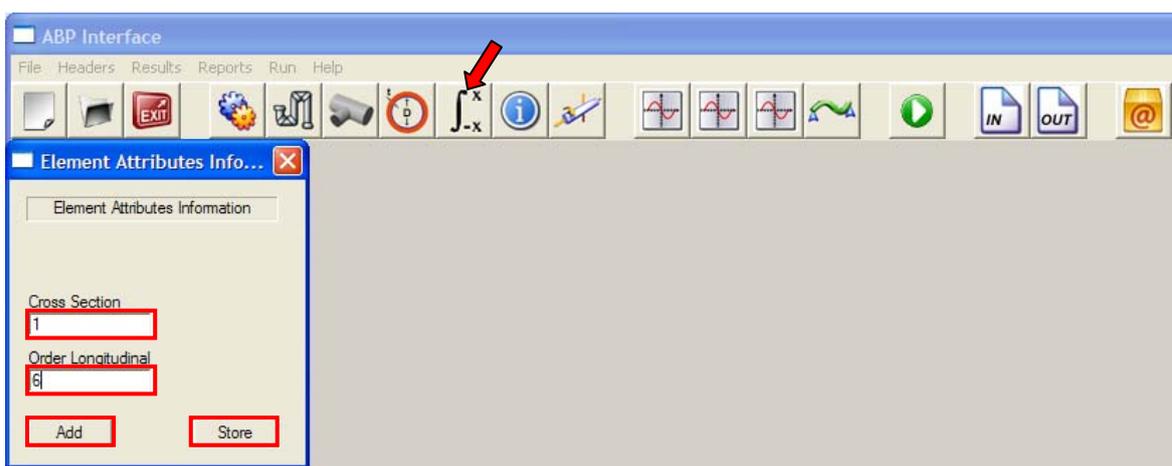


Figura 4.42 – Parâmetros de integração

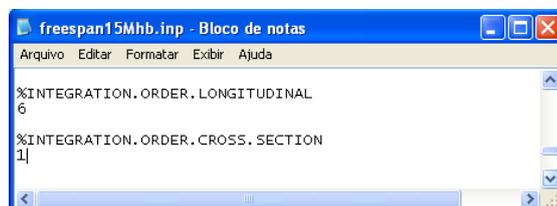


Figura 4.43 – Arquivo *freespan15Mhb.inp* com as informações de integração

O próximo passo nada mais é do que a associação dos dados anteriormente inseridos com a identificação do elemento, ou seja, é informar para o programa que o referido elemento tem uma determinada seção transversal, de um determinado material, com um determinado

coeficiente de mola para o solo entre os nós I, J e K, para inserir essas informações é só acionar o botão indicado pela seta na Figura 4.44, neste caso com os valores 50 para *Id*, 1 para *Plot Flag*, *Section Id*, *Material Id* e *Soil Spring*, 99 para *Node I*, 100 para *Node J* e 101 para *Node K*, após isto é só acionar o botão *Add* e para salvar as informação o botão *Store*, os valores podem ser conferidos posteriormente no arquivo *freespan15Mhb.inp*, como mostra a figura 4.45.

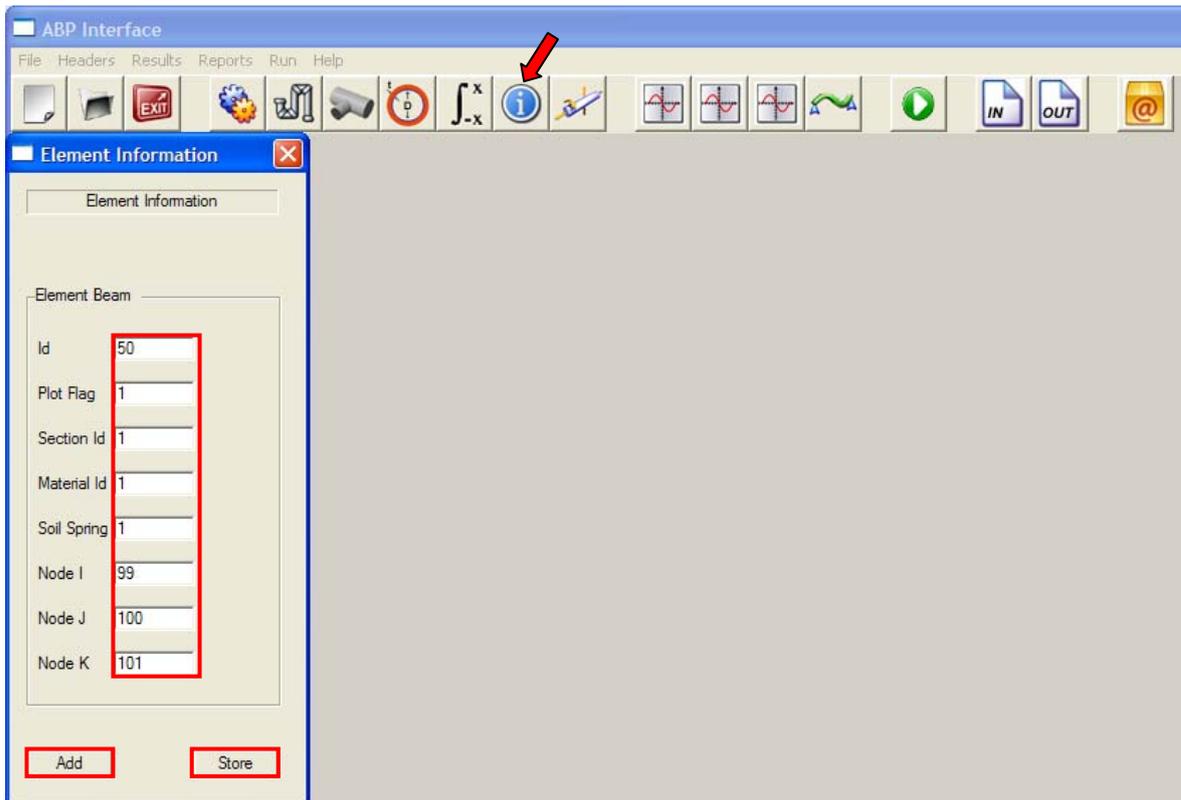


Figura 4.44 – Informações dos elementos

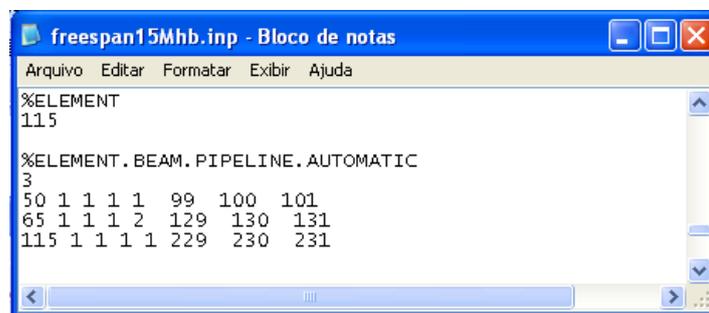


Figura 4.45 –Arquivo *freespan15Mhb.inp* com as informações dos elementos

Definido todos os parâmetros relativos aos elementos só resta definir o tipo de carregamento em que a tubulação será submetida, para inserir essas informações é só acionar o botão indicado pela seta na Figura 4.46, neste exemplo a tubulação esta solicitada apenas pela pressão interna com valor de 10.4, os demais parâmetros são relativos ao solo onde se encontra tubulação, após isto é só acionar o botão *Add* e para salvar as informações o botão *Store*, os valores podem ser conferidos posteriormente no arquivo *freespan15Mhb.inp*, como mostra a figura 4.47.

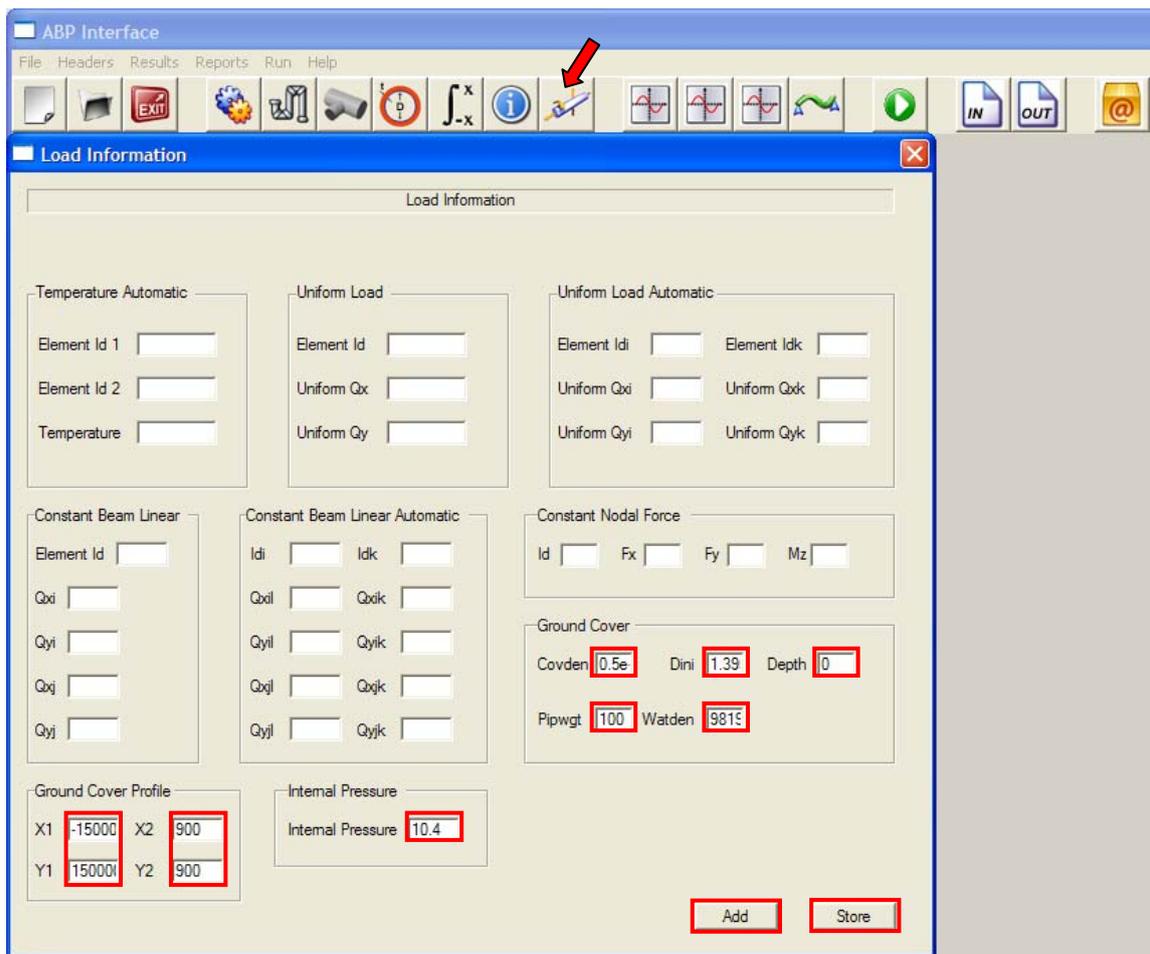


Figura 4.46 – Informações do Carregamento.



Figura 4.47 –Arquivo *freespan15Mhb.inp* com as informações dos carregamentos

Terminado o processo de entrada de dados e de execução do *ABP* passa-se para a etapa de pós-processamento dos dados, com os gráficos de momento, curvatura e força axial em relação ao eixo X, aqui o usuário tem a opção de procurar por valores de momento e identificar em quais coordenadas do eixo X estes valores se repetem, na Figura 4.48 e 4.49, por exemplo, são identificados os valores de momento máximo dado pelo valor 2.32×10^8 e mostrados na tabela ao lado, como existem muitos valores consecutivamente na coordenada X com o momento máximo o usuário pode dar um *Zoom* através dos botões de controle e aproximar a seleção dos valores.

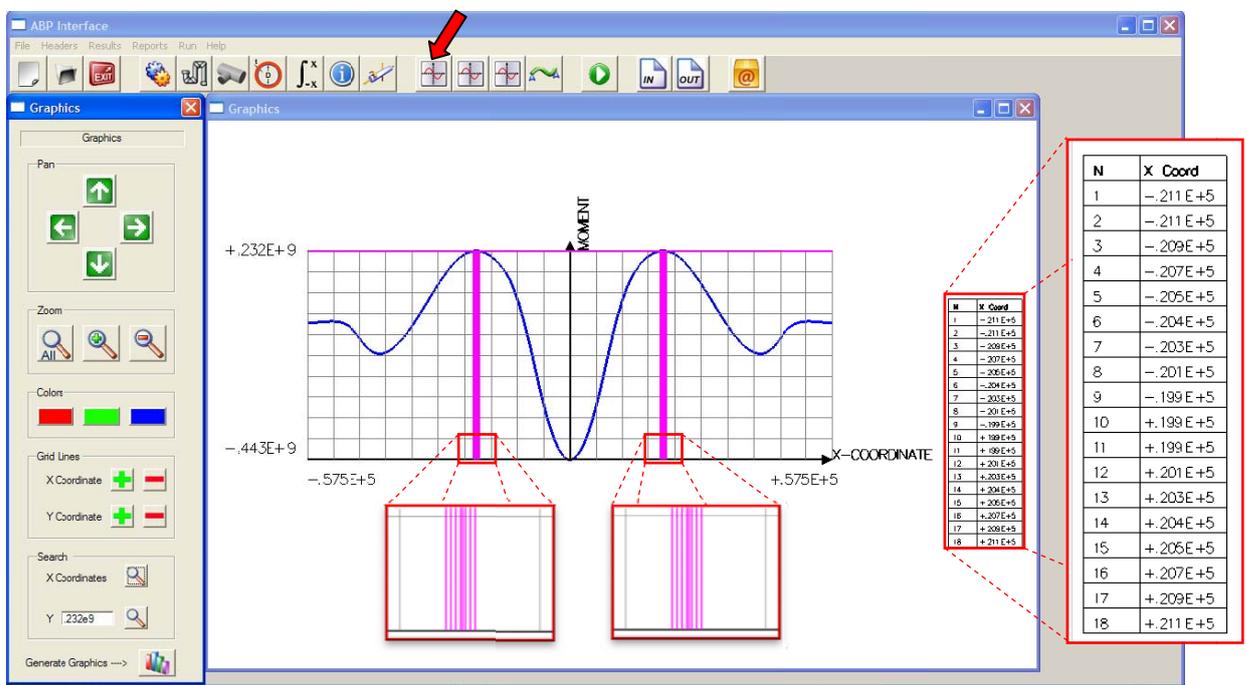


Figura 4.48 – Gráfico Coordenada X - Momento.

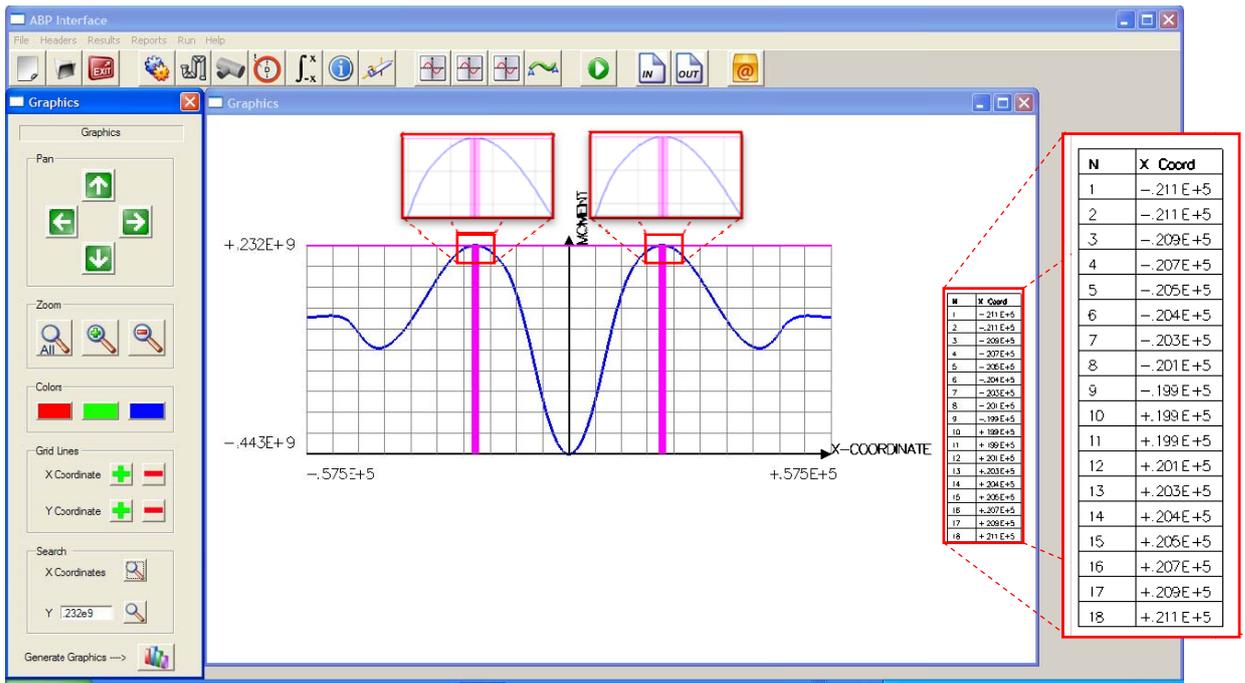


Figura 4.49 – Gráfico Coordenada X - Momento.

O mesmo ocorre para a curvatura e para a força axial, porém no outros dois casos é verificado os valores da coordenada X para curvatura igual a zero e força axial igual 2.7×10^2 , o procedimento acontece da mesma maneira como no gráfico de momento, onde pode ser visto nas Figuras 4.50, 4.51, 4.52 e 4.53, e o gráfico das deformações no topo e na base da tubulação conforme Figura 4.54.

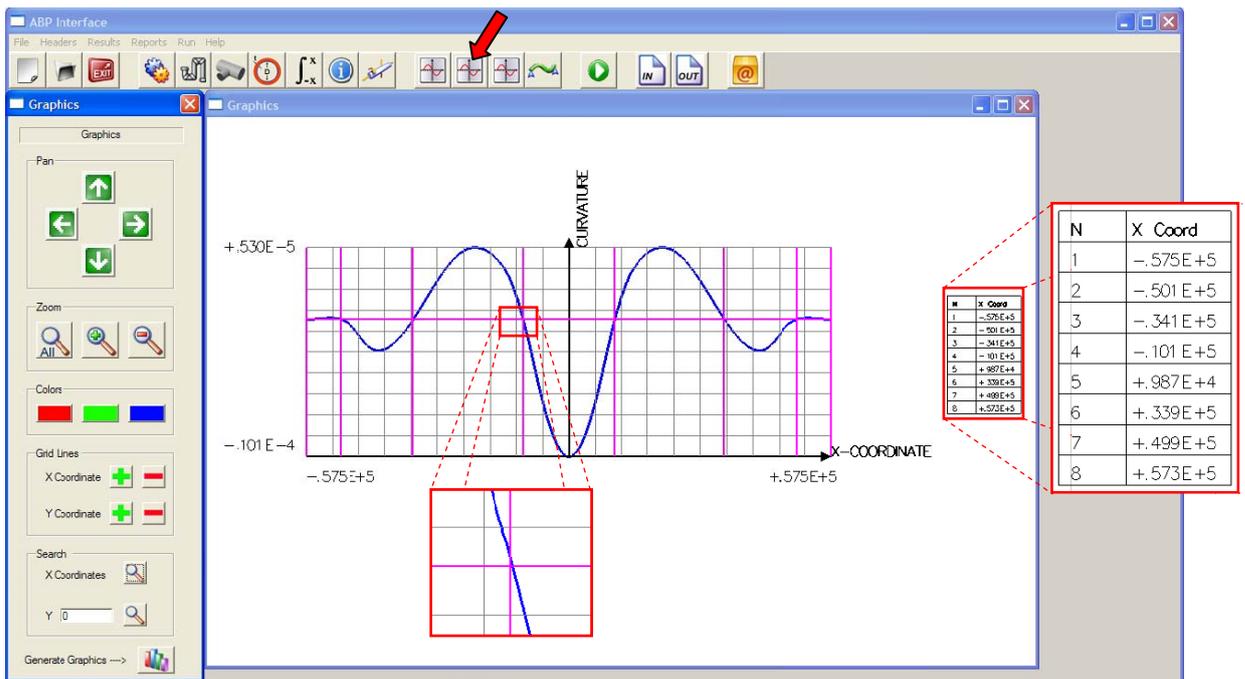


Figura 4.50 – Gráfico Coordenada X - Curvatura.

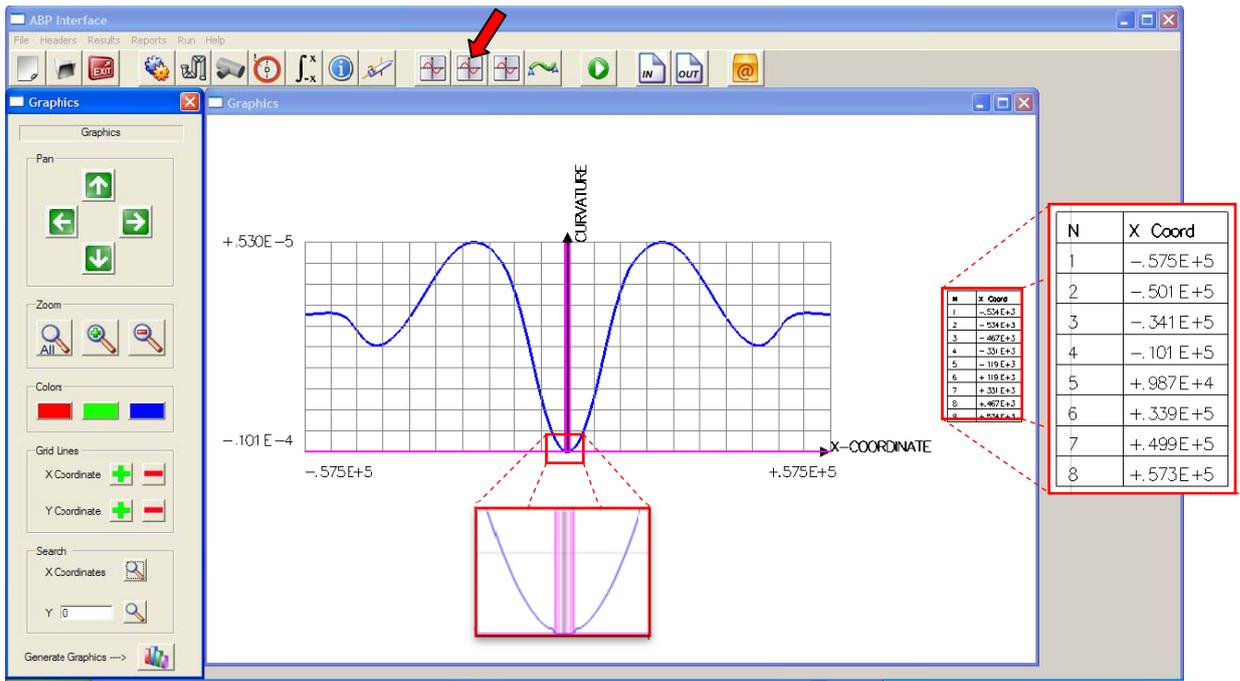


Figura 4.51 – Gráfico Coordenada X - Curvatura.

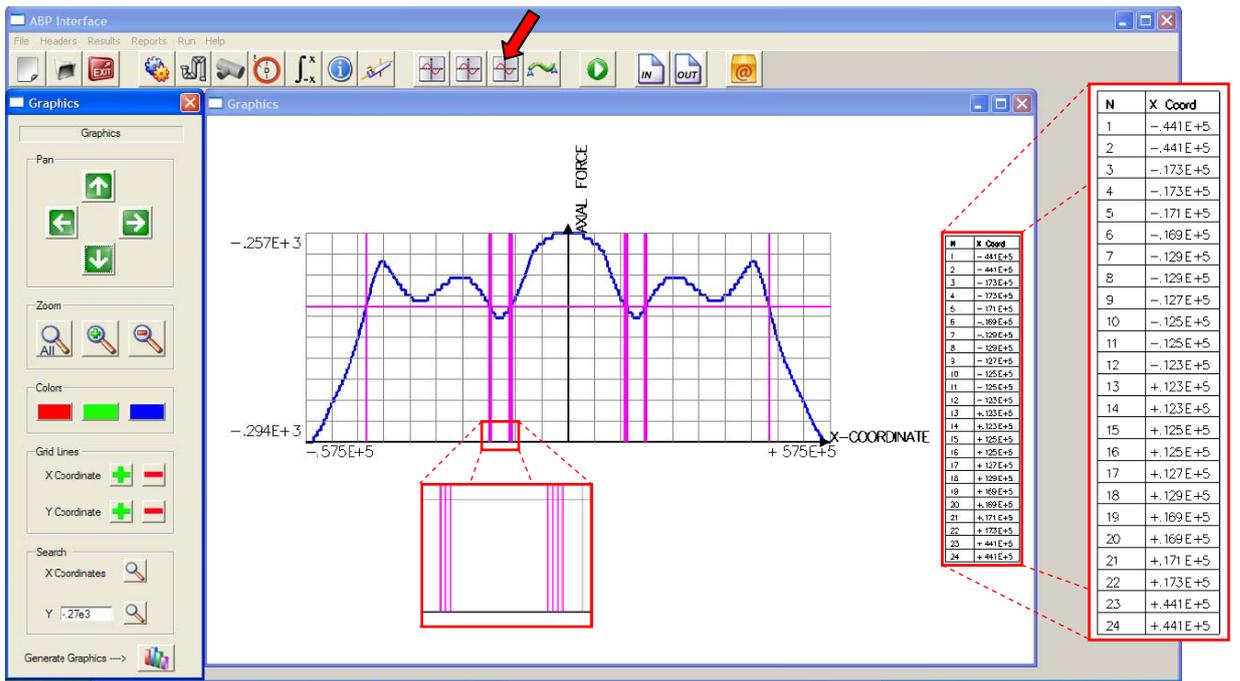


Figura 4.52 – Gráfico Coordenada X – Força Axial.

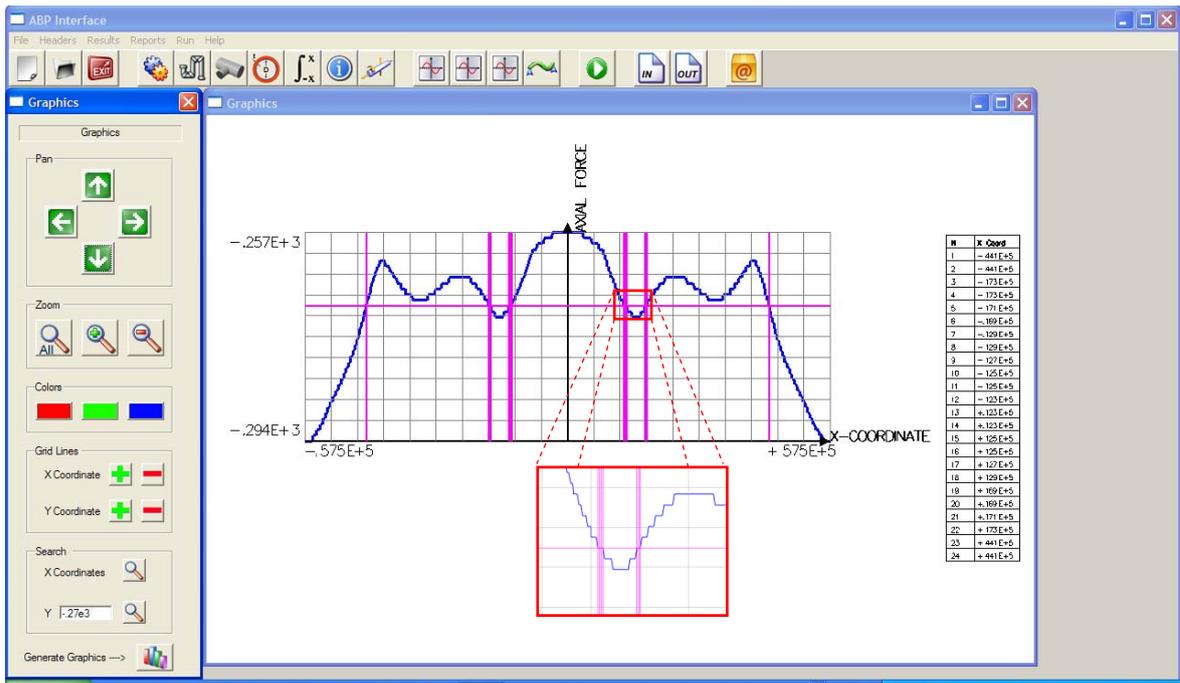


Figura 4.53 – Gráfico Coordenada X – Força Axial.

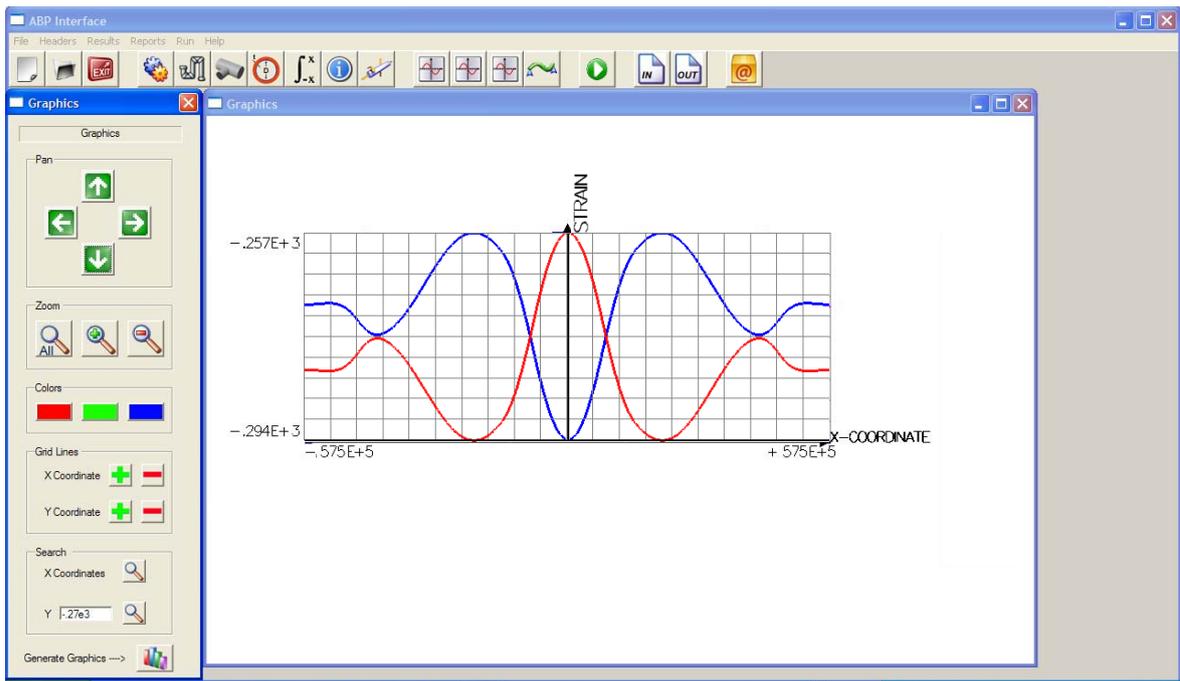


Figura 4.54 – Gráfico Coordenada X – Top Strain & Bottom Strain.

Por fim temos a última tela do pós-processamento, a qual simula a deformação da tubulação através de uma animação, para abrir a tela da deformada é só acionar o botão indicado pela seta na Figura 4.55, aqui o usuário pode ter opção de aumentar ou diminuir a velocidade da animação assim com o fator de escala ou até pausar a animação.

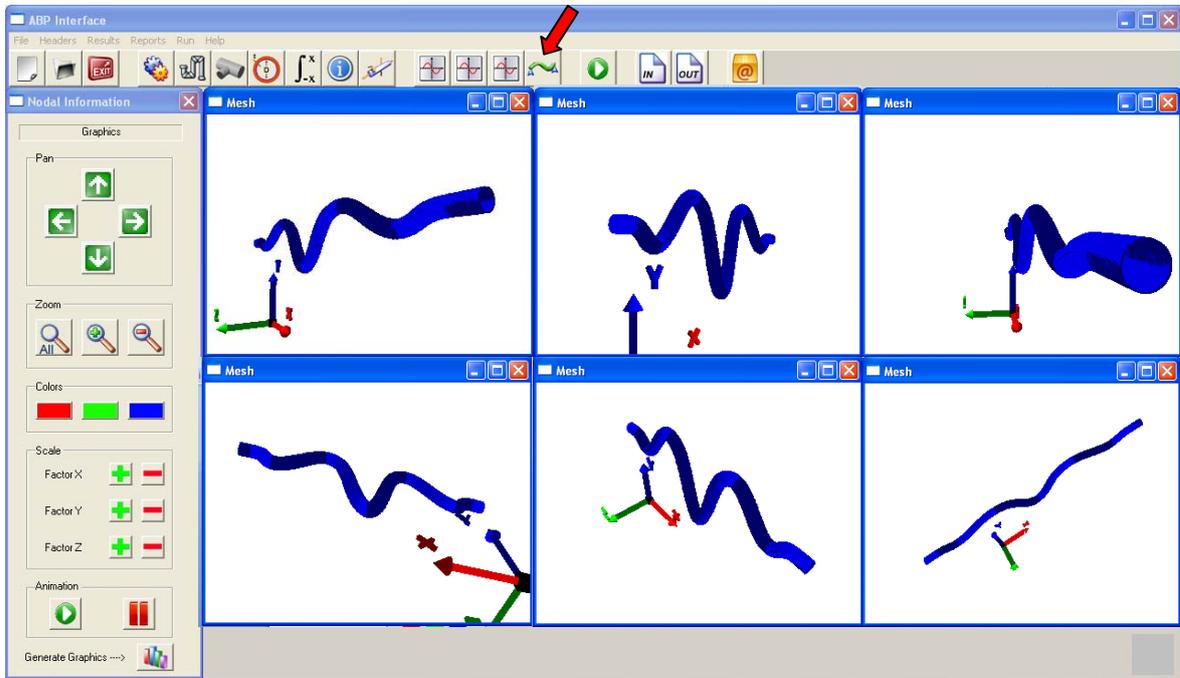


Figura 4.55 – Deformada.

5 – CONCLUSÕES E SUGESTÕES PARA TRABALHOS FUTUROS

A linguagem C++ se mostrou a mais adequada para o desenvolvimento do trabalho, isto porque ela permitiu acesso a qualquer dispositivo de *hardware* como teclado e *mouse*, por exemplo, muito utilizados como meios de navegação nas telas que contém gráficos, nas telas de visualização em 3D, por exemplo, o botão direito do *mouse* pressionado permitia movimentos de rotação dos objetos, já com o direito habilitava a opção de *zoom*, no caso do teclado as setas foram usadas para movimentos de *zoom* e as teclas *home* e *end* para movimentos de rotação horários e anti-horários e por fim *pageup* e *pagedown* para *zoom*.

Outro ponto forte da linguagem C++ é a possibilidade de durante a execução do programa de interface, permitir que seja chamado outro programa, neste caso o ABP, tendo ainda a possibilidade do envio de parâmetros desde que este executável esteja preparado para receber estes novos parâmetros.

O compilador *DEV-C++* não foi o compilador adequado para a criação da interface, devido principalmente por não oferecer uma biblioteca de objetos pré-definidos, como *buttons*, *edits*, *static* entre outros, gerando uma quantidade relativamente grande de parâmetros, cerca de 15 em média, para criação de objetos simples onde 4 desses parâmetros são respectivamente a posição em x e y e o tamanho em x e y, acarretando assim um tempo muito grande para a definição dos tamanhos e posições dos objetos, tomando como exemplo a tela de *Material Information* que contém 41 objetos e gerou 988 linhas de comando pra ser criada, compiladores como *Borland C++* seriam mais adequados para o desenvolvimento deste trabalho.

No que diz respeito ao uso das bibliotecas é importante destacar que a biblioteca gráfica *OpenGL* proporcionou uma grande qualidade gráfica aos elementos e se mostrou de fácil entendimento com comandos bastante simples porém extremamente eficazes, tomando por exemplo as funções para construção de primitivas gráficas como pontos, linhas, triângulos e quadriláteros e as funções para manipulação dos objetos como rotação, translação e escala.

A grande vantagem deste programa de interface é a capacidade de acoplamento com o executável do ABP, não sendo necessária a alteração do código fonte do programa, escrito em Fortran 77, sendo a interface gráfica um programa totalmente independente, a qual utiliza apenas a formatação dos dados de entrada do ABP para a criação do arquivo de entrada e o arquivo de saída para a geração de todo o pós-processamento.

Já a biblioteca *Win32* tem uma quantidade muito grande de objetos e mensagens, todos os utilizados no sistema operacional *Windows*, como o compilador utilizado não tinha uma biblioteca de elementos do mesmo tipo utilizados no *Windows*, o uso desta biblioteca foi a escolha mais adequada e que atendeu de maneira bastante satisfatória, também devido ao material encontrado no site *MSDN Library* que além de completo é bastante detalhado.

Apesar da capacidade do C++ ter em executar programas em tempo de execução, o ideal é que todos os parâmetros fossem passados diretamente ao programa fonte do ABP, escrito em Fortran 77, eliminando a geração de arquivos de dados no formato *txt* tanto para o pré-processamento quanto para o pós-processamento, diminuindo assim o tempo de execução do programa.

Visto as considerações acima fica como sugestões para a continuidade deste tema a adaptação da interface para outros sistemas operacionais, como Linux por exemplo. A utilização de outros compiladores em C++, que tenham objetos pré-definidos e conseqüentemente possibilitem a construção das telas de maneira mais prática e rápida, a compatibilização da interface com o programa fonte do ABP sem a utilização de arquivos de dados em *txt* e por fim a elaboração de mais elementos gráficos que facilitem ainda mais a análise do usuário.

REFERÊNCIAS BIBLIOGRÁFICAS

ALTA PIPE GROUP. Página oficial do Alta Pipe Group. Disponível em: <<http://www.altapipe.ualberta.ca/>> Acesso em: 10 de Janeiro de 2006

AZEVEDO, E; CONCI, A. Computação Gráfica Teoria Prática. Elsevier Ltda. Rio de Janeiro, 2003.

BJARNE STROUSTRUP'S HOMEPAGE. Página do professor Bjarne Stroustrup's. Disponível em: <<http://www.research.att.com/~bs/homepage.html>> Acesso em: 5 de Março de 2006

BOENTE, A. Aprendendo a Programar em C do Básico ao Avançado. Ed. Brasport. Rio de Janeiro, 2003

COHEN, M; MANSSOUR I. H. OpenGL Uma Abordagem Prática e Objetiva. Ed. Novatec Ltda. São Paulo, 2006.

COSTA, E. M. M. Animação no PC baseada em C para Windows. Ed. Alta Books Ltda. Rio de Janeiro, 2005.

COSTA, E. M. M. Programação em C para Windows. Ed. Érica Ltda. São Paulo, 2004.

CPLUSPLUS.COM - THE C++ RESOURCES NETWORK. Página com descrição da biblioteca padrão do C++. Disponível em: <<http://www.cplusplus.com/>>. Acesso em 25 de Janeiro de 2006

DAHL, O. J; MYRHAUG, B; NIGAARD, K. SIMULA Common Base Language. Norwegian Computing Center S-22. Norway, 1970

DAHL, O. J; HOARE, C. A. R. Hierarchical Program Construction in Structured Programming. Academic Press. New York, 1972

HUBBARD, J. R. Teoria e problemas de programação em C++. Trad. Edson Furmankiewicz. 2.ed. Bookman. Porto Alegre, 2003.

JOHN R. HUBBARD PROFESSOR OF MATHEMATICS AND COMPUTER SCIENCE. Página do professor John Hubbard. Disponível em: <<http://www.mathcs.richmond.edu/~hubbard/>> - Acesso em: 15 de Fevereiro de 2006.

KERNIGHAN, B. W; RITCHIE, D. M. The C Programming Language. Ed. Prentice Hall-USA. New Jersey, 1988.

KILGARD, M. J. OpenGL Programming for the X Window System. Ed. Addison Wesley Longman. Boston, 1994.

MOHAREB, M. (1994). Deformational Behavior of Line Pipe, PhD Thesis, Department of Civil Engineering, Univ. of Alberta. Edmonton, AB, 6G2G7- Canadá.

MSDN LIBRARY. Win32 Application Programming Interface. Disponível em: <<http://msdn.microsoft.com/library/default.asp>>. Acesso em: 17 de Maio de 2006.

PETZOLD, C. Programando para Windows 95, Tradução: Jeremias René Descarts Pereira dos Santos. Ed. MAKRON. São Paulo, 1996.

RICHARDS, M. BCPL The Language and Its Compiler. Cambridge University Press. England, 1980.

SCHILDT, H. C Completo e Total 3ª edição. Ed. MAKRON. São Paulo, 1997.

SCHREINER, D. OpenGL Architecture Review Board. OpenGL(R) Reference Manual: The Official Reference Document to OpenGL, Version 1.4. Addison-Wesley Professional, 4th Edition, 2004.

SOUZA, L.T.; MURRAY, D.W. (1994). Prediction of Wrinkling Behavior of Girth-Welded Line Pipe, Structural Engineering Report No. 197, Department of Civil Engineering, University of Alberta. Edmonton, AB, T6G 2G7.

STROUSTRUP, B. A linguagem de programação C++. Trad. Maria Lúcia Blanck Lisbôa e Carlos Arthur Lang Lisbôa. Ed. Bookman. Porto Alegre, 2000.

THE INDUSTRY'S FOUNDATION FOR HIGH PERFORMANCE GRAPHICS. Página Oficial da Biblioteca OpenGL. Disponível em: <<http://www.opengl.org/>>. Acesso em: 10 de Fevereiro de 2006.

WIN32 API TUTORIAL. Tutorial sobre a Win32 API. Disponível em: <<http://www.winprog.org/tutorial/>> Acesso em: 13 de Maio de 2006

WOODWARD, P. M. Algol 68-R Users Guide. Her Majesty's Stationery Office. England, 1974

WRIGHT, R. S. Jr; SWEET, M. OpenGL SuperBible. 2.ed. Waite Group Press. Indiana, 2000.

YOOSEF-GHODSI, N. (2002). Analysis of Buried Pipelines with Thermal Applications, PhD Thesis, Department of Civil Engineering, University of Alberta. Edmonton, AB. T6G 2G7 - Canadá.

ZHOU, Z. (1988). Numerical Structural Analysis of Buried Pipelines, PhD Thesis, Dept. of Civil Engineering, Univ. of Alberta. Edmonton AB, T6G 2G7 - Canadá.