Rafael Guimarães Ramos

Ambiente Virtual de Simulação e Visualização do Comportamento de Risers

Campinas
2013

Universidade Estadual de Campinas
Faculdade de Engenharia Elétrica e de Computação

Rafael Guimarães Ramos

Ambiente Virtual de Simulação e Visualização do Comportamento de Risers

Dissertação de mestrado apresentada ao Programa de Pós-Graduação em Engenharia Elétrica da Faculdade de Engenharia Elétrica e de Computação da Universidade Estadual de Campinas como parte dos requisitos exigidos para a obtenção do título de Mestre em Engenharia Elétrica. Área de concentração: Engenharia de Computação.

Orientador: Prof. Dr. José Mario De Martino

Este exemplar corresponde à versão final da dissertação defendida pelo aluno Rafael Guimarães Ramos, e orientada pelo Prof. Dr. José Mario De Martino

_____

Campinas
2013

Ficha catalográfica
Universidade Estadual de Campinas
Biblioteca da Área de Engenharia e Arquitetura
Elizangela Aparecida dos Santos Souza - CRB 8/8098

R147a

Ramos, Rafael Guimarães, 1987-
Ambiente virtual de simulação e visualização do comportamento de risers / Rafael Guimarães Ramos. – Campinas, SP : [s.n.], 2013.

Orientador: José Mario De Martino.
Dissertação (mestrado) – Universidade Estadual de Campinas, Faculdade de Engenharia Elétrica e de Computação.

1. Ambiente virtual. 2. Simulação computacional. 3. Estruturas marítimas - Hidrodinâmica. 4. Visualização. I. De Martino, José Mario,1958-. II. Universidade Estadual de Campinas. Faculdade de Engenharia Elétrica e de Computação. III. Título.

Informações para Biblioteca Digital

Título em inglês: Virtual environment for simulation and visualization of riser behavior
Palavras-chave em inglês:
Virtual environment
Computer simulation
Marine structures - Hydrodynamics
Visualization
Área de concentração: Engenharia de Computação
Titulação: Mestre em Engenharia Elétrica
Banca examinadora:
José Mario De Martino [Orientador]
Renato Pavanello
Léo Pini Magalhães
Data de defesa: 05-06-2013
Programa de Pós-Graduação: Engenharia Elétrica

COMISSÃO JULGADORA - TESE DE MESTRADO

Candidato: Rafael Guimarães Ramos

Data da Defesa: 5 de junho de 2013

Título da Tese: "Ambiente Virtual de Simulação e Visualização do Comportamento de Risers"

Prof. Dr. José Mario De Martino (Presidente): _____

Prof. Dr. Renato Pavanello: _____

Prof. Dr. Léo Pini Magalhães: _____

Dedico este trabalho à minha família e aos meus amigos.

Agradecimentos

Agradeço,

ao Prof. José Mario De Martino pela orientação e incentivo,

ao Prof. Celso Morooka e seus orientados Denis Shiguemoto, Maurício Suzuki e Raphael Tsukada, por todas as explicações sobre *risers* e pelo material de apoio,

ao Rodrigo Mologni, colega de sala, que em diversos momentos me ajudou no uso e instalação das ferramentas utilizadas,

aos voluntários da avaliação de usabilidade, pela disponibilidade e paciência em participar do teste,

ao CNPq pelo apoio financeiro concedido durante o período de mestrado,

à FEEC/UNICAMP a ótima estrutura que oferece aos estudantes e pesquisadores,

a todos os colegas de laboratório pelas diversas dicas e ótima convivência e companheirismo.

Idéia, do grego antigo *idea*, por sua vez derivado de *idein*, ver.

Online Etymology Dictionary (adaptado)

Resumo

Este trabalho apresenta um ambiente de visualização interativo em tempo-real para a simulação do comportamento de *risers* rígidos verticais. O *riser* é um duto cilíndrico essencial na extração de óleo em águas profundas e ultra-profundas, uma tarefa desafiadora que impõe diversas cargas sobre a estrutura. Ondas, correntes e movimentos da plataforma são algumas dessas fontes de tensão, que podem levar a danos por fadiga ou mesmo rupturas. Simulações computacionais são uma ferramenta de grande valia para prevenir e diagnosticar tais problemas, mas em geral apresentam a desvantagem de produzir um grande volume de dados numéricos de difícil interpretação. Técnicas de visualização científica podem ser utilizadas para representar os dados de uma maneira mais intuitiva e realista. Entretanto, os sistemas identificados na literatura apresentam limitações quanto à interação em tempo-real. A visualização é realizada como um *playback*, após a simulação ter sido completada, e sempre que os parâmetros de simulação são alterados o usuário deve esperar um tempo considerável enquanto os resultados são recalculados. Neste trabalho, o ambiente desenvolvido permite a visualização do comportamento do *riser* com interação em tempo-real, em que o novo comportamento do *riser* é obtido imediatamente após os parâmetros de simulação serem alterados pelo usuário.

Palavras-chave: Ambiente Virtual. Simulação Computacional. Estruturas Marítimas - Hidrodinâmica. Visualização.

Abstract

This work presents a visualization environment with real-time interaction for the simulation of vertical rigid risers. The riser is a cylindrical pipe essential for the extraction of oil in deep and ultra-deep waters, a challenging task that imposes several loads over the structure. Waves, currents and platform movements are some of these stress sources, that may lead to fatigue damage or even rupture. Computer simulations are a tool of great value for preventing and diagnosing such problems, but have usually the drawback of producing a large volume of numerical data difficult of being interpreted by the user. Scientific visualization techniques can be used to represent data in a more intuitive and realistic way. However, projects identified in the literature present limitations regarding real-time interaction. The visualization is performed as a playback, after the simulation has been completed, and whenever simulation parameters are changed, the user has to wait a considerable time for the results to be recalculated. In this work, the developed simulation environment allows visualization of riser behavior with real-time interaction, where the new riser behavior is obtained immediately after simulation parameters are changed by the user.

Key-words: Virtual Environment. Computer Simulation. Marine Structures - Hydrodynamics. Visualization.

Lista de Figuras

Lista de Tabelas

Lista de Acrônimos e Notação

K_X, K_Y	matriz de rigidez *inline* e transversal, respectivamente
D_X, D_Y	matriz de amortecimento *inline* e transversal, respectivamente
M_X, M_Y	matriz de massa *inline* e transversal, respectivamente
F_X, F_Y	vetor das forças sobre o *riser* nos planos *inline* e transversal, respectivamente
X, Y	vetor dos deslocamentos *inline* e transversal, respectivamente
$\dot{X}, \dot{Y}$	vetor das velocidades *inline* e transversal, respectivamente
$\ddot{X}, \ddot{Y}$	vetor das acelerações *inline* e transversal, respectivamente
VRP	*Visual Riser Prod*, interface para o *Solver RiserProd*

Sumário

Capítulo 1

Introdução

1.1 Motivação

A exploração de petróleo em águas profundas e ultraprofundas é uma tarefa desafiadora, sobretudo devido às condições ambientais adversas a serem enfrentadas. A grande importância deste recurso mineral no cenário global, entretanto, tem impulsionado essa atividade, especialmente no caso do Brasil, que possuiu aproximadamente 94% de suas reservas provadas de petróleo em campos marítimos (Agência Nacional do Petróleo 2012).

O processo de obtenção de petróleo em alto mar pode ser dividido em três etapas[1]: *Prospecção*, *Perfuração* e *Produção*. Na etapa de Prospecção, é feita a localização das reservas de petróleo, utilizando diferentes técnicas, que incluem o uso de ondas de choque, medidas das variações nos campos gravitacional e magnético, ou ainda farejadores de hidrocarbonetos. Na etapa seguinte, é feita a perfuração do poço utilizando-se uma broca. Por fim, na última etapa, Produção, o óleo é extraindo do poço e levado até a plataforma.

Uma estrutura essencial nas etapas de perfuração e produção é o *riser* (Figura 1.1), um tubo delgado cilíndrico que atende a múltiplas funções, entre elas a de transportar o petróleo desde o poço até a unidade de produção. De um modo geral, existem dois tipos de *riser*, os rígidos e os flexíveis, e a operação a ser realizada determina o tipo de *riser* utilizado.

Existem diversos tipos de configurações de *riser*. O *Top Tension Riser* (*TTR*) é um tubo rígido e vertical que tem as extremidades presas na cabeça do poço e na unidade de produção, e deve estar sempre tracionado no topo para garantir sua estabilidade. Também rígidas e verticais são as configurações *Collector*, em que o tubo fica preso apenas no topo, e *Tower*, que possui apenas a base presa. A configuração *Collector* é utilizada para coletar a água do mar, já o tipo *Tower* é utilizado em uma configuração híbrida conhecida como *HRT* (*Hybrid Riser Tower*), no qual além do duto de aço há uma bóia no topo do *riser* (topo que fica submerso) e tubos flexíveis chamados *jumpers* partindo do topo para a embarcação. Há ainda o *Steel Catenary Riser* (*SCR*), que é rígido e possui a forma de catenária, e o *Subsea Pipeline*, em que o duto

[1] Aqui a divisão em três etapas e os termos empregados foram escolhidos de forma a facilitar a compreensão das explicações posteriores e evitar ambiguidades. Diferentes nomenclaturas podem ser encontradas em outras fontes. Em algumas, é feita apenas a divisão entre Exploração e Produção, onde por Exploração entende-se a prospecção e estudo das reservas e Produção tanto a perfuração quanto a extração do petróleo. Utiliza-se ainda em algumas fontes o termo Explotação, designando a extração do recurso natural, contrapondo-se a Exploração.

é rígido e disposto na horizontal. Todos estes são *risers* rígidos e não podem sofrer grandes deflexões. Ademais, *TTR*, *Collector* e *Tower* compõem o subgrupo dos *risers rígidos verticais*, já que além de serem tubos rígidos são utilizados na posição vertical. Por fim, além dos *risers* rígidos, existe também a classe dos *risers* flexíveis, que toleram deflexões maiores.

O escolha pelo uso de cada configuração de *riser* varia conforme a situação. Durante a etapa de perfuração, apenas *risers* rígidos podem ser utilizados. O *riser* tem nesta etapa a função de guiar a broca desde a plataforma até a cabeça do poço, bem como transportar o fluido de perfuração. O fluido de perfuração é uma mistura de água, argila e outros compostos químicos que tem por função lubrificar e resfriar a broca, dar sustentação ao *riser* e impedir que o petróleo escape do poço assim que o reservatório é atingido. O *riser* utilizado na perfuração possui diâmetro variando entre 0,50m e 1,00m. Caso a embarcação provoque um deslocamento horizontal (*offset*) acima do limite seguro para o *riser*, este pode ser desconectado hidraulicamente da cabeça do poço.

Na etapa de produção, podem ser utilizados tanto *risers* rígidos como flexíveis. O *riser* nesta etapa tem a função de transportar os hidrocarbonetos da cabeça do poço até a embarcação. O *riser* rígido utilizado na produção possui diâmetro da ordem de 0,25m, não pode sofrer grandes deflexões e deve estar sempre tracionado. Já o *riser* flexível pode sofrer maiores deflexões e durante o uso permanece suspenso na forma de uma catenária. O diâmetro do *riser* flexível varia de 0,064m a 0,41m.

Durante seu tempo de uso, o *riser* é submetido a carregamentos que provocam a movimentação da estrutura, podendo levar a fraturas ou mesmo rupturas por fadiga do material. Correnteza, ondas, movimentação da plataforma, bem como a diferença de pressão entre a água do mar e o fluido interno, são alguns dos principais fatores que influenciam o comportamento do *riser*. Outro fator importante é a *VIV* (*Vortex Induced Vibration*), a vibração induzida pelo desprendimento de vórtices ao longo da parede do *riser*.

A necessidade de prevenir e diagnosticar danos no *riser*, somado aos altos custos tanto da produção do equipamento quanto de sua manutenção, tornam importante o uso de modelos computacionais que simulam o comportamento desta estrutura. Com o uso de um simulador, pontos de falha no *riser* podem ser detectados com antecedência, antes mesmo do *riser* ser fabricado. Outro caso de uso importante para um simulador é o de auxiliar no diagnóstico do problema de um *riser* em operação. O simulador pode indicar mais precisamente o tipo e a localização do problema, atenuando eventuais custos de manutenção. Entretanto, o grande volume de dados numéricos produzido pela simulação torna dificil a interpretação dos resultados pelo usuário. Uma forma de mais tornar mais intuitiva a compreensão destes dados é utilizar a visualização científica.

A visualização científica é um ramo específico da visualização de dados, às vezes chamada apenas de visualização, e que consiste em representar dados na forma de imagens[2]. Prova-

[2]A relação entre o que é visualização e o que é computação gráfica pode variar conforme a definição de cada um dos campos, podendo haver sobreposições entre as áreas. Uma distinção é de que a visualização trata de como representar dados, muitas vezes abstratos e sem correspondente visual óbvio, numa forma visual, enquanto a computação gráfica lida com as técnicas para se representar tais elementos gráficos num computador. Desta forma, a visualização em principio possuiria certa independência da computação, ao contrário da computação gráfica. Entretanto, a preponderância do computador como ferramenta para produção e manipulação de dados, bem como a importância da interatividade na visualização tornam forte a ligação entre visualização e computação

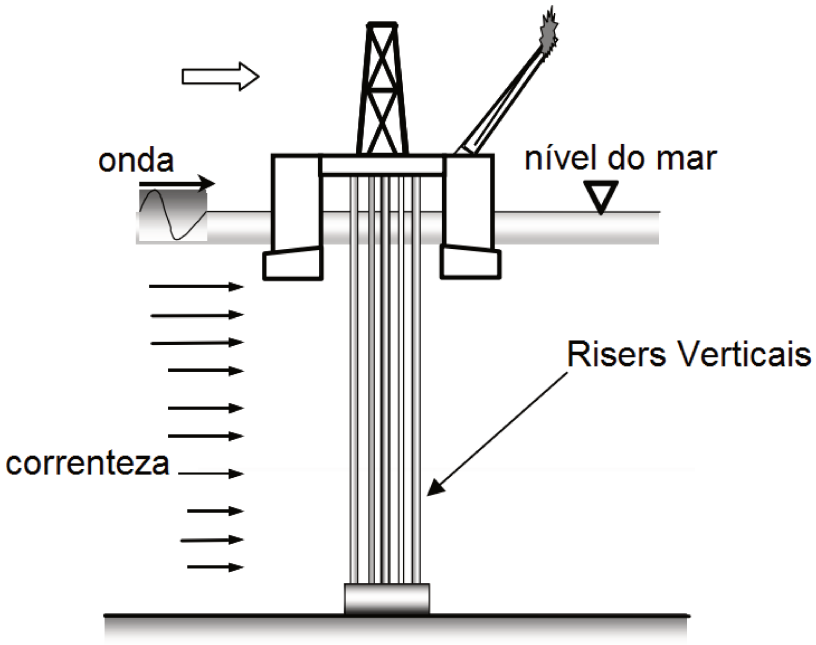

Figura 1.1: Plataforma com Risers Rígidos Verticais.

velmente o exemplo mais comum de visualização seja o dos mapas meteorológicos, mostrados na previsão do tempo dos jornais. A visualização também pode ser utilizada para representar modelos de engenharia, na representação de átomos e moléculas, de dados financeiros ou estatísticas sociais, ou ainda de redes como a Internet ou uma rede elétrica. A função da visualização é a de auxiliar o usuário a interpretar um conjunto de dados. Técnicas de visualização, se bem aplicadas, tem a capacidade de sintetizar grande quantidade de informação, provendo ao usuário uma leitura rápida e intuitiva da informação. Esta fluidez é ainda mais importante dentro de um contexto interativo, em que o ciclo de leitura e produção de dados repete-se diversas vezes.

A visualização científica trata especificamente de dados das áreas científicas e de engenharia. Estes dados, sendo geralmente grandezas físicas, apresentam certas características comuns, como a de serem dados escalares, vetoriais ou tensoriais, e de estarem distribuídos no tempo e no espaço. Sendo assim, um conjunto de técnicas de visualização científica pode ser utilizado para representar uma grande variedade de aplicações. Dentre as técnicas mais comuns está o uso de cores para representar escalares, setas para vetores, linhas de fluxos para campos e elipsóides para tensores.

Utilizando a visualização científica, os dados de simulação podem ser representados numa forma mais intuitiva para o usuário. As múltiplas grandezas do *riser*, como tensão, força e deslocamento de cada elemento, todas elas variantes no tempo, podem ser sintetizadas numa animação provendo uma visão global do comportamento do *riser*. Também os parâmetros de simulação, que determinam fatores como correnteza ou profundidade, podem ser representados visualmente na mesma cena. Um outro fator que enriquece ainda mais a compreensão do fenômeno é interatividade com a simulação, na qual é permitido que o usuário altere as condições

gráfica.

ambientais e do *riser*, explorando diferentes casos de estudo[3]. Entretanto, como será visto no Capítulo 2, a interatividade na visualização de simulações de *riser* é uma funcionalidade desejada e que apresenta limitações nos trabalhos indentificados na literatura. Nas aplicações mais completas em termos de visualização, a mudança de parâmetros é possível apenas interrompendo-se visualização, que só pode ser retomada após os resultados da simulação terem sido recomputados para o intervalo de estudo, o que acarreta um tempo de espera. Este tempo de espera, por sua vez, prejudica a fluidez da interação e da exploração.

1.2 Objetivo

O objetivo deste trabalho é especificar, desenvolver e avaliar um ambiente virtual interativo em tempo-real para simulação e visualização do comportamento de *risers* rígidos verticais. Neste sistema, o usuário poderá visualizar em gráficos animados os dados da simulação, tanto parâmetros quanto resultados, e poderá alterar os parâmetros, que correspondem às condições ambientais e do *riser*, sem que haja interrupção da visualização ou tempo de resposta longo a ponto de prejudicar a fluidez da interação. É importante ressaltar que, neste trabalho, o tempo-real refere-se à fluidez na interatividade, e não que o tempo de visualização da simulação corresponda necessariamente ao do fenômeno real.

1.3 Contribuições

São contribuições deste trabalho:

- O desenvolvimento de um módulo de simulação numérica (*Solver*) apto ao uso interativo em tempo-real, a partir de uma adaptação do simulador *RiserProd*, que não atende os requisitos para tal tipo de interação.
- O desenvolvimento de um módulo de visualização, responsável por representar visualmente os resultados e parâmetros do módulo de simulação e de gerenciar a interação com o usuário, através de uma interface que permite o ajuste dos parâmetros da simulação.
- A integração dos referidos módulos em um único ambiente de simulação e visualização de *risers*.
- Uma avaliação do desempenho do ambiente, revelando quais os limitantes para o uso interativo em tempo-real do ambiente desenvolvido.
- Uma avaliação da usabilidade do ambiente.

[3]Segundo Telea (Telea 2008), o processo de interação e interpretação dos dados representados visualmente pode se dar de duas formas distintas. Uma maneira é aquela em que o usuário possui *a priori* questões que deseja elucidar e a visualização deve auxiliar neste processo. Encontrar os pontos de máximo e mínimo num conjunto de dados ou verificar se os dados estão bem distribuídos no espaço são alguns exemplos dessas perguntas. Uma outra abordagem é quando o usuário não procura por nada específico, ou mesmo não sabe o que procurar, e a visualização releva características interessantes e inesperadas sobre o conjunto de dados. Vale lembrar que as duas abordagens podem ocorrer simultaneamente, complementando-se.

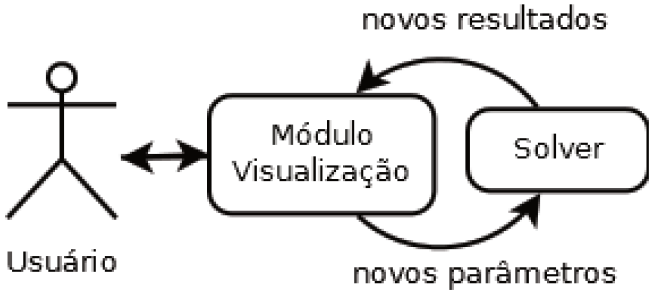

Figura 1.2: Módulos do ambiente desenvolvido.

1.4 Visão Geral

Sendo um ambiente de simulação e visualização, o sistema foi dividido em dois módulos, um de simulação (chamado de *Solver*) e um de visualização, que interagem compondo uma única aplicação (Figura 1.2).

No módulo de simulação, o comportamento do *riser* é calculado segundo as equações da hidrodinâmica de estruturas *offshore*, utilizando a abordagem numérica do Método de Elementos Finitos. Condições ambientais e do *riser* correspondem a parâmetros no modelo matemático, e o comportamento é expresso em variáveis como deslocamento, velocidade e força computadas pelo modelo.

Já no módulo de visualização, os dados da simulação, tanto parâmetros quanto resultados, são processados pelo *pipeline* de visualização e apresentados na forma gráfica para o usuário. Também é este módulo o responsável por gerenciar a interação com o usuário, devendo prover interfaces apropriadas para mudança de parâmetros. O módulo de visualização encapsula o de simulação, e o usuário não possui acesso ao segundo senão através do primeiro. Do ponto de vista do usuário, pode-se dizer que *a visualização é a simulação.*

Este trabalho, além de desenvolver os dois módulos e integrá-los numa única aplicação, avaliou o ambiente desenvolvido segundo sua performance e sua usabilidade. Quanto a performance, a avaliação visa identificar quais os fatores limitantes para um uso interativo em tempo-real. Já quanto a usabilidade, o objetivo é verificar se o ambiente é de fato útil para engenheiros trabalhando com *risers.*

Nos próximos capítulos, estes e outros aspectos do trabalho são detalhados e discutidos.

Em primeiro lugar, no Capítulo 2, Revisão Bibliográfica, são expostos resumidamente os trabalhos relacionados a este que, ou serviram de base para este trabalho, ou trataram do mesmo tema. Assim, são discutidos trabalhos da área de simulação de *riser* e também de visualização. Por fim, constata-se que, nos trabalhos identificados na literatura, a visualização da simulação com interatividade em tempo-real é ainda limitada.

No Capítulo 3, Metodologia, é descrito como o projeto foi realizado, desde sua modelagem matemática, requisitos do sistema, até ferramentas escolhidas. Também são descritos os métodos escolhidos para se avaliar o *Solver* e a interface desenvolvidos.

No Capítulo 4, os resultados obtidos seguindo-se a metodologia do capítulo 3 são descritos. A aplicação desenvolvida é apresentada, bem como uma visão geral do código e os resultados das avaliações da interface e do *Solver.*

Por fim, no Capítulo 5, são mostradas as conclusões obtidas com esse trabalho, suas limitações e os possíveis trabalhos futuros.

Capítulo 2

Revisão Bibliográfica

2.1 Simulação de *Riser*

A hidrodinâmica de estruturas *offshore* possui uma literatura consolidada. O livro de S. K. Chakrabarti, *Hydrodynamics of Offshore Structures* (Chakrabarti 1987) descreve a dinâmica de diversos tipos de estrutura, inclusive *risers*, segundo os vários modelos de onda existentes. Para o fenômeno de geração de vórtices, Kubota (Kubota 2003) indica que diversos modelos têm sido propostos, utilizando as equações de *Navier-Stokes* com dependência no tempo. Tais modelos variam significativamente quanto às considerações básicas e aproximações empregadas, de forma que cada modelo deve apenas ser utilizado quando tais pressupostos estiverem de acordo com a realidade.

Devido às dificuldades em se obter soluções analíticas para sistemas de equações diferenciais parciais como as da hidrodinâmica de estruturas *offshore*, frequentemente modelos numéricos são utilizados para o estabelecimento de soluções aproximadas com um grau desejado de precisão. Uma técnica importante é o *Método de Elementos Finitos*, muito utilizada para análise estrutural. Com a técnica, uma estrutura continua é discretizada em um conjunto de elementos finitos interligados, sendo o sistema de equações diferenciais parciais convertido para um sistema de equações algébricas ou de equações diferenciais ordinárias, que pode ser solucionado por métodos de álgebra linear ou métodos iterativos. Os livros *Finite Element Procedures* (Bathe 1996) e *The Finite Element Method in Engineering Science* (Zienkiewicz 1971) tratam do método em detalhe.

O trabalho de Ferrari e Bearman (Ferrari & Bearman 1999a) estabelece um modelo matemático de simulação, sendo este o utilizado no presente trabalho. No modelo, os coeficientes hidrodinâmicos utilizados nas equações da estrutura devem ser passados como parâmetros e o *Método de Elementos Finitos* é utilizado para obter a solução aproximada. Diversos trabalhos (Morooka, Coelho, Ribeiro, Ferrari & Franciss 2005, Morooka, Coelho, Kubota, Ferrari & Ribeiro 2004, Morooka & Tsukada 2011, Kubota 2003) seguiram-se ao de Ferrari e Bearman, aperfeiçoando o modelo proposto.

2.2 Visualização Científica

As principais técnicas para visualização de dados são tratadas no livro de Alexandru Telea, *Data Visualization* (Telea 2008). No livro, tanto a visualização científica quanto a visualização de outras aplicações são explorados, numa abordagem desvinculada de linguagens específicas. Engel et al., no livro *Real Time Volume-Graphics* (Engel, Hadwiger, Kniss, Rezk-Salama & Weiskopf 2006), tratam das técnicas para visualização de dados volumétricos, importante para certas aplicações dentro da visualização cientifica, como visualização de tomografia computadorizada. Já o livro *The Visualization Toolkit*, de Schroeder et al. (Schroeder, Martin & Lorensen 2006), trata da biblioteca VTK para visualização, cobrindo os diversos componentes oferecidos pelo VTK para a construção de aplicações de visualização.

2.3 Visualização da Simulação de *Riser*

Dentre os trabalhos de simulação de *riser* são relativamente poucos aqueles cujo foco principal é a visualização dos dados de simulação, a maioria tratando dos modelos matemáticos de simulação. Mesmo para estes trabalhos, entretanto, existe a necessidade de representar os dados de simulação. A seguir são descritas as abordagens utilizadas em diversos trabalhos, começando pelas mais simples, de trabalhos cujo foco não era a visualização, e terminando com aquelas mais completas, dos trabalhos que tinham por objeto de interesse a visualização em si.

Dentre as abordagens mais simples para visualização, há o uso de *plots* 2D, em que as grandezas do riser são representadas como curvas. O trabalho de Simantiras e Willis (Simantiras & Willis 2001) é um exemplo (Figura 2.1). Outro trabalho que utiliza *plots* para representar resultados de simulação é o *Visual RiserProd* (ou apenas *VRP*), um módulo de visualização desenvolvido para o simulador *RiserProd* (por ser uma interface já existente para o simulador utilizado neste trabalho, ela é discutida com mais detalhes na Seção 3.2.5 do Capítulo 3). Ainda restrito a gráficos bidimensionais, existem trabalhos que utilizam gráficos 2D para representar os vórtices formados (Figura 2.2). Cada imagem representa o vórtice em uma dada profundidade. Os trabalhos de Al-Jamal e Dalton (Al-Jamal & Dalton 2004), Chaplin et al. (Chaplin, Bearman, Cheng, Fontaine, Graham, Herfjord, Huarte, Isherwood, Lambrakos, Larsen, Meneghini, Moe, Pattenden, Triantafyllou & Willden 2005), Chain e Varyani (Chai & Varyani 2002) e Zhu et al. (Zhu, Lin, Jia & Yang 2010) utilizam esse tipo de representação, bem como *plots* 2D para representar outros resultados.

Outros trabalhos, por sua vez, fazem uso de gráficos tridimensionais para visualizar os resultados. O trabalho de Chen e Chen (Chen & Chen 2010) é um exemplo, onde o *riser* é mostrado como um tubo curvado pela correnteza, exibindo os vórtices em certos pontos do *riser* (Figura 2.3). Os vórtices são representados por linhas coloridas representando os fluxos. Uma abordagem muito semelhante é empregada por Ferrari (Flatschart, Meneghini & Jr. 2004). Zhu et al. (Zhu, Ou, Lin & Hu 2011), por outro lado, utilizam uma representação ligeiramente diferente. O *riser* é apresentado apenas como um cilindro, sem curvatura, porém os vórtices são em alguns casos apresentados de forma contínua ao longo do *riser* e em outros apenas para certas profundidades (Figura 2.4). Entretanto, tais visualizações são apenas figuras estáticas, não animações.

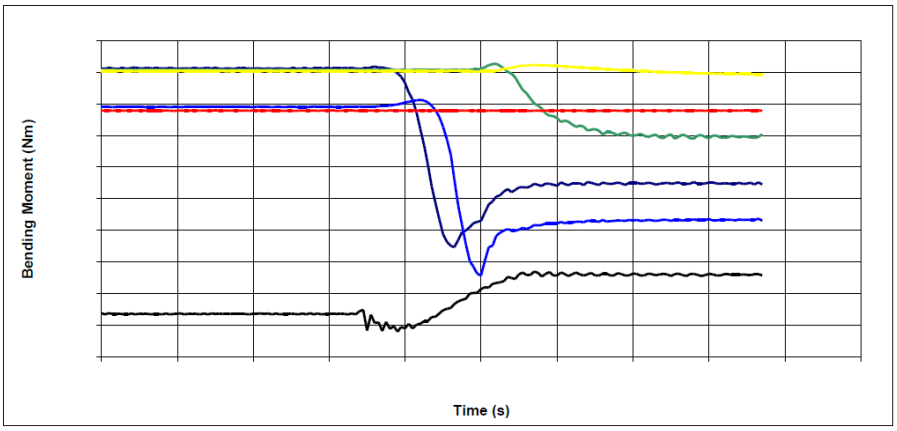

Figura 2.1: Representação por *plots*. Fonte: Simantiras e Willis 2001, página 13, Figura 11.

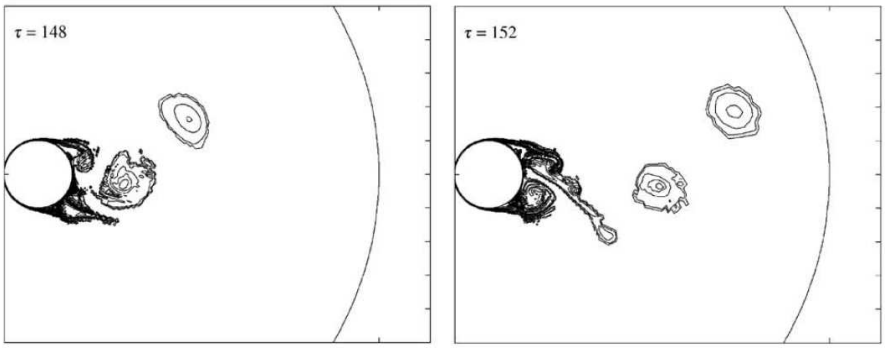

Figura 2.2: Representação de vórtices por imagem 2D. Adaptado de: Al-Jamal e Dalton 2004, página 88, Figura 13.

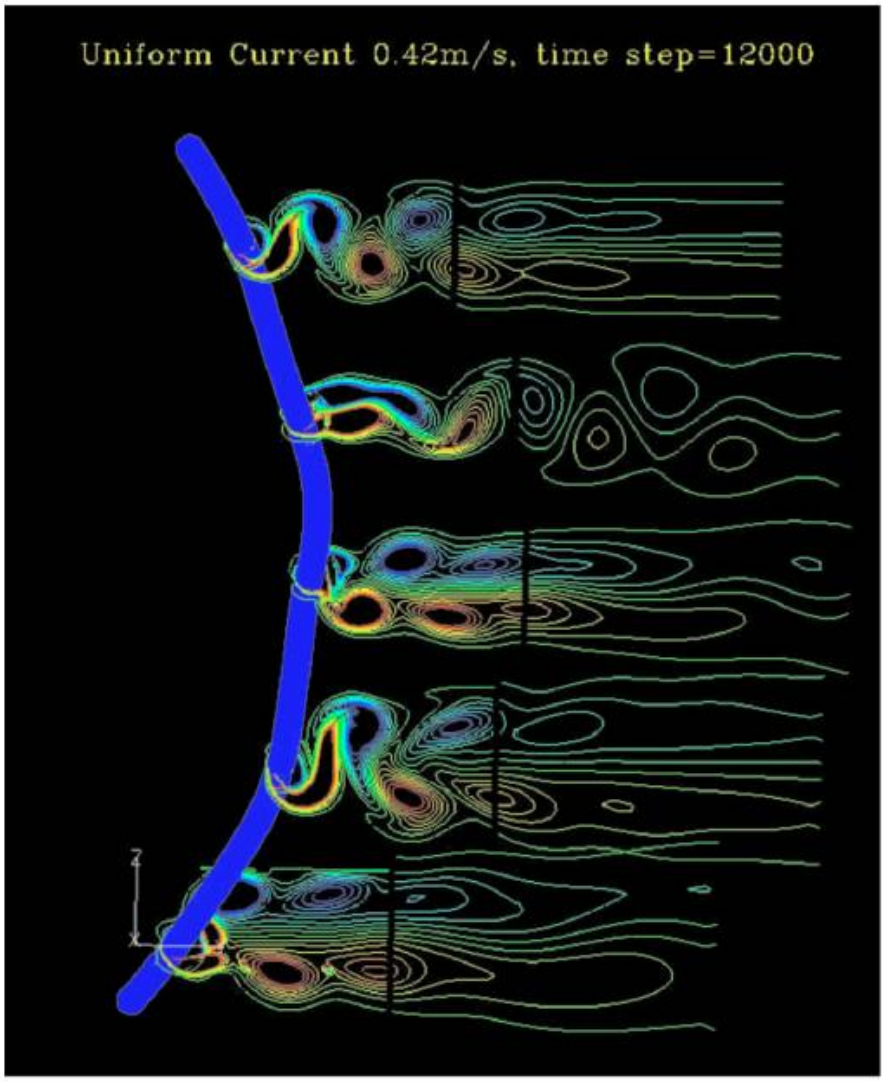

Figura 2.3: Representação de vórtices e curvatura do *riser* por gráficos 3D. Adaptado de: Chen e Chen 2010, página 5, Figura 5.

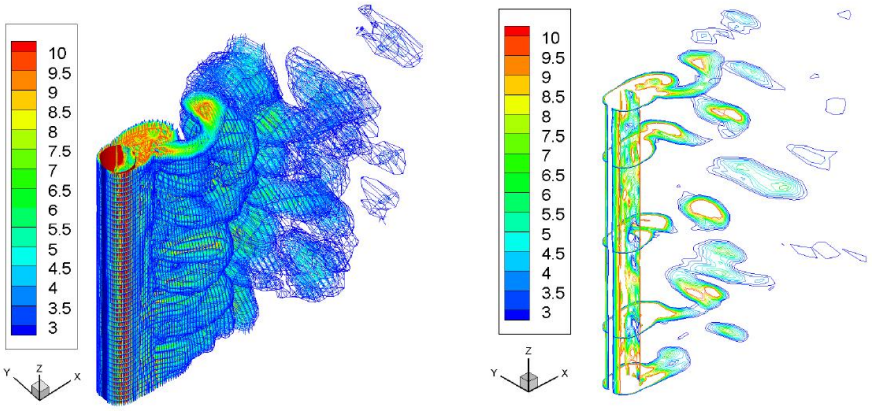

Figura 2.4: Outra representação de vórtices e curvatura do *riser* por gráficos 3D. Adaptado de: Zhu et al. 2011, página 791, Figura 3.

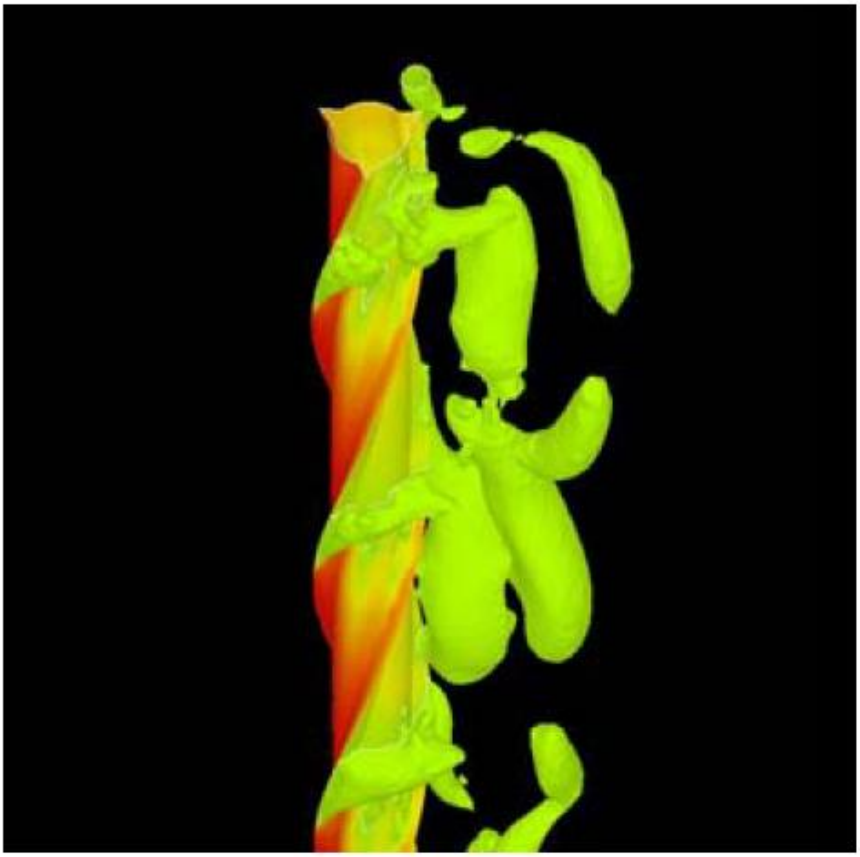

Figura 2.5: Isosuperíficies de pressão via gráficos 3D. Adaptado de: Holmes et al. 2006, página 7, Figura 11.

Gráficos 3D são utilizados também para outros fins além da representação de vórtices ou da curvatura do riser. Holmes et al. (Holmes, Jr. & Constantinides 2006) utilizam gráficos tridimensionais para representar as isosuperfícies de pressão sobre o *riser* (Figura 2.5), enquanto Alexander (Alexander 2003) os emprega para exibir a variação de esforços sobre o *riser* (Figura 2.6). No primeiro caso, diferentes isosuperfícies são exibidas em diferentes cores, enquanto no segundo as cores são utilizadas para representar a intensidade do esforço em cada região do *riser*.

Todos os trabalhos apontados acima tinham por foco a simulação e não a visualização dos resultados. Os trabalhos citados a seguir apresentam um foco maior na visualização da simulação, e combinam algumas das características acima descritas com outras originais.

O sistema *Anflex*(Petrobras 2005), desenvolvido em conjunto pela *Petrobras* e pela *PUC-Rio*, é um *software* para simulação de *risers* em que os resultados são visualizados em um ambiente gráfico 3D (Figura 2.7). Estes resultados podem ser visualizados apenas após toda a simulação ter sido computada. Na visualização, o *riser*, a embarcação, a superfície e o fundo do mar são representados de forma animada, e os esforços ao longo do *riser* são indicados através de cores.

Santos et al. desenvolveram o *Environ* (Santos, Soares, Carvalho & Raposo 2011), uma plataforma colaborativa para engenharia de petróleo que inclui visualização de *riser* (Figura 2.8). O *riser* é representado em gráficos 3D e as forças e outras grandezas físicas são exibidas

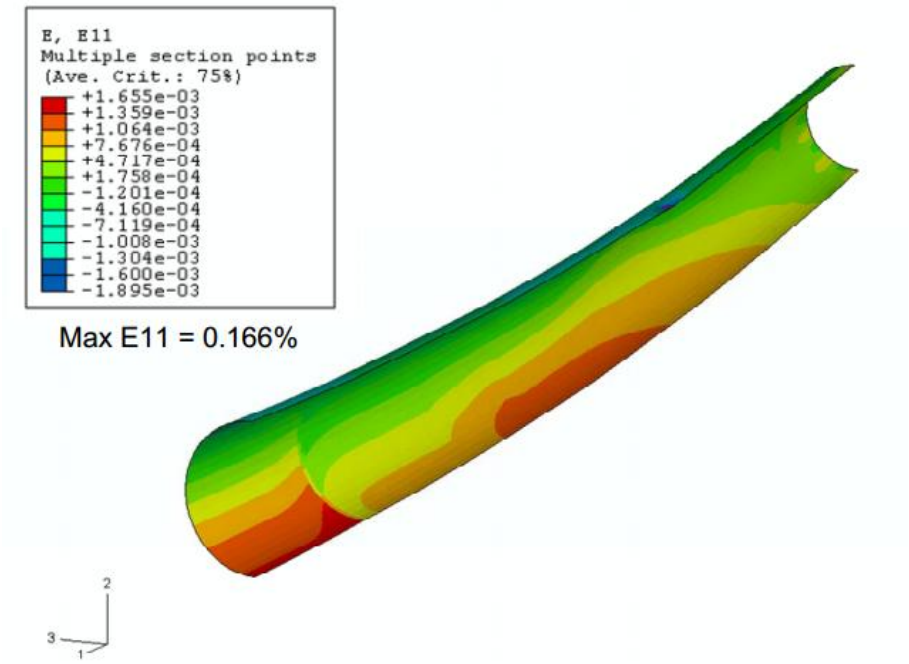

Figura 2.6: Cores em gráficos 3D para representar intensidade de esforço sobre o tubo. Adaptado de: Alexander 2007, página 81, Figura 27.

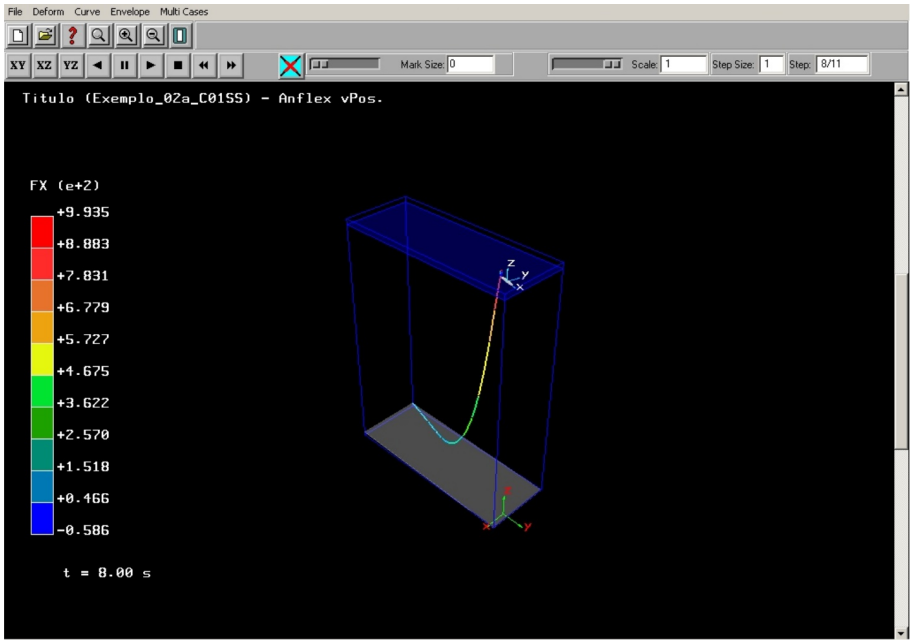

Figura 2.7: *Software* de simulação de *riser* com visualização dos resultados em ambiente gráfico 3D. Fonte: Petrobras et al. 2005, página 39, Figura 3.2.

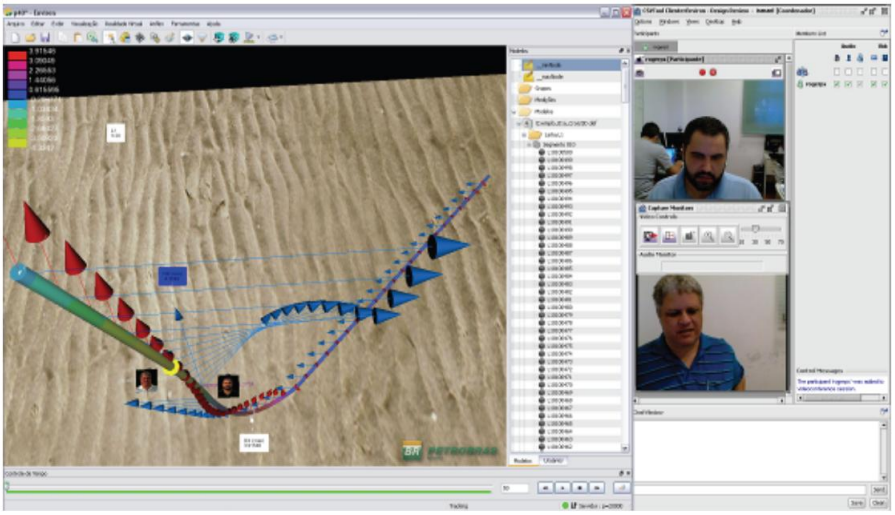

Figura 2.8: Ambiente colaborativo com visualização animada 3D do *riser*. Fonte: Santos et al. 2011, página 6, Figura 6.

com cores e setas. Estes dados são gerados previamente numa etapa de simulação, e só então é realizada a animação. Sendo um ambiente colaborativo, diversos usuários podem utilizar a plataforma simultaneamente, trocar informações e inserir anotações na cena 3D.

O trabalho de Bernardes (Bernardes 2004) também oferece um ambiente de visualização de *riser*, em que o *riser* e as grandezas físicas são representadas em gráficos 3D (Figura 2.9). O valores são computados também numa etapa anterior. O trabalho tem como diferencial a visualização de múltiplos *riser*, com detecção de colisão entre eles.

Já o trabalho de Morooka et al. (Morooka, Brandt, Matt & Franciss 2008) realiza visualização 3D não dos resultados mas dos parâmetros de simulação. Corrente, ondas e profundidade podem ser visualizados e alterados na interface (Figura 2.10). A simulação é feita em seguida e os resultados exibidos em *plots* 2D . No trabalho, há um foco no realismo da cena 3D, especialmente na representação da água.

Por fim, o *software OrcaFlex* (Orcina 2013b) é uma ferramenta comercial desenvolvida pela empresa *Orcina* que oferece simulação e visualização do comportamento de diversas estruturas *offshore*, entre elas, o *riser*. O usuário pode, através de um ambiente virtual (Figura 2.11), montar o cenário a ser simulado, adicionando e posicionando as diversas estruturas. Após isto, pode-se iniciar a simulação numérica. A medida que a simulação é executada, a representação visual da simulação é atualizada com os novos dados. Concluída a simulação, pode-se reproduzir a animação da simulação numa taxa de quadros mais fluida, bem como inspecionar os resultados através de gráficos (*plots*) e tabelas.

Assim, como uma característica comum para estes trabalhos, tem-se que nenhum deles oferece simulação e visualização interativas em tempo-real. A visualização é sempre uma etapa separada da simulação, o que prejudica a fluidez na interatividade, já que a cada mudança de parâmetros o usuário deve esperar pela simulação ser concluída. Uma ressalva deve ser feita para o *software OrcaFlex*, já que os resultados são visualizados na medida que são produzidos. Entretanto, as instruções de uso contidas no manual do *Orcaflex*(Orcina 2013a) indicam que a

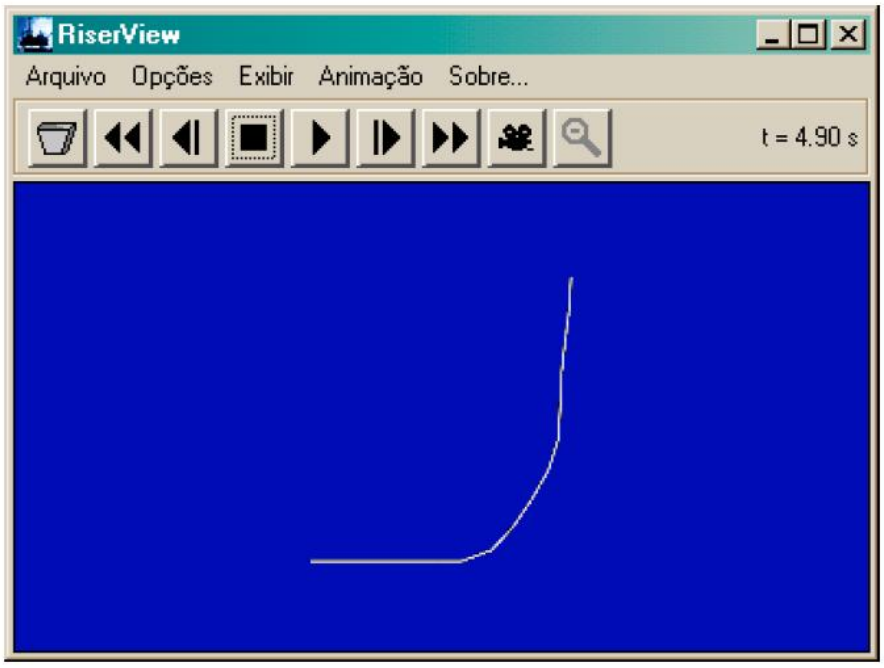

Figura 2.9: Outro ambiente com visualização animada 3D do *riser*. Segundo a fonte, trata-se de um esboço da interface. Fonte: Bernardes 2004, página 119, Figura 20.

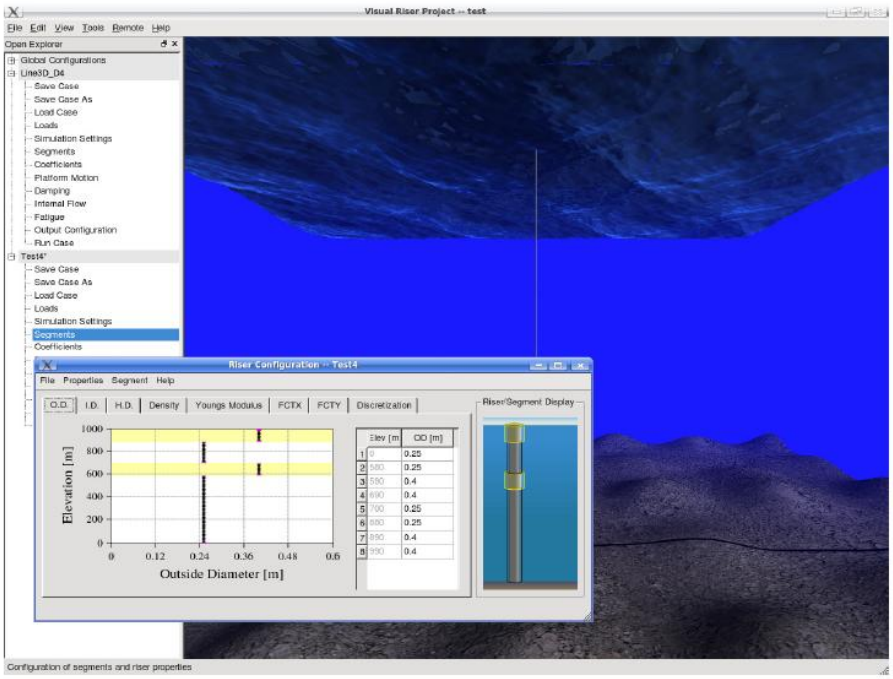

Figura 2.10: Ambiente de visualização 3D para ajuste de parâmetros de simulação. Fonte: Morooka et. al. 2008, página 7, Figura 7

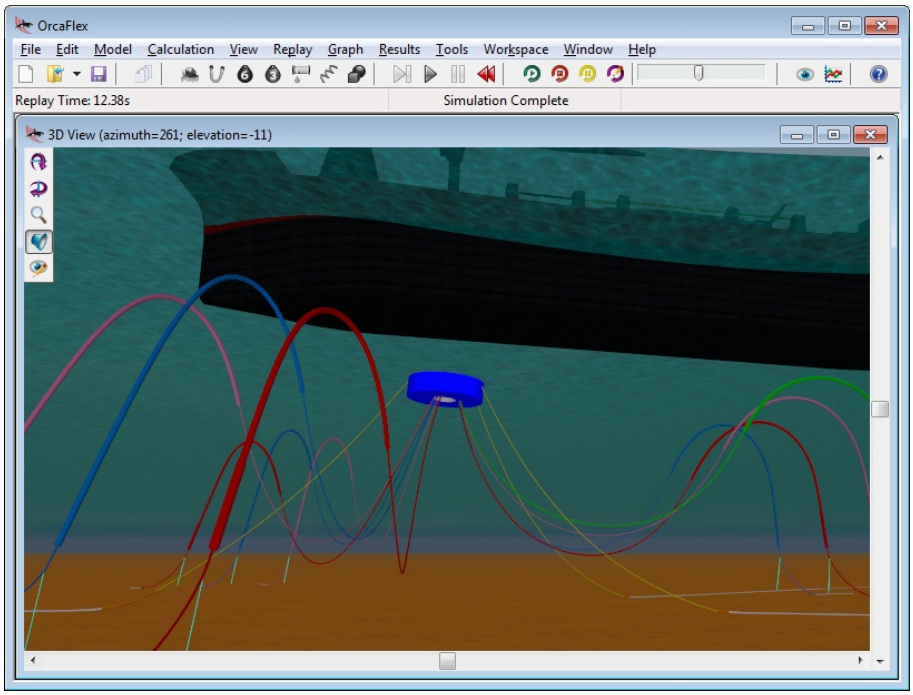

Figura 2.11: *Software* comercial *Orcaflex* desenvolvido por *Orcina Ltd.*. Fonte: Página *Web* de *Orcina Ltd.*

forma de uso padrão do programa é esperar toda a simulação ser executada para então visualizar os resultados.

2.4 Considerações Finais

Como visto neste capítulo, a simulação de *risers* e a visualização científica são áreas de estudo já bastante desenvolvidas. Entretanto, a visualização de *risers*, combinação dos dois campos, apresenta ainda certas limitações. Dentre os trabalhos de simulação de *riser* são poucos aqueles que tem por foco a visualização dos dados, sendo o foco da maioria dos trabalhos os modelos matemáticos de simulação. Alguns dos trabalhos utilizam *plots* 2D para representar as grandezas físicas variantes e imagens bidimensionais para exibir os vórtices para uma dada profundidade. Outros utilizam gráficos 3D, uns para visualizar os vórtices ao longo do tubo e sua curvatura, outros para mostrar com cores as variações de tensão sobre o *riser*. Os trabalhos mais completos de visualização combinam as diversas abordagens numa animação tridimensional.

Porém, mesmo estes trabalhos mais completos possuem limitações. Um quesito importante na visualização é a interatividade, que permite ao usuário explorar o conjunto de dados, auxiliando na busca por respostas específicas ou ainda revelando aspectos inesperados sobre o fenômeno estudado. Os atuais trabalhos sobre visualização de *riser* possuem a limitação de que simulação e visualização ocorrem em etapas separadas, de forma que o usuário deve esperar pela simulação ser concluída toda vez que mudar os parâmetros, o que compromete a fluidez na interação. Uma possível exceção é o *software* comercial *OrcaFlex*, em que a visualização é atualizada à medida que os resultados da simulação são produzidos; mesmo neste caso, entretanto, o

uso padrão da ferramenta consiste em esperar a simulação ser conclúida para só então realizar a visualização. Assim, como dito no Capítulo 1, este trabalho visa superar esta limitação, fazendo com que simulação e visualização ocorram concomitantemente e permitindo uma interação que não interrompa este processo.

Capítulo 3

Metodologia

Neste capítulo são descritos os diversos aspectos envolvidos no projeto e na implementação do ambiente de visualização e simulação. Tais aspectos incluem os fundamentos matemáticos da simulação, o projeto da interface de usuário, a estrutura do software implementado e as ferramentas utilizadas. Sendo um ambiente de visualização e simulação, o sistema desenvolvido está estruturado em dois grandes módulos: o *Solver* e o Módulo de Visualização. O *Solver* é o módulo que produz o comportamento do *riser* a partir de um modelo matemático [1]. Além de implementar o modelo matemático de simulação, o *Solver* deve permitir a alteração de parâmetros de simulação em tempo-real, mantendo sempre a produção dos resultados de simulação numa taxa compatível ao passo da animação. O Módulo de Visualização, por sua vez, encapsula o *Solver* e provê a visualização animada dos dados da simulação, tanto parâmetros quanto resultados, e gerencia a interação com o usuário, provendo funcionalidades que facilitem a mudança de parâmetros, inspeção de resultados e demais ajustes de configuração.

3.1 Simulação do Riser

3.1.1 Modelo teórico

O modelo teórico para simular o comportamento do *riser* tem por base as equações da hidrodinâmica de estruturas *offshore* e utiliza o método de elementos finitos para obter uma solução numérica. Proposto inicialmente por Ferrari e Bearman (Ferrari & Bearman 1999a), o modelo foi aprimorado por estes e outros autores em trabalhos subsequentes (Morooka et al. 2005, Morooka et al. 2004, Morooka & Tsukada 2011, Kubota 2003). Maiores detalhes sobre o modelo podem ser encontrados nestes trabalhos e aqui é apresentada uma explicação simplificada.

O *riser* é considerado como um conjunto de elementos de viga interligados entre si, sob o efeito de uma carga produzida pela combinação da correnteza, onda, pressões hidrostáticas, movimento da plataforma e peso do próprio *riser*, conforme mostrado na Figura 3.1. Esse conjunto de forças no espaço tridimensional é decomposto em dois planos ortogonais: o plano

[1] Para evitar qualquer confusão, *Solver* e simulador numérico são utilizados como sinônimos neste trabalho, referindo-se exclusivamente a parte do programa que soluciona numericamente as equações do modelo matemático do *riser*.

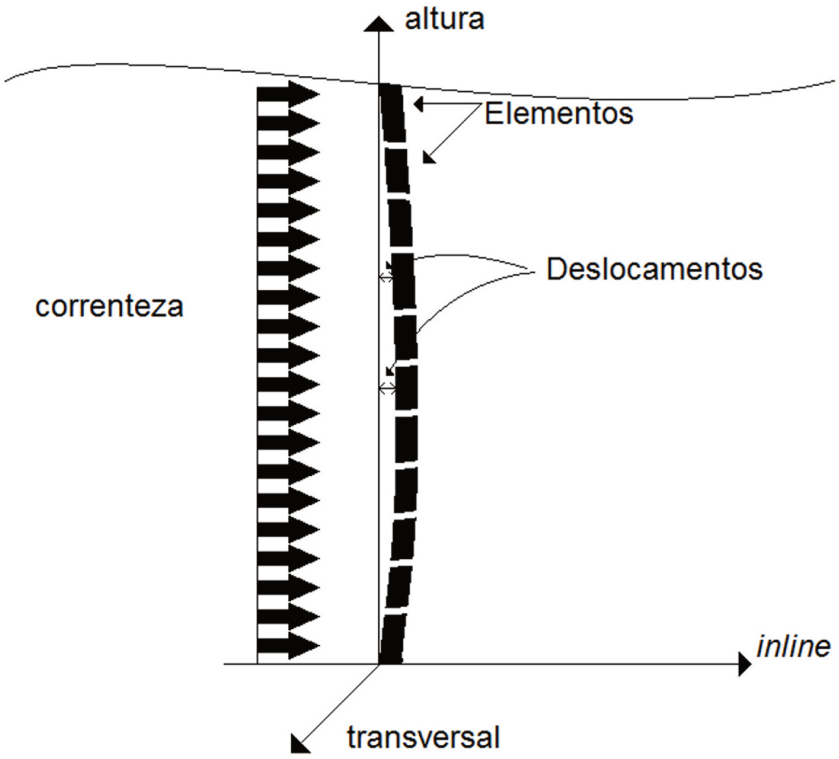

Figura 3.1: Representação esquemática do método de elementos finitos.

inline, paralelo à correnteza, e o plano transversal, ortogonal tanto à correnteza quanto ao fundo do mar. Esta decomposição simplifica o equacionamento do problema, que é representado matricialmente da seguinte forma:

$$M_X\ddot{X} + D_X\dot{X} + K_X X = F_X \tag{3.1}$$

$$M_Y\ddot{Y} + D_Y\dot{Y} + K_Y Y = F_Y \tag{3.2}$$

Esta é uma formulação similar à utilizada para a dinâmica de um corpo rígido, mas adaptada, com o uso de matrizes, para múltiplos elementos. Na Equação 3.1, para o caso *inline*, no lugar de uma massa, um coeficiente de amortecimento e um coeficiente de elasticidade, tem-se as matrizes de massa, de amortecimento e de rigidez, respectivamente M_X, D_X e K_X. X, $\dot{X}$, $\ddot{X}$ são por sua vez os vetores de posição, velocidade e aceleração de cada elemento, e F_X o vetor com as cargas sobre cada elemento. O problema transversal é representado de maneira semelhante (Equação 3.2), mas é importante notar que não só as cargas, como também as matrizes M_Y, D_Y e K_Y podem ser diferentes do plano *inline*.

Partindo-se destas equações, são realizadas três análises para se determinar o comportamento do *riser*: analise estática, de vibrações livres e dinâmica. A análise estática considera a estrutura em equilíbrio e visa encontrar o deslocamento e o ângulo de inclinação dos elementos. Como velocidade e aceleração são nulos, apenas a matriz de rigidez K é utilizada[2]. Como K depende,

[2]Para simplificar a notação, quando não importar se o caso é *inline* ou transversal, as matrizes M, D, K e o

em parte, da posição dos elementos (a chamada componente geométrica da matriz de rigidez), é utilizado o *método de Newton-Raphson*, um método iterativo que converge para a solução do problema, refinando a cada iteração, a matriz K e a posição. A análise de vibrações livres visa encontrar a frequência natural de vibração do *riser*, isto é, aquela na qual o *riser* pode entrar em ressonância. A análise considera apenas as matrizes de massa e rigidez, utilizando os valores calculados na análise anterior para a matriz de rigidez K. A frequência natural é determinada encontrando-se os autovalores do sistema. Por fim, a análise dinâmica computa o comportamento do *riser* para cada intervalo de tempo. Todas as matrizes M, D e K são consideradas, sendo D computada usando M, K e as frequências naturais. A análise é feita pela integração no tempo usando o método de Newmark-Beta (Newmark 1959)

3.1.2 *RiserProd*: Uma implementação já existente do simulador

Uma implementação já existente deste modelo de simulação é o programa batizado de *RiserProd*. Implementado na linguagem *Fortran 90*, com o módulo *LAPACK* para álgebra linear em *Fortran 77*, o *RiserProd* não foi feito pensando-se no uso em tempo-real, nem pode ser diretamente utilizado para tal. O *RiserProd* realiza a simulação do *riser* para um intervalo de tempo pré-determinado, produzindo todos os resultados para cada *time-step* em arquivos, na forma de um *log*. Os parâmetros de simulação são também armazenados num arquivo, que é lido pelo programa no início da simulação. O simulador apenas pára assim que todos os resultados tenham sido produzidos, de forma que, quanto maior o intervalo de estudo estipulado, mais tempo o programa leva para retornar os resultados, o que já é um problema para o uso em tempo-real. Além disso, o uso de arquivos é outro fator a contribuir para um tempo de execução maior. Estas limitações entretanto, em nada impediram que o *RiserProd* fosse tomado como base para a implementação de um novo simulador que estivesse apto ao uso em tempo-real.

3.1.3 Requisitos para um simulador interativo em tempo-real

Um simulador interativo em tempo-real deve permitir que os parâmetros de simulação sejam alterados a qualquer momento, sem que a simulação seja interrompida e sem um tempo de resposta perceptível toda vez que um parâmetro for alterado, o que prejudicaria a fluidez na interação. Além disto, neste tipo de sistema não é possível definir o intervalo de estudo da simulação *a priori*, já que a simulação deve prosseguir enquanto o usuário estiver utilizando o ambiente de visualização. Um intervalo arbitrário até poderia ser definido, sendo atualizado à medida que a simulação é executada, mas, ainda assim, quanto maior o intervalo estabelecido, maior o tempo para computar os dados, o que compromete o imperativo de tempo de resposta baixos na mudança de parâmetros. Assim, um modelo melhor para uma simulação interativa em tempo-real é, ao invés de pré-computar os resultados antes de visualizá-los, calculá-los sob demanda, à medida que a simulação for avançando. Para isto, entretanto, é necessário garantir que os dados sejam produzidos numa taxa compatível à frequência da animação, por exemplo, *30 quadros por segundo*.

vetor F serão utilizados, para se referir tanto as do caso *inline* quanto do transversal. Quando houver diferença entre os casos, esta será explicitamente indicada.

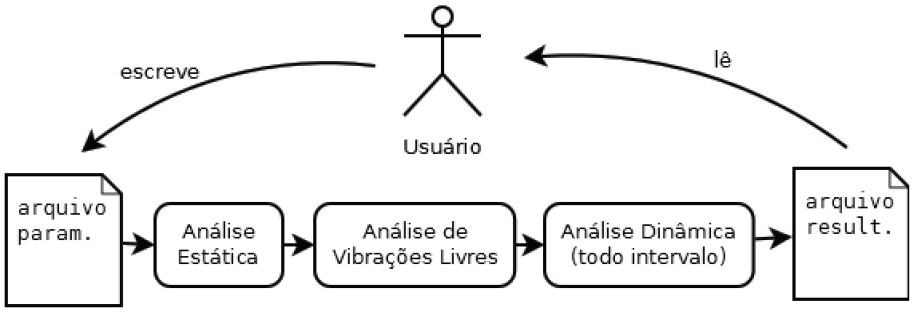

Figura 3.2: Fluxo do *Solver RiserProd*.

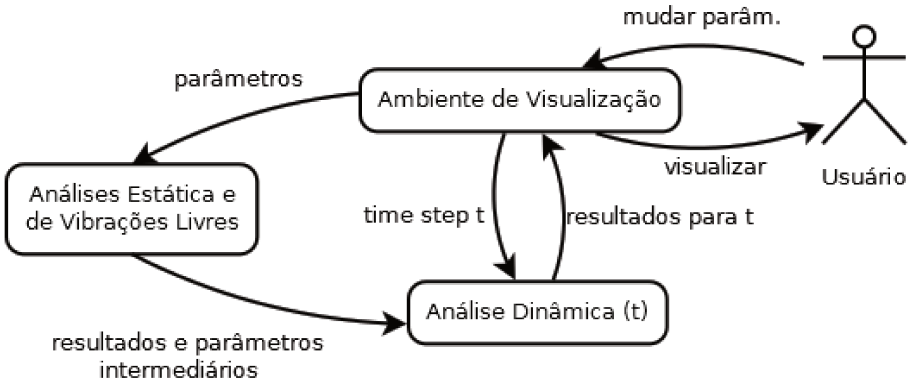

Figura 3.3: Fluxo do *Solver* aprimorado para interatividade em tempo-real.

3.1.4 Implementando o simulador interativo em tempo-real a partir do *RiserProd*

Um simulador interativo em tempo-real foi obtido a partir do *RiserProd* fazendo-se algumas modificações, de acordo com os requisitos apresentados na seção anterior. No *RiserProd*, uma simulação é um processo indivisível, que inicia com a passagem de um arquivo de parâmetros, seguida da realização das análise estática, de vibrações livres e dinâmica (para todo o intervalo de estudo), e por fim, escrevendo os resultados em arquivos de saída, conforme mostrado na Figura 3.2.

Para o novo simulador, esse fluxo deve ser reestruturado e o uso de arquivos eliminado, utilizando um *buffer* para armazenar os resultados e que pode ser compartilhado com a interface de visualização. Na nova estrutura (Figura 3.3), as análises estática e de vibrações continuam sendo realizadas sempre e apenas quando um novo conjunto de parâmetros é passado. Já a análise dinâmica, ao invés de ser realizada para todo um intervalo de estudo quando novos parâmetros são passados, passa a ser realizada a cada quadro de animação mas agora para um único passo da simulação. A integração no tempo realizada na análise dinâmica depende de certas condições de contorno tais como deslocamento e velocidade dos elementos. Os valores utilizados para tais condições são aqueles gerados no passo anterior, exceto na primeira vez em que a análise é executada, em que deslocamento e velocidade são iguais a zero.

Além destas reestruturações, a nova implementação difere do *RiserProd* por estar codificada em *C++*. Esta decisão ocorreu pois o *Fortran*, apesar de ainda popular na área de computação

científica, não é mais tão utilizado fora deste nicho, podendo haver problemas de compatibilidade dele com recursos mais recentes (como integração com a interface escrita em *Python*). Além disso, *C++* é uma linguagem que oferece orientação a objetos, proporcionando um código mais organizado, e conhecida por ter um alto desempenho em tempo de processamento. Quanto ao módulo *LAPACK* para álgebra linear, este foi substituído por sua versão em *C*, o *CLAPACK*.

3.1.5 Avaliando o desempenho do novo *Solver*

A capacidade do ambiente de simulação e visualização suportar interação em tempo-real é limitada pelo *Solver*, pois o custo computacional deste módulo é proporcional ao número do elementos do modelo de simulação. Para o caso do módulo de visualização, não há grande variação de desempenho. Assim, apenas o desempenho do *Solver* é avaliado de maneira formal.

O desempenho do *Solver* é avaliado segundo dois métodos: *profiling* pelo *Visual Studio* e medição direta do tempo. A medida dos tempos é feita utilizando a função *clock()* da biblioteca *time.h* e visa determinar o tempo de execução de cada uma das análises do *Solver*. Para o caso da análise dinâmica, ela é decomposta em duas etapas medidas separadamente, pois embora a análise dinâmica seja executada para cada passo da animação, certos cálculos de matrizes realizados no começo desta análise precisam ser executados apenas quando novos parâmetros são fornecidos. Assim, a medida dos tempos do *Solver* é realizada para quatro etapas: Análise Estática, Análise de Vibrações Livres, Montagem de Matrizes e Análise Dinâmica (aqui referindo-se a parte que é executada a cada passo da animação). Pelo *profiler* do *Visual Studio*, é possível descobrir quais os pontos específicos do programa que contribuem com maior carga de processamento, enquanto a medida direta retorna quanto tempo durou a execução de determinado trecho de código. O *Visual Studio* oferece duas formas diferentes para realizar o *profiling*. Na primeira, chamada de *CPU Sampling*, o estado do programa é capturado em diferentes momentos ao longo da execução e a análise é feita com estes dados. O outro método de *profiling* por sua vez realiza a contagem de quantas vezes cada função é chamada e o quanto de processamento ela realiza (para distinguir funções que processam dados realmente de outras que apenas chamam outras funções).

3.2 Visualização da Simulação

3.2.1 Fundamentos

Visualizar os dados de simulação significa transformar informação numérica em elementos visuais, como cores e formas. Segundo Telea (Telea 2008), o processo de visualização segue um conjunto bem definido de etapas, denominado de *Pipeline de Visualização*. São quatro as etapas: *importação*, *filtragem*, *mapeamento* e *renderização*. Na etapa de importação, dados brutos são importados de alguma fonte externa e armazenados numa estrutura de dados apropriada à visualização. Na etapa de filtragem, dados desnecessários são eliminados, ou ainda novos dados são calculados e incluídos no conjunto, processo chamado de enriquecimento dos dados. No mapeamento, o conjunto de dados filtrado e enriquecido é mapeado para formas visuais bi ou tridimensionais. Por fim, na etapa de renderização, estas formas visuais são desenhadas na tela,

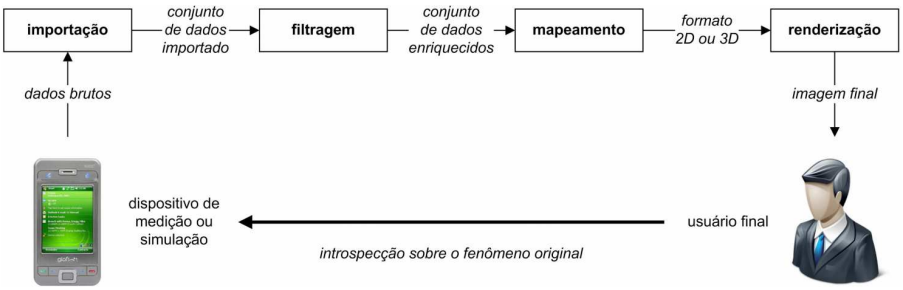

Figura 3.4: Pipeline de Visualização. Fonte: Mologni dos Santos 2011, página 18, Figura 2.3.

gerando a imagem final. A Figura 3.4 ilustra esse processo.

Na visualização científica, costuma-se trabalhar com dados constituídos de um conjunto de amostras e de suas propriedades geométricas e topológicas. Embora dados muitas vezes estejam disponíveis em sua forma contínua, como uma função, a manipulação de dados contínuos é mais difícil e lenta do que a de dados discretos. O conjunto de dados é efetivamente uma malha, onde cada vértice é uma amostra, podendo armazenar múltiplos valores escalares, vetoriais e tensoriais. A geometria dos dados estabelece a posição de cada vértice, e a topologia, como os vértices conectam-se entre si.

O *VTK* (Schroeder et al. 2006), sigla para *The Visualization Toolkit*, é um *framework* orientado a objetos para programação de aplicações de Visualização Científica. O *VTK* oferece classes de objetos que implementam as diversas etapas do *pipeline* de visualização, sendo eles componentes que devem ser conectados uns aos outros de forma a compor o *pipeline*. São fornecidas classes apropriadas para armazenar os dados em suas diversas formas intermediárias, bem como funcionalidades necessárias a maioria das aplicações gráficas, como criação de janelas e interação com o mouse. O *framework* é implementado em *C++*, utilizando *OpenGL* para renderização de gráficos, o que a torna bastante eficiente. Pode também ser utilizado em outras linguagens como *Tcl/Tk*, *Python* e *Java*, através de *bindings* com a implementação em *C++*.

3.2.2 Técnicas para Visualizar a Simulação

Uma forma intuitiva de visualizar os dados de simulação é criar uma cena que reproduza a situação real. Partindo-se deste princípio, aqui a simulação é visualizada mostrando o *riser* em movimento e seu ambiente, isto é, o mar, o leito marinho e a embarcação. Assim, dados como diâmetro do *riser*, deslocamento e velocidade de cada elemento, profundidade do oceano, ou mesmo o flutuadores dispostos ao longo do riser podem ser representados de maneira natural e intuitiva. Outros dados importantes como correnteza ou pressão do fluído interno, embora não sejam normalmente visíveis na natureza, são representados através de setas ou cores, duas técnicas já consagradas na visualização. Para grandezas vetoriais, como forças de excitação sobre o *riser* ou a correnteza, utilizam-se setas, enquanto para grandezas escalares como pressão, densidade ou elasticidade do material do *riser*, o uso de cores mostrou-se mais adequado. Através desta abordagem, é possível fornecer uma visão geral dos diversos aspectos da simulação. Detalhes e realismo na representação podem, em alguns casos, ser sacrificados, buscando-se uma visualização mais intuitiva do todo.

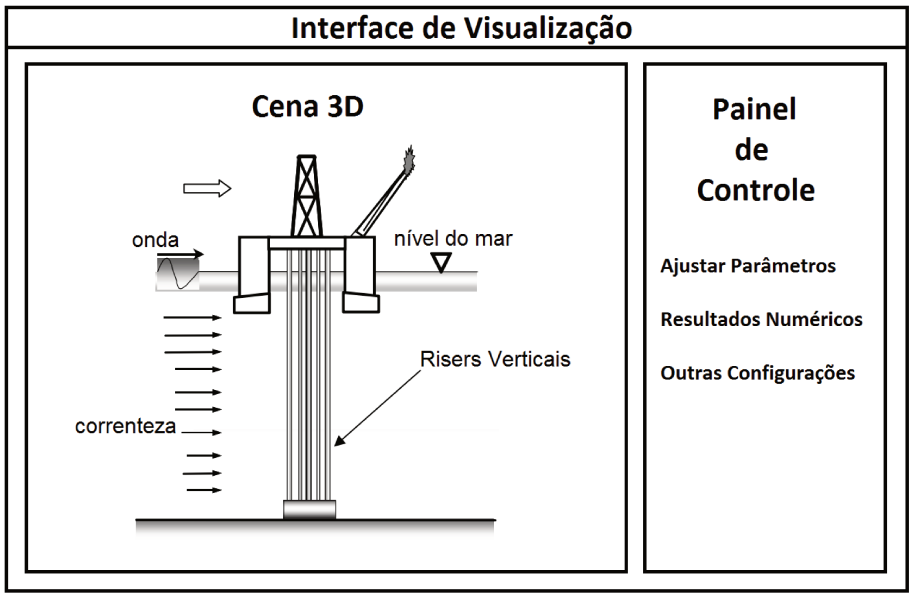

Figura 3.5: Representação por esquemática da interface de visualização, dividida em Cena 3D e Painel de Controle.

3.2.3 Interface de Usuário

A interface de usuário é responsável por realizar duas funções: apresentar a visualização e receber comandos do usuário. Partindo desta consideração, a interface está dividida em dois elementos: uma cena 3D e um painel de controle (ver esquema da Figura 3.5). Na cena 3D, é exibida a visualização da simulação, que pode ser ampliada, transladada e rotacionada pelo usuário. Já no painel de controle, o principal propósito é permitir ao usuário o ajuste dos parâmetros da simulação, embora também outras configurações do aplicativo possam ser alteradas via painel. O grande número de parâmetros, uns quantitativos outros qualitativos, reforçou a escolha pelo painel de controle como interface para ajuste de parâmetros. Uma possível alternativa, em que a interação com os parâmetros fosse de alguma maneira realizada na cena 3D (arraste e clique p.e.) poderia sobrecarregar a cena com muita informação, além do fato de que nem todos os parâmetros possuem representação visual intuitiva e em algum momento, uma representação textual ou numérica seria necessária. Um painel de controle prove uma melhor organização e os *widgets* são um recurso já bem estabelecido.

3.2.4 Implementando o Módulo de Visualização

Para implementar o Módulo de Visualização, foi escolhida a linguagem *Python* e os *frameworks Qt* para janelas e *widgets* e *VTK* para visualização. O *VTK* foi escolhido por implementar diversas técnicas de visualização e ser amplamente utilizado pela comunidade de visualização científica. A escolha pelo *VTK* acabou determinando em parte a escolha das outras ferramentas. O *VTK* pode ser utilizado com *Python* e apresentou um bom desempenho na visualização, alcançando taxas de mais de 30 quadros por segundo na animação. Além disso, *Python* é uma linguagem que acelera em muito o processo de programação, especialmente se comparada a *C/C++*. Por fim, o *Qt*, um *framework* para desenvolvimento de interfaces gráficas

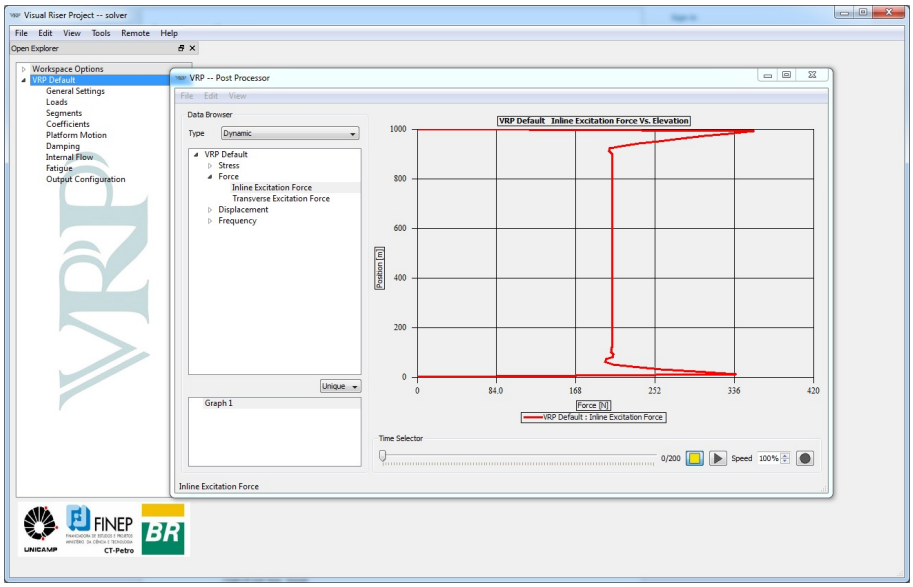

Figura 3.6: Janelas do *VRP*, interface para o *RiserProd* existente previamente a este trabalho.

de usuário bastante difundido, possui integração tanto com *Python* quanto com o *VTK* e é de fácil utilização, possuindo um ambiente gráfico para edição dos *widgets*.

3.2.5 VRP: um Módulo de Visualização já existente para o RiserProd

Já existia previamente a este trabalho um Módulo de Visualização para o *RiserProd* original, o *VRP* (*DutoRisk Manual de Usuário (Interface e Solver)* 2009), citado previamente na Seção 2.3 do Capítulo 2. O módulo consiste de uma interface gráfica organizada em janelas e botões através da qual o usuário pode iniciar uma simulação e analisar os resultados. Para a simulação, o usuário pode escolher qual *Solver* deseja utilizar, um deles sendo o *RiserProd*. Como não é uma ambiente interativo em tempo-real, o usuário primeiramente deve entrar com os parâmetros, executar a simulação para todo o intervalo de estudo, e só então visualizar os resultados. A visualização é realizada através de tabelas e *plots* 2D. Na Figura 3.6 é mostrada uma imagem da interface. Como tanto o *VRP* quanto o presente trabalho utilizam o mesmo *Solver*, o *VRP* mostra-se apropriado como base de comparação para o Módulo de Visualização aqui proposto. Tal comparação é efetuada na avaliação de usabilidade, discutida na Subseção 3.2.6 a seguir.

3.2.6 Avaliando a Visualização

Foi afirmado na Seção 1.1 do Capítulo 1 que um simulador é util no projeto, análise e diagnóstico de *risers*, e que recursos de visualização e interação em tempo-real contribuem para uma compreensão mais intuitiva da simulação; espera-se, portanto, que o ambiente desenvolvido apresente estas qualidades. Baseado nestas características, também acredita-se que o ambiente se destaque mais em permitir uma visão global da simulação, sintetizando seus diversos aspectos, do que na visualização de dados individuais.

Como forma de avaliar a presença ou não destas características, foi elaborado um teste para ser realizado com usuários que tenham experiência com *risers*. O teste consiste em colocar à disposição o ambiente de visualização, bem como instruções explicando o seu propósito e a forma de uso, e pedir para que o usuário utilize o programa e responda um questionário. A forma como o usuário deve utilizar o programa é livre, sendo pedido apenas que execute uma ou mais simulações e explore as diversas características do *software*. As questões cobrem diversos aspectos, abrangendo perguntas sobre a utilidade do ambiente para os diferentes propósitos de projeto, análise e diagnóstico de risers, sobre a qualidade da animação e da interatividade, e se a visualização da simulação é intuitiva, entre outras. Também são feitas algumas perguntas comparando o ambiente desenvolvido com o *VRP* (mencionado na Subseção 3.2.5). As questões devem ser respondidas dando notas de acordo com a escala de Likert, isto é, de 1 a 5, sendo 1 a pior e 5 a melhor nota. O voluntário pode adicionalmente comentar a nota dada no campo Justicativa, sendo isto opcional. O questionário aplicado encontra-se no Apêndice A.

3.3 Integrando Solver e Módulo de Visualização

Por fim, é necessário integrar o *Solver* e o Módulo de Visualização de forma a comporem uma única aplicação. A abordagem escolhida foi encapsular o *Solver* como módulo *Python* utilizando o *SWIG* e passá-lo ao Módulo de Visualização, onde poderá ser utilizado como um módulo *Python*. O *SWIG* é uma ferramenta muito utilizada (por exemplo, pelo Google) para realizar de forma automatizada o encapsulamento e *binding* de programas em *C/C++* com outras linguagens como *Python*, *Ruby* e *PHP*. Para tanto, deve-se escrever em um arquivo de extensão *.i* quais as classes e funções *C/C++* deverão estar disponíveis para uso pela outra linguagem. Executando o *SWIG* com este arquivo de entrada, um *header C/C++ .h* e um arquivo *Python .py* são gerados. Adicionando o *header* ao código *C/C++* e compilando como uma biblioteca, é gerado um arquivo que juntamente ao arquivo *.py* gerado anteriormente, permite que a biblioteca seja importada como um módulo *Python*.

Assim, os métodos do *Solver* podem ser invocados dentro do código *Python* e também os dados do *Solver* podem ser lidos e escritos. Uma única complicação existente é quanto ao acesso de dados vetoriais, já que no *C/C++* o vetor é um ponteiro. A solução para este caso é implementar no próprio arquivo *.i getters* e *setters* apropriados para esses dados, que dado o índice retornam o valor do elemento naquela posição do vetor.

3.4 Considerações Finais

Neste capítulo, foram mostrados os diversos aspectos necessários no projeto e na implementação do ambiente de visualização e simulação. Foi visto como o *software* do ambiente foi estruturado em dois módulos, um responsável pela simulação, o *Solver*, e um responsável pela visualização. Da parte do *Solver*, após expor os fundamentos matemáticos da simulação de riser e os requisitos necessários para operar com interação em tempo-real, foi explicado como o RiserProd, um *Solver* que já implementava o modelo matemático descrito, pôde ser adaptado para o uso em tempo-real. Foi também descrito o método utilizado para avaliar o desempenho

do *Solver* desenvolvido. Quanto ao Módulo de Visualização, além de discutir o propósito e os fundamentos da visualização científica, foram indicadas as técnicas escolhidas para visualizar a simulação de *riser*, bem como o *design* da interface de usuário desenvolvida para tal e as ferramentas utilizadas na implementação do módulo. Além disso, foi explicado o método proposto para avaliar a usabilidade do ambiente. Por fim, foi mostrado como os dois módulos que compõem o ambiente de visualização e simulação puderam ser integrados em um único aplicativo. No capítulo seguinte são mostrados e discutidos os resultados obtidos a partir dos métodos descritos aqui.

Capítulo 4

Resultados

O ambiente de simulação e visualização de *risers* foi implementado e avaliado conforme especificado no Capítulo 3. No presente capítulo, o protótipo desenvolvido é apresentado, sendo descritas suas principais características. Também uma visão geral do código do protótipo é mostrada. Maiores detalhes do código podem ser consultados na documentação apresentada no Apêndice B. Por fim, são apresentados e discutidos os resultados das análises de desempenho e usabilidade.

4.1 Protótipo Desenvolvido

O protótipo foi desenvolvido conforme especificado no Capítulo 3. Na Figura 4.1 é mostrada a janela principal da aplicação, que é constituída de duas partes principais: à esquerda, a visualização em gráficos tridimensionais e à direita o painel de controle. Adicionalmente, há também uma barra de menu na parte superior da janela.

Para iniciar uma simulação, o usuário deve primeiro determinar a configuração do *riser*, na opção *New->Model*, na barra de menu (Figura 4.2). Em seguida, o número de elementos do modelo de elementos finitos da simulação deve ser especificado. Feito isto, um conjunto padrão de parâmetros é estabelecido e a simulação está pronta para ser executada e visualizada. Através de um botão *Play*, o usuário pode iniciar ou pausar a simulação. Com o botão *Reset*, o *riser* é recolocado em sua posição inicial, reiniciando a simulação com o mesmo conjunto de parâmetros. Os parâmetros podem ser alterados a qualquer momento no painel de controle, na aba *Parameters*. Múltiplos parâmetros podem ser alterados de uma vez, mas é necessário ativar o botão *Apply* para indicar que os novos parâmetros podem ser submetidos. Também no painel de controle outras opções podem ser ajustadas.

A visualização da simulação reproduz em computação gráfica o cenário de operação de um *riser*, onde o usuário pode vê-lo movimentando-se sob o efeito das cargas ambientais. Sendo uma cena 3D, é possível aplicar transformações de rotação, translação e escala, como na Figura 4.3. Na cena, além do *riser*, são representados as boias, a embarcação, a superfície e o fundo do mar. Como o objetivo do sistema é prover uma visão geral e intuitiva da simulação, a relação de escalas entre alguns elementos, como entre o *riser* e a embarcação, não segue a da realidade, pois poderia dificultar a visualização. Também a passagem do tempo na visualização não é

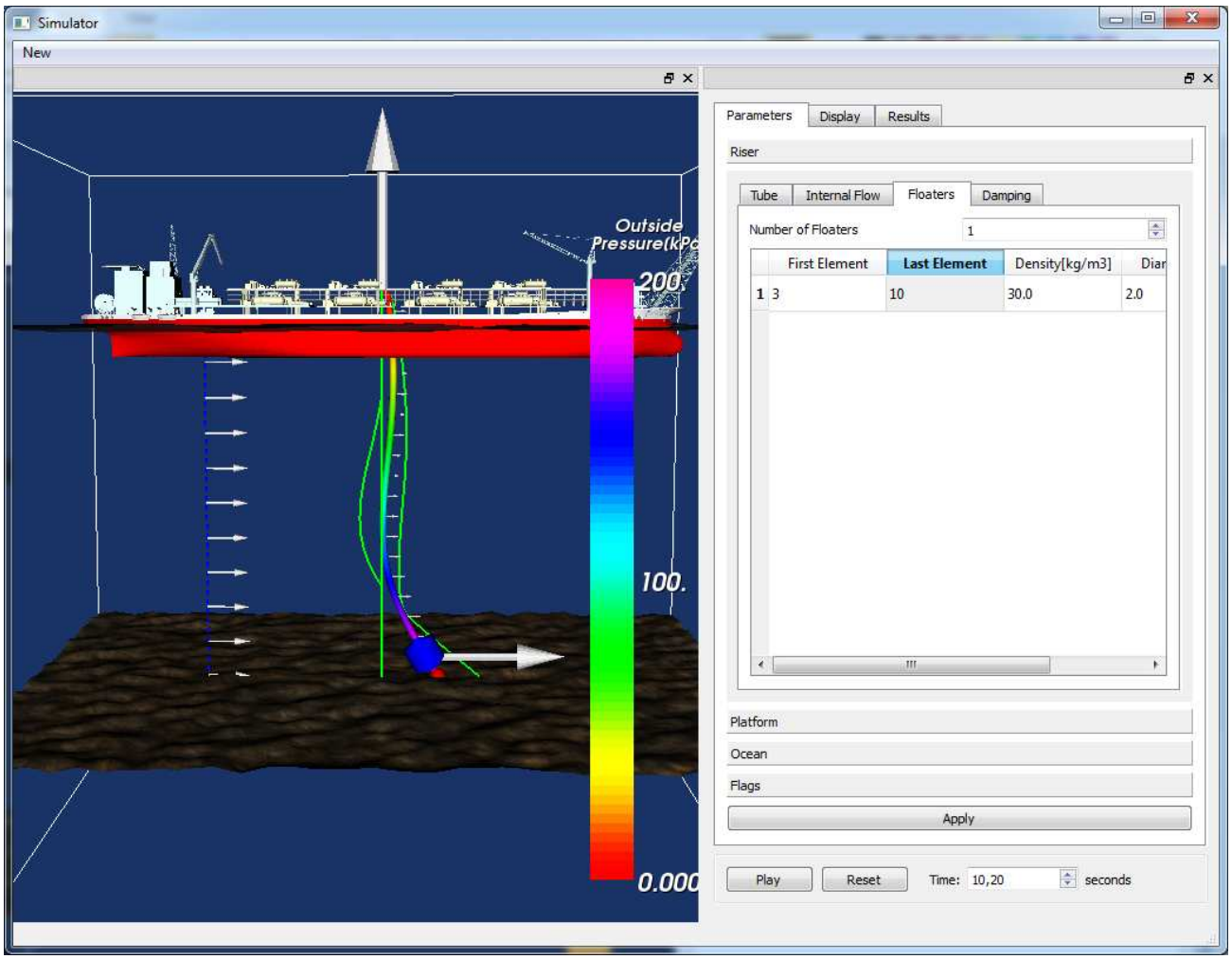

Figura 4.1: Janela principal do ambiente de visualização.

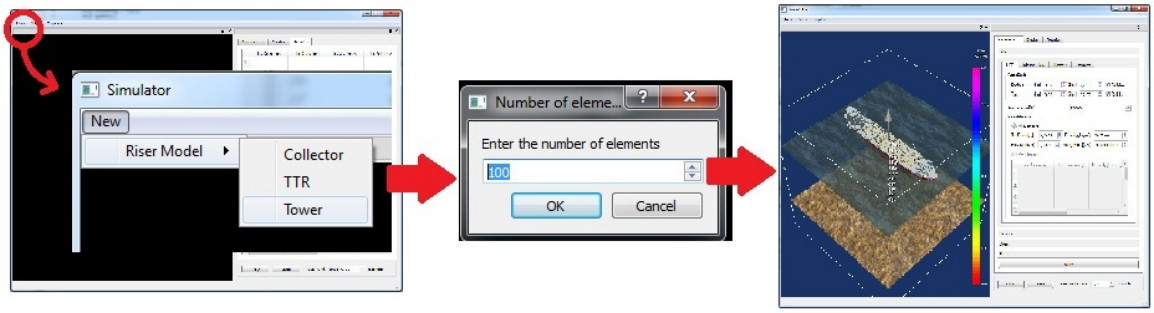

Figura 4.2: Iniciando uma simulação: escolhe-se uma configuração de *riser* e o número de elementos do modelo e a simulação está pronta para ser visualizada.

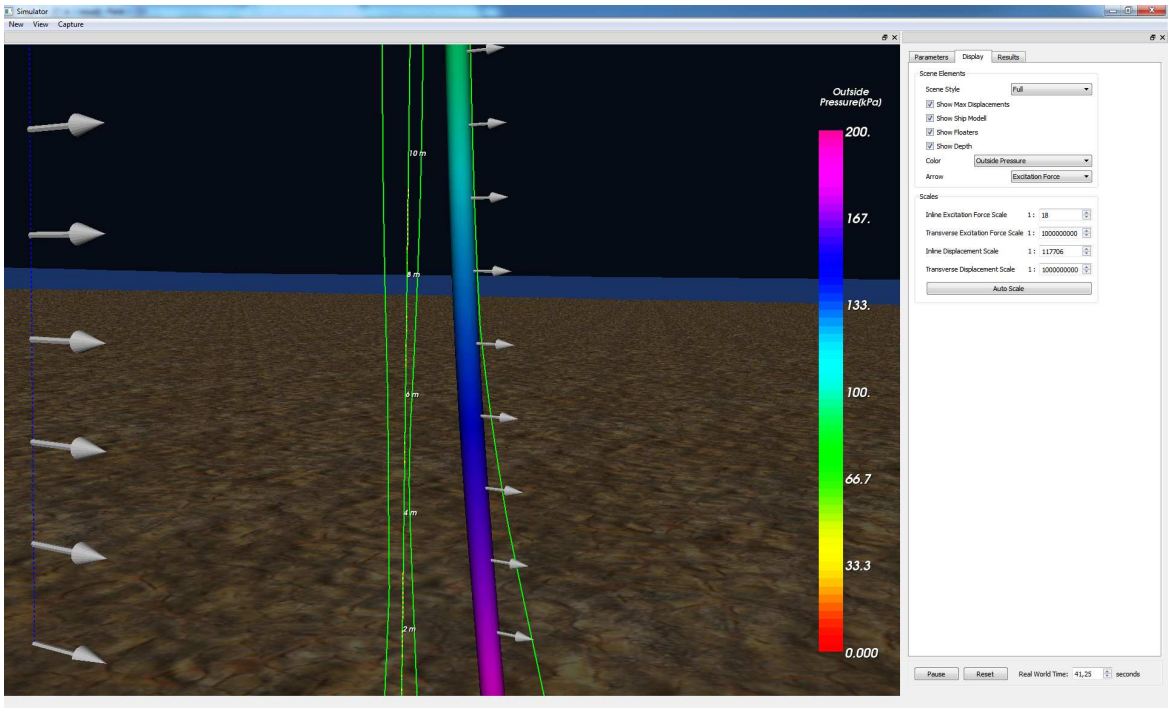

Figura 4.3: É possivel explorar o ambiente virtual aplicando rotação, translação e escala.

necessariamente igual a da simulação. São visualizados também os diversos dados da simulação, tanto parâmetros quanto resultados. Um conjunto de setas sobre o *riser* representa as forças de excitação, enquanto um outro conjunto de setas dispostas um pouco afastadas do *riser* mostra o perfil de correnteza marítima. Os máximos de deslocamento nos planos *inline* e transversal são marcados com linhas verdes. Por fim, cores são utilizadas sobre a parede do *riser* para representar diferentes dados da simulação, como coeficiente de arrasto ou pressão interna.

Alguns aspectos da visualização podem ser ajustados na aba *Display* do painel de controle, como qual a grandeza a ser representada por cores. Também nesta aba é possível ajustar as escalas do deslocamento e forças sobre o *riser*, já que estes podem ser muito pequenos para serem visualizados apropriadamente. Para facilitar este processo, através do botão *Auto Scale* é escolhido automaticamente um conjunto de escalas adequado. Objetos da visualização, como as linhas de máximo deslocamento ou o modelo da embarcação, podem ser desabilitados para deixar a visualização menos carregada de elementos visuais. Por fim, para aumentar o realismo e a imersão, é possível optar por um estilo de visualização como o da Figura 4.4, com o mar estendendo-se bem além da embarcação (e não contido por um *bounding box*), e também ativar o uso de visão estéreo tridimensional.

Além de visualizar os dados no ambiente de realidade virtual, é possível também inspecionar os dados na forma numérica, na aba *Results* do painel de controle. Nela, uma tabela com os dados de deslocamentos, velocidades e forças produzidos pela simulação são exibidos para cada elemento, divididos nas componentes *inline* e transversal. Clicando-se em um dos elementos da tabela, uma marcação é colocada sobre o *riser* no ambiente virtual, na posição do elemento correspondente. Inversamente, ao colocar o *mouse* sobre o riser e pressionar a tecla *P*, o elemento daquela posição é realçado na tabela (Figura 4.5.

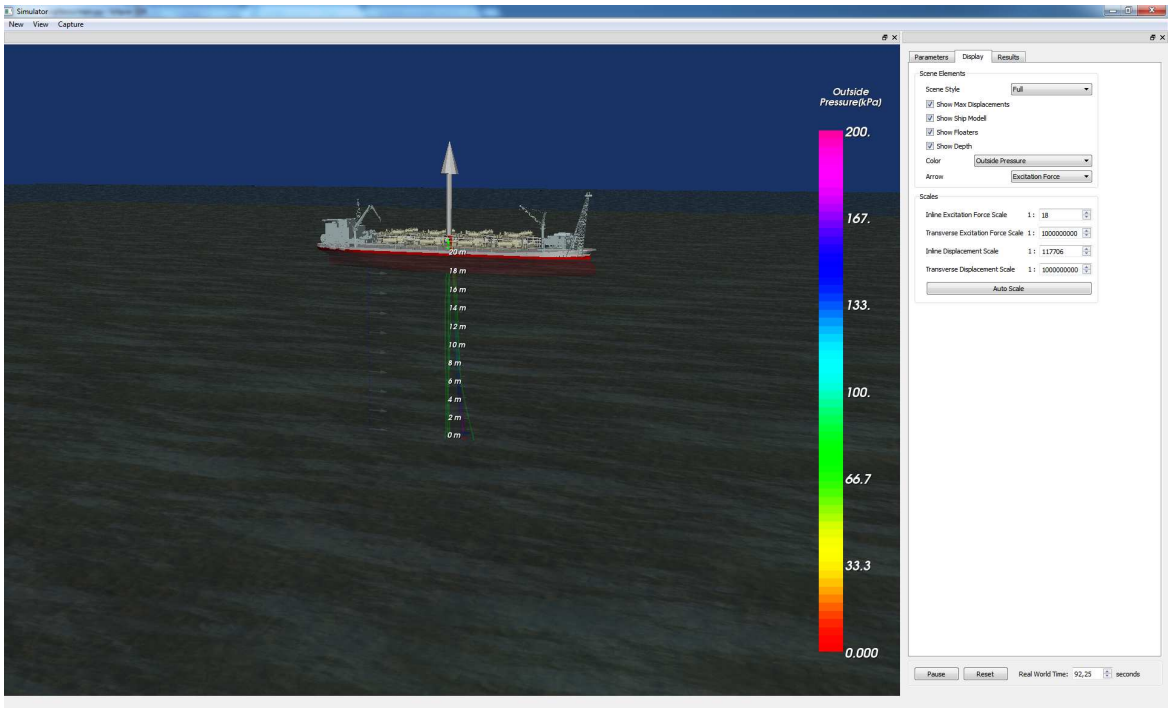

Figura 4.4: Para maior realismo e imersão, é possível optar por uma visualização com o mar extendendo-se até o horizonte, bem como utilizar visão stereo tridimensional.

O *Solver* do protótipo desenvolvido reproduz exatamente os cálculos do simulador original *RiserProd*. A validação dos resultados do novo simulador foi feita por comparação de alguns casos de simulação executados tanto no *Solver* desenvolvido quanto no *RiserProd*. Uma validação mais exaustiva e formal dos resultados do novo *Solver* porém não foi realizada neste trabalho.

4.2 Visão Geral do Código

Uma descrição geral do código do Protótipo desenvolvido é aqui apresentada. Maiores detalhes podem ser consultados na documentação apresentada no Apêndice B. Seguindo a arquitetura descrita no Capítulo 3, o código encontra-se dividido em dois grandes módulos: o código do *Solver* em C++ e o do Módulo de Visualização em Python. Os módulos operam em conjunto formando uma única aplicação, sendo os métodos do *Solver* invocados pelo Módulo de Visualização. Na Figura 4.6, é mostrado um diagrama com as classes principais do *Solver*, e na Figura 4.8, um diagrama das classes do Módulo de Visualização.

A simulação realizada pelo *Solver* é consituída de três análises, as Análises Estática, de Vibrações Livres e Dinâmica. Cada uma destas análises encontra-se implementada como uma classe no código do *Solver*, respectivamente nomeadas de *StaticAnalysis*, *FreeVibrationAnalysis* e *DynamicAnalysis*. Funcionando como uma *engine*, cada uma possui um método *run*, que recebendo como parâmetro um objeto do tipo *Riser*, realiza todos os cálculos correspondentes a determinada análise e armazena no objeto fornecido os resultados produzidos. O tipo *Riser* é definido no código do *Solver*, assim como subclasses que definem configurações especificas de *riser*: *TTR*, *Collector* e *Tower*. Fazendo uso de *sobrecarga de método* (*method overloading*), os

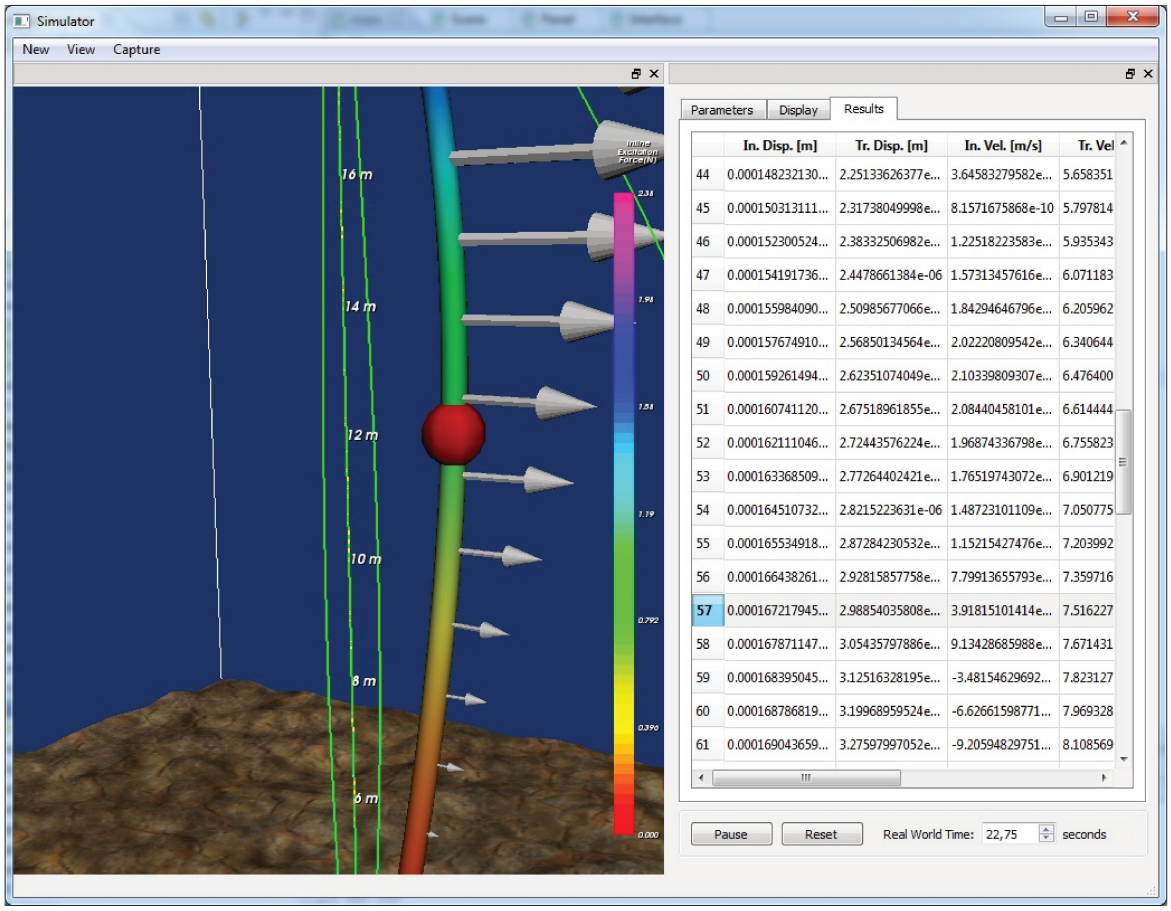

Figura 4.5: Apontando com o *mouse* sobre o *riser* e pressionando o botão *p*, o elemento correspondente é marcado no ambiente de visualização (esfera vermelha) e realçado na tabela da aba *Results*.

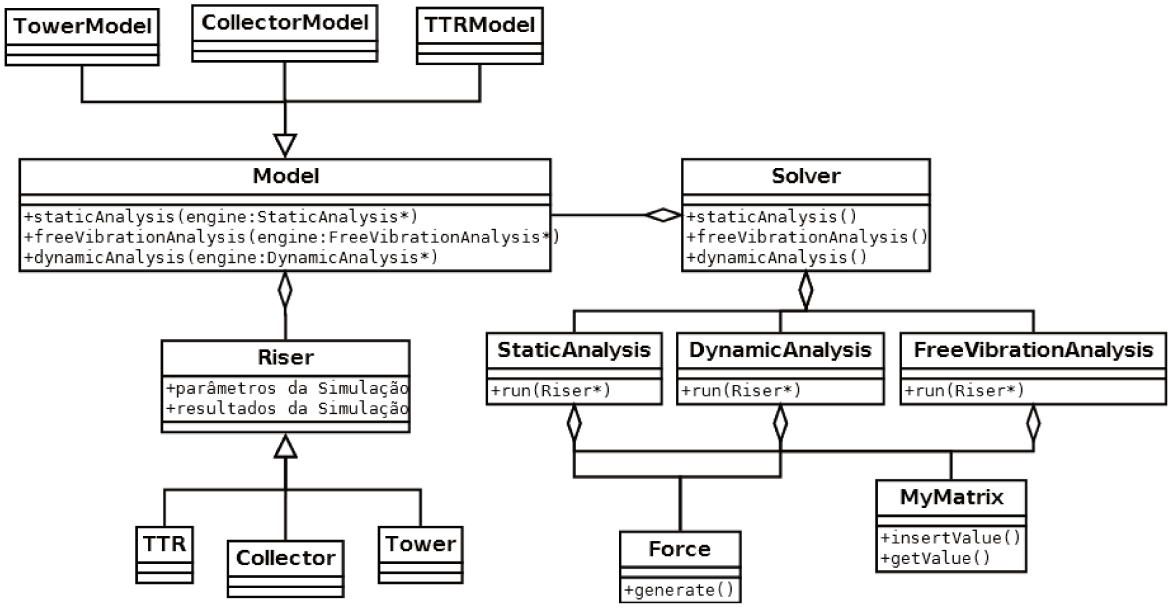

Figura 4.6: Diagrama com as classes principais que compõe o *Solver*. Para não sobrecarregar o diagrama, foram omitidas as subclasses de MyMatrix e Force.

métodos *run* das classes de análise aceitam como parâmetro cada um destes subtipos de *Riser*, realizando operações diferentes conforme o tipo passado.

Nos cálculos realizados pela análises, são calculadas e manipuladas diversas matrizes como as de Massa, Amortecimento e Rigidez, entre outras. Todas estas encontram-se definidas no código do *Solver* como filhas da classe base *MyMatrix*. Estas matrizes foram definidas como classes ao invés de utilizar diretamente vetores (*arrays*) de duas dimensões pois estas matrizes compartilham de algumas características não usuais. Em primeiro lugar, nem todos os elementos das matrizes apresentam valores diferentes de zero, estando estes concentrados em faixas bem definidas da matriz. Assim, de forma a economizar memória, é adotado um procedimento de compressão da matriz, na qual apenas os valores diferentes de zero são armazenados. Como um efeito disso, o acesso a elementos da matriz deixa de ser trivial, dado que é necessário uma conversão dos índices da matriz para a forma comprimida. Na classe base *MyMatrix*, encontram-se métodos que facilitam a interação com a matriz, como *getters* e *setters*. Além disso, algumas matrizes, com a de Massa, Amortecimento e Rigidez (nomeadas no código de MassMatrix, DampingMatrix e StiffnessMatrix, respectivamente), contam com métodos *assemble*, que, recebendo como parâmetro um objeto de uma subclasse de *Riser*, calculam os valores da matriz, preparando-a para ser usada pelas análises. Pelas análises também são utilizadas classes derivadas de Force, que implementam as cargas que atuam sobre o riser.

A classe *Riser* e suas subclasses servem basicamente de contêiner para outros dados da simulação, tanto parâmetros como resultados, e são passadas como argumento para diversos métodos. A classe *Riser* também contém métodos que tem função auxiliar, como calcular um certo dado do *riser* a partir de outros dados.

Finalmente, tem-se a classe *Solver* e a classe *Model*. A classe *Solver* é aquela que agrega as demais, funcionando como a classe principal do *Solver*. Possui, entre outros, os métodos *newModel* (para definir uma configuração de *riser*), *staticAnalysis*, *freeVibrationAnalysis* e *dynamicAnalysis*. A classe contém também entre seus atributos, ponteiros para objetos das classes *StaticAnalysis*, *FreeVibrationAnalysis* e *DynamicAnalysis*, e um ponteiro para um objeto da classe *Model*. Os objetos das análises servem obviamente para realizar as análises do *Solver*. A classe *Model* por sua vez, é uma classe que serve de intermediário entre o *Solver* e a classe *Riser*. A classe *Model* possui, como a classe *Riser*, subclasses para cada configuração de *riser* (*TTRModel*, *CollectorModel*, *TowerModel*). Cada uma possui como atributo um objeto da subclasse de *Riser* apropriada, bem como métodos *staticAnalysis*, *freeVibrationAnalysis* e *dynamicAnalyis*, que recebem como argumento a classe da análise correspondente e chamam por sua vez o método *run* dela passando o objeto do subtipo de *Riser* apropriado. A ideia por trás da aparente complexidade de classes e chamadas de métodos é se valer do polimorfismo e da sobrecarga de método para implementar as diversas análises para as diversas configurações de *riser* sem precisar armazenar ou verificar informações quanto ao tipo de configuração utilizada. Com esta organização, por exemplo, a classe *Solver* mantém apenas um atributo de tipo *Model*, sem precisar saber como qual tipo especifico está lidando (exceto no momento em que um é criado, no método *newModel*). O diagrama da Figura 4.7 ilustra esta lógica, mostrando como se dá o fluxo de chamadas de método ao realizar a análise estática para uma configuração TTR, por exemplo.

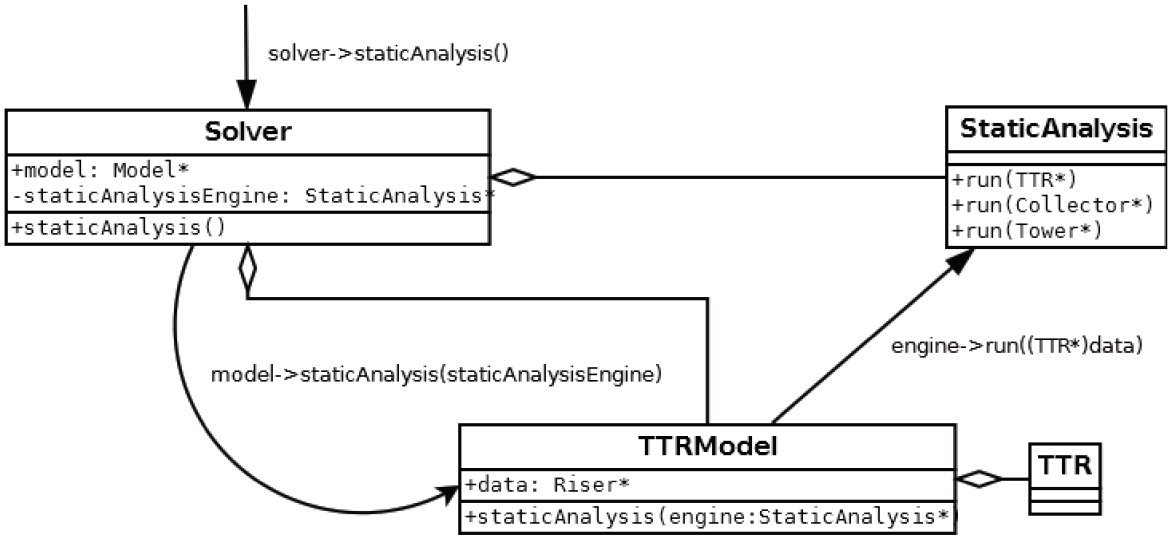

Figura 4.7: Diagrama representando o fluxo de chamadas (indicado por setas) que ocorre ao executar a Análise Estática para uma configuração TTR. O polimorfismo e a sobrecarga de métodos assegura que a análise é executada para a configuração de riser específica.

Para o código do Módulo de Visualização, os objetos correspondem em sua grande maioria aos objetos visuais da interface de visualização, exceto pela classe principal *App*. Tem-se, por exemplo, a classe *Interface*, que implementa a janela principal e seus diversos *widgets*. Esta por sua vez, contém um objeto *Scene*, para a cena 3D onde é feita a visualização propriamente, e *Panel*, onde é implementado o painel de controle. O objeto *Scene* contém ainda outros objetos, representando os elementos da cena, como *Riser* para o *riser*, *Water* para a superfície do mar, *WaterFlow*, para o perfil de correnteza ou *Forces*, para as forças de excitação representadas como setas.

Os métodos destas classes são vários e de propósitos bem específicos, mas todas possuem tanto um método construtor *__init__* quanto um *update*. O método *__init__* é chamado quando o objeto é instanciado e cria e inicializa seus atributos. Para o caso de classes como *Riser* ou *Water* que representam elementos da visualização, o método *__init__* cria os *mappers*, *actors* e demais objetos do *VTK* que compõem o *pipeline* de visualização, enquanto que para a classe *Panel*, o método conecta os diversos *widgets* do painel de controle ao seus respectivos *callback*. O método *update*, por outro lado, atualiza os dados do objeto a medida que a simulação é executada.

Por fim, a classe *App* é a classe que define como métodos as princiapis funcionalidades da aplicação, além de conter uma instância de *Interface* e uma de *Solver*, importado do *C++* via *SWIG*. Entre os métodos oferecidos pela classe tem-se *newModel*, para criar um novo modelo de simulação; *generate*, que invoca as análises estática e de vibrações livres do *Solver* e é chamada sempre que novos parâmetros são passados; *play*, *pause* e *reset*, para controlar a visualização da simulação, e *update*, que invoca um passo da análise dinâmica e chama o método *update* de *Interface*, que chama o *update* de *Scene* e de *Panel*, e assim por diante. A frequência com que o método *update* é invocado é controlado por um relógio disponível como classe do *Qt*.

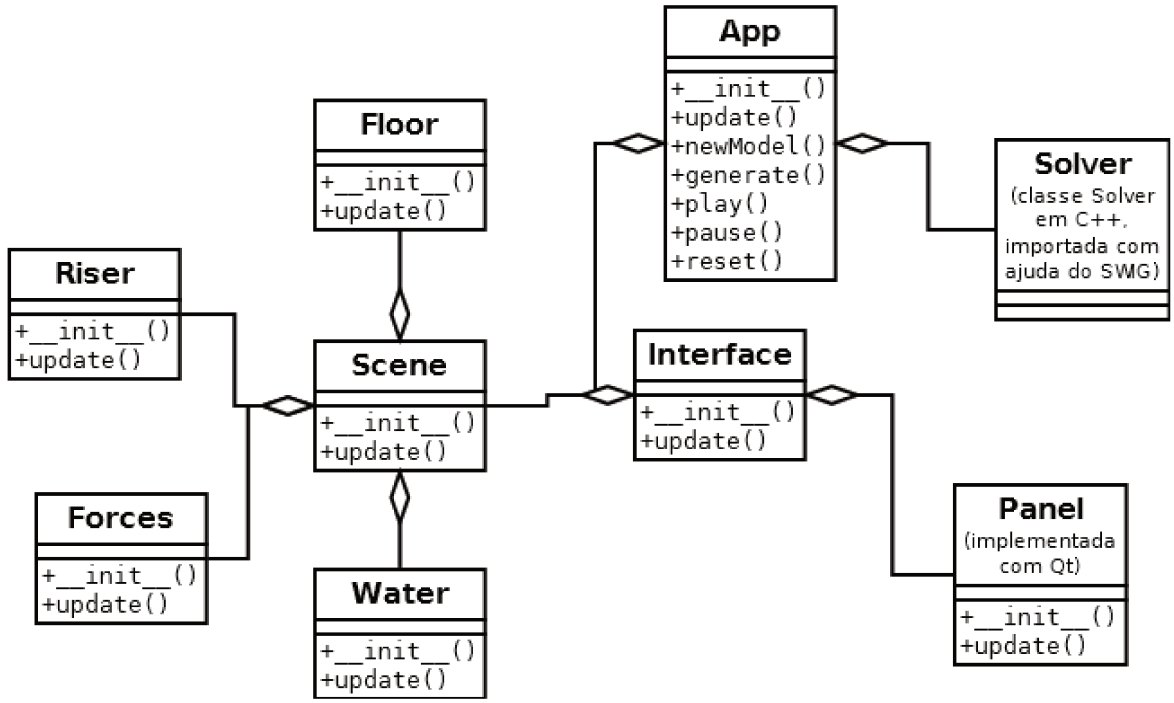

Figura 4.8: Diagrama de classes do Módulo de Visualização. Como Scene contém muitas classes, apenas algumas estão representadas, para não sobrecarregar o diagrama.

4.3 Avaliação do Desempenho do Solver

Os resultados da medida dos tempos em cada uma das quatro etapas em função do número de elementos, de acordo com o método descrito na Seção 3.1.5 do Capítulo 3, são mostrados na Figura 4.9 a seguir. Cada um dos valores apresentados foi obtido como a média de dez amostras medidas e as barras de erro nos gráficos representam o desvio padrão. O número de amostras foi considerado satisfatório devido ao baixo desvio padrão resultante. As medidas foram tomadas ao executar o *Solver* em uma máquina com processador *Intel Core i7 2.80 GHz* e 12GB de memória RAM e sistema operacional *Windows 7* de 64 bits.

A medida dos tempos permite analisar os limites para o uso interativo em tempo-real do *Solver*. Conforme discutido no Capítulo 3, são dois os requisitos que devem ser assegurados para interação em tempo-real. O primeiro é que, ao realizar uma mudança nos parâmetros da simulação, seja pequeno o tempo de espera para se obter o novo comportamento do *riser*. O segundo fator é que a taxa com que os resultados da simulação são calculados deve ser adequada ao passo da animação de 30 quadros por segundo.

O tempo de resposta após a mudança de parâmetros é igual a soma das etapas de Análise Estática, Análise de Vibrações Livres e Montagem de Matrizes. No gráfico da Figura 4.9, fica evidente como a contribuição da Análise Estática para o tempo total é pouco significativa se comparada a de Vibrações Livres e a Montagem de Matrizes. O gráfico da Figura 4.10 mostra o tempo de espera a cada mudança de parâmetros, em função do número de elementos. Considerando o limite de 1 segundo de resposta, o número máximo de elementos suportado é de 150; para 2 segundos, o limite sobe para 200 elementos.

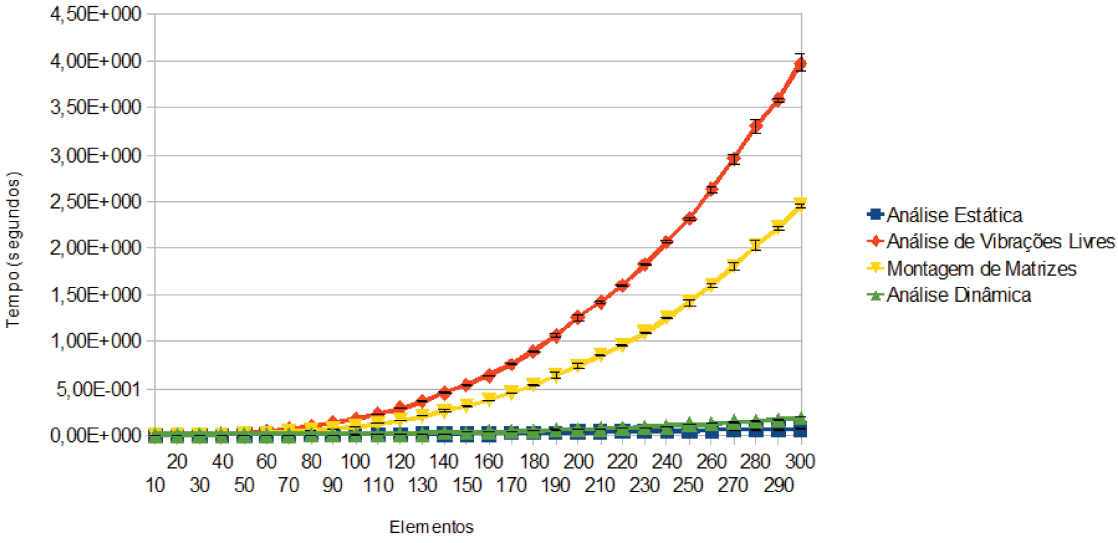

Figura 4.9: Medidas de tempo de execução do *Solver*, tomadas em quatro etapas distintas.

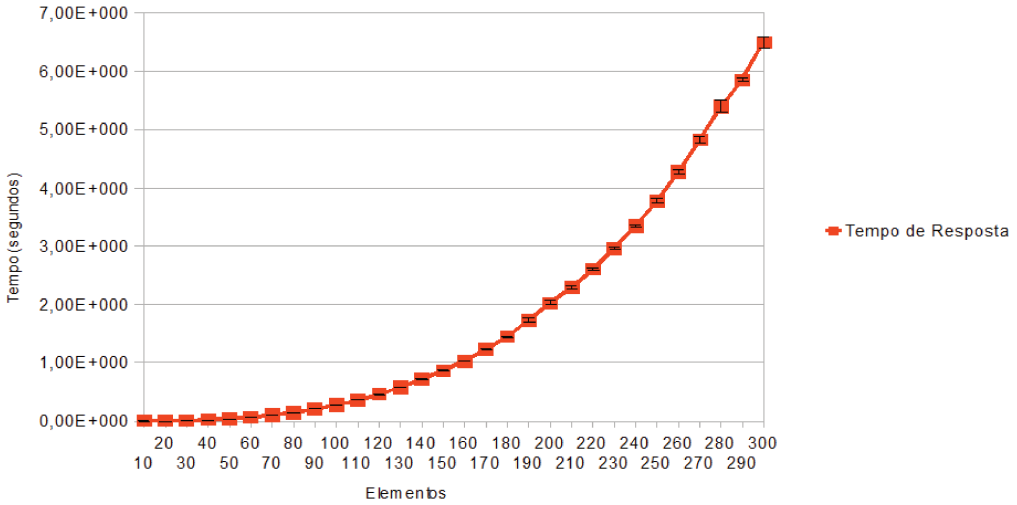

Figura 4.10: Tempo de espera a cada mudança de parâmetros, computado como a soma das três primeiras etapas. A Análise de Vibrações Livres e a Montagem de Matrizes são as que mais contribuem na medida

.

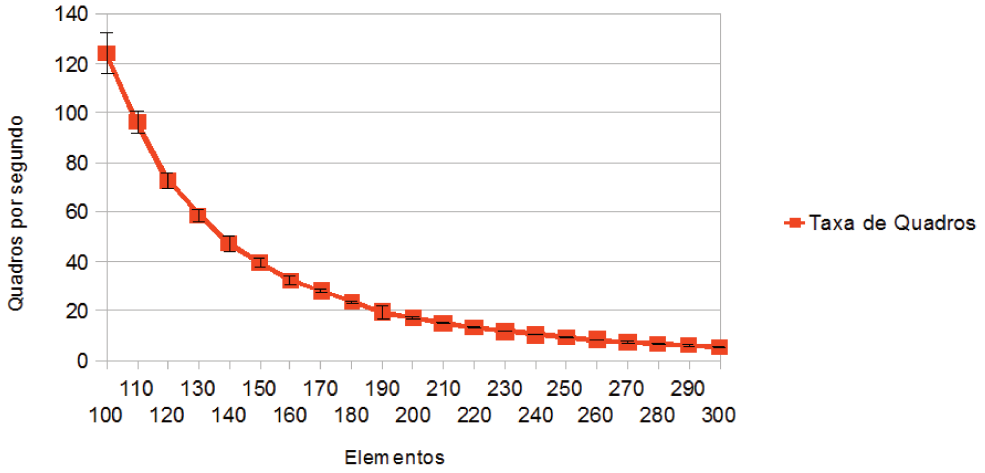

Figura 4.11: Taxa de quadros máxima suportada em função do número de elementos, tomado como o inverso das medidas da Análise Dinâmica.

A taxa com que os resultados são produzidos, por sua vez, está relacionada ao tempo gasto numa execução da Análise Dinâmica. A Figura 4.11 mostra a taxa suportada em função do número de elementos, que é obtido pelo inverso dos tempos medidos da Análise Dinâmica. Para manter uma taxa de 30 quadros por segundo, o número máximo de elementos suportado é de 170.

O *profiling* pela ferramenta do *Visual Studio* aponta os pontos de gargalo no processamento. As funções que consomem mais tempo de processamento segundo o *profiler* são duas, uma que realiza *Decomposição de Gauss* e outra que realiza *inversão de matriz*. A primeira é utilizada na Análise de Vibrações Livres enquanto a segunda é empregada na Montagem de Matrizes, condizendo com as medidas de tempo obtidas para estas etapas (Figura 4.9), que foram bem maiores do que das outras.

4.4 Avaliação da Usabilidade do Ambiente de Visualização e Simulação

A avaliação de usabilidade, descrita na Seção 3.2.6 do Capítulo 3, foi feita com nove usuários. Reconhece-se ser um número pequeno para uma amostra, mas o fato de ser necessário que os voluntários tivessem experiência com *risers* foi um grande limitante para a seleção de pessoal para o teste. Outra limitação foi o fato de apenas dois voluntários terem já utilizado o VRP, de forma que os demais não puderam responder as questões 13 a 19. Mesmo assim, o resultado dos testes revelou algumas informações interessantes.

A Tabela 4.1 mostra o resultado das avaliações realizadas e o gráfico da Figura 4.12 mostra a média das notas para cada questão.

Como existe espaço para subjetividade na hora de atribuir as notas, pois algumas pessoas tendem a evitar notas mais extremas ou evitar dar notas baixas, a média simples poderia não

Perguntas/Voluntários	1	2	3	4	5	6	7	8	9
Categoria Geral:									
1 - Para a finalidade de projeto de *risers*, o ambiente testado é:	5	3	5	3	3	5	-	5	3
2 - Para a finalidade de análise do comportamento de *risers*, o ambiente testado é:	5	4	5	3	3	5	-	5	4
3 - Para a finalidade de diagnóstico de problemas em *risers*, o ambiente testado é:	4	2	5	3	2	4	-	4	3
4 - Para a finalidade de visualizar o comportamento global do *riser*, o ambiente testado é: (	5	4	5	3	4	4	3	4	4
5 - Para a finalidede de visualizar individualmente cada resultado da simulação, o ambiente testado é:	5	2	5	3	4	4	3	4	4
6 - Quanto à facilidade de uso, o ambiente testado é:	4	5	4	4	5	5	4	5	4
7 - Quanto à aparência do ambiente testado, ela é:	5	5	4	4	4	5	4	5	4
Categoria Visualização:									
8 - Quanto à representação da simulação na forma de um ambiente de realidade virtual, ela é:	4	4	5	4	5	5	3	4	3
9 - Quanto à qualidade da animação (fluidez), ela é:	4	4	5	4	5	5	-	5	4
Categoria Interativade:									
10 - Quanto à interatividade, ela é:	5	5	4	4	5	5	4	5	4
11 - Quanto ao tempo de resposta ao mudar parâmetros, ele é:)	5	5	4	4	4	5	4	5	5
12 - Quanto à organização do painel de controle, ela é:	4	5	4	4	5	4	4	5	4
Categoria Comparação com o VRP:									
13 - Para a finalidade de projeto de *risers*, o ambiente testado é: [comparado ao VRP]	-	-	5	-	-	5	-	-	-
14 - Para a finalidade de análise do comportamento de *risers*, o ambiente testado é: [comparado ao VRP]	-	-	3	-	-	4	-	-	-
15 - Para a finalidade de diagnóstico de problemas em *risers*, o ambiente testado é: [comparado ao VRP]	-	-	3	-	-	4	-	-	-
16 - Para a finalidade de visualizar o comportamento global do *riser*, o ambiente testado é: [comparado ao VRP]	-	-	3	-	-	5	-	-	-
17 - Para a finalidede de visualizar individualmente cada resultado da simulação, o ambiente testado é: [comparado ao VRP]	-	-	3	-	-	4	-	-	-
18 - Quanto à facilidade de uso, o ambiente testado é: [comparado ao VRP]	-	-	5	-	-	5	-	-	-
19 - Quanto à aparência do ambiente testado, ela é: [comparado ao VRP]	-	-	5	-	-	5	-	-	-

Tabela 4.1: Notas dadas pelos voluntários para cada pergunta.

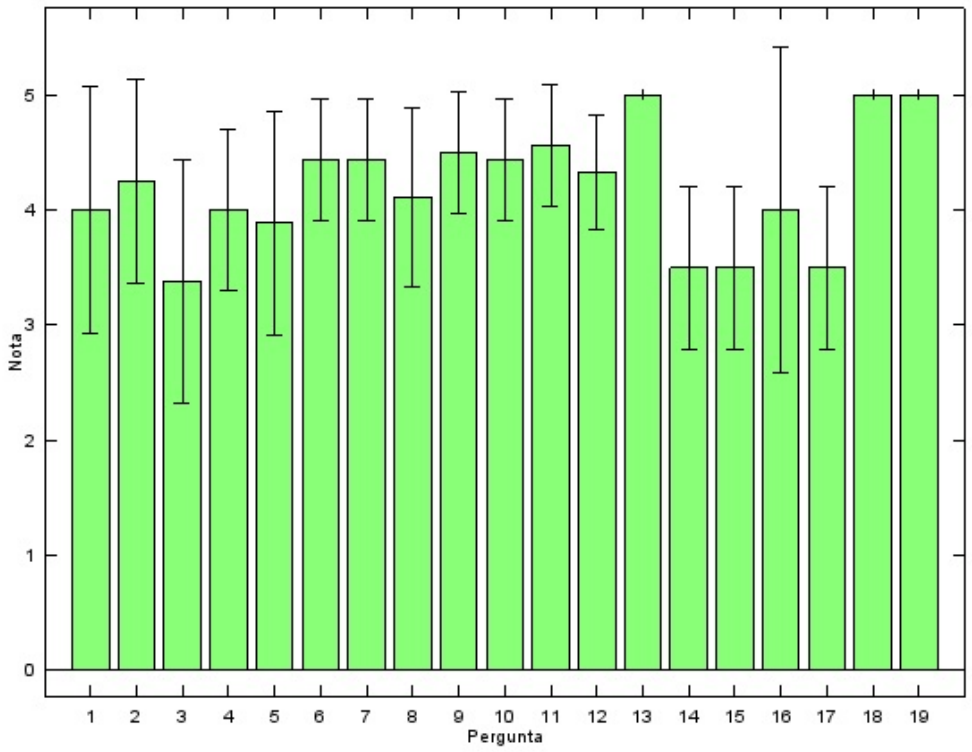

Figura 4.12: Nota média e desvio padrão de cada pergunta.

representar adequadamente os resultados. Assim, as notas dadas por cada voluntário foram divididas pela nota média dada por ele e então calculadas as médias das notas normalizadas para cada questão (Figura 4.13).

É fácil notar que não há muita disparidade nos gráficos das Figuras 4.12 e 4.13 (quanto a proporção entre as notas) e ambos apontam as mesmas conclusões. O ambiente desenvolvido aparenta ser mais apto para uso no projeto e, especialmente, na análise de *risers* do que para diagnóstico de problemas. Um dos voluntários apontou no campo Justificativa para este quesito (o da questão 3) que o ambiente *'não leva em consideração critérios de falha e segurança do riser, sendo que o usuário deve inspecionar visualmente o resultado'*. Para a finalidade de visualizar a simulação como um todo, a nota média foi maior do que para a visualização de resultados individuais, porém a diferença foi pequena, contrariando o esperado (de que fosse claramente melhor para visualização global da simulação), conforme discutido na Subseção 3.2.2 do Capítulo 3. Alguns usuários destacaram que seria desejável representar os resultados também em gráficos 2D ou permitir que sejam salvos em arquivos para serem utilizados externamente. As notas de 6 a 12 mostram que o ambiente de fato apresenta os diferenciais de visualização intuitiva e interativa em tempo-real e é de fácil uso. Nestes quesitos os comentários em geral consistiam de sugestões de novas funcionalidades, como exibir uma janela com dados de um elemento do *riser* ao clicar nele. Já as notas de 13 a 19, devido a baixa quantidade de amostras, impedem uma conclusão mais sólida, mas trazem ainda alguma informação. Os resultado para estes tópicos apontam para a facilidade de uso do programa, mostrando-o como melhor que a do VRP, além de indicar que o programa também é melhor que o VRP para projeto de *risers*, sendo equivalente

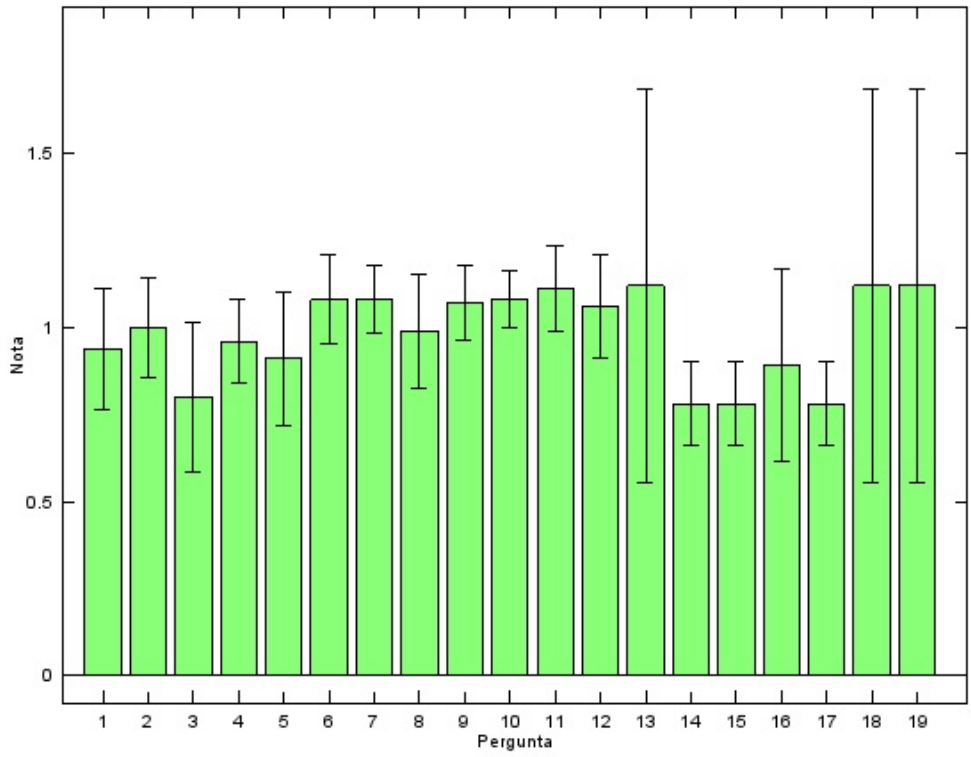

Figura 4.13: Notas média e desvio padrão de cada pergunta após normalização.

ao VRP nos demais aspectos.

Finalmente, tem-se que o ambiente teve notas médias nas perguntas sempre acima de três (considerado aqui o mínimo aceitável), com a média destas notas sendo de 4,2 num máximo de 5, o que sugere, pelo menos baseado na opinião dos voluntários, a boa qualidade do ambiente desenvolvido.

4.5 Considerações Finais

Foram discutidos neste capítulo os resultados obtidos pela aplicação dos procedimentos descritos no Capítulo 3. Em primeiro lugar, foi apresentado o protótipo desenvolvido, descrevendo e explicando suas principais características e funcionalidades. Então, foi fornecida uma visão geral do código do protótipo, apresentando as principais classes e como elas se relacionam entre si. Finalmente, foram mostrados os resultados das avaliações de desempenho e de usabilidade. A avaliação de desempenho revelou que o *Solver* suporta interação em tempo-real com uma taxa de animação de 30 quadros por segundo para até 170 elementos. A avaliação de usabilidade sugere que, segundo os usuários, o ambiente desenvolvido é de fácil uso, apresentando visualização intuitiva e boa interatividade em tempo-real. Além disso, foi revelado que o ambiente de visualização é mais adequado para projeto e análise do comportamento de *risers* do que para diagnóstico de problemas, e que é apenas ligeiramente melhor para visualizar o comportamento global do *riser* do que para visualizar características específicas.

Capítulo 5

Conclusões

Neste trabalho, foi proposto e desenvolvido um ambiente virtual para simulação e visualização de *risers* rígidos verticais. O *riser* é uma importante estrutura utilizada na perfuração e exploração de poços de petróleo *offshore*, sendo um duto que liga o poço à unidade de produção, e que é responsável, entre outras coisas, por guiar a broca de perfuração ou escoar o petróleo e o gás extraídos do poço. Devido às condições ambientais como ondas, correnteza e vórtices induzidos sobre o duto, o *riser* sofre o efeito de cargas que diminuem sua vida útil. Simulação por computador é uma ferramenta importante, seja para prever possíveis problemas como para diagnosticar aqueles que já ocorrem. A visualização científica é também de grande valia, pois permite representar o grande volume de dados numéricos produzidos por simulações em uma forma visual de mais fácil compreensão. O ambiente apresentado suporta a visualização de *risers* com o diferencial de permitir a interação em tempo-real com a simulação. No sistema desenvolvido, o usuário pode alterar dados de simulação, tais como o perfil de correnteza ou a elasticidade do material do *riser*, e visualizar o novo comportamento imediatamente, sem precisar interromper a visualização ou esperar toda a simulação ser recomputada. Nos outros sistemas de visualização de *riser* identificados na literatura, é preciso primeiro realizar toda a simulação para só então visualizá-la, o que compromete a fluidez da interação, especialmente em casos onde o usuário deseja explorar sequencialmente diferentes conjuntos de parâmetros.

O sistema foi desenvolvido como dois módulos que interagem compondo uma única aplicação: o *Solver*, módulo responsável pela simulação, e o módulo de visualização. O *Solver* segue o modelo matemático de Ferrari e Bearman para a simulação, que utiliza uma abordagem de Elementos Finitos, tendo sido aperfeiçoado em trabalhos posteriores e culminado no software *RiserProd*. Como o *RiserProd* original não está apto para uso com interação em tempo-real, foi necessário aprimorá-lo para tal uso. Os resultados do novo simulador foram validados comparando-os com os do *RiserProd* original, para alguns casos de estudo. A visualização da simulação é feita através de uma representação virtual do *riser* em seu cenário de operação. O tubo, o mar e a embarcação são representados com auxílio de técnicas de computação gráfica 3D, bem como correnteza, forças e pressões sobre o *riser* e outros dados da simulação, que são visualizados por meio de setas ou cores no ambiente virtual. Parâmetros da simulação e outros ajustes do programa podem ser alterados através de um painel de controle.

Além de desenvolver o ambiente de simulação e visualização, foram realizadas também ava-

liações de desempenho e de usabilidade. A avaliação de desempenho teve como objetivo medir a capacidade do *Solver* suportar interatividade em tempo-real, bem como identificar os gargalos no processamento. A avaliação de usabilidade por sua vez, visou identificar se o ambiente desenvolvido atende àquilo a que se propõe, isto é, de ser uma ferramenta útil ao engenheiro que trabalha com *risers*, apresentando os diferenciais de visualização intuitiva e interatividade em tempo-real.

Após detalhar os métodos utilizados para desenvolver o ambiente e avaliá-lo, foram mostrados e discutidos os resultados obtidos. Após apresentar o protótipo implementado e explicar seu funcionamento, uma visão geral do código foi descrita, indicando as principais classes e métodos e como elas se relacionam. Foram apresentados também os resultados das avalições realizadas. A avaliação de desempenho revelou que o *Solver* em seu atual estado é capaz de suportar uma interatividade em tempo-real para até 170 elementos para uma taxa de 30 quadros por segundo para a animação. A avaliação de desempenho indica também que as etapas de maior custo são aquelas associadas a inversão de matrizes. A avaliação de usabilidade por sua vez sugere que o ambiente desenvolvido oferece uma representação visual intuitiva do comportamento do *riser* e com interativadade em tempo-real satisfatória. Revelou também que o ambiente é mais adequado para o projeto e análise do comportamento de *risers* do que para diagnóstico de problemas.

Assim, são contribuições deste trabalho:

- O desenvolvimento de um módulo de simulação numérica (*Solver*) apto ao uso interativo em tempo-real, a partir de uma adaptação do simulador *RiserProd*, que não é capaz de ser utilizado para tal tipo de interação.
- O desenvolvimento de um módulo de visualização, responsável por representar visualmente os dados do módulo de simulação e de gerenciar a interação com o usuário, através de uma interface que permite ajuste dos parâmetros da simulação.
- A integração dos referidos módulos em um único ambiente de simulação e visualização de *risers*.
- Uma avaliação do desempenho do ambiente, revelando quais os limitantes para o uso interativo em tempo-real do ambiente desenvolvido.
- Uma avaliação da usabilidade do ambiente.

5.1 Limitações

Este trabalho possui, naturalmente, algumas limitações, que não puderam ser superadas por questões de escopo do projeto ou pelo tempo e volume de trabalho adicionais que demandariam.

Uma das limitações deste trabalho é o número máximo de elementos do modelo de simulação em que é possivel obter uma visualização interativa em tempo-real. A avaliação de desempenho mostrou que o limite máximo é de 170, acima do qual a fluidêz da interação passa a ser comprometida. A princípio, seria desejável alcançar um número da ordem de 1000 elementos, o

que proporcionaria uma simulação precisão adequada. Para alcançar este número, são necessárias otimizações no *Solver*, especialmente nos gargalos que limitam melhores performances. A avaliação revelou que estes estão ligados a etapas que realizam inversão de matrizes.

Algumas deficiências também foram reveladas pela avaliação de usabilidade. O ambiente desenvolvido foi apontado como não tão adequado para diagnóstico de problemas pois não apresenta recursos para aplicar os critérios de falha e segurança de *risers*. Alguns usuários apontaram limitações no modelo de simulação, como a impossibilidade de especificar um número de elementos maior para certas regiões do riser, refinando a precisão em áreas críticas. Modificações no modelo da simulação, porém, foram consideradas fora do escopo deste trabalho.

Por fim, pode-se citar também o baixo número de voluntários utilizado na avaliação de usabilidade, fator que torna as análises derivadas dos resultados menos categóricas.

5.2 Trabalhos Futuros

O projeto realizado e as limitações apresentadas sugerem possíveis continuações para este trabalho. Uma das possibilidades é realizar uma validação mais formal e detalhada dos resultados produzidos pelo *Solver* desenvolvido. Outro possibilidade consiste em otimizar o *Solver* de forma a permitir interação em tempo-real com um número maior de elementos. Um número que conferiria precisão adequada seria da ordem de 1000 elementos. Pelas informações reveladas pelo *profiling* do *Solver*, o atual gargalo no processamente são a decomposição de Cholesky e a de Gauss, ambas ligadas a inversão de matrizes, operação tradicionalmente custosa em termos computacionais. Uma abordagem possível para otimizar estas operações seria explorar o paralelismo em *GPU* ou *multicore*, pois diversos cálculos semelhantes são realizados para cada elemento do *riser*, o que sugere uma solução paralela.

Ainda quanto ao *Solver*, alguns voluntários no teste de usabilidade citaram limitações do modelo de simulação, como a impossibilidade de especificar diferentes números de elementos para certas regiões específicas do *riser* (em geral, maior resolução nas extremidades, onde os esforços são críticos). Outra possibilidade é, ao invés de trabalhar para aprimorar o modelo atual, permitir que o módulo de visualização interaja com diferentes *Solvers*. Para isso, além de implementar o módulo de visualização de forma a ser mais flexível, qualquer novo *Solver* deve atender aos requisitos que possibilitem uso em tempo-real, conforme mostrados neste trabalho.

Também para o módulo de visualização, é possível aprimorá-lo acrescentando novos recursos. Alguns foram já sugeridos pelos voluntários do teste de usabilidade, como funcionalidades que apliquem os critérios de segurança de *risers*, ou que mostrem numa janela *popup* os dados de um elemento clicado. Outra possibilidade é oferecer mais opções para controle e representação de escalas no ambiente, tanto na dimensão espacial como temporal. Atualmente, a escalas dos deslocamentos e forças do *riser* podem ser alteradas, mas as dimensões relativas entre o *riser* e a embarcação, por exemplo, não refletem a da realidade. Embora isso ocorra para facilitar a visualização, oferecer opções para controlar também as escalas destes e outros elementos da visualização aparenta ser um recursos desejável.

Por fim, é possível refinar a avaliação de usabilidade, incluindo novos voluntários no teste ou mesmo incluindo novas perguntas que por ventura sejam consideradas pertinentes, como

perguntas que avaliem a imersão e realismo do ambiente.

Bibliografia

Agência Nacional do Petróleo, G. N. e. B. (2012). Anuário estatístico brasileiro do petróleo, gás natural e biocombustíveis - 2012.

Al-Jamal, H. & Dalton, C. (2004). Vortex induced vibrations using large eddy simulation at a moderate reynolds number, *Journal of Fluids and Structures* **79**: 73–92.

Alexander, C. R. (2003). *Development of a Composite Repair System for Reinforcing Offshore Risers*, PhD thesis, Texas A&M University.

Bathe, K. (1996). *Finite Element Procedures*, Prentice-Hall Inc.

Bernardes, J. L. (2004). *Desenvolvimento de um ambiente para visualização tridimensional da dinâmica de risers*, Master's thesis, Escola Politécnica da Universidade de São Paulo.

Chai, Y. & Varyani, K. (2002). Three-dimensional lump-mass formulation of a catenary riser with bending, torsion and irregular seabed interaction effect, *Ocean Engineering* **29**: 1503–1525.

Chakrabarti, S. K. (1987). *Hydrodynamic of Offshore Structures, Computational Mechanics Publications*, Springer Verlag.

Chaplin, J., Bearman, P., Cheng, Y., Fontaine, E., Graham, J., Herfjord, K., Huarte, F. H., Isherwood, M., Lambrakos, K., Larsen, C., Meneghini, J., Moe, G., Pattenden, R., Triantafyllou, M. & Willden, R. (2005). Blind predictions of laboratory measurements of vortex-induced vibrations of a tension riser, *Journal of Fluids and Structures* **21**: 25–40.

Chen, K. H. C. & Chen, C. R. (2010). Vertical riser viv simulation in uniform current, *Journal of Offshore Mechanics and Arctic Engineering* **132**: 337–341.

DutoRisk Manual de Usuário (Interface e Solver) (2009). Manual Impresso, Laboratórito de Sistemas Marítimos de Produção e Risers.

Engel, K., Hadwiger, M., Kniss, J. M., Rezk-Salama, C. & Weiskopf, D. (2006). *Real-Time Volume Graphics*, A K Peters/CRC Press.

Ferrari, J. A. & Bearman, P. (1999a). Hydrodynamic loading and response of offshore risers, *Proceedings of the 18th International Conference on Offhosre Mechanics and Arctic Engineering*.

Flatschart, R. B., Meneghini, J. R. & Jr., J. A. F. (2004). Parallel simulation of a marine riser using mpi, *Proceedings of OMAE04*.

Holmes, S., Jr., O. H. O. & Constantinides, Y. (2006). Simulation of riser viv using fully three dimensional cfd simulations, *Proceedings of OMAE2006*.

Kubota, H. Y. (2003). *Comportamento dinâmico de um riser rígido de produção*, Master's thesis, Unicamp. Biblioteca Digital da Unicamp.

Morooka, C. K., Brandt, D. M., Matt, C. G. C. & Franciss, R. (2008). Features of a time domain simulation tool for rigid riser design, *Anais da Rio Oil & Gas Expo and Conference 2008*.

Morooka, C. K., Coelho, F. M., Kubota, H. Y., Ferrari, J. A. & Ribeiro, E. J. B. (2004). Investigations on the behavior of vertical production risers, *Proceedings of OMAE2004*.

Morooka, C. K., Coelho, F. M., Ribeiro, E. J. B., Ferrari, J. A. & Franciss, R. (2005). Dynamic behavior of a vertical riser and service life reductu, *Proceedings of OMAE2005*.

Morooka, C. K. & Tsukada, R. I. (2011). Dynamic behavior of pipelines and risers due to vortex-induced vibration in time domain, *Marine Systems & Ocean Technology* **6**: 17–26.

Newmark, N. M. (1959). A method of computation for structural dynamics, *Journal of Engineering Mechanics ASCE 85, EM3* pp. 67–94.

Orcina (2013a). Orcaflex manual, `http://www.orcina.com/SoftwareProducts/OrcaFlex/Documentation/OrcaFlex.pdf`. Acessado em Junho de 2013.

Orcina (2013b). Orcaflex website, `http://www.orcina.com/SoftwareProducts/OrcaFlex/`. Acessado em Junho de 2013.

Petrobras, T.-R. (2005). Anflex multilinhas - manual do usuário, versão 6.4.1 r5.5.

Santos, I. H. F., Soares, L. P., Carvalho, F. & Raposo, A. (2011). A collaborative vr visualization environment for offshore engineering projects, *Proceedings of the 10th International Conference on Virtual Reality Continuum and Its Applications in Industry*.

Schroeder, W., Martin, K. & Lorensen, B. (2006). *The Visualization Toolkit*, Kitware.

Simantiras, P. & Willis, N. (2001). Steel catenary risers - allegheny offshore viv monitoring campagin and large scale simulation of seabed interaction, *Deep Offshore Technology International*.

Telea, A. C. (2008). *Data Visualization*, A K Peters, Ltd.

Zhu, H. J., Ou, Z. P., Lin, Y. H. & Hu, F. F. (2011). Large eddy simulations of unsteady wakes behind riser in offshore deep water, *Advanced Materials Research* **268 - 270**: 787–792.

Zhu, H., Lin, Y., Jia, Q. & Yang, X. (2010). Simulations of suppressive effect of viv on marine riser with splitter plates, *ICCSIT*, pp. 337–341.

Zienkiewicz, O. C. (1971). *The Finite Element Method in Engineering Science*, McGraw-Hill Publishing Company Limited.

Apêndice A

Questionário para Avaliação de Usabilidade

Segue abaixo o questionário elaborado para a avaliação de usabilidade do sistema desenvolvido neste trabalho, conforme exposto na Seção 3.2.6 do Capítulo 3. Os resultados do teste são apresentados e discutidos na Seção 4.4 do Capítulo 4.

Questionário para Avaliação do Ambiente de Visualização de Simulação de *Risers*

1 - Introdução

O presente documento é um questionário para avaliação do ambiente de simulação e visualização de *riser* desenvolvido como parte do projeto de mestrado de Rafael Guimarães Ramos e orientado pelo Professor José Mario De Martino, da Faculdade de Engenharia Elétrica e de Computação da Unicamp.

O ambiente desenvolvido tem a finalidade de auxiliar o engenheiro de petróleo no estudo de *riser*, bem como no projeto e no diagnóstico de problemas deste equipamento. Através do programa, o usuário pode realizar uma simulação do comportamento do *riser* e visualizar os dados de simulação num ambiente virtual em gráficos 3D. Os parâmetros de simulação podem ser ajustados durante a visualização e o novo comportamento do *riser* é reproduzido imediatamente.

2 - Instruções de Uso do Ambiente

2.1 Iniciando a simulação

Para iniciar uma simulação, após iniciar o programa, o usuário deve primeiro criar um modelo do *riser*, clicando em *New -> Model ->* e escolhendo o modelo (Figura 1). O modelo determina se o *riser* é preso no topo (*Collector*) , no fundo (*Tower*) ou em ambos (*TTR*). Em seguida, é necessário escolher o número de elementos do modelo (Figura 2). Quanto maior o número de elementos maior a precisão nos resultados, porém a execução pode ficar lenta dependendo do hardware. Nas máquinas testadas, o ambiente manteve bom desempenho até 180 elementos.

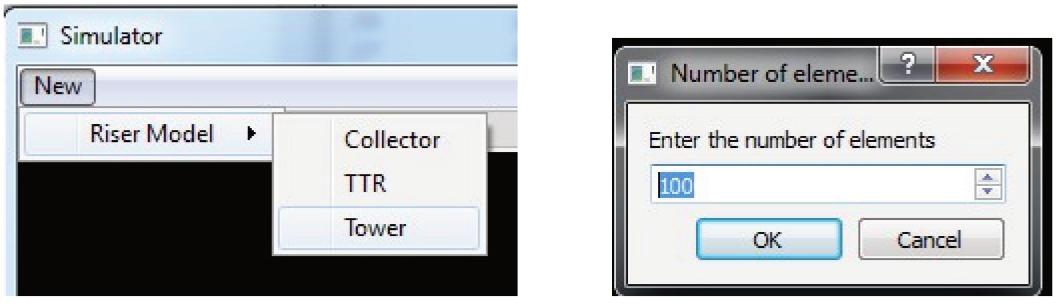

Figura 1, criando novo modelo (esquerda), e Figura 2(direita), escolhendo número de elementos

Criado um modelo, a janela obtida deve ser semelhante a da Figura 3. Um conjunto padrão de parâmetros é utilizado para a simulação, que pode ser iniciada clicando no botão *Play* no painel e pausada pelo mesmo botão. Por fim, para melhor visualizar, ajuste as opções de visualização na aba *Display*. Como os deslocamentos do *riser* são muito pequenos em comparação seu comprimento, é necessário ajustar a escala destas grandezas. Clicando-se no botão *Auto Scale* dentro da aba *Display*, a escala é ajustada automaticamente para melhor visualizar o deslocamento do *riser*. Alternativamente, estas escalas podem ser ajustadas manualmente nesta aba. Ainda na aba *Display*, é possível escolher qual dentre certos atributos da simulação é visualizado por cores.

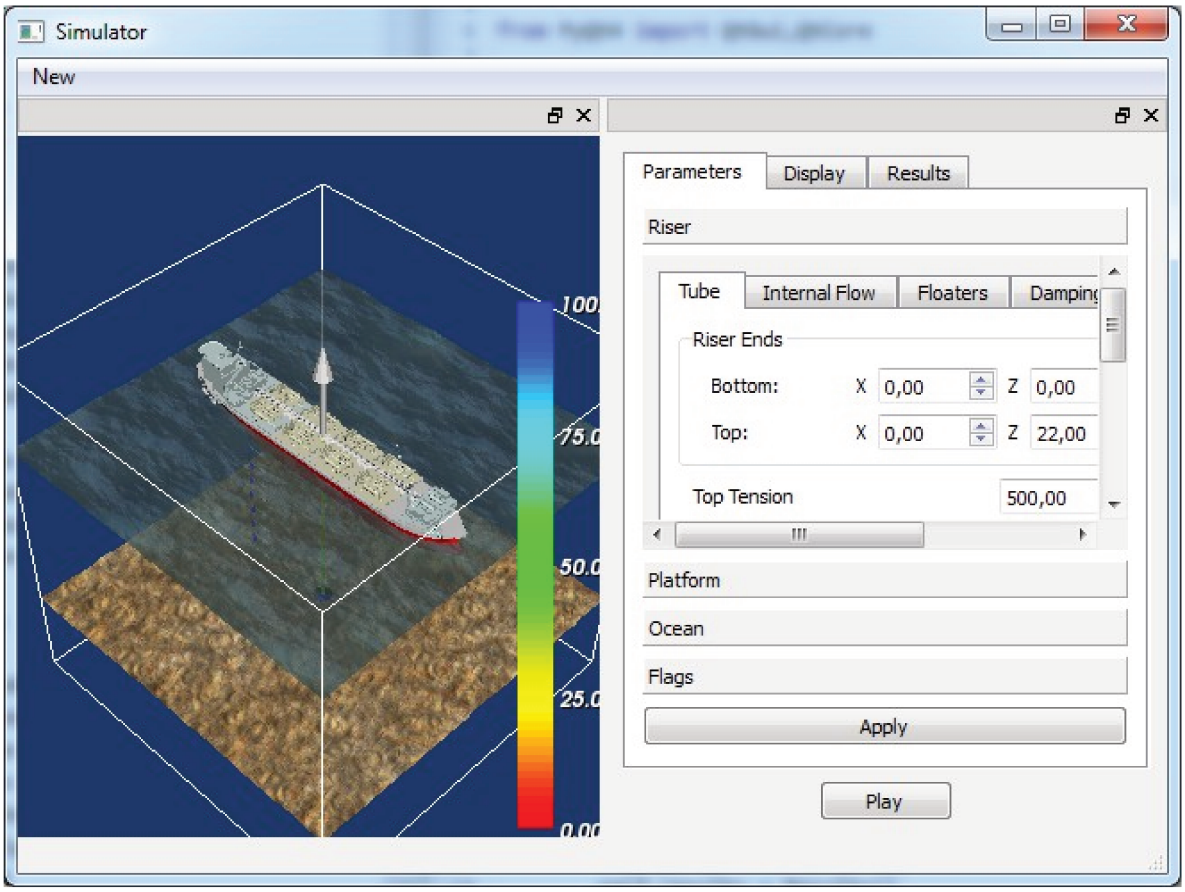

Figura 3 - janela do ambiente, com ambiente de visualização à esquerda e o painel à direita. Os botões Apply e Play podem ser visto na parte inferior do painel.

A qualquer momento, com a simulação rodando ou pausada, é possível alterar os parâmetros da simulação. Isto é feito na aba *Parameters*. Os diferentes parâmetros encontram-se agrupados em grupos e subgrupos dentro desta aba. Mais de um parâmetro pode ser alterada de uma única vez, mas é necessário clicar no botão *Apply* para submeter os novos valores.

Por fim, ao lado do botão *Play*, há o botão *Reset*, que apenas recoloca o *riser* em sua posição inicial, reiniciando a simulação, mas sem alterar os parâmetros.

2.2 Visualização da Simulação

A visualização da simulação contém diversos elementos representando os diversos aspectos da simulação. Primeiramente, temos o *riser*, representado como um tubo no centro da cena, a embarcação, o fundo e a superfície do mar e os flutuadores sobre o *riser*. Há também diversos elementos visuais que representam grandezas físicas da simulação. Além do tubo (que representa o *riser*) se movimentar, sobre ele são colocadas setas que representam a força de excitação e cores para representar diferentes grandezas (conforme escolhido na aba *Display*). Uma unica seta apontada para cima é utilizada para exibir a tensão de topo e curvas na cor verde são traçadas para indicar o máximo deslocamento do *riser*. Por fim, setas dispostas um pouco afastadas do *riser* são utilizadas para representar a correnteza. A Figura 4 ilustra estes elementos.

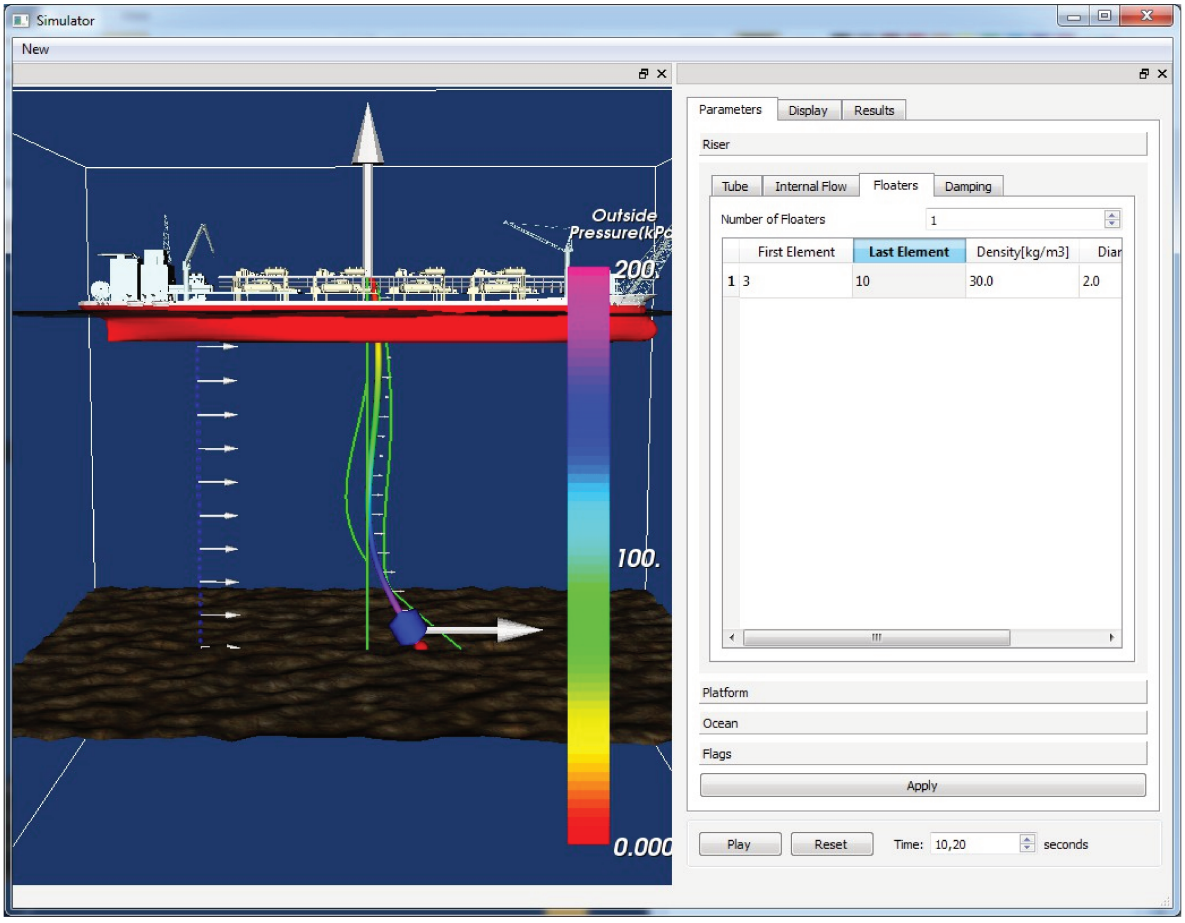

Figura 4 - Visualização da simulação

Sendo uma visualização 3D, é possível interagir com a cena clicando e arrastando o *mouse*. Clicando e arrastando com o botão esquerdo, é feita a rotação da cena; com o do meio, a translação, e com o da direita, escala (a escala também pode ser feita rolando o botão do meio).

Os valores numéricos das forças, deslocamentos e velocidades de cada elemento são exibidos na aba *Results*. Ao clicar sobre algum destes valores na tabela, uma esfera vermelha é colocada sobre o *riser* na posição do elemento correspondente. Inversamente, ao colocar o mouse sobre uma parte do *riser* e pressionar o botão 'p' do teclado, o elemento na posição indicada é selecionado e realçado na tabela.

3 - Instruções para o Questionário

Tendo lido as instruções, você terá até 10 minutos para explorar livremente o programa e se familiarizar com seu uso. Crie uma simulação, inicie a animação, altere parâmetros, interaja com a visualização, conforme a instruções da seção 2; repita quantas vezes quiser. Em caso de dúvida, pergunte ao supervisor presente.

Após a familiarização, avise o supervisor e então responda o questionário presente na seção 4. No questionário são feitas perguntas quanto a utilidade do programa como ferramenta para auxiliar o engenheiro de petróleo tanto no estudo do comportamento de um *riser*, quanto para o projeto de *risers* ou diagnóstico de problemas. São feitas também perguntas quanto a qualidade da visualização e da interatividade do programa. Por fim, há perguntas comparando o ambiente aqui apresentado com o *VRP*, um outro ambiente para visualização de *risers* desenvolvido na Unicamp. Caso nunca tenha utilizado o programa ou não lembre do seu funcionamento, não é necessário responder esta parte do questionário.

As questões são todas objetivas, e você deverá marcar a opção que mais se adequar a sua percepção. No quadro Justificativa, pede-se que, se possível, escreva sucintamente uma justificativa quanto à opção marcada. Durante o questionário, você pode utilizar o ambiente sempre que achar necessário, para confirmar suas impressões antes de marcar a resposta.

4 - Questionário. Responda dando notas de 1 a 5 e explique sucintamente o porquê no quadro Justificativa

Categoria Geral

1) Para a finalidade de projeto de *risers*, o ambiente testado é :

(1) nada útil
(2) pouco útil
(3) útil
(4) muito útil
(5) extremamente útil

Justificativa:

2) Para a finalidade de análise do comportamento de *risers*, o ambiente testado é :

(1) nada útil
(2) pouco útil
(3) útil
(4) muito útil
(5) extremamente útil

Justificativa:

3) Para a finalidade de diagnóstico de problemas em *risers*, o ambiente testado é :

(1) nada útil
(2) pouco útil
(3) útil
(4) muito útil
(5) extremamente útil

Justificativa:

4) Para a finalidade de visualizar o comportamento global do *riser*, o ambiente testado é:

(1) nada adequado
(2) pouco adequado
(3) adequado
(4) muito adequado
(5) extremamente adequado

Justificativa:

5) Para a finalidade de visualizar individualmente cada resultado da simulação, o ambiente testado é:

(1) nada adequado
(2) pouco adequado
(3) adequado
(4) muito adequado
(5) extremamente adequado

Justificativa:

6) Quanto à facilidade de uso, o ambiente testado é:

(1) muito difícil
(2) difícil
(3) médio
(4) fácil
(5) muito fácil

Justificativa:

7) Quanto à aparência do ambiente testado, ela é:

(1) muito ruim
(2) ruim
(3) média
(4) boa
(5) muito boa

Justificativa:

Categoria Visualização

8) Quanto a representação da simulação na forma de um ambiente de realidade virtual, ela é:

(1) nada intuitiva
(2) pouco intuitiva
(3) intuitiva
(4) muito intuitiva
(5) extremamente intuitiva

Justificativa:

9) Quanto à qualidade da animação (fluidez), ela é:

(1) muito ruim
(2) ruim
(3) média
(4) boa
(5) muito boa

Justificativa:

Categoria Interatividade

10) Quanto à interatividade, ela é:

(1) muito limitada
(2) limitada
(3) média
(4) boa
(5) muito boa

Justificativa:

11) Quanto ao tempo de resposta ao mudar parâmetros, ele é:

(1) muito inadequado
(2) inadequado
(3) médio
(4) adequado
(5) muito adequado

Justificativa:

12) Quanto à organização do painel de controle, ela é:

(1) muito confusa
(2) confusa
(3) média
(4) intuitiva
(5) muito intuitiva

Justificativa:

Categoria Comparação com o VRP (não responder caso nunca tenha utilizado o VRP)

13) Para a finalidade de projeto de *risers*, o ambiente testado é :

(1) muito pior que o VRP
(2) pior que o VRP
(3) tão bom quanto o VRP
(4) melhor que o VRP
(5) muito melhor que o VRP

Justificativa:

14) Para a finalidade de análise do comportamento de *risers*, o ambiente testado é :

(1) muito pior que o VRP
(2) pior que o VRP
(3) tão bom quanto o VRP
(4) melhor que o VRP
(5) muito melhor que o VRP

Justificativa:

15) Para a finalidade de diagnóstico de problemas em *risers*, o ambiente testado é:

(1) muito pior que o VRP
(2) pior que o VRP
(3) tão bom quanto o VRP
(4) melhor que o VRP
(5) muito melhor que o VRP

Justificativa:

16) Para a finalidade de visualizar o comportamento global do *riser*, o ambiente testado é:

(1) muito pior que o VRP
(2) pior que o VRP
(3) tão bom quanto o VRP
(4) melhor que o VRP
(5) muito melhor que o VRP

Justificativa:

17) Para a finalidade de visualizar individualmente cada resultado da simulação, o ambiente testado é:

(1) muito pior que o VRP
(2) pior que o VRP
(3) tão bom quanto o VRP
(4) melhor que o VRP
(5) muito melhor que o VRP

Justificativa:

18) Quanto à facilidade de uso, o ambiente testado é:

(1) muito pior que o VRP
(2) pior que o VRP
(3) tão bom quanto o VRP
(4) melhor que o VRP
(5) muito melhor que o VRP

Justificativa:

19) Quanto à aparência do ambiente testado, ela é:

(1) muito pior que o VRP
(2) pior que o VRP
(3) tão bom quanto o VRP
(4) melhor que o VRP
(5) muito melhor que o VRP

Justificativa:

Apêndice B

Documentação do Código do Sistema Desenvolvido

Segue abaixo a documentação do código do sistema desenvolvido neste trabalho. O código foi desenvolvido de modo a atender a metodologia apresentada no Capítulo 3 e uma visão geral do código é apresentada na Seção 4.2 do Capitulo 4. Na documentação a seguir, maiores detalhes são fornecidos.

Documentação do Código do Ambiente de Visualização e Simulação de *Risers*

Esta é a documentação do código do Ambiente de Visualização e Simulação de *Risers* desenvolvido como parte do mestrado de Rafael Guimarães Ramos e orientado pelo professor José Mario De Martino, da Faculdade de Engenharia Elétrica e de Computação (FEEC) da Unicamp. A documentação descreve as classes do código e seus métodos, explicando o propósito de cada uma e como se relacionam entre si. Primeiramente, apresentamos as classes do *Solver* e, em segundo lugar, as classes do Módulo de Visualização.

Documentação do *Solver*

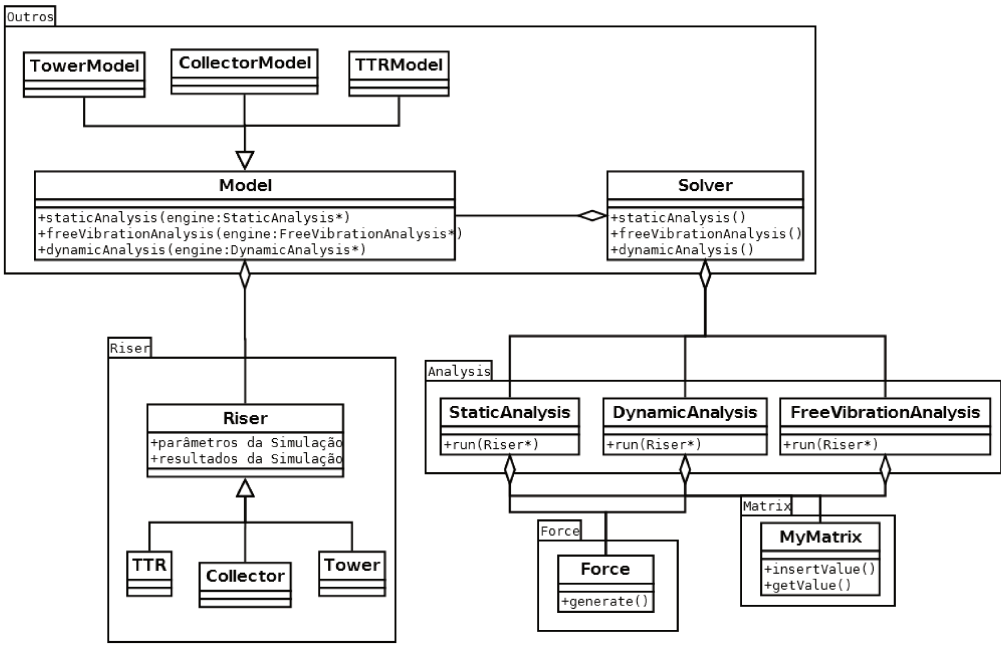

Figura 1 – Diagrama de classe do *Solver*.

A descrição das classes do *Solver* está dividida em grupos de acordo com o papel que cada uma cumpre na simulação. Na Figura 1 são mostradas as principais classes do *Solver*, como se relacionam e os grupos a que pertencem. As classes do grupo *Analysis* são aquelas responsáveis por executar a simulação propriamente dita, recebendo um objeto do tipo *Riser* como parâmetro, realizando os cálculos de acordo com estes valores e armazenando no mesmo objeto os resultados obtidos. As classes do grupo *Riser*, que inclui a classe *Riser* mencionada acima, são as classes que funcionam como contêiner de dados. O grupo *Matrix* consiste das classes que implementam as diversas matrizes utilizadas nos cálculos da simulação (matriz de rigidez, massa, etc) e o grupo *Force*, similarmente, os vetores de força aplicados sobre o *riser*. Por fim, o grupo *Outros* contém as demais classes, como a classe *Solver*, que é a classe principal deste módulo e encapsula as demais.

Marcações: Algumas classes e métodos encontram-se marcados na documentação com um dos seguintes sinais.

(P) O método/classe marcado implementa simulação da configuração Pipeline. Esta configuração, que é a de um riser rígido *não-vertical*, não é tratada neste trabalho (já que a visualização deste tipo de configuração diferiria razoavelmente das outras, *verticais*). Como o RiserProd original realiza a simulação desta configuração, a implementação chegou a ser iniciada, mas não foi completada por estar fora do escopo do trabalho.

(R) Embora corretamente implementado, é possível que exista uma forma melhor de organizar a implementação do método/classe, entretanto isto não foi explorado.

Grupo Analysis

As classes do grupo *Analysis* implementam as três análises que constituem a simulação: análise estática (classe *StaticAnalysis*), de vibrações livres (*FreeVibrationAnalysis*) e dinâmica (*DynamicAnalysis*). A forma de uso delas é semelhante. Basta instanciar um objeto da classe e invocar o método *run* dela, passando o ponteiro para um objeto de uma subclasse de *Riser*. A análise é feita de acordo com os parâmetros armazenados neste objeto e os resultados são nele escritos.

==StaticAnalysis==

Classe que implementa a análise estática. A forma de uso consiste em instanciar um objeto da classe e invocar o método *run*, passando o ponteiro de um objeto de alguma subclasse de *Riser* (*TTR*, *Collector* ou *Tower*). Os parâmetros da análise são tomados a partir deste argumento, e nele são escritos os resultados da análise.

A análise estática considera o *riser* como uma estrutura em equilíbrio estático representável na forma K*X = F e visa calcular a posição dos elementos do *riser* (vetor X) para o conjunto de cargas imposta sobre a estrutura (vetor F). O método utilizado é o *método de Galerkin*, que iterativamente estima a matriz de rigidez (o K da equação), que em parte depende da posição dos elementos, para estimar um vetor X, o que leva a uma nova estima de K, e assim em diante.

StaticAnalysis();

Construtor da classe. Não faz nada de adicional.

void run(TTR*);

Método que executa a análise estática completa, *inline* e transversal, para *TTR*. Chama o *template* do método *inlineAnalysis* e *transAnalysis* para o tipo *TTR**.

void run(Tower*);

Método que executa a análise estática completa, *inline* e transversal, para *Tower*. Chama o *template* do método *inlineAnalysis* e *transAnalysis* para o tipo *Tower**.

void run(Collector*);

Método que executa a análise estática completa, *inline* e transversal, para *Collector*. Chama o *template* do método *inlineAnalysis* e *transAnalysis* para o tipo *Collector**.

void run(Pipeline*);

(P) Método que executa a análise estática completa, *inline* e transversal, para *Pipeline*. Chama o *template* do método *inlineAnalysis* e *transAnalysis* para o *tipo Pipeline**.

```
template <class SpecificRiserType> void inlineAnalysis(SpecificRiserType*);
```

Template que executa a análise estática no plano *inline*. Chama *initInline* para inicializar a análise, *iterativeInlineAlgol* para o algoritmo principal e *finalInline* para finalizar. É utilizado um *template* pois, embora a análise seja semelhante para todas as configurações, a matriz de rigidez criada varia conforme a configuração de *riser*.

```
template <class SpecificRiserType> void transAnalysis(SpecificRiserType*);
```

Template que executa a análise estática no plano transversal. Chama *initTrans* para inicializar a análise, *iterativeTransAlgol* para o algoritmo principal e *finalTrans* para finalizar. É utilizado um *template* pois, embora a análise seja semelhante para todas as configurações, a matriz de rigidez criada varia conforme a configuração de *riser*.

```
template <class SpecificRiserType> void initInline(SpecificRiserType*);
```

Template que aloca as estruturas usadas na análise estática *inline*. É utilizado um *template* pois, embora a inicialização seja semelhante para todas as configurações, a matriz de rigidez criada varia conforme a configuração de *riser*.

```
void finalInline();
```

Desaloca as estruturas criadas na análise estática *inline*.

```
template <class SpecificRiserType> void iterativeInlineAlgol(SpecificRiserType*);
```

Realiza o algoritmo do *método de Galerkin* para o plano *inline*. É utilizado um *template* pois, embora o método seja semelhante para todas as configurações, a matriz de rigidez criada varia conforme a configuração de *riser*.

```
template <class SpecificRiserType> void initTrans(SpecificRiserType*);
```

Template que aloca as estruturas usadas na análise estática transversal. É utilizado um *template* pois, embora a inicialização seja semelhante para todas as configurações, a matriz de rigidez criada varia conforme a configuração de *riser*.

```
void finalTrans();
```

Desaloca as estruturas criadas na análise estática transversal.

```
template <class SpecificRiserType> void iterativeTransAlgol(SpecificRiserType*);
```

Realiza o algoritmo do *método de Galerkin* para o plano transversal. É utilizado um *template* pois, embora o método seja semelhante para todas as configurações, a matriz de rigidez criada varia conforme a configuração de *riser*.

```
void tension_sr(Riser*);
```

Calcula a tensão ao longo do *riser*. Aceita a ponteiro para classe *Riser*, mas é efetivamente utilizado apenas pelas subclasses *Tower* e *TTR*.

void tension_sr(Collector*);

Calcula a tensão ao longo do *riser*, específico para configuração *Collector*.

void tension_sr(Pipeline*);

(P) Calcula a tensão ao longo do *riser*, específico para configuração Pipeline.

void desbf_stat();

Função auxiliar chamada pelos métodos *iterativeInlineAlgol* e *iterativeTransAlgol*,

==FreeVibrationAnalysis==

Classe que implementa a análise de vibrações livres. A forma de uso consiste em instanciar um objeto da classe e invocar o método *run*, passando o ponteiro de um objeto de alguma subclasse de *Riser* (*TTR*, *Collector* ou *Tower*). Os parâmetros da análise são tomados a partir deste argumento, e nele são escritos os resultados da análise.

A análise de vibrações livres considera o *riser* como uma estrutura em movimento oscilatório não amortecido sem cargas aplicadas sobre a estrutura, representável na forma K*X + M*X" = 0. Tal equação leva a outra, (M^-1*K-w^2*I)*Y = 0, onde X = Y*e^iwt e w = (w1,w2,...,wn), as frequências naturais do sistema. A equação é solucionada encontrando os autovalores do sistema, w^2.

FreeVibrationAnalysis();

Construtor da classe. Não faz nada de adicional.

void run(TTR*);

Método que executa a análise de vibrações livres completa, *inline* e transversal, para *TTR*. Chama o *template* do método *inlineAnalysis* e *transAnalysis* para o tipo *TTR**.

void run(Tower*);

Método que executa a análise de vibrações livres completa, *inline* e transversal, para *Tower*. Chama o *template* do método *inlineAnalysis* e *transAnalysis* para o tipo *Tower**.

void run(Collector*);

Método que executa a análise de vibrações livres completa, inline e transversal, para Collector. Chama o template do método inlineAnalysis e *transAnalysis* para o tipo *Collector**.

void run(Pipeline*);

(P) Método que executa a análise de vibrações livres completa, *inline* e transversal, para Pipeline. Chama o *template* do método *inlineAnalysis* e *transAnalysis* para o tipo *Pipeline**.

`template <class SpecificRiserType> void inlineAnalysis(SpecificRiserType*);`

Realiza a análise de vibrações livres no plano *inline*. Após alocar as matrizes de massa e rigidez, elimina um grau de liberdade das matrizes originais(*reduce*), completa com os elementos simétricos (*simetry*), realiza a *decomposição de cholesky* (*choleskyInline*) gerando a matriz para a qual são calculados os autovalores (*eigen*). Por fim, desaloca as estruturas.

`template <class SpecificRiserType> void transAnalysis(SpecificRiserType*);`

Realiza a análise de vibrações livres no plano transversal. Após alocar as matrizes de massa e rigidez, elimina um grau de liberdade das matrizes originais(*reduce*), completa com os elementos simétricos (*simetry*), realiza a *decomposição de cholesky* (*choleskyInline*) gerando a matriz para a qual são calculados os autovalores (*eigen*). Por fim, desaloca as estruturas.

`void elem(TTR*,int*);`

Função que auxilia na redução dos graus de liberdade.

`void elem(Tower*,int*);`

Função que auxilia na redução dos graus de liberdade.

`void elem(Collector*,int*);`

Função que auxilia na redução dos graus de liberdade.

`void elem(Pipeline*,int*);`

(P) Função que auxilia na redução dos graus de liberdade.

`template <class SpecificRiserType> void reduce(SpecificRiserType*);`

Reduz os graus de liberdade das matrizes de massa e rigidez, eliminando rotação e mantendo apenas translação *inline* e transversal.

`void simetry(Riser*);`

Gera a versão simétrica das matrizes reduzidas.

`void choleskyInline(Riser*);`

Usando a *decomposição de Cholesky* para inversão de matrizes, o método calcula a matriz A = M^-1*K (M e K com dimensões reduzidas e M simétrico) da qual serão calculados os autovalores. O *método de Cholesky* é usado pois gera uma matriz L tal que M = L*Lc (Lc sendo a transposta conjugada de L) que, por ser triangular, é mais fácil de ser invertida.

`void eigen(Riser*);`

Calcula os autovalores da matriz A = (M^-1)*K. Usa funções da biblioteca CLAPACK para isso.

==DynamicAnalysis==

Classe que implementa a análise dinâmica. A forma de uso consiste em instanciar um objeto da classe e invocar o método *run*, passando o ponteiro de um objeto de alguma subclasse de *Riser* (*TTR*, *Collector* ou *Tower*). Os parâmetros da análise são tomados a partir deste argumento, e nele são escritos os resultados da análise.

A análise dinâmica considera o *riser* como uma estrutura em movimento oscilatório amortecido com cargas aplicadas sobre a estrutura, representável na forma K*X + D*X' + M*X" = F. A matriz de amortecimento D é calculada a partir das matrizes M e K e das frequências naturais, enquanto o F é em parte constante e em parte variável no tempo. A posição X do *riser* é calculada nesta análise para cada instante de tempo, através do método de *Newmark-Beta*.

DynamicAnalysis();

Construtor da classe. Inicializa alguns atributos do objeto.

void run(TTR*);

Método que executa a análise dinâmica completa, *inline* e transversal, para *TTR*. Chama o *template* do método *inlineAnalysis* e *transAnalysis* para o tipo *TTR**.

void run(Tower*);

Método que executa a análise dinâmica completa, *inline* e transversal, para *Tower*. Chama o *template* do método *inlineAnalysis* e *transAnalysis* para o tipo *Tower**.

void run(Collector*);

Método que executa a análise dinâmica completa, *inline* e transversal, para *Collector*. Chama o *template* do método *inlineAnalysis* e *transAnalysis* para o tipo *Collector**.

void run(Pipeline*);

(P) Método que executa a análise dinâmica completa, *inline* e transversal, para *Pipeline*. Chama o *template* do método *inlineAnalysis* e *transAnalysis* para o tipo *Pipeline**.

void reset(Riser*);

Reinicia a análise dinâmica, colocando o *riser* na sua posição inicial.

void deleteMatrixes();

Desaloca matrizes utilizadas na análise dinâmica.

Real time();

Retorna o instante da simulação (não o tempo real que a simulação durou até o momento, mas o tempo no modelo de simulação).

```
template <class SpecificRiserType> void createMatrixes(SpecificRiserType*);
```

Cria as matrizes utilizadas na análise dinâmica. É utilizado um *template* pois embora este método em si não varie conforme a configuração de *riser*, as matrizes criadas pedem um tipo específico de *riser*.

```
template <class SpecificRiserType> void iterativeAlgol(SpecificRiserType*);
```

Executa o algoritmo principal da análise dinâmica. É utilizado um *template* pois embora este método em si não varie conforme a configuração de *riser*, as matrizes utilizadas pedem um tipo específico de *riser*. Chama *inlineTimeAnalysis* e *transTimeAnalysis*, que executam a análise para cada plano, *inline* e transversal.

```
void initInlineTimeAnalysis(Riser*);
```

Inicializa a análise dinâmica no plano *inline*, gerando as forças nesta direção.

```
void initTransTimeAnalysis(Riser*);
```

Inicializa a análise dinâmica no plano transversal, gerando as forças nesta direção.

```
template <class SpecificRiserType> void inlineTimeAnalysis(SpecificRiserType*);
```

Realiza a análise dinâmica no plano *inline*.

```
template <class SpecificRiserType> void transTimeAnalysis(SpecificRiserType*);
```

Realiza a análise dinâmica no plano transversal.

```
void storePreviousValues(Riser*);
```

Armazena os valores de deslocamento, velocidade e força atuais para poderem ser utilizados no próximo time step da análise dinâmica como "valores anteriores".

```
void inLoad(Riser*,Real*,bool);
```

Calcula as cargas sobre o *riser* no plano *inline*. Utiliza o *generate* de *InlineForce*.

```
void trLoad(Riser*,Real*,bool);
```

Calcula as cargas sobre o *riser* no plano transversal. Utiliza o *generate* de *TransForce*.

Grupo Riser

As classes do grupo *Riser* implementam as classes que armazenam os parâmetros e resultados da simulação. A classe *Riser* é a classe principal deste grupo, contendo em si os diversos dados do *riser* como diâmetro interno e externo, resistência do material (estes, parâmetros da simulação), bem como deslocamento, velocidade e força sobre os elementos do *riser* (resultados da simulação). Certos atributos de *Riser* são representados pelas outras classes do grupo, como *Environment* (para os dados correspondentes ao ambiente) e Platform (para dados da plataforma). A classe *Riser* possui quatro subclasses, *TTR*, *Tower*, *Collector* e *Pipeline* (cuja implementação foi descontinuada por estar fora do escopo do trabalho). Embora

as classes não contenham atributos novos ou implementem novos métodos, são utilizadas para diferenciar diversos métodos de outras classes (como os métodos *assemble* de algumas matrizes do grupo *Matrix*, mais adiante), que variam conforme a configuração de *riser* utilizada.

==Riser==

A classe *Riser* é o principal contêiner de dados do Solver, contendo os parâmetros e resultados da simulação. Suas subclasses *TTR*, *Collector* e *Tower* (e *Pipeline*, cuja implementação foi descontinuada por estar fora do escopo do trabalho) definem os tipos específicos de *riser* tratados na simulação. Objetos dessas classes são passados (por ponteiro) como argumento dos métodos *run* das classes *StaticAnalysis*, *FreeVibrationAnalysis* e *DynamicAnalysis*.

Dentre os dados contidos na classe temos informações do *riser*, como os diâmetros interno e externo (*inDiam* e *exDiam*), a densidade do material (*matDensity*) e diversos outros. Contém também objetos que representam dados do ambiente (*Environment*), da plataforma (*Platform*), do fluído interno (*InternalFlow*) e dos flutuadores ao longo do *riser* (*Floaters*).

Muitos dos dados, em geral os vetores com um item para cada elemento do *riser*, tem seus valores gerados a partir de outros dados contidos em *Riser*. É o caso, por exemplo, dos coeficiente de arrasto (CD) e massa adicional (CA), que são dados inicialmente para alguns pontos e interpolados para cada elemento. O método *enrichParameters* é responsável por esta etapa, de inicializar atributos de *Riser* a partir do valor de outros atributos do mesmo objeto (processo referido como “enriquecimento” dos dados).

```
Riser(int,int);
```

Construtor da classe. Aloca os vetores utilizados.

```
~Riser();
```

Destrutor da classe. Desaloca os vetores criados antes.

```
void defaultParameters();
```

(R) Atribui um conjunto de parâmetros padrão. Será removida.

```
void enrichParameters();
```

Inicializa dados de *Riser* a partir de outros atributos do mesmo objeto. Na realidade, apenas chama outras funções mais específicas que fazem isto.

```
void _riser();
```

"Enriquece" dados específicos do *riser*, como dimensões dele (*_dimensions*), flutuadores (*_floaters*) e fluxo interno (*_flow*).

```
void _platform();
```

"Enriquece" dados ligados a plataforma.

void _environment();

"Enriquece" dados ligados ao ambiente, calculando a pressão hidrostática externa e chamando *_current* para correnteza, *_waves* para onda e *_drag* para as constantes hidrodinâmicas.

void _weights();

"Enriquece" dados ligados ao peso do *riser*. Diversos pesos médios por elemento são calculados, como peso médio acima da água, peso abaixo, peso apenas da parede do *riser* e peso com flutuador.

void _dimensions();

"Enriquece" dados ligados as dimensões, calculando o comprimento total acima e abaixo do nível do mar e a inclinação do *riser*.

void _floaters();

Interpola para cada elemento do *riser* os dados dos flutuadores

void _flow(Real*);

"Enriquece" dados ligados ao fluxo interno, chamando a função apropriada conforme o tipo de fluido interno.

void flowoil();

Chamado por *_flow*, no caso de fluxo para apenas óleo.

void flowog();

Chamado por *_flow*, no caso de fluxo para óleo e gás.

void flownone();

Chamado por *_flow*, no caso sem fluxo.

void _waves();

"Enriquece" dados referentes as ondas.

void _current();

"Enriquece" dados referentes a correnteza, interpolando para cada elemento o perfil de correnteza.

void _drag();

"Enriquece" dados referentes a constantes hidrodinâmica (CA e CD), interpolando para cada elemento o perfil de correnteza.

```
void interpolate(Profile*,Real*);
```

Utilizado para interpolar as constantes hidrodinâmicas (CA e CD).

```
Real elemLength(int);
```

Retorna o tamanho do elemento dado o índice dele.

```
Real wall(int);
```

Retorna o peso da parede do elemento dado o índice dele.

```
Real floaters_effect(int);
```

Retorna o peso do flutuador naquele elemento dado o índice dele.

```
Real buoyancy(int);
```

Retorna o efeito do empuxo no elemento dado o índice dele.

```
Real mass(int);
```

Retorna a massa do elemento dado o índice dele.

```
Real weight(int);
```

Retorna o peso do elemento, contando flutuador e efeito do empuxo, dado o índice do elemento.

```
Real addMassInline(int);
```

Retorna a massa adicional *inline* do elemento dado o índice dele.

```
Real addMassTrans(int);
```

Retorna a massa adicional transversal do elemento dado o índice dele.

```
void print(int);
```

Imprime dados do *riser*. Apenas para *debug*.

==TTR, Collector, Tower e Pipeline==

Apenas definem as classes, sem conter diferenças nos métodos.

==Floater==

A classe *Floater* representa os flutuadores presos ao *riser*. Não é necessário um objeto desta classe para cada flutuador, mas um único já é responsável por armazenar todos os dados dos flutuadores (*Riser* possui apenas uma instância desta classe). Os vetores atributos desta classe tem tamanho igual ao número de flutuadores, cada elemento do vetor correspondendo a um flutuador. Para o k-ésimo flutuador, a posição k do vetor *first* indica qual o primeiro elemento de *riser* que contém este flutuador, e *last* o último. O vetor *density* guarda a densidade do flutuador e *exDiam* o diâmetro externo. Estes vetores são alocados pelo método *init*, cujo argumento é o número de flutuadores, mas os vetores devem ser preenchidos diretamente.

`void init(int);`

Método utilizado para indicar o número de flutuadores, alocando memória para eles.

`Floater();`

Construtor da classe. Inicializa atributos.

==Platform=

Classe que contêm informações sobre a plataforma.

`void defvals();`

Inicializa um conjunto padrão de parâmetros.

==Wave==

Classe que contêm informações sobre as ondas.

`void defvals();`

Inicializa um conjunto padrão de parâmetros.

==Current==

Classe que contêm informações sobre o perfil de correnteza. O perfil é dado em pontos chave que depois são interpolados linearmente pela função *_current* de *Riser*. O vetor *depth* contém a profundidade de cada um destes pontos e *current* a correnteza. É necessário utilizar

initProfile para alocar estes vetores, mas eles devem ser preenchidos diretamente.

Current();

Construtor da classe. Inicializa algumas variáveis.

void defvals();

Inicializa um conjunto padrão de parâmetros.

void initProfile(int);

Aloca um perfil de correnteza com o número de pontos especificado no argumento.

==Profile==

Classe utilizada para guardar a informação do perfil de variação dos coeficientes de arrasto e de massa adicional (CA e CD, tanto *inline* quanto transversal). Basta instanciar a classe e utilizar *addPoint* para incluir um novo ponto, passando a posição e o valor do coeficiente. O método *_drag* de *Riser* posteriormente utiliza estes dados para interpolar linearmente os coeficientes para cada elemento do *riser*.

Profile();

Construtor da classe. Inicializa atributos.

void reset();

Remove todos os pontos do perfil.

void addPoint(Real c,Real v);

Adiciona um ponto, passando posição e valor do coeficiente.

==Environment==

Classe que representa os dados do ambiente. Contém outras classes mais específicas como *Current* e *Wave*.

void defvals();

Inicializa um conjunto padrão de parâmetros.

==InternalFlow==

Classe que representa o fluxo interno no *riser*, que pode ser de óleo, óleo e gás, ou nenhum. Os métodos da classe são utilizados pelos métodos *flowoil*, *flowog* e *flownone* da classe *Riser*.

Grupo Matrix

(R) As classes do grupo *Matrix* implementam as diversas matrizes utilizadas na análises da simulação. Como os elementos concentram-se em faixas bem definidas das matrizes convencionais, nestas classes são linearizados e armazenados em vetores unidimensionais para poupar memória. As classes *MyMatrix* e também as derivadas desta, *AsymmetricMatrix* e *SymmetricMatrix*, implementam as funcionalidades básicas deste tipo de matriz. As demais implementam matrizes mais especificas, como as de massa, rigidez e amortecimento, sendo que estas mesmas possuem múltiplas versões (*lumped* x consistente, *inline* x transversal).

O conjunto de classes definidas pode ser reorganizado, talvez definindo uma classe base para matrizes na forma *lumped* e na forma consistente e passando os métodos *decompGauss* e *decompGauss2* para elas.

==MyMatrix==

Classe base para as demais matrizes, implementando diversos métodos utilizados por elas. Ao invés de armazenar os elementos numa matriz comum (duas dimensões), os dados são linearizados e armazenados num vetor unidimensional, já que os valores diferentes de zero concentram-se apenas em uma faixa bem definida da matriz original. Assim, o acesso aos elementos da matriz é feito pelos métodos *insertValue*, *addValue* e *getValue*. O mapeamento para a forma linear é feito pelo método virtual *map*, que é nesta classe apenas definido, sendo implementado nas subclasse *AsymmetricMatrix* e *SymmetricMatrix*.

A classe dispõe de três construtores. Um aceita três parâmetros: tamanho total do vetor, dimensão original e tamanho da banda onde concentram-se os elementos. Outro aceita apenas dimensão e banda, sendo o tamanho calculado como o produto deles. Um terceiro construtor não pede argumentos mas também não inicializa nenhum destes atributos nem aloca o vetor.

Por fim, temos os métodos *decompGauss* e *decompGauss2*. Os métodos realizam a decomposição de Gauss da matriz, com a diferença de que o primeiro é utilizado para matrizes na forma consistente e o segundo na forma *lumped*.

```
MyMatrix();
```

Construtor para classe, não realiza nada.

```
MyMatrix(int,int,int);
```

Construtor que recebe tamanho do vetor, dimensão da matriz original e banda da matriz, alocando o vetor.

```
MyMatrix(int,int);
```

Construtor que recebe apenas dimensão e banda, calculando o tamanho como o produto dos dois e alocando o vetor baseado neste tamanho.

~MyMatrix();

Destrutor da classe, desaloca vetor.

bool insertValue(Real,int,int);

Insere valor na posição dada (internamente, converte o par de índices para o índice linearizado.

bool addValue(Real,int,int);

Adiciona valor na posição dada (internamente, converte o par de índices para o índice linearizado.

Real getValue(int,int);

Retorna o valor da posição dada (internamente, converte o par de índices para o índice linearizado.

void decompGauss(Real*,Real*);

(R) Realiza decomposição gaussiana para matrizes da forma consistente. Ideal seria mover este método para outra classe, talvez alguma nova que defina uma consistente genérica.

void decompGauss2(Real*,Real*);

(R) Realiza decomposição gaussiana para matrizes da forma *lumped*. Ideal seria mover este método para outra classe, talvez alguma nova que defina uma *lumped* genérica.

void print(bool);

Imprime informações sobre a matriz. Para debug.

==SymmetricMatrix==

Semelhante a classe base *MyMatrix*, acrescenta apenas a implementação do método *map* na versão para matriz simétrica.

==AsymmetricMatrix==

Semelhante a classe base *MyMatrix*, acrescenta apenas a implementação do método *map* na versão para matriz assimétrica.

==StiffnessMatrix==

Implementa a matriz de rigidez na forma consistente, sendo utilizada na análise estática e na de vibrações livres. Nos construtores é feita a alocação da memória do vetor que guarda os dados da matriz e de outras estruturas internas que variam conforme o tipo de *riser*. O

método *genMatrix* calcula os valores dos elementos da matriz, o que também varia de acordo com a configuração de *riser*. Este calculo leva em conta dois componentes, o geométrico (ligado a posição do elementos) e o elástico (ao coeficiente de elasticidade).

StiffnessMatrix(TTR*);
Construtor para configuração *TTR*. Aloca vetor e outras estruturas.

StiffnessMatrix(Tower*);
Construtor para configuração *Tower*. Aloca vetor e outras estruturas.

StiffnessMatrix(Collector*);
Construtor para configuração *Collector*. Aloca vetor e outras estruturas.

StiffnessMatrix(Pipeline*);
(P) Construtor para configuração *Pipeline*. Aloca vetor e outras estruturas.

~StiffnessMatrix();
Destrutor da classe, desaloca estruturas.

virtual Real* genMatrix(TTR*);
Calcula o valor dos elementos da matriz para a configuração *TTR*.

virtual Real* genMatrix(Tower*);
Calcula o valor dos elementos da matriz para a configuração *Tower*.

virtual Real* genMatrix(Collector*);
Calcula o valor dos elementos da matriz para a configuração *Collector*.

virtual Real* genMatrix(Pipeline*);
(P) Calcula o valor dos elementos da matriz para a configuração *Pipeline*.

template <class SpecificRiserType> void assembleMatrix(SpecificRiserType*);
Algoritmo principal do cálculo da matriz. É chamado por alguma das versões de *genMatrix*. É um *template* pois o algoritmo não varia conforme o tipo de *riser*, mas *geomMatrix* e *elastMatix* sim.

void geomMatrix(Riser*,int,Real[6][6]);
Calcula um setor 6x6 do componente geométrico da matriz. Utilizado para configurações diferentes de *Pipeline*.

void geomMatrix(Pipeline*,int,Real[6][6]);
(P) Calcula um setor 6x6 do componente geométrico da matriz. Utilizado para configuração *Pipeline*.

void elastMatrix(Riser*,int,Real[6][6]);
Calcula um setor 6x6 do componente elástico da matriz. Utilizado para configurações diferentes de *Pipeline*.

void elastMatrix(Pipeline*,int,Real[6][6]);
Calcula um setor 6x6 do componente elástico da matriz. Utilizado para configuração *Pipeline*.

==LumpedStiffnessMatrix==

Implementa a matriz de rigidez na forma *lumped*, sendo utilizada na análise dinâmica. No construtor é feita a alocação da memória do vetor que guarda os dados da matriz. O método *genMatrix* calcula os valores dos elementos da matriz a partir da versão consistente. Caso se chame o método *genMatrix* sem passar o segundo argumento (que seria a matriz consistente), o método cria a matriz consistente antes de prosseguir.

LumpedStiffnessMatrix(int);
Construtor da classe, aloca vetor interno de acordo com o número de elementos passado.

virtual Real* genMatrix(TTR*);
Gera a matriz para a configuração *TTR*. Cria uma matriz de rigidez consistente (*StiffnessMatrix*) e passa para *genMatrix(TTR*,StiffnessMatrix*)*.

virtual Real* genMatrix(Tower*);
Gera a matriz para a configuração *Tower*. Cria uma matriz de rigidez consistente (*StiffnessMatrix*) e passa para *genMatrix(Tower*,StiffnessMatrix*)*.

virtual Real* genMatrix(Collector*);
Gera a matriz para a configuração *Collector*. Cria uma matriz de rigidez consistente (*StiffnessMatrix*) e passa para *genMatrix(Collector*,StiffnessMatrix*)*.

virtual Real* genMatrix(Pipeline*);
(P) Gera a matriz para a configuração *Pipeline*. Cria uma matriz de rigidez consistente (*StiffnessMatrix*) e passa para *genMatrix(Pipeline*,StiffnessMatrix*)*.

Real* genMatrix(TTR*,StiffnessMatrix*);
Calcula para *TTR* os valores da matriz de rigidez na forma *lumped* a partir da forma consistente.

Real* genMatrix(Tower*,StiffnessMatrix*);
Calcula para *Tower* os valores da matriz de rigidez na forma *lumped* a partir da forma consistente.

Real* genMatrix(Collector*,StiffnessMatrix*);
Calcula para *Collector* os valores da matriz de rigidez na forma *lumped* a partir da forma consistente.

Real* genMatrix(Pipeline*,StiffnessMatrix*);
(P) Calcula para Pipeline os valores da matriz de rigidez na forma *lumped* a partir da forma consistente.

template <class SpecificRiserType> void assembleMatrix(SpecificRiserType*,StiffnessMatrix*);
Algoritmo principal que calcula a matriz. Chamado por *genMatrix* e utiliza os métodos que vem abaixo.

==MassMatrix==

Implementa a matriz de massa na forma consistente, sendo utilizada na análise de vibrações livres. Nos construtores é feita a alocação da memória do vetor que guarda os dados da matriz e de outras estruturas internas que variam conforme o tipo de *riser*. Os métodos *genMatrix* e *assembleMatrix* calculam os valores dos elementos da matriz, utilizando o método auxiliar *localMatrix*, que calcula um setor 6x6 da matriz. Este método é apenas virtual, sendo implementado nas subclasses *InlineMassMatrix* e *TransMassMatrix*.

==InlineMassMatrix==

Verão *inline* da classe base *MassMatrix*, acrescentando apenas a implementação do método *localMatrix* para o caso *inline*.

==TransMassMatrix==

Verão transversal da classe base *MassMatrix*, acrescentando apenas a implementação do método *localMatrix* para o caso transversal.

==LumpedMassMatrix==

Implementa a matriz de massa na forma *lumped*, sendo utilizada na análise dinâmica. Nos construtores é feita a alocação da memória do vetor que guarda os dados da matriz e de outras estruturas internas que variam conforme o tipo de *riser*. Os métodos *genMatrix* e *assembleMatrix* calculam os valores dos elementos da matriz, utilizando o método auxiliar *localMatrix*, que calcula um setor 2x2 da matriz. Este método é apenas virtual, sendo implementado nas subclasses *InlineMassMatrix* e *TransMassMatrix*.

==LumpedInlineMassMatrix==

Verão *inline* da classe base *LumpedMassMatrix*, acrescentando apenas a implementação do método *localMatrix* para o caso *inline*.

==LumpedTransMassMatrix==

Verão transversal da classe base *LumpedMassMatrix*, acrescentando apenas a implementação do método *localMatrix* para o caso transversal.

==DampingMatrix, InlineDampingMatrix e TransDampingMatrix==

Implementam a matriz de amortecimento, utilizada na análise dinâmica, sendo a primeira a base para as outras duas, específicas para o caso *inline* e transversal, respectivamente. O construtor apenas aloca a memória interna necessária e *genMatrix* gera a matriz a partir das matrizes de massa e rigidez.

==MATR1, inlineMATR1, transMATR1, MATR2, inlineMATR2, transMATR2, MATR3, inlineMATR3 e transMATR3==

Matrizes utilizadas na análise dinâmica, são calculadas a partir das matrizes de massa, rigidez e amortecimento.

Grupo Force

==Force==

Classe base que representa as forças agindo sobre o *riser*. O construtor aloca o vetor interno de acordo com o número de elementos passado como parâmetro. O método *generate*, que preenche o vetor com os valores apropriados, é apenas declarado, sendo implementado mas subclasses.

==InlineForce==

Esta subclasse de *Force* implementa a força no plano *inline*, sendo utilizada na análise

dinâmica. As forças são calculadas de acordo com a e*quação de Morrison*. O método *generate* invoca os métodos *inertia*, *drag*, *added*, e *plat*, que aplicam as diferentes cargas (vindo da inércia, arrasto, massa adicional e plataforma respectivamente).

==TransForce==

Esta subclasse de *Force* implementa a força no plano transversal, sendo utilizada na análise dinâmica. O método *generate* invoca os métodos *viv*, *drag*, *added*, que aplicam as diferentes cargas (vindo da viv, arrasto, massa adicional).

==StaticForce==

Esta subclasse de *Force* implementa a força sobre o *riser* na análise estática. Serve de base para outras duas classes mais específicas, *InlineStaticForce* e *TransStaticForce*.

==StaticInlineForce==

Implementa a força *inline* para a análise estática. O método *generate* calcula os valores apropriados.

==StaticTransForce==

Implementa a força transversal para a análise estática. O método *generate* calcula os valores apropriados.

==Load==

Utilizada na análise estática, encapsula a classe *StaticForce* e a utiliza para calcular o vetor Q. O método *gen* calcula este vetor, chamando por sua vez *load*.

Outros

==Solver==

A classe *Solver* é a classe principal do simulador e é com ela principalmente que o Módulo de Visualização interage. O método *newModel* cria um novo modelo de simulação, recebendo como parâmetro o tipo de *riser* (na forma de *string*) e o número de elementos. Com um modelo criado, é possível executar as análises chamando as funções *staticAnalysis*, *freeVibrationAnalysis* e *dynamicAnalysis*. O método time retorna o instante da simulação (da análise dinâmica) e *reset* reinicia a análise dinâmica.

==Model==

A classe *Model* encapsula a classe *Riser* e como esta, possui sublclasses para cada configuração de *riser* (*TTRModel*, *TowerModel*, *CollectorModel* e *PipelineModel*). Com elas é possível, utilizando polimorfismo e sobrecarga de método, invocar as análises sem precisar de variáveis e condicionais para fazer os cálculos de modo diferenciado conforme o tipo de *riser*.

Por exemplo, ao criar um novo modelo de *TTR* pelo método *newModel* de *Solver*, é instanciada a subclasse de *Model* apropriada, *TTRModel*, mas que fica referenciada pelo ponteiro *model* de tipo *Model**. Ou seja, o *Solver* não sabe qual o tipo especifico de *Model* está alocado. Ao chamar a análise estática pela classe *Solver*, isto é, *solver->staticAnalysis*, é chamado por ela o método *model->staticAnalysis*, que executa não o método da classe base *Model*, mas da classe especifica *TTRModel*. Este método, por fim, invoca o método *run* do objeto *staticAnalysis* passado como argumento, passando para ele o objeto data de tipo *TTR*.

==TTRModel, TowerModel, CollectorModel e PipelineModel==

Classes que representam os modelos específicos de Model.

Documentação do Módulo de Visualização

A descrição das classes do Módulo de Visualização está aqui agrupada de acordo com as classes a que pertencem. A Figura 2 mostra as principais classes do módulo e como se relacionam entre si. A classe principal é a classe *App*, que contem as classes *Interface* e *Solver*, está última importada do módulo *Solver*. A classe *Interface* por sua vez contém as classes *Scene*, que implementa a cena 3D onde é visualizada a simulação, e *Panel*, que implementa o painel de controle. A cena 3D é implementada utilizando o *VTK* e o painel utilizando o *Qt*.

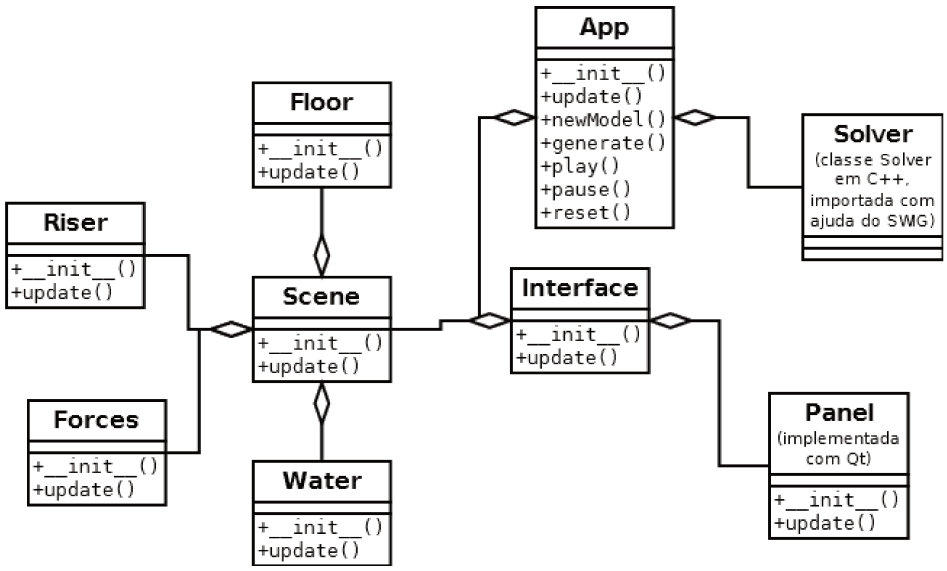

Figura 2 – classes do Módulo de Visualização

Scene.py

Contém as classes que implementam a representação virtual do cenário de operação de um *riser*. A classe *Scene* atua como o contêiner para as demais classes que instanciam os elementos específicos da visualização (*riser*, mar, forças). Além de alguns métodos mais específicos, todas possuem um construtor *__init__* e um método *update*. O primeiro cria e conecta os objetos *VTK* do *pipeline* de visualização e o segundo atualiza os dados de entrada do pipeline (p.e. atualizando posições ou escala dos elementos).

==Scene==

Classe que implementa a representação virtual do cenário de operação de um *riser*. Contém objetos que implementam os elementos específicos da cena, como o *riser*, a embarcação, o mar, o perfil de correnteza, etc.

def **__init__**(*self*,app,iren):
Construtor. Cria e inicializa o *renderer* (fornecido pelo VTK) a ser utilizado para a visualização. Cria os demais elementos da cena e adiciona seus respectivos *actors* na cena.

def **scaleToScene**(*self*,val):
Converte distâncias para a escala da cena.

def **scaleToModel**(*self*,val):
Faz o processo inverso do método anterior.

def **EndPickEvent**(*self*,obj,event):
Detecta um pick event, utilizado para selecionar um elemento do *riser*.

def **frontView**(*self*):
Posiciona a câmera em vista frontal.

def **sideView**(*self*):
Posiciona a câmera em vista lateral.

def **topView**(*self*):
Posiciona a câmera em vista superior.

def **hideShipModel**(*self*,hide):
Oculta ou mostra o modelo da embarcação.

def **showMaxDisplacements**(*self*,show):
Oculta ou mostra os máximos deslocamentos.

def **record**(*self*):

def **visualStyle**(*self*,option):
Seleciona o modo de visualização: contido no *bounding box* ou com o mar extenso.

def **hideFloaters**(*self*,hide):
Oculta ou mostra os flutuadores ao longo do *riser*.

def **hideNumbers**(*self*,hide):
Oculta ou mostra a profundidade em números na cena.

def **reset**(*self*):
Zera as linhas de máximo deslocamento.

def **update**(*self*):
Invoca *update* dos elementos da cena.

==SceneObj==

Classe base para as demais classes que representam os elementos da visualização (*riser*, mar, embarcação, etc). Implementa funcionalidades comuns a todas, como criar o *Actor* do VTK.

==Mesh==

Classe que insere na cena um modelo 3D (*mesh*) externo. Utilizada para representar a embarcação.

def **update**(*self*,timeStamp):
Atualiza a posição da embarcação, fazendo-a balançar com o movimento das ondas. No momento, apenas para propósito ilustrativo.

==Terrain==

Implementa o fundo do mar. Utiliza *Perlin Noise* sobre um plano para gerar a rugosidade e uma textura para maior realismo.

==Water==

Implementa a superficie do mar. Utiliza *Perlin Noise* sobre um plano para gerar as ondas e uma textura para maior realismo.

def **__init__**(*self*,scene,app,texture=None):
def **update**(*self*,timeStamp):
Atualiza a posição das ondas. Embora o Solver leve em conta parâmetros de onda, no momento as ondas representadas não seguem estes parâmetros, sendo apenas ilustrativas.

==Riser==

Representa o *riser*. Utiliza um *spline* para interpolar os elementos, gerando um tubo com curvatura suave. Cores são mapeadas sobre a superfície do *riser* para representar diversos dados da simulação, como forças sobre o duto ou coeficientes de arrasto.

def **update**(*self*):
Atualiza a posição do *riser* e o mapeamento de cores.

==Box==

Representa o *bounding box* que envolve os demais elementos da cena virtual (exceto no modo sem o *bounding box*).

==Force==

Representa com setas ao longo do *riser* as forças que agem sobre o tubo. Escala e orienta as setas de acordo com magnitude das componentes *inline* e transversal (isto é, as componentes não são representadas separadamente).

def **update**(*self*):
Atualiza a posição, orientação e magnitude das setas, fazendo-as acompanhar o movimento do riser.

==TopTension==

Representa com uma seta apontada para cima a tensão de topo aplicada no *riser*.

==Cursor==

Implementa uma esfera vermelha que marca o elemento do *riser* selecionado (ao apontar com o mouse e pressionar a tecla P).

==Floaters==

Representa os flutuadores dispostos sobre o *riser*.

def **update**(*self*):
Atualiza a posição dos flutuadores, para acompanharem o *riser*.

==RefPole==

Implementa linhas verticais, utilizadas p.e. para marcar a posição inicial do *riser*.

==MaxProfile==

Implementa as linhas de máximo deslocamento,

Panel.py

O arquivo *Panel.py* contém uma única classe, *Panel*, que implementa o painel de controle do módulo de visualização, utilizando diversos recursos do *Qt*. O método *__init__* inicia as estruturas do *Qt* utilizadas, criando o painel e as funcionalidades dos *widgets*. Grande parte do painel foi criada através do ambiente visual *QtCreator*, de modo que o código apenas carrega as informações contidas no arquivo *Panel.ui* gerado pelo *QtCreator*. Para atribuir as funcionalidades dos *widgets*, é utilizado o método *connect* do *widget* correspondente, passando-se a função a ser chamada quando o *widget* for utilizado. Os demais métodos da classe *Panel*, de um modo geral, implementam as funções chamadas pelos *widgets*, ou são métodos auxiliares.

Interface.py

O arquivo *Interface.py* contém uma única classe, *Interface*, que implementa a interface do módulo de visualização. Serve de contêiner para dois outros objetos, um da classe *Panel* e outro da classe *Scene*, que implementam respectivamente o painel de controle e a cena 3D onde é feita a visualização. Implementa também funcionalidades da barra de menu superior.

main.py

O arquivo *main.py* contém uma única classe, *App*, que é a classe principal do módulo. Serve de contêiner para a classe Interface e para a classe *Solver*, esta última importada do módulo *Solver*.