JOÃO LUIZ BERNARDES JÚNIOR

Desenvolvimento de um ambiente para visualização tridimensional da dinâmica de *risers*

Dissertação apresentada à Escola Politécnica da Universidade de São Paulo para obtenção do título de Mestre em Engenharia.

Área de Concentração: Engenharia Mecânica
Orientador: Prof. Dr. Clóvis de Arruda Martins

São Paulo
2004

FICHA CATALOGRÁFICA

Bernardes Júnior, João Luiz
Desenvolvimento de um ambiente para visualização tridimensional da dinâmica de risers / J.L. Bernardes Júnior. -- São Paulo, 2005.
202 p.

Dissertação (Mestrado) - Escola Politécnica da Universidade de São Paulo. Departamento de Engenharia Mecânica.
Orientador: Prof. Dr. Clóvis de Arruda Martins

1.Detecção de colisões 2.Engenharia de programação 3.Realidade virtual 4.Risers 5.Vizualização científica 6.Vórtices I.Universidade de São Paulo. Escola Politécnica. Departamento de Engenharia Mecânica II.t.

FOLHA DE APROVAÇÃO

João Luiz Bernardes Júnior

Desenvolvimento de um ambiente
para visualização tridimensional
da dinâmica de *risers*

Dissertação apresentada à Escola Politécnica da Universidade de São Paulo para obtenção do título de Mestre em Engenharia.

Prof. Dr. _____

Instituição: _____ Assinatura: _____

Prof. Dr. _____

Instituição: _____ Assinatura: _____

Prof. Dr. _____

Instituição: _____ Assinatura: _____

DEDICATÓRIA

Dedico este trabalho a meus pais, João Luiz e Diana, por seu amor, paciência e apoio persistentes e inefáveis ao longo dos anos, sem os quais a realização deste trabalho, desde seu início, teria sido impossível.

AGRADECIMENTOS

Ao Prof. Dr. Clóvis de Arruda Martins, por todas as lições ao longo dos anos e por sua paciência, camaradagem e apoio durante sua inestimável orientação deste trabalho.

Aos membros do Grupo de Pesquisas de Jogos do Interlab-USP, por sua contribuição em diversas questões relacionadas a Engenharia de *Software*, Computação Gráfica e *Hardware* gráfico pertinentes à elaboração deste trabalho.

Ao Prof. Dr. Júlio Romano Meneghini, por seu auxílio na área de dinâmica dos fluidos.

Ao Prof. Dr. Marcelo Knorich Zuffo, por ter me introduzido a técnicas de visualização científica e principalmente ao VTK.

A Escola Politécnica da Universidade de São Paulo, principalmente a seu departamento de Engenharia Mecânica, por permitir a realização deste trabalho em seu programa de mestrado.

À Coordenação de Aperfeiçoamento de Pessoal de Nível Superior do Ministério da Educação, por conceder bolsa de mestrado para a realização deste trabalho.

RESUMO

BERNARDES JR., J. L. **Desenvolvimento de um ambiente para visualização tridimensional da dinâmica de *risers***. 2004. 202 f. Dissertação (Mestrado) - Escola Politécnica, Universidade de São Paulo, São Paulo, 2004.

A importância da exploração marítima de petróleo, em especial para o Brasil, é indiscutível e risers são estruturas essenciais para essa atividade. Uma melhor compreensão da dinâmica dessas estruturas e dos esforços a que estão submetidas vem resultando de pesquisa constante na área, pesquisa que gera um grande volume de dados, freqüentemente descrevendo fenômenos de difícil compreensão. Este trabalho descreve o desenvolvimento de um ambiente que combina técnicas de realidade virtual (como ambientes 3D, navegação e estereoscopia) e visualização científica (como mapeamento de cores, deformações e glifos) para facilitar a visualização desses dados. O ambiente, batizado como RiserView, permite a montagem de cenas tridimensionais compostas por risers, relevo do solo, superfície marítima, embarcações, bóias e outras estruturas, cada um com sua dinâmica própria. Permite ainda a visualização do escoamento para que a formação de vórtices na vizinhança dos risers e a interação fluido-mecânica resultante possam ser estudadas. O usuário pode controlar parâmetros da visualização de cada elemento e da animação da cena, bem como navegar livremente por ela. Foi desenvolvido também um algoritmo de baixo custo computacional (graças a simplificações possíveis devido à natureza do problema) para detecção e exibição em tempo real de colisões entre risers. O Processo Unificado foi adaptado para servir como metodologia para o projeto e implementação do aplicativo. O uso do VTK (API gráfica e de visualização científica) e do IUP (API para desenvolvimento de interfaces com o usuário) simplificou o desenvolvimento, principalmente para produzir um aplicativo portável para MS-Windows e Linux. Como opções de projeto, a visualização científica e a velocidade na renderização das cenas são privilegiadas, ao invés do realismo e da agilidade na interação com o usuário. As conseqüências dessas escolhas, bem como alternativas, são discutidas no trabalho. O uso do VTK e, através dele, do OpenGL permite que o aplicativo faça uso dos recursos disponíveis em placas gráficas comerciais para aumentar sua performance. Em sua versão atual a tarefa mais custosa para o RiserView é a atualização das posições de risers, principalmente descritos no domínio da freqüência, mas o trabalho discute aprimoramentos relativamente simples para minimizar esse problema. Apesar desses (e de outros) aprimoramentos possíveis, discutidos no trabalho, o ambiente mostra-se bastante adequado à visualização dos risers e de sua dinâmica bem como de fenômenos e elementos a eles associados.

Palavras-chave: Detecção de Colisões. Engenharia de Programação. Realidade Virtual. Risers. Visualização Científica. Vórtices.

ABSTRACT

BERNARDES JR., J. L. **Development of an environment for tridimensional visualization of riser dynamics**. 2004. 202 f. Dissertation (Master's) - Escola Politécnica, Universidade de São Paulo, São Paulo, 2004.

The importance of offshore oil exploration, especially to Brazil, cannot be argued and risers are crucial structures for this activity. A better understanding of the dynamics of these structures and of the efforts to which they are subject has been resulting from constant research in the field, research that generates a large volume of data, often describing phenomena of difficult comprehension. This work describes the development of a software environment that combines elements of virtual reality (3D environments, navigation, stereoscopy) and scientific visualization techniques (such as color mapping, deformations and glyphs) to improve the understanding and visualization of these data. The environment, christened RiserView, allows the composition of tridimensional scenes including risers, the floor and surface of the ocean and ships, buoys and other structures, each with its own dynamics. It also allows the visualization of the flow in the neighborhood of the risers so that vortex shedding and the resulting fluid-mechanic interactions may be studied. The user may control parameters of the scene animation and of the visualization for each of its elements, as well as navigate freely within the scene. An algorithm of low computational cost (thanks to simplifications possible due to the nature of the problem), for the detection and exhibition of collisions between risers in real time, was also developed. The Unified Process was adapted to guide the software's project and implementation. The use of VTK (a scientific visualization and graphics API) and IUP (a user interface development API) simplified the development, especially the effort required to build an application portable to MS-Windows and Linux. As project choices, scientific visualization and the speed in rendering scenes in real time were given higher priority than realism and the agility in the user interaction, respectively. The consequences of these choices, as well as some alternatives, are discussed. The use of VTK and, through it, OpenGL, allows the application to access features available in most commercial graphics cards to increase performance. In its current version, the most costly task for RiserView are the calculations required to update riser positions during animation, especially for risers described in the frequency domain, but the work discusses relatively simple improvements to minimize this problem. Despite these (and other) possible improvements discussed in the work, the application proves quite adequate to the visualization of risers and their dynamics, as well as of associate elements and phenomena.

Keywords: Collision Detection. Risers. Scientific Visualization. Software Engineering. Virtual Reality. Vortices.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

SUMÁRIO

1. INTRODUÇÃO

Como seu título explicita, este trabalho consiste no "projeto e desenvolvimento de um ambiente para a visualização em três dimensões da dinâmica de *risers*". Neste capítulo inicial esta afirmação é explicada de forma mais clara. Embora só caiba à introdução uma abordagem de teor mais geral, que é aprofundada nos próximos capítulos, seu objetivo é fornecer ao leitor uma imagem bem definida do que este trabalho se propõe a fazer, de quais são seus objetos de interesse, do porquê de sua execução bem como da maneira em que esta execução é realizada.

Na primeira seção deste capítulo os objetivos do trabalho, que problema que este se presta a resolver, são mais bem detalhados. Em seguida é descrito o contexto em que este trabalho se insere. Este contexto é importante não só para que se compreenda algumas das decisões tomadas durante o projeto, como também para explicar a sua motivação.

A metodologia utilizada, ou seja, o modo como o trabalho é realizado, é discutida em seguida e, por fim, a estrutura do trabalho é apresentada, explicitando a forma como seu conteúdo está organizado nos capítulos seguintes.

Como o primeiro parágrafo afirma, já a partir desta próxima seção (que descreve os objetivos) são discutidos neste capítulo diversos temas que serão retomados com

mais profundidade em outros pontos do trabalho. Quando isso ocorrer, o ponto em que esta discussão mais aprofundada ocorre será citado aqui.

1.1. Objetivos

O objeto de maior interesse neste trabalho são os *risers*. Embora essas estruturas sejam discutidas com mais detalhe em 2.1, a discussão dos objetivos e motivação deste trabalho faz necessária a sua definição. *Risers* são estruturas tubulares utilizadas na exploração marítima de petróleo para fazer a ligação entre os poços, no solo oceânico, e as plataformas ou navios na superfície.

Durante seu uso, essas estruturas estão sujeitas a diversos esforços internos e externos. Estes últimos são causados principalmente por fatores de origem ambiental, como ondas e correntes marítimas, que por sua vez causam movimentos na plataforma e podem causar esteiras de vórtices ao redor dos *risers* (vórtices são discutidos na subseção 2.1.2). Estes esforços a que o *riser* está submetido são de natureza bastante complexa e determinam seu projeto e sua vida útil. Além disso, no Brasil produz-se em profundidades raramente alcançadas em outras partes do mundo. Por tudo isso é necessária uma intensa pesquisa para compreender e modelar a dinâmica dos *risers* e de como respondem aos esforços a que estão submetidos.

Esta pesquisa, bem como as diversas ferramentas computacionais de análise dos *risers* que resultam dela, geram um grande volume de dados. Dados estes de natureza tridimensional ou até envolvendo mais dimensões, já que podem variar no tempo além

do espaço e serem grandezas não só escalares mas também vetoriais ou tensoriais. Além disso, muitas vezes traduzem fenômenos não-lineares ou cuja compreensão não é intuitiva. E é este o problema que é atacado por este trabalho: a dificuldade de compreensão dos grandes volumes de dados multidimensionais resultantes das pesquisas e análises da dinâmica de *risers*.

Para resolver este problema, propõe-se, como objetivo deste trabalho, o desenvolvimento de um ambiente interativo para visualização tridimensional de *risers*, isoladamente ou em conjunto, comportando-se de acordo com uma dinâmica calculada externamente.

Sua entrada são os resultados de outros programas que estudam a interação do *riser* com o escoamento onde está imerso, o movimento e a dinâmica resultantes dessa interação. O ambiente servirá como uma saída gráfica para esses programas.

Como *risers* podem trabalhar em conjunto, em diversas configurações espaciais, o que pode levar a colisões entre eles, o ambiente também deve permitir a detecção e visualização dessas colisões. Além disso, deve mostrar, também, diversos elementos associados aos *risers*, como por exemplo embarcações, bóias, o fundo e a superfície do oceano e vórtices. A visualização científica[1] da dinâmica do *riser* deve ser privilegiada, ao invés do realismo da cena. A visualização científica é discutida com mais cuidado na seção 2.2.

---

[1] O processo de explorar, transformar e exibir dados como imagens ou outras formas sensoriais para facilitar o entendimento e a percepção desses dados.

Além disso, o programa deve ser desenvolvido seguindo uma metodologia formal de projeto, deve ser eficiente, expansível e portável para as plataformas MS Windows e Linux. A metodologia utilizada é discutida brevemente na seção 1.3. Seus aspectos teóricos são discutidos em 3.1 e o quarto capítulo é inteiramente dedicado a mostrar como esta metodologia foi aplicada durante o desenvolvimento do projeto e como o *software* resultante é descrito através dela. A discussão da questão da portabilidade do código pode ser encontrada na seção 2.6 e a solução adotada para esse problema é discutida em 3.2.

Esses requisitos do projeto são definidos de maneira mais completa e formal no capítulo 4. A figura 1, abaixo, ilustra os objetivos do trabalho através das entradas e principalmente das saídas do aplicativo a ser desenvolvido, batizado de **RiserView**. Na figura, a cena com um *riser* renderizado em cinza representa as tarefas de renderização.

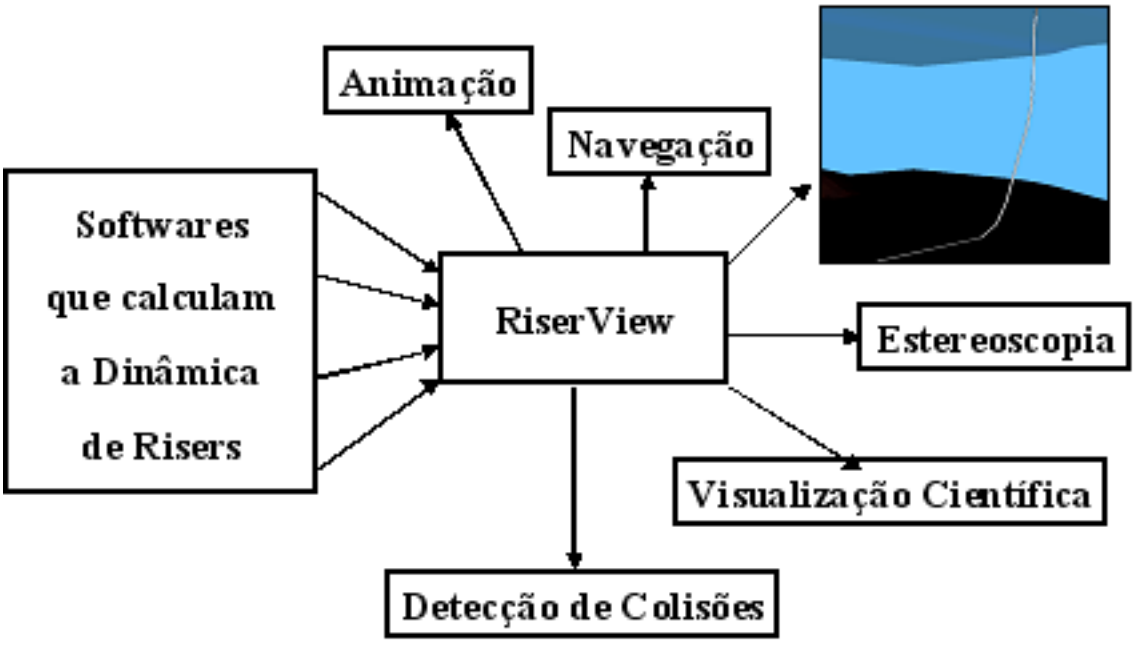

Figura 1 - Entradas e saídas do RiserView

1.2. Contexto e Motivação

Como dito anteriormente, a compreensão do contexto em que este trabalho se insere é importante não só para compreensão dos fatores que o motivaram mas também para compreender algumas das decisões tomadas durante o projeto.

Em primeiro lugar é preciso analisar a importância da exploração marítima de petróleo no Brasil. Acredita-se não ser necessário discutir a importância do petróleo em si para o mundo moderno, portanto a atenção aqui está voltada especificamente à sua exploração marítima. Essa atividade é de grande importância para o Brasil (e também para outros países), tanto do ponto de vista econômico quanto do estratégico. Somente 18% do petróleo produzido no país, de acordo com a Petrobras (2004), é extraído em terra firme. Do grande volume restante, retirado do leito oceânico, a maior parte é produzida em águas profundas (com profundidade acima de quatrocentos metros) onde também se encontram boa parte das reservas brasileiras ainda não exploradas.

Além disso, o ritmo em que esta atividade vem evoluindo é bastante rápido. Há apenas trinta e cinco anos essa modalidade de exploração tinha início no Brasil a uma profundidade de menos de trinta metros. De acordo com a Petrobras (2004), ela é atualmente a campeã mundial em profundidade de exploração, produzindo a quase dois mil metros no campo de Roncador.

O grande volume de produção, bem como a evolução desta atividade, são ao mesmo tempo causa e conseqüência de um grande número de pesquisas em diversas áreas. A Escola Politécnica, em particular, vem realizando pesquisas nesta área há mais

de uma década, envolvendo mais de um departamento e inclusive projetos temáticos da FAPESP, constantemente produzindo trabalhos na área e obtendo resultados importantes.

A existência de todos estes trabalhos e pesquisas é outro importante motivador para o desenvolvimento deste trabalho em particular. São eles, afinal, que geram o grande volume de dados que faz necessário, para sua compreensão, o uso de uma ferramenta como a que se propõe desenvolver neste mestrado. Aliás, este é um dos motivos pelos quais se faz a opção de projeto pela visualização científica, em vez do realismo.

Além disso, as mesmas necessidades que motivam todos estes trabalhos (não só a importância mas a evolução da exploração marítima de petróleo e suas altas profundidades, que emprestam à atividade no país características únicas) são também fortes razões para a proposta discutida aqui.

Uma outra forma na qual a existência destes trabalhos serve como estímulo é a seguinte: espera-se que seus desenvolvedores possam, no futuro, fazer uso do produto deste projeto e que esse uso contribua para seus trabalhos de alguma forma. A possibilidade que a ferramenta desenvolvida venha a ser de fato útil no futuro é um grande motivador.

A importância que estas pesquisas desenvolvidas na Escola Politécnica têm como motivador para este trabalho pode ser demonstrada através de uma conseqüência direta delas numa decisão de projeto. Muitas destas pesquisas, principalmente as mais recentes, vêm sendo desenvolvidas utilizando o sistema operacional Linux. Atualmente

já se pode contar inclusive com dois *clusters* de computadores, totalizando mais de duzentas máquinas trabalhando em paralelo e utilizando este sistema, assim como um projetor estereográfico. Daí a importância do desenvolvimento de uma ferramenta que também possa ser utilizada sobre o Linux. Esta importância foi traduzida no projeto através do requisito de que a aplicação deveria ser multiplataforma.

Um outro fator importante, de caráter mais pessoal, que motiva o desenvolvimento deste trabalho é o envolvimento prévio do autor com as pesquisas na área citadas anteriormente, realizadas na Escola Politécnica. Desde 1995 o autor participa de pesquisas nessa área, onde espera ter adquirido um conhecimento do problema que será útil neste trabalho.

Durante a pesquisa bibliográfica pôde-se perceber também uma deficiência acadêmica com relação à visualização científica de dados nessa área em particular. Ainda que o autor conheça mais de um *software* que faça uso destas técnicas para exibir seus resultados e tenha inclusive trabalhado no desenvolvimento de pelo menos um deles, foi muito difícil encontrar artigos discutindo este aspecto. Ao que parece, os desenvolvedores das ferramentas de análise da dinâmica dos risers, consideram este seu foco principal, enquanto a visualização dos resultados é "apenas uma ferramenta secundária" que merece muito pouca discussão na literatura, se alguma. Por outro lado, há o desenvolvimento das técnicas e aplicações para visualização dos dados, sem a preocupar com a análise que os gera. Para estes os *risers* são "apenas uma aplicação" para suas técnicas e também recebem pouca atenção na literatura, que neste caso enfoca mais as técnicas em si. Some-se a isso o fato de que muitas dessas pesquisas são desenvolvidas a serviço da indústria de Petróleo de forma que as aplicações resultantes

são de uso restrito. Espera-se, com este trabalho, contribuir apresentando uma visão mais integrada destes dois aspectos. Ainda que o trabalho se concentre na visualização científica, em nenhum momento o objeto desta visualização é esquecido ou posto de lado. Esta possível contribuição através de uma visão integrada da visualização científica e de sua aplicação no problema dos *risers* é outro estímulo para o trabalho.

A própria utilização de uma metodologia formal no desenvolvimento do aplicativo é um dos objetivos deste trabalho, pelos motivos discutidos adiante.

1.3. Metodologia

Como pesquisa científica que é, este projeto deve contar fortemente com duas características: a boa documentação do projeto bem como mecanismos que auxiliem em sua repetibilidade, ou seja, que permitam que outros pesquisadores possam alcançar resultados semelhantes seguindo aproximadamente os mesmos passos trilhados aqui.

Para conseguir estas características num projeto que trata, em grande parte, de desenvolvimento de *software*, optou-se por utilizar uma metodologia formal para este desenvolvimento, que norteie os passos do processo e que esteja bem clara tanto para o desenvolvedor quanto para o leitor.

Optou-se ainda por utilizar uma metodologia dentre as já existentes e conhecidas no mercado. Na seção 3.1 a questão do uso da metodologia, suas vantagens e desvantagens, é abordada com mais profundidade. Nesta seção o Processo Unificado (*Unified Process* ou UP) é descrito de maneira breve e explica-se como foi também

analisada outra alternativa de metodologia para este trabalho: o *Extreme Programming* (XP), e porque o Processo Unificado foi escolhido. Por tudo isso, a metodologia não será novamente descrita aqui.

Outro aspecto importante que norteia todo o desenvolvimento do trabalho, seu projeto e implementação, e que inclusive permite que se utilize o UP, é a decisão de utilizar neste projeto o paradigma de programação orientada a objetos, abordado na seção 2.4.1.

1.4. Estrutura do Trabalho

Neste primeiro capítulo, **Introdução**, é oferecida uma rápida visão panorâmica do trabalho, seus objetivos, contexto e motivação, metodologia e estrutura.

O capítulo seguinte, **Revisão Bibliográfica**, discute com mais detalhe os principais tópicos envolvidos na pesquisa e execução deste trabalho, baseado na literatura, como a exploração marítima de petróleo e risers, computação gráfica e visualização científica, orientação a objetos e alguns tópicos relacionados a engenharia de *software* etc.

No capítulo três, **Tecnologias**, a revisão bibliográfica continua, porém discutindo questões mais intrinsecamente ligadas à tecnologias atuais e portanto, menos perenes. O Processo Unificado e as APIs utilizadas nesse trabalho são os principais tópicos desse capítulo.

O quarto capítulo, **a Metodologia e o RiserView**, aborda os aspectos mais práticos do projeto e implementação do ambiente de visualização de risers. Descreve-se a customização do UP para este projeto em particular, os passos que foram seguidos para atingir os resultados finais, bem como os artefatos resultantes do processo. Esta descrição e estes artefatos servem também para descrever abrangentemente e de diversas formas a aplicação resultante.

O capítulo cinco, **Resultados**, relata principalmente como foram realizados os testes do aplicativo desenvolvido, os motivos para que fossem realizados dessa forma e os resultados obtidos.

Uma discussão do trabalho como um todo e dos resultados relatados no capítulo cinco de forma mais aprofundada é feita no último capítulo, **Conclusões**. Parte importante deste capítulo é também a discussão de possíveis aprimoramentos passíveis de serem pesquisados e implementados em trabalhos futuros.

Os Apêndices A e B são resultantes da captura de requisitos realizada durante a aplicação da metodologia ao projeto. O Apêndice C resume os resultados dos testes de caixa-preta e o Apêndice D mostra como estão organizados os conteúdos do CD que é parte integrante deste trabalho.

2. REVISÃO BIBLIOGRÁFICA

Para guiar a revisão bibliográfica neste trabalho, é útil ter em mente o problema que ele se presta a resolver: a visualização do grande volume de dados resultante de diversas formas de análise relativas a *risers* em seu ambiente de trabalho. A forma como se propõe a resolver esse problema é através do desenvolvimento de um ambiente interativo para visualização tridimensional de *risers*, isoladamente ou em conjunto, movendo-se de acordo com uma dinâmica calculada externamente, que permita também a visualização das colisões entre os *risers* assim como de outros elementos na cena (como embarcações, o fundo e a superfície do oceano e vórtices). Além disso a visualização científica da dinâmica do *riser* é privilegiada, ao invés do realismo da cena. No entanto, como Upson (1987) já previa, estes dois fatores (visualização científica e realismo) não são necessariamente exclusivos e muitas técnicas utilizadas na indústria do entretenimento para aumentar o realismo da computação gráfica podem também ser utilizadas para facilitar a análise de dados científicos. Sendo assim, alguns elementos para aumentar o realismo, serão detalhados mais adiante. Além disso, o aplicativo deve ser desenvolvido seguindo uma metodologia formal de projeto, deve ser eficiente, expansível e portável para múltiplas plataformas (principalmente MS Windows e Linux). Tendo isto em mente, foi decidido também que o ambiente será desenvolvido utilizando conceitos de orientação a objetos.

A execução destes objetivos envolve o estudo de diversos tópicos. Neste capítulo, estes tópicos são apresentados de forma sucinta, organizados do geral para o específico. Inicialmente, são discutidos os objetos da visualização, os *risers*, e seu papel na exploração submarina de petróleo. Em seguida, conceitos de visualização e de projeto de *software* e por fim alguns detalhes e técnicas levados em conta durante a implementação.

2.1. Risers e a Exploração Marítima de Petróleo

A descobertas de jazidas petrolíferas sob o solo dos oceanos, cada vez em maiores profundidades, não só é de grande importância econômica e estratégica para diversos países como também exige soluções técnicas viáveis e eficientes para sua exploração. No Brasil, esta exploração se iniciou em 1968, a meros 30 metros de profundidade, aproximadamente, nos litorais do Espírito Santo e de Sergipe. Atualmente, de acordo com a Petrobras (2004), ela é líder mundial em exploração de petróleo a grandes profundidades, produzindo a 1.853 metros de profundidade no campo de Roncador. O Brasil está entre os poucos países que dominam todo o ciclo de perfuração submarina em águas profundas e ultraprofundas e atualmente 46% das reservas de petróleo do país estão localizados em profundidade de água de 400 a 1.000 m e 29,9% em profundidade de água com mais de 1.000 m, ou seja, mais de 75% de todas as reservas se encontram em águas profundas e ultraprofundas. Os diagramas apresentados nas figuras 2 e 3 demonstram a situação atual e a evolução da produção de petróleo em águas profundas.

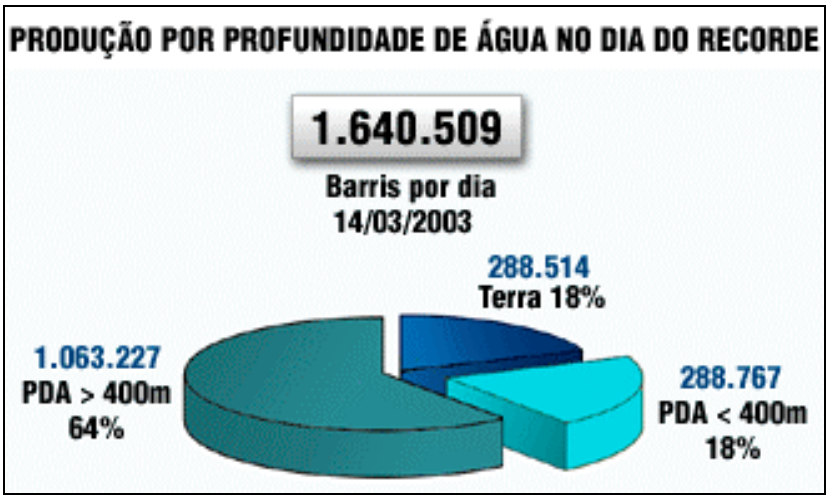

Figura 2 - Produção de petróleo no Brasil (*fonte: Petrobras*)

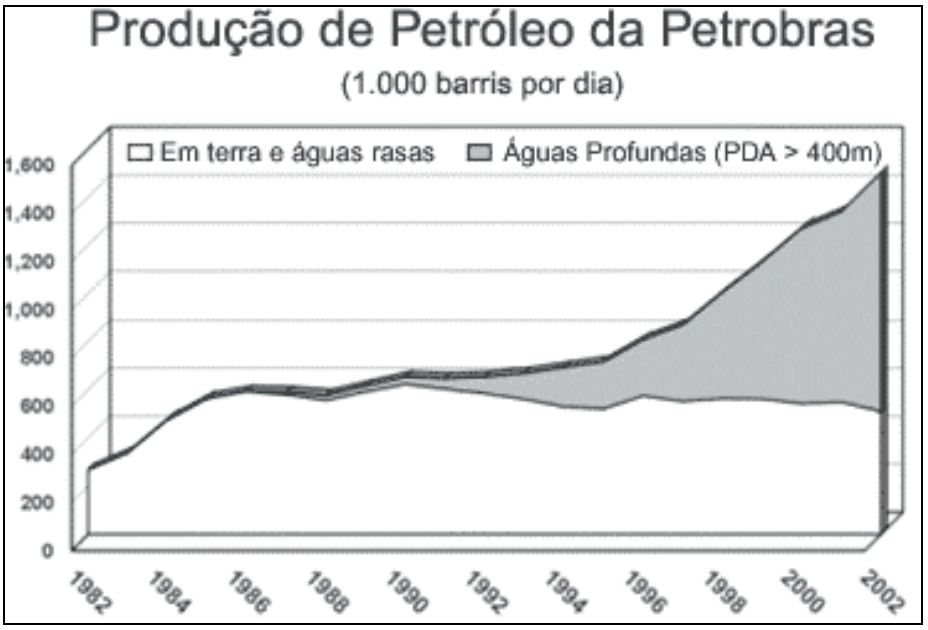

Figura 3 - Evolução da produção em águas profundas (*fonte: Petrobras*)

Esta evolução faz necessária uma intensa pesquisa em diversas áreas, dentre elas o comportamento de estruturas mecânicas submetidas aos esforços de diferentes naturezas que caracterizam a exploração em grandes profundidades. A Escola Politécnica vem realizando pesquisa nesta área há mais de uma década, tendo produzido diversos trabalhos e resultados importantes. Como já foi mencionado na introdução, essas pesquisas são mais um fator que motiva a criação do ambiente de visualização proposto neste trabalho.

Para produzir petróleo em profundidades tão diversas como as citadas anteriormente são utilizadas plataformas de exploração, que podem ser fixas ou flutuantes. As fixas têm fundação no leito oceânico e resistem melhor aos esforços ambientais, mas estão restritas a profundidades baixas ou intermediárias. Já as plataformas flutuantes são construídas sobre cascos ancorados ao fundo do mar. Além de poderem trabalhar em maiores profundidades, podem ser transportadas de um poço a outro sem maiores problemas, mas são mais sensíveis aos esforços ambientais.

Para transportar fluidos e potência entre os campos de exploração no leito oceânico e as plataformas ou navios localizados na superfície é que são utilizados os *risers*, que serão descritas em mais detalhe a seguir.

2.1.1. Risers

Ao exercer sua função, ligar os poços de exploração no solo oceânico às plataformas ou navios na superfície, os *risers* ficam submersas na lâmina d'água em

diversas configurações e são utilizados principalmente para perfuração e produção. Antes de discutir estas funções, no entanto, é interessante classificar os *risers* em rígidos e flexíveis.

Os ***risers* rígidos**, quando usados como risers verticais para perfuração, não são projetados para suportar grandes curvaturas, de acordo com Ferrari (1998). Têm um tensionador em seu topo, para evitar a flambagem e esforços elevados, e uma junta articulada em sua extremidade inferior que permite deflexões de até 10 graus se necessário. Caso o movimento da plataforma ameace causar uma deflexão maior que esta estrutura pode suportar, o *riser* pode ser desconectado hidraulicamente do poço. No entanto, *risers* rígidos também tem sido utilizados como uma importante solução para exploração em grandes profundidades, em catenária[2].

Já os ***risers* flexíveis** são projetados para que esta flexibilidade auxilie a suportar os movimentos da plataforma e os esforços hidrodinâmicos. Portanto devem ter alta rigidez axial e baixa resistência à flexão.

Na atividade de **perfuração** são utilizados os *risers* rígidos e verticais. Em sua extremidade são instaladas brocas guiadas por dispositivos na cabeça do poço. Nestes *risers* é injetada a Lama de Perfuração, um composto de argila, produtos químicos e água que tem diversas funções como lubrificar e resfriar a broca, fornecer sustentação ao *riser* e impedir a subida dos hidrocarbonetos quando atingidos.

---

[2] A curva formada por um fio suspenso.

Já na **produção** podem ser usados *risers* flexíveis ou rígidos. Durante esta atividade o papel do *riser* é transportar os produtos fluidos do poço até a plataforma.

A figura 4 ilustra um *riser* vertical e um *riser* em catenária ligando uma plataforma ao poço:

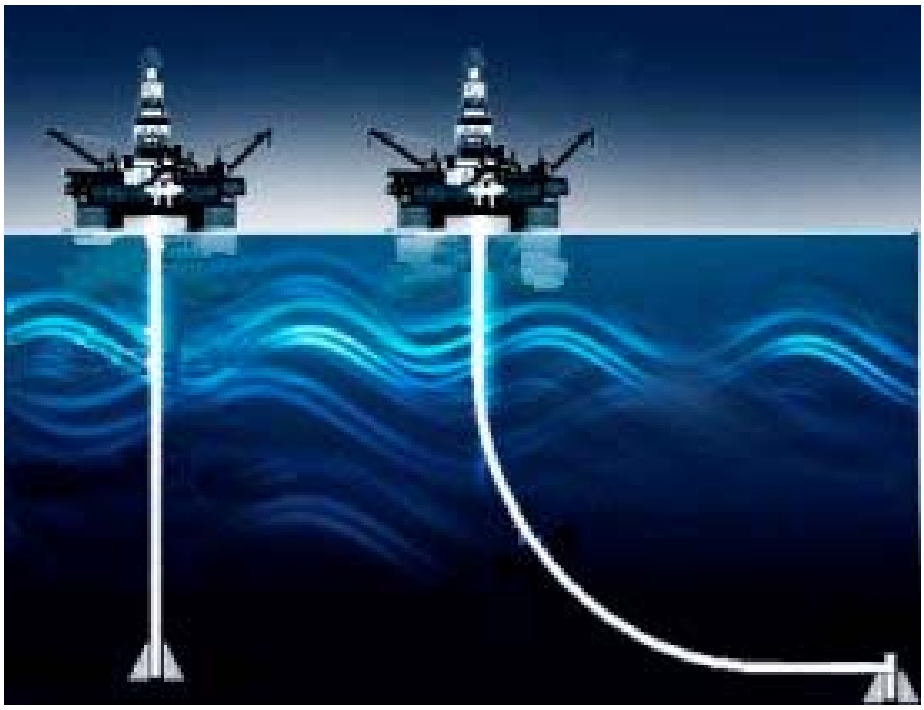

Figura 4 - *Risers*

Seja durante a perfuração ou durante a produção, os *risers* estão sujeitos a diversos esforços internos e externos. Os internos provém principalmente da pressão hidrostática dos fluidos transportados em seu interior. Os externos são causados por vários fatores, principalmente de origem ambiental, como ondas e correntes marítimas, que por sua vez causam movimentos na plataforma e podem causar esteiras de vórtices ao redor dos *risers*.

2.1.2. Vórtices

Blevins (1990) conta que desde a Grécia antiga o fenômeno de desprendimento de vórtices num escoamento ao redor de um corpo rombudo já é conhecido. Corpos rombudos são aqueles em que a separação do escoamento ocorre ao longo de uma grande parcela de sua superfície. No caso de interesse deste trabalho, os *risers*, com sua seção circular, podem ser classificados como tal. De acordo com Ferrari (1998), quando a camada limite se separa formam-se duas camadas cisalhantes que se deslocam à jusante do escoamento de cada um dos lados do cilindro. Estas camadas tendem a revolver-se formando uma esteira de vórtices à jusante do cilindro. Gerrard (1986) *apud.* Meneghini (1993) afirma que é a interação entre as duas camadas cisalhantes a principal responsável pela geração e emissão destes vórtices.

Jeong & Hussain (1995) discutem como a própria definição do que constitui um vórtice é problemática e apresentam algumas alternativas, a maioria mais complexa que o necessário para este trabalho. Citam também, no entanto, a seguinte definição: um vórtice é um conjunto de partículas materiais que gira ao redor de um centro comum. A figura 5, gerada pelo **RiserView**, ilustra o desprendimento de vórtices à jusante de um cilindro (em preto, visto de cima). As regiões em rosa e verde representam vórtices que se desprendem na periferia do cilindro e se deslocam para a direita da figura. As regiões em rosa giram no sentido horário enquanto as verdes giram no sentido contrário.

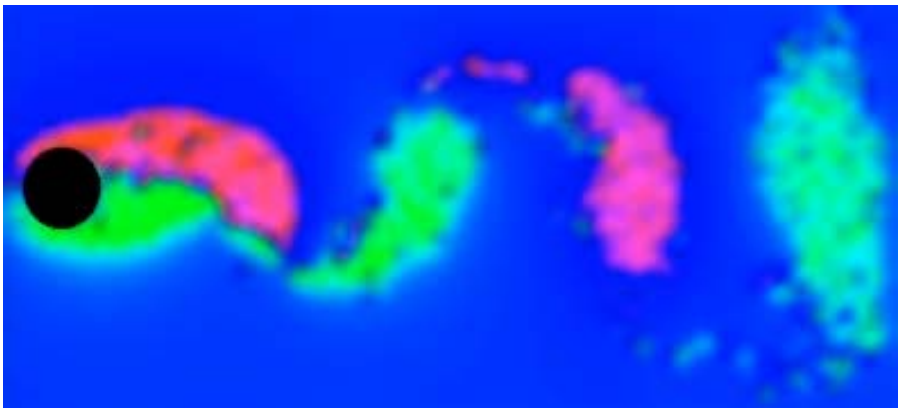

Figura 5 - Vórtices

Na figura 5 a variável que está mapeada em cores é a vorticidade, uma medida local da rotação do escoamento.

A formação e a existência de uma esteira de vórtices como esta gera forças cíclicas sobre o *riser* que causam o fenômeno de Vibrações Induzidas por Vórtices (*Vortex-Induced Vibrations* ou VIV). Como essas forças contribuem para reduzir a vida útil dos *risers* devido à fadiga, o estudo desse fenômeno é considerado de grande importância para a indústria petrolífera.

2.2. Computação Gráfica e Visualização Científica

Com o objeto da visualização, os *Risers*, seu contexto e importância definidos, é possível voltar a atenção para a visualização em si. Esta discussão começa com a área da Computação Gráfica.

Schroeder et al. (1998) dão uma definição bastante simples de **Computação Gráfica**. Para eles, essa área engloba nada mais ou menos que o processo de criar

imagens utilizando um computador. Sua definição inclui tanto técnicas de desenho ou pintura em duas dimensões quanto técnicas mais sofisticadas de desenho (ou renderização) em 3D. Foley et al. (1996) acrescentam que a área também engloba, além da criação, o armazenamento e a manipulação de modelos e imagens de objetos. Quando se fala em imagens bidimensionais, deve-se também mencionar a área de **Processamento de Imagens** (*imaging* ou *image processing*), que inclui técnicas para transformar, analisar e aprimorar essas imagens.

Angel (2000) classifica as principais aplicações de computação gráfica nas seguintes áreas:

- Exibição de Informação;
- Projeto;
- Simulação;
- Interfaces Homem-Máquina.

De acordo com essa classificação, aplicações de entretenimento, por exemplo, seriam um caso particular de exibição de informação (onde a informação seria um filme, uma história, um comercial...) ou de simulação (como é o caso de alguns jogos). Já um sistema de CAD é claramente uma aplicação de projeto e um simulador médico ou militar, uma aplicação de simulação.

Para a aplicação desenvolvida nesse trabalho, fica claro que a principal área de interesse é a de exibição de informação. Embora seja feito uso de interfaces gráficas

com o usuário, elas são apenas uma ferramenta, enquanto a exibição de informação é o principal objetivo do trabalho.

Sobre essa área, Angel (2000) afirma que:

"Técnicas gráficas clássicas surgiram como um meio de transmitir informação entre pessoas. Apesar das linguagens escrita e falada servirem a um propósito similar, ***o sistema visual humano não tem paralelo tanto como um processador de dados quanto como um reconhecedor de padrões.****"* (Angel, 2000, p. 2, grifo nosso)

Com a grande quantidade de dados que os computadores de hoje em dia permitem ser gerados por pesquisadores, cada vez mais é necessário fazer uso dessa ferramenta (o sistema visual humano) para assimilar e interpretar essas informações. Angel conta ainda, como exemplo, que, na área de escoamento de fluidos, imagens geradas pela conversão de dados em entidades geométricas que podem ser exibidas têm permitido novas revelações sobre processos complexos.

Esta é justamente a função da área de **Visualização Científica**. De acordo com Schroeder et al., "*Visualização é o processo de explorar, transformar e exibir dados como imagens (ou outras formas sensoriais) para ganhar entendimento e percepção desses dados.*" (Schroeder et al., 1998, p. 5)

Keim (2001) relata os resultados do projeto "How Much Information" da Universidade da Califórnia que estimam que anualmente, hoje em dia, em todo mundo, é gerado 1 Exabyte (1 milhão de Terabytes) de dados e que praticamente 100% desses dados está disponível em forma digital. Keim concorda com Angel quando diz que a representação visual desses dados reduz o trabalho cognitivo necessário não só na assimilação e interpretação desses dados, como também na sua busca.

Esse é exatamente o objetivo deste trabalho. Permitindo a visualização em um ambiente interativo e tridimensional dos dados gerados nas pesquisas sobre a dinâmica de *risers*, espera-se facilitar a assimilação e interpretação desses dados.

2.2.1. Breve Histórico

A computação gráfica começou, como dizem Foley et al. (1996), logo após o aparecimento dos computadores em si, com a exibição de dados em impressoras e monitores CRT[3]. Tão cedo quanto em 1950 o computador *Whirlwind* do MIT já utilizava monitores CRT para saída de dados. A origem do conceito de interação com gráficos de computador pode ser traçada até Ivan Sutherland e seu projeto *SketchPad* de 1963. Já em 1964 a General Motors utilizava o sistema DAC para projeto de automóveis auxiliado por computador.

No entanto, até o começo da década de 80, tratava-se de uma área pequena e especializada, devido ao alto custo do *hardware* de que necessitava e da existência de poucas (e caras) aplicações gráficas de simples utilização. A partir daí, com o barateamento e a popularização dos computadores pessoais com dispositivos gráficos de *raster* (ou varredura) como os Apple Macintosh e os IBM PC, popularizou-se o uso de gráficos matriciais ou *bitmap*[4] para a interação entre o computador e o usuário. Quando

---

[3] *Cathode Ray Tube*

[4] Um *bitmap* é uma matriz retangular de pontos (ou *pixels*) contendo valores de cores nesses pontos (inicialmente somente 0 e 1 para preto e branco) que representa uma imagem.

os gráficos matriciais se tornaram acessíveis, inúmeras aplicações gráficas baratas e de simples uso surgiram logo depois.

Da mesma forma que gráficos 2D eram incomuns até o começo dos anos 80 devido ao custo do hardware de que necessitavam, até o começo da década de 90 gráficos 3D continuavam incomuns pelo mesmo motivo. Foi nessa época que começaram a ser produzidos chips de custo acessível especializados em exibir e iluminar objetos 3D formados por polígonos. A partir daí a área de computação gráfica em três dimensões popularizou-se e idéias que eram consideradas excêntricas a menos de uma década, como a busca pelo realismo, passaram a ser objeto de pesquisas rotineiras e distribuídas por todo o mundo.

Um indicador desta popularização da visualização em três dimensões é a proliferação de APIs[5] especificamente desenvolvidas com essa finalidade nos últimos anos. Para citar somente algumas das mais populares: OpenGL, VRML, Java 3D, Open Scene Graph e Direct 3D. Historicamente é preciso citar também o sistema PHIGS e o GKS-3D, desenvolvidos em 1988. Dentre estas APIs, destaca-se o OpenGL por ter se tornado atualmente um padrão de fato do mercado, de acordo com Schroeder et al. (1998) e Angel (2000). É interessante mencionar também a API Mesa, criada em por Paul (2004), muito semelhante ao OpenGL (utiliza a mesma sintaxe, comandos e máquina de estados) e que, durante muito tempo após sua distribuição pela *Internet* em 1995, foi a única opção em diversas plataformas para o uso do OpenGL.

---

[5] *Application Programmer Interfaces*, ou seja, bibliotecas de funções que fazem a interface entre o programador de uma aplicação e um sistema de mais baixo nível, seja ele de *hardware* ou *software*.

Já a Visualização Científica, embora tenha sido utilizada desde antes mesmo da existência de computadores, só recebeu esta denominação e foi considerada uma área de estudo distinta da computação gráfica a partir de 1987, com o relatório da NSF entitulado *Visualization in Scientific Computing*. Mas desde o século XVIII, com a chegada dos gráficos estatísticos, representações gráficas de dados já eram criados. Os ganhos recentes em memória e capacidade de processamento dos computadores, no entanto, têm ao mesmo tempo popularizado e expandido as técnicas desta disciplina e também as tornado mais necessárias, devido ao grande volume de dados gerados, como já foi mencionado.

2.2.2. Técnicas de Visualização Científica

A visualização científica e sua importância foram definidas, mas com base nas definições dadas até agora, a distinção entre computação gráfica, processamento de imagens ou mesmo disciplinas matemáticas, como a geração de gráficos estatísticos não fica clara.

Schroeder et al. (1998) dizem que, de maneira geral, o campo da Visualização Científica se distingue pelas seguintes características:

- Os dados tratados pela visualização têm pelo menos três dimensões, e normalmente mais. Para dados de dimensionalidade menor existem muitos métodos bem conhecidos que são bastante adequados e se encaixam melhor em outras áreas como o processamento de imagens ou mesmo a matemática.

- Uma das características principais da visualização é a transformação dos dados. Isso significa que a informação é repetidamente criada ou modificada para incrementar o significado dos dados.

- A visualização é inerentemente interativa, incluindo o elemento humano diretamente no processo de criação, transformação e exibição dos dados.

Como pode-se perceber, a visualização científica é focada sobre os dados e suas transformações, e estes normalmente têm três ou mais dimensões.

Com base nessas características pode-se observar que a visualização de fenômenos ligados à exploração marítima de petróleo de fato pode ser classificada como visualização científica. Em primeiro lugar, trata-se de problemas eminentemente tridimensionais (por exemplo a distribuição de linhas de amarração ou a configuração de grupos de *risers*) variando ao longo do tempo (uma quarta dimensão) e com dados de diversas naturezas (escalares, vetoriais e tensoriais) associados a elementos como os *risers*, as embarcações e a própria água do mar. Silveira et al. (2000) dão um exemplo prático de outra aplicação que utiliza técnicas de visualização científica para tratar do problema de amarração. Outro exemplo interessante é a aplicação de técnicas de visualização científica para manutenção de reservatórios de petróleo, mencionado por Birken & Versteeg (2000). Esta aplicação será explorada com um pouco mais de detalhe durante a discussão sobre Realidade Virtual.

Uma das técnicas mais utilizadas na Visualização Científica é o **mapeamento de cores** (*color mapping*) para representar o valor de um escalar (como temperatura ou pressão, por exemplo) ao longo de uma superfície, uma linha ou um volume. Esse

mapeamento é feito associando cada valor possível para o escalar a uma cor através de uma tabela (*lookup table*) ou de uma função de transferência. A figura 6 mostra o uso do mapeamento de cores sobre um *riser*.

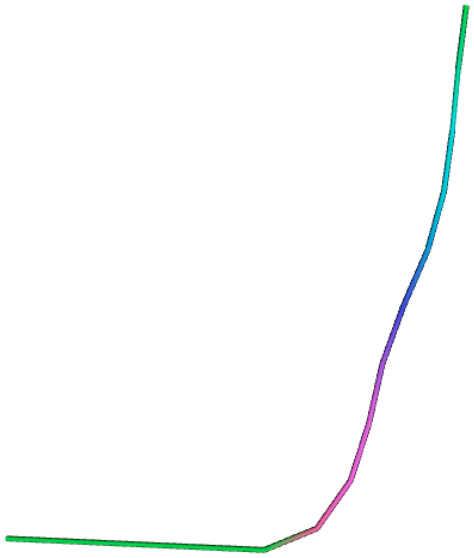

Figura 6 - Mapeamento de cores sobre um *riser* no RiserView

Outra técnica comunmente utilizada com esse fim é a de **curvas de nível ou iso-superfícies** (*countouring*) e um exemplo dessa técnica sendo usada no estudo de vórtices pode ser visto na figura 7.

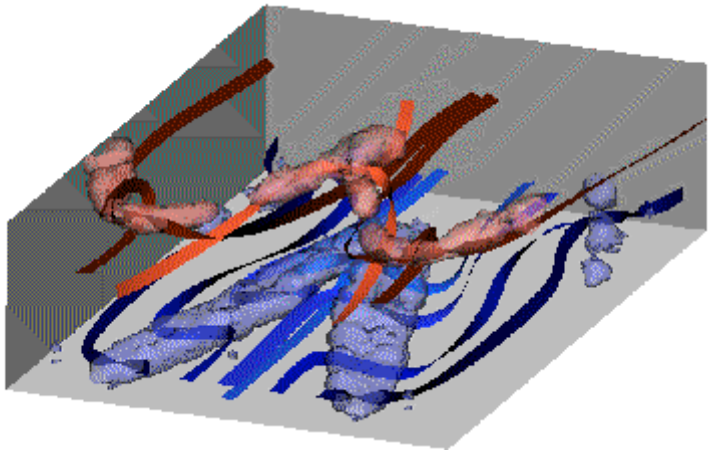

Figura 7 - Iso-superfícies na visualização de vórtices (*fonte: Günther et al., 2004*)

No ambiente desenvolvido neste trabalho, está prevista a visualização de diversos campos escalares associados a uma esteira de vórtices utilizando a técnica do mapeamento de cores. Outro uso para esta técnica é o mapeamento de escalares, como módulo da tensão total ou curvatura, ao longo dos *risers*.

Já para visualizar grandezas vetoriais (por exemplo, um campo de velocidades de corrente marítima) é comum utilizar **glifos**[6] (uma linha, uma seta, um triângulo, um cone...) com a origem no ponto associado ao vetor, a mesma orientação do vetor naquele ponto e escala proporcional ao seu módulo. Como vetores estão comunmente associados a movimento (no caso de vetores que representam velocidades ou deslocamentos), uma outra técnica bastante utilizada consiste na **deformação** (*warping*) de uma geometria de acordo com o campo vetorial. Uma extensão natural dessas técnicas relacionadas a movimento é a **animação** de uma geometria no tempo. Por fim, conectando as posições de pontos ao longo dos passos de tempo de uma animação pode-se obter **linhas de corrente** (*streamlines*), que é outra técnica bastante utilizada.

O **RiserView** faz extenso uso das técnicas de animação e deformação para representar o movimento dos risers ao longo do tempo e permite utilizar ainda glifos animados ao longo do tempo para representar as posições e deslocamentos de vórtices discretos formando uma esteira.

---

[6] Glifo (*Glyph*): uma representação 2D ou 3D tal como uma seta ou um cone orientados que pode ter sua orientação e escala modificadas, mas que ainda assim é usada repetidamente numa cena.

Existem também técnicas para visualizar grandezas tensoriais, mas essas técnicas não foram estudadas neste trabalho.

As figuras 8 e 9 exemplificam o uso de glifos e de linhas de corrente, respectivamente.

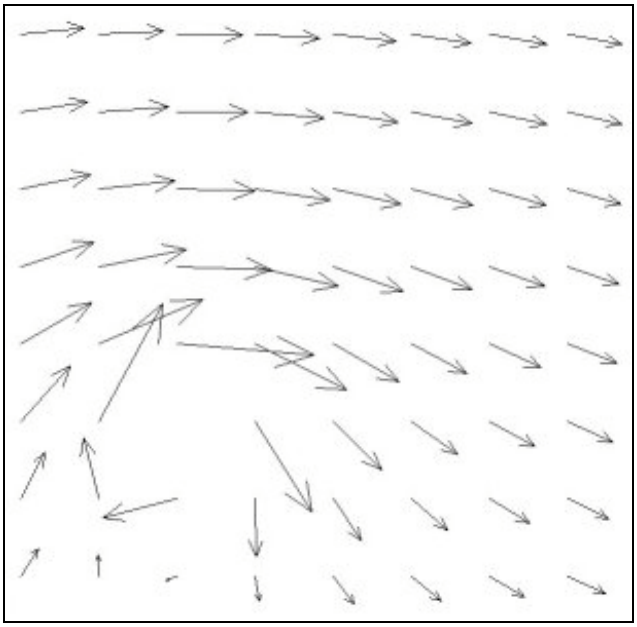

Figura 8 - Uso de glifos (*fonte: PV-Wave*)

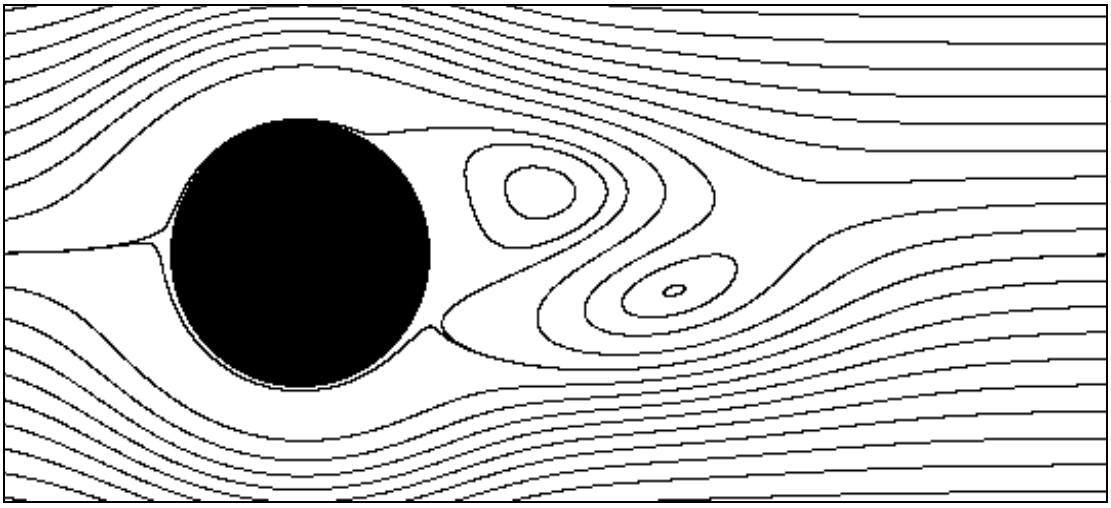

Figura 9 - Linhas de corrente (*fonte: György, 2004*)

2.2.3. Animação

Anteriormente a animação foi mencionada como uma técnica utilizada na visualização científica. No entanto, esta técnica é utilizada em diversas outras áreas, da simulação ao entretenimento, da educação a aplicações industriais.

De acordo com Foley et al. (1996), a animação tradicional, onde artistas desenham cada um dos quadros que compõem uma animação, é uma disciplina por si só, cuja origem se confunde com a do cinema. Atualmente existem aplicativos para auxiliar essa forma de animação (por exemplo interpolando posições entre os quadros principais desenhados pelo artista).

Thalmann & Thalmann (1990) definem animação como a técnica através da qual uma ilusão de movimento (ou qualquer outra mudança dinâmica, como crescimento) é criada exibindo uma seqüência de imagens (ou quadros) a um certo ritmo, onde cada imagem é uma alteração da anterior.

Tipicamente, o ritmo de troca das imagens é de 24 quadros/s ou 30 quadros/s, mas até ritmos tão baixos quanto 5 quadros por segundo ainda fornecem a ilusão de movimento, ainda que a transição entre os quadros torne-se perceptível (como "pulos") nesse caso.

Numa animação genérica, podem ocorrer diversas mudanças com os objetos observados (posição, orientação, tamanho, forma, cor, transparência), com o observador

(posição, ponto de referência, ângulo) e com as fontes de luz (posição, intensidade, cor), ou até com a animação em si (velocidade das imagens).

No **RiserView** é permitida animação de três formas: os elementos (objetos) da cena são modificados para refletir suas mudanças de estado ao longo do tempo, o usuário controla a posição do observador e também a velocidade da animação.

Em aplicações como neste ambiente, em que a animação deve ser interativa e conviver com uma interface gráfica com o usuário, surgem alguns conflitos que devem ser tratados. Ao mesmo tempo que a animação deve estar sendo atualizada constantemente, as ações do usuário também devem ser constantemente monitoradas para que o *software* possa dar as respostas adequadas a essas ações.

2.3. Realidade Virtual e Ambientes Virtuais

Não há dúvidas que o ambiente desenvolvido neste trabalho utiliza muitos elementos em comum com ambientes virtuais. Uma cena, ou mundo, virtual é construída em três dimensões e o aplicativo permite que o usuário navegue por essa cena. O capítulo 6 discute, inclusive, como o uso de uma forma de navegação típica de ambientes virtuais melhora a usabilidade desse aspecto do aplicativo. Há ainda o recurso de visualização estereoscópica da cena, a animação independente de seus elementos, a preocupação com a realização das tarefas em tempo real e com o realismo quando este não interfere com a visualização científica.

Devido a esses pontos em comum, julga-se necessário uma breve exposição sobre o tópico de Realidade Virtual (RV).

Mas apesar disso, devido ao formalismo científico que se espera de um trabalho como este, desde seu início persistiu uma dúvida quanto à sua natureza: o **RiserView** trata-se de um ambiente virtual, de acordo com a definição formal do termo? Para responder a essa pergunta, é preciso analisar as definições de RV e Ambientes Virtuais, embora muitas vezes autores classifiquem sistemas como de RV meramente com base na presença de alguns dos elementos citados acima.

Burdea & Coiffet (1994) discutem algumas das definições equivocadas (na sua opinião) de realidade virtual. Uma das definições com a qual discordam é a que tenta definir a realidade virtual através das ferramentas que utiliza, como HMDs[7] ou luvas, por exemplo. Burdea & Coiffet dão exemplos de outras formas de implementar ambientes virtuais sem fazer uso destes dispositivos. Ao fim desta análise, oferecem a sua definição de realidade virtual: "*Realidade Virtual é uma interface de usuário de alta complexidade que envolve simulação e interações através de múltiplos canais sensoriais.*" (Burdea & Coiffet, 1994, p. 4) Elaboram ainda mais dizendo que a partir desta definição fica claro que um ambiente virtual deve ser imersivo (por isso os múltiplos canais sensoriais) e interativo.

Kirner (1998) define realidade virtual como uma técnica de interface com o usuário onde ele pode realizar imersão, interação e navegação em um ambiente virtual tridimensional gerado por computador, utilizando canais multi-sensoriais.

A **interatividade** é, de fato, considerada por todos os autores como um "ingrediente" essencial da realidade virtual. Sem a capacidade de alterar o mundo virtual e ver as conseqüências de suas ações, o usuário dificilmente tem o sentimento de imersão.

Já a **imersão** é mais problemática, principalmente porque também não tem uma definição clara. Burdea & Coiffet definem imersão como "a sensação de fazer parte da ação [se desenrolando] na tela." (Burdea & Coiffet, 1994, p. 4) E acrescentam que, ao estimular mais de um sentido, essa sensação é aumentada. Já Vince (1995) não dá uma definição exata para imersão, mas afirma que um elemento crítico para se obter esta sensação é um sistema gráfico que dê ao usuário uma vista realista e em primeira pessoa do mundo virtual e que seja diretamente controlada pelo usuário. Vince, no entanto, não considera que a imersão seja um componente essencial para realidade virtual, como pode ser visto por sua classificação de sistemas de RV em imersivos, não-imersivos ou híbridos.

Para Vince, sistemas híbridos são sistemas que combinam imagens do mundo real com imagens virtuais, formando o que costuma se chamar **Realidade Aumentada**. Já a classificação como imersivo ou não-imersivo é feita (como condenam Burdea & Coiffet) com base nos dispositivos de saída utilizados. Quando se utilizam "dispositivos imersivos" (HMDs, óculos estereoscópicos, dispositivos que exibem imagens que ocupam uma grande parte do campo de visão etc.), trata-se de um sistema imersivo. Mas mesmo que se utilize meramente um monitor convencional o sistema ainda pode

---

[7] *Head Mounted Displays*

ser considerado um sistema de realidade virtual. É como se o mundo virtual existisse mas pudesse ser visto somente através de uma "janela" (a tela do monitor).

Pimentel & Teixeira (1995) concordam com a idéia que um ambiente virtual pode existir e ser visualizado sem nenhum dispositivo especial. Estes autores dão uma definição mais abrangente de imersão como uma variável contínua (ao invés de booleana, imersivo ou não) que depende da combinação de diversos fatores, como interatividade, velocidade de atualização, complexidade da imagem, som (Pimentel & Teixeira não citam outros sentidos, mas poderiam ser encaixados aqui conforme a definição de Burdea & Coiffet), resolução da imagem, estereoscopia, tamanho do campo de visão, sistema gráfico, dentre outros. Sendo assim, uma aplicação pode ser bastante imersiva (e classificada como RV) mesmo utilizando um monitor convencional, se os outros fatores compensarem este detalhe.

Continua, no entanto, a dúvida quanto a este ambiente para visualização de *risers*. Todos os autores dão grande importância à interatividade como elemento fundamental para um sistema de realidade virtual, o que sugere uma análise um pouco mais detalhada de como é a interatividade no ambiente em questão.

Como já foi dito, o usuário pode controlar seu ponto de vista, navegando livremente pela cena, assim como a velocidade da animação dos objetos do mundo. Pode ainda escolher quais objetos são exibidos ou não. Por isso, a aplicação é sem dúvida interativa.

No entanto, o usuário não tem nenhum controle sobre os objetos da cena em si. Ele não pode mudar um *riser* ou uma plataforma de lugar (o que não faria sentido, já

que o objetivo do ambiente é visualizar a dinâmica do sistema **sem** essas interferências) nem ver o efeito ao "furar" uma onda com seu "corpo virtual". O usuário nem mesmo colide com elementos da cena (isso poderia fazer com que alguns pontos de vista de interesse fossem difíceis ou impossíveis de alcançar dependendo da configuração da cena). Ele também nunca pode ver nenhuma parte de seu "corpo" (um braço, mão ou uma face refletida em uma superfície qualquer por exemplo). Ou seja, o usuário consegue interagir com a visualização **mas não tem nenhuma maneira de interagir diretamente com o mundo** que está sendo mostrado na cena. Isso, mais que qualquer outro fator, é o que não permite a classificação do **RiserView** como um ambiente virtual, ainda que possua diversas características em comum com um. De fato, o **RiserView** é um ambiente que combina elementos de RV com Visualização Científica. Esta conclusão parece concordar com a descrição que Hearn e Baker (1997) dão para realidade virtual: uma tecnologia que permite ao usuário adentrar uma cena e interagir com elementos daquele ambiente.

A discussão sobre Realidade Virtual prossegue com um breve relato de sua história. Depois disso será explorada também a estereoscopia. Por fim, será discutido o uso destas técnicas na industria de exploração marítima de petróleo.

2.3.1. Breve Histórico

De acordo com Burdea & Coiffet (1994), a origem da realidade virtual pode ser traçada, para surpresa dos que imaginam ser uma disciplina bastante moderna, a 1962.

Este foi o ano em que Morton Heilig (um cinematógrafo, não um engenheiro) registrou seu invento, o "Sensorama". Tratava-se de uma estação com vídeo 3D colorido (utilizando duas câmeras de 35 mm), movimento (através do assento que podia se mover e vibrar), som estéreo, aromas e até vento (através de pequenos ventiladores próximos da cabeça do usuário).

Antes mesmo do Sensorama, Heilig já havia patenteado algo similar a um HMD, mas foi Sutherland, em 1966 que criou o primeiro HMD usando dois CRTs, uma configuração que persiste até os dias atuais em diversas aplicações. Como os CRTs da época eram consideravelmente mais pesados que os atuais, no entanto, Sutherland teve que montar um braço mecânico para suportar parte do peso. Este braço servia também para determinar a posição da cabeça do usuário (*head-tracking*) para que as imagens exibidas pudessem ser modificadas conforme seus movimentos. Esta técnica também é usada até os dias de hoje, mas são utilizados meios que não exigem contato para determinar a posição (por exemplo magnetos, ultra-som, infravermelho ou laser).

Em 1973 Evans e Sutherland desenvolveram o "gerador de cenas", o precursor dos modernos aceleradores gráficos, para utilizar cenas geradas por computador em vez de imagens analógicas provenientes de câmeras nos seus HMDs. Esse "gerador de cenas" era capaz de renderizar (sintetizar, iluminar e exibir) cenas com 200 a 400 polígonos a aproximadamente 20 quadros/s.

Nas décadas de 70 e 80 as pesquisas na área foram estimuladas pelo interesse militar, principalmente na área de simuladores de vôo para treinamento de pilotos. Com finalidade semelhante a NASA, em 1981, cria o primeiro HMD com telas de cristal líquido (LCD).

Em 1988 Fisher e Wendel criaram o primeiro *hardware* capaz de manipular som 3D e em 1992 foi desenvolvida a primeira versão do "WorldToolkit", um pacote de *software* comercial para o desenvolvimento de mundos virtuais que indicava que a tecnologia já não era mais tão especializada e restrita.

2.3.2. Sensação de Profundidade e Estereoscopia

De acordo com Burdea & Coiffet (1994), a percepção de profundidade pela visão humana pode ocorrer com um olho só ou com a cooperação de ambos. Se somente um olho é usado, essa percepção é baseada em "pistas" (*cues*) inerentes à cena, como:

- Perspectiva;
- Oclusão dos objetos que estão atrás de outros;
- Nível de detalhe percebido (que é menor quanto maior a distância);
- Paralaxe para objetos se movendo em relação ao observador (objetos mais próximos parecem se mover mais e mais rápido);
- Sombras (de um objeto sobre o outro).

A maioria dessas pistas de profundidade já é levada em conta nos sistemas gráficos que renderizam cenas em três dimensões, embora ocasionalmente seja necessário um cuidado maior com a paralaxe, e o cálculo de sombras de um objeto sobre outro nem sempre seja factível em tempo real.

O mecanismo mais importante para a maioria das pessoas para percepção de profundidade, no entanto, ocorre quando ambos os olhos registram imagens ligeiramente diferentes para uma cena e o cérebro interpreta a diferença entre essas imagens como medida de profundidade. A isto é que chamamos estereoscopia.

A figura 10 ilustra como um objeto é visto diferentemente por cada olho. Quão diferente são essas vistas é função da distância entre o objeto e o observador (quanto menor a distância, maior a diferença). A linha vermelha indica o centro da cena para cada olho para servir como referência e não faz parte da cena renderizada para cada olho, mas foi acrescentada posteriormente. O resultado da interpretação pelo cérebro dessas duas imagens se cada uma estivesse sendo vista ao mesmo tempo somente pelo olho correto seria um objeto com aparência de profundidade, tridimensional, como se o navio "saltasse" do papel.

Figura 10 - Estereoscopia

Para reproduzir a estereoscopia normalmente é necessário algum *hardware* específico para esse fim. Gerar uma imagem diferente para cada olho é bastante simples, basta, por exemplo, mover o observador (ou a câmera) para a posição adequada e renderizar cada cena. Fazer com que cada olho do usuário enxergue a cena correta, e somente a cena correta, no entanto, faz necessário o uso de dispositivos como o HMD mencionado anteriormente, com uma tela para cada olho. Outra opção são óculos que, em sincronia com o sistema de exibição (seja um projetor ou um monitor, por exemplo) permitam que se veja ora através de uma de suas lentes, ora através de outra. Uma solução mais barata e comum que permite separar a imagem vista por cada olho é a geração de imagens com cores diferentes para cada olho, cores que são filtradas por óculos com lentes coloridas usados pelo usuário.

2.3.3. Uso destas Técnicas na Indústria do Petróleo

Como foi dito anteriormente, vários problemas nesta indústria são eminentemente tridimensionais e portanto podem ter sua visualização aprimorada utilizando estereoscopia. Atualmente as maiores aplicações de RV na indústria de petróleo estão nas áreas de exploração de reservatórios e projeto e construção de instalações para a exploração desses reservatórios, embora no futuro a possibilidade do uso de RV para o monitoramento da produção não pareça remota.

Na área de projeto de instalações, projetos como o Tanque de Provas Numérico (2004) ilustram como a realidade virtual pode facilitar a compreensão de fenômenos que

afetam estruturas como *risers* e plataformas e utilizar essa compreensão no projeto dessas estruturas. Outro exemplo é a aplicação para análise de amarração de plataformas citada por Silveira *et al*. (2000).

Birken & Versteeg (2000) citam uma aplicação interessante de métodos avançados de visualização científica que também são beneficiados por técnicas como estereoscopia, na área de exploração de reservatórios. Trata-se da interpretação de dados provenientes de testes sísmicos 4D (ou 3D com lapso de tempo) e testes de GPR 4D[8]. O princípio destes testes é simples. Comparando resultados de testes feitos em intervalos de tempo pequenos o suficiente de forma que não possa ocorrer deslocamentos significativos das parcelas sólidas do solo, as diferenças entre os testes se devem, necessariamente, a deslocamentos de fluidos e a partir daí pode-se determinar propriedades hidrogeológicas do solo. De acordo com a Petrobras (2001), a interpretação dessas informações por especialistas permite uma compreensão mais clara da geometria externa bem como da arquitetura interna dos reservatórios, permitindo não só planejar aonde, mas como explorá-los.

De acordo com Birken & Versteeg (2000) os testes sísmicos que eram pioneiros no começo da década de 90 hoje já são bem conhecidos e utilizados pela indústria de petróleo para monitorar mudanças nos reservatórios de hidrocarbonos e fazer manutenção desses reservatórios. O uso de RV na interpretação desses dados, conforme a Petrobras (1999), permite que o tempo para a interpretação desses dados para o mapeamento de reservatórios caia de meses para meras horas. O que Birken & Versteeg

---

[8] *Ground Penetrating Radar*. O 4D refere-se às três dimensões do espaço, mais o tempo.

propõem é usar a técnica semelhante com GPR para manutenção de reservatórios subterrâneos de água potável.

2.4. Orientação a Objetos, Metodologia de Projeto e UML.

Nesta seção são discutidos alguns aspectos importantes para a compreensão da metodologia através da qual é implementada a solução (utilizando as técnicas de realidade virtual e visualização científica discutidas acima) para o problema da visualização da dinâmica de *risers*. A metodologia em si é discutida no capítulo 3 e seu uso é relatado no capítulo 4. Como um dos objetivos deste trabalho é que o desenvolvimento do *software* siga uma metodologia formal, deve ser feita uma análise das vantagens e desvantagens dessa decisão para justificá-la.

A maior desvantagem que é sempre levantada contra o uso de uma metodologia formal no desenvolvimento de *software* é o tempo que é dedicado a atividades que não geram sequer uma linha de código. Em primeiro lugar há o tempo de aprendizagem do processo, depois o tempo de planejamento do projeto e até durante o desenvolvimento há o tempo utilizado para a documentação exigida pela metodologia.

Já algumas das principais vantagens, de acordo com Jacobson et al. (1999a), são as seguintes:

Em projetos envolvendo equipes de programadores (e projetistas, supervisores etc.), o uso de um processo bem definido permite que cada um saiba sempre exatamente o que está fazendo e como isto se encaixa com o projeto como um todo. Os

programadores também podem compreender melhor o que os outros membros da equipe estão fazendo, o que facilita, por exemplo, as tarefas de integração. Os cronogramas ficam bem definidos e cada um sabe quando pode contar com outros módulos do sistema (ou pelo menos sabe quando ocorrem atrasos).

O processo não só impõe uma fase de planejamento como também fornece uma estrutura para apoiar e guiar este planejamento. Um planejamento bem feito não só evita a necessidade de retrabalhos como facilita o reaproveitamento de código (identificando a priori elementos em comum que podem ser encapsulados para serem reutilizados) e uma arquitetura bem planejada tipicamente é mais facilmente expansível. Isso vale até mesmo para projetos envolvendo uma só pessoa, mas torna-se mais e mais crítico conforme o número de participantes aumenta (pois os retrabalhos e mudanças de arquitetura, se necessários, são muito mais complexos neste caso).

A metodologia também fornece uma estrutura bem definida para a geração da documentação do projeto. Se todos os envolvidos estão familiarizados com esta estrutura, encontrar a informação de que precisam na documentação torna-se simples e rápido. Mas ainda mais importante, a estrutura tenta garantir que informações importantes não sejam perdidas durante o processo. Se o sistema deve ser expansível para ser modificado no futuro, possivelmente por uma equipe diferente, uma documentação completa e organizada é essencial.

Jacobson et al. (1999) ainda citam diversas outras vantagens, como o aumento da repetibilidade do desenvolvimento (ou seja, o processo de desenvolvimento sempre segue os mesmos passos, mesmo para projetos bastante diferentes, e esta estrutura

permite que se estime melhor os tempos e recursos necessários em cada etapa, baseando-se em projetos anteriores), mas as citadas acima são as principais.

Com base nesta análise, foi considerado interessante aplicar uma metodologia formal mesmo num projeto de pequena escala como o desenvolvido neste trabalho, com somente um programador. Além disso, uma das características de uma boa metodologia é que seja adaptável a projetos de diferentes escalas.

A metodologia escolhia e que será descrita no capítulo 3, o Processo Unificado (UP), exige a utilização da orientação a objetos e da linguagem UML. Por isso, antes de uma discussão mais detalhada do UP, é interessante fazer uma breve exposição sobre esses dois temas.

2.4.1. Orientação a Objetos

Takahashi (1990) afirma que o paradigma de objetos é a grande evolução na área de desenvolvimento de *software* após as idéias de programação estruturada da década de 70. Flatt et al. (1998) contam que este paradigma surgiu devido à escala cada vez maior dos sistemas de *software* sendo desenvolvidos e da necessidade de se reutilizar o maior número de componentes possível. O paradigma de objetos, apesar de sua origem, não está limitado à programação. Luckas & Dörner (2000) exemplificam uma transposição deste paradigma para a área de animação 3D, onde seus *animation elements* se comportam como classes de objetos. Mas qual é a diferença entre sistemas orientados a objetos e sistemas convencionais?

Schroeder et al. (1998) afirmam que a maior diferença está na forma como ambos abstraem os dados. Num sistema procedural esta abstração está limitada a tipificação dos dados (*data typing*). Já nos sistemas orientados a objetos são criadas abstrações tanto para os dados quanto para as operações que podem ser aplicadas sobre eles.

Com base nessa afirmação, pode-se definir objeto como "*uma abstração que modela o estado e o comportamento de entidades em um sistema.*" (Schroeder et al., 1998, p. 21). O estado do objeto é definido pelos seus **atributos**, ou seja, os dados intrínsecos ao objeto. Já seu comportamento é definido por seus **métodos**, que são as operações que podem ser feitas sobre o objeto. Atributos e métodos de um objeto analisados em conjunto constituem as **propriedades** deste objeto.

Objetos com as mesmas propriedades (mas ainda assim objetos distintos, pois os valores destas propriedades podem variar) podem ser agrupados através da classificação. Uma **classe** de objetos, dessa forma, pode ser usada para definir as propriedades de todos os objetos daquela classe. Para utilizar um objeto de uma classe definida assim, cria-se uma **instância** desta classe. Takahashi (1990) explica que a instanciação é a "operação inversa" da classificação. Ao instanciar uma classe, cria-se um objeto daquela classe com uma identidade própria e valores iniciais para os seus atributos.

Um conceito importante da orientação a objetos é a **herança**. Através deste mecanismo, a criação de novas classes que diferem pouco de classes já existentes é significativamente simplificado. Uma classe que estende ou deriva uma classe já existente herda as propriedades desta e só é necessário acrescentar as propriedades que

distinguem a nova classe da antiga, ou modificar (*override*) métodos da classe antiga. Assim não é necessário duplicar código já existente.

Organizar objetos numa hierarquia de classes traz também o benefício de possibilitar a alteração de vários objetos simultaneamente ao se alterar uma classe de que todos derivam. Essa hierarquia pode ser criada através dos processos de especialização (indo das classes mais gerais para as mais específicas) ou de generalização (agrupando elementos de classes específicas em classes mais gerais). Podem existir classes, inclusive, cuja única finalidade é servirem como **superclasses** na hierarquia (ou seja, terem outras classes derivadas a partir delas), sem nunca serem instanciadas. De acordo com Schroeder et al. (1998), essas classes geralmente não podem ser instanciadas são denominadas **abstratas**.

Esta facilidade de acrescentar novas classes ao sistema para aumentar sua funcionalidade explica porque a orientação a objetos favorece a expansibilidade do sistema.

Outra característica importante da orientação a objetos é o **polimorfismo**. Sebesta (2000) explica que classes que estendem uma classe abstrata podem implementar seus métodos de forma diferente. No entanto, esses métodos continuam tendo o mesmo nome (o nome que têm na superclasse). Dessa forma, chamadas a métodos de mesmo nome e com os mesmos parâmetros podem ter comportamentos diferentes, dependendo do tipo do objeto que está sendo utilizado. Isso não só pode simplificar o código mas também permite que este seja padronizado e que sua compreensão seja facilitada ao se usar nomes de métodos que descrevem melhor a função que realizam. Por exemplo, no ambiente de visualização de *risers* classes tão

distintas quanto a que representa as ondas do mar e a que representa os *risers* têm em comum a função "Show", que controla a exibição destes elementos na tela.

Funções de uma mesma classe também podem ter o mesmo nome, desde que tenham parâmetros distintos (*overloading*). Por exemplo, duas funções que movem um objeto, uma que recebe as novas coordenadas como três números distintos e outra que as recebe como um vetor de três números, podem ambas chamar-se "Move".

2.4.2. A Unified Modeling Language (UML)

Booch et al. (1998) definem a UML como uma linguagem padrão para criação de "plantas" (ou modelos) de um sistema de *software*, da mesma forma que outras áreas da engenharia se utilizam de esquemas ou desenhos para melhor visualizar e comunicar idéias e estruturas. Além disso, a UML tem uma semântica bem definida, de forma que um diagrama confeccionado utilizando esta linguagem pode ser interpretado de forma não-ambígua, inclusive por ferramentas computacionais. Outra característica importante é que a UML é voltada especificamente para a especificação e documentação de sistemas de *software*.

A UML define diversos tipos diferentes de diagrama, cada um voltado para a descrição de um aspecto da arquitetura do sistema. Uma descrição de todos estes diagramas (e dos elementos que os compõem) não faz parte do escopo deste trabalho e é desnecessária, pois é feita de forma bastante completa por Booch et al. (1998). No entanto, dois diagramas merecem menção por serem os mais comuns na modelagem de

sistemas orientados a objetos. O primeiro é o diagrama de classes, que representa um conjunto de classes, suas propriedades e inter-relacionamentos. O segundo são diagramas de casos de uso, que representam as relações entre conjuntos de casos de uso e atores. Os principais elementos componentes destes diagramas são apresentados na figura 11.

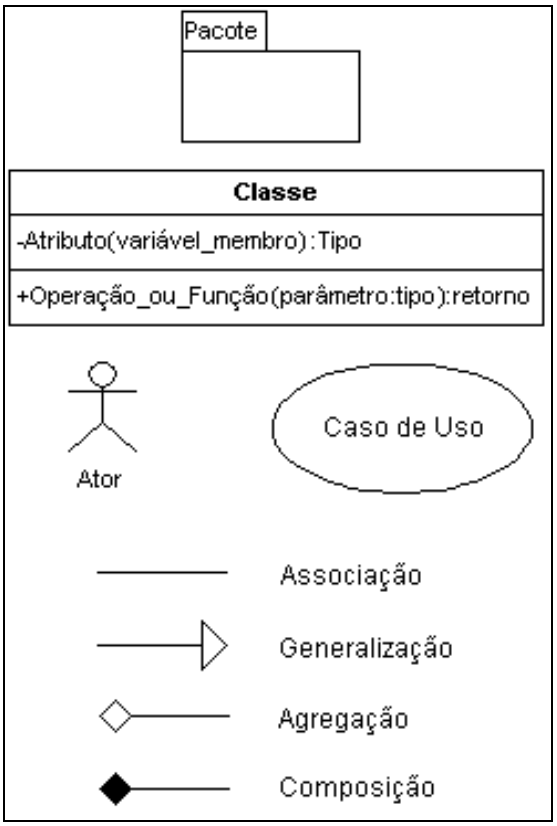

Figura 11 - Elementos da UML

Nessa figura, um pacote representa um conjunto de classes que implementam um subconjunto do sistema. As definições de classe e atributo foram dadas na seção anterior. Operação ou função são outras formas de se referir aos métodos de uma classe também explicados anteriormente. Casos de uso (e atores) são conceitos importantes para a metodologia escolhida mas serão definidos somente nos capítulos 3 e 4 onde essa

metodologia é discutida. Uma associação simplesmente explicita alguma relação entre duas classes ou objetos e normalmente é acompanhada de uma descrição. Uma generalização explicita a relação entre uma classe e sua classe base (casos de uso também podem ser generalizados). Uma agregação indica que uma classe é formada por objetos de outras classes (que estão agregados a ela) e uma composição é um caso particular de agregação que indica unicidade, ou seja, a classe tem somente um componente daquele tipo e o componente pertence somente àquela classe. O exemplo da figura 12 é um diagrama de classes auxilia na compreensão desses conceitos:

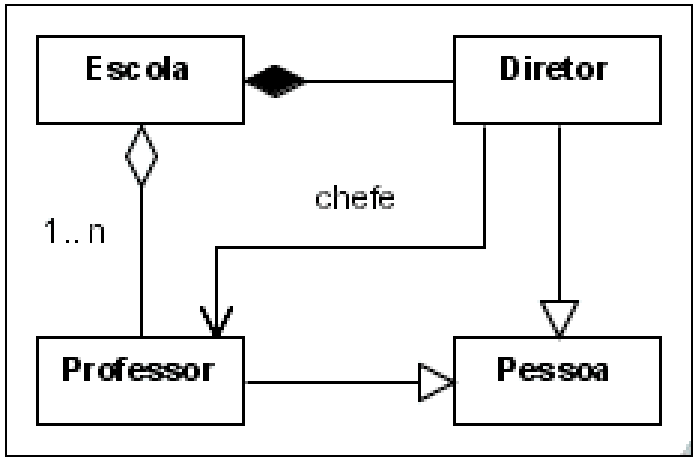

Figura 12 - Exemplo de diagrama de classes em UML

A agregação indica que uma escola conta com vários professores (e a multiplicidade está indicada pelo "1..n", que poderia ser também "1..*"), mas a composição indica que conta com somente um diretor. Tanto Diretor quanto Professor são derivados da classe mais geral, Pessoa, e existe uma associação indicando que o diretor é o chefe dos professores.

Além disso, ao listar atributos e operações das classes indica-se se são públicos, protegidos ou privados com os seguintes sinais antepostos ao nome do elemento:

+ indica um elemento público;

- indica um elemento privado;

indica um elemento protegido.

2.5. Padrões de Projeto e Arquitetura

Jacobson et al. (1998) afirma que para desenvolver um sistema com uma boa arquitetura pode-se lançar mão da experiência que outros desenvolvedores acumularam ao longo dos anos ao se utilizar de algumas "soluções genéricas" que foram evoluindo para tratar de classes de problemas recorrentes. Essas soluções são chamados de **padrões** (*patterns*). Padrões de Arquitetura (*Architecture Patterns*), como o padrão de Camadas (*Layers*) ou o MVC (*Model-View-Controller*) são soluções para aspectos da arquitetura do sistema como um todo. Já Padrões de Projeto (*Design Patterns*) são soluções para problemas de natureza mais específica e pontual que ocorrem comumente durante o projeto de *software*. Exemplos destes padrões são as Fábricas de Objetos (*Object Factories*) e Observadores (*Observers*).

A idéia dos padrões surgiu na área de arquitetura (não arquitetura de *software*, mas a que se encarrega de casas, prédios etc.) na década de 70 com Alexander, que afirma:

> *"Cada padrão descreve um problema que ocorre repetidamente em nosso ambiente e então descreve o cerne de uma solução para aquele problema, de forma que pode-se utilizar esta solução um milhão de vezes, sem nunca fazê-lo da mesma forma."* (Alexander et al., 1977, apud. Gamma et al., 1997)

O ambiente desenvolvido neste trabalho faz uso de diversos destes padrões, de forma explícita ou implícita. O padrão de Camadas, por exemplo, está implícito na arquitetura do *software* ao se utilizar uma biblioteca para desenvolvimento da GUI[9] e uma biblioteca gráfica (que por sua vez representa uma camada sobre o OpenGL). Este padrão, como explicam Jacobson et al. (1998), organiza os componentes de um sistema em camadas, de forma que cada componente só pode acessar outros componentes na camada diretamente abaixo da sua. O VTK (a biblioteca gráfica utilizada, que será discutida com mais detalhe adiante) também se utiliza de diversos Padrões de Projeto que precisam ser utilizados no desenvolvimento do sistema. Os padrões de Fábricas de Objetos, Observador e Estratégia (*Strategy*) existentes no VTK, por exemplo, são manipulados diretamente pelo sistema.

Explicitamente o padrão mais visível na arquitetura deste sistema é o MVC. Veit & Herrmann (2003) relatam que apesar deste paradigma ter se popularizado com o Smalltalk no começo da década de 80, até hoje o MVC é um padrão muito usado. Por exemplo, é levado em conta na maioria das bibliotecas modernas de *widgets* para interfaces gráficas. A função deste padrão é separar dados intrínsecos ao problema que o *software* se dispõe a solucionar de dados necessários para a visualização ou interface. O **Modelo** (*Model*) é uma classe que contém estes dados intrínsecos à aplicação, é uma abstração de uma entidade relacionada a um domínio (uma entidade física, ou gráfica, ou matemática...) que não tem nenhum conhecimento da interface. **Vistas** (*Views*) são as representações do Modelo na interface. Um modelo pode ter diversas vistas. Por exemplo, numa aplicação hipotética, o modelo de um *riser* pode ter uma vista que é

---

[9] *Graphic User Interface* ou Interface Gráfica com o Usuário

uma tabela com coordenadas e valores de escalares em alguns de seus pontos, outra vista composta por gráficos bidimensionais com as projeções do *riser* nos planos cartesianos e uma terceira vista com uma representação do *riser* em três dimensões, cada uma com uma interface distinta. Cada Vista é gerenciada por um **Controlador** (Controller) que é responsável pelas ações definidas na Vista com relação ao modelo. O controlador é que "traduz" mensagens de interface para a lógica da aplicação. Sempre que o modelo é modificado (seja por uma ação do usuário, seja por outro motivo qualquer) as Vistas e os Controladores devem ser modificados para se atualizarem, o que aliás, como aponta Gamma et al. (1997) constitui um clássico exemplo do padrão de observador. O encapsulamento de informações relativas à interface nas Vistas e seu gerenciamento pelos Controladores leva a um código mais limpo e elegante do modelo, que é a base do *software* e só precisa tratar do seu domínio.

No ambiente de visualização de *risers*, o Modelo contém os dados que representam todos os objetos que fazem parte da cena. Existe uma Vista principal, que mostra a cena, mas alguns elementos da interface (como caixas de diálogo de opções que podem alterar dados do Modelo) podem ser consideradas Vistas secundárias.

Sem entrar numa discussão detalhada sobre a arquitetura da aplicação neste ponto, pode-se citar o uso de alguns padrões de projeto no sistema em estudo:

- Fábricas de Objetos para instanciação de classes de *risers* com dados de tipos distintos (domínio da freqüência ou do tempo);

- Compostos (*Composites*) para lidar uniformemente com elementos de cena (vórtices, solo, ondas) e com listas de elementos de cena (listas de *risers*, listas de modelos);

- Estado (*State*) para guardar os parâmetros da câmera.

É desnecessário descrever estes padrões aqui, uma vez que Gamma et al. (1997) já catalogam estes (e diversos outros) padrões com riqueza de detalhes.

2.6. Portabilidade

O termo portabilidade é bastante abrangente. Pode aplicar-se ao código (que será explicada adiante), ao usuário (em questões relativas à interface), a equipamentos (por exemplo no caso de *drivers*), ao programador (ao usar APIs), a dados, dentre outros. Nesse trabalho, no entanto, a palavra é usada para designar especificamente portabilidade do código.

Levy et al. (1996) afirmam que para maximizar a utilização e o retorno de investimento num *software*, ele deveria ser capaz de ser executado sobre diversas plataformas com um *look-and-feel* (a aparência e o comportamento que se esperam de um *software* numa determinada plataforma) apropriado. A essa capacidade dá-se o nome de Portabilidade, que é, inclusive, um dos requisitos da aplicação desenvolvida neste trabalho, como citado anteriormente. Esta capacidade, no entanto, é difícil de se alcançar sem utilizar ferramentas adequadas, principalmente no caso de aplicações gráficas. Os principais fatores que dificultam a portabilidade são:

- diferenças de hardware (por exemplo, ordenação dos bytes, endereçamento de memória, resolução de cores);
- diferenças entre sistemas operacionais e sistemas de arquivos (por exemplo, capacidade de multiprocessamento, sensibilidade a maiúsculas e minúsculas);
- diferenças entre compiladores (por exemplo, tamanho do inteiro);
- diferenças entre as APIs (e sua complexidade) para cada sistema de interface gráfica.

De acordo com Figueiredo et al. (1993) uma das maneiras de reduzir estes problemas é utilizar padrões de fato, como ANSI C e OpenGL. Mas existem outras alternativas. Pode-se utilizar uma linguagem independente de plataforma, como Java. Ou ainda lançar mão de APIs que sejam elas mesmos portáveis, tornando o sistema desenvolvido sobre elas portável também. É esta última alternativa que é utilizada neste trabalho, e as bibliotecas que o permitem são discutidas no capítulo 3.

2.7. A *Pipeline* de Renderização, *Anti-Aliasing* e Texturas

Como se afirmou anteriormente, a *pipeline* de visualização é um componente essencial no VTK. É importante deixar claras as diferenças entre este componente e a *pipeline* de renderização. Além disso, agora que já se discutiu o objetivo que motiva este trabalho e as ferramentas que serão utilizadas, o momento é oportuno para discutir alguns detalhes mais específicos de implementação. Embora este trabalho não

implemente uma forma de renderização (o OpenGL se encarrega disto), é importante conhecer as operações envolvidas neste processo.

Em primeiro lugar é preciso definir o que é um ***pipeline***. Trata-se de um padrão de projeto em que uma atividade é subdividida em uma série de atividades menores, modulares, onde a saída de cada módulo é a entrada para o próximo. A figura seguinte ilustra esse padrão:

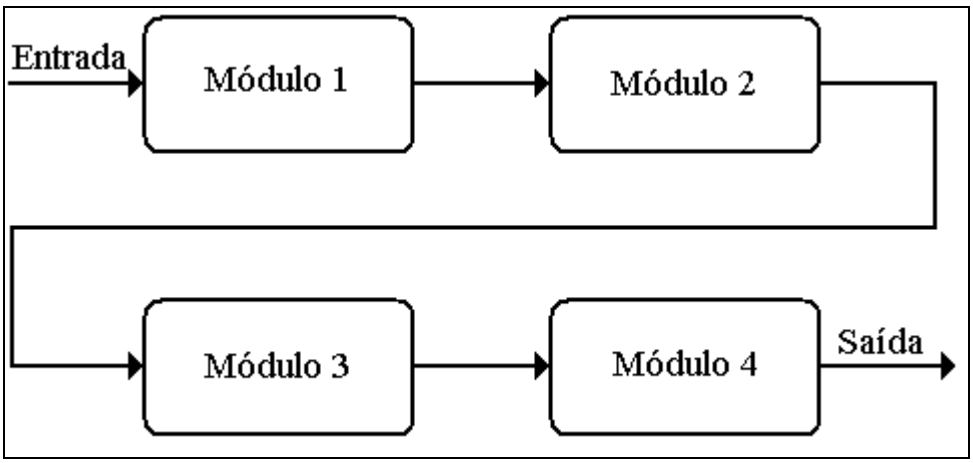

Figura 13 - Uma *pipeline* genérica com quatro módulos.

Já **renderização** é o processo que engloba as transformações e operações necessárias para exibir num dispositivo uma entidade geométrica. De acordo com Angel (2000), as duas principais formas de implementar este processo são a **baseada em objetos** e a **baseada na imagem**.

A **renderização baseada em objetos** itera sobre os objetos a serem exibidos para montar a imagem. Para cada objeto determina-se como ele deve ser exibido. Já a **baseada em imagens** itera sobre os elementos da imagem (os *pixels*). Para cada *pixel*

esta implementação determina quais objetos influem naquele *pixel* e determina sua cor (implementações de *ray tracing* podem ser exemplos desta estratégia).

Angel afirma que a maioria das implementações de renderização baseada em objetos é feita sob forma de uma *pipeline* com módulos distintos e interligados, sejam eles de *hardware* ou de *software*, para realizar cada uma das operações necessárias. Em particular este é o caso de APIs importantes como o OpenGL e o Direct X. A maioria das placas gráficas atuais também faz uso desta arquitetura.

2.7.1. A *Pipeline* de Renderização

As quatro operações que Angel (2000) afirma serem necessárias para a renderização são:

- A modelagem da geometria a ser renderizada (normalmente feita pelo usuário ou por uma aplicação distinta do renderizador e passada para este);
- Processamento Geométrico;
- Rasterização ou *Scan Conversion*;
- Exibição.

A *pipeline* tem módulos que realizam cada uma destas operações. Cada objeto da cena (por exemplo, polígonos) passa por estes módulos em seqüência.

No processamento geométricos são realizadas as seguintes operações:

- **Transformação das coordenadas** do objeto para o sistema do observador (ou câmera), **projeção** (normalmente ortogonal, mas pode ser isométrica, perspectiva etc.) do objeto e **normalização** de suas coordenadas (Angel demonstra como a normalização facilita a operação de *clipping*). As transformações de coordenadas e projeções são feitas através de multiplicações de matrizes 4x4 que normalmente são feitas por *hardware* nas placas gráficas.

- ***Culling*** e ***Clipping***: Trata-se da remoção de objetos (ou partes de objetos) que estão fora do campo de visão (normalmente uma janela retangular). *Culling* é a eliminação de objetos por inteiro e *Clipping* é uma operação mais complexa que "recorta" os objetos deixando somente a parte contida no campo de visão. Embora esta operação normalmente seja realizada aqui, alguns sistemas optam por executá-la após a rasterização, sobre a imagem bidimensional resultante.

- **Iluminação** dos objetos (*Lighting ou Shading*), ou seja, a determinação da cor em cada ponto do objeto com base nas fontes de luz em suas características (sua cor em cada ponto e valores de normal, sejam estes calculados a partir da geometria ou fornecidos com o modelo). Dependendo do algoritmo utilizado, pode ser mais conveniente fazer a iluminação neste ponto (ou até mesmo antes do *Clipping*) ou mais tarde, durante a rasterização. Foley et al. (1996) ilustram a diferença na ordem das operações em uma *pipeline* que utilize o algoritmo de Gouraud e uma que utilize Phong, por exemplo. No OpenGL, a iluminação é feita durante a rasterização.

- **Remoção de superfícies** oclusas por outros objetos. Assim como ocorre com a iluminação, o local onde esta operação deve ser feita depende do algoritmo

utilizado. Algoritmos que se baseiam numa ordenação (de trás para frente ou de frente para trás) das superfícies para determinar quais são visíveis e quais estão oclusas devem fazer esta ordenação antes da rasterização. O OpenGL utiliza outro algoritmo, o ***z-buffer***. De acordo com Angel este é o algoritmo mais amplamente utilizado para esta tarefa, devido à facilidade de sua implementação, seja por *software* ou por *hardware*, e ao fato de se adequar muito bem à arquitetura de *pipelines*, processando cada objeto (e cada vértice) na ordem que os recebe na *pipeline* sem necessidade de ordenação. O *z-buffer* é executado juntamente com a rasterização.

Após o processamento geométrico, vem a rasterização. Até este ponto, os objetos ainda são tratados como entidades geométricas. A rasterização é que discretiza estas entidades em conjuntos de *pixels*. Normalmente se refere à saída da rasterização como o ***frame-buffer***, que nada mais é que uma matriz retangular onde cada elemento representa um *pixel* da janela onde os objetos são renderizados e armazena o valor de sua cor. Como já foi dito, no OpenGL é durante a rasterização que é feita a remoção das superfícies oclusas e o *shading* das superfícies, utilizando o algoritmo de z-buffer em conjunto com a rasterização.

O *z-buffer* é uma matriz semelhante ao *frame-buffer*, com as mesmas dimensões, mas que, ao invés da cor, armazena qual a profundidade na cena (em relação ao observador) do ponto que determina a cor daquele *pixel*. Comparando a profundidade do ponto de um objeto que esteja passando pelo algoritmo com a profundidade armazenada no z-buffer, é simples determinar se este ponto está mais próximo que os que o precederam. Se estiver, o *frame-buffer* e o *z-buffer* são atualizados. Rogers (2001)

demonstra como o cálculo dos valores de profundidade armazenados no *z-buffer* utilizam a maior parte de sua precisão para os objetos próximos do observador. Alguns sistemas utilizam o *w-buffer* que procura resolver este problema. O *w-buffer* nada mais é que uma linearização dos valores de profundidade no *z-buffer*. Essa linearização pode trazer problemas de precisão em alguns casos, mas não sacrifica tanta precisão somente para os objetos mais próximos do observador.

Por fim, a operação de exibição consiste em passar o *frame-buffer* para o dispositivo que irá exibir esta imagem bidimensional resultante da renderização.

2.7.2. Anti-Aliasing

Como foi visto acima, durante a rasterização entidades geométricas, como polígonos ou linhas, são discretizados em *pixels*. Analisando esta operação sob a ótica da área de processamento de sinais como fazem Foley et al. (1996), as entidades geométricas podem ser vistas como sinais contínuos no domínio do espaço e sua discretização em *pixels* como um processo de amostragem. Ao exibir a imagem resultante na tela, o que se está fazendo é reconstruindo o sinal com base nestas amostras. Sabe-se que este processo de amostragem de um sinal contínuo e posterior reconstrução pode resultar em perdas e neste caso em particular as perdas são bem visíveis como o fenômeno conhecido como *aliasing*.

O *aliasing* é o responsável pelas linhas "quebradas", lembrando degraus, observadas com freqüência em gráficos de computador, como por exemplo na figura a

seguir. Além de reduzir o realismo da imagem, o *aliasing* ainda pode causar desconforto visual para o observador.

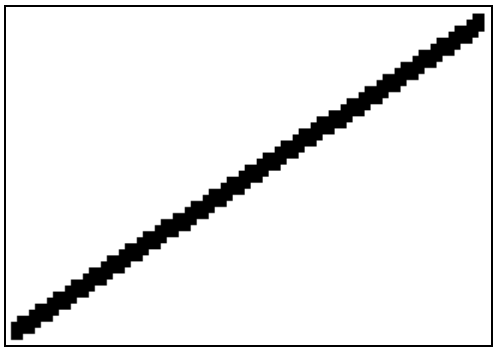

Figura 14 - *Aliasing*

Existem diversas técnicas e algoritmos para amenizar este problema, conhecidas como técnicas de *Anti-Aliasing*. Schroeder et al. (1998) citam as seguintes:

Durante a rasterização, a cor de um pixel vizinho aos identificados como pertencentes a uma linha, por exemplo, é influenciada pela fração daquele pixel que seria ocupada pela linha se esta fosse contínua. Esta técnica é conhecida como amostragem ou média de áreas (*area sampling* ou *area averaging*).

Uma solução similar envolve dividir cada *pixel* em *subpixels* (ocupando bem mais memória) e fazer a rasterização normalmente, determinando a cor de cada *pixel* pela média das cores de seus *subpixels*. Isto equivale a renderizar uma imagem com resolução maior e depois fazer uma operação de redução de escala.

A última técnica consiste em gerar diversas imagens com *aliasing* e combiná-las numa imagem *anti-aliased*. Estas imagens são obtidas movendo o observador e o ponto focal menos de um pixel em direções perpendiculares à direção de projeção.

Glassner (1990) e Foley et al. (1996) citam ainda como possível solução a convolução da imagem com filtros apropriados, como o gaussiano ou o cúbico, filtrando os sinais de alta freqüência que normalmente resultam de *aliasing*.

Outra forma de *aliasing* que ocorre durante animações (e nunca em imagens estáticas) é o *aliasing temporal*. Dachille & Kaufmann (2000) descrevem este fenômeno, que ocorre por um motivo análogo ao do *aliasing* espacial. Durante uma animação o movimento contínuo de uma entidade é amostrado em quadros discretos da animação. É por causa do *aliasing* temporal que rodas de carroça ou hélices de aviões em filmes antigos parecem rodar ao contrário quando atingem certas velocidades. Outra maneira simples de perceber este fenômeno é movendo um dedo rapidamente na frente de um monitor de computador. Neste caso a taxa de atualização do monitor só é suficiente para iluminar o dedo em algumas posições e o resultado são as imagens residuais resultantes.

Angel (2000) descreve outro motivo que pode causar *aliasing* temporal. Em sistemas que utilizam *ray casting*[10], um objeto pequeno ou distante da câmera pode se mover com uma velocidade que faça com que por vezes ele não seja atingido por nenhum dos raios. Isso faz com que o objeto fique aparecendo e desaparecendo da cena de forma bastante incômoda.

Assim como para o espacial, existem diversas soluções para o problema de *aliasing* temporal. Para o problema descrito por Angel, uma solução trivial é lançar

---

[10] Processo de renderização que ilumina os objetos de uma cena lançando "raios de luz" virtuais e calculando seu efeito sobre os objetos. Tipicamente lança-se um raio de luz por pixel.

mais raios de luz, ao invés de um para cada *pixel*. Dachille & Kaufmann citam o uso de alguns filtros. Wloka et al. (1995) descrevem uma técnica interessante de renderizar cada quadro utilizando informações do quadro anterior para acelerar o processo que tem como conseqüência uma suavização do *aliasing* temporal.

De acordo com Angel (2000), o que todas as técnicas de *anti-aliasing*, espacial ou temporal, têm em comum são um alto custo computacional e o fato de que, na prática, só são utilizadas quando uma imagem de alta qualidade é necessária e não em tempo real. Hoje em dia, com computadores e placas gráficas mais poderosas, isto está deixando de ser verdade. Algumas placas já implementam algoritmos de *anti-aliasing* espacial por *hardware*.

No aplicativo desenvolvido neste trabalho, existe a opção de ativar ou não *anti-aliasing* espacial por *software*, utilizando a terceira técnica descrita por Schroeder. Devido às características de escala e velocidade dos objetos que estão sendo visualizados, o *aliasing* temporal é imperceptível e portanto não houve necessidade de implementar mecanismos para eliminá-lo.

2.7.3. Texturas

De acordo com Angel (2000), texturas são ferramentas bastante úteis para fornecer um nível de detalhe (e realismo) maior às superfícies sem a necessidade de aumentar a complexidade dos modelos geométricos utilizados. O uso desta ferramenta está previsto neste trabalho. Texturas são padrões, sejam extraídos de uma imagem

digital, seja gerados por uma função matemática, que podem ser aplicados (ou mapeados) sobre a geometria. A cada ponto da geometria são associadas **coordenadas de textura** que associam aquele ponto a um dos pontos do padrão que constitui a textura. Estas coordenadas podem ser fornecidas junto com o modelo ou determinadas computacionalmente a partir da geometria e de como se deseja realizar o mapeamento. Durante a renderização, a cor da textura associada a um ponto da geometria contribui na determinação de cor deste ponto.

2.8. Detecção de Colisão

Dado um conjunto de objetos onde pelo menos um destes objetos tem movimento, podendo conter inclusive elementos estáticos de cenário, o objetivo da detecção de colisão é determinar todos os contatos geométricos que ocorram entre estes objetos. Alguns autores, como Cerny (2000) definem detecção de colisão como somente a determinação de contato entre os objetos que se movem e o ambiente, mas a maioria inclui colisões entre os objetos móveis como importante parte desta definição.

De acordo com Cohen et al. (1995), a detecção de colisão é um problema fundamental nas áreas de animação por computador, simulação física, ambientes virtuais e robótica. Neste trabalho, um dos objetivos é determinar e destacar as colisões que ocorrerem entre os *risers*.

O método trivial para detecção de colisão entre geometrias formadas por polígonos, verificar se cada face de um objeto tem interferência com todos os outros

polígonos da cena que não pertencerem ao mesmo objeto, é de complexidade $O(n^2)$ em relação ao número de polígonos e se mostra proibitivo.

Chung & Wang (1996) fazem uma análise abrangente de diversos algoritmos de detecção de colisão. A maioria discretiza a geometria dos objetos que podem colidir utilizando um volume limitante (*bounding volume*), ou uma hierarquia destes volumes, que circunscreve o objeto. Este volume normalmente é um paralelepípedo alinhado com o sistema de coordenadas ou uma esfera (*bounding box* e *bounding sphere*, respectivamente). O uso destes volumes simplifica a detecção de colisão, pois inicialmente só é preciso checar as colisões entre os volumes, que além de serem em bem menor número que os polígonos ainda tem formas simples que facilitam a detecção. Alguns algoritmos inclusive separam o espaço em setores, utilizando *octrees*[11], e classificam os objetos de acordo com o setor aonde estão para reduzir o número de checagens, antes mesmo de checar colisões entre volumes limitantes.

No entanto, a menos que se deseje trocar precisão por velocidade, quando dois corpos estão no mesmo setor e existe colisão entre seus volumes limitantes é preciso utilizar algum método de determinação exata de colisão. Chung e Wang mencionam diversas maneiras de acelerar esta determinação exata.

A maioria destas estratégias visa determinar quais são as faces dos objetos envolvidos próximas do ponto de colisão para que não seja necessário fazer a checagem de todas as faces de um objeto contra todas as faces do outro. Isto pode ser feito

---

[11] Estruturas de dados que separam o espaço em octantes cada vez menores, cada octante representado por uma "folha" na estrutura de árvore.

armazenando quais são os pares de faces mais próximos para cada par de objetos, subdividindo os volumes limitantes para determinar a região de colisão ou utilizando estruturas de dados adequadas para acessar os polígonos em uma determinada posição rapidamente.

Outro algoritmo para determinação exata de colisão entre dois objetos é encontrar um plano que separe estes objetos. Se for possível encontrar este plano, eles não colidem e a afirmação inversa também é verdadeira.

Dois conceitos importantes explorados para acelerar diversos destes algoritmos são os de coerência temporal e espacial. A coerência temporal é a suposição de que o estado da aplicação não sofre mudanças drásticas de um passo de tempo para o próximo. Quer dizer, os passos de tempo são pequenos o suficiente para que as mudanças sofridas pelos objetos sejam pequenas. Isto se traduz na coerência espacial, uma vez que a geometria destes objetos, definida pelas posições de seus vértices, também não passa por grandes alterações. Algoritmos que calculam dados como quais são os pontos mais próximos entre pares de objetos, ou que planos separam estes pares, podem armazenar os dados calculados em cada passo para servirem como iteração inicial no cálculo dos dados do passo seguinte, acelerando o processo.

O problema que este trabalho se propõe a solucionar, no entanto, possui duas características em particular que simplificam o problema consideravelmente.

A primeira é que, apesar da representação do *riser* como um conjunto de polígonos seja conhecida, uma representação mais simples (e inclusive com mais significado físico) também pode ser utilizada. Trata-se de representar os *risers* como

conjuntos de segmentos cilíndricos de raio e posição conhecidos. Embora, de acordo com Eberly (2000b), a detecção de colisões entre troncos de cilindros seja um processo de alto custo computacional, a colisão entre **cápsulas**[12] se reduz a checar se a distância entre os segmentos de reta que as originam é menor que a soma de seus raios. Para encontrar essa distância, Eberly (2000a) afirma que basta minimizar a função distância entre dois pontos pertencentes aos segmentos de reta. Embora esse seja um problema de minimização com fronteiras, o tratamento das fronteiras pode ser feito com um algoritmo relativamente simples, pois Eberly (2000a) demonstra que caso os pontos de mínimo da função não pertençam aos segmentos de reta, o mínimo necessariamente ocorre entre uma de suas extremidades e outro ponto qualquer. O algoritmo de baixa complexidade e custo para detecção de colisões entre cápsulas descrito por Eberly (2000a) pode ser usado no **RiserView** uma vez que as "tampas" semi-esféricas das cápsulas só devem colidir com segmentos do mesmo *riser*, em configurações normais dos mesmos, e essas colisões nunca devem ser checadas.

A segunda característica que simplifica o problema, para a qual não foi encontrado paralelo na literatura, é que o movimento de cada *riser* isoladamente é conhecido a priori. Trata-se de uma ocorrência bastante comum, pois se repete quando se está exibindo resultado de simulações pré-calculadas, mas ainda assim não foram encontradas soluções para detecção de colisão para esse caso. Como os *risers* não necessariamente vibram com a mesma freqüência, isso não significa que as interações entre eles sejam conhecidas, mas traz algumas simplificações ao problema (semelhantes

---

[12] Cápsula: O conjunto de pontos eqüidistantes de um segmento de reta. Pode ser visualizada como um cilindro limitado por semi-esferas, e não por planos.

às trazidas pela coerência temporal e espacial) que serão exploradas na solução do problema.

3. TECNOLOGIA

Este capítulo continua a descrição da revisão bibliográfica iniciada no capítulo 2. No entanto, os tópicos aqui descritos têm uma característica importante que os distingue dos discutidos anteriormente: tratam de questões mais intimamente ligadas a soluções tecnológicas atuais. São, portanto, discussões menos perenes que as anteriores.

O próprio processo unificado, a metodologia escolhida para o desenvolvimento do ambiente de visualização de *risers*, é o resultado de uma evolução contínua, como será visto adiante, e ainda sofre constantes revisões (o processo revisado, juntamente com ferramentas de *software* para auxiliar em sua aplicação, é conhecido como *Rational Unified Process* ou RUP). O processo descrito e utilizado nesse trabalho, no entanto, corresponde àquele descrito por Jacobson et al. (1999). A primeira seção deste capítulo fornece os conceitos necessários sobre esse processo para que sua customização e aplicação, no capítulo 4, possa ser compreendida, enquanto a UML já foi abrangida no capítulo 2.

O capítulo continua com a descrição das APIs utilizadas no trabalho e das alternativas estudadas. Por fim são abordados *shaders*, uma forma de programação para os processadores gráficos.

3.1. Processo Unificado

Antes de uma discussão mais aprofundada do processo em si, deve-se analisar os motivos para a sua escolha. Uma breve definição e revisão do histórico do processo vem então. São abordados com mais detalhe, por fim, alguns dos termos e conceitos que serão utilizados no capítulo 4.

3.1.1. Vantagens e Desvantagens

No capítulo 2, foram citadas as principais vantagens e desvantagens normalmente associadas à utilização de uma metodologia formal em um caso geral. Aqui será analisada a parcela destas vantagens e desvantagens que se aplica diretamente a este projeto - um projeto de porte relativamente pequeno, com somente um programador.

Uma das vantagens é o planejamento imposto pela metodologia, bem como a estrutura para apoiar e guiar este planejamento, reduzindo a necessidade de retrabalhos, facilitando o reaproveitamento de código e simplificando o esforço para se criar uma arquitetura expansível. Isso vale até mesmo para projetos envolvendo uma só pessoa.

A estrutura bem definida para a geração da documentação do projeto é outra vantagem que se aplica a esse projeto. Como uma de suas preocupações é que ele seja expansível, permitindo a continuação da pesquisa e o crescimento e manutenção do

projeto no futuro, provavelmente por uma equipe diferente, uma documentação abrangente e organizada é essencial.

Não se pode esquecer que este projeto é, acima de tudo, um trabalho de pesquisa científica. Como tal, uma descrição precisa da metodologia usada e dos passos seguidos durante seu desenvolvimento é essencial, conferindo não só repetibilidade ao processo, como também um maior valor como fonte de conhecimento para futuros pesquisadores. O uso de uma metodologia preexistente e bem conhecida permite que o pesquisador se concentre em seu objeto de estudo e não tanto no processo de desenvolvimento e permite ainda que outros pesquisadores, já familiarizados com a metodologia, possam compreender sua aplicação com maior facilidade.

Quanto à maior desvantagem relativa ao uso deste tipo de processo, o tempo dispendido em sua aprendizagem e aplicação, pode-se afirmar que uma das características de uma boa metodologia é que seja adaptável a projetos de diferentes escalas. Jacobson et al. (1998) afirmam que o UP, deve ser customizado para cada projeto, dependendo de fatores como escala do projeto, familiaridade da equipe com as tecnologias utilizadas, dentre outras. Smith (2002) detalha mais essa customização ao comparar o Unified Process a outra metodologia, o *Extreme Programming* (XP). Smith cita, por exemplo, que para projetos de pequeno porte e baixo risco, as fases de concepção e elaboração (que serão estudadas em mais detalhe adiante) podem ser muito mais curtas, partindo mais rapidamente para a fase de construção. Esta customização, considerada essencial no UP, pode reduzir em muito o tempo utilizado com a aplicação da metodologia.

Esta adaptabilidade do Processo Unificado, confirmada por Smith (2002), foi o principal fator que levou à sua escolha. O uso da metodologia Extreme Programming (XP), que tem muitos elementos similares ao UP, também foi levada em conta, principalmente por ser recomendada para projetos de menor porte. No entanto, conforme Beck (2000) apud Smith (2002), o XP é um processo bem menos flexível, estando limitado a equipes com no máximo 10 a 20 programadores. Outros fatores que influenciaram na escolha do UP foram o fato de ser uma metodologia voltada para a orientação a objetos e que utiliza a Unified Modeling Language (UML) como ferramenta, uma ferramenta com a qual o autor já tinha alguma familiaridade.

Com isto em mente, pode-se passar à uma discussão mais aprofundada do processo em si.

3.1.2. Definição e Histórico

De acordo com Jacobson et al. (1998), o processo unificado é uma metodologia que combina contribuições de diversas fontes, tanto de desenvolvedores quanto de usuários dos processos envolvidos, começando com o processo da Ericsson de 1967, que descreve a arquitetura de um sistema como diagramas de blocos que representavam os componentes do sistema e interfaces. Em 1976 o CCITT, um órgão internacional de padronização na indústria de telecomunicações institui a SDL (Specification and Description Language) uma precursora da UML. Em 1987 Jacobson cria o processo "Objectory" que formaliza a utilização dos **casos de uso** (conceito que será explicado

em seguida). Em 1995 a Rational Software Corp. compra o Objectory e a ele acrescenta algumas de suas marcas, como a ênfase na arquitetura e o desenvolvimento iterativo, lançando em 1997 o Rational Objectory Process (ROP). Neste período a Rational adquiriu diversas outras empresas cujas práticas foram somadas ao ROP causando sua evolução no que se chamou Processo Unificado. Novas evoluções da metodologia são englobadas no *Rational Unified Process* e recentemente a Rational é adquirida pela IBM.

Jacobson et al. começam a definir o UP dizendo que não se trata de um único processo, mas de uma *Framework* que pode (e deve) ser especializada para diversos tipos de projetos diferentes. No entanto, os três termos que definem o UP na opinião de seus criadores são:

- guiado por casos de uso;
- centrado na arquitetura;
- iterativo e incremental

Casos de uso são maneiras formais de se descrever os requisitos funcionais de um sistema. Nas palavras de Jacobson et al.:

"*Casos de uso são descrições de conjuntos de seqüências de ações, incluindo variantes, que um sistema realiza e que resultam num resultado observável e com valor para um dado ator.*" (Jacobson et al., 1998, p. 432)

Nesta definição ator significa um usuário do caso de uso (e não necessariamente um usuário do *software*, um ator pode ser, por exemplo, um subsistema). Ou seja, um

caso de uso nada mais é que uma descrição de uma conjunto de ações que, juntas, realizam uma função de valor observável para um usuário (o ator). No capítulo seguinte serão levantados os casos de uso deste projeto em específico e o conceito deve tornar-se mais claro.

A arquitetura de um sistema é a organização deste sistema, quais (e como) são seus componentes e as interfaces entre estes componentes, como eles se combinam para formar sistemas maiores, como o sistema será usado e todas estas decisões que definem a forma do *software* sob pontos de vista distintos sem se preocupar com detalhes de implementação.

A última expressão, "iterativo e incremental" significa que o UP se desenvolve em iterações bem definidas, cada uma adicionando mais funcionalidade ao sistema. Cada uma destas iterações se encaixa em uma das fases do projeto, como será visto adiante.

Outras características do UP são o fato do sistema sendo desenvolvido ser descrito como componentes que se comunicam por interfaces e o uso da UML para descrever o sistema de *software* graficamente, como as plantas e desenhos usados nas outras áreas da engenharia.

Com uma imagem do UP se formando mais claramente, pode-se discutir alguns termos e conceitos mais específicos que serão necessários no capítulo 4.

3.1.3. Fases, Iterações, Etapas e Artefatos no Processo Unificado

No Processo Unificado, o desenvolvimento do projeto está dividido em 4 **fases**. Cada uma destas fases tem um foco diferente, bem como objetivos diferentes.

Cada fase, por sua vez, pode ser composta por uma ou mais **iterações**. No processo unificado, o conceito de iteração é muito importante, pois define a forma como esta metodologia é implementada incrementalmente. Uma iteração é um "mini-projeto" cujo resultado final necessariamente traz um incremento na funcionalidade do que está sendo desenvolvido. As iterações são normalmente feitas de forma seqüencial, mas tarefas relativamente independentes podem ser desenvolvidas em paralelo, desde que as decisões de arquitetura quanto à interface entre seus produtos já tenham sido tomadas e que estas iterações em paralelo sejam seguidas em algum momento por um iteração para a integração destes produtos. O conceito de iteração dentro do Processo Unificado deve tornar-se mais claro ao se descrever as iterações em que é dividida a etapa de Construção neste trabalho.

Cada iteração divide-se em **etapas**, seguidas sempre na mesma ordem. Cada etapa descreve um conjunto de atividades do "mini-projeto" que é a iteração bem como **artefatos** que devem ser gerados.

As fases que se repetem no UP, sempre nesta ordem, são: **Concepção**, **Elaboração**, **Construção** e **Transição**. Cada uma tem foco e objetivos distintos, indo do mais geral para o mais específico. Ao fim de cada uma delas deve ser atingido um

milestone, um artefato ou conjunto de artefatos (distintos para cada fase e cada projeto) que demonstram que os objetivos daquela fase foram cumpridos.

A fase de **Concepção** tem como principal preocupação determinar a viabilidade do projeto, esboçando somente partes importantes do sistema como casos de uso e tentativas de arquitetura para determinar se o projeto é economicamente factível. Ao fim desta fase, o escopo do projeto deve estar definido e deve-se ter elaborado um estudo técnico e econômico quanto à viabilidade do projeto, estimando os recursos e prazos necessários e outros fatores desta natureza.

A fase de **Elaboração** detalha mais o modelo criado na concepção para determinar a infra-estrutura econômica necessária para o projeto. Isso inclui a identificação e redução de riscos, estimativa de custos e recursos necessários e planejamento detalhado do projeto (incluindo cronogramas, por exemplo). Somente após esta fase é que se pode fazer, por exemplo, uma proposta comercial detalhada relacionada ao projeto.

A fase de **Construção** é onde o projeto toma corpo de fato. Ainda que nas fases anteriores os requisitos e a arquitetura do projeto tenham sido esboçadas com certo detalhe para permitir uma análise do projeto como um todo, é nesta fase que estes requisitos e arquitetura são definidos com todos os seus detalhes e então implementados sob forma de código e executáveis.

Por fim, a fase de **Transição** preocupa-se principalmente com a correção de defeitos e problemas não identificados durante as fases anteriores, bem como com um "ajuste fino" do *software* aos usuários. Realiza-se quando os usuários (ou um grupo

seleto mas significativo deles) já estão utilizando uma versão beta do *software*. Isto não significa que todo o *software* deva ser testado somente nesta fase! Pelo contrário, em cada iteração, dentro de cada fase, devem ser feitos testes para determinar a funcionalidade do que está sendo desenvolvido e devem-se corrigir os eventuais erros. A fase de transição trata dos erros que persistem apesar destes testes (tipicamente causados por grande número de usuários concorrentes, grandes períodos de utilização, utilização do programa para analisar um grande volume de dados, ou dados muito complexos, ou seja, condições custosas para se testar durante a implementação) e das questões e melhorias que inevitavelmente são levantadas pelos usuários mas que não tinham sido previstas nas fases anteriores.

Cada iteração apresenta as **etapas** de **Captura de Requerimentos**, **Análise**, **Projeto**, **Implementação** e **Testes**. Cada um destas etapas tem mais ou menos ênfase dependendo principalmente da fase em que se encontra o projeto.

A etapa de **Captura de Requerimentos** traduz como **casos de uso** e outros artefatos os requisitos ou requerimentos do *software*. Ou seja, o que o produto deve fazer é especificado de uma maneira formal nesta etapa. Embora formais, casos de uso descrevem os requisitos de forma bastante simples. Cada caso de uso descreve uma ação iniciada por um ator e que traz benefícios ou resultados observáveis. Nesta etapa normalmente a interface de usuário também é esboçada.

Durante a etapa de **Análise**, um subconjunto dos casos de uso mais significativos para o *software* é detalhado, assim como as classes mais importantes utilizadas na realização de cada um dos casos de uso escolhidos. Trata-se de um refinamento dos requerimentos capturados na etapa anterior, assim como um primeiro passo no projeto

do sistema. O projeto é dividido em pacotes ou subsistemas e generalizações entre classes são identificadas. Jacobson et al. (1998), no entanto, afirmam que o modelo obtido na análise nem sempre precisa ser mantido nas outras etapas do projeto e que, para projetos de pequeno porte, esta etapa pode ser inclusive eliminada (na verdade, absorvida pelas etapas de requerimentos, projeto e implementação).

Na etapa de **Projeto** o programa toma forma. É aqui que é definida em detalhe a arquitetura do *software* que é capaz de atender de forma eficiente aos requisitos definidos anteriormente. Aqui também são decididas outras "condições de contorno" do projeto, como por exemplo APIs utilizadas no sistema. O modelo obtido durante esta etapa é uma abstração da implementação do sistema.

É na etapa de **Implementação** que a arquitetura, os requerimentos e sua realização deixam de ser abstrações e são traduzidos em código compilável.

Por fim, na etapa de **Testes** é testado o que foi implementado. Os testes são planejados de acordo com uma estratégia. Nesta etapa também são criados vários artefatos necessários para a realização dos testes como por exemplo:

- casos de uso para descrever procedimentos de teste;
- arquivos ou bases de dados necessárias para rodar os testes;
- ferramentas que automatizem parte destes testes.

Em cada uma dessas etapas podem ser gerados diversos **artefatos** que descrevem o resultado do trabalho realizado. Um artefato é conjunto de informações que representa, à sua forma, o *software* ou uma parte do *software* sendo desenvolvido.

O objetivo aqui é deixar bem claro o conceito de artefato, e não descrever cada artefato do Processo Unificado. A descrição do subconjunto de artefatos utilizados neste projeto será feita no capítulo 4.

Para compreender melhor o conceito de artefato, é útil fazer uso de uma metáfora, utilizando a construção civil como objeto de comparação. Imagine que um projeto de uma obra possa ser completamente descrito (ainda que com alguma redundância) através das seguintes formas:

- Uma descrição verbal que registre os desejos do cliente que está contratando aquela obra;
- Um esboço de um artista traduzindo esta descrição;
- Uma maquete;
- Uma planta arquitetônica mostrando as paredes, portas, janelas, ambientes e outros elementos que compõem o espaço físico da obra;
- Uma planta do sistema elétrico da obra;
- Uma planta do sistema de encanamentos de água e esgoto da obra;
- Uma planta do sistema de ventilação e ar-condicionado da obra.

Cada uma destas formas de descrever a obra seria um artefato. Cada uma descreve a obra à sua maneira, com foco em aspectos diferentes, elaborada e destinada a ser vista por pessoas diferentes. Os esboços e maquetes são mais destinados à observação dos clientes ou dos arquitetos que partirão daquele conceito para detalhar a

obra, por exemplo, enquanto as plantas mais detalhadas, como a elétrica, são elaboradas e destinadas aos profissionais que projetam e implementam estas partes da obra. Nenhum destes artefatos é mais importante que o outro (embora cada um possa ter uma importância maior em determinada fase do projeto ou etapa de uma iteração) e **todos** descrevem a obra, cada um à sua maneira.

Da mesma forma, as descrições dos casos de uso (um artefato criado na etapa de captura de requerimentos), por exemplo, são uma descrição tão importante para um projeto de *software* quanto os diagramas de classe (outro artefato que descreve a arquitetura e a estrutura do *software*) ou quanto os códigos compiláveis e comentados. Cada um destes artefatos oferece uma visão com maior ou menor grau de detalhe e de um "ângulo" diferente. Cada um é mais adequado a determinada situação. Mas todos descrevem o produto de alguma forma.

Ao descrever cada etapa do desenvolvimento deste projeto, serão descritos também os artefatos que serão gerados naquela etapa e então o próprio artefato gerado é registrado.

3.2. Bibliotecas

Já foi definido anteriormente o que é uma API ou biblioteca no contexto de programação, e também já foi destacada a importância que essas ferramentas têm no desenvolvimento de aplicações portáveis. No entanto esta não é a maior vantagem que o uso de bibliotecas traz no desenvolvimento de *software*. Pode-se dizer que a maior

vantagem desta prática é permitir que os desenvolvedores se concentrem na implementação da aplicação que têm em mente, ao invés de se preocupar com detalhes periféricos (como *handlers* de janelas ou *pipelines* de renderização). E permite ainda que se utilize da experiência de especialistas em cada uma destas áreas periféricas à aplicação sendo desenvolvida, ao invés de se "reinventar a roda".

O uso de APIs traz também algumas desvantagens. A principal é que se cria uma dependência entre a aplicação e a biblioteca utilizada. Se a biblioteca pára de evoluir e se atualizar com os avanços de *software* e *hardware*, por exemplo, torna-se difícil atualizar a aplicação. Outra desvantagem é que justamente por encapsular alguns conhecimentos (sua função), a API pode acabar por isolar o desenvolvedor de informações que lhe seriam úteis no desenvolvimento do *software* ou na compreensão do sistema e de seu comportamento. Esta segunda desvantagem pode ser facilmente contornada com um estudo dos conhecimentos em questão, sem a necessidade de implementar cada detalhe.

A seguir serão discutidas as duas principais bibliotecas utilizadas no desenvolvimento deste trabalho, o VTK e o IUP e depois serão apresentadas algumas alternativas a estas bibliotecas que foram descartadas. Antes disto, porém, faz-se uma breve exposição sobre o processo de seleção destas bibliotecas.

Tanto o VTK quanto o IUP são bibliotecas gratuitas (mas que não exigem que produtos que as utilizem também o sejam), com código fonte disponível e excelente documentação, além de serem portáveis para diversas plataformas e atualizadas com boa freqüência.

O VTK é orientado a objetos, baseado em OpenGL, e tem um número tão elevado de recursos que por vezes não é tão simples saber qual utilizar para realizar uma tarefa. Dentre estes recursos destaca-se a possibilidade de paralelismo, que será discutida mais a fundo adiante.

O IUP, apesar de não ser orientado a objetos, tem como pontos fortes (além da excelente documentação já citada) a grande simplicidade de utilização e a facilidade de criação de janelas usando o seu *layout* abstrato.

3.2.1. O *Visualization Toolkit*

De acordo com Schroeder et al. (2000), o *Visualization Toolkit* (VTK) é um sistema de *software open-source*, portável e orientado a objetos para computação gráfica em 3D, visualização científica e processamento de imagens. É implementado em C++ mas também suporta Tcl, Python e Java. Por ser *open-source* e largamente utilizado, está em constante atualização. Sua página oficial (Kitware, 2004) disponibiliza até uma versão atualizada diariamente (a "*nightly release*"), além da versão oficial.

Schroeder et al. (1996) afirmam que devido à sua variedade de recursos e ao fato de ser *open-source*, docentes em diversas universidades utilizam o VTK como ferramenta de ensino e de pesquisa. Ahrens et al. (2000) relatam como o Laboratório de Los Alamos adaptou o VTK para processamento paralelo de grande escala, graças ao seu modelo de *visualization pipeline*, que facilita a paralelização. Law et al. (2001)

descrevem inclusive uma aplicação de grande escala (simulação de clima) que faz uso desta ferramenta. Esta versão paralela do VTK chama-se ParaView e também se encontra disponível como *open-source*. Além disso o VTK é usado em projetos comerciais nas áreas de visualização médica e de volumes, exploração de petróleo, acústica, CFD (*Computational Fluid Dynamics*), análise de elementos finitos e reconstrução de superfícies a partir de medidas com LASER.

Diversos padrões de projeto (*design patterns*) bem conhecidos, como fábrica de objetos, são utilizados nesse *toolkit* para garantir sua portabilidade e extensibilidade.

O VTK é composto principalmente por dois subsistemas, o modelo gráfico e a *Visualization Pipeline*. O modelo gráfico é uma camada abstrata sobre a linguagem gráfica. Atualmente, o VTK só suporta o OpenGL como linguagem gráfica, já que ele foi adotado como padrão pela indústria. No entanto, outras linguagens já foram suportadas e, mantendo esse modelo gráfico abstrato, é possível atualizar o VTK sem problemas de compatibilidade com versões antigas se outros padrões forem adotados no futuro.

A *Visualization Pipeline* (ou rede de visualização) transforma dados em formas que podem ser exibidas pelo sistema gráfico (imagens 2D, polígonos ou volumes). Ela é construída conectando objetos de dados e objetos de processo (também chamados de filtros).

Objetos de dados representam informação como um campo (um vetor de vetores) contendo informações geométricas (as coordenadas dos pontos) e, opcionalmente, informações topológicas (como os pontos estão conectados, formando linhas, faces etc.) Além disso, objetos de dados podem armazenar atributos para cada

ponto. Esses atributos podem ser escalares (como a temperatura em um ponto), vetores (por exemplo, velocidade), tensores, normais, ou texturas (o "ou" não é exclusivo, um mesmo ponto pode ter mais de um atributo de tipos diferentes, mas somente um atributo de cada tipo).

Objetos de processo operam sobre objetos de dados, seja gerando dados a partir de uma equação ou arquivo, transformando-os em outros tipos de dados (por exemplo, VTKContourFilter transforma dados volumétricos em polígonos) ou conectando esses dados ao sistema gráfico para que possam ser exibidos. Os objetos de processo que geram dados são chamados de *Sources* (fontes). Os que transformam dados são chamados de *Filters* (filtros) e os que conectam a *pipeline* ao sistema gráfico são chamados de *Mappers* (mapeadores). *Mappers* também podem, ao invés de exibir o objeto, salvá-lo num arquivo.

A execução dessa *pipeline* é controlada cuidadosamente para garantir a fidelidade dos dados e evitar operações desnecessárias. Filtros só são re-executados se houver uma mudança em seus parâmetros ou em sua entrada. Cada objeto no VTK tem um *time-stamp* interno que é alterado toda vez que o objeto em si sofre alguma mudança. Objetos de dados e de processos tem adicionalmente um *time-stamp* relacionado com sua execução dentro da *pipeline*. O sistema compara esses tempos para determinar quais objetos estão desatualizados e, portanto, que parte da *pipeline* deve ser re-executada. Dessa forma, o controle da execução dessas operações está distribuído entre os objetos. Muitos sistemas de visualização utilizam uma função centralizada para controlar essa execução, o que se torna um gargalo em aplicações altamente paralelizadas.

3.2.2. IUP - Interface com o Usuário Portável

A motivação para o desenvolvimento do IUP/LED, contam Levy et al. (1996), foi criar uma ferramenta para construção de interfaces que fosse de fácil aprendizado e de uso eficiente. Para tanto, as seguintes características figuraram entre os requisitos de projeto dessa ferramenta:

- Derivar aplicações para múltiplas plataformas a partir de uma única especificação e utilizando o mesmo ambiente de desenvolvimento, porém mantendo a aparência e comportamento (*look-and-feel*) de cada plataforma.

- Permitir a prototipação rápida de interfaces e a rápida incorporação de modificações às mesmas.

- Minimizar a quantidade de funções ou ferramentas que devem ser conhecidas pelo usuário antes que ele se torne produtivo.

- Ser expansível.

Conclui-se que um UIT[13] portável que suportasse um modelo de *layout* abstrato[14] das interfaces e que permitisse (mas não impusesse) a interpretação de descrições da interface em tempo de execução seria uma boa base para atender a esses requisitos. Com isso em mente, o IUP/LED foi desenvolvido com as seguintes características:

- Uma linguagem simples para descrição das interfaces (LED) de rápido aprendizado. A descrição da interface no LED é feita através da função de seus elementos. A especificação da aparência desses elementos é opcional.

- Um *toolkit* "virtual" (pois converte a especificação para outros *toolkits*, como SDK ou Motif) de suporte ao LED, o IUP. O IUP conta com funções que convertem especificações em LED para objetos de interface nativos, associam ações do usuário sobre a interface com funções da aplicação e obtém e modificam atributos de elementos da interface. Como simplicidade é um objetivo, o IUP tem um conjunto pequeno de funções (apenas 40).

- Um modelo simples de especificação da interface utilizando layout abstrato.

---

[13] User Interface Toolkits: Biblioteca de objetos de interface que implementam diferentes técnicas de interação com o usuário. Tipicamente incluem formas para simplificar a descrição e composição das interfaces, que vão desde uma linguagem a editores gráficos. Exemplos são SDK do MS Windows, OSF/Motif, XView e Macintosh Toolbox.

[14] O Layout concreto define a posição e tamanho exatos de cada componente da interface, enquanto o layout abstrato define as posições relativas entre esses objetos, sem a necessidade de calcular coordenadas exatas. O layout abstrato tipicamente é mais simples para ser especificado e permite que a interface se reconfigure para se adaptar a mudanças de tamanho (por exemplo quando o usuário maximiza uma janela).

- *Look-and-feel* nativo (aquele que o usuário de uma determinada plataforma espera, indicado para usuários que usam diversos programas em uma mesma plataforma) ou fixo (imposto pelo aplicativo, indicado no caso de o usuário utilizar o mesmo aplicativo em diversas plataformas distintas).

- Interpretação do LED em tempo de execução com *overhead* mínimo, para prototipação rápida, permitindo modificar-se a interface sem a necessidade de recompilar todo o código. A interface também pode ser criada diretamente no código, sem utilizar ou interpretar LED, utilizando o IUP.

- Portável para diversas plataformas, sem que o programador seja obrigado a ter conhecimento de cada uma dessas plataformas.

- Expansível.

- Permite acesso direto ao elemento de interface nativo do sistema (através de handlers do MS-Windows, por exemplo).

- Utiliza o modelo de funções *callback* para integração da interface com a aplicação.

Como já foi mencionado, essas características tiveram sucesso em tornar o IUP uma ferramenta bastante flexível e ao mesmo tempo simples de se utilizar. Mas de acordo com Levy et al. (1996) essa ferramenta ainda apresenta algumas deficiências, como:

- Não é orientado a objetos.

- Falta de suporte para *help* sensível a contexto e operações com a área de transferência (*cut-and-paste*).

- Não suporta MDI (*Multiple Document Interface*).

Essas duas funcionalidades podem ser implementadas utilizando o IUP/LED (por exemplo utilizando um Canvas, um objeto do IUP sobre o qual se pode desenhar, para desenhar os múltiplos documentos), porém seria um processo razoavelmente complexo, ao contrário da simplicidade exibida pela ferramenta para realizar as outras funções.

3.2.3. Alternativas Pesquisadas

Dentre as alternativas ao uso do VTK pesquisadas, destacam-se:

- **Performer**: A Silicon Graphics, Inc. (2002) descreve o Performer como uma API gráfica (também com base no OpenGL) voltada para renderização de cenas complexas e realistas em tempo real. Originalmente essa API era desenvolvida somente para o ambiente IRIX, mas recentemente (final de 2002) tornou-se compatível com os ambientes Linux e Windows NT, 2000 e XP. Sua arquitetura prevê e facilita paralelização de código. Utiliza uma pipeline para renderização e grafos (*scene graphs*) para modelar as cenas. É *software* proprietário da SGI[15].

[15] *Silicon Graphics, Inc.*

Apesar do Performer ser uma biblioteca bastante poderosa e flexível e também disponível na Escola Politécnica, foi preterido pelo VTK pois este tem mais recursos de visualização científica e é *open-source* ao invés de proprietário como era o Performer no início do trabalho.

- **Java 3D**: Esta biblioteca para Java desenvolvida pela Sun também conta com um grande número de recursos para exibição, navegação e interação com a cena e, assim como o Java, é gratuita e independente de plataforma. Em sua *homepage*, consultada em 2004, encontra-se uma documentação bastante completa. O uso da combinação Java/Java 3D tornaria desnecessário o uso do IUP, uma vez que o Java já conta com ferramentas para desenvolvimento de interface e que, mais uma vez, não dependem de plataforma. O Java 3D também utiliza *scene graphs* ou grafos de cena para abstrair a modelagem das cenas e é de uso relativamente simples. No entanto atribuir dados sobre a estrutura de polígonos usada pelo Java 3D não é trivial, o que dificulta o uso de técnicas de Visualização Científica. Além disso, ainda que a eficiência seja uma preocupação importante no desenvolvimento desta API, não deixa de ser uma linguagem interpretada e, portanto, é mais lenta e demanda mais memória que um programa compilado. Embora para muitas aplicações esta diferença não seja perceptível com a velocidade e capacidade de memória das máquinas atuais, este não é o caso neste trabalho. Recentemente, em 2004, o Java 3D passou a ser um projeto *open-source*.

- **VRML**: A VRML (*Virtual Reality Modeling Language*) é uma linguagem para descrição de mundos 3D, voltada para exibição destes mundos na *Internet*. Seu uso é bastante simples, porém não conta com um grande número de recursos. Pode ser

combinada, de forma limitada, com JavaScript ou Java para aumentar suas capacidades. No entanto, também é uma linguagem interpretada e não conta com um interpretador único que obedeça a todos os detalhes de sua especificação, como é o caso da máquina virtual do Java. O interpretador mais popular, por exemplo, o Cosmo, não tem capacidade de lidar com estereoscopia. Outros interpretadores fazem isso, mas não conseguem interpretar código em Java ou Java 3D *embedded* no VRML. Por esse motivo a especificação do VRML foi abandonada em favor de uma nova especificação, a X3D, que ainda estava em desenvolvimento durante a execução deste trabalho, mas recentemente teve uma primeira versão concluída. Pode-se encontrar mais informações sobre o VRML e o X3D na *homepage* do Web3D Consortium, consultada em 2004.

- **Open Inventor**: De acordo com Bicho *et al*.(2002), trata-se de uma *toolkit* orientada a objetos para o desenvolvimento de aplicações interativas gráficas 3D em C/C++, construída sobre o OpenGL, também baseada no conceito de grafos de cena (como o Java 3D) e que dispõe também de um formato de arquivo de dados 3D padrão para troca de informações entre aplicações e plataformas. Criado pela SGI, é *open-source* e independente de plataforma. Não conta, no entanto, com recursos para visualização científica como o VTK, apesar de ser uma excelente alternativa para outras aplicações.

Sem dúvida existem outras alternativas que poderiam ter sido pesquisadas mais a fundo, como o Open Scene Graph, no entanto ao longo desta pesquisa o VTK foi se mostrando cada vez mais adequado e por fim foi adotado.

Da mesma forma, o IUP se adequou tão bem às necessidades do projeto que não foi feita uma pesquisa aprofundada de alternativas para essa API. Tai (2004) e a própria documentação do IUP, entretanto, fazem um levantamento de diversas alternativas. Embora as informações nesta *homepage* nem sempre estejam completas (como é o caso da informação sobre as plataformas suportadas pelo IUP), ela serve como um ótimo ponto de partida para quem se dispuser a fazer uma análise de alternativas mais detalhada. A própria documentação do IUP cita alternativas ao seu uso.

3.3. *Shaders*

É oportuno fazer aqui uma exposição dos conceitos básicos relativos a *shaders*, pois esses conceitos são utilizados em diversos pontos do capítulo 6, ao se discutir trabalhos futuros, ainda que esse recurso não seja usado em outros pontos neste trabalho, pelo motivo que será discutido adiante.

De acordo com Kessenich et al. (2004), *Shaders* são programas simples que poder ser executados diretamente nas placas gráficas, nos dois pontos da *pipeline* que são programáveis no *hardware* mais recente: *Vertex Shaders* são programas que agem sobre todos os vértices que são enviados à *pipeline* e *Pixel* ou *Fragment Shaders* agem sobre os pixels antes que sejam exibidos. De acordo com Barrera (2004), *shaders* estão disponíveis no OpenGL desde 2002, pouco após o surgimento de *hardware* gráfico programável, através de duas de suas extensões. Era necessário, no entanto, programá-los em Assembly, o que dificultava e limitava consideravelmente seu uso. Após o surgimento, mais recente, de linguagens de alto nível para *shaders* como a HLSL da

Microsoft e a Cg da NVidia, foi desenvolvida também uma linguagem de *shading* para o OpenGL, na tentativa de se estabelecer um padrão. Essa linguagem, a GLSL (GL Shading Language), só passou a ser suportada por *hardware* na segunda metade de 2004, pelos *drivers* da série 60 da NVidia e pelas placas GeForce da série FX. Barrera conta que esses programas podem ser usados para implementar diretamente no processador gráfico diversos efeitos como modelos mais realistas de iluminação, *bump-mapping*, sombras entre muitos outros. Como o suporte de *hardware* para a GLSL só surgiu quando este trabalho já estava em sua fase de conclusão, esse recurso não é usado, exceto na discussão de trabalhos futuros.

4. A METODOLOGIA E O RISERVIEW

Este capítulo tem uma dupla finalidade. Em primeiro lugar discute a metodologia usada no desenvolvimento do **RiserView** - o ambiente de visualização de *risers*. Mas vai além disso. A metodologia é exposta através de sua aplicação neste projeto e, dessa forma, o próprio projeto é exposto, através das diversas vistas oferecidas por ela. Este é o segundo objetivo deste capítulo: oferecer uma visão do **RiserView** que vai desde o geral até um grau maior de detalhe. É um capítulo de aspecto mais prático, em sua maior parte. Embora todos os aspectos teóricos discutidos anteriormente tenham sido, necessariamente, levados em consideração durante todo o processo, este capítulo não se preocupa em descrever como, mas sim em exibir o processo de projeto e implementação do Ambiente bem como o Ambiente em si.

Primeiramente são detalhadas e justificadas as customizações feitas à metodologia para o projeto do **RiserView**. Em seguida a parte mais importante do capítulo exibe como cada passo desta metodologia customizada é aplicado e implementado, oferecendo não só uma visão da metodologia como várias visões do **RiserView**.

4.1. Customização da Metodologia

Conforme já se espera ter deixado claro, o Processo Unificado não se trata de uma única metodologia, mas sim de uma *framework* completa que pode ser adaptada como metodologia para projetos com escalas e características bastante distintas. Por isso conta com uma grande variedade e quantidade de artefatos, para que tenha esta adaptabilidade. Assim sendo, a adaptação ou customização da metodologia para um determinado projeto, a escolha do subconjunto de artefatos mais significativos, é um trabalho necessário para seu uso. Além disso, adequando o processo à escala do projeto reduz-se em muito o desperdício de tempo com a criação e manutenção de artefatos desnecessários.

Enquanto até agora neste capítulo tenha sido necessário discutir o Processo Unificado de uma maneira mais geral, a partir de agora passa-se a discutir sua aplicação no projeto em questão.

A primeira simplificação feita na metodologia para este projeto está relacionada às suas **fases**. De acordo com Smith (2002), as fases de concepção e elaboração, por exemplo, podem ser bastante reduzidas em projetos de pequena escala e baixo risco. Estas fases foram percorridas (o projeto, inicialmente, foi especificado suficientemente para que se pudesse analisar a viabilidade de executá-lo com o tempo e os recursos físicos e humanos disponíveis, foi feito um cronograma para o projeto etc.), mas isto foi feito de forma bastante abreviada, não só devido à escala do projeto, como também porque o foco deste trabalho são os aspectos técnicos de projeto e arquitetura de *software*, computação gráfica, visualização científica e outros pontos discutidos no

capítulo anterior e não aspectos econômicos. Desta forma, propositadamente não estão registradas neste trabalho as fases de concepção e elaboração do projeto. Já a fase de transição só será iniciada com o fim deste trabalho e a liberação do programa para um maior número de usuários e portanto também não pode ser descrita aqui. Daqui em diante este trabalho tratará, portanto, unicamente da fase de construção.

Outra simplificação importante está relacionada com as iterações do processo. O desenvolvimento deste projeto foi dividido em três iterações incrementais, que serão discutidas na próxima seção. No entanto, ao invés de discutir as etapas de cada uma destas iterações separadamente, elas serão discutidas de uma só vez. Os artefatos registrados são o resultado de todas as iterações. Como as iterações são incrementais, os artefatos também crescem com cada iteração. Ao invés de mostrar os artefatos gerados em cada uma - o que iria requerer um trabalho adicional e gerar mais redundância nestes artefatos - são exibidos os artefatos finais, resultantes da última iteração.

Sendo assim, a descrição do projeto pode ser feita como o registro de uma única seqüência de etapas e dos artefatos produzidos em cada uma delas. É desta forma que este capítulo é estruturado a partir da próxima seção.

Como já se chamou à atenção, Jacobson et al. (1998) afirmam que o modelo obtido na etapa de análise nem sempre precisa ser mantido nas outras etapas do projeto e que, para projetos de pequeno porte, esta etapa pode ser inclusive eliminada (na verdade, absorvida pelas etapas de requerimentos, projeto e implementação). Esta simplificação também foi adotada neste projeto.

Por fim, em cada uma das etapas que serão descritas a seguir foi escolhido um subconjunto de artefatos adequados. Este subconjunto e as razões para sua escolha estão listados em cada etapa.

4.2. Iterações

Uma das principais características do Processo Unificado é se tratar de um processo iterativo. Cada uma de suas fases é composta por uma ou mais iterações. Já foi afirmado que, neste trabalho, as fases de concepção e elaboração foram feitas abreviadamente em somente uma iteração. A fase de construção, que será detalhada aqui, no entanto, é composta por três iterações principais:

i. Na primeira iteração a preocupação é exibir e animar em três dimensões objetos de interesse, a princípio separadamente, com base em arquivos de dados descrevendo ao longo do tempo os estados dos diversos objetos (principalmente *risers*, mas também vórtices representados discretamente ou por campos de escalares, a superfície da lâmina d'água, o fundo oceânico e modelos para representar embarcações ou outras estruturas rígidas). A navegação do usuário pelas cenas exibindo estes objetos também é implementada nesta fase, bem como a iluminação da cena, aplicação de texturas e *anti-aliasing*.

ii. A segunda iteração é mais relacionada com a interface de usuário do programa. Nesta iteração é elaborada a interface e é integrada à ela a visualização de cada um dos objetos, que tinha sido desenvolvida na iteração anterior. Também são

implementados mecanismos para controle da animação (parar, avançar, voltar, quadro a quadro, acelerada...) e de detalhes da visualização (cores, texturas, *anti-aliasing*, vistas, o que exibir etc.).

iii. Por fim, na última iteração foi implementada a detecção de colisões entre *risers*.

Como já se disse, embora em cada uma destas iterações tenha-se tratado de um subconjunto dos casos de uso e se preocupado com somente parte da arquitetura, o que será apresentado neste capítulo será o resultado completo e final do trabalho, sem a preocupação de registrar aqui os artefatos gerados em cada uma das iterações.

4.3. Captura de Requerimentos

Conforme indicado na seção anterior, com as simplificações feitas na etapa de customização do Processo Unificado para este projeto, sua descrição pode ser feita como um registro de uma única seqüência das seguintes etapas: Captura de Requerimentos, Projeto, Implementação e Testes. Estas etapas serão descritas principalmente por um conjunto de artefatos gerados em cada uma delas. Cada um destes artefatos, deve-se lembrar, oferece uma vista do **RiserView**. Nesta seção descreve-se a primeira destas etapas, a Captura de Requerimentos.

Como esta é a etapa que vai nortear todo o trabalho daqui por diante, deve ser feita da maneira mais cuidadosa possível. Um erro nesta etapa pode causar uma grande quantidade de retrabalho no projeto. Por isso, esta é a etapa que conta, neste projeto, com o maior número de artefatos: diagramas de casos de uso, uma lista de atores, uma

lista de casos de uso com uma descrição textual de cada caso, uma lista de requerimentos especiais (não associados a um caso de uso específico), uma descrição textual dos requerimentos como um todo, um glossário e um protótipo da interface com o usuário.

4.3.1. Descrição dos Requerimentos

Este primeiro artefato nada mais é que uma descrição textual dos requerimentos do *software*, menos detalhada que os casos de uso. É um primeiro esboço de especificação do *software*, serve para nortear os próximos passos bem como para fornecer uma rápida visão panorâmica do **RiserView**.

O *software* desenvolvido neste trabalho deve ser capaz de exibir cenas em três dimensões compostas por diversos objetos distintos. Os objetos de maior interesse são os *risers*. Outros objetos que podem compor a cena são representações de vórtices, da superfície da lâmina d'água, do fundo e modelos rígidos para representar embarcações ou estruturas. Exceto pelo fundo, cada um destes objetos tem sua dinâmica própria, que deve ser mostrada utilizando animação e outras técnicas de visualização científica. A descrição dos objetos e de sua dinâmica é dada por arquivos com formatos definidos, que são selecionados pelo usuário para montar a cena. Para facilitar a geração de cenas, deve ser possível salvar arquivos que associem mais de um objeto em um só arquivo.

O usuário deve ser capaz de navegar pela cena utilizando mouse e teclado, aproximando-se ou afastando-se dela, girando a cena ou a arrastando em qualquer direção. A navegação, no entanto, deve ser restrita a vistas verticais ou horizontais.

O usuário também deve ser capaz de controlar a animação, interrompendo e retomando seu fluxo a qualquer momento (desde que haja algo para animar), avançando ou voltando um quadro por vez, ou, ao contrário, em velocidade acelerada. Quando não acelerada, a animação deve ocorrer em tempo real, se possível. A interface deve exibir um mostrador com o tempo atual da animação, em segundos.

O controle da exibição ou não de cada tipo de objeto também deve ser permitido pela interface. Para os vórtices, deve ser permitido escolher se devem ser exibidos vórtices discretos ou campos escalares. O usuário deve também ser capaz de controlar os parâmetros de exibição dos *risers* (número de faces no cilindro e fatores de amplificação para o diâmetro e para o movimento), as cores do fundo da tela, dos modelos do solo e da superfície oceânica, bem como as texturas e coordenadas de textura para o solo e a superfície. Outras opções que podem ser controladas pelo usuário são a ativação ou não de *anti-aliasing*, estereoscopia e detecção de colisões entre *risers*. O usuário deve ser capaz de salvar estas opções em arquivo e recuperá-las posteriormente.

Caso a detecção de colisões seja ativada, a amplificação dos diâmetros e dos movimentos dos *risers* é desativada e, cada vez que ocorre uma colisão, a cor dos *risers* envolvidos é modificada na vizinhança do ponto de colisão, uma mensagem de alerta é mostrada pela interface e a animação é interrompida, de forma a permitir que o usuário possa registrar o momento e local da colisão.

4.3.2. Atores

A lista de atores é um artefato auxiliar, que tipicamente serve como referência para os diagramas e descrições dos casos de uso. No contexto dos casos de uso, Jacobson et al. (1998) definem um ator como "um conjunto coerente de papéis que um usuário dos casos de uso desempenha quando interagindo com estes casos de uso". Por exemplo, num sistema de leilões pela *Internet*, podem existir 3 atores: o comprador (representando todos que fazem ofertas por um item), o vendedor (dono atual do item) e o leiloeiro. Nem todos os atores precisam ser humanos. No caso anterior, o leiloeiro pode ser um subsistema que interage com casos de uso do sistema de leilões. No caso do **RiserView**, ao menos para esta versão, só existe um ator, o usuário, definido abaixo:

Usuário: O usuário final do programa é o único ator do sistema e só existe um tipo de usuário. Este ator acessa todas as capacidades do programa, podendo criar cenas a partir de arquivos com dados dos objetos de interesse, controlar a exibição de elementos da cena, sua visualização e animação e também navegar pela cena.

4.3.3. Diagramas de Casos de Uso

Este artefato tem como finalidade exibir de uma maneira gráfica e rápida os casos de uso que serão detalhados em outro artefato. Como o número de casos de uso de um sistema tende a ser muito grande, muitas vezes torna-se útil agrupá-los numa forma que agilize sua identificação visual. Isto é especialmente verdadeiro em sistemas mais complexos, com maior número de atores se relacionando de maneiras distintas com os casos de uso, mas vale mesmo para sistemas mais simples. Nos diagramas de casos de uso também podem ser exibidas outras relações entre eles, como relacionamentos ou generalizações (analogamente aos relacionamentos entre classes, descritos no capítulo anterior).

No caso específico do projeto do **RiserView**, os diagramas de casos de uso servem a dois propósitos principais:

Primeiro, classificar os casos de uso de acordo com as funções que realizam, dividindo-os em subgrupos de mais fácil compreensão e permitindo ver quais são todos os casos de uso relacionados àquela função com uma consulta bastante rápida ao diagrama. Os cinco subgrupos em que os casos de uso estão divididos são: Criação de Cena, Controle de Exibição, Visualização, Animação e Navegação.

O segundo objetivo é exibir algumas generalizações utilizadas. Para simplificar a descrição dos casos de uso e evitar redundâncias, alguns casos de uso que são muito semelhantes para elementos diferentes da cena (por exemplo, adicionar *riser*, modelo,

ondas etc.) foram generalizados em um único caso, que é o que está descrito. Todos os casos e estas generalizações, no entanto, estão listados nos diagramas a seguir.

Na figura 15 está o diagrama dos casos de uso de criação de cena. Estes primeiros casos de uso permitem ao usuário adicionar elementos à cena ou, em contrapartida, eliminar todos os seus elementos. Neste diagrama, os casos Adicionar *Riser*, Adicionar Vórtices, Adicionar Ondas, Adicionar Fundo e Adicionar Modelo são extensões do caso Adicionar Objeto. Listas permitem ao usuário adicionar mais de um objeto à cena com somente uma operação.

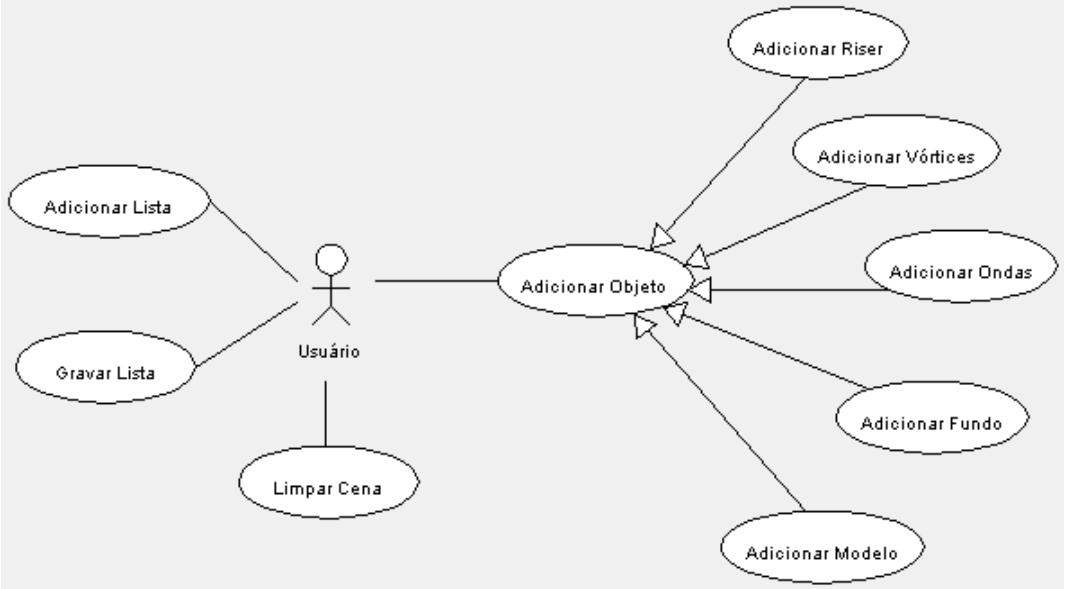

Figura 15 - Artefato: diagrama de casos de uso para criação de cena

Os casos de uso na figura 16 permitem ao usuário escolher quais elementos são exibidos ou não na cena, bem como ativar ou não algumas características do *software* (estereoscopia, *anti-aliasing*, colisões). Mais uma vez a generalização é utilizada para exibir ou não os objetos, exceto no caso dos vórtices, que tem um tratamento especial.

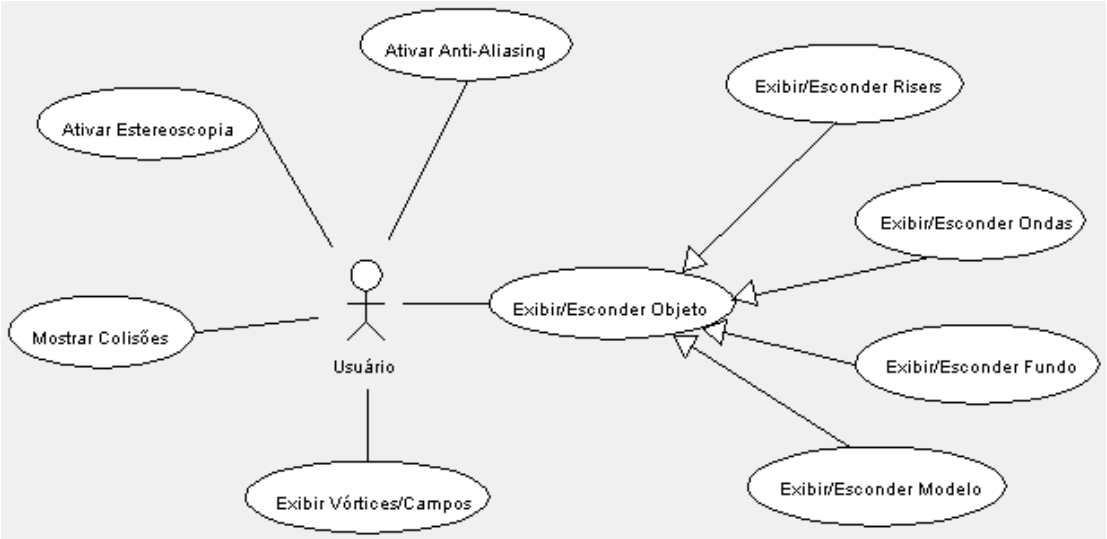

Figura 16 - Artefato: diagrama de casos de uso de controle de exibição

O diagrama da figura 17 agrupa os casos de uso que permitem ao usuário modificar a visualização da cena. Além disso, as opções escolhidas pelo usuário podem ser salvas em arquivo para serem usadas em outras cenas, posteriormente.

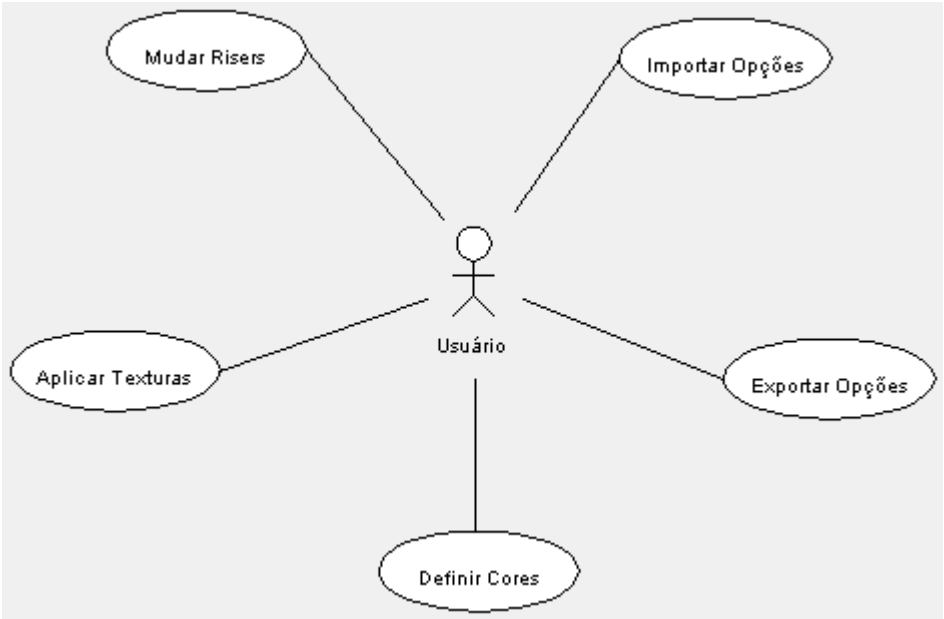

Figura 17 - Artefato: diagrama de casos de uso de visualização

No diagrama da figura 18 estão reunidos os casos de uso que permitem que o usuário controle a animação.

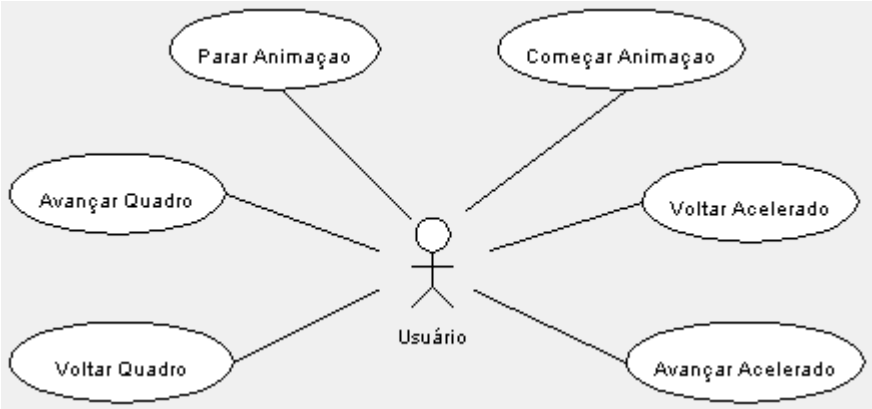

Figura 18 - Artefato: diagrama de casos de uso de animação

Por fim, a figura 19 contém os casos de uso que permitem que o usuário navegue pela cena.

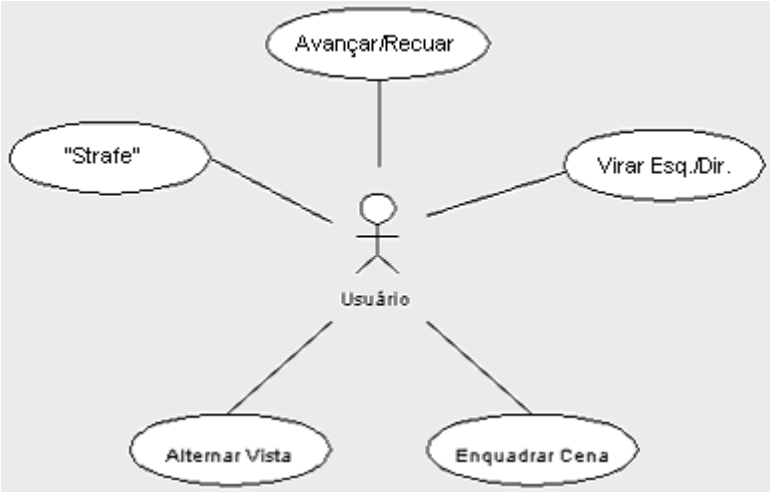

Figura 19 - Artefato: diagrama de casos de uso de navegação

4.3.4. Descrição dos Casos de Uso

Este artefato é possivelmente o mais importante nesta etapa. A captura dos requerimentos como casos de uso é um dos pontos centrais do Processo Unificado (bem como de muitas outras metodologias). Casos de uso, que já foram discutidos no capítulo anterior, são uma forma relativamente simples de capturar requerimentos funcionais, uma vez que sempre descrevem uma ação que desempenhe algo de valor para seu ator ou usuário. Num desenvolvimento guiado pelos casos de uso, sempre se está acrescentando funcionalidade ao *software*. Além disso, a descrição dos casos de uso se presta como um ótimo guia para a realização de testes do tipo "caixa-preta" (ou seja, testes em que se está preocupado somente com a funcionalidade e não se tem conhecimento da implementação), durante a etapa de testes.

Neste trabalho a descrição dos casos de uso está isolada no Apêndice A. A decisão de registrar os casos de uso num Apêndice não quer dizer que sejam de menor importância, muito pelo contrário. Foi tomada simplesmente porque este artefato é relativamente extenso e incluí-lo no corpo deste capítulo poderia interferir no fluxo do texto e das idéias.

4.3.5. Requerimentos Especiais

Apesar de sua importância e utilidade, os casos de uso, pela sua própria definição, não são capazes de traduzir todos os requisitos de um sistema. Como sempre traduzem uma ação que realize algo de valor para o usuário, os casos de uso só podem capturar os **requisitos funcionais** do sistema. No entanto, existem outros requisitos que devem ser então registrados em outro artefato, o de requerimentos especiais. A seguir é listado os únicos requerimentos especiais do **RiserView**.

i. *software* deve rodar sobre o MS-Windows e o X-Windows, minimamente.

ii. A navegação pela cena deve ser restrita a permanecer sempre numa vista ou vertical ou horizontal.

4.3.6. Glossário

Este artefato é bastante útil, não para apresentar uma vista do projeto, mas para permitir que todos os envolvidos falem a mesma linguagem.

O glossário utilizado ao longo deste projeto pode ser encontrado no Apêndice B. Deve-se ter em mente que, embora muitos dos termos referidos no glossário possam ser

utilizados em outros capítulos deste trabalho, o glossário do Apêndice B aplica-se a este capítulo, à metodologia.

4.3.7. Esboço da Interface

Este último artefato, um esboço da interface com o usuário, tipicamente é criado na etapa de captura de requerimentos. Neste projeto, isto foi feito durante a segunda iteração, já que na primeira a interface não era uma preocupação.

O uso do IUP, como discutido no capítulo anterior, facilitou consideravelmente a confecção deste artefato, uma vez que simplifica a criação de um protótipo da interface. Este protótipo foi feito de forma bastante rápida e continha todos os elementos da interface, mas sem funcionalidade.

A interface do **RiserView** é bastante simples. Consiste principalmente de uma janela onde o usuário pode navegar pela cena (para compreensão de como é feita a navegação, ver a descrição dos casos de uso relacionados). A barra de ferramentas contém funções úteis durante a visualização da cena, como limpar a cena, controle da animação e da câmera. na barra de ferramentas é exibido ainda o tempo atual da simulação. Por fim, a interface conta com menus para permitir que o usuário inicie a maioria dos casos de uso detalhados anteriormente. A figura 20 mostra o protótipo da interface num ambiente MS-Windows. Graças ao IUP o **RiserView** tem o *look-and-feel* de cada ambiente em que é executado, por isso esta mesma interface num ambiente Linux, por exemplo, seria similar mas não idêntica.

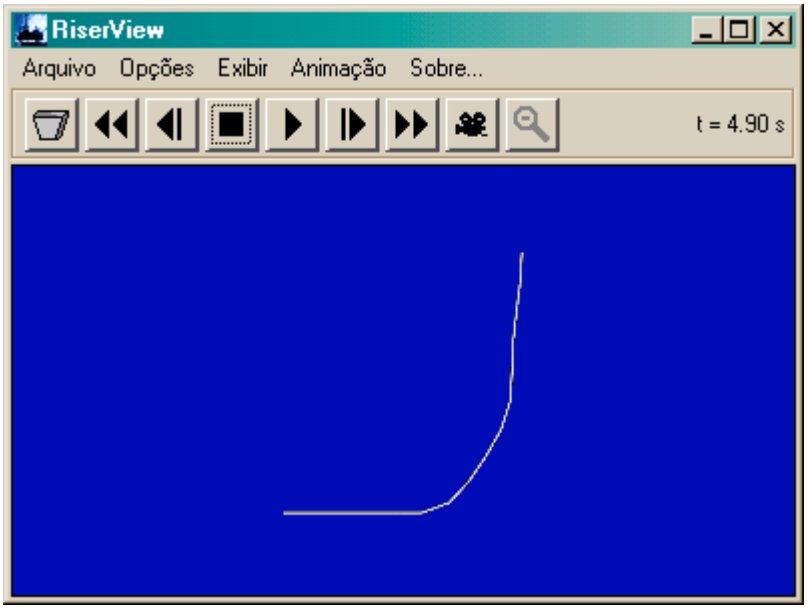

Figura 20 - Artefato: esboço da interface

No **manual do usuário**, um outro artefato gerado durante a etapa de implementação, cada um destes botões da barra de tarefas, menus e seus submenus, bem como as caixas de diálogo utilizadas no programa são exibidos e explicados em detalhe. Embora todos estes elementos tenham sido prototipados nesta etapa, esta informação não será repetida aqui para evitar redundância no texto.

4.4. Projeto

Esta é a etapa em que a arquitetura do projeto é definida, desde as APIs a serem utilizadas até a estrutura de classes detalhada que será implementada na etapa seguinte. Esta seção está organizada de uma forma diferente da anterior. Enquanto na seção de captura de requerimentos cada artefato era uma subseção, aqui é feita uma divisão em tópicos relativos à arquitetura, organizados do geral para o específico, que quando

analisados em conjunto devem ser capazes de conferir ao leitor uma visão bastante clara da arquitetura, da estrutura e do funcionamento do **RiserView** (embora um entendimento completo deve necessariamente passar por uma análise mais profunda das APIs utilizadas, o IUP e o VTK, cuja exposição com maior detalhe que o conferido no capítulo anterior foge ao escopo deste trabalho).

Primeiramente é apresentado um diagrama de subsistemas mostrando como o *software* se relaciona com as APIs utilizadas em seu desenvolvimento. Embora o **RiserView** em si seja composto por somente um pacote, as APIs nas camadas de *middleware* e de sistema podem ser consideradas como seus subsistemas.

Em seguida a estrutura de classes é mostrada por meio de diagramas com algumas simplificações importantes: os construtores, destrutores e as funções de acesso aos atributos das classes nunca são exibidos nesses diagramas, nem tampouco são mostrados os métodos herdados de uma superclasse, mesmo quando são redefinidos. Por fim, em muitas classes os atributos que correspondem a elementos da *pipeline* de visualização do VTK são omitidos. Todas essas simplificações visam facilitar a visualização dos diagramas, privilegiando a compreensão da arquitetura do aplicativo e não seus detalhes (que podem ser melhor analisados no código fonte).

O primeiro diagrama de classes, mais simples, mostra como o padrão Model-View-Controller (MVC, discutido no capítulo anterior) está implementado no **RiserView**. A classe de controle, embora seja bastante extensa, com um grande número de métodos, é também bastante simples. A grande maioria desses métodos trata das operações com menus e outros elementos da interface, chamando métodos adequados do modelo ou da vista ou simplesmente atribuindo um valor a uma variável de estado do

sistema. Por isso, o controle é retratado de forma simplificada, visto que nestes diagramas não se deseja passar este nível de detalhe.

Os próximos diagramas de classes mostram o Modelo em mais detalhe. O último diagrama de classes detalha a estrutura utilizada para armazenar dados utilizados durante a detecção de colisão entre *risers*.

Antes da discussão sobre detecção de colisão, no entanto, o laço principal do programa é detalhado por meio de dois diagramas. O diagrama de atividades é um simples fluxograma ilustrando este laço de forma simplificada. O diagrama de seqüência é um diagrama um pouco mais complexo, mostrando os objetos e mensagens envolvidos durante a atualização da cena. Este diagrama de seqüência também auxilia na compreensão de como o padrão MVC é implementado, mostrando as mensagens trocadas entre o modelo, a vista, o controle e os objetos que o compõem durante a atualização da cena.

Em conjunto, estes diagramas mostram praticamente a totalidade das classes desenvolvidas, com exceção de algumas classes auxiliares de menor importância. Juntamente com os comentários que os acompanham, é possível compreender com clareza a estrutura e o funcionamento do **RiserView**. Poderiam ser criados diversos outros diagramas, por exemplo diagramas de atividades e seqüência para cada um dos casos de uso. No entanto, da mesma forma que nos diagramas de classe as funções mais simples do controle, relativas somente à interface, foram suprimidas por serem detalhes de menor importância, acredita-se que estes diagramas também não contribuiriam consideravelmente para um melhor entendimento do **RiserView**. A maioria dessas operações é bastante simples e os casos de uso já estão detalhados suficientemente. Para

verificar detalhes da implementação, uma vez que a estrutura do programa tenha sido assimilada, pode-se recorrer às suas fontes comentadas.

4.4.1. As APIs

Conforme discutido no capítulo anterior, a utilização de APIs adequadas no desenvolvimento do **RiserView** foi uma decisão tomada desde o início do projeto. Estas APIs têm uma dupla função neste projeto: reduzir a preocupação com questões relativas às plataformas sobre as quais o programa deveria rodar bem como evitar a necessidade da implementação de funções gráficas de baixo nível, permitindo que se concentre a atenção no desenvolvimento da aplicação em questão e nos conceitos de mais alto nível envolvidos no seu desenvolvimento, como a visualização científica e a metodologia de projeto. Naquele capítulo discute-se as principais características do VTK e do IUP, levadas em conta durante sua seleção.

Essa seleção baseou-se em diversos fatores, como disponibilidade, quantidade de recursos disponível, freqüência de atualização da biblioteca, orientação a objetos, eficiência (por exemplo se a API utilizava uma linguagem interpretada ou não), simplicidade de uso e qualidade da documentação disponível. Este último quesito é bastante importante, pois pouco auxilia uma biblioteca com um sem número de recursos mas nenhuma indicação de como utilizá-los.

A figura 21 mostra o diagrama de subsistemas do **RiserView**, mostrando estas APIs como subsistemas do programa e indicando as relações de dependência entre esses

subsistemas. Este diagrama ilustra também como um dos padrões de arquitetura discutidos no capítulo 2, o padrão de camadas, está implícito na arquitetura do sistema devido ao uso destas APIs. Este padrão organiza os componentes de um sistema em camadas, de forma que cada componente só pode acessar outros componentes na camada diretamente abaixo da sua.

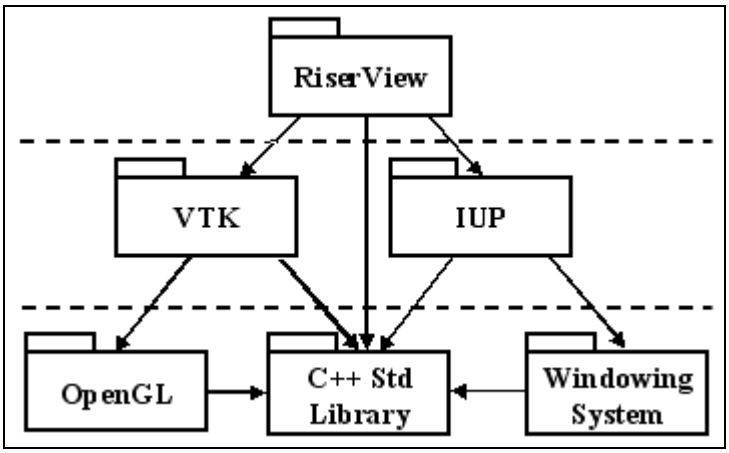

Figura 21 - Artefato: diagrama de subsistemas

4.4.2. O Padrão Model-View-Controller

Este padrão é utilizado neste projeto para separar os dados e operações do Modelo de preocupações com a interface. Recordando o que foi dito quanto ao MVC no capítulo 2, sua função é justamente separar dados intrínsecos ao problema que o *software* se dispõe a solucionar de dados necessários para a visualização ou interface. O **Modelo** (*Model*) é uma classe que contém estes dados intrínsecos à aplicação, é uma abstração de uma entidade relacionada a um domínio (uma entidade física, ou gráfica, ou matemática...) que não tem nenhum conhecimento da interface. **Vistas** (*Views*) são as

representações do Modelo na interface. Cada Vista é gerenciada por um **Controlador** (Controller) que é responsável pelas ações definidas na Vista com Relação ao modelo. O controlador é que "traduz" mensagens de interface para a lógica da aplicação. Sempre que o modelo é modificado (seja por uma ação do usuário, seja por outro motivo qualquer) as Vistas e Controladores devem ser notificados para se atualizarem. O encapsulamento de informações relativas à interface nas Vistas e seu gerenciamento pelos Controladores leva a um código mais limpo e elegante do Modelo, que é a base do *software* e só precisa tratar do seu domínio.

No **RiserView**, o Modelo representa a cena e seus elementos, estáticos ou dinâmicos: *risers*, o fundo do oceano, sua superfície, representações de vórtices, corpos rígidos modelando objetos como navios, plataformas, "árvores de natal" etc. O programa conta com uma única Vista, uma renderização desta cena em três dimensões que permite que o usuário navegue por ela.

A figura 22 ilustra, através de um diagrama de classes, como este padrão está implementado no **RiserView**, quais classes correspondem ao Modelo, à Vista e ao Controle e como estas classes se relacionam. A classe com o mesmo nome da aplicação serve como um *container* para as classes de Modelo, Vista e Controle.

Conforme foi dito, a classe de modelo é bastante complexa e será detalhada nos próximos diagramas e a classe de controle, apesar de bastante extensa por tratar de todas as diversas mensagens de usuários, sua interação com menus, janelas etc., é razoavelmente simples e seu detalhamento não é de vital importância para a compreensão da arquitetura. Por isso estas duas classes aparecem de maneira simplificada neste diagrama.

A notificação da Vista sobre as mudanças feitas no Modelo é feita através da *pipeline* de visualização do VTK, discutida no capítulo 2. Da mesma forma, as ações do usuário passam da Vista para o Controle através das funções de *callback* do IUP (funções globais associadas a elementos da Vista que por sua vez chamam funções do Controle).

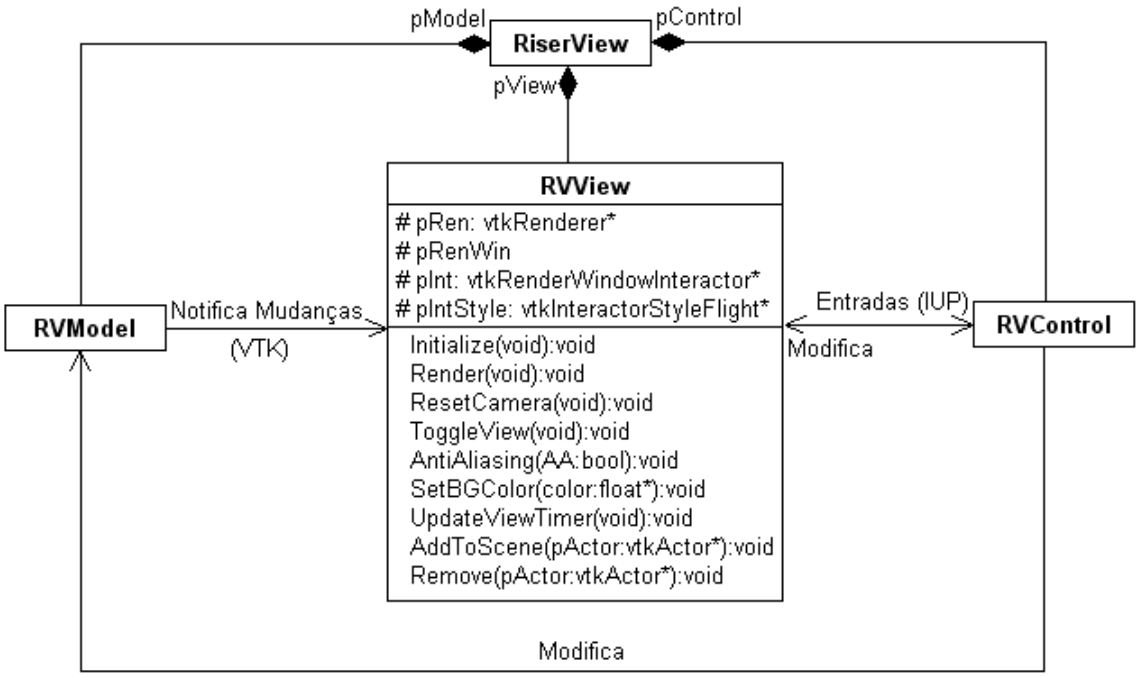

Figura 22 - Artefato: diagrama de classes do MVC

4.4.3. O Modelo

Como já foi dito, o Modelo representa a cena e seus elementos. Nele estão armazenadas as estruturas de dados que descrevem a geometria e a dinâmica de cada um destes elementos que compõem a cena. Devido à complexidade dessa classe e dessas estruturas, elas estão representadas em dois diagramas complementares. O

diagrama da figura 23 mostra principalmente as agregações que compõem o modelo. Já na figura 24 são representadas principalmente as generalizações utilizadas.

Os elementos de cena representados nos diagramas 23 e 24, bem como as classes utilizadas em sua implementação, são:

- *Risers*: os principais objetos da visualização são implementados através da classe **Riser**, que se especializa em duas outras classes, **RiserFreq** e **RiserTime**, para implementar *risers* cuja dinâmica é descrita seja no domínio da freqüência ou no do tempo. Como a cena pode conter diversos *risers*, a classe **RiserCol** agrupa estes objetos. Por fim, a classe **RiserFactory** é a responsável pela criação ora de um objeto do tipo **RiserFreq** ora do tipo **RiserTime**. Este padrão, onde uma classe distinta é responsável pela criação de objetos que podem ser de classes diferentes, foi discutido no capítulo anterior e é conhecido como "Fábrica de Objetos". A classe **RiserCol** é ainda responsável por gerenciar a detecção de colisão, que será discutida mais adiante.

- Representações de Vórtices: no **RiserView** as esteiras de vórtices que se formam a jusante dos *risers* podem ser representadas como conjuntos de vórtices discretos ou como campos de grandezas escalares, como pressão, módulo da velocidade ou vorticidade. Ambas estas representações são planas, mas podem ser mostradas em diversos planos em diferentes alturas do *riser* para ilustrar o fenômeno tridimensional. A classe **RVVortex** contém os dados referentes a essas representações.

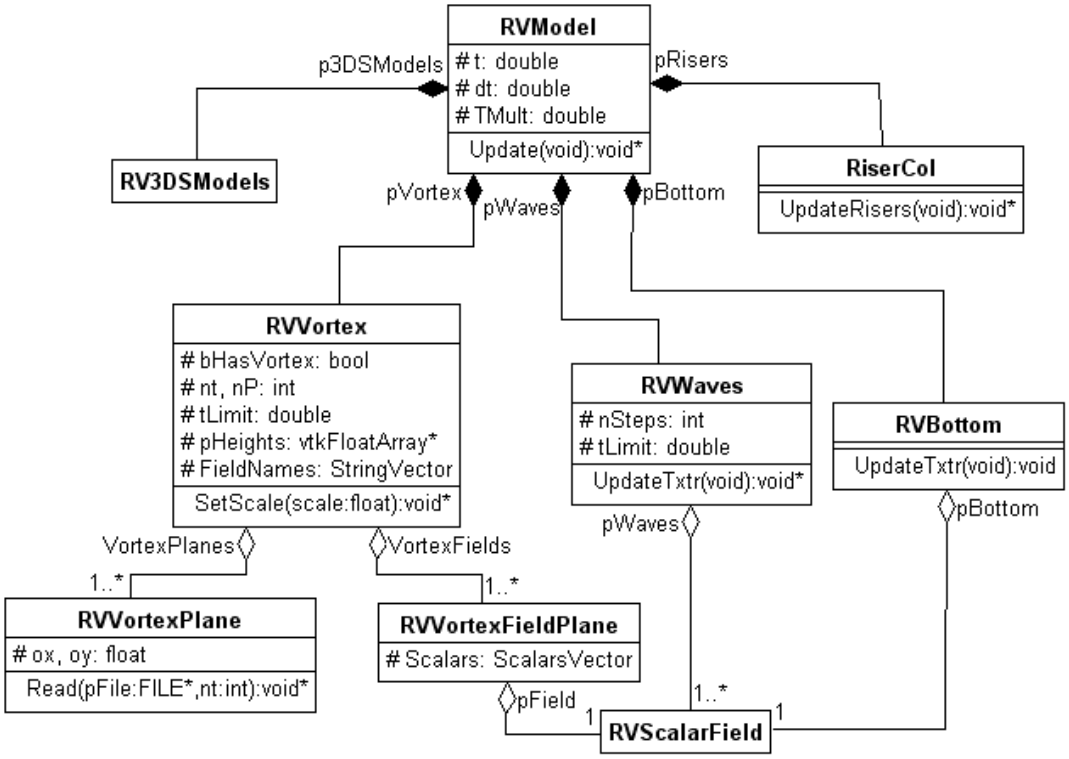

Figura 23 - Artefato: diagrama de classes do modelo (agregações)

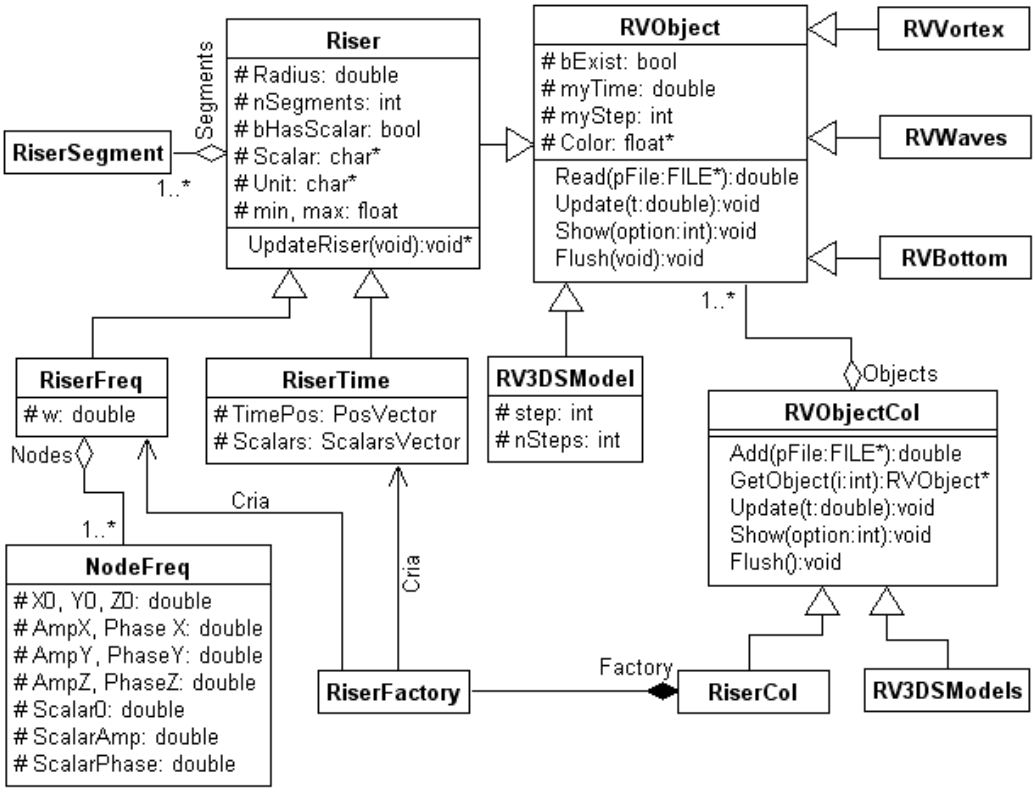

Figura 24 - Artefato: diagrama de classes do modelo (generalizações)

- Superfície do Oceano: a dinâmica da superfície pode ser representada neste programa através da classe **RVWaves**.

- Solo Oceânico: O relevo do solo também pode ser representado, de maneira estática, no **RiserView**. A classe que armazena essa geometria é a **RVBottom**.

- Corpos Rígidos: Diversos corpos rígidos podem compor a cena e podem inclusive possuir dinâmica própria. Atualmente, estes corpos são importados como modelos do 3D Studio, com um arquivo auxiliar separado que descreve sua dinâmica. A classe **RV3DSModel** armazena os dados de geometria e dinâmica desses corpos. Assim como no caso dos *risers*, uma classe agrupa todos os elementos deste tipo que podem fazer parte da cena. Esta classe é a **RV3DSModels**.

Outro aspecto ao qual se deve atentar nesses diagramas é o uso da generalização no caso dos elementos da cena e das coleções de elementos, que derivam, respectivamente, das classes **RVObject** e **RVObjectList**.

4.4.4. O Laço Principal

No capítulo 2 discutiu-se que animações que podem ser manipuladas dinamicamente exigem um tratamento especial no que se refere ao laço principal do programa. Tipicamente em aplicações que seguem o paradigma de janelas e ponteiros, este laço se inicia com o início da aplicação e roda até que ela se encerre, aguardando por entradas do usuário, capturando, interpretando e traduzindo estas mensagens em ações do programa.

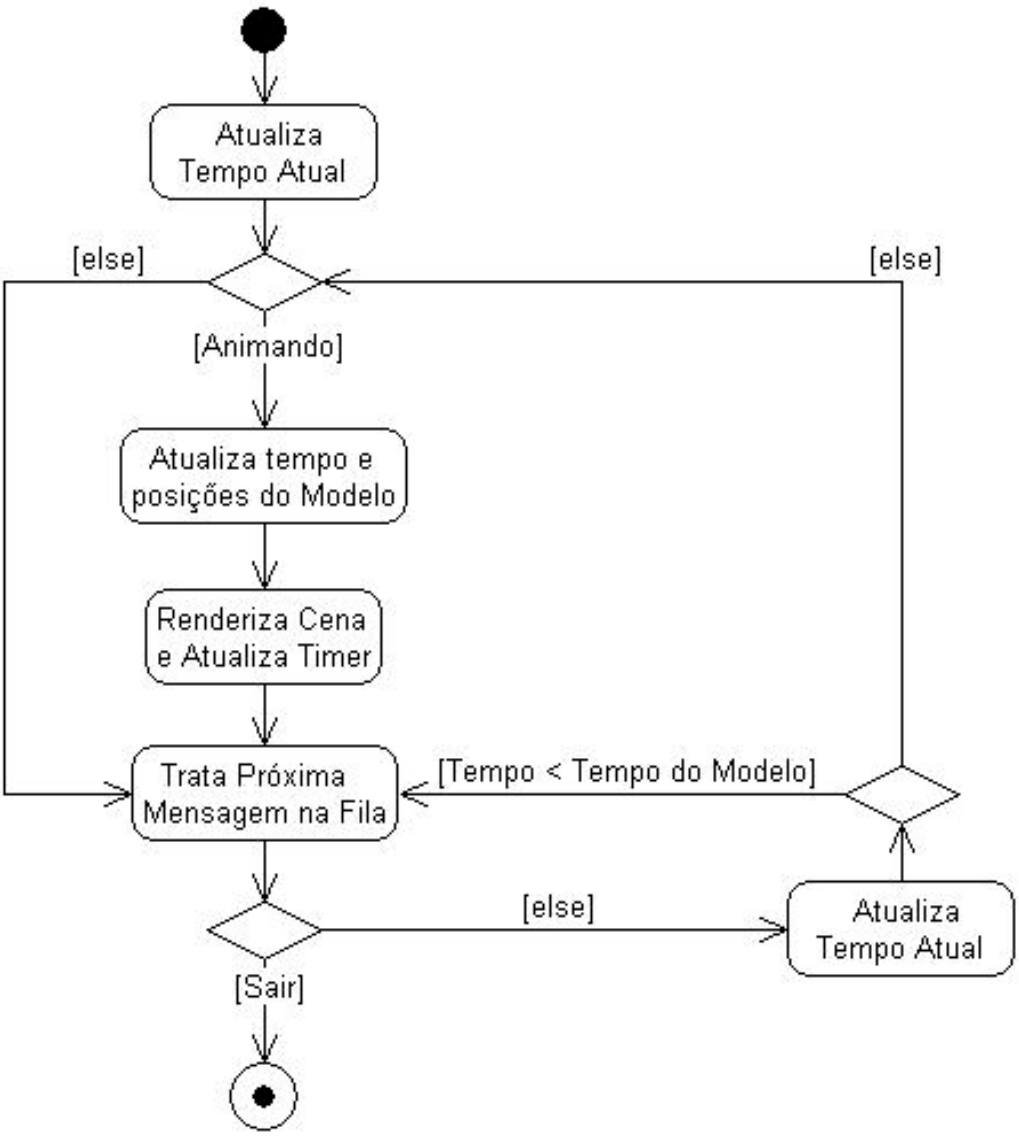

Figura 25 - Artefato: diagrama de atividades do laço principal

Durante uma animação, no entanto, é necessário que exista um outro laço que também é executado ao longo de toda a duração da animação, atualizando a posição de seus objetos em cada instante de tempo.

O modo como estes dois laços coexistem no **RiserView** é ilustrado na figura 25 através de um diagrama de atividades, ou um fluxograma.

Deve-se notar, neste diagrama, que a prioridade é dada a animação, e não ao tratamento das mensagens do programa. Primeiramente, quando a animação está ativada, a cena é atualizada e renderizada e só então, no tempo que sobra, as mensagens são tratadas. Esta animação ocorre em tempo real, ou seja, o tempo do Modelo avança no mesmo ritmo que o tempo real, a não ser que a atualização da cena não consiga ser feita no período de tempo entre dois quadros da animação. Nesse caso, a animação ocorre com atraso.

O diagrama da figura 26 ilustra, através de um diagrama de seqüência, um passo deste laço, mostrando como as mensagens de atualização se propagam entre o Controle, a Vista e o Modelo, bem como os elementos que o compõem, de acordo com o padrão MVC. Cada elemento, ao ser modificado, cuida de se marcar como tal, para que a Vista, ao renderizar a cena, saiba quais partes da *pipeline* de visualização devem ser re-executadas. A função **IupLoopStep** é a que trata a próxima mensagem na fila. Deve-se notar que após este passo, é preciso renderizar a cena novamente.

No diagrama de seqüência, os objetos envolvidos numa operação são representados como os retângulos no topo do diagrama, com os nomes e tipos do objeto, organizados da esquerda para a direita em ordem de subordinação. As mensagens trocadas entre estes objetos são representadas pelas setas entre eles. As mensagens são organizadas de cima para baixo em ordem cronológica. A seta tracejada representa um retorno de função. As linhas tracejadas verticais são chamadas de *lifelines* e representam a existência do objeto (por exemplo, se uma mensagem de destruição fosse mandada para um objeto, a linha tracejada seria interrompida naquele ponto) e os retângulos finos

verticais representam períodos em que os objetos estão ativos realizando atividades relacionadas com a operação que o diagrama ilustra.

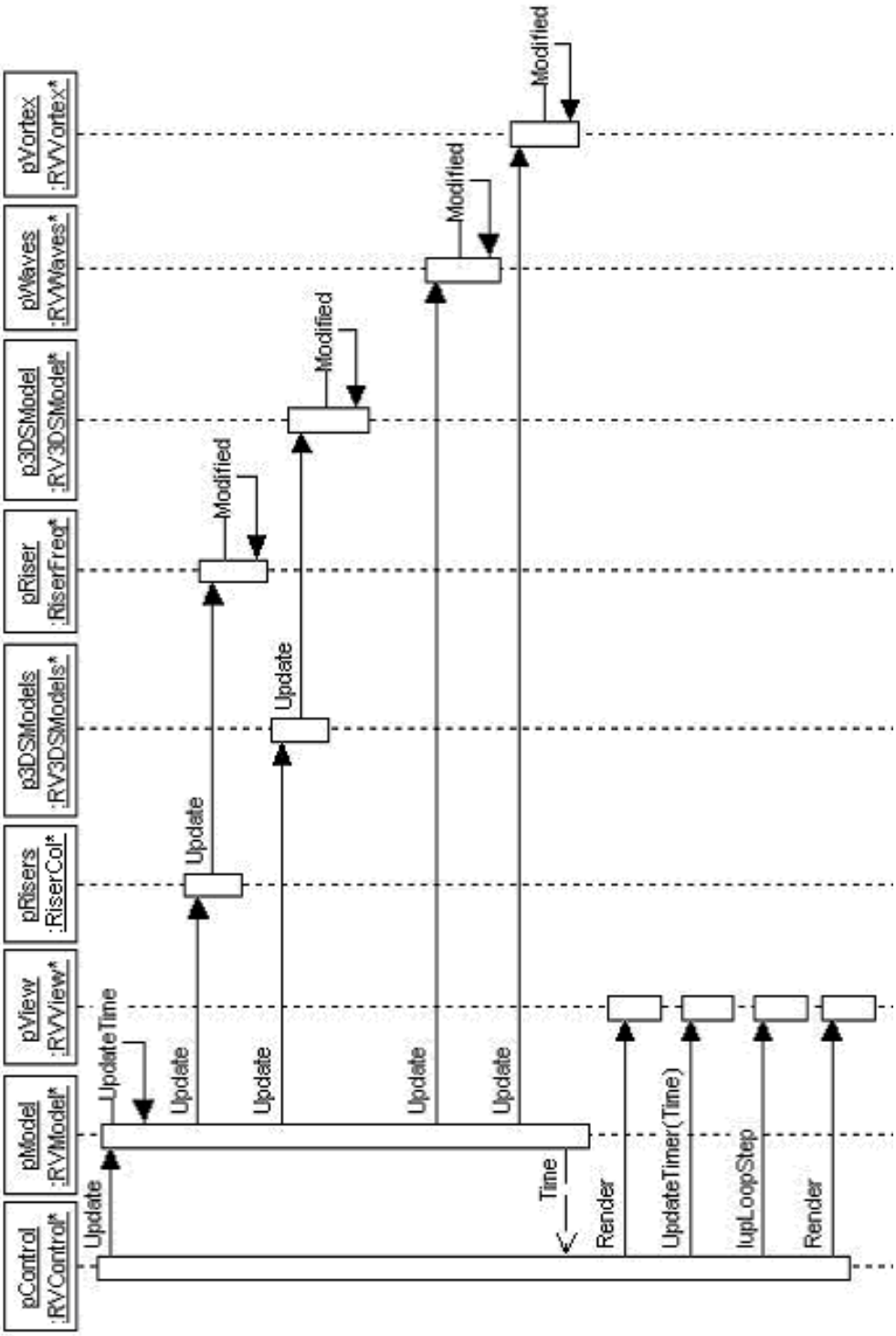

Figura 26 - Artefato: diagrama de seqüência do laço principal

Em particular no diagrama mostrado na figura 26, todas as mensagens são do tipo síncrono, ou seja, chamadas de funções.

4.4.5. Detecção de Colisão

A detecção de colisões entre *risers*, nesse aplicativo, na verdade é feita entre os segmentos que compõem os *risers*, e nunca entre segmentos do mesmo *riser*. No capítulo 2 são citadas algumas características particulares que simplificam problema de detecção de colisões no **RiserView**. Uma delas é a de que o comportamento de cada *riser* já é conhecido antes da animação.

O modo como esta característica simplifica o problema é o seguinte: como a amplitude máxima dos deslocamentos de cada segmento já é conhecida, essa amplitude pode ser usada para criar uma envoltória ao redor dos segmentos (e do *riser*) que pode ser usada com um volume limitante para uma primeira aproximação da detecção de colisões. Dessa forma, é possível, através de um pré-processamento realizado no momento em que cada *riser* é adicionado à cena, determinar, para cada segmento do novo *riser*, com quais segmentos dos *risers* já presentes na cena existe a possibilidade de haver colisão. Esse pré-processamento utiliza o mesmo algoritmo para detecção de colisão entre cápsulas usado para determinar a colisão exata entre os *risers*, basta somar ao raio de cada segmento a amplitude de seu movimento. Sua envoltória tem também o formato de uma cápsula. Para cada *riser*, os resultados do pré-processamento são armazenados na estrutura de dados mostrada na figura 27.

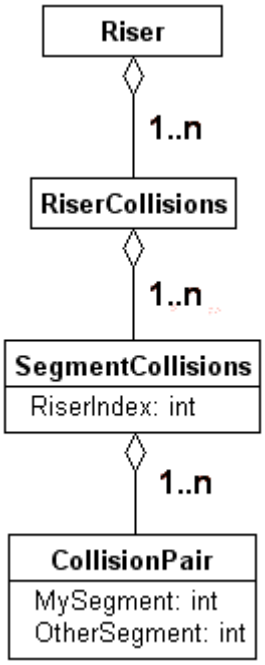

Figura 27 - Artefato: diagrama de classes dos pares de colisão

A figura mostra como cada *riser* armazena uma lista de com quais outros *risers* ele pode colidir através de uma instância da classe RiserCollisions. Essa classe é uma agregação de objetos SegmentCollision que, além de guardar o índice do *riser* com o qual a colisão ocorre, guardam uma lista dos pares de segmentos que podem colidir (os pares de colisão), representados por objetos da classe CollisionPair.

Dessa forma, em cada passo da animação, somente esses pares de colisão (para os quais existe alguma possibilidade de colisão de acordo com o pré-processamento) precisam ser checados. Essa checagem é feita utilizando o algoritmo descrito por Eberly (2000a) e abordado em 2.8. A classe que gerencia essas checagens de colisão entre os *risers* é a classe **RiserCol** através da função **CheckCollisions**.

4.5. Implementação e Testes

As fontes comentadas e o executável do aplicativo, bem como um manual do usuário em HTML, encontram-se no CD anexo a este trabalho. Estes três são os principais artefatos gerados durante a etapa de implementação

Já na etapa de testes foram realizados testes para verificar cada um dos casos de uso e dos requerimentos especiais do projeto, assim como testes de *stress* com grandes números de polígonos para verificar o comportamento do programa. Os arquivos utilizados nestes testes também estão no CD anexo e são os únicos artefatos gerados nesta etapa. Os resultados dos testes serão discutidos no capítulo seguinte. Uma lista de defeitos (ou *bugs*) encontrados também é gerada e utilizada durante o procedimento dos testes, mas estes defeitos vão sendo corrigidos a medida que são encontrados portanto esta lista é um artefato de cunho temporário, que embora tenha sido utilizado não fica registrado neste trabalho.

Com esta seção encerra-se o capítulo de metodologia. Acredita-se que não só foi possível demonstrar como o Processo Unificado foi aplicado como metodologia deste projeto, mas também que o conjunto de artefatos gerados durante este processo (glossário; descrição dos requerimentos, atores e casos de uso; listagem dos requerimentos especiais; esboço da interface; diagramas de casos de uso, subsistemas, classes, atividades e seqüência; fontes comentadas; programa executável; manual do usuário e conjunto de arquivos de teste) contribuam para uma compreensão facilitada do **RiserView** bem como constituam uma documentação completa do projeto. No próximo capítulo são discutidos os resultados obtidos nos testes do programa.

5. RESULTADOS

Neste capítulo são apresentados, principalmente, os resultados dos testes realizados sobre o **RiserView**. Uma discussão mais aprofundada desses resultados e de eventuais soluções para os problemas encontrados será feita somente no capítulo 6. No entanto, onde for necessário, serão discutidos aqui alguns pontos para justificar a forma como os testes foram feitos.

Todos os testes foram realizados num sistema com a seguinte configuração:

- Processador Pentium 4 de 1,7GHz
- Memória RAM de 512MB
- Placa Gráfica GeForce FX 5200 com 128MB de memória
- Capacidade Nominal de Processamento de $6 \cdot 10^7$ vértices/segundo
- AGP-8X (a placa mãe, no entanto, é somente AGP-4X)
- Driver versão 61.77
- Sistema Operacional: Windows XP Pro Versão 2002 com o Service Pack 2

Além disso, durante os testes o **RiserView** era o único aplicativo (exceto pelos processos do sistema operacional) sendo executado no sistema, que estava desconectado de qualquer tipo de rede e com antivírus e *firewall* desativados.

Este capítulo está estruturado da seguinte forma: Em primeiro lugar são discutidos brevemente os testes de "caixa-preta". A próxima seção relata como foram realizados os testes de *stress* do aplicativo e os resultados obtidos. Em seguida o sistema de detecção de colisões entre *risers* é analisado. Por fim, a portabilidade do código para o Linux é discutida.

Testes de usabilidade[16] para verificar a adequação da interface não foram realizados de maneira formal, por isso não são relatados aqui. Ainda assim, através da utilização do aplicativo por diferentes usuários pôde-se chegar a algumas conclusões interessantes quanto a esse aspecto, que serão discutidas no próximo capítulo.

5.1. Testes de Caixa-Preta

Durante esses testes, o objetivo é verificar se a funcionalidade do aplicativo equivale à sua especificação, sem uma preocupação com a implementação do *software*. Como discutido no capítulo anterior, os casos de uso obtidos para o **RiserView** são utilizados como guia para esses testes. Cada caso de uso, bem como cada variação de

[16] A efetividade, eficiência e satisfação com a qual os usuários podem realizar tarefas em um sistema. Alta usabilidade significa que o sistema é: simples de se usar e aprender, eficiente, visualmente agradável e rápido ao se recuperar de erros. (*fonte: Free Online Dictionary of Computing).*

um caso, é reproduzido utilizando o aplicativo e desvios em relação à especificação são registrados.

Esses testes foram realizados ao longo de todo o desenvolvimento. Muitos dos erros identificados foram corrigidos nesse processo, mas a identificação e correção desses erros não foi registrada. O que está registrado neste trabalho são os *bugs* que permaneceram na versão do aplicativo entregue como anexo.

A lista de *bugs* resultante, com as explicações necessárias para cada um, bem como uma lista de melhorias simples para a interface sugeridas por usuários do **RiserView**, está registrada no Apêndice C, meramente para preservar a continuidade do texto deste capítulo.

5.2. Testes de *Stress*

O objetivo deste conjunto de testes é determinar quais os limites do aplicativo. Para tanto, porém, é necessário separar e identificar que partes do sistema estão submetidas ao esforço imposto pelos testes, não só para conhecer os reais limites do aplicativo como também para permitir conclusões mais precisas sobre esses limites.

Três fatores, em particular, foram identificados antes da realização dos testes como possíveis limitantes no aplicativo:

i. Capacidade de processamento: com isso deseja-se especificar o uso do processador principal do sistema, e não o processador gráfico (que é tratado como um fator

distinto). Apesar do **RiserView** não realizar nenhuma forma de simulação ou análise física, uma vez que a dinâmica de todos os elementos da cena é pré-calculada por outros aplicativos, algumas das tarefas que realiza são custosas em termo de processamento. A principal destas tarefas é a determinação, em cada passo da animação, da posição e do comprimento de cada um dos segmentos cilíndricos que compõem os *risers*, com base nas posições de suas extremidades. As coordenadas das extremidades devem ser convertidas em dois ângulos que serão usados para rotacionar o segmento e a distância entre elas passa a ser seu comprimento. É necessário recalcular o comprimento uma vez que o aplicativo se utiliza da técnica de deformação da estrutura ao amplificar os movimentos dos *risers* para permitir sua visualização. Outra tarefa, ainda mais custosa (mas não tão necessária), é o *anti-aliasing* da cena feito por *software*. A detecção de colisões entre *risers* é uma terceira tarefa com custo para o processador, mas será discutida separadamente na próxima seção.

ii. Memória: inicialmente havia a preocupação de que a memória consumida pelas estruturas de dados relativamente complexas utilizadas pelo VTK e pela arquitetura do **RiserView** pudesse também ser um limitante de sua performance. Ainda que atualmente se possa contar com uma quantidade relativamente alta de memória virtual, o acesso a essa memória é muito mais lento e poderia impossibilitar a renderização de cenas em tempo real. Durante a realização de todos os testes, no entanto, pode-se constatar que muito antes de o aplicativo ocupar uma parcela significativa da memória disponível, ora o processamento, ora

o processamento gráfico alcançavam seu limite. Sendo assim, não foram realizados mais testes específicos para analisar o consumo de memória do **RiserView**.

iii. Processamento Gráfico: a preocupação ao tratar desse fator é diferenciar que parte do processamento é realizada pela placa gráfica e que parte é realizada pela CPU. A solução encontrada foi a renderização de uma cena bastante complexa (com uma quantidade de polígonos próxima do limite do sistema gráfico utilizado) mas que fosse resultado de poucos cálculos ou outras formas de manipulação de dados pelo processador principal. Esse cuidado ainda não permite, no entanto, determinar que parte do sistema gráfico atingiu seu limite. Para determinar, por exemplo, se o limitante durante os testes foi a comunicação de dados para a placa gráfica ou o processamento desses dados foi necessária uma análise mais complexa.

Tendo isso em mente, serão discutidos com mais detalhe os testes de *stress* tanto para a capacidade de processamento como para o processamento gráfico.

5.2.1. Capacidade de Processamento

Para testar a capacidade de processamento, foi decido utilizar cenas compostas por números cada vez maiores de *risers* com sua dinâmica descrita no domínio da freqüência. Isso porque cenas com um número de polígonos relativamente baixo formadas somente por esses *risers* são consideravelmente pesadas em termos de processamento. Para cada segmento, em cada passo de tempo, é preciso realizar as seguintes operações:

- Calcular a posição atual das extremidades (atualmente envolve o cálculo de 6 cosenos).

- Calcular a distância entre essas extremidades (envolve uma raiz quadrada) e modificar o comprimento do segmento para que se torne igual a essa distância.

- Calcular os dois ângulos necessários para rotacionar o segmento (envolve o cálculo de dois arco-senos, um coseno e uma checagem para corrigir o quadrante dos ângulos).

- Transladar e Rotacionar o segmento.

Somente o último item é responsabilidade do processador gráfico. E os cálculos de cosenos, arco-senos e raízes quadradas são feitos utilizando as funções da biblioteca padrão de matemática do C, que utiliza métodos iterativos para obter esses valores com alta precisão.

Por isso esses *risers* foram considerados os mais adequados para testar a capacidade de processamento. Foram gerados (automaticamente através de um programa que translada e rotaciona um *riser* original) 144 arquivos de *risers* descritos no domínio da freqüência, cada um com somente 11 segmentos. Para diferentes valores de passo de tempo da animação e de número de polígonos na cena (que podia variar conforme a discretização dos segmentos cilíndricos), esses *risers* foram sendo acrescentados à cena até que o **RiserView** não fosse mais capaz de calcular e renderizar os *risers* em tempo real, no passo de tempo pedido. O aplicativo fornece uma indicação visual quando não pode renderizar a cena em tempo real - o mostrador de tempo na

barra de ferramentas passa de azul para vermelho. Durante esses testes, *anti-aliasing*, estereoscopia e detecção de colisões estavam desativados.

Para um passo de tempo de 50ms, no sistema utilizado para os testes, foi possível atualizar e renderizar em tempo real somente 36 *risers* de 11 segmentos cada. Isso equivale a calcular quase 8 mil segmentos por segundo, com todas as operações mencionadas anteriormente.

Nessas condições, o processador gráfico trabalha bem abaixo de sua capacidade. Inicialmente, o teste foi realizado com somente 18 vértices por cilindro, o que resultava em aproximadamente 140 mil vértices/s (bem abaixo dos 60 milhões de capacidade da placa utilizada). Foi possível aumentar em 5 vezes o número de vértices por segmento (de 18 para 90) antes que fosse perceptível uma queda em performance, o que confirma o fato de que o limitante é o processamento numérico, não o gráfico.

Com um passo de tempo duas vezes maior, de 100ms, foi possível renderizar pouco mais que o dobro de *risers*, 76. A tabela 1 resume esses resultados:

Tabela 1 - Testes de *stress*: capacidade de processamento

Passo de Tempo (ms)	50	50	100
Número de Quadros por Segundo	20	20	10
Número de Risers	36	36	76
Número de Segmentos por *Riser*	11	11	11
Número de Vértices por Segmento	18	90	90
Número de Segmentos por Segundo	7.920	7.920	8.360
Número de Vértices por Segundo	1,43E+05	7,13E+05	7,52E+05

O *anti-aliasing* por *software*, mesmo com um número pequeno de *risers* (entre 1 e 5), só pode ser feito com um passo de tempo de 200ms.

5.2.2. Processamento Gráfico

Para renderizar uma cena com um grande número de polígonos mas que taxasse pouco o processador principal do sistema, decidiu-se utilizar um elemento estático - o solo do oceano - discretizado em um número cada vez maior de polígonos até que a cena não mais pudesse ser renderizada no passo de tempo desejado.

Durante a realização dos testes foi necessário ter o cuidado de exibir toda a cena na janela, para evitar que o processo de *culling* (discutido no Capítulo 2) eliminasse para o resto da *pipeline* os vértices que estivessem fora do campo de visão. No entanto, a variação de performance entre os testes com somente uma pequena parte da cena sendo exibida e aqueles em que ela era exibida em sua totalidade foi pequena, o que leva a crer que o fator limitante não foi o processamento da cena em si, mas sim a passagem de dados para o processador gráfico (que acontece antes do *culling*). Outra explicação, menos plausível, é um sistema de *culling* mau implementado.

Para analisar esse aspecto de transferência de dados, é necessário estimar o tamanho de um vértice em bytes. Cada vértice contém suas três coordenadas, três valores para codificar sua cor, um para sua transparência, três valores para parametrizar sua normal e dois para coordenadas de textura. Com uma precisão de 32 bits para esses valores, obtém-se um tamanho de 48 bytes para cada vértice, sem levar em conta a

transmissão de informação topográfica (como os vértices estão ligados para formar os polígonos).

Para um passo de tempo de 100ms, a maior cena que se pôde renderizar tinha aproximadamente um milhão e meio de vértices (o arquivo "stress.btm" contém a grande maioria desses vértices). Com 15 milhões de vértices por segundo, essa cena corresponde a somente 25% da capacidade nominal da placa. Isso equivale, no entanto, a aproximadamente 725MB/s, o que está bem mais próximo do topo de performance, de acordo com MSI (2004), para o modo de transferência AGP-4X, que é de até 1GB/s. O aplicativo também foi testado (menos formalmente) em outros sistemas, incluindo um sistema com AGP-8X, onde mostrou performance notavelmente superior. Esse sistema, no entanto, tinha também capacidade de processamento numérico e gráfico superiores ao descrito no início desse capítulo, de forma que a comparação dos resultados obtidos nos dois sistemas é problemática.

A cena foi renderizada com de um até oito *risers* descritos no domínio da freqüência e somente a partir daí pode-se perceber uma queda na performance. Durante esses testes, o *software* ocupava apenas aproximadamente 70MB da memória RAM.

A tabela 2 resume os resultados desses testes:

Tabela 2 - Testes de *stress*: processamento gráfico

Número de Vértices	1.500.000	1.500.000
Passo de Tempo (ms)	100	100
Número de Quadros por Segundo	10	10
Número de Risers	1	8
Número de Segmentos por Segundo	110	880
Número de Vértices por Segundo (c/ Risers)	1,50E+07	1,51E+07
Tamanho de um Vértice (Bytes)	48	48
Taxa de Transferência (Bytes/s)	7,24E+08	7,25E+08

5.3. Detecção de Colisões

Os testes do sistema detecção de colisões entre *risers* tinham como objetivo determinar três pontos principais:

i. Se as colisões de fato eram sempre detectadas, se falsas colisões eram detectadas e se, ao detectar as colisões, o **RiserView** se comportava conforme descrito no caso de uso 7 (ver Apêndice A).

ii. Qual o pior caso para o sistema, ou seja, a condição em que é feito o maior número de checagens de colisão entre os *risers*.

iii. No pior caso, qual a performance do aplicativo quando a detecção está ativa.

Durante os testes, todas as colisões foram detectadas. Foram detectadas também colisões que visualmente parecem falsas. Isso se deve ao fato de as colisões serem detectadas entre os segmentos cilíndricos que formam os *risers*, e não entre sua representação por polígonos (o que, como já foi discutido, não só é mais simples como também mais preciso). Colisões falsas de fato não ocorreram. O funcionamento do sistema correspondeu ao especificado pelos casos de uso.

Entre dois *risers*, o pior caso detectado foi quando não só cada *riser* estava incluso na envoltória do outro, mas também eram divididos em segmentos de maneira igual e estavam colocados em paralelo. Nessa configuração, o pré-processamento indica a possibilidade de colisão de cada segmento com o seu equivalente no outro *riser*, bem como com os segmentos imediatamente adjacentes a ele. Isso porque o comprimento dos segmentos era sempre bem maior que o seu deslocamento (subdividir os *risers* em segmentos ainda menores fazia muito pouca diferença na visualização e enquanto o comprimento dos *risers* é medido em centenas de metros, os deslocamentos são medidos em unidades). Assim, nessa configuração, são gerados pouco menos que três pares de colisão para cada segmento do *riser*. A figura 28 ilustra essa configuração. Na figura, o segmento em azul pode colidir com os três segmentos em vermelho. Em cinza está indicada a envoltória que inclui o segmento azul em qualquer posição que esse possa assumir.

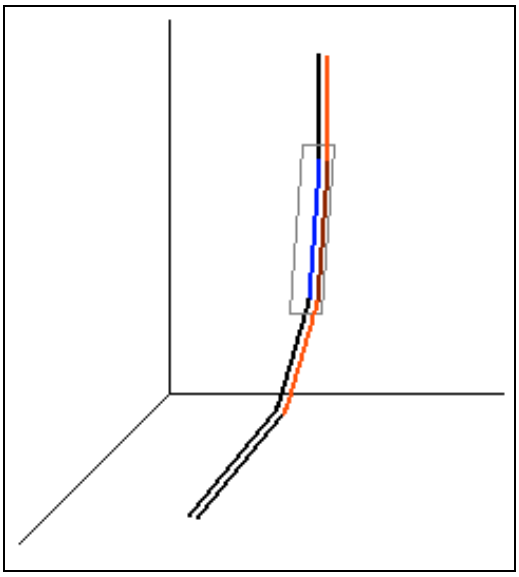

Figura 28 - Pares de colisão

Quando os segmentos têm o mesmo tamanho mas não estão paralelos entre si, muitas vezes só existe a possibilidade de colisão do segmento com dois outros ao invés de três. E quando têm tamanhos diferentes, muitas vezes um segmento menor só pode colidir com um dos segmentos maiores.

Determinar uma configuração espacial e de deslocamentos de mais de dois *risers* que gere o maior número de colisões, no entanto, é mais complexo. Para que o número de pares de colisão seja maximizado, é importante que os *risers* estejam paralelos. Nesse caso, com *risers* em catenária, o pior caso encontrado foi com *risers* lado a lado, com deslocamentos maiores que a espessura dos segmentos de forma que cada *riser* poderia colidir não só com os imediatamente adjacentes mas também com os próximos. Para *risers* verticais existem possibilidades mais complexas, como arranjos circulares concêntricos dos *risers*.

Por fim, para verificar a performance do aplicativo ao realizar a detecção de colisões, foi utilizado um passo de 50ms. *Risers* idênticos, exceto pela posição, foram acrescentados um ao lado do outro de forma que cada um só pudesse colidir com os dois imediatamente adjacentes, até que o aplicativo não fosse mais capaz de executar a animação e as checagens de colisão em tempo real. Nenhuma colisão era de fato detectada para que a animação nunca fosse interrompida. Para esses *risers* e configuração em particular, com 11 segmentos cada, foram obtidos 29 pares de colisão para cada par de risers.

Uma hipótese que teve que ser respeitada durante os testes foi a de que os deslocamentos de cada segmento em um passo de tempo sempre fossem menores que a menor dimensão de um segmento. Isso porque o sistema testa somente a colisão entre os segmentos, e não entre as trajetórias que percorrem, como discutido anteriormente. Caso essa hipótese não seja respeitada, podem ocorrer colisões entre segmentos que não são detectadas, pois eles se atravessam ao passar de um passo de tempo para o próximo mas nunca estão colidindo em um mesmo passo.

É importante notar que, nesse teste, é impossível isolar quanto do processamento está sendo usado para a detecção de colisões e quanto está sendo usado para a atualização das posições dos *risers*. É possível, no entanto, comparar esses resultados com os descritos em 5.2.1.

O número máximo de *risers* que foi possível acrescentar à cena foi de 24 *risers* (enquanto sem a checagem de colisão esse número, para o mesmo passo de tempo, foi de 36), o que significa a atualização da posição de 5280 segmentos por segundo e a checagem de mais de 13 mil colisões por segundo. A tabela 3 resume esses resultados:

Tabela 3 - Detecção de colisões

Passo de Tempo (ms)	50
Número de Quadros por Segundo	20
Número de Risers	22
Número de Segmentos por *Riser*	11
Pares de Colisão entre 2 Risers	29
Checagens de Colisão por Segundo	13.340
Número de Segmentos por Segundo	5.280

5.4. Portabilidade

Portar o código para o Linux foi um processo relativamente simples. Como já era esperado, as únicas mudanças necessárias foram na classe RVView, na função Initialize que, entre outras tarefas, estabelece a interface entre o VTK e o IUP. O tipo das variáveis que no MS Windows era HWND passa a ser Widget e o tipo da variável pRenWin deixa de ser vtkWin32OpenGLRenderWindow e passa a ser vtkXMesaRenderWindow. A instrução de #include para o cabeçalho da classe de *render window* teve que ser trocado da mesma forma.

Ao invés de gerar um *makefile* manualmente foi usada a ferramenta KDevelop para gerenciar o projeto, o que tornou a tarefa bastante simples.

A compilação tanto do IUP quanto do VTK também foi facilitada pelo uso das ferramentas recomendadas por seus desenvolvedores, respectivamente, TecMake e

CMake. A documentação do IUP, em particular, discute em detalhe como compilar a API no Linux e usá-la nesse sistema.

Exceto pelo *look-and-feel* nativo do Linux o aplicativo se comportou da mesma forma que no MS Windows. Os testes de *Stress* não foram repetidos nesse sistema.

6. CONCLUSÕES

Antes mesmo de se discutir pontos específicos levantados por esse trabalho, pode-se afirmar que a principal conclusão a que se pôde chegar foi a de que ele se trata apenas dos primeiros passos na pesquisa e no desenvolvimento de um ambiente de visualização como este.

Acredita-se que, na forma em que se encontra, o **RiserView** já é uma ferramenta útil para auxiliar na pesquisa da dinâmica de *risers*, permitindo que outros pesquisadores possam se concentrar na modelagem dos sistemas envolvidos e em sua simulação mas que ainda assim possam contar com uma visualização que facilite a compreensão dos resultados que obtém.

A pesquisa realizada para o desenvolvimento deste ambiente, por si só, pode ser considerada de interesse para a área, principalmente pelo seu caráter multi-disciplinar. No início do trabalho o autor estava familiarizado com aspectos da modelagem e simulação da dinâmica de *risers* e com programação orientada a objetos, principalmente para a plataforma Windows e sem uma preocupação com uma metodologia de projeto. A esses conhecimentos foi possível, de fato necessário, acrescentar conceitos sobre vórtices, incluindo métodos para sua simulação, engenharia de *software*, computação gráfica, realidade virtual e visualização científica, dentre outros. Durante a elaboração

deste texto houve a preocupação de, devido justamente a esse caráter multi-disciplinar, explicar de forma clara, ainda que por vezes necessariamente breve, os conceitos básicos das áreas envolvidos no trabalho - explicações que possivelmente seriam desnecessárias em um trabalho de escopo mais estreito. Espera-se que com isso seja possível facilitar e até estimular futuras pesquisas nessa linha.

Apesar disso tudo, ficou claro durante a redação desse texto, o uso do aplicativo e a realização dos testes que restam ainda diversas melhorias importantes a serem feitas no trabalho. Até mesmo durante o desenvolvimento, diversas vezes foram levantados possíveis aperfeiçoamentos e nem todos foram implementados ou pesquisados devido a restrições de tempo. Embora muitas dessas mudanças sejam relativamente simples, como as listadas no Apêndice C, nesse capítulo serão discutidas algumas de caráter mais profundo e menos imediato. É por isso, também, que se afirma que esse trabalho é apenas um primeiro passo.

Isto posto, são tratados em seguida alguns pontos específicos. Para cada ponto, quando conveniente, discute-se possíveis futuros trabalhos a ele relacionados. Ao contrário dos outros capítulos, essa discussão apresenta diversas conjecturas sem a preocupação de demonstrá-las, pois embora seja plausível extrapolar os efeitos que esses futuros trabalhos possam ter, ou mesmo a sua necessidade, com base nos conhecimentos atuais, é impossível oferecer provas para essas conjecturas antes que os trabalhos sejam de fato realizados.

Em primeiro lugar são tratadas questões relativas à performance do *software*, incluindo a detecção de colisão. Diversas questões referentes à visualização vêm então e nesse item algumas questões relativas à interface também são discutidas. Por fim o uso

das principais tecnologias no desenvolvimento (as APIs IUP e VTK e a própria metodologia, o Processo Unificado), são abordadas.

6.1. Performance

Os principais tópicos que merecem discussão com referência à performance são a atualização dos *risers*, *anti-aliasing*, a renderização de cenas complexas, detecção de colisões entre *risers* e, por fim, a possibilidade de paralelizar o código.

6.1.1. Atualização dos *Risers*

Uma das principais conclusões dos testes de *stress* foi a confirmação de como a atualização dos *risers*, principalmente quando descritos no domínio da freqüência, é custosa em termos de processamento. A 20 quadros por segundo, um valor relativamente baixo para uma animação, só foi possível renderizar 36 desses *risers* no sistema utilizado para os testes, e com somente 11 segmentos cada. Tratam-se de números pequenos, principalmente se for levado em conta o fato de que foram obtidos sem a inclusão de nenhum outro elemento na cena e sem detecção de colisões. Comparando o número de vértices utilizados nesses testes com os limites do *hardware*, ou mesmo com os testes relatados em 5.2.2, pode-se concluir que a operação crítica é, de fato, a atualização da posição dos *risers*, e não sua renderização. E embora seja possível lançar mão de sistemas com maior capacidade de processamento e essa

capacidade aumente rapidamente com o tempo, conclui-se ser necessário reduzir o custo dessa tarefa, mesmo porque podem ser encontrados outros usos para o processamento disponível (como alguns mencionados nesse capítulo). Além disso, como é discutido a seguir, essa redução pode ser atingida de maneiras relativamente simples.

Tanto os *risers* descritos no domínio da freqüência quanto os descritos no domínio do tempo têm a necessidade de utilizar funções trigonométricas, para converter a descrição das coordenadas das extremidades de seus segmentos de um sistema cartesiano para um sistema polar, permitindo a definição das rotações que o segmento deve sofrer bem como de seu comprimento. Para os *risers* descritos no domínio da freqüência, no entanto, esse problema é agravado, uma vez que cada uma das três coordenadas das extremidades de seus segmentos é definida a partir do cálculo de um coseno. Converter uma descrição no domínio da freqüência para o domínio do tempo é um processo trivial e, por isso, poderia ser levantada a hipótese de se abandonar, no **RiserView**, o tratamento do domínio da freqüência, convertendo qualquer *riser* para uma descrição no domínio do tempo e utilizando somente essa descrição durante a animação (essa foi a solução utilizada, por exemplo, para descrever a dinâmica da superfície da água). Essa modificação, no entanto, não é interessante pelos seguintes motivos.

A descrição de *risers* no domínio da freqüência é válida para literalmente qualquer instante no tempo e portanto independente do passo utilizado, podendo ser utilizada com facilidade em sistemas com passo de tempo variável. Descrições no domínio do tempo são válidas para um único passo de tempo. Além disso, descrições no domínio do tempo exigem o uso de mais memória para armazenar cada *riser*, bem como

de arquivos maiores e mais lentos para serem lidos. Determinar a envoltória utilizada para detecção de colisão também se torna mais custoso nesse domínio. Como aprimoramentos simples (discutidos adiante) podem reduzir drasticamente o *overhead* na atualização de *risers* no domínio da freqüência, as vantagens dessa descrição tornam pouco atraente a opção de eliminá-la.

Uma dessas otimizações simples para o cálculo da posição de *risers* no domínio da freqüência está em separar o cálculo de cada parcela do coseno na determinação das coordenadas de seus pontos. Cada coordenada para um ponto i é calculada a partir de uma equação semelhante a:

$$x_i = A_i \cos(\omega.t + \varphi_i)$$

Como a fase φ_i não varia com o tempo, mas somente com cada ponto, poderiam ser armazenados nas estruturas de dados os valores de seno e coseno dessa fase, ao invés de seu valor, e utilizar a seguinte equação para o cálculo das coordenadas:

$$x_i = A_i\left[\cos(\omega.t)\cos(\varphi_i) - \text{sen}(\omega.t)\,\text{sen}(\varphi_i)\right]$$

E dessa forma seria necessário calcular o coseno e seno de ωt uma única vez para cada *riser* em um passo de tempo, ao invés de calcular um coseno para cada um de seus segmentos, uma vez que as parcelas relativas à fase estariam pré-calculadas e armazenadas.

Embora essa mudança melhore o desempenho da atualização da posição dos *risers*, mesmo com ela ainda restam os cálculos de dois arco-senos, um coseno e uma raiz quadrada para determinar as rotações e o comprimento de cada segmento,

independente de como o *riser* é descrito. Uma solução relativamente simples para reduzir o custo desses cálculos é gerar tabelas para essas duas funções trigonométricas e usar a interpolação dessas tabelas ao invés das funções padrão do C. Como essas funções estão sendo usadas para determinar posições para serem renderizadas (com precisão limitada pela definição do dispositivo de saída utilizado), e como o cálculo não é iterativo, não havendo acúmulo de erros de um passo de tempo para o próximo, não há a necessidade da precisão que é fornecida pelas funções do C. Dependendo da precisão das tabelas utilizadas, uma simples interpolação linear poderia ser suficiente. Determinar uma combinação de precisão das tabelas e de ordem de sua interpolação e implementar essa mudança é outra possibilidade interessante para futuros trabalhos.

Ainda outra alternativa que inclusive remove a necessidade de calcular raízes quadradas durante a animação, mas que se aplica somente para o domínio do tempo e mais uma vez aumenta o consumo de memória é pré-calcular a posição e o tamanho de cada segmento para cada passo de tempo (por exemplo, quando os arquivos são carregados) e armazenar esses valores. Não é possível substituir as coordenadas das extremidades dos segmentos pela sua posição e tamanho, pois essas coordenadas são usadas durante a detecção de colisão. Todas essas informações precisariam ser armazenadas.

6.1.2. Anti-Aliasing

Outra tarefa, ainda mais custosa que a atualização da posição dos *risers*, é a de *anti-aliasing*. Muitas das placas gráficas atuais tem a opção de *anti-aliasing* por *hardware*. No início dos testes, no entanto, os efeitos desse recurso eram imperceptíveis (ao menos para placas da NVidia, que foram as únicas testadas). Com os novos *drivers* para essas placas, no entanto, é possível perceber uma melhoria significativa na qualidade da imagem ao se ativar esse recurso. Mas além de ele não estar presente em todas as placas, sua ativação ou não de maneira independente do aplicativo deve ser feita antes que o aplicativo em questão seja executado e mudanças em tempo de execução não têm efeito. Por esses motivos foi decidido manter a opção de *anti-aliasing* por *software* presente no **RiserView**. Mesmo com seu alto custo, pelo menos em relação à capacidade dos sistemas atuais, o *anti-aliasing* do **RiserView** mostrou-se útil em diversos momentos, como para visualização de cenas simples com um passo de tempo maior (como 200ms), ou quando não havia preocupação com a exigência de sincronia da animação com o tempo real, ou mesmo para obter capturas de tela de cenas com a animação parada.

Em placas gráficas com o recurso de *anti-aliasing*, cada aplicativo pode ativar ou não esse recurso através de comandos do OpenGL, por exemplo. Uma melhoria interessante para o **RiserView** seria estudar como determinar se um sistema gráfico tem ou não essa capacidade e utilizá-la, ao invés do *anti-aliasing* por *software*, quando o usuário ativa essa opção no aplicativo. O *anti-aliasing* por *software* seria mantido para os casos em que o *hardware* não possuísse esse recurso. É interessante notar que, em

sua versão atual (4.2), o VTK, apesar de através do OpenGL aproveitar bem os recursos das placas gráficas, não o faz no caso do *anti-aliasing*. Se com futuras atualizações do VTK ou do OpenGL isso mudar, essa preocupação pode deixar de ser importante para o **RiserView**.

Uma outra alternativa interessante é estudar o uso de *shaders* com o VTK. O uso desse recurso para obter mais realismo será discutido mais adiante, mas *Pixel Shaders*, em particular, podem ser usados para implementar *anti-aliasing*. Como pode ser verificado em Kessenich et al. (2004), a GLSL possui funções para estimar as derivadas parciais de propriedades dos *pixels* (HLDL e Cg também contam com esse recurso) e com as derivadas da cor é possível implementar um algoritmo simples e eficiente de *anti-aliasing*. Essa opção se torna ainda mais atrativa se *shaders* forem também usados para realizar outras funções no aplicativo. É importante lembrar, no entanto, que pelo menos atualmente esse ainda não é um recurso comum.

6.1.3. Cenas Complexas

O limite de 15 milhões de vértices por segundo que foi atingido durante os testes, bem abaixo da capacidade nominal do processador gráfico, a princípio não parece um ponto preocupante: esse número de vértices é suficiente para gerar cenas complexas mesmo com passos de tempo pequenos (como por exemplo 25ms - nesse passo de tempo seria possível renderizar cenas com aproximadamente 200 mil triângulos). E como os testes nos levam a concluir que o fator limitante é a taxa de

comunicação, o uso de AGP-8X mais do que dobra esse número. Caso a comunicação de dados para o processador gráfico se torne um problema no futuro, entretanto, existem algumas alternativas para contorná-lo.

A *pipeline* de visualização do VTK está otimizada, no **RiserView**, para renderizar cenas com grande número de polígonos que se modificam em cada quadro, pois julga-se que, numa cena típica, a maior parte de seus elementos será dinâmica. Caso cenas muito complexas sejam utilizadas, mas que contenham um grande número de vértices que não mudam de um quadro para o outro, essa parametrização da *pipeline* pode ser mudada para permitir que mais vértices sejam guardados na memória da placa gráfica e não precisem ser comunicados a cada quadro.

Outro recurso que não está presente no **RiserView** mas que seria um acréscimo interessante - e que é indispensável para a renderização de cenas de alta complexidade - é o tratamento de níveis de detalhe (LOD, ou l*evel of detail*). Trata-se de um recurso que ajusta a complexidade de cada objeto da cena (ou até de partes de objetos) à distância entre esse objeto e o observador, de forma que objetos mais distantes sejam apresentados com um menor número de polígonos, já que não podem ser vistos em detalhe daquela distância. Reduzindo o nível de detalhe de objetos distantes é reduzido também o número de vértices que é preciso comunicar ao processador gráfico.

6.1.4. Detecção de Colisões

Quanto à detecção de colisões entre *risers*, a principal conclusão que pôde ser atingida ao longo do trabalho é o quanto as simplificações que foram utilizadas, com base no conhecimento dos objetos sujeitos à colisão, reduziram o custo de uma tarefa geralmente cara sem a necessidade de algoritmos complexos. A possibilidade de montar os pares de colisão num pré-processamento, e não durante a animação, que se deve ao conhecimento prévio que se tem da dinâmica de cada *riser*, elimina a necessidade de usar outros filtros para detecção (como por exemplo setores ou *bounding volumes*). Por se saber a geometria do *riser* é possível fazer a detecção exata com cilindros e não entre polígonos, o que é outro fator que reduz o número de checagens necessárias, e é possível também decisão de utilizar cápsulas ao invés de troncos de cilindro na detecção, o que permite que se use um algoritmo com custo computacional bem menor. Por fim, o conhecimento da coerência espacial e temporal do sistema permite que não seja necessário checar a colisão entre as trajetórias percorridas pelos segmentos em um passo de tempo.

Por tudo isso, o autor não é capaz de levantar possibilidades de trabalhos futuros para aprimorar esse recurso do **RiserView**. Ainda assim, durante os testes houve uma queda no número de *risers* que podiam ser atualizados em tempo real de quase 35% ao se ativar a detecção de colisão. Deve-se ter em mente, porém, que esses testes foram realizados para o pior caso de colisão, com o maior número de pares de colisão possível e resultando em mais de 13 mil checagens de colisão por segundo além da cara atualização dos *risers*. Essa condição dificilmente será reproduzida durante o uso

normal do aplicativo para simular situações reais (de outra forma, ocorreria um grande número de colisões entre *risers* a todo instante na realidade). E ao se comparar mesmo esse pior caso para o algoritmo utilizado com a detecção de colisão por "força bruta" (ou seja, checar a colisão de cada objeto com todos os outros) o algoritmo implementado se mostra muito superior. Enquanto no pior caso, para dois *risers* com n elementos, o **RiserView** realiza no máximo 3n checagens de colisão a cada quadro, por força bruta seria necessário fazer n^2 checagens. O custo dessa performance é o pré-processamento necessário para montar os pares de colisão. Mesmo para grandes números de *risers* (da ordem de 100), no entanto, esse tempo de pré-processamento foi praticamente imperceptível durante os testes.

6.1.5. Paralelismo

Uma das vantagens do VTK citadas no capítulo 3 foi a facilidade da sua utilização (na verdade do ParaView, uma API derivada do VTK) para tirar vantagem de processamento em paralelo. De fato, sabendo do custo de atualização dos *risers* desde o início, uma futura paralelização do código, ao menos para essa tarefa, parecia atraente. Até mesmo o balanço de carga entre processadores poderia ser estimado, a princípio, pelo número de segmentos passado para cada um. Com base nos resultados obtidos, entretanto, acredita-se que, implementando alguns dos aprimoramentos mais simples sugeridos em 6.1.1 o **RiserView** seja capaz de lidar com cenas complexas e com grande número de *risers* sem a necessidade do multiprocessamento. Mesmo o *anti-aliasing* pode ser utilizado através da sua implementação por *hardware* com placas gráficas mais

avançadas e esse e outros aprimoramentos para aumentar o realismo da cena podem ser implementados com o uso de *shaders*. Essas opções permitem que o **RiserView** seja utilizado em um número maior de sistemas (placas gráficas programáveis são bem mais comuns que multiprocessamento) e, sendo assim, a paralelização do código passa a ser um último recurso, provavelmente só necessário se forem implementadas alguns dos recursos mais avançados para aumentar o realismo ou algumas das visualizações mais complexas citadas adiante.

6.2. Visualização e Interface

Os primeiros tópicos a serem discutidos nessa seção, que já foram inclusive citados anteriormente, são relativos ao passo de tempo da animação. Em seguida, a questão do realismo é discutida. Novas formas de visualização, e a visualização científica de outros objetos também são abordados. Por fim, discute-se uma conclusão interessante relativa à navegação pela cena, que foi atingida apesar de não terem sido realizados testes formais de usabilidade do sistema. A própria realização desses testes constitui uma possibilidade interessante de um futuro trabalho.

6.2.1. Passo de Tempo

Nessa primeira versão do **RiserView**, o passo de tempo da animação é fixo. Além disso, o usuário não tem controle sobre ele. O passo é determinado pelos

elementos da animação. Cada um (exceto o relevo do solo, que é estático, e os *risers* descritos no domínio da freqüência) tem um passo de tempo associado pois são descritos através de seqüências de posições em pontos discretos no tempo. Idealmente, as simulações de todos os elementos que compõem uma cena são feitas de forma compatível e geram arquivos descrevendo esses elementos com os mesmos passos de tempo. Essa condição era, inclusive, necessária nas versões iniciais do **RiserView** durante o desenvolvimento. Na versão atual, os passos de tempo podem ser diferentes e o menor deles é adotado, mas não há nenhuma interpolação de posição dos elementos, de forma que em cenas compostas por elementos descritos com passos de tempo distintos, ocorrem discrepâncias na posição relativa dos elementos em alguns quadros da animação.

Permitir que o usuário determine o passo é trivial. No entanto, caso isso seja implementado sem a interpolação de posições, tem pouca utilidade e torna-se mais um fator capaz de gerar discrepâncias na visualização. Cada elemento continuará a ser atualizado no seu próprio passo (a menos que o usuário escolha um passo de tempo maior que o que descreve o elemento).

Além de permitir que o usuário escolha um passo para a animação e mitigar as discrepâncias que ocorrem entre elementos cuja dinâmica é descrita com passos diferentes, a interpolação das posições dos elementos entre duas posições estabelecidas pela descrição de sua dinâmica permite que duas grandezas que são de fato distintas sejam assim tratadas pelo **RiserView**: o passo de tempo da animação e o passo (ou passos) de tempo que descreve a dinâmica dos elementos da cena. Atualmente essas grandezas estão vinculadas, mas seria interessante que fossem independentes.

Principalmente porque passos de tempo que podem ser pequenos o suficiente para algumas simulações e para descrever a dinâmica de objetos (como 150ms ou 200ms) são grandes demais para uma animação suave. Assim sendo, a interpolação de posições é um possível futuro trabalho relativamente simples, mas capaz de promover uma melhoria significativa no aplicativo.

Um trabalho mais complexo, mas que vale a pena ser mencionado, é a implementação de uma opção de passo de tempo adaptativo. Atualmente, conforme descrito no capítulo 4, o laço principal do programa dá máxima prioridade à atualização e renderização da cena em tempo real, usando um passo de tempo fixo, e o tratamento de mensagens de interface é feito apenas com o tempo que sobra (mas pelo menos uma mensagem é tratada em cada passo de tempo). Embora isso garanta uma atualização mais precisa da cena, para cenas complexas (seja devido à atualização dos *risers* ou ao número de polígonos) a navegação pela cena fica difícil, "saltada" e desagradável. Por isso, determinar um meio termo entre priorizar a atualização da cena ou o tratamento de mensagens é um trabalho interessante. Um modo de se implementar esse meio termo seria com um passo de tempo adaptativo para a animação que tente sempre manter sua qualidade mas também tratar as mensagens de usuário. Esse recurso depende, é claro, da implementação da interpolação de posições.

6.2.2. Realismo

Nos capítulos anteriores foi deixado clara uma opção de projeto deste trabalho: embora esses dois fatores nem sempre estejam em conflito, sempre que o realismo prejudicasse a visualização científica dos elementos da cena, a visualização seria privilegiada. Um exemplo é o mapeamento de escalares com cores sobre os *risers* ao invés do uso de texturas e *bump-mapping*. Essa opção está de acordo com os principais objetivos do trabalho e não se mostrou inadequada. O realismo, no entanto, também facilita a assimilação da cena e, em pontos em que não tem efeito sobre a visualização, poderia ser bastante aprimorado no **RiserView**.

Algumas desses possíveis aprimoramentos são: a inclusão de sombras, de um céu, inclusive com nuvens, e de horizontes, tanto para o céu quanto para a superfície da água (quando o observador está submerso); a renderização de uma superfície da água mais realista, inclusive com efeitos de espuma; efeitos de *bump-mapping*, principalmente para o solo; o teste de modelos de iluminação mais complexos; dentre outros. O uso de *shaders* pode facilitar, ou até viabilizar, a implementação de alguns desses recursos.

Outra possibilidade a ser estudada nessa linha, em trabalhos futuros, é a do uso de técnicas de realidade aumentada (a combinação de imagens renderizadas e reais) para aumentar o realismo da cena. Esse recurso poderia ser utilizado, por exemplo, para inclusão de estruturas, como plataformas, na cena.

6.2.3. Visualização

A visualização de vórtices da forma como está implementada atualmente no **RiserView** está intrinsecamente ligada a duas formas de simulação numérica específicas: métodos de vórtices discretos (que levam ao uso de glifos para representá-los) e métodos de elementos finitos ou diferenças finitas em malhas estruturadas ou não (que podem ser representados aqui somente através dos campos escalares). Além disso, as estruturas de dados prevêem somente essas simulações em planos e não em três dimensões, apesar de haver a possibilidade de utilizar vários desses planos numa mesma cena para estudar efeitos tridimensionais. Mesmo com essas limitações, ainda é possível fornecer, em futuros trabalhos, outras formas de visualização para o escoamento, como linhas de corrente e glifos com tamanho e direção variáveis (como setas) para representar grandezas vetoriais.

Mais interessante, no entanto, é a renderização de vórtices como estruturas tridimensionais através, por exemplo, do uso de iso-superfícies como sugerem Jeong & Hussain (1995). O próprio VTK conta com mais de um algoritmo para a tesselação (a geração de superfícies através da justaposição de polígonos) de iso-superfícies baseada em conjuntos volumétricos de dados, de forma que o maior desafio técnico desse trabalho se encontra na geração desses dados e na escolha das variáveis e valores usados na geração das superfícies, e não tanto nos algoritmos para a visualização em si. A animação e geração em tempo real dessas superfícies (ainda que com base em dados pré-calculados), pode ser uma tarefa suficientemente pesada para justificar seu multiprocessamento.

Uma limitação, atualmente, da visualização no **RiserView** é que *risers* e vórtices são os únicos elementos da cena que contam com esses recursos. Outro trabalho possível é estudar a visualização científica para outros elementos, como por exemplo para mostrar a configuração das correntes marítimas (longe dos *risers*) ou para a visualização da dinâmica da superfície da água.

6.2.4. Navegação

Uma observação interessante feita durante o desenvolvimento, apesar da não realização de testes formais de usabilidade, é relativa à navegação pela cena. Inicialmente essa navegação era implementada de forma que toda cena era encarada como um único objeto (como se fosse um globo) que podia ser afastado, aproximado, rotacionado ou arrastado em várias direções para facilitar sua visualização. Apesar de intuitiva para o autor, essa forma de navegação mostrou muitas vezes ir diretamente contra a intuição de vários usuários, que esperavam navegar pela cena como se estivessem imersos nela, como é tradicionalmente feito em ambientes virtuais ou jogos. A substituição do estilo de navegação anterior por este aparentemente aumentou consideravelmente a usabilidade deste aspecto da interface.

Outro ponto relativo à navegação está relacionado aos tipos de atores no sistema. No capítulo 4 afirma-se que o **RiserView** tem somente um tipo de ator, responsável tanto pela navegação das cenas quanto por sua criação. Pode ser interessante separar essas responsabilidades para dois atores distintos e refletir essa separação na interface

(com um módulo de criação de cenas e um de navegação, por exemplo). Se no futuro for implementado um meio de expandir o sistema sem a necessidade de criar extensões do código e recompilá-lo, com o uso de *scripts* por exemplo, um terceiro tipo de ator pode surgir, responsável pela inclusão desses scripts.

6.3. Tecnologias

Nesta seção procura-se analisar as principais vantagens e desvantagens do uso das principais tecnologias do trabalho: o Processo Unificado (UP), o VTK e o IUP.

Uma das primeiras observações que se pode fazer a respeito do uso do UP é o quanto o trabalho de customização do *framework* para adequá-lo ao projeto e à sua escala de fato simplificou o uso da metodologia. Basta comparar os passos relatados no capítulo 4 e o número de artefatos gerados com a descrição do processo por Jacobson et al. (1998). Ainda assim, o uso do UP resultou em uma documentação mais completa e em uma arquitetura mais elegante. Na verdade, o processo só foi selecionado e aplicado depois que a primeira iteração descrita no capítulo 4 já havia sido concluída, pois no início do trabalho havia uma maior preocupação com a viabilidade do uso do VTK e do âmbito do projeto do que com a metodologia, e o autor tinha pouco conhecimento da área de engenharia de *software*. Uma vez que o processo foi aplicado, a arquitetura que já existia pôde ser consideravelmente aprimorada através do uso de padrões de projeto e de generalizações que ficaram claras com o uso da metodologia. Outra benefício bastante visível do uso do UP foi a possibilidade de usar os casos de uso como guia para os testes de caixa-preta, como relatado no capítulo 5.

Quanto ao uso do VTK, essa API forneceu uma infra-estrutura orientada a objetos que não existiria caso o OpenGL tivesse sido usado diretamente. Mas os maiores benefícios obtidos com seu uso foram a portabilidade do código e a disponibilidade em um único pacote de algoritmos para diversas tarefas de visualização científica, *anti-aliasing* e estereoscopia. Durante os testes foi possível observar ainda como o VTK, através do OpenGL, faz uso dos recursos disponíveis nas placas gráficas atuais. Embora no capítulo 5 tenha sido registrado o conjunto de testes mais formal somente, testes menos formais foram realizados em plataformas sem aceleração gráfica e que utilizavam placas gráficas com diferentes performances (TNT 2, GeForce 2 e GeForce 4, além da GeForce FX citada no capítulo 5) e a diferença de performance entre esses sistemas foi bem visível. Uma observação interessante foi a de como, para o *anti-aliasing*, o VTK ainda não aproveita os recursos disponíveis em *hardware*.

A portabilidade do código não foi conseqüência unicamente do VTK. O uso do IUP no desenvolvimento da interface também permitiu que isso ocorresse. Além disso, a simplicidade dessa API permitiu que a interface fosse desenvolvida de forma bastante rápida.

A principal desvantagem do uso do VTK foi que o acesso a recursos do OpenGL diretamente torna-se mais complicado, de forma que para fazer uso de tecnologias mais recentes torna-se necessário um trabalho maior (como a biblioteca é *open-source* sempre é possível extendê-la por conta própria e, apesar de complexa, sua arquitetura e boa documentação simplificam esse trabalho de extensão), ou então aguardar uma nova versão da API. Já a principal desvantagem do uso do IUP foi a necessidade de violar a orientação a objetos para o tratamento de suas mensagens, uma vez que a API não segue

esse paradigma. Uma desvantagem importante do uso dessas duas bibliotecas em conjunto é que não se conseguiu implementar uma forma de visualização em tela-cheia (*fullscreen*) para o aplicativo (embora fazê-lo separadamente no IUP ou no VTK é trivial). Isso torna a estereoscopia, por exemplo, bem menos acessível e prejudicou os testes desse recurso, pois atualmente placas capazes de renderizar cenas estereoscópicas dentro de janelas são muito mais caras que as que o fazem em modo de tela-cheia.

Apesar dessas desvantagens pode-se afirmar que, sem o uso dessa bibliotecas, ou ao menos de similares (e nesse caso existem bem mais APIs similares ao IUP que ao VTK), esse trabalho não poderia ter atingido a complexidade e os objetivos que atingiu, de forma que considera-se seu uso plenamente justificado.

REFERÊNCIAS BIBLIOGRÁFICAS[17]

AHRENS, J. et al. **A Parallel Approach for Efficiently Visualizing Extremely Large Time-Varying Datasets**. Los Alamos National Laboratory, 2000. (Technical Report - LAUR-00-1620).

ANGEL, E. **Interactive Computer Graphics**: A top-down approach with OpenGL. 2nd. Ed. Addison-Wesley, 2000. 612 p.

BARRERA, B. Introdução ao OpenGL Shading Language. **JogosPRO e-MAGAZINE**, n. 1, p. 20-22, set. 2004. Disponível em: http://www.jogospro.com.br. Acesso em: 20 out. 2004.

BICHO, A. et al. Programação Gráfica 3D com OpenGL, Open Inventor e Java 3D. **REIC**, v. 2, n. 1, p. 1-43, March 2002.

BIRKEN, R. & Versteeg, R. Use of four-dimensional ground penetrating radar and advanced visualization methods to determine subsurface fluid migration. **Journal of Applied Geophysics**. n. 43, p. 215-226, 2000.

BLEVINS, R. **Flow-Induced Vibration**. 2nd. Ed. New York : Van Nostrand Reinhold, 1990. 451 p.

BOOCH, G.; Rumbaugh, J & Jacobson, I. **The Unified Modeling Language User Guide**. Addison-Wesley, 1999. 482 p.

BURDEA, G. & Coiffet, P. **Virtual Reality Technology**. Wiley-Interscience, 1994. 400 p.

---

[17] De acordo com: ASSOCIAÇÃO BRASILEIRA DE NORMAS TÉCNICAS. **NBR 6023**: informação e documentação: referências: elaboração. Rio de Janeiro, 2002.

CERNY, D. & Sochor, J. Collision Detection with Extended Fixed Direction Hulls. In: WORKSHOP DE REALIDADE VIRTUAL, 3, 2000. **Anais...** SBC, 2000. p. 205-213.

CHUNG, K. & Wang, W. Quick Collision Detection of Polytopes in Virtual Environments. ACM Symposium on Virtual Reality Software and Technology, 1996. **Anais...** ACM, 1996. pp. 1-4.

COHEN, J. et al. I-Collide: An Interactive and Exact Collision Detection System for Large-Scale Environments. SYMPOSIUM ON INTERACTIVE 3D GRAPHICS, 1995. **Anais...** ACM, 1995. p. 183-191.

DACHILE, F. & Kaufman, A. High Degree Temporal Antialiasing. In: COMPUTER ANIMATION 2000. **Anais...** 2000. p. 49-55.

EBERLY, D. Distance Between Two Line Segments in 3D. 2000. Disponível em: http://www.magic-software.com. Acesso em: 20 out. 2004.

EBERLY, D. Intersection of Cylinders. 2000. Disponível em: http://www.magic-software.com. Acesso em: 20 out. 2004.

FERRARI, J. A. **Hydrodynamic Loading and Response of Offshore Risers**. 1999. 254 p. Tese de Doutorado - Imperial College of Science, Technology and Medicine, London.

FIGUEIREDO, L. H. Uma estratégia de portabilidade para aplicações gráficas interativas. In: SIBIGRAPI, 6, 1993. **Anais...** SBC, 1993. p. 203-211.

FLATT, M.; Krishamurthi, S & Felleisen, M. Classes and Mixins. In: ACM SIGPLANSIGACT SYMPOSIUM ON PRINCIPLES OF PROGRAMMING LANGUAGES, 25, 1998. **Anais...** ACM, 1998. p. 171-183.

FOLDOC - Free Online Dictionary of Computing. Disponível em: http://foldoc.doc.ic.ac.uk/foldoc/index.html. Acesso em: 20 out. 2004.

FOLEY, J. et al. **Computer Graphics**: Principles and Practice. 2nd. ed. Addison-Wesley, 1996. 1175 p.

GAMMA, E. **Design Patterns**: elements of reusable object-oriented software. Addison-Wesley, 1995. 395 p.

GLASSNER, A. S. (Ed.). **Graphic Gems**. Academic Press, 1990. 833 p.

GÜNTHER, G. et al. Non-local Methods for Vortex Extraction from Vector Data. in SCIENTIFIC VISUALIZATION CONFERENCE, Dagstuhl 2000. **Anais...** 2000.

GYÖRGY, K. Advection in the von Kármán vortex street and in the field of the leapfrogging vortex pairs Disponível em: http://www.me.bme.hu/munkatrs/karolyi/. Acesso em: 20 out. 2004.

IUP - Portable User Interface. Disponível em: http://www.tecgraf.puc-rio.br/iup. Acesso em: 20 out. 2004.

JACOBSON, I.; Booch, G. & Rumbaugh, J. **The Unified Software Development Process**. Addison-Wesley, 1999. 463 p.

JAVA Technology. Disponível em: http://java.sun.com. Acesso em: 20 out. 2004.

JEONG, J. & Hussain, F. On the identification of a vortex. **Journal of Fluid Mechanics**, v. 285, p. 69-94, 1995.

KESSENICH, J.; Baldwin, D. & Rost, R. The OpenGL Shading Language. Language Version 1.10, rev. 59, April 2004. Disponível em: http://www.opengl.org. Acesso em: 20 out. 2004.

KITWARE VTK. Disponível em: http://www.kitware.com/vtk. Acesso em: 20 out. 2004.

KEIM, D. Visual Exploration of Large Data Sets. **Communications of the ACM**. v. 44, n. 8, p. 38-44, August 2001.

KIRNER, C. & Pinho, M. **Introdução à Realidade Virtual**. Apostila do Minicurso ministrado na XV JAI, durante o XVI Congresso da SBC, Recife, PE, agosto de 1996, 60 pp.

LAW, C.; Henderson, A & Ahrens, J. An Application Architecture for Large Data Visualization: A Case Study. In: IEEE 2001 SYMPOSIUM ON PARALLEL AND LARGE-DATA VISUALIZATION AND GRAPHICS. IEEE, 2001. p. 85-92.

LEVY, C. H. et al. IUP/LED: A Portable User Interface Development Tool. **Software: Practice and Experience**. June 1996, p. 737-762.

LUCKAS, V. & Dörner, R.: Experiences from the Future: Using Object-Oriented Concepts for 3D Visualization and Validation of Industrial Cenarios. **ACM Computing Surveys**. v. 32, n. 1, March 2000. Article No. 38.

MAHONEY, D. P. Modeling for Virtual Reality. **Computer Graphics World**. v. 18, n. 10, p. 1-3, 1995.

MENEGHINI, J. **Numerical simulation of bluff body flow control using a discrete vortex method**. 1993. Tese de Doutorado - Faculty of Engineering of the University of London.

PAUL, B. Mesa Homepage. Disponível em: http://www.mesa3d.org/. Acesso em: 20 out. 2004.

PETROBRAS. Disponível em: http://www.petrobras.com.br. Acesso em: 20 out. 2004.

PETROBRAS Magazine, 7, 26, pp. 20-23, 1999.

PETROBRAS Magazine, 7, 33, pp. 14-17, 2001.

PIMENTEL, K. & Teixeira, K. **Virtual Reality**: Through the Looking Glass. 2nd. Ed. New York: Intel, 1995. 438 p.

PV-WAVE. Disponível em: http://www.sv.vt.edu/classes/ESM4714/Student_Proj/ class97/vlachos/pvlachos/pvwave.htm. Acesso em: 20 out. 2004.

ROGERS, D. Z-buffering, Interpolation and More W-buffering. Disponível em: http://developer.nvidia.com/object/Z_Buff.html. Acesso em: 20 out. 2004.

SCHIRSKI, M. et al. ViSTA FlowLib: A Framework for Interactive Visualization and Exploration of Unsteady Flows in Virtual Environments. WORKSHOP ON VIRTUAL ENVIRONMENTS, 2003. **Anais...** 2003. p. 77-85.

SCHROEDER, W.; Avila, L. & Hoffman, W. Visualizing with VTK: A Tutorial. **IEEE Computer Graphics and Applications**. October 2000, p. 20-27.

SCHROEDER, W. et al. The Design and IMplementation of an Object-Oriented Toolkit for 3D Graphics and Visualization. In: CONFERENCE ON VISUALIZATION, 7, 1996. **Anais...** ACM, 1996. p. 93-101.

SCHROEDER, W.; Martin, K & Lorensen, B. **The Visualization Toolkit**. 2nd. Ed. Prentice Hall, 1998. 645 p.

SGI - Open Inventor. Disponível em: http://www.sgi.com/software/inventor. Acesso em: 20 out. 2004.

SGI - OpenGL Performer. Disponível em: http://www.sgi.com/software/performer. Acesso em: 20 out. 2004.

SGI. **OpenGL Performer**: Real-Time 3D Rendering for High-Performance and Interactive Graphics Applications. SGI, 2002. (White Paper).

SILVEIRA, E. et al. Um sistema computacional integrado para análise dinâmica não-linear geométrica de linhas de ancoragem. In: IBERIAN LATIN-AMERICAN CONGRESS ON COMPUTATIONAL METHODS IN ENGINEERING, 21, 2000. **Anais...** 2000. v. 7, p. 26.1-26.20.

SMITH, J. **A Comparision of RUP and XP**. Rational Software, 2002. (White Paper)

TAI, L. The GUI Toolkit, Framework Page. Disponível em: http://www.atai.org/guitool. Acesso em: 20 out. 2004.

TAKAHASHI, T.; Liesenberg, H. K. & Xavier, D. T. **Programação orientada a objetos**: uma visão integrada do paradigma de objetos. IME-USP, 1990. 335 p.

TANQUE de Provas Numérico. Disponível em: http://not.poli.usp.br/. Acesso em: 20 out. 2004.

THALMANN, D. & Thalmann, N. M. **Computer Animation**: Theory and Pratice. 2nd. Ed. Springer-Verlag, 1990. 240 p.

UPSON, C. et al. The Physical Simulation and Visual Representation of Natural Phenomena. **Computer Graphics**, v. 21, n. 4, p. 335-336, 1987.

VAN LIERE, R.; Harkes, J. & Leeuw, W. A Distributed Blackboard Architecture for Interactive Data Visualization. In: CONFERENCE ON VISUALIZATION, 1998. **Anais...** IEEE, 1998, p. 225-231, 537.

VEIT, M. & Herrmann, S. Model-View-Controller and Object Teams: A Perfect Match of Paradigms. In: INTERNATIONAL CONFERENCE ON ASPECT-ORIENTED SOFTWARE DEVELOPMENT, 2, 2003. **Anais...** 2003, p. 140-149.

WEB3D Consortium. Disponível em: http://www.web3d.org. Acesso em: 20 out. 2004.

WLOKA, M.; Zeleznik, R. & Miller, T. Practically Frameless Rendering. 1995. (Relatório Técnico Brown University, CS-95-06).

APÊNDICE A - CASOS DE USO

1. Adicionar Objeto

Participantes: Usuário

Descrição: Este caso é iniciado quando o usuário utiliza o menu da interface para adicionar um objeto à cena. O objeto pode ser um *riser*, uma representação de vórtices, da superfície da lâmina d'água, do solo ou um modelo rígido para representar embarcações ou estruturas. O objeto é gerado a partir de um arquivo de dados contendo seu comportamento ao longo do tempo, escolhido pelo usuário através de uma caixa de diálogo que deve exibir inicialmente somente os arquivos do tipo correto. O nome do arquivo aberto deve ser armazenado para permitir que se gere uma lista dos arquivos usados na cena. No caso de *risers* e modelos, o objeto é simplesmente adicionado à uma coleção de objetos semelhantes. Já a superfície, o fundo e vórtices são únicos para cada cena, de forma que um novo objeto substitui o anterior ao ser adicionado. O objeto adicionado é exibido na tela, e a câmera é deslocada para uma posição adequada. Objetos dinâmicos (todos exceto o fundo), ao serem adicionados, liberam o controle da animação para o usuário. Caso a animação tenha sido iniciada anteriormente, novos objetos são adicionados na posição correspondente ao tempo atual da animação.

Variações: Caso o arquivo escolhido pelo usuário seja inválido, a interface deve gerar uma mensagem de erro e o processo é interrompido.

Caso o passo de tempo contido no arquivo seja diferente do utilizado pelos outros objetos, a interface deve gerar uma mensagem de alerta.

2. Adicionar Lista de Objetos

Participantes: Usuário

Descrição: Este caso é iniciado quando o usuário utiliza o menu da interface para carregar um arquivo contendo uma lista de arquivos descrevendo a dinâmica de objetos que devem ser adicionados à cena. Estes objetos podem ser *risers*, uma representação de vórtices, da superfície da lâmina d'água, do solo ou modelos rígidos para representar embarcações ou estruturas. O nome de todos os arquivos constantes da lista devem ser armazenados para permitir que se gere uma nova lista dos arquivos usados na cena. No caso de *risers* e modelos, os novos objetos são simplesmente adicionado a uma coleção de objetos semelhantes. Já a superfície, o fundo e vórtices são únicos para cada cena, de forma que um novo objeto substitui o anterior ao ser adicionado. Os objetos adicionados são exibidos na tela, e a câmera é deslocada para uma posição adequada. Objetos dinâmicos (todos exceto o fundo), ao serem adicionados, iniciam a animação, caso não tenha sido iniciada ainda, e liberam o controle da mesma para o usuário. Caso a animação já tenha sido iniciada, novos objetos são adicionados na posição correspondente ao tempo atual da animação.

Variações: Caso o arquivo escolhido pelo usuário seja inválido, a interface deve gerar uma mensagem de erro e o processo é interrompido.

Caso o passo de tempo contido num arquivo seja diferente do utilizado pelos outros objetos, a interface deve gerar uma mensagem de alerta.

Caso qualquer um dos arquivos da lista seja inválido (por exemplo, foi apagado após a geração da lista), a interface deve gerar uma mensagem de erro, mas o processo deve continuar para os outros objetos.

3. Gravar Lista de Objetos

Participantes: Usuário

Descrição: Este caso é iniciado quando o usuário utiliza o menu da interface para solicitar que seja gravado um arquivo contendo a lista de arquivos que descrevem a dinâmica dos objetos que compõem a cena atual. Dessa forma a cena pode ser reconstruída no futuro com somente uma operação. A interface deve permitir que o usuário escolha o nome e diretório do arquivo a ser salvo, bem como alertá-lo se já existir um arquivo com o mesmo nome, antes de sobrescrevê-lo.

Variações: Caso haja um erro na gravação do arquivo (por exemplo, disco cheio ou usuário sem autorização para gravar o arquivo no local escolhido) a interface deve gerar uma mensagem de erro.

4. Limpar Cena

Participantes: Usuário

Descrição: Este caso é iniciado quando o usuário utiliza o menu da interface para solicitar que a cena atual seja apagada. Isso significa remover todos os objetos da cena, fazendo as reinicializações necessárias para que uma nova cena possa ser criada com novos objetos, limpar a lista de arquivos utilizados, parar e desabilitar a animação enquanto não há objetos ativos na cena e zerar o tempo de animação.

Antes desta operação ser realizada, o programa pergunta ao usuário se deseja salvar a lista de objetos que compões a cena, bem como as opções atuais.

5. Exibir/Esconder Objeto

Participantes: Usuário

Descrição: Este caso é iniciado quando o usuário utiliza o menu da interface para escolher quais objetos serão exibidos ou não. No caso de *Risers* e Modelos, todos os objetos deste tipo são exibidos ou ocultos com este comando. Ondas e fundo são únicos para cada cena e portanto também são exibidos ou ocultos como um todo. Mesmo que não haja nenhum objeto na cena ainda, a exibição ou não de objetos daquele tipo pode ser definida e é válida para objetos adicionados à cena posteriormente. Vórtices são um caso especial, tratado abaixo.

6. Exibir Vórtices ou Campos

Participantes: Usuário

Descrição: Este caso é iniciado quando o usuário utiliza o menu da interface para escolher qual representação dos vórtices deve ser exibida. O usuário pode escolher não exibir nenhuma representação, exibir conjuntos de vórtices discretos ou campos escalares (por exemplo pressão, vorticidade ou módulo da velocidade). A opção de exibir campos só é habilitada caso o arquivo atual de dados de vórtices contenha uma descrição desta representação. Independente da opção escolhida, vórtices são exibidos em planos horizontais e portanto só podem ser visualizados se a vista atual for horizontal.

7. Mostrar Colisões

Participantes: Usuário

Descrição: Este caso é iniciado quando o usuário utiliza o menu da interface para ativar ou não a exibição de colisões entre os *risers*. A amplificação dos diâmetros e dos movimentos dos *risers* é desativada e cada vez que uma colisão acontece a cor dos *risers* envolvidos é modificada na vizinhança do ponto de colisão, uma mensagem de alerta é mostrada pela interface e a animação pára, de forma a permitir que o usuário possa registrar o momento e local da colisão.

8. Ativar Estereoscopia

Participantes: Usuário

Descrição: Este caso é iniciado quando o usuário utiliza o menu da interface para ativar a estereoscopia, ou quando a tecla "3" é pressionada durante a navegação. O programa passa a gerar pares de imagens estereoscópicas até que esta opção seja desativada.

Variações: Caso o sistema gráfico não seja capaz de exibir imagens estereoscópicas, esta opção deve permanecer desabilitada.

9. Ativar *Anti-Aliasing*

Participantes: Usuário

Descrição: Este caso é iniciado quando o usuário utiliza o menu da interface para ativar a opção de *anti-aliasing*. A cena é exibida com uma menor intensidade dos efeitos de *aliasing* espacial (discutido no capítulo 2) até que a opção seja desativada.

10. Modificar Visualização dos *Risers*

Participantes: Usuário

Descrição: Este caso é iniciado quando o usuário utiliza o menu da interface de opções para os *risers*. A interface deve permitir ao usuário modificar o número de faces com que o perfil circular do *riser* deve ser discretizado, bem como fatores de amplificação

para o diâmetro e para o movimento dos *risers*. Estes fatores existem para facilitar a visualização da cena como um todo, uma vez que os *risers* são estruturas muito delgadas e seus movimentos tipicamente são muito pequenos em relação a seu comprimento.

11. Aplicar Texturas

Participantes: Usuário

Descrição: Este caso é iniciado quando o usuário utiliza o menu da interface de opções de texturas. A interface deve permitir ao usuário selecionar um arquivo do tipo BMP para servir como textura para o fundo oceânico, e um para servir como textura para a superfície da lâmina d'água. Deve ainda permitir ao usuário determinar quantas vezes a textura deve ser repetida ao longo das superfícies.

12. Definir Cores

Participantes: Usuário

Descrição: Este caso é iniciado quando o usuário utiliza o menu da interface de opções de cores. A interface deve permitir que o usuário selecione cores para o fundo da cena, o leito oceânico e para a superfície da lâmina d'água.

13. Importar Opções

Participantes: Usuário

Descrição: Este caso é iniciado quando o usuário utiliza o menu da interface para selecionar um arquivo contendo um conjunto de opções pré-definidas para serem aplicadas à cena. Este arquivo deve conter opções para visualização dos *risers*, cores, e texturas.

Variações: Caso o arquivo escolhido pelo usuário seja inválido, a interface deve gerar uma mensagem de erro e o processo é interrompido.

14. Exportar Opções

Participantes: Usuário

Descrição: Este caso é iniciado quando o usuário utiliza o menu da interface para gravar um arquivo contendo o conjunto de opções utilizadas na cena atual. Este arquivo deve conter opções para visualização dos *risers*, cores e texturas.

Variações: Caso haja um erro na gravação do arquivo (por exemplo, disco cheio ou usuário sem autorização para gravar o arquivo no local escolhido) a interface deve gerar uma mensagem de erro.

15. Parar Animação

Participantes: Usuário

Descrição: Este caso é iniciado quando o usuário utiliza o menu ou a barra de ferramentas da interface para parar a animação. Esta opção só é habilitada se houver pelo menos um objeto ativo na cena. Os objetos dinâmicos cessam seu movimento e o contador de tempo também pára. A navegação pela cena é permitida.

16. Começar Animação

Participantes: Usuário

Descrição: Este caso é iniciado quando o usuário utiliza o menu ou a barra de ferramentas da interface para começar a animação. Esta opção só é habilitada se houver pelo menos um objeto ativo na cena. Os objetos ativos da cena passam a se mover novamente de acordo com sua dinâmica em tempo real (se o *hardware* permitir), a partir do momento em que estavam parados. O contador de tempo volta a correr também.

17. Voltar Quadro

Participantes: Usuário

Descrição: Este caso é iniciado quando o usuário utiliza o menu ou a barra de ferramentas da interface para voltar um quadro na animação. Esta opção só é habilitada se houver pelo menos um objeto ativo na cena e se o tempo atual da animação for maior que zero (valores de tempo negativos não são permitidos). A animação volta para o quadro anterior e pára. O contador de tempo recua um passo de tempo (o tamanho do passo é definido nos arquivos de dados).

18. Avançar Quadro

Participantes: Usuário

Descrição: Este caso é iniciado quando o usuário utiliza o menu ou a barra de ferramentas da interface para avançar um quadro na animação. Esta opção só é habilitada se houver pelo menos um objeto ativo na cena. A animação avança um quadro e pára. O contador de tempo avança um passo de tempo (o tamanho do passo é definido nos arquivos de dados).

19. Voltar Acelerado

Participantes: Usuário

Descrição: Este caso é iniciado quando o usuário utiliza o menu ou a barra de ferramentas da interface para avançar um quadro na animação. Esta opção só é habilitada se houver pelo menos um objeto ativo na cena e se o tempo atual da animação for maior que zero (valores de tempo negativos não são permitidos). A animação acontece voltando no tempo em velocidade acelerada, acompanhada pelo marcador de tempo, até que o usuário dê outro comando para manipular a animação, ou até que o tempo atinja zero.

20. Avançar Acelerado

Participantes: Usuário

Descrição: Este caso é iniciado quando o usuário utiliza o menu ou a barra de ferramentas da interface para avançar um quadro na animação. Esta opção só é habilitada se houver pelo menos um objeto ativo na cena. A animação avança no tempo em velocidade acelerada, acompanhada pelo marcador de tempo, até que o usuário dê outro comando para manipular a animação.

21. Alternar Vista

Participantes: Usuário

Descrição: Este caso é iniciado quando o usuário utiliza o menu ou a barra de ferramentas da interface para alternar entre as vistas vertical ou horizontal, ou quando pressiona a barra de espaço durante a navegação. A posição da cena em cada vista deve ser armazenada, de forma que ao alternar entre vistas esta posição não seja perdida.

22. Enquadrar Cena

Participantes: Usuário

Descrição: Este caso é iniciado quando o usuário utiliza o menu ou a barra de ferramentas da interface para enquadrar a cena. A cena é reposicionada para que todos os elementos possam ser vistos na tela.

23. Avançar/Recuar

Participantes: Usuário

Descrição: Este caso é iniciado quando o usuário pressiona (ou mantém pressionadas) as teclas de seta para cima ou para baixo, movendo-se para frente ou para trás na direção em que está vendo a cena, em velocidade constante.

24. Virar para a Esquerda ou Direita

Participantes: Usuário

Descrição: Este caso é iniciado quando o usuário pressiona (ou mantém pressionadas) as teclas de seta para esquerda ou direita, rotacionando a posição do observador da cena, ou seja, a direção em que ele visualiza a cena.

25. "Strafe"

Participantes: Usuário

Descrição: Este caso é iniciado quando o usuário pressiona (ou mantém pressionada) qualquer uma das setas direcionais juntamente com a tecla "shift". O observador se move para os lados, para cima ou para baixo sem mudar a direção em que a cena esta sendo visualizada.

APÊNDICE B - GLOSSÁRIO PARA METODOLOGIA

Animação: técnica através da qual uma ilusão de movimento (ou qualquer outra mudança dinâmica, como crescimento) é criada exibindo uma seqüência de imagens a um certo ritmo, onde cada imagem é uma alteração da anterior. Tipicamente, o ritmo de das imagens é de 24 quadros/s ou 30 quadros/s, mas até ritmos tão baixos quanto 5 quadros por segundo ainda fornecem a ilusão de movimento, ainda que a transição entre os quadros torne-se perceptível (como "pulos") nesse caso. Numa animação genérica, podem ocorrer diversas mudanças com os objetos observados (posição, orientação, tamanho, forma, cor, transparência), com o observador (posição, ponto de referência, ângulo) e com as fontes de luz (posição, intensidade, cor), ou até com a animação em si (velocidade das imagens).

Anti-aliasing: técnicas e algoritmos utilizados para amenizar o problema de *aliasing* que surge devido à discretização de imagens e entidades geométricas em *pixels*, normalmente percebida como linhas "quebradas", lembrando degraus, observadas com freqüência em gráficos de computador.

APIs: *Application Programmer Interfaces*, ou seja, bibliotecas de funções que fazem a interface entre o programador de uma aplicação e um sistema de mais baixo nível, seja ele de *hardware* ou *software*.

Aplicação, Aplicativo: um *software* ou programa que realiza diretamente uma ou mais tarefas para o usuário (ao contrário de bibliotecas, por exemplo, que podem ser utilizadas para facilitar a criação de aplicações mas não realizam tarefas por si só).

Barra de Ferramentas: elemento da interface composto por um conjunto de botões alinhados horizontalmente lado a lado (ou, menos comunmente, verticalmente um sobre o outro) que permitem ao usuário um rápido acesso a algumas das funções mais utilizadas de um *software*.

Campos: neste trabalho esta palavra normalmente se refere a campos escalares, ou seja, regiões do espaço ao longo das quais uma grandeza escalar assume valores de forma contínua.

Cena: imagem exibida pelo programa, composta por um conjunto de objetos com dinâmicas próprias.

Cursor: pequeno ícone que indica a posição do mouse e, eventualmente, sua função.

Estereoscopia: mecanismo mais importante para a maioria das pessoas para percepção de profundidade, onde ambos os olhos registram imagens ligeiramente diferentes para uma cena e o cérebro interpreta a diferença entre essas imagens como medida de profundidade. Para reproduzir este mecanismo criando uma ilusão de profundidade com imagens projetadas numa tela plana, gera-se uma imagem diferente para cada olho e faz-se uso de algum mecanismo para separa o que é enxergado por cada olho.

Fundo: no contexto deste projeto, referências a "Fundo" normalmente estão relacionadas ao leito oceânico, ao solo. Quando a palavra for usada com outro significado (por exemplo, a cor de fundo da tela), isto será explicitado no texto.

Interface: neste trabalho esta palavra é usada normalmente em referência à interface gráfica entre o programa e o usuário, ou seja, o conjunto de elementos gráficos que podem ser manipulados pelo usuário para lhe permitir controle sobre o programa. Quando a palavra for usada com outro significado (por exemplo, a interface entre os componentes de um programa) isto será explicitado no texto.

IUP: biblioteca gratuita e multiplataforma para criação de interfaces de usuário.

Menu: elemento comum da interface composto por um conjunto de palavras alinhadas horizontalmente lado a lado, normalmente no topo da janela da aplicação. Cada palavra pode estar associada a um submenu (um novo conjunto análogo de palavras alinhadas verticalmente uma sobre a outra) ou acessar uma função da aplicação.

Navegação: manipulação da cena de modo a permitir que o usuário a visualize de ângulos diferentes.

Objeto: neste projeto, objeto normalmente se refere a um elemento da cena. Um *riser*, uma representação da superfície da lâmina d'água, do fundo ou de vórtices ou modelos rígidos para representar .

Objeto Ativo: objetos que têm uma dinâmica própria e não são necessariamente estáticos. No caso desta aplicação, todos os objetos, exceto o fundo.

Ondas: outro nome usado para se referir à superfície da lâmina d'água neste projeto, em referência a seu movimento.

OpenGL: API gráfica para renderização de cenas em três dimensões que atualmente se tornou um padrão de fato no mercado.

Passo de Tempo: quantidade de tempo utilizada para discretizar uma simulação ou animação.

Pixels: *Picture Element*, ou elemento de imagem, a mínima unidade de área em que imagens são discretizadas nos dispositivos de exibição.

Quadro: imagem estática que, junto ao outros quadros, compõe a animação.

Risers: estruturas tubulares que ligam os poços de exploração no solo oceânico às plataformas ou navios na superfície.

Textura: padrões, sejam extraídos de uma imagem digital, seja gerados por uma função matemática, que podem ser aplicados (ou mapeados) sobre a geometria. Texturas são ferramentas bastante úteis para fornecer um nível de detalhe (e realismo) maior à superfícies sem a necessidade de aumentar a complexidade dos modelos geométricos utilizados.

UIT: *User Interface Toolkits*, ou bibliotecas de objetos de interface que implementam diferentes técnicas de interação com o usuário. Tipicamente incluem formas para simplificar a descrição e composição das interfaces, que vão desde uma linguagem a editores gráficos. Exemplos são SDK do MS Windows, OSF/Motif, XView e Macintosh Toolbox.

Vista: plano no qual a cena é visualizada. O ambiente de visualização desenvolvido neste trabalho só permite que o usuário navegue por vistas verticais ou horizontais.

Vórtices: Áreas num fluxo onde o fluido adquire um movimento circular devido ao desprendimento da camada limite à jusante de um corpo presente no fluxo. Vórtices induzem variações de pressão e velocidade ao longo do fluxo e podem gerar esforços cíclicos sobre o corpo inserido nele.

VTK: sistema de *software open-source*, portável e orientado a objetos para computação gráfica em 3D, visualização científica e processamento de imagens, baseado no OpenGL. É implementado em C++ mas também suporta Tcl, Python e Java.

APÊNDICE C - *BUGS* E APRIMORAMENTOS

Esse apêndice reúne os *bugs*, ou defeitos, conhecidos da versão atual do RiserView, a maioria descoberta durante os testes de caixa preta, bem como pequenas melhorias, principalmente na interface, sugeridas por usuários ou planejadas mas não implementadas por questões de tempo. O apêndice está dividido em duas seções, descrevendo bugs e aprimoramentos, respectivamente.

C.1. Bugs

Nesta seção, os *bugs* estão agrupados conforme o caso de uso (ver Apêndice A) a que se referem. No final da seção são relatados os problemas que não se referem a um caso de uso em particular. Quando possível, explicações quanto a possíveis causas do problema e sugestões para sua solução são também mencionados.

CASO DE USO 1 - ADICIONAR OBJETO

- Com a versão mais atual do IUP (2.2.2) e o sistema operacional Windows XP (e possivelmente também em outros sistemas operacionais), as caixas de diálogo para abrir ou salvar arquivo não mostram mais a extensão correta dos arquivos daquele tipo, mostrando ao invés disso arquivos de todos os tipos (*.*). **Possível causa**: a função IupGetFile, usada para exibir a caixa de diálogo para abrir ou salvar

arquivos, não se comporta mais de acordo com a documentação, pelo menos nesse sistema. A função recebe como parâmetro uma string com a extensão do tipo de arquivo, mas esse parâmetro parece não ter efeito. Em versões anteriores isso não ocorria.

- Tentar importar um arquivo de modelo inválido gera uma janela de erro do VTK e por vezes pode "travar" a aplicação que deve, então, ser fechada. O problema das mensagens de erro do VTK é um problema mais geral: nesse trabalho não se determinou uma maneira de evitar com que essas janelas sejam exibidas e, ao invés disso, que se permita que o RiserView, e não o VTK, trate os erros e mande as mensagens necessárias. A forma de contornar esse problema foi tentando prever o máximo possível de erros antes que sejam enviados ao VTK. Sendo assim, uma possível solução intermediária para reduzir esse problema seria inserir um identificador (uma *tag* no arquivo) para que o RiserView possa detectar se o arquivo de modelo é válido ou não. Os arquivos de modelo (.mdl), no entanto, são constituídos por duas partes: uma descrição do movimento do modelo e o nome do arquivo do 3D Studio (.3DS) que contém as informações geométricas. A mudança sugerida só soluciona o problema de o usuário escolher um arquivo .mdl inválido, e não a possibilidade de o arquivo .3DS ser inválido. Além disso, uma das sugestões de aprimoramento da interface envolve a separação dessas informações, como será visto adiante, o que inviabiliza essa solução.

- Durante a leitura de arquivos de vórtice (.vtx) o RiserView exibe uma caixa de diálogo mostrando o progresso da leitura do arquivo, uma vez que esse tipo de arquivo costuma ter tamanho relativamente grande e leitura demorada. Nessa caixa

de diálogo há um botão que permite ao usuário cancelar a leitura do arquivo. A funcionalidade desse botão não está implementada.

CASO DE USO 2 - ADICIONAR LISTA DE OBJETOS

- Ao abrir uma lista de objetos que não contenha nenhum objeto ativo mas contenha um arquivo com a descrição do solo, o RiserView atualmente permite a animação, e não deveria (a animação só deve ser permitida se a cena contém ao menos um objeto ativo).

CASO DE USO 8 - ATIVAR ESTEREOSCOPIA

- Atualmente o RiserView não detecta se um sistema é capaz ou não de mostrar imagens estereoscópicas para ativar ou não essa opção do menu. A opção, assim, fica sempre desativada. Resta ainda a opção de ativar a estereoscopia pelo teclado, apertando a tecla "3", opção que gera uma mensagem de erro do VTK caso o sistema não seja capaz de exibir esse tipo de imagem. O fato de o RiserView não ter a opção *fullscreen*, como já foi discutido, limita os sistemas que podem exibir imagens estereoscópicas geradas por ele.

CASO DE USO 17 - VOLTAR QUADRO

- Algumas vezes, devido a problemas de arredondamento, o aplicativo permite que o usuário volte o quadro uma vez a mais, mesmo quando o tempo zero já foi atingido, ao contrário do que estabelece o caso de uso. Isso não acarreta tempo negativo (o tempo é simplesmente zerado mais uma vez) mas é um comportamento indesejado.

A solução é utilizar uma tolerância para a comparação do tempo com zero (como é utilizado, por exemplo, na implementação da classe RVModel).

CASO DE USO 21 - ALTERNAR VISTA

- As informações de câmera não são armazenadas ao se alternar entre as duas vistas como manda o caso de uso.

OUTROS PROBLEMAS

- A implementação do mapeamento de cores para a visualização de escalares sobre *risers* e sobre campos escalares para a visualização de vórtices trouxe um problema para a interface não previsto no projeto. Como cada *riser* pode estar exibindo uma grandeza diferente mapeada sobre si, e os vórtices ainda outra grandeza, a exibição das escalas de cores para todas essas grandezas atrapalha consideravelmente a visualização. Embora a mera exibição dessas escalas seja trivial (basta acrescentar uma instância de vtkScalarBarActor à cena e parametrizá-la corretamente), a escolha de quais escalas exibir e onde provou ser mais complexa. Atualmente, nenhuma dessas escalas é mostrada. Uma opção seria exibir a escala somente para objetos selecionados pelo usuário, com o mouse por exemplo, mas isso depende da implementação de *picking* (ver próxima seção). Outra solução é uma caixa de diálogo que reuna todas as grandezas mapeadas na cena e permita que o usuário escolha qual deve ser exibida. Somente nos objetos que exibem aquela grandeza o mapeamento de cores seria feito e somente uma escala mostrada. Embora isso não seja propriamente um *bug*, está listado aqui pois considera-se que exibir o

mapeamento de cores para escalares sem exibir uma escala para essas cores constitui de fato um erro, pois reduz drasticamente a utilidade do mapeamento.

C.2. Aprimoramentos

Ao contrário das mudanças discutidas no capítulo 6, os aprimoramentos listados aqui são mais simples e geralmente relacionados a detalhes da interface, ou não tão urgentes quanto, por exemplo, as otimizações para a atualização dos *risers*.

Uma das mudanças já mencionadas é a separação, na interface e no formato de arquivo, dos arquivos que descrevem a dinâmica de um modelo de corpo rígido e do arquivo do 3D Studio que descreve sua geometria e aparência. Atualmente, um único arquivo é aberto pelo usuário (.mdl) que contém as informações do movimento do corpo bem como o nome do arquivo do 3D Studio. Com essa mudança, o usuário poderia utilizar o mesmo arquivo de movimento com diferentes modelos sem a necessidade de criar diversos arquivos .mdl. A importação de outros tipos de arquivos suportados pelo VTK, como por exemplo arquivos VRML, também é simples e interessante.

Atualmente o aplicativo não checa, ao salvar um arquivo de lista ou de opções, se aquele arquivo foi salvo com a extensão adequada (e muitas vezes o arquivo é salvo pelo usuário, por engano, sem extensão). Embora não seja recomendado forçar o usuário a utilizar uma extensão, emitir um alerta quando o arquivo tiver uma extensão diferente da padrão, ou acrescentar a extensão padrão caso o arquivo não tenha extensão alguma, podem ser mudanças úteis.

A seleção de cores, atualmente, é feita escolhendo valores entre 0 e 255 para as componentes RGB da cor de cada elemento. Sugere-se a criação de uma interface mais amigável, que permita que o usuário escolha a cor desejada visualmente.

Nos capítulo 6 discute-se como a troca do modo de navegação melhorou a usabilidade nesse aspecto para a maioria dos usuários, mas uma alternativa não implementada e interessante seria manter ambos os modos e permitir sua escolha em tempo de execução através da interface.

Implementar o *picking* ou seleção de objetos da cena com cliques do mouse é trivial (o VTK já prevê essa operação), exceto para *risers* e modelos de corpos rígidos, porque são, na verdade, formados por vários outros objetos. Esses elementos da cena, principalmente os *risers*, no entanto, são justamente os que têm mais necessidade desse recurso, por exemplo para a mudança de opções de exibição de objetos individuais da cena ou para a exibição da escala de cores mapeadas sobre cada *riser* como discutido anteriormente.

A classe de *riser* pode ser usada sem o menor problema, também, para representar em parte a amarração de embarcações (visualmente representações de fios, tubos ou cabos não têm diferença). Seria necessário fazer modificações na classe de modelo e na interface, no entanto, para tratar os objetos de maneira distinta (por exemplo separando suas opções de visualização), mas são modificações simples.

Por fim, como o RiserView é um projeto *open source*, a tradução da interface e dos comentários para o inglês bem como o uso de uma ferramenta de *software* para

controle de versões poderiam respectivamente aumentar a base de desenvolvedores que poderiam contribuir com o projeto bem como facilitar a integração dessas contribuições.

APÊNDICE D - CONTEÚDOS DO CD ANEXO

A estrutura de diretórios do CD anexo, que é parte integrante deste trabalho, bem como seus conteúdos, são descritos nesse apêndice. Diretórios e subdiretórios são descritos hierarquicamente como itens e sub-itens na lista abaixo.

- Software: Diretório contendo o RiserView e as ferramentas necessárias para sua compilação e uso.
    - Windows
        - RiserView_Fonte: Fontes comentadas do projeto. O arquivo de projeto é um arquivo do MS Visual Studio 6 que provavelmente deverá ser modificado para refletir a nova estrutura de diretórios do projeto, especialmente a localização do IUP e do VTK.
        - RiserView: Executável compilado (para *release*) e demais arquivos necessários para sua execução.
        - VTK: Instalação da versão do VTK usada no RiserView.
        - IUP: Instalação da versão do IUP usada no RiserView.

- Linux
    - RiserView_Fonte: Fontes comentadas do projeto, sem um makefile ou arquivo de projeto.
    - VTK: Fontes compiláveis da versão do VTK usada no RiserView e a ferramenta CMake para auxiliar na compilação.
    - IUP: Fontes compiláveis da versão do IUP usada no RiserView e a ferramenta TecMake para auxiliar na compilação.
- Entradas: Diversos arquivos de entrada usados durante os testes ou como exemplos.

- Docs: Manual do usuário do RiserView em HTML (Man.html e arquivos do subdiretório Man), cópia eletrônica dessa dissertação (RiserView.pdf) e do artigo em inglês entregue com ela e publicado no SVR 2004 (RiserView_Paper.pdf).
    - VTK: Documentação e fontes da versão do VTK utilizada.
    - IUP: Documentação da versão do IUP utilizada.