Reengenharia e evolução do visualizador e modelador do sistema *Freeflow3D*

Fábio de Toledo Pereira

Reengenharia e evolução do visualizador e modelador do sistema *Freeflow3D*

Fábio de Toledo Pereira

Orientador: ***Prof. Dr. Antonio Castelo Filho***

Dissertação apresentada ao Instituto de Ciências Matemáticas e de Computação - ICMC-USP, como parte dos requisitos para obtenção do título de Mestre em Ciências de Computação e Matemática Computacional. *EXEMPLAR DE DEFESA.*

USP – São Carlos

Maio/2011

Ficha catalográfica elaborada pela Biblioteca Prof. Achille Bassi
e Seção Técnica de Informática, ICMC/USP,
com os dados fornecidos pelo(a) autor(a)

d436r

de Toledo Pereira, Fabio
Reengenharia e evolução do visualizador e modelador do sistema Freeflow3D / Fabio de Toledo Pereira; orientador Antonio Castelo Filho -- São Carlos, 2011.
116 p.

Tese (Doutorado - Programa de Pós-Graduação em Ciências de Computação e Matemática Computacional) -- Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, 2011.

1. Computação Gráfica. 2. Engenharia de Software. 3. Mecânica de Fluidos. I. Castelo Filho, Antonio, orient. II. Título.

AGRADECIMENTOS

Acima de tudo, gostaria de agradecer a Deus, quem me guia e fortalece, por ser fiel e estar sempre ao meu lado durante todo este período, sendo abrigo sempre que preciso e dando forças para seguir adiante. A Ele seja dada toda a honra, a glória e o louvor.

Agradeço à minha esposa Laila, que me apoiou na decisão de concorrer ao programa de mestrado, me incentivou durante todos esses anos, parte deles à distância, tendo por vezes abdicado de seu tempo em detrimento desse trabalho. Seu amor, seu carinho e sua compreensão são motivadores em tudo o que eu faço e foram decisivos para que os resultados deste trabalho fossem alçados.

A toda a minha família, e em especial aos meus pais Ariovaldo e Mônica, pelo amor, pela paciência, pela formação como pessoa e pelo apoio de sempre, principalmente durante os anos em que vivi na distante São Carlos.

Ao Prof. Dr. Antonio Castelo Filho, por acreditar em mim, pela atenção e pelo apoio durante o processo de definição e orientação.

À Universidade de São Paulo, onde construí minha vida acadêmica, pela oportunidade de realização do curso de mestrado.

À CI&T e à CPM Braxis Capgemini, empresas nas quais trabalhei durante o mestrado, a todos os meus colegas e em especial aos meus coordenadores Rodrigo Ribeiro, Paulo Morias e Daniel Porto, pelo apoio e compreensão durante a elaboração desta tese.

Aos meus amigos, que sempre me apoiaram no percurso do mestrado. Entre eles, gostaria de agradecer em especial aos meus amigos Gabriel Andery, Fabio Freire e Marcel Tanaka, que, além do contínuo apoio, ajudaram ativamente na conclusão deste trabalho.

RESUMO

PEREIRA, F. de T. **Reengenharia e evolução do visualizador e modelador do *Freeflow3D*.** 2011. 116 p. Dissertação (Mestrado) – Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, São Paulo, 2011.

Programas para simulação numérica de escoamento de fluido são inerentemente aplicações complexas que envolvem múltiplas disciplinas de ciências de computação, matemática e engenharia. Normalmente, além do simulador, faz parte do ambiente de simulação os modeladores (pré-processamento) e o visualizadores (pós-processamento). O *Freeflow3D* é um desses ambientes, desenvolvido pelo laboratório LCAD (Laboratório de Computação de Alto Desempenho) do ICMC (Instituto de Ciências Matemáticas e de Computação) da Universidade de São Paulo-USP. O simulador é bastante ativo com vários pesquisadores desenvolvendo ao mesmo tempo. Contudo, o visualizador e o modelador são praticamente os mesmos desde a criação do programa anos atrás, principalmente porque a estrutura desses programas dificultou suas evoluções e melhorias ao longo do tempo. Este trabalho propõe uma análise do modelador e visualizador do *Freeflow3D* a fim de executar uma reengenharia no *software*, propondo uma nova estrutura que promova uma mudança em sua dinâmica de funcionamento, modernizando-o, deixando-o principalmente mais flexível em relação a futuras evoluções e manutenções como também adicionando novas funcionalidades que promovam uma melhora na interface com o usuário.

ABSTRACT

PEREIRA, F. de T. **Reengineering and evolution of *Freeflow3D* viewer and modeler.** 2011. 116 p. Dissertação (Mestrado) – Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, São Paulo, 2011.

Programs for numerical simulation of fluid flow are inherently complex applications that involve multiple disciplines of computer science, mathematics and engineering. Usually, the simulation suite consists of the simulator itself, modelers (preprocessing) and viewers (post processing). The Freeflow3D is one of those environments, developed by the LCAD lab (Laboratório de Computação de Alto Desempenho) of the ICMC (Instituto de Ciências Matemáticas e de Computação), USP (Universidade de São Paulo). The simulator development is quite active with several researchers working on it. However, its viewer and modeler are virtually the same since their creation years ago, mainly because their evolution and maintenance over time have been hampered by its structure. This paper proposes an analysis of the Freeflow3D modeler and viewer to implement a software reengineering, proposing a new structure that changes their dynamics, modernizing their structure and making they particularly flexible for future developments and maintenance as well as adding new features that promote an improved user interface.

LISTA DE FIGURAS

LISTA DE TABELAS

SUMÁRIO

1. Introdução

Atualmente, é notável o uso do computador na resolução de problemas e no auxílio de pesquisas para novas descobertas nas mais diversas áreas do conhecimento. Uma das áreas da computação que tem constantemente contribuído para esse fenômeno é a de simulação. Através de algoritmos de simulação e considerável poder de computação (cada vez mais disponível por custos mais baixos), um pesquisador pode ter total controle sobre o seu experimento, podendo coletar dados e propriedades que dificilmente seriam obtidos através da utilização de apenas medições e experimentos reais, isso de forma ágil e cada vez mais precisa (devido ao avanço de *hardware* e dos algoritmos de simulação).

Dentre as diversas subáreas da simulação computacional, uma das mais importantes é a de simulação de escoamento de fluido. As aplicações de simulação de escoamento de fluido são inúmeras, pode-se simular, por exemplo, todo um reservatório de água e seu comportamento através do tempo (inclusive sob condições raras e anormais), além de coletar dados individualmente, em cada intervalo de tempo, sobre propriedades do fluido como pressão, velocidade e viscosidade.

Apenas para exemplificar a utilidade deste tipo de informação no nosso cotidiano, pode-se dizer que com os dados de pressão obtidos com a simulação computacional é possível dimensionar melhor as paredes de um reservatório de água real e fortificá-las em áreas onde é exercida maior pressão, o que pode proporcionar maior segurança na construção e manutenção destes reservatórios.

Uma das soluções para simulação de fluido desenvolvida no Laboratório de Computação de Alto Desempenho do Instituto de Ciências Matemáticas e Computação-ICMC da Universidade de São Paulo-USP (LCAD) é denominada *Freeflow3D* [1]. O *Freeflow3D* é um ambiente de aplicações completa que além do simulador numérico, conta com um modelador (de cenário inicial para simulação) e um visualizador (para visualização dos resultados da simulação).

O presente trabalho propõe uma reengenharia do *Freeflow3D*, porém apenas dos módulos modelador e visualizador do *Freeflow3D*, sem que haja qualquer alteração no seu simulador. Assim, sempre que for utilizada a expressão “reestruturação do *Freeflow3D*”

neste trabalho, tal reestruturação estará se referindo apenas à reestruturação do modelador e do visualizador do *Freeflow3D*.

A reestruturação do *Freeflow3D* tem o intuito de agregar novas funcionalidades ao ambiente e modernizar as funcionalidades já existentes. Como será visto no capítulo 2, o modelador e o visualizador do *Freeflow3D* possuem certas limitações e as mudanças necessárias nos programas para corrigi-las estão cada vez mais complexas e custosas de serem executadas, dificultando o uso e a evolução do *Freeflow3D*. Para tanto, o presente trabalho propõe uma nova estrutura para esses módulos de modelagem e visualização, apresentando uma nova arquitetura baseada nos conceitos de *componentes e fluxos de execução* (cuja teoria será abordada no capítulo 4 e sua aplicação no capitulo 5), que garante um método organizado e bem definido para evoluções e novas funcionalidades, além de proporcionar uma nova abordagem para a interface com o usuário.

A estrutura do presente trabalho foi montada basicamente em duas grandes partes, a primeira é teórica e visa trazer os conceitos mais importantes para a compreensão de como a nova arquitetura usada na reestruturação do *Freeflow3D* foi concebida. Terminada esta primeira parte mais conceitual, o trabalho segue para a parte empírica, na qual se descreve como foi feita a reestruturação do *Freeflow3D*, desde a sua concepção até a aplicação da nova arquitetura, aplicando-se os conceitos descritos neste trabalho até tal momento.

No primeiro capítulo do presente trabalho, o escopo maior foi introduzir o assunto descrito nos parágrafos acima, assim como apresentar a motivação e os objetivos do trabalho. O capítulo 2, por sua vez, apresenta o *Freeflow3D* propriamente dito, definindo suas características. Já o capítulo 3, introduz a computação 3D em tempo real, a aceleração através das placas gráficas e o *Opengl*, além de alguns conceitos que serão utilizados na reestruturação, melhoria e expansão das funcionalidades do *Freeflow3D*.

No capítulo 4 serão apresentados os conceitos de Engenharia de *Software* que nortearam a reestruturação do *software* em estudo neste trabalho. Há a introdução da matéria acerca da reengenharia de *software* e seus objetivos, além do estabelecimento de uma série de atividades das quais algumas delas são pertinentes à reestruturação do *software*. O conceito de engenharia de *software* baseado em componentes também é

apresentado nesse capítulo, conceito este que é importante para o entendimento da concepção e aplicação de uma nova arquitetura aos aplicativos do *Freeflow3D*.

Para tratar da parte mais prática do trabalho há o capítulo 5 relatando com mais detalhes o trabalho desenvolvido na reestruturação do *Freeflow3D*, atendo-se principalmente à aplicação dos conceitos computacionais apresentados nos capítulos anteriores na concepção da nova arquitetura e sua aplicação visando o principal objetivo do trabalho. Por fim, o capítulo 6 apresenta os resultados obtidos ao final do desenvolvimento do presente trabalho, e o capítulo 7 dispõe sobre a conclusão e o futuro do *Freeflow3D* após a sua reestruturação.

2. Freeflow

2.1. Sobre o ambiente de aplicações Freeflow3D

O Freeflow3D consiste em um ambiente integrado de simulação de escoamento tridimensional de fluido com superfícies livres.

O conjunto de *software*s que compõem o *Freeflow3D* forma um ambiente completo de simulação de escoamento de fluido. Faz parte desse ambiente um modelador tridimensional de cenário inicial para simulação (*Modflow3D*), um simulador numérico de escoamento de fluido (*Simflow3D*) e um visualizador dos resultados das simulações (*Visflow3D*). Contudo, somente o modelador e o visualizador usam o módulo gráfico do sistema, e, devido a esta propriedade, serão esses os módulos tratados a seguir.

O *Modflow3D* é o módulo interativo do *Freeflow3D* no qual se criam os objetos que participarão da simulação e no qual se estabelecem os parâmetros inicias da simulação. Através da interface gráfica, o usuário do *Modflow3D* pode criar os recipientes e os bicos injetores de fluido por meio do uso e de instanciamento de primitivas (caixa retangular aberta trapezoidal; caixa retangular fechada; caixa retangular com obstáculos retangulares; e retângulo arredondado vazado, entre outras).

O usuário também utiliza a interface gráfica do *Modflow3D* para estabelecer os dados iniciais do cenário de simulação, tais como: posição inicial, velocidade, viscosidade, pressão, tipo, etc.

Já o *Visflow3D* é o modulo interativo pelo qual são visualizados os dados resultantes da simulação efetuada pelo *Simflow3D*. Os comportamentos dos recipientes, fluidos, e entrada e saída de fluido, criados no *Modflow3D* e simulados no *Simflow3D,* são visualizados no *Visflow3D*.

Portanto, a função primordial do *Visflow3D* é criar uma forma para melhor visualizar a simulação produzida para o usuário e as propriedades do fluido, o que é feito através de uma textura colorida calculada utilizando os dados fornecidos pelo simulador.

Em ambos os módulos, *Modflow3D* e *Visflow3D,* a visualização do espaço em três dimensões é fundamental. Só é possível modelar os objetos que farão parte da simulação se estes puderem ser vistos em 3D e se transformações gráficas e geométricas que facilitem o processo forem utilizadas. Os resultados das simulações numéricas também dependem da manipulação da representação 3D através do visualizador para serem compreendidos em sua totalidade.

Tanto no *Modflow3D* como no *Visflow3D*, usam-se algumas propriedades básicas de um módulo gráfico 3D, ou seja, é possível rotacionar, transladar e redimensionar a câmera, mudar seus parâmetros (alterar sua distância, seu tipo de projeção - ortogonal ou perspectiva - etc.), alterar os parâmetros do *rendering* (*WireFrame, Flat, Phong* ou *Gouraud shading*), mudar características da iluminação (escolhendo entre iluminação direta ou ambiente), mudar a colorização dos objetos (cor e transparência – Sistema *Red Green Blue Alpha* - RGBA), entre outras funcionalidades básicas.

Além das funções descritas acima, o módulo visualizador do *Freeflow3D* usa outras técnicas de visualização, a texturização do fluido, através de uma escala de cores correspondente às suas características de pressão e velocidades, é um exemplo, e a possibilidade de prover cortes paralelos nos eixos principais (x, y, z) da cena, para melhor visualizar as características internas do fluido, é outro bom exemplo.

Todas essas funcionalidades do módulo gráfico do ambiente *Freeflow3D* são obtidas através do uso da biblioteca *Xview* do sistema de janelas *XWindows* do sistema *Unix/Linux*. O *render* do *Freeflow3D* é desenvolvido *in house*, ou seja, não faz uso de placas gráficas e implementa o *rendering* via *software*, aplicando os algoritmos de *scan-line*[2] e colorização *half-tone*[3].

Ambas as soluções, modelador e visualizador, são atualmente defasadas, tanto em desempenho como em resultado visual. É objetivo deste trabalho reestruturar esses programas, adicionando e modernizando suas funcionalidades, bem como propondo uma nova estrutura para facilitar a utilização, manutenção e evolução dos referidos programas. No item 5.1, segue a análise dos programas legados e de suas limitações e, a partir disso, ao longo dos itens 5.2 e 5.3 é descrito todo o processo de reestruturação que visa superar as limitações encontradas nos programas legados.

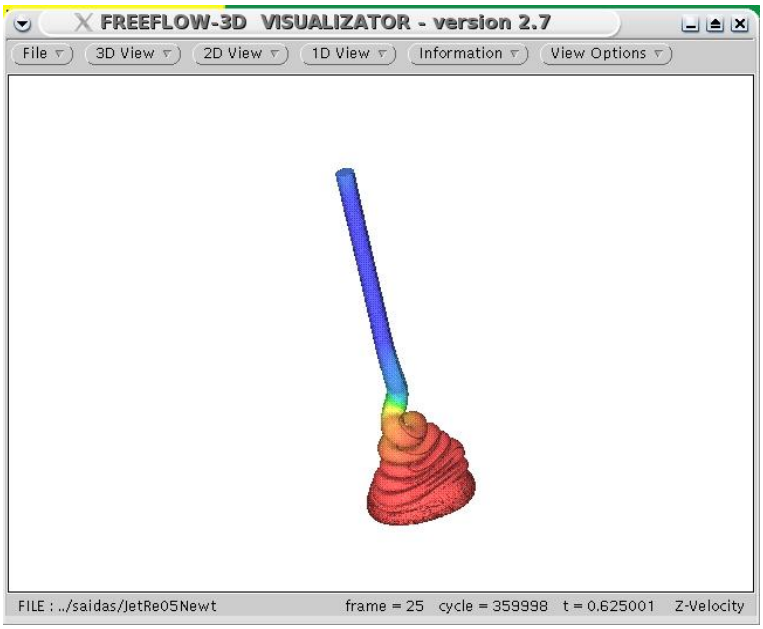


Figura 1: Visualizador do *Freeflow3D* antes da reestruturação realizada no presente trabalho.

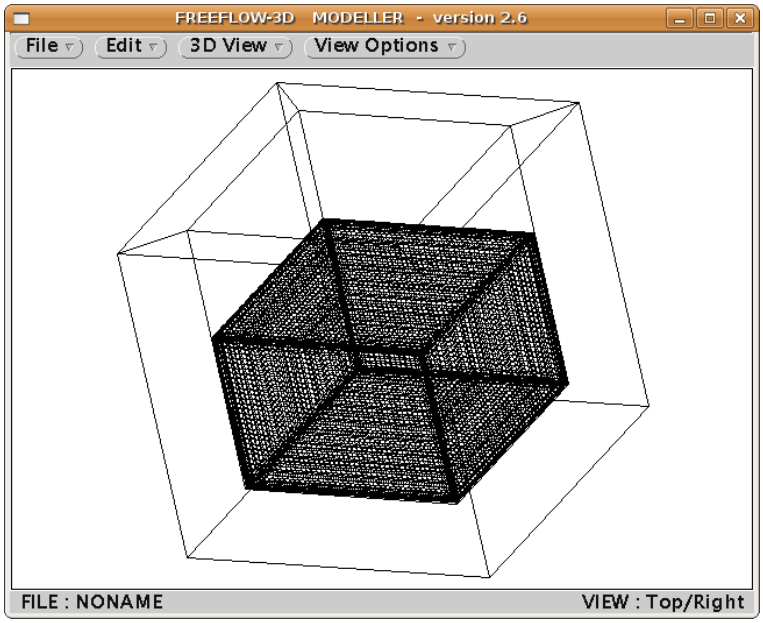


Figura 2: Modelador do *Freeflow3D* antes da reestruturação realizada no presente trabalho.

3. Computação Gráfica 3D

Para melhor compreensão do presente trabalho e da reestruturação realizada no *Freeflow3D* é importante fazer uma breve explanação sobre a Computação Gráfica 3D e seus conceitos utilizados no desenvolvimento do problema.

3.1. Sobre a Computação Gráfica 3D

A definição mais comum de Computação Gráfica é "conjunto de métodos e técnicas para transformar dados em imagens através de um dispositivo gráfico", por Jonas Gomes e Luiz Velho [3]. Contudo, ainda na opinião destes autores, o estudo da Computação Gráfica abrange muito mais do que transformar dados em imagens, podendo ser dividida em quatro áreas distintas, conforme segue:

- **Visão** – Também conhecida como "Análise de Imagem", tem como objetivo obter informações através da análise de uma ou mais imagens. Normalmente, algoritmos de visão combinam o uso de algoritmos de processamento de imagens para extrair delas dados que podem ser usados para extração de informação, como por exemplo, uma descrição das imagens. É uma área de grande interesse para a robótica, que estuda meios de interpretar as imagens captadas pelas câmeras dos robôs e gerar dados de entrada para os algoritmos de inteligência artificial.

- **Modelagem Geométrica** – É a área da Computação Gráfica que tem a preocupação de descrever e estruturar os dados geométricos, através de primitivas geométricas e de uma estrutura de dados que as definam e

mantenham as relações internas entre essas primitivas. Exemplos de programas modeladores geométricos são o *Blender* e o *3D Max*.

- **Processamento de Imagens** – O objetivo da área consiste no estudo de um algoritmo que processará uma imagem de entrada e gerará uma imagem de saída transformada, normalmente utilizada para corrigir a imagem de entrada ou realçar algum detalhe para posterior extração de informações. São exemplos de mecanismos de processamento de imagens: os filtros de detecção de borda, a escala de cinza, o ajuste de contraste, entre outros.

- **Visualização** – Também conhecida como "síntese de imagem", a área é responsável por sintetizar uma imagem através de uma descrição geométrica que pode ser visualizada por um dispositivo de saída.

No presente trabalho, a área da Computação Gráfica em questão é a última das descritas acima, a de Visualização, ou seja, a de "síntese de imagens" que, como visto, é a que se atém na geração de uma imagem a partir de uma descrição geométrica, e esta última pode ser duas ou três dimensões.

No caso do modelador e visualizador do *Freeflow3D*, as descrições geométricas dos fluidos, domínios, etc., normalmente são realizadas em 3D e, portanto, adentrando em uma área ainda mais específica da Computação Gráfica, tal fato torna as técnicas e conceitos da área de Computação Gráfica 3D de grande relevância para este trabalho.

Depois de conceituada a Computação Gráfica em sua esfera mais abrangente, cabe tratar da Computação Gráfica 3D, que consiste na síntese de imagens a partir de descrições tridimensionais de cenas. Ocorre que as primitivas geométricas passam por uma série de transformações geométricas para formarem uma imagem 2D, além de ser combinadas com algoritmos de iluminação e de coloração, que podem ser não realísticos ou, ainda, fisicamente inspirados, gerando uma cena foto-realística.

São alguns exemplos da aplicação da Computação Gráfica, em especial a Computação Gráfica 3D, os seguintes:

- visualização de protótipos de objetos antes de sua concepção;

- análise e visualização de simulações físicas (e dos dados relacionados à simulação, tais como velocidade, entre outras propriedades); e

- utilização em aplicações de realidade virtual, simulando nosso ambiente e situações para fins de entretenimento, treinamento, etc.

No âmbito da Computação Gráfica 3D as imagens são sintetizadas a partir de descrições de cenas em 3D e existem diversas técnicas de síntese de imagens que diferem no custo computacional, no propósito, na qualidade gráfica, etc. O intuito dos programas do *Freeflow3D* é gerar imagens em tempo real, ou seja, em taxas interativas. Dentro dessa categoria têm-se os algoritmos de rasterização e, atualmente, devido ao grande avanço do poder computacional, os algoritmos de traçado de raios.

O traçado de raios consiste, basicamente, em simular a interação dos raios de luz com os objetos até que esses encontrem a câmera, formando a imagem. Para isso, são lançados, para cada *pixel* que compõe a imagem, um ou mais raios com origem na câmera e em direção à cena. Esses raios percorrem a cena e, para cada objeto que este intersecta, é calculada a contribuição de cor que essa intersecção dará ao *pixel*, levando em conta as fontes de luz da cena. Não obstante, dependendo das características do objeto, como translucidez, coeficiente de reflexão e topologia da superfície, novos raios serão lançados a partir do objeto com uma energia menor, sendo que suas subseqüentes intersecções com a cena também contribuirão para estabelecer o valor final do *pixel* (contribuição essa limitada pela energia do raio) [4].

O método descrito acima é classificado como método de iluminação global, pois segundo Jonas Gomes e Luiz Velho, a aproximação da função de iluminação “corresponde a contribuição direta das fontes de luz e indireta da reflexão das superfícies”. Essa abordagem reproduz efeitos como reflexão, refração de objetos transparentes, sombras e remoção de superfícies encobertas (ou “não visíveis” na imagem final), o que não ocorre com a abordagem de rasterização. Embora a abordagem do método de iluminação global não seja nativamente suportada e acelerada nas placas gráficas atuais, as placas mais novas

proporcionam ao usuário a oportunidade de reprogramá-la, o que já disponibiliza implementações de traçado de raios acelerados por esse tipo de *hardware* dedicado com processamento altamente paralelizado.

Com a rasterização, diferentemente do que ocorre com o método de iluminação global descrito acima, o algoritmo gerará, a partir da descrição da cena 3D e das informações como a projeção e a câmera, uma imagem em 2D representando a cena desejada e para cada *pixel* da imagem gerada é calculada a sua função de colorização, sempre obedecendo as fontes de luz e a função de iluminação da cena.

Geralmente, o algoritmo trata do preenchimento de polígonos e pode ser codificado levando em conta a ordem de objetos ou a ordem das linhas da imagem. A abordagem baseada na ordem das linhas da imagem (também conhecida por rasterização por linhas – *scanline rendering*) é a mais comum e gera a imagem processando as linhas horizontais da imagem, uma a uma, seguindo uma estrutura de dados descritiva dos polígonos da cena, para que, depois de calculados os valores de uma linha, cada *pixel* calculado seja processado executando sua função de colorização.

A *scanline rendering* tem vantagens em sua implementação em *hardware* e nas operações de *antialiasing*, pois o processamento das linhas pode ser altamente paralelizado, assim como as operações de colorização por *pixel* (*per pixel*)[4]. Diferentemente do algoritmo de traçado de raios, é preciso que técnicas auxiliares sejam usadas para implementar alguns efeitos gerados nativamente por ele, como algoritmos para: remoção de superfícies encobertas (ex.: *Z buffer* [4]), sombras (ex.: sombras volumétricas [4]), reflexão (ex.: mapeamento cúbico [4]), dentre outros. Por fim, o processamento da função de colorização *per pixel* classifica o método como situado no campo da iluminação local, pois somente a interação local com as fontes de luz é capaz de contribuir para o cálculo da cor do *pixel*. [3]

As placas aceleradoras gráficas 3D, criadas para acelerar o processo de síntese de imagens de descrição de cenas 3D em taxas interativas, implementam nativamente a abordagem de rasterização, mais especificamente a *scan-line*[2]. Para acesso às funções da placas gráficas, duas principais *API*s (*Application Programming Interfaces*), o *Opengl* e o *DirectX,* são disponibilizadas ao desenvolvedor, sendo que, no presente trabalho, a *API*

utilizada é o *Opengl*, principalmente em razão de ser uma solução *cross-plataform*. O *Opengl* será assunto do próximo item do presente trabalho.

3.2. Opengl

O *Opengl[*2] foi uma das primeiras *API*s gráficas criadas, com o intuito de desenvolver aplicações 2D ou 3D, portáveis a qualquer sistema. Nascido no início da década de 90, dentro dos laboratórios da *Silicon Graphics*, hoje é o sistema para criação de aplicações de computação gráfica com o auxílio de placas gráficas mais difundido, acessível e suportado no mercado.

Por ter acesso direto ao *hardware* gráfico, o *Opengl* ("*OGL"*), suportado atualmente por praticamente todas as fabricantes de placas de vídeo para computadores, se torna independente de qualquer sistema operacional ou linguagem de programação. Assim, o uso do *Opengl* é o mesmo nas mais diversas linguagens de programação (é possível utilizá-lo a partir da linguagem *Ada, C, C++, Fortran, Python, Perl* e *Java*) e de sistemas operacionais (dentre os sistemas compatíveis com o *OGL*: todos os sistemas Unix/*Linux*, *Windows*, *Mac OS, OS/2* e *BeOS*).

Além disso, o progresso da linguagem *Opengl* é cuidadosamente acompanhado pela indústria da informática e tem seu uso livre, características estas que auxiliaram na sua popularização. Atualmente, quem administra sua evolução é o consórcio *Opengl ARB* (A*rchitecture Review Board*), que foi criado em 1992 e que possui como membros as principais empresas fabricantes de *softwares* e de *hardware* gráfico (são membros do consórcio *nVidia, ATI, Microsoft, IBM, SUN, Silicon Graphics*, etc.).

Assim, a *OGL ARB* visa preservar a estrutura básica da linguagem (o que garante a retro-compatibilidade com os códigos escritos sobre versões prévias do *OGL*), fazendo também análises periódicas para possíveis melhorias (provenientes principalmente do avanço do *hardware* gráfico). As melhorias, depois de aprovadas pelos membros do consórcio, são incorporadas ao *OGL* através de bibliotecas de extensão, independentes da

versão original da linguagem, garantindo contínuo desenvolvimento da linguagem sem a necessidade de modificação de códigos escritos sobre suas versões obsoletas.

Sendo uma *API* de uso livre, não existem requerimentos de licença para os usuários finais ou para as empresas de criação de *software*, ademais, o "uso livre" significa que os códigos-fonte escritos em *OGL*, seu código-fonte e as suas especificações são disponibilizados pelo *OGL ARB* para as empresas licenciadas a usar essas informações na criação do suporte necessário para seus novos *hardware*s.

Na reestruturação dos programas do *Freeflow3D*, foi utilizada a *API* Opengl, pois, além de acrescentar um maior desempenho de rasterização, devido à aceleração feita em *hardware*, o fato de a *API* ser livre se coaduna com um dos objetivos do presente trabalho, que é o de tornar o *software* multiplataforma.

3.3. Renderer e Gerenciador de Cena

O *renderer* e o gerenciador de cena são dois módulos comuns aos visualizadores em geral O gerenciador de cena tem a função de organizar a cena internamente, isso é, organizar cada elemento que a compõe, suas inter-relações e seus atributos, disponibilizando uma interface de acesso a ela. Já o render acessa os objetos da mesma através da interface do gerenciador de cena e os percorre, desenhando-os Ambos os módulos podem ser implementados de diversas maneiras, com diferentes ganhos em produtividade (melhor organização da cena) e desempenho (*rendering* inteligente). Passa-se agora a um breve relato acerca dos referidos *render* e gerenciador de cena que foram implementados durante o presente trabalho.

- **Renderer**

O *renderer* é o módulo responsável pelos cálculos gráficos necessários para transformar os dados concebidos em 3D em uma imagem plana, ou seja, de duas dimensões

("2D"), que será impressa no monitor. É sua função gerenciar os modelos de iluminação, de luzes dinâmicas existentes, de propriedades da câmera, etc.

A implementação do *render* em tempo real é geralmente realizada de duas maneiras: por rasterização (varredura das linhas de visualização, o que privilegia aplicações em tempo real) e traçado de raio (que, de outro lado, privilegia o realismo da cena), como visto anteriormente no item 3.1. No presente trabalho, será desenvolvido um *renderer* baseado em rasterização.

- **Gerenciador de Cena**

O gerenciador de cena é o módulo responsável por organizar os objetos a serem visualizados na cena. É a partir dele que se tem acesso aos objetos, às transformações no espaço, e é nele que a cena é organizada, a fim de que o *render* transforme-a em uma imagem final, a ser exibida para o usuário.

Algumas funcionalidades da computação gráfica 3D só são possíveis de ser implementadas caso o gerenciador de cena disponibilize informações sobre como a cena está organizada, através de controles de posicionamento no espaço, estado dos objetos, etc. Além disso, através de algoritmos de gerenciamento de cenas é possível implementar algoritmos de otimização de *rendering*, como por exemplo o algoritmo de *Frustrum Culling*[5].

3.4. Materiais

O termo "material", no âmbito da Computação Gráfica, define um conjunto de parâmetros que descreverá as características da superfície do objeto virtual e que, em contato com a iluminação, darão a ele o aspecto similar ao objeto real que se pretende simular. Esses parâmetros, por exemplo, podem ser a cor do objeto, ou os coeficientes de reflexão da luz, de refração (caso o objeto seja transparente), de rugosidade, entre outros.

Segundo Tomas Akenine-Moller, em seu trabalho denominado *Real Time Rendering*[6], em um sistema de tempo real, os parâmetros para material são os coeficientes difuso, ambiente, especular, brilho e emissão. A cor da superfície é determinada por esses parâmetros, ou melhor, pelos parâmetros das fontes de luz e pelo modelo de iluminação.

O modelo de iluminação da placa gráfica utilizará esses parâmetros básicos do material para calcular a cor do *pixel* correspondente. As placas gráficas implementam nativamente os modelos de iluminação *flat* e *gourand*. O escopo deste trabalho não permite nos aprofundarmos na descrição dos modelos de iluminação *flat* e *gourand* , porém detalhes sobre esses dois modelos de iluminação e como os parâmetros dos materiais influenciam-nos podem ser encontrados no *Opengl Programming Guide* [7] e na já citada obra *Real Time Rendering* [6].

Resumidamente, é possível apontar que ambos os modelos são aproximações das leis físicas de iluminação, mas que possuem várias limitações, principalmente quanto ao realismo da imagem gerada. Para aumentar o realismo, é possível lançar mão de duas funcionalidades das placas gráficas: o mapeamento de texturas e os programas *shaders,* que serão objeto dos próximos itens do presente trabalho.

3.4.1. Texturas

O mapeamento de textura é um método para adicionar detalhes, textura ou cor a uma superfície. A textura pode ser aplicada em uma imagem 2D ou 3D, em uma função ou ainda em outra estrutura de dados. O principal objetivo desta técnica é adicionar detalhes às superfícies dos modelos 3D, gerando um maior realismo na representação das imagens, sem ter que recorrer a uma modelagem mais complexa. Logo, o uso de texturas pode gerar ganhos em desempenho, em economia de memória e na consecução de uma melhor modelagem.

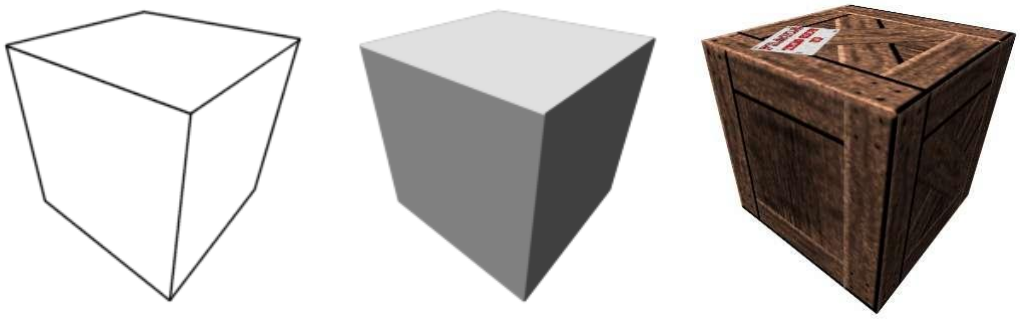


Figura 3: Mapeamento de Textura – À esquerda, a geometria simplificada do objeto. Ao centro, o objeto preenchido sem textura. À direita, o objeto texturizado. Disponível em: <http://homepages.paradise.net.nz/youp/tut/3dtex.html>. Acesso: 03 de março de 2008.

O exemplo clássico do uso eficiente das texturas está na representação de um muro de tijolos. Ao invés de modelar tijolo por tijolo, o que geraria a necessidade da criação de um alto número de polígonos, usa-se somente um polígono, no caso do muro de tijolos apenas um polígono plano, e mapeia-se sobre ele uma imagem de um muro. O processamento de um simples polígono, um plano, mesmo levando em conta o mapeamento de textura, é incomparavelmente mais rápido que o processamento de um modelo altamente detalhado, com muitos polígonos (um para cada tijolo do muro), além de economizar uma grande parcela de memória da placa gráfica.

O mapeamento de textura usado para adicionar uma imagem à superfície do objeto 3D é a função mais básica da texturização e é nativamente suportada pelas placas gráficas e pelas suas *APIs*, como o *Opengl* [7], além de outras funcionalidades, como o uso de multitexturas combinado com transparência, e com os filtros e os níveis de detalhe.

Figura 4: Exemplo de Multitexturização – Do lado esquerdo, as duas texturas bases; do lado direito, ambas as texturas aplicadas sobre o mesmo modelo, portanto sobrepostas. Disponível em: <http://www.lighthouse3d.com/Opengl/glsl/index.php?textureMulti>. Acesso em 03 de março de 2008.

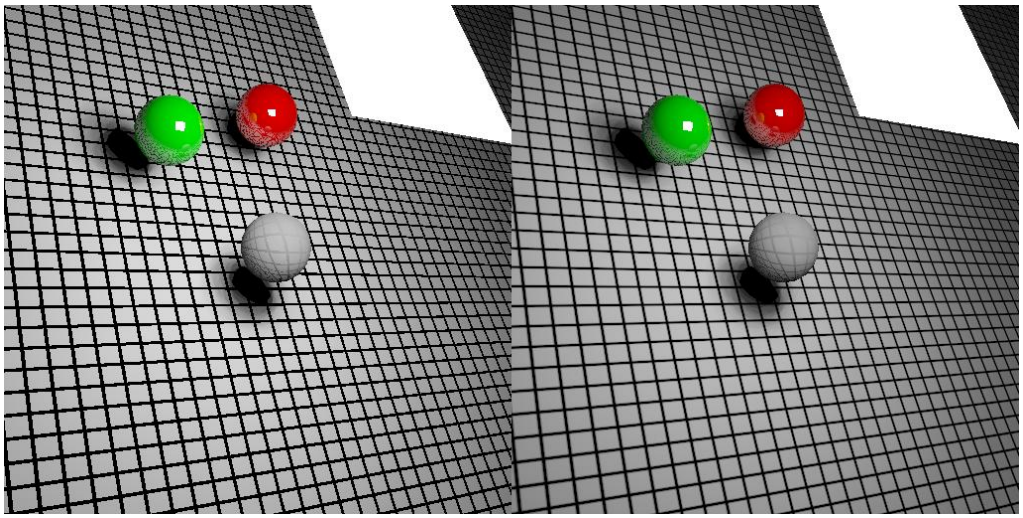


Figura 5: Exemplo de filtro de suavização de textura – Imagem à esquerda sem o filtro e imagem à direita com o filtro aplicado. Disponível em <http://voxelizator3d.files.wordpress.com/2006/10/tex.PNG>. Acesso em 03 de março de 2008.

Além de armazenar informação de cor mapeada para a superfície do objeto 3D, a textura pode ser usada também para armazenar outro tipo de informação sobre o objeto, como por exemplo, seus vetores normais. Embora o modelo de iluminação básico das placas gráficas somente utilize a informação da textura para compor a cor do objeto, é possível alterar o modelo de iluminação através de programas de *rendering* programável (“*shaders”)* para utilizar outros tipos de informações armazenados na textura. Para melhor compreensão, os programas *shaders* serão objeto do item seguinte.

3.4.2. *Shaders*

O *rendering* programável (ou programas *shaders*) é uma opção oferecida pelas placas gráficas mais recentes no mercado para alterar algumas etapas da *pipeline* padrão da placa gráfica.

É possível escrever programas para o processador de vértice (*vertex processor* e seus programas conhecidos como *vertex shaders*), para o processador de fragmentos (*fragment processor*, e seus programas conhecidos como *pixel shaders*) e, nas placas mais avançadas,

também para o processador geométrico (*geometry processor*). Além disso, são disponibilizadas ao desenvolvedor algumas linguagens específicas para auxiliar na criação dos programas de *shading* (ex.: GLSL [8], Cg[9], etc.), substituindo a linguagem *assembly*, usada nas primeiras placas com o recurso de *rendering* programável.

Segue abaixo algumas características dos principais processadores gráficos, o *vertex* e o *fragment processor*[10], citados nos parágrafos anteriores:

- O **Processador de Vértice** (***Vertex Processor*)** é capaz de atuar no cálculo da iluminação, do material e da geometria e pode substituir partes da *pipeline* fixa do *Opengl*, como as transformações do vértice, das normais, da iluminação, da normalização, da escala, da aplicação de material, da transformação e da geração de coordenadas de textura.

- O **Processador de Fragmentos (*Fragment Processor*)** é capaz de acessar a textura, e interpolar e operar sobre o *pixel*. Adicionalmente, o referido programa pode substituir a *pipeline* fixa do *Opengl* nas seguintes funcionalidades: operações sobre os dados interpolados de vértices, zoom do *pixel*, acesso à textura, aplicação da textura, aplicação de neblina (*fog*), convolução, entre outras.

O exemplo mais comum de aplicação prática dos programas *shaders* pode ser encontrado na substituição do modelo de iluminação padrão da placa gráfica por modelos mais avançados. Por padrão, a placa gráfica calcula a iluminação dos objetos seguindo os modelos *Flat* e *Gouraud shading* [7], que são modelos simples. Alterando as unidades de *pixel* e *vertex shading* é possível utilizar modelos mais avançados de iluminação, como *Phong shading* [11], além de aplicar efeitos gráficos, como: reflexo (*cube mapping* - mapeamento cúbico [4]), otimizações (*normal-mapping* [6]), entre outros.

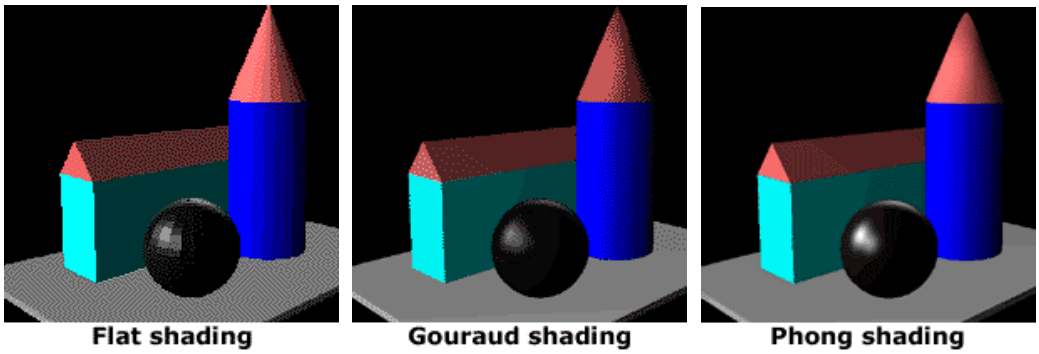


Figura 6: Comparação entre os modelos de iluminação – Comparação entre os modelos de iluminação *Flat*, *Gouraud* e *Phong shading*, nesta ordem. Disponível em: <http://www.meko.co.uk/shading.shtml>. Acesso em 1º de novembro de 2008.

4. Engenharia de Software

Este capítulo apresenta os conceitos de Engenharia de *Software* estudados para a execução da reestruturação do *Freeflow3D*. O item 4.1 aborda o ramo da reengenharia de *software* e descreve os conceitos que nortearão a reestruturação dos programas legados. Já o item 4.2 apresenta os conceitos de Engenharia de *Software* Baseada em Componentes, conceito que terá papel importante na definição da arquitetura de referência utilizada pelos novos visualizador e modelador do *Freeflow3D*.

4.1. Reengenharia de Software

Segundo a definição da IEEE [12], engenharia de *software* é "(1) aplicação de uma abordagem sistemática, disciplinada e quantificável, para desenvolvimento, operação e manutenção do software, isto é a aplicação da engenharia ao software. (2) Os estudos de abordagens como as de (1)."

Logo, da leitura acima, engenharia de *software* pode ser conceituada como o conjunto de abordagens bem definidas aplicadas sobre o desenvolvimento, a operação e a manutenção de um *software*.

Contudo, um sistema, independentemente do domínio ou da tecnologia utilizada, inevitavelmente sofre erosão em sua estrutura com o passar do tempo, principalmente devido aos seguintes fatores: constantes evoluções não controladas, falta de documentação, desvio da arquitetura original, entre outros. Ainda que um programa tenha sido criado seguindo as melhores práticas e os melhores conceitos da engenharia de *software*, há sempre o risco da referida erosão, ou seja, dessa degradação da estrutura original que faz com que as manutenções subseqüentes se tornem cada vez mais dispendiosas, caras e inviáveis [13].

Quando o *software* atinge um estado de degradação que impede sua manutenção e expansão seguindo uma arquitetura bem definida, e tal *software* realiza tarefas relevantes, portanto, tendo sua necessariedade comprovada, então é preciso que ocorra sua reengenharia para voltar a tornar possível sua manutenção e expansão.

Reengenharia de *software*, segundo Preesman [14], é o processo de recriar um *software*, ou apenas partes dele, sendo que é realizado quando tal *software* é usado regularmente, porém possui erros constantes e de difícil ou demorada manutenção, para que, após a recriação, o *software* passe a ter novas funcionalidades, melhor desempenho, maior confiabilidade e manutenção melhorada.

Logo, os principais objetivos da reengenharia de *software*, segundo Sneed[15], são:

- melhorar a manutenibilidade do *sofware*, isolando as suas funcionalidades em módulos melhores (processo denominado "componentização") e em interfaces bem definidas, por exemplo;

- realizar a migração da linguagem ou SO do *software*, mudando-os para uma linguagem ou SO mais novo ou mais eficiente;

- alcançar maior confiabilidade no sistema, corrigindo defeitos (*bugs)* que, após a reengenharia do programa, ficaram mais aparentes ao usuário; e

- preparar o sistema para sua eventual expansão, pois com a reengenharia do *software* em módulos menores, que proporciona interface bem definida e estruturas de dados normalizadas, é possível adicionar novas funcionalidades ou ainda incrementar as funcionalidades existentes com uma alteração local do código, portanto, tornando o procedimento de expansão mais simples do que antes da realização da reengenharia.

Presman[14], em sua obra supracitada, ainda define um modelo de processo de reengenharia com seis tipos de atividades, que podem ser executadas dependendo do objetivo do processo de reengenharia. São elas:

- **Análise de Inventário:** esta atividade realiza o levantamento dos dados da aplicação, tais como a idade, o tamanho, a criticidade do negócio, etc.

- **Reestruturação da Documentação:** cria um plano de ação para documentação do *software,* ou, ao menos, das suas partes críticas.

- **Engenharia Reversa:** cria uma abstração em alto nível do código fonte, com o fim de recuperar informações sobre o projeto de dados arquitetural e procedural da aplicação. Normalmente, a engenharia reversa é aplicada quando não há documentação suficiente para entender como o *software* funciona atualmente, servindo de início para a reestruturação do mesmo.

- **Reestruturação de Código:** também chamada de reengenharia do código, tem como função apontar partes do código de difícil compreensão, teste e manutenção, para que, posteriormente, sejam reescritos e documentados seguindo o padrão arquitetural do projeto. Pode-se usar um *software* de análise de código para aferir quais partes do código não seguem uma diretiva da qualidade pré-estabelecida.

- **Reestruturação de Dados:** foca na melhora da arquitetura de dados do programa, pois um programa com uma arquitetura de dados fraca é de difícil adaptação e expansão. Como a arquitetura de dados está fortemente ligada à arquitetura do *software* e à algoritmos que lidam com os dados, invariavelmente essa reestruturação acarretará em reestruturação de código e em mudanças arquiteturais.

- **Engenharia Progressiva:** é o mesmo processo de desenvolvimento de *software,* ou seja, transforma as abstrações de alto nível do programa em implementação física, porém, na engenharia progressiva a transformação é feita renovando o programa, ou seja, melhorando a sua qualidade e a sua performance, além de

possibilitar a adição de novas funcionalidades. Essa tarefa pode ser assistida por alguma ferramenta de reengenharia especializada no domínio do problema, que auxiliará no processo.

Algumas das técnicas descritas acima serão utilizadas para executar a reengenharia do modelador e visualizador do *Freeflow3D* e serão citadas ao longo do capítulo 5, que descreve o trabalho realizado.

4.1.1. Padrão de Projetos – auxílio na reestruturação de código

Para proporcionar um pano de fundo para o tema dos padrões de projetos, cabe citar Christopher Alexander [16] que diz que*: "Cada padrão descreve um problema recorrente em nosso ambiente, e então descreve o núcleo da solução daquele problema, de uma forma que você possa usar esta solução um milhão de vezes, sem nunca ter utilizado-a da mesma maneira."*

Diante da definição, pode-se afirmar que padrões de projeto, no âmbito das Ciências de Computação, descrevem soluções para problemas recorrentes no desenvolvimento de *software*s orientados a objetos. Desse modo, o uso desses padrões visa aplicar soluções consolidadas a problemas comuns, geralmente agilizando e simplificando o desenvolvimento de um *software*, além de prover a ele uma arquitetura mais organizada e documentada.

Um padrão de projeto, ainda segundo o autor Christopher Alexander [16], deve compreender as seguintes características:

- **Encapsulamento:** um padrão encapsula um problema ou solução bem definida. O padrão deve ser independente, específico e formulado de maneira a manifestar claramente em que ele se aplica.

- **Generalidade:** todo padrão deve permitir a construção de outras realizações a partir deste padrão.

- **Equilíbrio:** quando um padrão é utilizado em uma aplicação, o equilíbrio dá a razão, relacionada com cada uma das restrições envolvidas, para cada passo do projeto. Uma análise racional que envolva uma abstração de dados empíricos, uma observação da aplicação de padrões em artefatos tradicionais, uma série convincente de exemplos e uma análise de soluções ruins ou fracassadas, pode ser a forma de se encontrar este equilíbrio.

- **Abstração:** os padrões representam abstrações da experiência empírica ou do conhecimento cotidiano.

- **Abertura:** um padrão deve permitir a sua extensão para níveis inferiores de detalhe.

- **Combinatoriedade:** os padrões são relacionados hierarquicamente, isso é, padrões de nível superior podem ser compostos ou relacionados com padrões que tratam de problemas de nível mais baixo.

Tratando sobre o tema do desenvolvimento de *softwares*, a obra de referência sobre padrões de projeto é de Gamma [17], que define vários padrões para projetos orientados a objetos. A utilização desses padrões no desenvolvimento de projetos visa facilitar o desenvolvimento de *software* e prover um código que possa ser melhor entendido no futuro, já que os padrões de projeto tendem a simplificar o código além de, por serem amplamente conhecidos, facilitar a documentação das soluções e o entendimento de tais soluções por futuros codificadores.

Durante a reestruturação do *Freeflow3D*, os padrões de projeto auxiliaram o desenvolvimento e promoveram melhorias no código e nas soluções. Essa reengenharia de código gerou soluções mais legíveis, eficientes e de fácil manutenção. Ao longo do capítulo 5, sobre o trabalho realizado, o uso de padrões de projeto é evidenciado, junto aos benefícios trazidos pela sua utilização.

4.1.2. XML – Auxílio na reestruturação de dados

Segundo a definição da W3C[18]:

> "XML (*Extensible Markup Language*) é um formato de texto simples e flexível, derivado da SGML (ISO 8879), originalmente criada para suprir os desafios da publicação eletrônica em larga escala, mas que hoje vem tendo um importante papel na troca de uma grande variedade de dados na Web ou em qualquer lugar".

O formato XML definido acima é amplamente utilizado pela indústria de *software* na descrição e no armazenamento de informações, dadas as suas importantes características, abaixo resumidas:

- é extensível, há a possibilidade de criar etiquetas (*tags*) que permitem adaptar a estrutura de um documento XML para praticamente qualquer situação;

- os documentos elaborados neste formato XML são auto-descritivos e, portanto, relativamente fáceis de interpretar e manipular, além de permitir meios simples de busca de informações;

- é simples, porém permite criar estruturas bastante complexas (árvores, grafos, listas, etc.);

- é extremamente flexível, possibilitando a representação de dados estruturados e semi-estruturados;

- com a definição de um esquema, torna possível efetuar a validação automática de documentos, isto garante a validação dos dados para a aplicação;

- o conteúdo de um documento em formato XML pode ser facilmente manipulado pelas aplicações de *software* (recorrendo às *API*s existentes); e

- é um padrão aberto, os documentos em formato XML são independentes das aplicações, dos sistemas operacionais, etc.

A descrição acima é relevante na medida em que os arquivos e dados de comunicação e de configuração, de descrição de cena, e de descrição de interface do *Freeflow3D* utilizaram o referido formato XML. Desse modo, a aplicação objeto do presente trabalho fará uso de todas as características acima citadas, das quais vale ressaltar a facilidade de expansão, a flexibilidade, o acesso e a leitura simplificada, que são grandes vantagens do formato XML. A reestruturação de dados na forma da utilização do formato XML pode ser conferida no item 4.1.2 do capítulo 4, que descreve o trabalho de reengenharia e reestruturação dos programas do *Freeflow3D*.

4.2. Engenharia de Software baseada em componentes

Devido ao seu processo de desenvolvimento (muitas vezes inexistente – caótico – ou ineficaz), ao seu tamanho e à sua alta complexidade, grande parte dos *softwares* desenvolvidos vêem sua estrutura se tornar monolítica e de difícil entendimento. O código tende a ser tornar interdependente e cada manutenção ou evolução pode potencialmente afetar grande parte do *software*, propagando mudanças pelo sistema e tornando o desenvolvimento do *software* lento, trabalhoso e custoso. [19]

Uma alternativa apresentada pela engenharia de *software* para combater o cenário descrito acima, muito comum no desenvolvimento de *softwares*, é a engenharia de *software* baseada em componentes conhecida pela sigla ESBC (*component-based software engineering*). A ESBC prega o desenvolvimento de sistemas através do uso de componentes, ou seja, da análise e separação das funcionalidades do sistema em componentes independentes, que se comunicam através de uma interface bem definida.

Esse paradigma de desenvolvimento, baseado em componentes, promete lidar com a crescente complexidade dos sistemas e com as dificuldades de efetuar mudanças neles, já

que as mudanças tendem a ficar localizadas internamente nos componentes e não ser propagadas por todo o sistema [19]. Não obstante, existe também um grande apelo comercial envolvendo o uso da ESBC no desenvolvimento de *software*, não só pelos ganhos em manutenção ao longo prazo, mas também pelos ganhos a curto prazo no desenvolvimento inicial de sistemas que, através do reuso de componentes (tanto os desenvolvidos *in-house,* como os disponíveis no mercado), podem garantir a redução dos custos e do tempo de desenvolvimento do sistema. [14].

Como a ESBC se baseia na análise e na separação das funcionalidades do sistema em componentes independentes, que se comunicam através de uma interface bem definida, como já foi dito, é importante para o desenvolvimento do presente trabalho que se tenha definido o que é um componente e uma interface, o que se fará a seguir:

- **Componente:**

Existem diversas definições para "componentes" na literatura. A *Microsoft*, por exemplo, define componente como um "pedaço de *software* compilado, que está oferecendo um serviço"[20]. Já o *Gartner Group* define componente como um "pacote dinamicamente vinculável de um ou mais programas tratados como uma unidade e acessados através de uma interface documentada que pode ser descoberta em tempo de execução"[21]. Uma das definições mais difundidas é a *Brown*, que diz que um componente é "um conjunto de serviços reutilizáveis distribuídos independentemente"[22]. Quando *Brown* diz que um componente é um conjunto de serviços reutilizáveis, ele se refere ao fato de que o componente deve ter seus serviços bem definidos e especificados e que também deve ter um meio para que outros componentes utilizem seus serviços e, ao utilizar seus serviços, esses componentes não tenham que estar cientes de como esses serviços são executados. Por outro lado, quando *Brown* se refere à questão dos componentes serem distribuídos independentemente, ele quer dizer que os componentes não devem ser dependentes entre si e tipicamente não devem ter conhecimento do contexto nos quais estão inseridos para executar certa função. Existem outras definições, que não serão detalhadas neste trabalho, que são muito semelhantes entre si, e que, no geral, definem os

componentes como um pedaço de *software*, auto-contido, reutilizável, independente e que pode ser ligado a outros componentes para solucionar um problema maior.

- **Interface:**

A definição de "interface" complementa a definição de componentes descrita no parágrafo acima, pois ela que define quais são os meios para acessar as funcionalidades dos componentes. Segundo *Brown*, uma interface "sumariza como um cliente deve interagir com o componente, mas esconde os detalhes de implementação"[22]. *Brown* também enfatiza que uma interface deve existir separadamente dos componentes que a implementam, pois ela define, de certo modo, o comportamento e as responsabilidades do componente (que devem estar presentes em qualquer implementação da interface). Este último fato, levantado por *Brown*, garante que componentes possam ser trocados por outras implementações, desde que elas mantenham a mesma interface e garantam o comportamento e as responsabilidades esperados.

Em suma, a aplicação da ESBC no desenvolvimento de sistemas traz benefícios nas seguintes áreas [19]:

a) Reuso – componentes com funcionalidade e interface bem definidas, documentados e testados podem ser reutilizados no próprio sistema e em novos projetos. Esta reutilização traz aumento de produtividade e redução do custo de desenvolvimento, desde que o sistema absorva bem o uso dos componentes (isso é, que não sejam necessárias grandes mudanças para uso do componente no sistema receptor, do contrário as mudanças poderiam anular os ganhos com o reuso dos componentes).

b) Evolução/Substituição – os componentes podem ser facilmente substituídos por versões melhores e mais eficientes, desde que a nova versão provenha as mesmas funcionalidade e interface.

c) Manutenção – como discutido anteriormente, por trabalhar com funcionalidades encapsuladas, é possível alterar funcionalidades individualmente, sem ter que obrigatoriamente afetar todo o sistema, a manutenção do *software* pode ser feita pontualmente.

d) Escalabilidade – uma arquitetura baseada em componentes pode crescer facilmente adicionando novos componentes ou trocando os existentes por novas versões com novas funcionalidades mais eficientes.

Pressman [14] descreve um processo para o uso da ESBC em projetos, dividida em dois sub-processos: engenharia de domínio (ED) e desenvolvimento baseado em componentes (DBC). Segundo ele, a "ED identifica, constrói, cataloga e dissemina um conjunto de componentes de *software* em um domínio de aplicação particular". Já a "DBC qualifica, adapta e integra os componentes para uso no novo sistema", podendo também "desenvolver novos componentes baseados nos requisitos específicos desse novo sistema".

Os conceitos de ESBC permeiam um paradigma de programação que, por ter influenciado de certa maneira o presente trabalho, merece ser citado: o *Flow-based programming*. O termo foi cunhado por J. Paul Morrison [23] e se refere a um método de programação baseado em uma rede de processos "caixa preta", que trocam dados entre si em conexões pré-definidas para completar uma determinada tarefa. Essas "caixas pretas" são, na verdade, componentes e rede de processos interconectados e podem ser chamadas também de "fluxo de execução", já que a execução dessas "caixas pretas" interligadas computam um resultado esperado. Uma das grandes vantagens da abordagem do *Flow-based programming* é ser adequado para arquiteturas paralelizadas, pois os fluxos de dados percorrem a rede de "caixas pretas" de modo assíncrono, orientadas a eventos, e não de modo seqüencial como normalmente os problemas são resolvidos.

No presente trabalho, as características de *Flow-based programming* referentes a como os componentes se comunicam, a sua natureza orientada a eventos e altamente paralelizada, etc., não foram utilizadas e, por isso, o paradigma não será abordado em detalhes. Contudo, a idéia do uso de fluxos de execução, ou seja, do uso de uma rede de "caixas pretas" (componentes) ligadas entre si, em interfaces pré-definidas para resolver um

problema em comum, influenciou grande parte dos trabalhos aqui realizados, como poderá ser visto nos itens do próximo capítulo.

5. Trabalho Realizado - Reengenharia e Evolução do Visualizador e Modelador do *Freeflow3D*

Este capítulo tem como intuito descrever o trabalho realizado na reengenharia e evolução dos *softwares* do *Freeflow3D* e analisar como os conceitos vistos nos capítulos anteriores foram aplicados no desenvolvimento da solução.

Fazendo uma breve antecipação do que será analisado no presente capítulo, o item 5.1 descreve a análise dos *softwares* legados, ou seja, quais as deficiências das versões atuais do visualizador e do modelador do *Freeflow3D* e quais os principais requisitos que a reengenharia abordará. O próximo item 5.2, interligado com o primeiro, trata de como a nova arquitetura do *Freeflow3D* foi concebida para suprir os requisitos levantados no item 5.1. O funcionamento da nova arquitetura e os mecanismos desenvolvidos são abordados em detalhes no item 5.3. Já o item 5.4 descreve como o modelador e o visualizador foram construídos seguindo a nova arquitetura. Por fim, o item 5.5 exemplifica os ganhos trazidos pelo uso da nova arquitetura com o exemplo da criação de um novo componente do tipo *renderer* que utiliza *shaders*.

5.1. Análise do software legado: levantamento dos requisitos da reestruturação

Como insumo inicial para a reestruturação dos programas legados do *Freeflow3D*, era essencial que fosse executada uma análise do legado propriamente, a fim de elencar as principais deficiências do mesmo, os possíveis pontos de melhoria e ainda definir quais os requisitos que nortearam a definição da nova arquitetura e das novas funcionalidades.

A análise foi realizada baseando-se no código fonte e em pontos levantados junto ao desenvolvedor da versão legada, no caso o orientador do presente trabalho. As deficiências e pontos de melhoria levantados foram:

- **Código não portável:** os programas atuais rodam somente em Linux e dependem de uma biblioteca de janelas depreciada (*xview*) e que não vem instalada por padrão nas novas distribuições do Linux

- **Dificuldade de manutenção/evolução:** não existe uma arquitetura de referencia por trás da implementação dos programas legados. O programa é desenvolvido na linguagem C e por isso não contém as abstrações trazidas pela orientação a objetos. Não obstante, o código é pouco coeso e altamente acoplado, com a lógica misturada ao código de construção da interface por exemplo. As mudanças ou evoluções no código eram custosas, pois muitas vezes as alterações tinham de ser propagadas por todo o código.

- ***Renderer* via *software*:** durante o desenvolvimento dos *software*s legados, em meados de 1993, o uso de placas de aceleração gráfica 3D não era viável (tais placas não eram difundidas, eram caras e não existia um padrão de uso estabelecido no mercado). Logo, o *renderer* 3D do *software* legado é uma solução *in-house* que executa toda a computação necessária via *software*, o que significa que não se utiliza dos ganhos de performance trazidos pelas aceleradoras gráficas 3D hoje presentes em praticamente todos os computadores. Não obstante, devido às limitações de vídeo da época, as imagens geradas usavam o algoritmo de *halftone*[3] para simular 16.000.000.000 (dezesseis milhões) de cores usando somente 256 (duzentos e cinqüenta e seis), o que diminui a fidelidade das mesmas.

- **Interface com o usuário pouco intuitiva:** as limitações da biblioteca de interface gráfica utilizada na época, somado ao desenvolvimento incremental dos *software*s legados, contribuíram para que algumas deficiências e pontos de melhorias fossem identificados na interface com o usuário. São elas:

    a) Telas sem separação semântica de dados: algumas telas para inserção de dados não têm uma separação semântica, ou seja, dados de categorias diferentes são exibidos juntos, potencialmente confundindo o usuário.

Um exemplo da falta de separação semântica é a tela de criação do domínio no modelador, na qual dados relativos aos parâmetros físicos, computacionais, entre outros, são mostrados na mesma tela (vide Figura 7).

b) Mudança de parâmetros: nos programas legados, não é possível alterar os valores dos parâmetros uma vez que eles tenham sido definidos. Logo, caso o usuário errasse algum dado de entrada, seria necessário que ele começasse o processo todo desde o inicio. Desse modo, não é possível visualizar em 3D o efeito de sucessivas mudanças a fim de um ajuste fino, já que havia a dificuldade relativa à necessidade de reiniciar o processo para cada novo valor de parâmetro a ser testado. Nos programas legados, mesmo após inserir o valor em um campo, o usuário devia pressionar a tecla *Enter* para que o valor pudesse ser enviado ao programa, comportamento este que muitas vezes gera resultados incorretos quando o usuário deixa de apertar *Enter* no final da edição do parâmetro e, conseqüentemente, o valor não é atualizado.

c) Interação com a cena: dentre outros pontos que poderiam ser citados, identificou-se que não era possível selecionar objetos, alterar sua posição ou a posição da câmera, de modo ágil usando o mouse diretamente na cena 3D. Toda mudança era feita através dos menus, em telas secundárias e após vários cliques de mouse, deixando o processo de mudança mais lento.

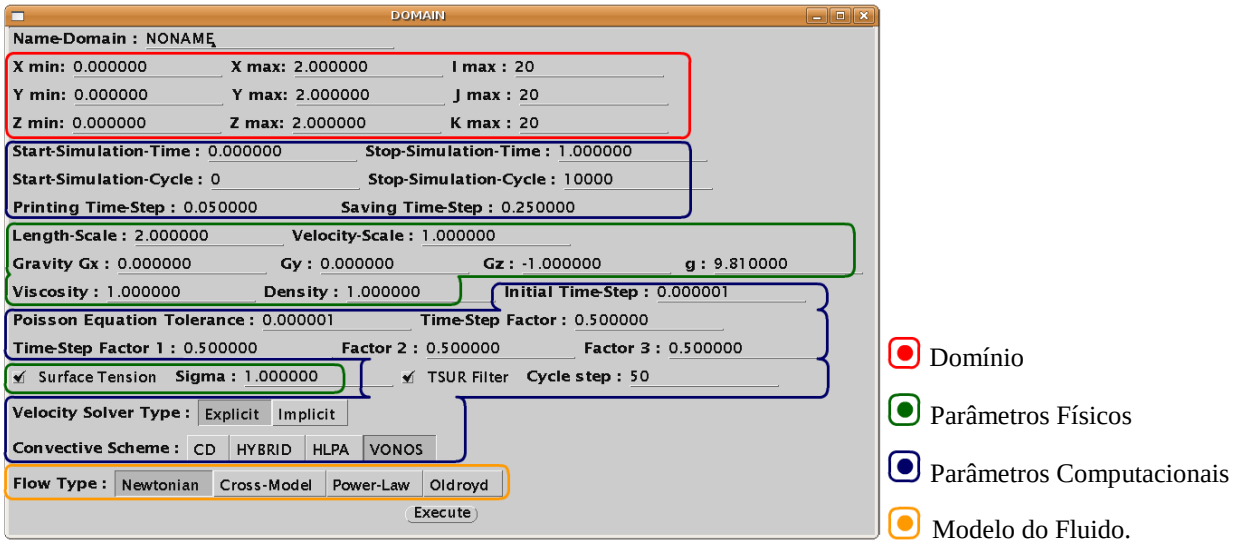


Figura 7: Tela sem separação semântica de dados – *Screenshot* da tela de definição de parâmetros para simulação do modelador legado. Dados sobre o domínio, parâmetros físicos, parâmetros computacionais e modelos do fluido se encontram espalhados pela tela, intitulada "*DOMAIN*".

Os pontos de melhorias transcritos acima se tornaram nos requisitos a serem considerados na reestruturação do programa legado, sendo esses priorizados junto ao pesquisador responsável pelo *Freeflow3D* formando a seguinte lista (vide Tabela 1 abaixo):

Prioridade Alta	Evoluções/Manutenções
	Portabilidade
	Performance e *Rendering* acelerada em *hardware*
Prioridade Baixa	Melhorias na interface gráfica e interface com o usuário

Tabela 1: Ordem de prioridades dos requisitos para reestruturação do *Freeflow3D.*

A lista de prioridade dos requisitos para reestruturação do *Freeflow3D* norteou a criação da nova arquitetura, e se tornou a base para a reestruturação dos programas legados, como pode ser visto no próximo item deste capítulo 5.

5.2. Definição de uma nova arquitetura - baseada em componentes

Segundo a IEEE, a arquitetura de *software* pode ser definida da seguinte forma: *"Arquitetura é a organização fundamental de um sistema incorporada em seus componentes, seus relacionamentos com o ambiente, e os princípios que conduzem seu design e evolução."*[24].

Como visto acima, a definição de uma arquitetura de *software* deve servir como guia do design e da evolução do sistema. Para a reestruturação dos programas legados do *Freeflow3D*, objetivo deste trabalho, viu-se que era necessário definir uma nova arquitetura guia. Essa nova arquitetura guia tem como objetivo suprir as deficiências da arquitetura legada e, conseqüentemente, os requisitos previamente levantados no item 5.1 a partir da análise dos programas, priorizadas na seguinte ordem: portabilidade; evoluções e manutenção; performance e *rendering* em *hardware*; e melhorias na interface com o usuário.

Dos requisitos e dos estudos realizados sobre a Engenharia de *Software* Baseada em Componentes (item 4.2), segue a definição básica elaborada no presente trabalho para a nova arquitetura do *Freeflow3D*:

1. *Os novos visualizador e modelador do Freeflow3D devem ter suas funcionalidades representadas por componentes.*

2. *A execução desses programas deve ser baseada em um fluxo de execução, que é definido pela ligação dos componentes.*

3. *Essas ligações entre componentes, ou seja, o fluxo de execução, representam o funcionamento do programa.*

A definição básica da nova arquitetura provê uma idéia geral de como a reestruturação do *Freeflow3D* deve ser executada, e a partir dela é possível detalhar como cada item levantado funcionará e como se pretende atender aos requisitos da

reestruturação através deles. Segue análise item a item da definição da nova arquitetura do *Freeflow3D*:

1. *Os novos visualizador e modelador do Freeflow3D devem ter suas funcionalidades representadas por componentes.*

Uma vez que os *softwares* legados forem analisados, suas funcionalidades serão separadas e darão origem a componentes semanticamente coesos. Esses componentes têm como objetivo isolar as funcionalidades do legado, facilitando a manutenção dos mesmos. Além disso, eles devem ter uma interface (funcional) bem definida, de modo que possam ser substituídos por versões melhores e assim contribuam para simplificar as evoluções dos programas. Não é objetivo da presente arquitetura reutilizar os componentes criados em outras aplicações, mas sim reutilizar os componentes em fluxos de execuções diferentes, conforme será explicado a seguir.

2. *A execução desses programas deve ser baseada em um fluxo de execução, que é definido pela ligação dos componentes.*

Para a arquitetura de *software*, um fluxo de execução é uma organização de componentes com a finalidade de resolver um problema proposto às aplicações legadas. Logo, fluxos de execuções guias serão extraídos da análise de como o *software* legado resolve os problemas, tanto de visualização como de modelagem. Esses fluxos de execução guia extraídos do *software* legado definirão como os componentes podem ser ligados entre si, ou seja, a sua interface de execução. A vantagem principal dessa abordagem é o reuso dos componentes em fluxos de execuções diferentes (como será visto no item 5.3, sobre a aplicação da nova arquitetura). Todos os benefícios e características do fluxo de execução podem ser conferidos no item 5.2.2 do presente trabalho.

3. *Essas ligações entre componentes, ou seja, o fluxo de execução, representam o funcionamento do programa.*

Este item descreve que os componentes, ligados entre si e formando um fluxo de execução, representam a execução do programa. Isso quer dizer que toda a execução dos programas desenvolvidos sob a nova arquitetura será definida por um fluxo de execução único, projetado e descrito pelo usuário para resolver certo problema. Todo esse aparato, ou seja, os componentes e os fluxos de execução, deve ser uma interface ao núcleo do *Freeflow3D*, que contém as funcionalidades mais básicas da solução de mecânica de fluidos (por exemplo: algoritmos de geométrica computacional e geração de malha, algoritmos de leitura e escrita de dados, etc. veja detalhes no item 5.2.4). Uma das vantagens da abordagem trazida pela nova arquitetura é o reuso de fluxos de execução já desenvolvidos como ponto de partida para a resolução de problemas semelhantes. O item 5.3.1 descreve como uso e reuso de fluxos de execução foi desenvolvido e o item 5.3.4 apresenta uma ferramenta para auxiliar o usuário na definição e no desses fluxos de execução.

Em suma, a nova arquitetura desenvolvida no presente trabalho é uma adaptação do uso de componentes e de seu processo de desenvolvimento (para maiores detalhes vide item 5.2.1) com alguns aspectos do paradigma de programação orientada a fluxos e fluxos de dados, que influenciaram a forma como os componentes são ligados formando os fluxos de execução (mais detalhes no item 5.2.2).

O objetivo principal da arquitetura de *software* é facilitar o uso dos programas pelo usuário final e facilitar o desenvolvimento e evolução do mesmo. A nova arquitetura, através do uso de componentes e fluxos de execução, deve facilitar a interface com o núcleo do *Freeflow3D* gerando os arquivos de entrada para a simulação, no caso do modelador, e realizando a leitura dos arquivos gerados pelo simulador, no caso do visualizador (vide Figuras 8 e 9 abaixo).

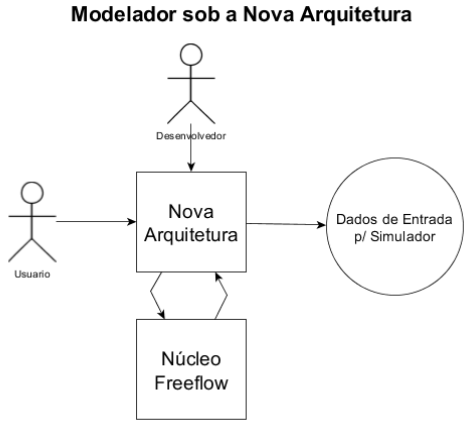


Figura 8: Nova arquitetura no modelador – A nova arquitetura se propõe a facilitar o uso do programa tanto pelo usuário e o desenvolvedor, fazendo interface com o núcleo do *Freeflow3D* para, como resultado final, gerar uma cena de entrada para o simulador.

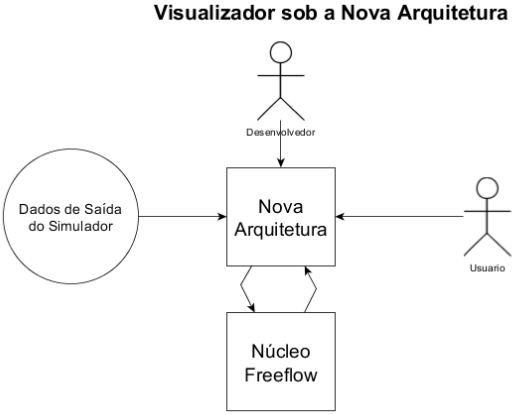


Figura 9: Nova arquitetura no visualizador – A nova arquitetura se propõe a facilitar o uso do programa tanto pelo usuário e o desenvolvedor, fazendo interface com o núcleo do *Freeflow3D* para, como resultado final, possam ser visualizadas as cenas criadas a partir da simulação numérica.

Com o propósito de aperfeiçoar o entendimento do que foi explanado até aqui, os itens abaixo (5.2.1. e 5.2.2.) definem os conceitos de componentes e fluxos de dados para o presente trabalho, quais são os seus papéis na nova arquitetura e como eles garantem o seu funcionamento. Em seguida, o item 5.2.3 descreve a escolha das ferramentas para o desenvolvimento dessa arquitetura e a estratégia escolhida para a reestruturação do código. Ainda com o mesmo intuito didático, o item 5.2.4 prossegue com os conceitos e define o que significa o "Núcleo do *Freelow*" para os fins do presente trabalho e como a nova arquitetura

faz interface com ele. Ao final, o item 5.2.5 conclui este tema descrevendo as novas funcionalidades previstas no desenvolvimento sob a nova arquitetura.

5.2.1. Componentes - simplificação, flexibilidade e manutenabilidade

Uma das principais deficiências percebidas no *software* legado é a sua característica monolítica, com baixa coesão do código e alto acoplamento entre os códigos fontes. Essas características dificultam o entendimento do código, sua manutenção e conseqüente evolução.

Como visto no item 4.2, o uso de componentes favorecem o reuso, a substituição e a evolução dos módulos, além de favorecer a escalabilidade da solução, isto é, é possível adicionar, com certa facilidade, mais componentes e funcionalidades à solução final. Para atingir esses objetivos, definiu-se que os programas legados seriam analisados e quebrados em componentes semanticamente coesos, como já foi dito anteriormente. Esses componentes resultantes da "quebra" deveriam seguir algumas características que garantissem à nova arquitetura o nível de modularidade e manutenabilidade requerida, sendo elas:

- Os componentes devem ser altamente coesos, ou melhor, devem representar o funcionamento de uma parte bem específica da aplicação e conter interface, dados e comportamento compatíveis com essa funcionalidade. Esta característica tem como intuito favorecer a organização e manutenção do código.

- Os componentes devem ter uma interface bem definida que possa favorecer a troca e expansão dos mesmos.

- Os componentes devem ser definidos de uma forma que favoreça sua criação e utilização, isso é, devem ser construídos a partir de uma base comum.

- Os componentes devem ter uma estrutura de dados interna flexível, que possa crescer, ser acessada e modificada facilmente.

A partir das características contidas nos parágrafos acima, tem-se a seguinte representação gráfica do componente para a nova arquitetura:

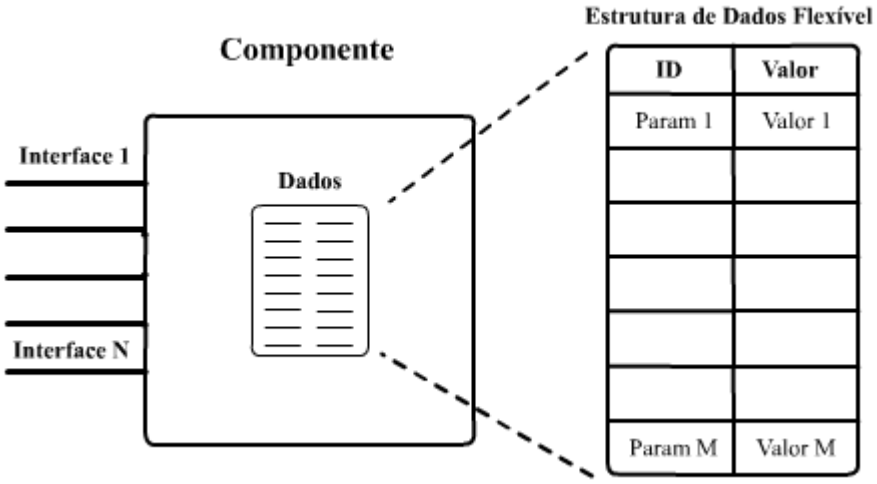


Figura 10: Representação gráfica inicial do componente para a nova arquitetura – Deve conter uma interface bem definida e uma estrutura de dados flexível para facilitar o armazenamento de informações.

É importante salientar que o conceito de componentes seguido pela nova arquitetura difere em alguns aspectos do que é sugerido pela ESBC. A ESBC recomenda que o componente não mantenha informações de estado e que o mesmo possa ser reutilizado em qualquer aplicação. Diferente do que sugere a ESBC, na nova arquitetura objeto do presente trabalho o componente não tem como objetivo ser reutilizado em outras aplicações, mas sim em diversos fluxos de execuções na mesma aplicação.

Logo, um componente, para a nova arquitetura, mantém certas características pregadas pela ESBC, entre elas ter uma interface bem definida, ser passível de substituição e reuso, ter atribuições funcionais coesas com seu propósito. Contudo, por ter sua área de

atuação restrita às aplicações do *Freeflow3D*, seu design pode ser flexibilizado para facilitar o uso da arquitetura. O fato dos componentes manterem o estado da execução é o principal exemplo dessa flexibilização de design, e é o que garante o funcionamento do componente durante todo o fluxo de execução. A estrutura de dados interna dos componentes também pode ser considerada um fruto dessa flexibilidade de design e tem como intuito facilitar o desenvolvimento de componentes, além de proporcionar ao usuário uma interface mais direta e a possibilidade de configuração dos componentes.

Essa característica dos componentes terem seu campo de atuação restrito ao ambiente do *Freeflow3D* também teve influência no processo de uso dos componentes na arquitetura. Como visto no item 4.2, existem dois subprocessos relacionados ao uso de componentes em um projeto, o de engenharia de domínio (ED) e o de desenvolvimento baseado em componentes (DBC). De acordo com o processo de ED, a nova arquitetura deve identificar as necessidades do domínio de aplicação dos *softwares* do *Freeflow3D* e definir os requisitos para componentes necessários, porém não deve se preocupar em buscar e catalogar no mercado componentes já existentes. Já segundo o processo do DBC, a nova arquitetura deve se ater somente a desenvolver os componentes identificados pelo processo de ED e integrá-los no novo sistema, sem se preocupar em qualificar e adaptar componentes que porventura sejam adquiridos no mercado.

Em resumo, o componente definido para a nova arquitetura tem características e comportamento peculiares. Tal componente prima por favorecer o reuso, a manutenção e a expansão do sistema, porém algumas das características dos componentes vistos na ESBC foram adaptadas pela nova arquitetura. Essas adaptações visam viabilizar o uso dos componentes tanto pelos futuros desenvolvedores dos novos programas como também facilitar o uso dos novos programas pelos usuários do *Freeflow3D* (mais detalhes no item 5.3., sobre a aplicação da nova arquitetura).

Portanto, os componentes da nova arquitetura são únicos ao domínio de aplicação do *Freeflow3D*, o que fez com que o processo de uso de componentes também fosse adaptado para as necessidades do presente trabalho.

Como visto na definição da arquitetura (item 5.2), os componentes devem ser ligados entre si através dos fluxos de execuções. O item 5.2.2 a seguir trata de como o conceito de fluxo de execuções foi adaptado para uso na nova arquitetura e define como os

componentes devem ser ligados entre si, e pode ser considerado como uma continuação do presente item, já que o uso dos componentes nos fluxos de execuções complementam a definição do componente na nova arquitetura.

5.2.2. Fluxo de execução - ligando componentes

Como visto no item 4.2, o fluxo de execução é uma rede de "caixas pretas" interligadas entre si através de uma interface pré-definida, e que tais "caixas pretas" se comunicam para resolver certo problema. Para a nova arquitetura, esse conceito foi adaptado e, em uma definição simples, representa o funcionamento do programa através da ligação de componentes visando à resolução de um determinado problema.

As ligações que formam um fluxo de execução devem representar as etapas de funcionamento do programa, ligando os componentes em uma espécie de linha de produção. Em outras palavras, os componentes devem ser ligados representando sua interdependência na semântica do problema.

Essas ligações devem ser hierárquicas, obedecendo à semântica do problema. Ou seja, caso o componente A seja dependente semanticamente do componente B, o componente B deve ser pai do componente A (vide Figura 11 abaixo). Isso adiciona aos componentes relações de interdependência do tipo "pai" e "filho" e define um fluxo de componentes.

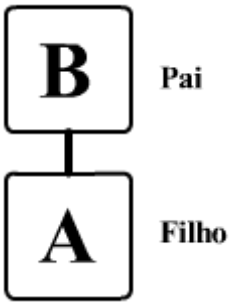


Figura 11: Exemplo da ligação de dependência hierárquica – Se A é semanticamente dependente de B no âmbito do problema, então existe uma interdependência entre os componentes e A é "filho" de B.

Com as ligações hierárquicas entre componentes, é possível manter em cada componente uma espécie de rotina interna, que é responsável, junto à interface do componente, pela execução de sua funcionalidade.

As rotinas internas são chamadas seguindo a ordem das ligações entre os componentes e, diferentemente das chamadas diretas pela interface do componente, elas podem contar com o resultado da execução do componente anterior para auxiliar em sua execução. Logo, a execução do fluxo é a execução cíclica da rotina interna dos componentes, seguindo a ordem de ligações hierárquicas entre eles (vide Figura 12).

É através das referidas rotinas internas que os componentes devem se comunicar com o Núcleo do *Freeflow3D*, ou seja, com o nível mais baixo de operações do programa, para então gerar o resultado esperado.

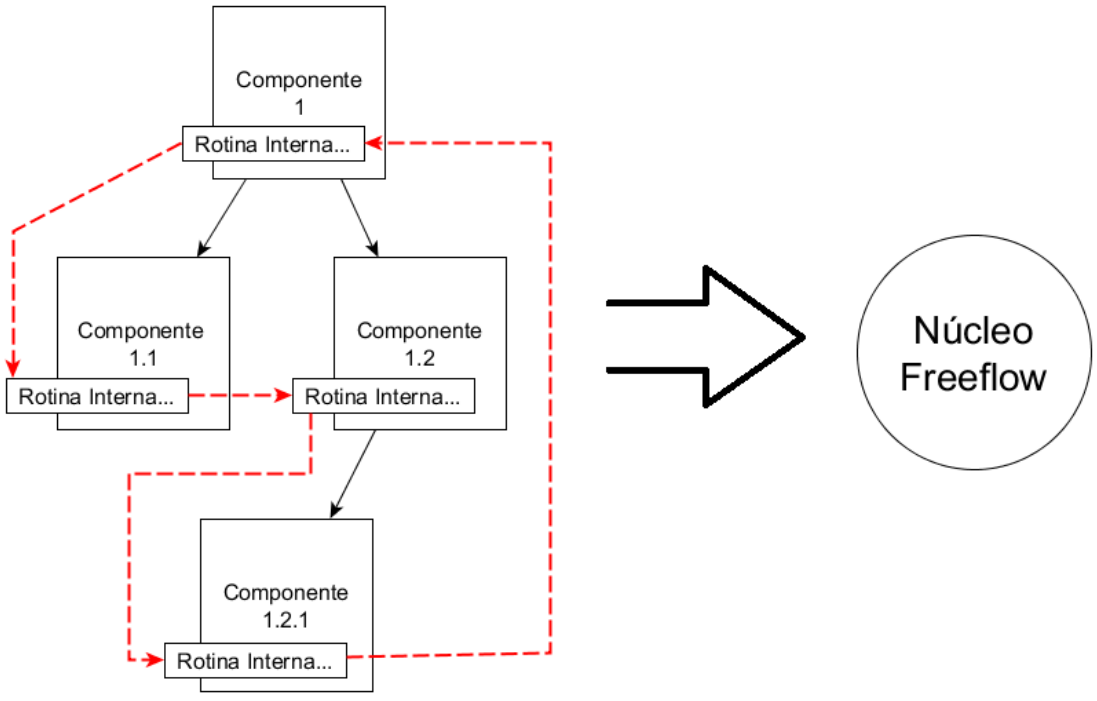


Figura 12: Rotina de execução do fluxo – Os componentes são percorridos em um clico de execução, seguindo suas ligações hierárquicas (linha tracejada indica a ordem) executando suas rotinas internas. Através dessa rotina interna, os componentes podem se comunicar com o núcleo do *Freeflow3D*.

O componente base, inicialmente definido no item 5.2.1, teve sua definição complementada pelas características necessárias para seu funcionamento junto ao fluxo de execução. A representação gráfica do componente base, com as adições de funcionalidades devido ao fluxo de execução, pode ser conferida na Figura 13 a seguir:

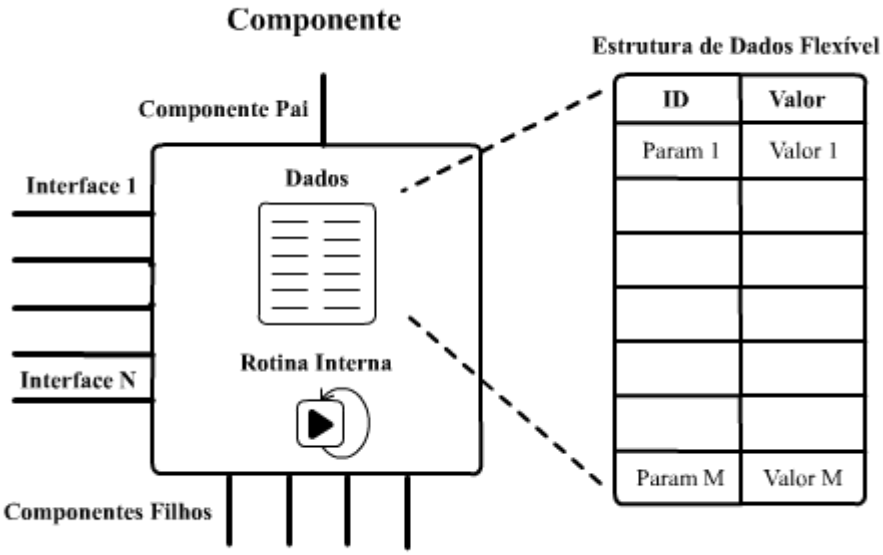


Figura 13: Representação gráfica final componente – Características adquiridas para funcionamento junto aos fluxos de execução.

Existem, na arquitetura criada neste trabalho, dois tipos de fluxos de execução. Primeiramente, temos os fluxos de execução guia, que são criados a partir da análise do problema e tem como objetivo definir quais componentes serão necessários para resolver esse problema e quais serão suas interfaces (mais detalhes sobre o processo de criação de fluxos guias pode ser conferido no item X). É a partir do fluxo de execução guia que o usuário final pode criar seu próprio fluxo de execução, escolhendo as implementações dos componentes que irão ser interligadas seguindo as interfaces definidas pelo fluxo de execução guia, e que resultarão na resolução de certo problema (no caso do presente trabalho, os fluxos de execução resolveram problemas de modelagem ou de visualização de cena, conforme o caso). Vale ressaltar que nem todos os componentes definidos no fluxo de execução guia são necessários na criação de um fluxo de execução customizados, visto que

algumas ligações entre componentes não são obrigatórias, ou recebem 0 a N ou 1 a N ligações.

Uma vez definido o fluxo de execução para a nova arquitetura e como eles são estabelecidos, é possível compreender quais benefícios são trazidos por essa abordagem para a reestruturação dos programas do *Freeflow3D*. Esses benefícios serão abordados em mais detalhes e através de exemplos nos próximos itens do trabalho, mas cabe uma rápida descrição:

- Reuso de componentes: os componentes podem ser reutilizados em diferentes organizações de fluxos de execução definidos pelo usuário final.

- Troca de componentes: no caso de diversas implementações de componentes garantirem as funcionalidades e responsabilidades de certa interface, esses componentes podem ser trocados no fluxo de execução. Este fato favorece a evolução da solução, já que novas versões dos componentes podem ser criadas, com novas funcionalidades ou trazendo mais eficiência, e serem trocadas sem prejuízo ao usuário final ou ao resultado final.

- Reuso dos fluxos de execução para resolução de problemas semelhantes: é possível reutilizar um fluxo de execução criado para resolver um problema parecido, servindo o fluxo de execução original como versão inicial para o novo fluxo. O fluxo de execução original já contém todas as características necessárias para resolver o problema inicial, cabendo ao usuário somente alterar ou adicionar as informações necessárias para resolver seu problema, não tendo que recriar todo o fluxo novamente.

- Nova maneira de interface com o usuário: para que o usuário possa usufruir de todas essas características do fluxo de execução, a interface entre ele e o programa deve mudar. O que era outrora definido por meio de menus, sub-menus e telas pouco coesas, agora é definido diretamente no fluxo de

execução. O fluxo de execução provê uma melhor imagem de todas as peças que compõem a solução e facilita o entendimento da mesma.

5.2.3. *Java* e Padrões de Projeto - reestruturação do código legado

O programa legado (entende-se por programa legado o modelador e visualizador do *Freeflow3D*) foi desenvolvido com a utilização da linguagem C. Várias das funcionalidades do programa legado estão codificadas juntamente com o código da interface, o que resulta em uma baixa coesão e em um alto acoplamento e, conseqüentemente, na dificuldade do seu entendimento e de sua manutenção. Ademais, o programa legado não é portável e roda originalmente apenas no *Linux*, e depende da biblioteca de interface gráfica *xview*, que é limitada e foi depreciada.

Diante do cenário descrito no parágrafo acima acerca do programa legado e levando em conta os requisitos da reestruturação dos programas e a arquitetura concebida para tanto, foi preciso decidir em qual linguagem a reestruturação do código seria desenvolvida e quais estratégias seriam utilizadas para produzir um aperfeiçoamento de todo o código.

Como visto nos itens anteriores deste capítulo, a nova arquitetura objeto do presente trabalho propõe o uso de fluxos de execuções e componentes. Os componentes devem ser objetos “auto-contidos”, ou seja, encapsulados com interface bem definida. O conceito de “auto-contido” está muito próximo ao de classes de objetos do paradigma orientado a objetos. Já o funcionamento dos fluxos de execução implica em resolver qual implementação do componente usar (instanciar) em tempo de execução.

Diante destas necessidades impostas, escolheu-se neste trabalho a linguagem *Java* para toda a reestruturação dos programas do *Freeflow3D*. O motivo da escolha está no fato de que a referida linguagem é portável e garante que os novos programas possam funcionar nas três principais plataformas de computadores pessoais, quais sejam: *Windows*, *Linux* e *Mac*. A linguagem *Java* é orientada a objetos e tem a capacidade de instanciar classes em tempo de execução, processo esse chamado de reflexão (“*Reflection*”), que é essencial para o funcionamento dos fluxos de execução.

É possível dividir o código legado do *Freeflow3D* em duas categorias distintas: a) a categoria dos códigos que representam funcionalidades básicas do *Freeflow3D*; e b) a dos códigos que representam as funcionalidades de alto nível, ou seja, que utilizam o conjunto de funcionalidades mais básicas para atingir um objetivo maior. Para facilitar sua identificação, neste trabalho as funcionalidades básicas (item "a") serão chamadas de "Núcleo do *Freeflow3D*" (ex.: rotinas de geometria computacionais, entre outras) e as funcionalidades de alto nível (item "b") serão chamadas de "macro-funcionalidades" (ex.: inserir objetos na cena; aplicar uma propriedade como textura do fluido; etc.).

Como será visto ao longo do presente capítulo, e principalmente nos próximos itens, que abordam a aplicação da nova arquitetura, as macro-funcionalidades dos programas do *Freeflow3D* foram transformadas em componentes. Logo, o código dessas macro-funcionalidades será refeito a partir da análise do legado em *Java*. Em toda a nova arquitetura, desde o componente base que auxilia o desenvolvimento de todos os componentes - como os programas que auxiliaram no funcionamento da nova arquitetura- até as telas e interfaces gráficas, foram empregadas as melhores técnicas de desenvolvimentos, utilizando padrões de projeto que garantem a funcionalidade e a legibilidade de código (os padrões de projeto serão citados ao longo deste capítulo, quando relevantes para o funcionamento de algum mecanismo).

Outra grande vantagem do uso do *Java* é a existência das bibliotecas e dos *frameworks* de mercado disponíveis, que agilizam o desenvolvimento do *software* e disponibilizam funcionalidades úteis, documentadas e largamente testadas. Vale citar os principais *frameworks* utilizados no presente trabalho:

- *Swing*[25]: responsável pela interface gráfica e por todos os seus componentes.

- *SwingX*[26] e *L2FProd*[27]: extensões do *Swing* que provêm novos elementos gráficos.

- *Log4J*[28]: auxilia na geração de *logs* do sistema

- *Spring*[29]: tem diversas funcionalidades que auxiliam na construção de *software*s. No presente trabalho, foi utilizado somente o módulo de inversão de controle (instanciação e configuração de objetos a partir de arquivos de configuração em tempo de execução).

- *Castor*[30]: gera classes representando os elementos de arquivos XML automaticamente, com base na descrição dos arquivos XSD[x] (linguagem de esquematização e descrição de arquivos XML).

- *Jogl* [31]: abstração das chamadas *Opengl* para *Java*.

- *Maven*[32]: resolve as dependências do projeto automaticamente, buscando-as em repositórios *online* e facilitando assim a compilação do projeto.

5.2.4. Núcleo do *Freeflow3D*

Como definido no item anterior, o código do *software* legado pode ser dividido em duas categorias: o "Núcleo do *Freeflow3D*" e suas "macro-funcionalidades". Não é objetivo do presente trabalho alterar o "Núcleo do *Freeflow3D*", isso é, modificar a parte que trata das suas estruturas internas, da geração de modelos geométricos, de suas operações, da manipulação das árvores necessárias para simulação e da gravação dos modelos de dados em disco para servir como entrada para o simulador.

A nova arquitetura desenvolvida para a reestruturação do *Freeflow3D* foi criada com o intuito de facilitar o uso, a manutenção e a evolução dos programas do *Freeflow3D* através da manipulação de operações básicas no Núcleo do *Freeflow3D*. Isso significa que os componentes e fluxos de execução, que são elementos da nova arquitetura, implementarão as macros-funcionalidades e farão interface com o "Núcleo do *Freeflow3D*" para operações

mais básicas, como, por exemplo, gerar a malha 3D dos objetos da cena, gravar a cena no formato base do *Freeflow3D*, etc.

Para acesso às funcionalidades do Núcleo do *Freeflow3D* foram consideradas duas opções: criar uma interface *Java* -> C, usando *JNI (Java Native Interface)*[33], o que implicaria em criar uma interface bem definida de comunicação, e isolar o código C em bibliotecas, etc.; ou, como segunda opção, traduzir o código C para a linguagem *Java*.

Com o intuito de obter um maior controle sobre o que estava acontecendo no Núcleo do *Freeflow3D*, foi decidido que o caminho a ser seguido no presente trabalho seria o de tradução de todo o Núcleo do *Freeflow3D* para a linguagem *Java*.

Não é objetivo deste trabalho analisar a tradução da linguagem C para a linguagem *Java*, mas vale ressaltar algumas características deste processo:

- Processo manual: a tradução de linguagem teve que ser realizada manualmente, o que implicou em grande dispêndio de tempo.

- Manutenção do código original: durante a tradução houve uma grande preocupação em tentar manter, na medida do possível, o código mais parecido com o original para que não houvesse grandes alterações estruturais.

- Volume: para exemplificar melhor o volume de trabalho envolvido na execução da tradução pode-se apresentar os seguintes números: 55 (cinqüenta e cinco) classes criadas e cerca de 10.000 (dez mil) linhas de código.

Para validar a tradução, foram modeladas duas cenas idênticas para entrada no simulador, tanto no programa legado (escrito em C e executado no *Linux*) quanto no *software* reestruturado (escrito em *Java* e executado no *Windows*). Essas cenas utilizaram extensivamente o "Núcleo do *Freeflow3D*", mais particularmente as suas funções para gerar a malha dos objetos, para configurar as árvores da simulação, para gravar o arquivo final, etc. No fim do processo, os binários resultantes das cenas montadas foram comparados e estavam idênticos, validando o funcionamento das funções básicas do *Freeflow3D* traduzidas para *Java*.

5.2.5. Novas funcionalidades - engenharia progressiva na reestruturação

Como parte do processo de engenharia progressiva, previsto na reengenharia de *software*, foram estabelecidas novas funcionalidades para serem adicionadas na reestruturação dos programas do *Freeflow3D*. Segue abaixo a descrição das novas funcionalidades (suas implementações podem ser conferidas ao longo dos próximos itens, em especial o item 5.3.6, que dispõe sobre os mecanismos desenvolvidos).

- Interface com o mouse: no item 5.1 do presente trabalho foi dito que a interação com a cena feita pelo usuário no programa legado era realizada apenas com a utilização de menus. Na reestruturação do programa, foi estabelecido que o usuário pudesse utilizar o mouse para navegar e alterar a cena, facilitando e agilizando o uso do programa.

- Edição da cena em tempo de execução: como descrito no item 5.1, no programa legado o usuário não tem a opção de editar a cena após alguma definição, isso é, caso o usuário entre com algum dado errado, é necessário que ele reinicialize o programa e comece todo o processo do inicio. Nos programas reestruturados, o usuário deve possuir a habilidade de alterar os parâmetros da cena e dos objetos em tempo de execução, sem a necessidade de reiniciar o programa e perder todo o trabalho feito até então.

- Facilidade de criação de interface gráfica: antes da reestruturação, não era simples para o desenvolvedor adicionar um novo campo na interface gráfica. O programa reestruturado apresenta como nova funcionalidade um método mais fácil de criação e alteração de interface gráfica. Essa nova funcionalidade, que gera interface gráfica de modo automático, pode ser conferida em mais detalhes nos itens 5.3.5 e 5.3.6.

- Reuso e alteração de cenas do modelador: no *software* legado, não era possível reutilizar cenas já criadas no modelador, sendo necessário recriá-las, mesmo que a

cena alvo fosse bem semelhante a uma cena já criada anteriormente. Durante a reestruturação, o uso de fluxos de execuções deve alterar esse comportamento, possibilitando, como nova funcionalidade, o reuso das cenas já criadas.

- Múltiplas telas do visualizador: o visualizador legado só possuía uma área de *rendering* 3D. Como nova funcionalidade, o visualizador reestruturado fornece um número variado de áreas de *rendering*, cada uma com sua própria câmera e suas propriedades individuais. Pode-se dizer que com tal funcionalidade é possível visualizar a mesma cena de ângulos diferentes ou, ainda, visualizar diferentes características dos fluidos em áreas de *rendering* diferentes (ex.: na área de *rendering* 1 visualiza-se dados da velocidade na eixo X; na área de *rendering* 2 visualiza-se dados de pressão; e assim por diante, veja Figura 38 do item 5.4.2 - sobre o visualizador - que ilustra essa funcionalidade).

5.3. Aplicação da nova arquitetura

Até esse ponto do capítulo 5, foram abordados os pontos que definiram a arquitetura que guiará o processo de reestruturação do *Freeflow3D*. A partir de agora, será descrito como a arquitetura concebida foi efetivamente aplicada para gerar os resultados e requisitos esperados desse processo.

O item 5.3.1 mostra como foram definidos os componentes e fluxos de execuções a partir da análise do programa legado. O item 5.3.2 trata de como os componentes foram desenvolvidos enquanto o item 5.3.3 mostra como os elementos criados, tanto componentes quanto fluxos de execuções, foram descritos em forma de arquivo.

Já os itens 5.3.4 e 5.3.5 descreve a criação dos dois programas responsáveis pelo funcionamento da nova arquitetura, o *WorkflowDrawer* (auxilia o desenho de fluxos de execução) e o *WorkflowRunner* (executo os fluxos de execução) respectivamente. Os

mecanismos relevantes, que foram implementados durante todo o processo de aplicação da nova arquitetura, são explicados no item 5.3.6.

Todos esses itens explicam como a nova arquitetura foi aplicada sobre o *software* legado para gerar a reestruturação do visualizador e do modelador e servem de base para que se entenda o resultado do trabalho apresentado no próximo item (vide item 5.4 acerca da execução dos programas reestruturados do *Freeflow3D*).

5.3.1. Definindo os componentes básicos e os fluxos de execução guia

Como visto na definição da arquitetura (item 5.2), a reestruturação dos programas *Freeflow3D* se baseiam nos componentes e em como eles são ligados formando os denominados “fluxos de execução”. Como visto nos itens 5.2.1 e 5.2.2 deste trabalho, que tratam dos componentes e fluxos de execução, esses componentes são definidos a partir de uma análise do programa legado.

Retomando os pontos vistos anteriormente, para execução desta análise do programa legado os componentes supracitados devem representar uma parte funcional do espaço do problema, ou seja, devem ser semanticamente coesos. Ser “semanticamente coeso” significa dizer que os componentes devem ser pensados de forma que possam ser trocados no futuro por versões melhores ou que contenham mais novas funcionalidades, ou melhor, que devem possuir uma interface bem definida e representar funcionalidades nas quais a possibilidade de troca e evolução possa ser uma vantagem.

Em paralelo a essa análise para componentização, tem-se também a análise para a criação de fluxos de execução guia, que indica como os componentes se ligarão para representar o funcionamento do programa e deve considerar que componentes podem ser reutilizados entre os fluxos guias. Logo, as análises de componentização das funcionalidades e de definição dos fluxos de execução se complementarão na definição dos componentes a serem criados na reestruturação do *Freeflow3D*.

Considerando que o presente trabalho tem como escopo reestruturar os programas legados do *Freeflow3D*, sendo eles o visualizador e o modelador, logo, como resultado final

da análise do funcionamento desses programas espera-se a definição de dois fluxos de execução guia, um para cada aplicação, bem como a definição dos componentes básicos que fazem parte desses fluxos e que representam funcionalidades bem definidas.

Antes de prosseguir, cabe definir melhor o que é um “componente básico”. Os componentes básicos são componentes abstratos, ou seja, eles não possuem implementação e simplesmente definem o comportamento e as responsabilidades que as implementações devem possuir. De fato, os componentes básicos são, seguindo a definição de componentes do item 5.2.1, as interfaces dos componentes que serão posteriormente implementados.

Para obter os fluxos guias para cada programa (o visualizador e o modelador do *Freeflow3D*) e os componentes básicos que farão parte de desses fluxos, foi elaborada como primeira etapa do processo de reestruturação em questão uma macro-análise dos programas, realizada junto ao orientador do presente trabalho e idealizador do *Freeflow3D*, dividindo-os em módulos semanticamente coesos (ou seja, com sua funcionalidade bem definida) e interdependentes.

Nessa macro-análise descrita no parágrafo anterior, considerou-se que os módulos, além de coesos, pudessem ser áreas do programa que potencialmente seriam alteradas em uma possível evolução, ou trocadas por uma diferente implementação da mesma funcionalidade. Por exemplo, um módulo de “Parâmetros Computacionais” pode receber, em uma versão futura, mais alguns parâmetros e comportamentos, portanto, se o módulo de “Parâmetros Computacionais” for elaborado na forma de um componente independente, pode ser criada uma nova versão do componente adicionando essas novas funcionalidades sem maiores dificuldades.

Outro exemplo relevante é o caso dos “modelos de fluido”, que mudam dependendo da estratégia utilizada para simulação e podem ser representados por N componentes intercambiáveis entre si. Essa macro-análise dos programas estabeleceu módulos interdependentes que foram o ponto de partida para a criação dos componentes básicos e dos fluxos de execução guias para a aplicação da nova arquitetura e para a conseqüente reestruturação do *Freeflow3D*.

O resultado da macro-análise do modelador segue representado na Figura 14 abaixo:

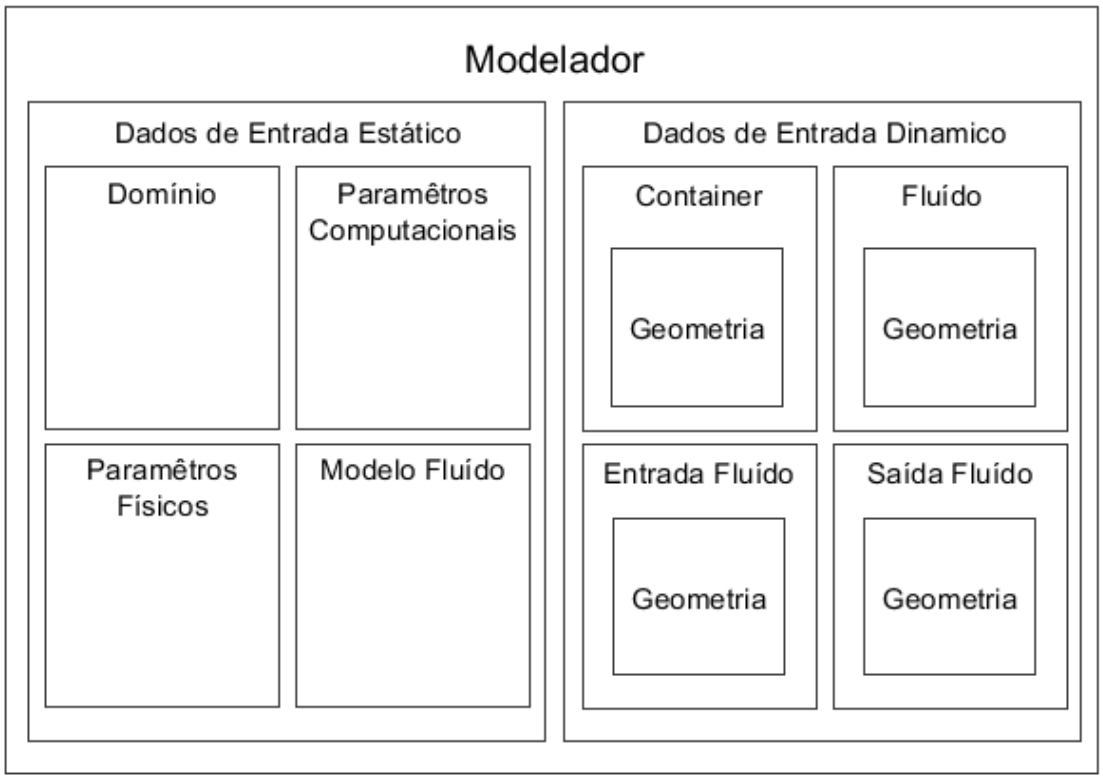


Figura 14: Macro-análise do *Freeflow3D* – Módulos propostos após análise do funcionamento do modelador.

Esta macro-análise mostra os potenciais componentes básicos da solução e suas prováveis ligações para formarem os fluxos de execução. Porém, ela levou em conta somente aspectos relacionados ao *Freeflow3D*, tais como, mas sem limitar-se a: parâmetros para computação, modelos de fluidos, e contêineres. A partir desse panorama, com os módulos referentes ao *Freeflow3D* definidos, foram adicionados módulos necessários para o funcionamento do programa e passíveis de componentização (como por exemplo, o *renderer*), sendo então transformados em componentes, com suas ligações (tanto obrigatórias como opcionais), dando origem aos dois fluxos de execução guia: o fluxo de execução para a modelagem e o fluxo de execução para o visualizador. Esses fluxos guias podem ser conferidos nas Figuras 15 e 16 abaixo:

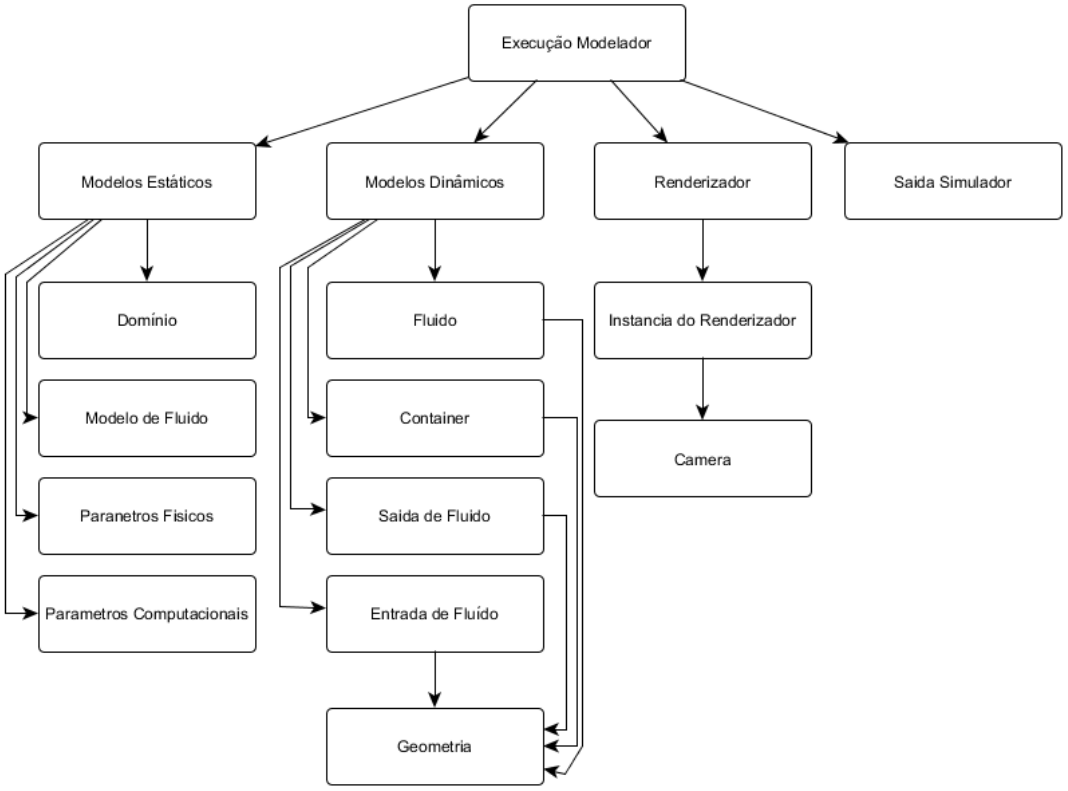


Figura 15: Fluxo de execução guia para o modelador - Gerado a partir da macro-análise das funcionalidades do modelador e das funcionalidades associadas a seu funcionamento (ex.: ramo de *rendering*).

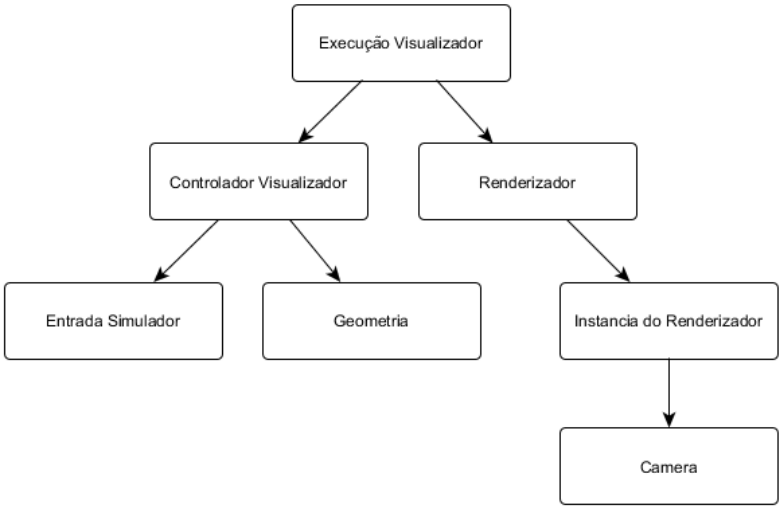


Figura 16: Fluxo de execução guia da visualização – Gerado a partir da macro-análise do funcionamento do visualizador legado do *Freeflow3D*.

É possível concluir, da análise dos fluxos de execução guia criados, que alguns componentes básicos podem ser reutilizados tanto no fluxo de execução do modelador quanto no fluxo de execução do visualizador (exemplos: componentes de geometria e de *rendering*). Embora esse fato evidencie o reuso de componente e de código entre os programas *Freeflow3D*, a grande vantagem do uso de fluxo de execução está mais explicitamente no reuso de componentes e fluxos de execução para resolver problemas similares.

Para ilustrar a evidência do reuso de componentes descrita no parágrafo anterior, o presente trabalho se utilizará de um exemplo. Imagine-se que o usuário do modelador crie uma cena (doravante denominada “Cena 1”) com um contêiner e um fluido (dando origem a um “Fluxo de Execução 1”) e que depois este mesmo usuário deseja criar uma cena bem semelhante (doravante, “Cena 2” e “Fluxo de Execução 2”, respectivamente), só que agora adicionando mais um contêiner na cena. Para tanto, o usuário do modelador, ao invés de recriar a Cena 2 desde o inicio, tendo que configurá-la e setar todos os dados novamente, ele poderá simplesmente abrir o Fluxo de Execução 1 da Cena 1, e utilizar todos os parâmetros já setados na Cena 1, e apenas adicionar as particularidades da Cena 2 (no caso o novo contêiner), criando assim um novo fluxo de execução, o Fluxo de Execução 2 (vide Figura 17 representando este exemplo).

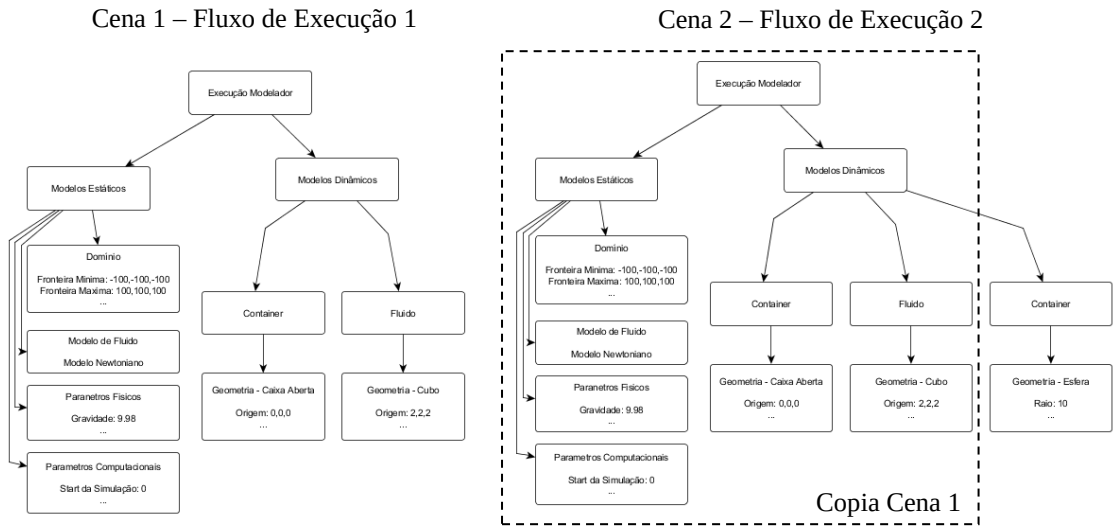


Figura 17: Reuso de fluxo de execuções – No exemplo, o fluxo de execução criado para descrever a Cena 1 foi usado como ponto inicial para a criação da Cena 2. Todos os valores já configurados na Cena 1 foram reutilizados na Cena 2 sem que o usuário tivesse que reentrar todos estes dados.

No exemplo utilizado acima, parte do Fluxo de Execução 1, seus componentes e suas configurações, foram reutilizadas no Fluxo de Execução 2, o que alcança o objetivo do exemplo, que é evidenciar o reuso de componentes entre fluxos de execução, conforme citado no item 5.2.2.

Como já foi visto durante o desenrolar deste trabalho, os fluxos de execução guia gerados definem os componentes básicos e como eles devem ser ligados entre si, ou seja, define a interface de execução. Porém, a interface funcional dos componentes, isto é, a interface que definirá como as funcionalidades e os dados serão acessados (e que deve ser criada de modo a proporcionar implementações diferentes do mesmo módulo), pode ser conferida no próximo item 5.3.2.

5.3.2. Desenvolvimento dos componentes – definição de interface

A partir dos fluxos de execução guia, têm-se definidos os componentes básicos e a sua interface de execução, ou seja, como os componentes devem ser ligados. Contudo, falta definir a interface funcional desses componentes, ou melhor, como suas funcionalidades e seus dados são acessados.

Antes de detalhar o processo de definição da interface funcional, vale relembrar a definição do componente base feita no item 5.2.2, reproduzida em mais detalhes na Figura 13. É possível observar da leitura da definição e da análise da referida Figura, que há uma distinção entre a interface de execução - que define a rotina interna e conecta o componente com seu componente “pai” e componentes “filhos” - e a interface funcional - que acessa os dados internos e executa as funcionalidades.

O componente base, como o próprio nome faz menção, deve ser a base para a construção de todos os componentes básicos definidos pelos fluxos de execução guia e, ainda, deve prover algumas funcionalidades padrão, tais como a definição da rotina interna, das interfaces de execução de entrada (“pai”) e saída (“filhos”) e da interface funcional para

acesso à estrutura de dados flexível (acesso por ID), entre outras funcionalidades básicas (veja definição detalhada do componente base na Figura 18 abaixo)

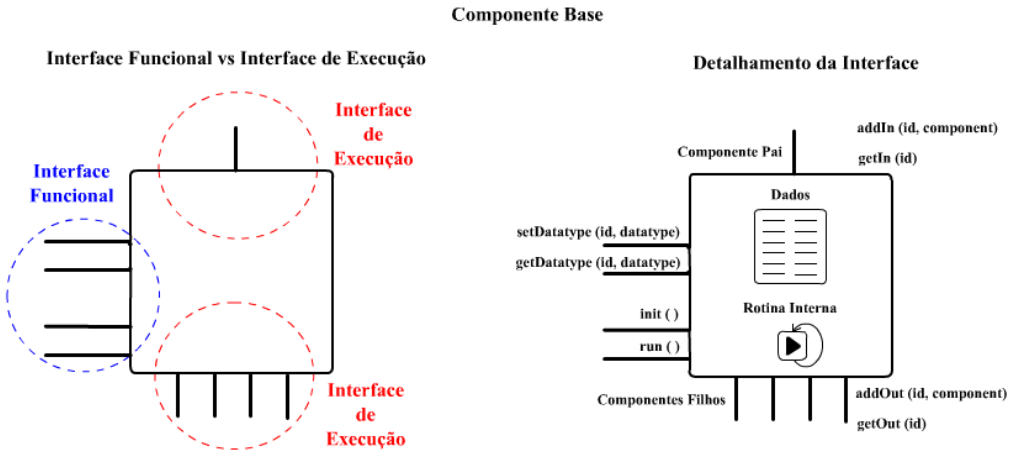


Figura 18: Detalhamento do componente base – À esquerda, o componente base ilustra a diferença entre interface funcional (acesso a funcionalidade e aos dados) e a interface de execução (ligação entre componentes). À direita tem-se em maiores detalhes a interface do componente base, presente em todos os componentes da nova arquitetura.

Tomando como ponto de partida os componentes básicos definidos no item anterior pelos guias, têm-se 20 (vinte) componentes. Tecnicamente, eles são representados por interfaces *Java* e podem ter diversas implementações (o presente trabalho garante, ao menos, uma implementação padrão de cada componente). Influenciados pelo seu papel no fluxo de execução, os componentes básicos podem ser separados em quatro grupos descritos abaixo (vide também a Figura 20):

- **Componentes Funcionais:** são componentes que executam uma funcionalidade pré-definida através de sua interface funcional. Podem ter dados internos que são alterados via interface funcional e manipulados através da rotina interna ou de *listeners,* chamados a cada mudança de valor. São exemplos desses componentes o "*Renderer*" (interface *IRenderer*), que tem a função de gerenciar e realizar o *rendering* da cena, e o "Geometria" (interface *IGeometry*), que pode definir diversas implementações de geometria, sólidos e comportamentos, entre outros. Podem fazer parte desses componentes alguns subcomponentes que

definem funcionalidades específicas e complementares ao componente básico (vide Figura 19 abaixo).

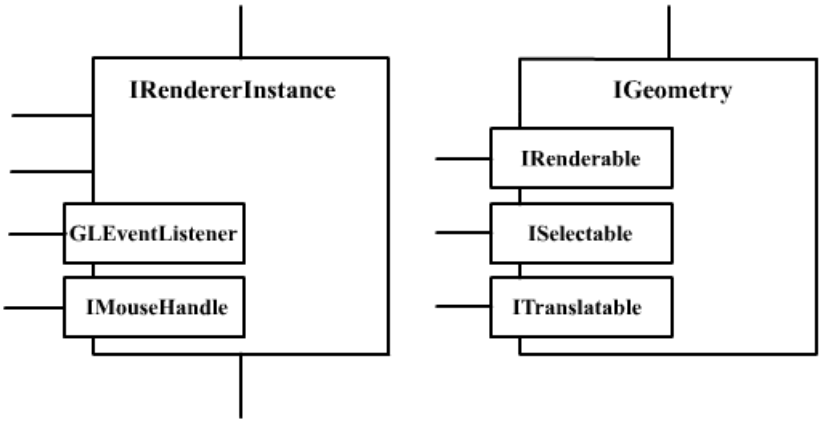


Figura 19: Figura representando subcomponentes dentro de componentes adicionando interfaces funcionais – No caso do componente *IRenderInstance*, o subcomponente *GLEventListener* adiciona funcionalidades de acesso ao *OpenGL* e o subcomponente define interface funcional para lidar com eventos do mouse. No caso *IGeometry*, o subcomponente *IRenderable* define a interface funcional para a geometria ser desenhada na cena pelo *renderer,* o *ISelectable* define a interface necessária para a geometria ser selecionada via mouse e o *ITranslatable* define a interface para transladar o objeto da cena.

- **Componentes de Entrada de Dados:** são componentes mais simples que os componentes funcionais do item anterior. Eles não possuem uma funcionalidade específica, mas representam uma parte semanticamente coesa da aplicação e possuem dados relacionados ao papel representado na aplicação que devem ser informados pelo usuário. Um exemplo é o componente básico "Parâmetros Físicos" (interface *IPhysicalParameters*), que deve agregar todas as informações necessárias referente aos parâmetros físicos para então informar ao Núcleo do *Freeflow3D*. É importante ressaltar, ainda referente ao exemplo dado, que pode haver adições de parâmetros físicos em uma possível evolução ou cenário do modelador e, nesse caso, pode haver uma nova implementação do mesmo

componente disponível para o usuário final na hora de montar um fluxo de execução.

- **Componente de Controle:** são componentes que não tem funcionalidades nem dados, mas servem para organizar o fluxo de dados semanticamente e prover controle para que toda a arquitetura funcione corretamente. Eles implementam somente a interface de execução e são responsáveis por garantir que todos os componentes necessários à execução sejam definidos no fluxo e agrupados segundo sua funcionalidade. São exemplos desses componentes o de “Execução” (interface *IExecution*), que é o ponto de partida para execução dos fluxos, o de “Modelos Estáticos” (interface *IStaticModel*), que agrega os componentes que definem os dados estáticos da cena para simulação, entre outros.

- **Subcomponentes de Auxílio:** definem interfaces funcionais para comportamento presente em mais de um componente básico. Um exemplo é o subcomponente de “Objeto Renderizável” (interface *IRenderableObject*), que pode ser adicionado a qualquer componente base que queira ser tratado e *renderer* por uma instância do *renderer*.

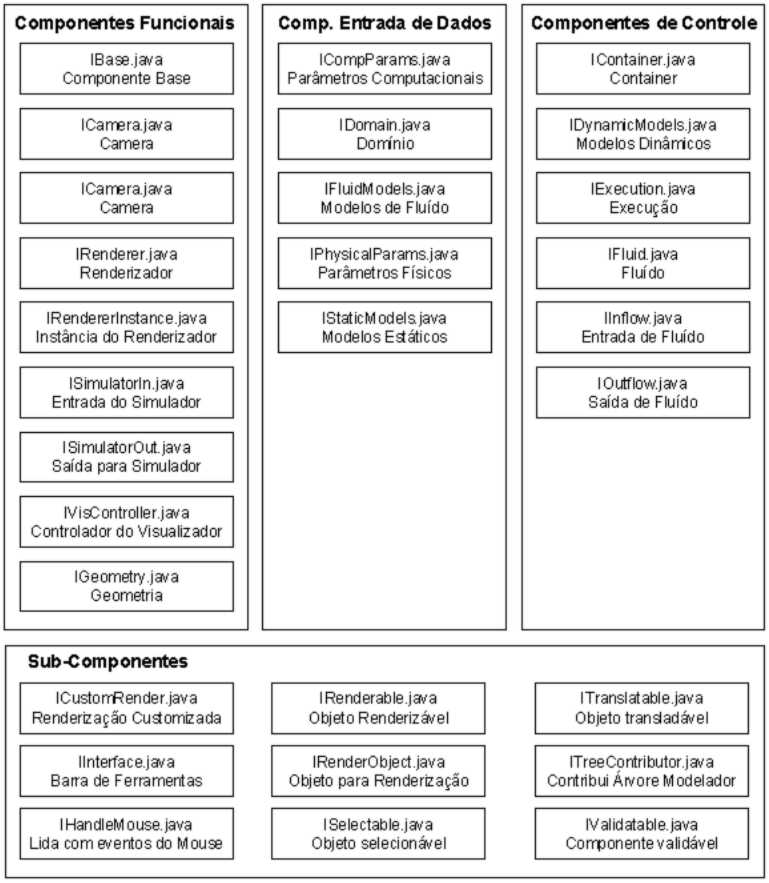


Figura 20: Classificação dos componentes – São classificados em quatro grupos (componentes funcionais, de entrada de dados, de controle e subcomponentes), segundo sua função.

A Figura 20 acima define e classifica todos os componentes e subcomponentes criados para a aplicação da nova arquitetura na reestruturação do *Freeflow3D* até o presente momento.

Com todos os componentes e suas interfaces definidas, iniciou-se a etapa de desenvolvimento de suas implementações. Na Figura 21 (abaixo) há um exemplo interessante da implementação de componentes funcionais, com duas implementações diferentes (e com comportamento também diferentes) para o mesmo componente básico.

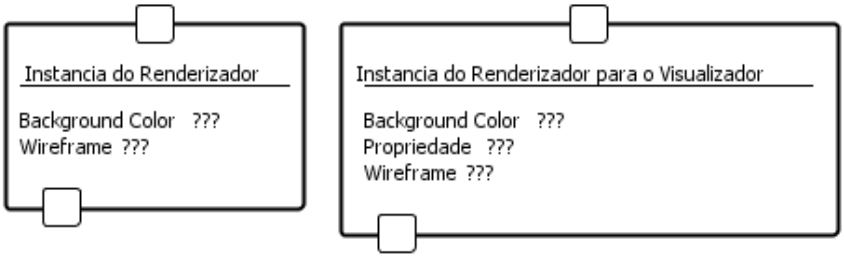


Figura 21: Duas implementações para o mesmo componente básico – Esses dois componentes representados na Figura são implementações do componente básico *IRendererInstance*. O da esquerda é a implementação default e cuida do desenho dos objetos da cena. O da direita exerce o mesmo papel que o componente da esquerda, porém o usuário pode informar uma propriedade para ser apresentada como textura no fluido, ou seja, uma nova funcionalidade se comparada com a implementação padrão. Na montagem do fluxo de execução do visualizador, o usuário pode escolher qualquer uma dessas implementações caso o componente pai tenha uma interface de execução para um componente do tipo *IRendererInstance*.

Com os componentes definidos através de suas interfaces e com as implementações padrão para cada um dos componentes básicos, o próximo passo é ter uma forma para possibilitar que o desenvolvedor possa ligar as implementações criadas aos componentes básicos e para que o usuário final possa ligar as implementações dos componentes em fluxos de execução para resolver seus problemas. O item 5.3.3 trata de como foi desenvolvido o método de descrição dos componentes e fluxos de execuções para os novos programas do *Freeflow3D*.

5.3.3. Descrevendo os componentes e fluxo de execução - metadados e a reestruturação de dados

A nova arquitetura define que os programas reestruturados do *Freeflow3D* devem ser formados por componentes ligados, formando fluxos de execução. Contudo, a definição dos fluxos de execução guia vistos no item 5.3.1, a implementação do componente base, e a definição dos componentes básicos e suas implementações vista no item 5.3.2, não possuem qualquer utilidade prática caso não exista um programa que gerencie todos esses elementos, que leia os fluxos de execução definidos pelo usuário, que instancie as implementações dos componentes utilizados por ele, e que faça funcionar o fluxo de

execução, ou seja, que exerça o papel de “coreógrafo” e controlador de toda a execução do programa.

A implementação e o funcionamento deste “coreógrafo” citado acima será abordada no item 5.3.5., que trata do aplicativo de execução de fluxos. Porém, convém adiantar que para o “coreógrafo” funcionar corretamente, é necessário que as implementações dos componentes disponíveis estejam descritas e informadas como um parâmetro de entrada, assim como haja uma descrição do fluxo de execução montado pelo usuário (vide Figura 22 abaixo).

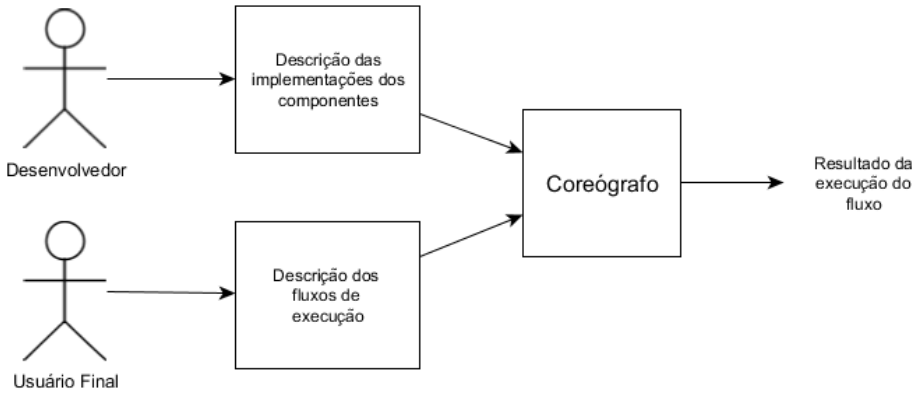


Figura 22: Entradas para o “coreógrafo” – Para seu funcionamento, é necessário que se tenha, como parâmetro de entrada, a descrição das implementações dos componentes, feita pelo desenvolvedor, e a descrição do fluxo a ser executado, feito pelo usuário final.

No programa legado, todas as escolhas realizadas pelo usuário eram gravadas na memória e, uma vez definidas, não eram passiveis de alteração. Esse fato trazia dois inconvenientes ao programa: a) caso o usuário errasse uma entrada, ele deveria reiniciar o programa e refazer todo o processo; e b) não era possível gravar o estado atual das escolhas do usuário para reutilizá-las em uma nova execução do programa. Para contornar esses problemas, a nova arquitetura lançou mão do uso de fluxos de execução, como já foi largamente explicado e exemplificado nos itens anteriores.

Porém era necessário que a definição dos fluxos de execução na nova arquitetura fosse a mais transparente possível, e que também fosse intuitiva e de fácil edição para o usuário final. Logo, baseado no estudo e na análise dos benefícios do uso de linguagens de marcação, principalmente o formato XML (visto no item 4.1.2), foi definido que na

reestruturação do programa o dado de entrada do usuário que antes era armazenado na memória e não era passível de edição passaria a ser gravado em arquivos XML. Esse arquivo XML deve descrever as escolhas do usuário na definição do fluxo de execução e na definição dos parâmetros de cada componente, ou seja, representará as escolhas que eram feitas pelo usuário no programa legado e garantirá todas as vantagens do uso do XML (flexibilidade na representação de dados e estruturas complexas, facilidade de edição e evolução, etc.).

Vale apontar que a dita facilidade de reuso das definições de fluxos de execução (comportamento visto na Figura 17) em arquivos de XML, principalmente como uma versão inicial para resolver problemas semelhantes, é garantida pela simples duplicação do arquivo original.

Como descrito no inicio do presente item, o programa “coreógrafo” deve receber como entrada as definições das implementações dos componentes básicos e as definições do fluxo de execução pelo usuário. A entrada do usuário foi reestruturada para ser informada via “arquivo formato XML” de definição de fluxos de execução. Para facilitar o trabalho do desenvolvedor na definição das implementações dos componentes básicos que podem ser utilizados pelo usuário final na montagem dos fluxos de execução, também foi criado um arquivo XML de descrição de componentes.

Esses dois arquivos XML definirão as entradas para o “coreógrafo”, e este será o responsável que fará com que a nova arquitetura funcione. Na Tabela 2 pode ser conferido um exemplo de definição de fluxo de execução utilizando a estrutura criada no arquivo XML, com explicações para cada *tag xml*. Já a Tabela 3 traz o detalhamento do arquivo XML para definição das implementações dos componentes.

Exemplo de descrição de implementação de componente no arquivo XML

```
<work-flow-def>
   <wf-block>
      <name>Instancia do Renderer</name>
      <base>IRendererInstance</base>
      <impl>DefaultRendererInstanceImpl</impl>
      <class>br.com.ft.plugin.defaultImpl.DefaultRendererInstanceImpl</class>
      <ui-style>PropertySheetStyleUI</ui-style>
      <interface>
         <in id="renderer" type="IRenderer" name="Renderer" required="true"/>
         <out id="backgroundColor" type="Color" name="Background Color" required="true"/>
         <out id="camera" type="ICamera" name="Camera"/>
         <out id="wireframe" type="Boolean" name="Wireframe"/>
      </interface>
   </wf-block>

</work-flow-def>
```

Definição das *tags* do arquivo XML	
`<work-flow-def>`	Abrange as definições de implementações da arquitetura. Pode ter uma ou mais definições de implementações, denominadas <*wf-block*>
`<wf-block>`	Define uma implementação de componente.
`<name>`	Define o nome do componente
`<base>`	Define qual é a base do componente, isso é, qual o componente básico que ele implementa, ou em termos mais técnicos, a interface *Java* que o componente implementa
`<class>`	Qual a classe *Java* da implementação
`<ui-style>`	Define qual é a forma de apresentação do componente e seus dados na interface gráfica gerada automaticamente. Pode ser *PropertySheetStyleUI* ou *OutlookStyleUI*. (veja mais detalhes sobre essas opções no item 5.3.6, sobre os mecanismos desenvolvidos).
`<interface>`	Define a interface de execução, ou seja, com quais componentes essa implementação é ligada, como também define a interface funcional de definição de dados. Essas definições são feitas por *tags* internas <*in*> e <*out*>.
`<in id="" type="" name="" required=""/>`	Defina a interface de execução de entrada, ou seja, quem é o pai do componente. Possui alguns atributos de configuração: • *Id*: código que identifica a interface • *Type*: componente básico (interface *Java*) esperada • *Name*: nome da interface (para exibição) • Required (*true*/*false*): se é uma interface obrigatória para executar o fluxo. É opcional.
`<out id="" type="" name="" required=""/>`	Defina a interface de execução de saída, ou seja, quem são os filhos do componente. Também pode definir uma interface funcional de dados para a estrutura de dados flexível. Possui alguns atributos de configuração: • *Id*: código que identifica a interface • *Type*: componente básico (interface *Java*) esperada ou tipo de dado esperado (ex.: *Integer*, *String*, etc.) • *Name*: nome da interface (para exibição) • *Required* (*true*/*false*): se é uma interface obrigatória para executar o fluxo. É opcional
`<list-out id="" types="" name=""/>`	Defina a interface de execução de saída, podendo receber mais de um componente. Possui alguns atributos de configuração: • *Id*: código que identifica a interface • *Types*: componentes básicos (interfaces *Java*) esperadas. • *Name*: nome da interface (para exibição)

Tabela 2: Exemplo de descrição de implementação de componente em XML – No trecho do exemplo, segue a definição da instância de *renderer* padrão do sistema. Na parte inferior da Tabela as *tags* do arquivo XML são detalhadas.

Exemplo de descrição de fluxos de execução

```
<ui-def-blocks>
   <ui-block>
      <ui-id>12</ui-id>
      <name>Instancia Renderer Basico</name>
      <base>IRendererInstance</base>
      <impl>DefaultRendererInstanceImpl</impl>
      <list-values>
         <value id="execution" type="IExecution">1</value>
         <value id="backgroundColor" type="Color">255|255|255</value>
         <value id="wireframe" type="Boolean">true</value>
      </list-values>
   </ui-block>
</ui-def-blocks>
```

Definição das *tags* do arquivo XML

`<ui-def-blocks>`	Define os componentes que fazem parte do fluxo de execução.
`<ui-block>`	Define um componente do fluxo.
`<ui-id>`	Define o *ID* do fluxo. Esse *ID* deve ser um número seqüencial único para cada componente.
`<name>`	Nome do componente, para efeitos de exibição
`<base>`	Componente básico (interface *Java*) que o componente implementa.
`<impl>`	Classe que implementa o componente
`<list-values>`	Lista de valores de interface do componente. Podem ser interfaces de execução (conexões entre componentes, tanto do tipo pai quanto do tipo "filho") ou interfaces funcionais, como entrada de dados.
`<value id="" type="">`	Define o valor de um item da interface e deve usar o mesmo *ID* definido no arquivo descritor de componentes e o tipo de dado, seja ele o componente básico que a ligação representa ou o tipo de dado de entrada. Pode representar uma interface funcional ou de execução: • Interface de execução (valor é o *id – ui-id*) único do componente a quem esta se ligando): *<value id="execution" type="IExecution">1</value>* • Interface funcional (entrada de dados) *<value id="wireframe" type="Boolean">true</value>*

Tabela 3: Exemplo de descrição de um fluxo de execução em XML – No trecho do exemplo, segue a definição do componente do fluxo de execução que representa uma implementação do componente básico *IRendererInstance*. Na parte inferior da Tabela as *tags* do arquivo XML são detalhadas.

As definições encontradas nesses dois arquivos serão muito importantes para o funcionamento dos aplicativos de suporte à arquitetura, sendo eles o aplicativo de desenho dos fluxos de execução e o aplicativo de execução de fluxos ("coreógrafo"), abordados nos itens 5.3.4. e 5.3.5, respectivamente.

5.3.4. Aplicativo de desenho de fluxos de execução - facilitando o uso e reuso dos componentes

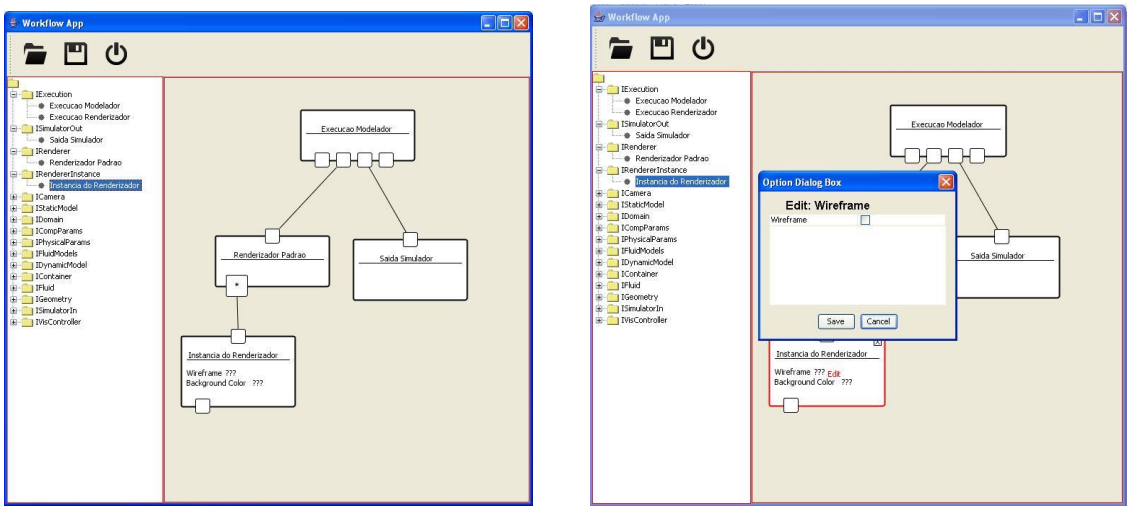


Figura 23: *WorkflowDrawer* em funcionamento.

Como visto nos itens anteriores, a nova arquitetura é formada pelos fluxos de execução de seus componentes. Um programa “coreógrafo” executará os fluxos de execução criados pelos usuários utilizando os componentes implementados pelos desenvolvedores. No item anterior, foram definidos que as duas entradas para o funcionamento do “coreógrafo”, ou seja, a definição das implementações dos componentes e a definição do fluxo de execução criada pelo usuário final, seriam descritas em um arquivo XML.

O arquivo que configura as implementações dos componentes é elaborado pelo desenvolvedor do programa, é um arquivo único contendo as informações sobre todas as implementações disponíveis para o usuário final criar seu fluxo de execução. Contudo, a não ser que sejam criadas novas implementações, esse arquivo permanece o mesmo, isto é, suas versões são incrementais e tendem a ter poucas mudanças.

Por outro lado, o arquivo que define um fluxo de execução criado pelo usuário não tem a mesma característica de versões incrementais, ocorre que o usuário tem a opção de construir fluxos de execuções diferentes para resolver os mais diversos problemas do escopo

do *Freeflow3D*, tanto no modelador quanto no visualizador. Enquanto o arquivo que define as implementações dos componentes funciona como um arquivo de configuração para o "coreógrafo" feito pelo desenvolvedor, o arquivo que define um fluxo de execução funciona como a entrada principal do usuário que definirá o funcionamento de todo o programa.

Embora o arquivo de definição de fluxos seja descrito em um arquivo XML, de fácil entendimento e edição, editar um arquivo de texto para configurar o funcionamento do programa não é a melhor forma de interação com usuário final. Visando melhorar o uso do programa e a interatividade com o usuário final, e, ainda, facilitar a edição, visualização e alteração dos fluxos de execução e, conseqüentemente, dos arquivos XMLs que os descrevem, foi desenvolvido o aplicativo de desenho de fluxos de execução (chamado de *WorkflowDrawer* – vide Figura 23).

O objetivo principal do *WorkflowDrawer* é facilitar a montagem do fluxo de execução pelo usuário final. Para isso, ele provê uma representação gráfica de cada componente e meios para que o usuário ligue as interfaces de execução dos componentes e insira dados via interface funcional, formando um fluxo de execução. Ademais, o *WorkflowDrawer* provê meios de criação e edição de fluxos de execução, ou seja, é possível abrir fluxos de execução já criados e alterá-los criando novos fluxos e facilmente utilizar fluxos de execução já criados e configurados como ponto de partidas para resolução de novos problemas.

A grande vantagem desse programa, além de facilitar a montagem e edição de fluxos pelo usuário, é a facilidade com a qual o desenvolvedor o configura. Utilizando o mesmo arquivo XML de descrição das implementações que serve de entrada para o "coreógrafo", o *WorkflowDrawer* se auto-configura, criando as representações gráficas dos componentes disponíveis, listando suas propriedades para visualização e alteração, validando as ligações entre os componentes, etc. Portanto, caso uma implementação de componente seja criada, o desenvolvedor deverá alterar o arquivo de descrição de implementações para o funcionamento do "coreógrafo" e esse mesmo arquivo serve como entrada para o *WorkflowDrawer*, configurando-o e disponibilizando a nova implementação para o usuário final. Na Figura 24 abaixo é possível conferir todo o processo, desde a definição dos componentes, passando pela montagem das suas representações gráficas, alteração de propriedade e, por fim, a geração do arquivo descritivo do fluxo criado.

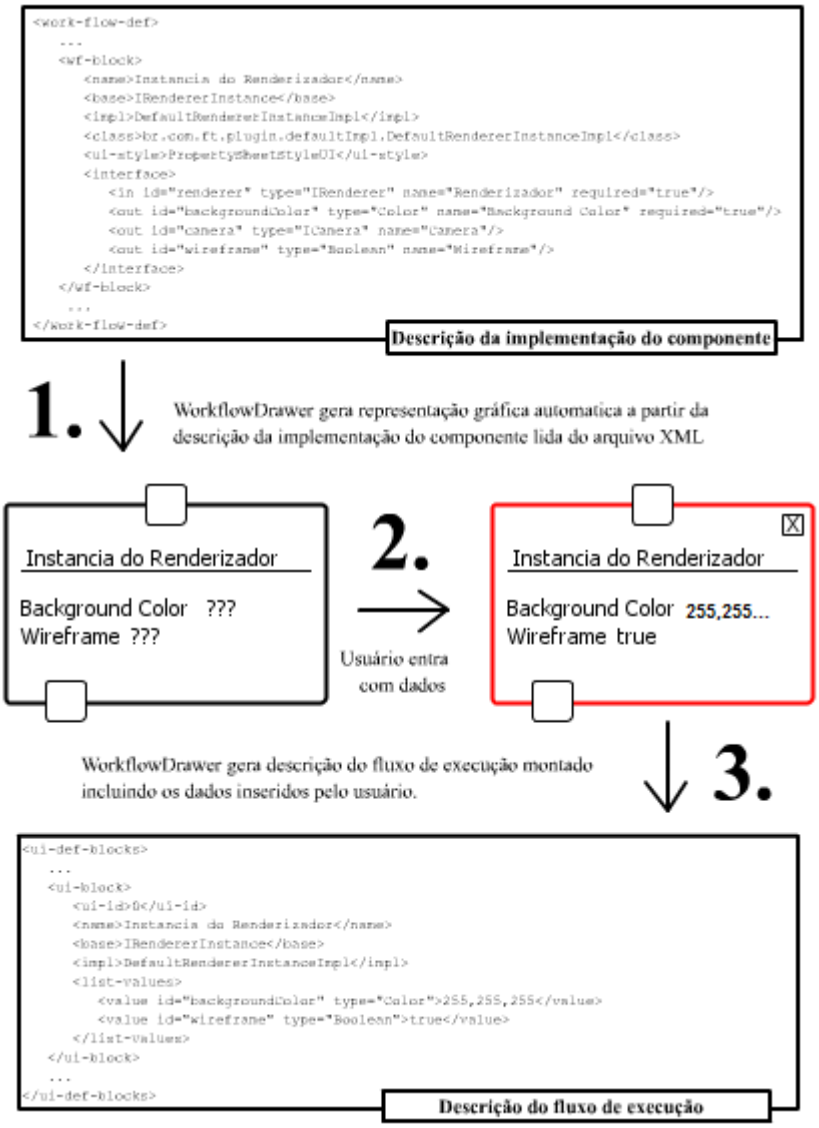
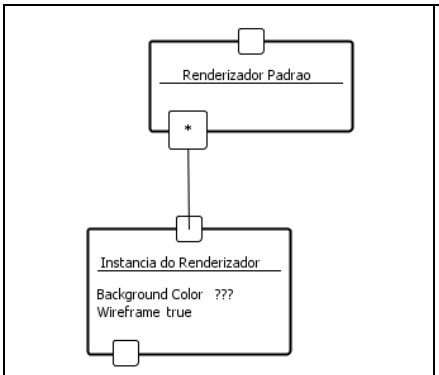


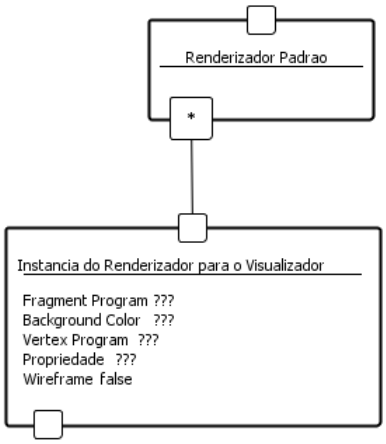
Figura 24: Esquema de funcionamento do *WorkflowDrawer* – Desde a definição da implementação do componente, geração automática da representação gráfica, uso do componente pelo usuário para gerar o fluxo de execução, entrada de dados pelo usuário e gravação da descrição do fluxo de dados.

Com o uso do programa, também se torna simples a operação de troca de implementações. No item 5.5 será abordado o estudo de caso prático de troca de implementações, no caso de instâncias de *renderer*es, mudando da implementação padrão que utiliza as funções padrões do *Opengl* para uma implementação que utiliza programas *shaders* customizados. Para o usuário final, funciona da seguinte forma, caso ele queira trocar a implementação da instância do *renderer*, ele deve abrir o fluxo de execução, remover o componente com a implementação padrão e colocar no lugar o componente

substituto, no caso do exemplo, a instância de *rendering* com *shaders*. Vide processo de troca de componentes na Figura 25 (abaixo).




```
<ui-def-blocks>
   <ui-block>
      <ui-id>0</ui-id>
      <name>Renderer Padrao</name>
      <base>IRenderer</base>
      <impl>DefaultRendererImpl</impl>
      <list-values>
         <value id="renderers"
                type="IRendererInstance">1</value>
      </list-values>
   </ui-block>
   <ui-block>
      <ui-id>1</ui-id>
      <name>Instancia  do Renderer</name>
      <base>IRendererInstance</base>
      <impl>DefaultRendererInstanceImpl</impl>
      <list-values>
         <value id="renderer" type="IRenderer">0</value>
         <value id="wireframe" type="Boolean">true</value>
      </list-values>
   </ui-block>
</ui-def-blocks>
```




```
<ui-def-blocks>
   <ui-block>
      <ui-id>0</ui-id>
      <name>Instancia do Renderer
                para o Visualizador</name>
      <base>IRendererInstance</base>
      <impl>DefaultVisShaderRendererInstanceImpl</impl>
      <list-values>
         <value id="renderer" type="IRenderer">1</value>
      </list-values>
   </ui-block>
   <ui-block>
      <ui-id>1</ui-id>
      <name>Renderer Padrao</name>
      <base>IRenderer</base>
      <impl>DefaultRendererImpl</impl>
      <list-values>
         <value id="renderers"
                type="IRendererInstance">0</value>
         <value id="wireframe" type="Boolean">false</value>
      </list-values>
   </ui-block>
</ui-def-blocks>
```

Figura 25: Exemplificação da troca de componentes – No quadro superior, tem-se um fluxo de execução sendo montado no *WorkflowDrawer* no lado esquerdo e na direita sua representação em XML. No quadro inferior, o componente que representa a instância do *renderer* do fluxo de execução do quadro acima foi trocada por outra implementação, com diferentes dados e funcionalidades (note a mudança na descrição do fluxo de execução). Porém, como ambos os componentes eram implementações do componente básico *IRendererInstance*, a simples troca de componente já basta para que o fluxo de execução tenha um novo *renderer* em sua próxima execução.

5.3.5. Aplicativo de execução dos fluxos - provendo execuções personalizadas e facilidade de acesso aos dados

Partindo do pressuposto, amplamente discutido através dos itens anteriores, de que a nova arquitetura precisa de um programa "coreógrafo", ou seja, um programa que possa lidar com os dois principais elementos da reestruturação do *Freeflow3D* (os componentes e os fluxos de execuções), e os faça funcionar em conjunto para resolver um problema, seja ele de modelagem ou visualização, foi necessário transferir o papel de "coreógrafo" da nova arquitetura a um programa chamado *WorkflowRunner*.

O *WorkflowRunner* é o mais importante programa da reestruturação do *Freeflow3D*, isto porque é através dele que os fluxos de execuções definidos pelo usuário serão executados e criarão os resultados esperados de modelagem e visualização.

Na criação do *WorkflowRunner*, foram levantados alguns requisitos que guiaram o seu desenvolvimento, sendo eles essenciais para o sucesso do uso da ferramenta, tanto pelo usuário final, quanto pelos futuros desenvolvedores do *Freeflow3D*, sendo eles os seguintes:

- Ser independente dos componentes: o programa não deve depender dos componentes que fazem parte dos fluxos de execução, muito menos de suas implementações. Essa característica visa dar o máximo de flexibilidade ao desenvolvedor, que poderá criar seu componente sem se preocupar com o programa que o executará. Em suma, o desenvolvimento de novos componentes não implica em alterações no programa, já que o mesmo independe dos componentes e resolve os fluxos de execução dinamicamente em tempo de execução.

- Facilitar o uso dos componentes: o programa deve ter meios que facilitem a interface com os componentes, por meio de interface gráfica. O desenvolvedor do componente não deve se preocupar com a interface gráfica, principalmente com a entrada de dados pelo usuário final, o que significa dizer que o programa deve proporcionar meios para que o desenvolvedor crie interfaces complementares com o

usuário final e que manipule os eventos gerados por ele (eventos de mouse, por exemplo), caso necessário.

- Ter uso intuitivo: o programa deve fornecer meios intuitivos de uso para o usuário final, deve prover uma interface que faça uso de elementos gráficos normalmente presentes em programas largamente utilizados pelo usuário, fazendo com que ele rapidamente se familiarize com as funcionalidades do programa e as assimile. O programa deve principalmente ter uma interface que provenha acesso rápido ao fluxo de execução para edição de valores dos componentes, o que pode ser feito através de sua interface funcional.

Retomando o que foi visto na Figura 23 através da Figura 26 abaixo, o programa *WorkflowRunner* deve receber como entrada as definições das implementações dos componentes e do fluxo de execução montado pelo usuário final. Como visto no item 5.3.3, essas definições necessárias como entrada do programa devem ser escritas em formato XML, seguindo o padrão definido.

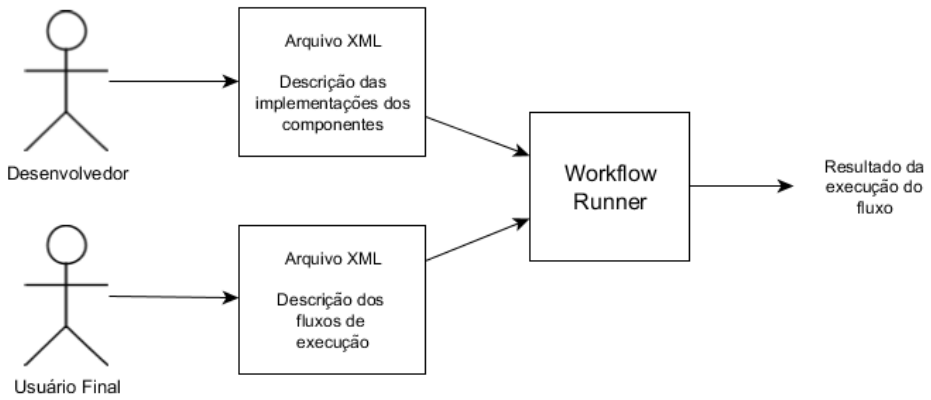


Figura 26: Representação das entradas do *WorkflowRunner* – Os arquivos XMLs descrevendo as implementações dos componentes disponíveis (pelos desenvolvedores) e o fluxo de execução a ser executado (e criado pelo usuário final).

Como citado nos requisitos originais do programa, o *WorkflowRunner* não deve ter ciência dos componentes e das suas implementações, isto é, qualquer tipo de componente, inclusive os que porventura sejam criados, tem de ter seu funcionamento garantido pelo programa. O programa desenvolvido implementa este requisito, mas tem três exceções a essa regra: os componentes *IRenderer*, *IRendererInstance* e *IExecution*. Duas dessas exceções são referentes às classes de *rendering* 3D e acontecem porque, por ser um programa que usa a *rendering* como principal interface de resultado, ele deve ter conhecimento do *renderer* configurado no fluxo de execução para exibir seu resultado na área principal do programa. Já o componente *IExecution* define o ponto de partida dos fluxos de execuções e é como o programa sabe por onde iniciar a execução dos fluxos.

Ao ser iniciado, o *WorkflowRunner* abre o fluxo de execução criado pelo usuário e, com o auxílio do arquivo descritor das implementações dos componentes, instância dinamicamente cada componente, os liga formando o fluxo de execução e seta os dados já definidos pelo usuário (presentes no arquivo de fluxo de execução). Essa instanciação dinâmica do fluxo de execução, baseado em descrições textuais sobre quais implementações de componentes foram utilizadas e como elas estão arranjadas, só é possível por causa do uso de *Reflection* do *Java*, que é a funcionalidade da linguagem capaz de procurar e instanciar objetos dinamicamente e em tempo de execução baseado na definição de suas classes *Java*. Veja a Figura 27 (abaixo) que ilustra o processo.

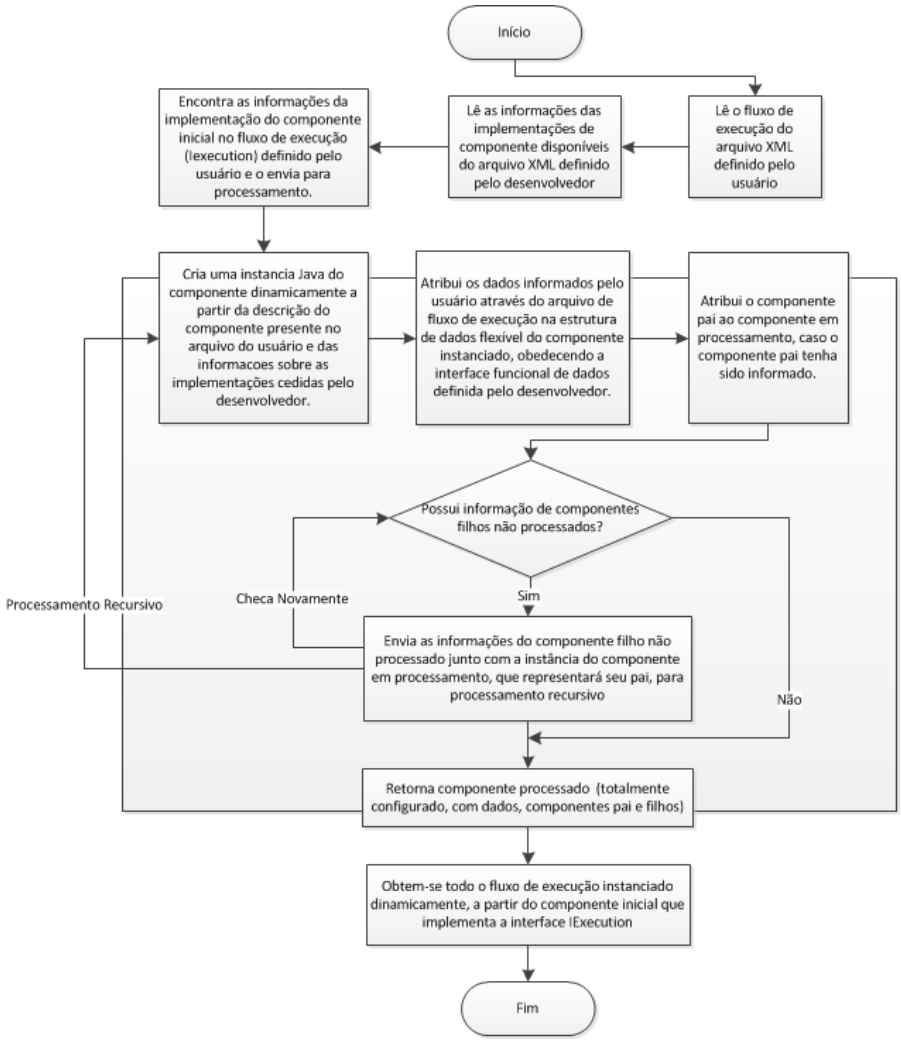


Figura 27: Fluxos de Execução – Fluxograma explicativo da criação dinâmica dos fluxos de execução.

Com o fluxo de execução montado em tempo de execução, o *WorkflowRunner* cria a interface gráfica para acesso às estruturas de dados dos componentes, processo este que também independe dos componentes e acontece de forma automática. A partir da descrição da interface de dados, tanto no arquivo de descrição de componentes, quanto no arquivo de descrição dos fluxos de execução, o programa cria meios de acesso, levando em conta o tipo de dado para que o usuário possa inserir ou alterar valores em tempo de execução. A Figura 28 (abaixo) ilustra o processo e mais detalhes sobre essa funcionalidade podem ser conferidos no item 5.3.6, que trata sobre os mecanismos desenvolvidos.

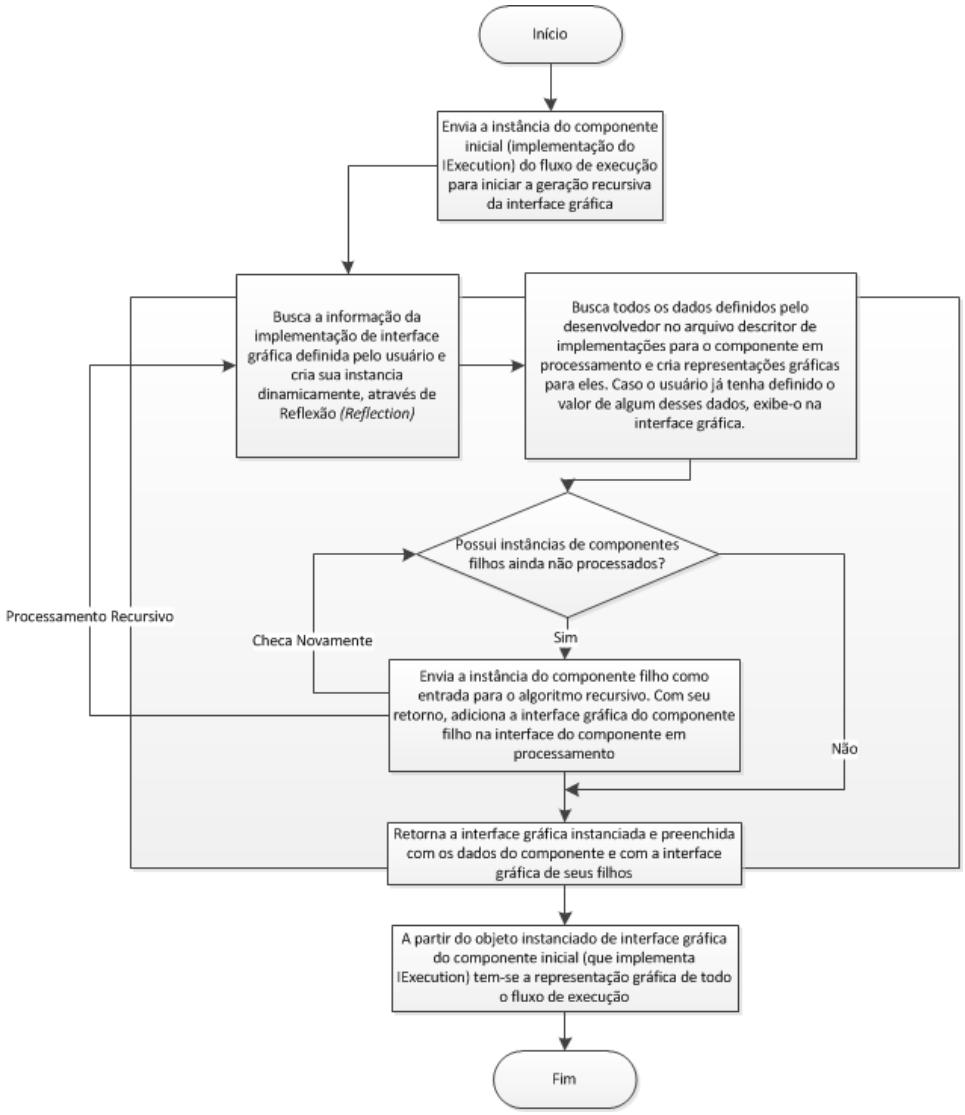


Figura 28: Interface gráfica – Fluxograma explicativo da criação automática de interface gráfica

Alguns componentes podem utilizar sub-componentes disponibilizados pelo "coreógrafo", e estes sub-componentes disponibilizam meios para que o componente acesse ou modifique o *WorkflowRunner*, caso seja necessário. Os exemplos mais utilizados para exemplificar a situação são o *IInterface* (cria barra de ferramentas), *IMouseHandle* (lida com os eventos do mouse), GLEventListener (instância do *renderer OpenGL* criado pelo JOGL [31]), *IValidatable* (interface para *feedback* de validação de cena), entre outros. O item 5.3.6, sobre os mecanismos criados, descreve como alguns desses sub-componentes funcionam.

Uma vez instanciado dentro do programa o fluxo de execução definido pelo usuário e criada a interface gráfica para acesso aos componentes, o *WorkflowRunner* inicia as *Threads* de execução da rotina interna do fluxo de execução (explicado no item 5.2.2) e também da

rotina de *rendering*. A freqüência com que essas rotinas são executadas podem ser configuradas via interface gráfica (vide Figura 29 abaixo) e podem ser ajustadas pelo usuário final para garantir o melhor nível de performance.

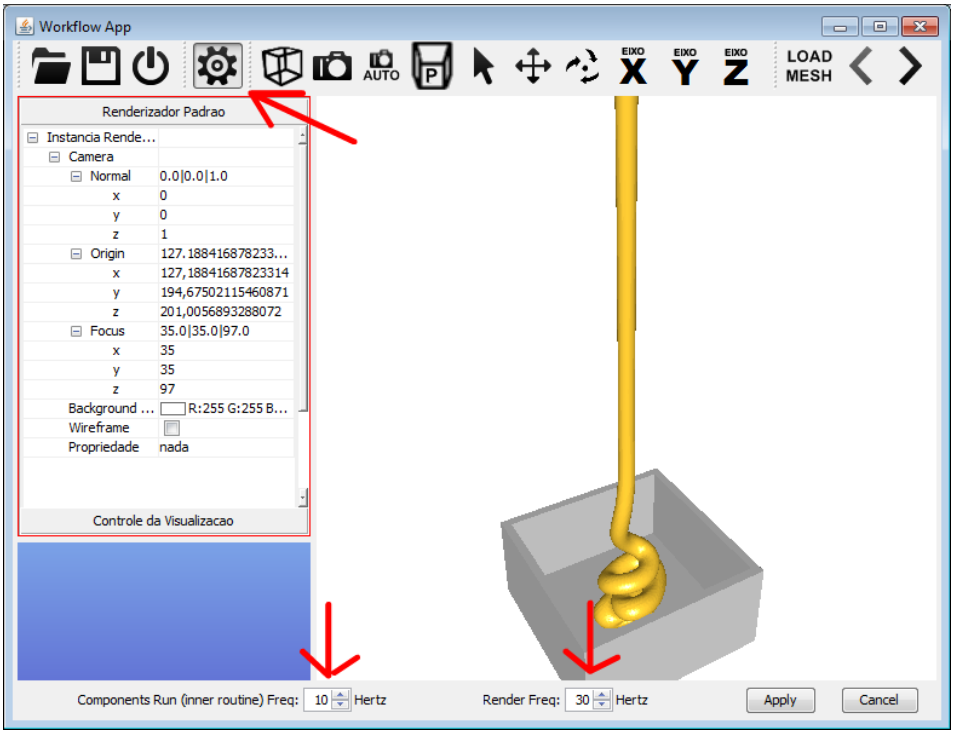


Figura 29: Alteração da freqüência de execução de *threads* – Clicando no botão indicado pela seta, é aberta a caixa na parte inferior da tela, na qual pode-se alterar a freqüência da ciclo de execução do fluxo de execução como também a freqüência de *rendering*.

Com o desenvolvimento do *WorkflowRunner*, a execução de fluxos de execução, tanto de modelagem quanto de visualização do *Freeflow3D*, é possível, já que ele provê mecanismos para que os fluxos de execuções sejam montados dinamicamente, e, através da execução dos seus componentes, gerem o resultado esperado pelo usuário final. No próximo item 5.3.6 é possível conferir alguns dos mecanismos criados para o *WorkflowRunner* e que são relevantes para o funcionamento da nova arquitetura. Já no item 5.4 seguinte, os dois programas reestruturados do *Freeflow3D*, seu modelador e visualizador, são apresentados rodando no *WorkflowRunner* a partir de seus respectivos fluxos de execução guia, demonstrando todas as suas funcionalidades e características implementadas e executadas a partir dos componentes criados.

5.3.6. Mecanismos desenvolvidos

Durante o desenvolvimento dos elementos que compõe a nova arquitetura (que foram utilizados durante toda a reestruturação do *Freeflow3D)*, desde componentes, fluxos de execuções e aplicativos de suporte ao funcionamento da arquitetura, alguns mecanismos foram criados e, posteriormente, mostraram-se essenciais para o sucesso de todo o trabalho realizado.

Por serem mecanismos relevantes ao funcionamento dos programas reestruturados, segue abaixo uma breve descrição de suas principais características e de como eles funcionam:

- **Componente base e estrutura de dados flexível:**

O componente base é o ponto de partida para qualquer implementação de componente na nova arquitetura. Sua estrutura básica já foi detalhada no item 5.3.2 e na Figura 18, que é reproduzida novamente na Figura 30 (abaixo). É possível observar que o componente base define a interface funcional e a execução dos componentes. A rotina interna do componente deve ser implementada por quem herda o componente base, constituindo um exemplo do padrão de projeto *Template Method*. Outro mecanismo importante implementado no componente base é a estrutura de dados flexível. O componente base, e conseqüentemente todos os componentes do sistema, possui um *HashMap* que armazena todos os dados, é indexado por um *id* em formato *string* e acessado por via das interfaces funcionais *setDatatype* (*id, datatype*) e *getDatatype* (*id, datatype*). Para flexibilizar a definição desses dados, eles são definidos dinamicamente a partir do arquivo XML, que descreve as implementações dos componentes. Logo, para o desenvolvedor criar ou excluir um dado do componente, basta ele alterar o arquivo XML descritor de sua implementação. Toda a interface gráfica para interface com esses dados, tanto no *WorklfowDrawer* quanto no *WorkflowRunner*, é criada automaticamente a partir

da descrição presente no arquivo XML, e tem acesso aos dados garantido através dessa mesma interface funcional (*setDatatype* e *getDatatype*) presente em todos os componentes.

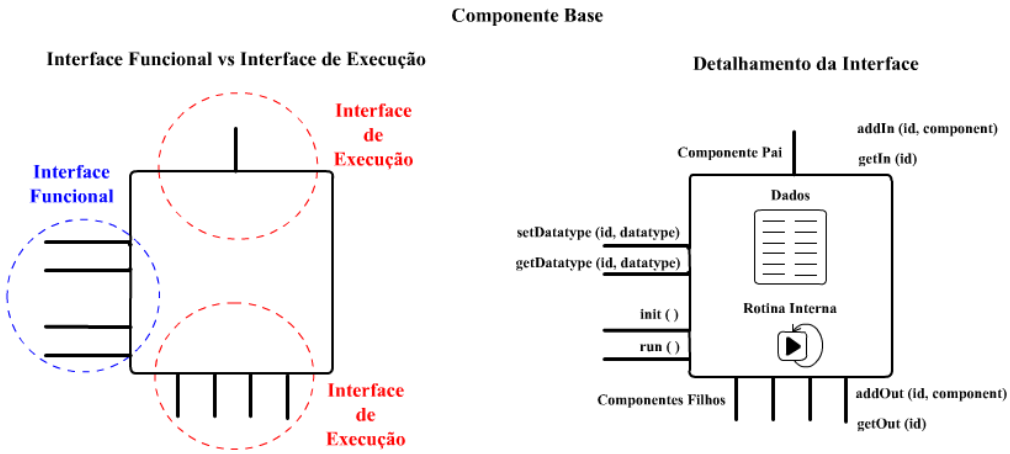


Figura 30: Componente base – Retomando representação do componente base.

- **Geração do fluxo de execução:**

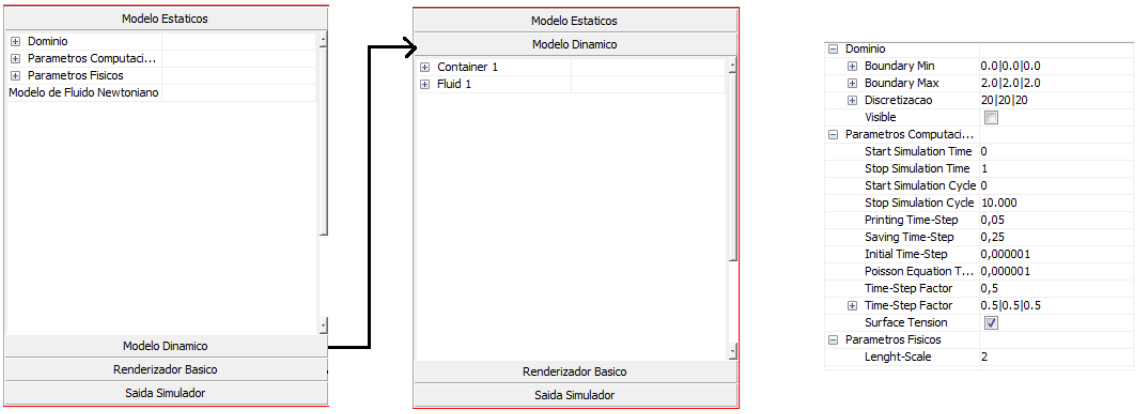
Como descrito no item 5.3.5, o fluxo de execução é resolvido dinamicamente em tempo de execução. A partir da descrição do fluxo de execução presente no arquivo de XML criado pelo usuário final, para cada componente presente no fluxo o programa encontra a sua implementação descrita no arquivo, e a procura na fábrica de implementações (padrão de projeto *factory*) que a instancia dinamicamente através das informações contidas no arquivo XML de descrição de implementações criado pelo desenvolvedor. Esse mesmo mecanismo seta os dados, definidos no arquivo de descrição do fluxo de execução, nos componentes, através da interface funcional disponibilizada pelo mecanismo de estrutura de dados flexível descrito no subitem anterior. Os componentes “filhos” são tratados recursivamente por esse mesmo mecanismo até que todos os componentes que compõem o fluxo de execução estejam instanciados e corretamente ligados, finalizando assim o processo de geração de fluxo de execução.

- **Geração automática de interface gráfica:**

Este mecanismo foi abordado item 5.3.5. Ele serve para gerar uma interface gráfica intuitiva para que o usuário final interaja com os componentes que fazem parte de seu fluxo de execução. Não obstante, ele também serve para auxiliar o desenvolvedor, que não precisa se preocupar em criar uma interface gráfica para acesso aos dados e às funcionalidades de seus componentes.

O mecanismo disponibiliza dois tipos de interface gráfica para os componentes (reimplementadas a partir da biblioteca *L2FProd* [27]): a) a *OutlookStyleUI* que trata somente da interface de execução dos componentes (tem representações somente das ligações dos componentes e não para seus dados) e conta com painéis animados e expansíveis; e b) a *PropertySheetStyleUI*, que representa tanto a interface de execução como também a interface funcional dos componentes, provendo acesso aos dados internos e exibindo os componentes "filhos" na forma de itens de árvore expansíveis (vide Figura 31, abaixo, mostrando as duas possibilidades).

Existe uma interface comum compartilhada por essas duas implementações, o que possibilita que novas implementações de interfaces gráficas possam ser criadas no futuro.



OutlookStyleUI *PropertySheetStyleUI*

Figura 31: *OutlookStyleUI* e *PropertySheetStyleUI* – Representação das duas possibilidades de interface: a *OutlookStyleUI* e a *PropertySheetStyleUI*.

Seu funcionamento é baseado na descrição dos dados dos componentes, nos dados atribuídos pelo usuário final no arquivo de descrição do fluxo de execução e em um arquivo que determina quais são as estruturas de dados básicas do sistema. A partir dessas três informações, o *WorkflowRunner* usa suas implementações de interface gráfica disponíveis

(*OutlookStyleUI* e *PropertySheetStyleUI*) e aplica o algoritmo representado pelo fluxo ilustrado na Figura 28.

Os dados alterados na interface gráfica são propagados ao componente através da implementação do padrão de projeto *observer,* e a representação gráfica de cada tipo de dados é resolvida através de uma fábrica de implementações, evidenciando o uso do padrão de projeto *factory*.

- ***Rendering* e gerenciamento de cena:**

O principal meio de interface com o usuário, tanto no modelador quanto no visualizador do *Freeflow3D*, se dá através da representação da cena em 3D. Tal representação 3D é possível no programa através do uso do *OpenGL*, que disponibiliza uma biblioteca com funções primárias de *rendering* 3D acelerados na placa gráfica.

É possível efetuar *rendering* dos objetos de forma caótica, ou seja, manter simplesmente um laço que percorre os objetos e chama as funções *Opengl* (que desenham suas primitivas), ou então utilizar uma estrutura organizada, que mantém controle sobre todos os objetos renderizáveis, os organiza de forma que facilite funcionalidades além da *rendering* propriamente dita (ex.: transformações geométricas como translação, etc.) ou que promova alguma melhora, como, por exemplo, a melhora no desempenho ou na qualidade da imagem gerada.

Para a reestruturação do *Freeflow3D* e a aplicação da nova arquitetura, foram projetados dois componentes e alguns subcomponentes que definem o comportamento básico esperado do *renderer* e quais funcionalidades ele deve possuir. Na Tabela 4 (abaixo) é possível conferir quais são estes componentes e quais são suas responsabilidades na atual solução.

IRenderer	Componente que representa o funcionamento do renderer da solução. Ele gerencia os componentes de instâncias de rendering (IRendererInstance)
IRendererInstance	Componente que define uma instância de rendering. É o responsável por tratar os objetos que se submeterão ao rendering através do uso do Opengl
IRenderable	Sub-componente que define que um componente tem algo para ser processado pela instância de rendering através de uma interface bem definida.
ISelectable	Sub-componente que define o comportamento que o componente deve possuir para ser selecionado via mouse
ITranslatable	Sub-componente que define a interface que deve ser implementada para que o componente possa ter sua posição manipulada através do mouse.

Tabela 4: *Rendering* do sistema - Tabela descrevendo os componentes e sub-componentes básicos (interfaces) criados para resolver o *rendering* do sistema.

A implementação padrão codifica todas as funcionalidades definidas pelos componentes básicos e seus sub-componentes. Seu gerenciador de cena, mesmo não sendo tão avançado, mantém uma árvore da cena e possibilita transladar objetos, além de manter dados sobre as fronteiras de cada objeto (árvore *Axis Align Boundary Box[5]*) para implementação da técnica de *picking*[5] (seleção do objeto na cena 3D através do mouse). Veja na Figura 32 um esquema representativo do funcionamento implementação padrão do gerenciador de cena, presente no componente *DefaultRendererInstance*.

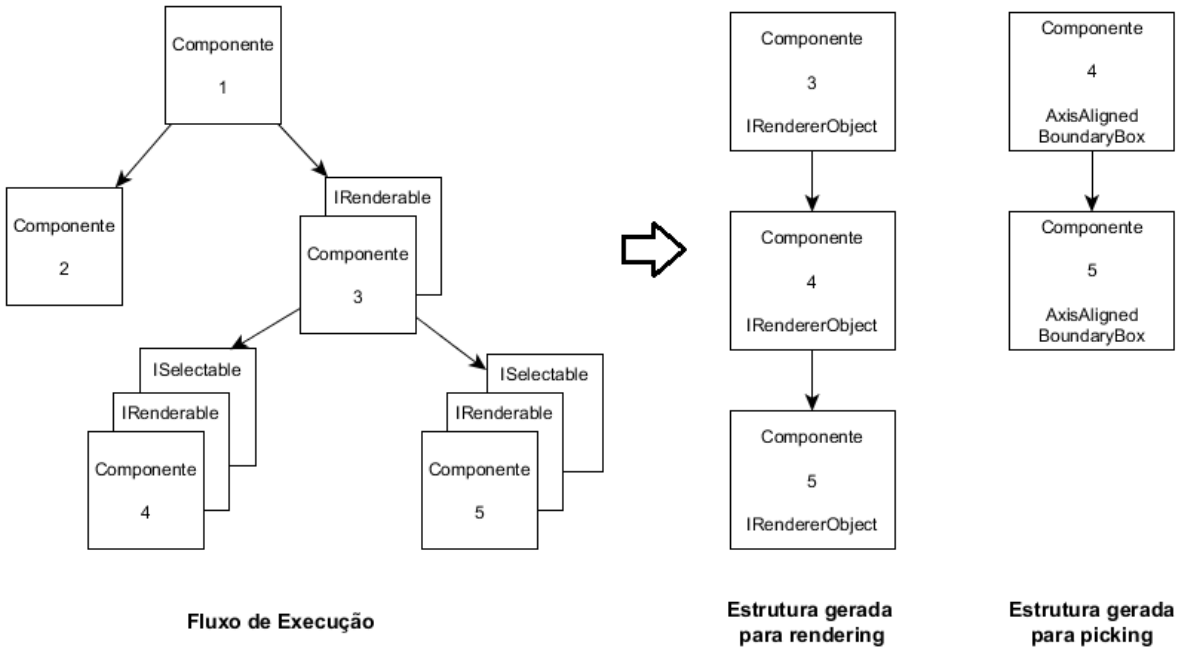


Figura 32: Exemplo do funcionamento do gerenciador de cena – O fluxo de execução é processado gerando estruturas internas que facilitam as funcionalidades do *renderer*, o exemplo da Figura ilustra as estruturas criadas para o processo de *rendering* e de execução da funcionalidade de *picking*.

Todo o sistema de *rendering* foi projetado para que novas implementações dos componentes de *rendering* e gerenciamento de cena possam ser criados no futuro, com novas funcionalidades ou otimizações de *rendering* (como, por exemplo, *Frustrum Culling*[5], entre outras) ou que tragam mais realismo à cena, com novos modelos de iluminação implementados via *shaders*. Um exemplo de *renderer* diferente da implementação padrão pode ser conferida no item 5.5.

O *rendering* dos objetos só é possível por causa do uso do padrão de projeto *composite*, implementado através do subcomponente *IRenderable*. A instância do *renderirg* percorre todo o fluxo de execução à procura de componentes que implementem o subcomponeente em questão, para então utilizar sua interface funcional para efetuar o *rendering* do objeto.

- **Validação de Cena e Barra de Ferramentas:**

Há dois sub-componentes criados para que os componentes possam se comunicar com o programa executor de fluxos de execução, o *WorkflowRunner*, exercem papel

importante na interação do usuário final com o programa, além de prover aos desenvolvedores de componentes ferramentas para implementação das funcionalidades requeridas. São eles o *IInterface* e o *IValidation*.

Caso o componente implemente o sub-componente *IInterface*, ele pode definir botões que serão alocados na barra de ferramenta do programa principal. Na inicialização do *WorkflowRunner*, após o programa montar todo o fluxo de execução, ele irá percorrer todos os componentes procurando por implementações do sub-componente *IInterface* para só então montar, em tempo de execução, a sua barra de ferramenta. Logo, dependendo da escolha dos componentes e da escolha ou não da implementação deste sub-componente, a barra de ferramentas do programa será potencialmente diferente. Veja na Figura 33 (abaixo) o esquema de uso do subcomponente:

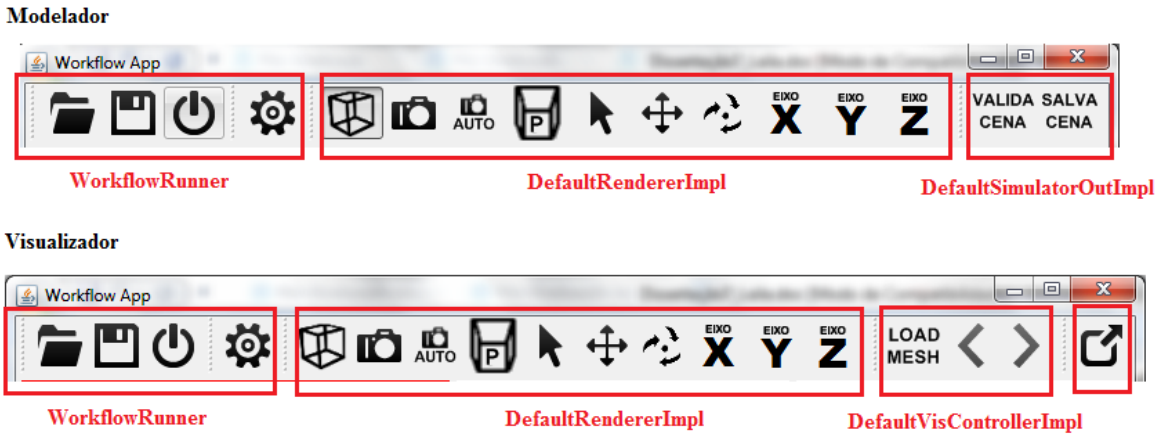


Figura 33: Representação do funcionamento do sub-componente *IInterface* – A barra de ferramentas é definida em tempo de execução pelas implementações dos componentes que compõem os fluxos de execução. Na Figura é possível observar exemplos de barras de tarefas geradas por fluxos de execução para modelagem e visualização, destacando quais implementações são responsáveis pelas barras de ferramentas disponibilizadas para o usuário final.

Com relação ao sub-componente *IValidatable,* ele pode ser utilizado caso o desenvolvedor queira trazer ao usuário final algum tipo de *feedback* sobre suas ações. Um exemplo de *feedback* é a possibilidade do usuário final alterar os dados dos componentes em tempo de execução, o que muitas vezes pode invalidar alguma condição fundamental para o funcionamento do componente e, conseqüentemente, do programa. Através do *IValidatable*, o componente pode enviar ao programa principal uma mensagem de aviso que

será exibida ao usuário na forma de caixas de contexto. Assim que o usuário corrigir a operação que originou a mensagem de erro na validação, o componente, através do sub-componente *IValidatable*, retirará a mensagem de erro. Veja na Figura 34 abaixo o mecanismo em funcionamento:

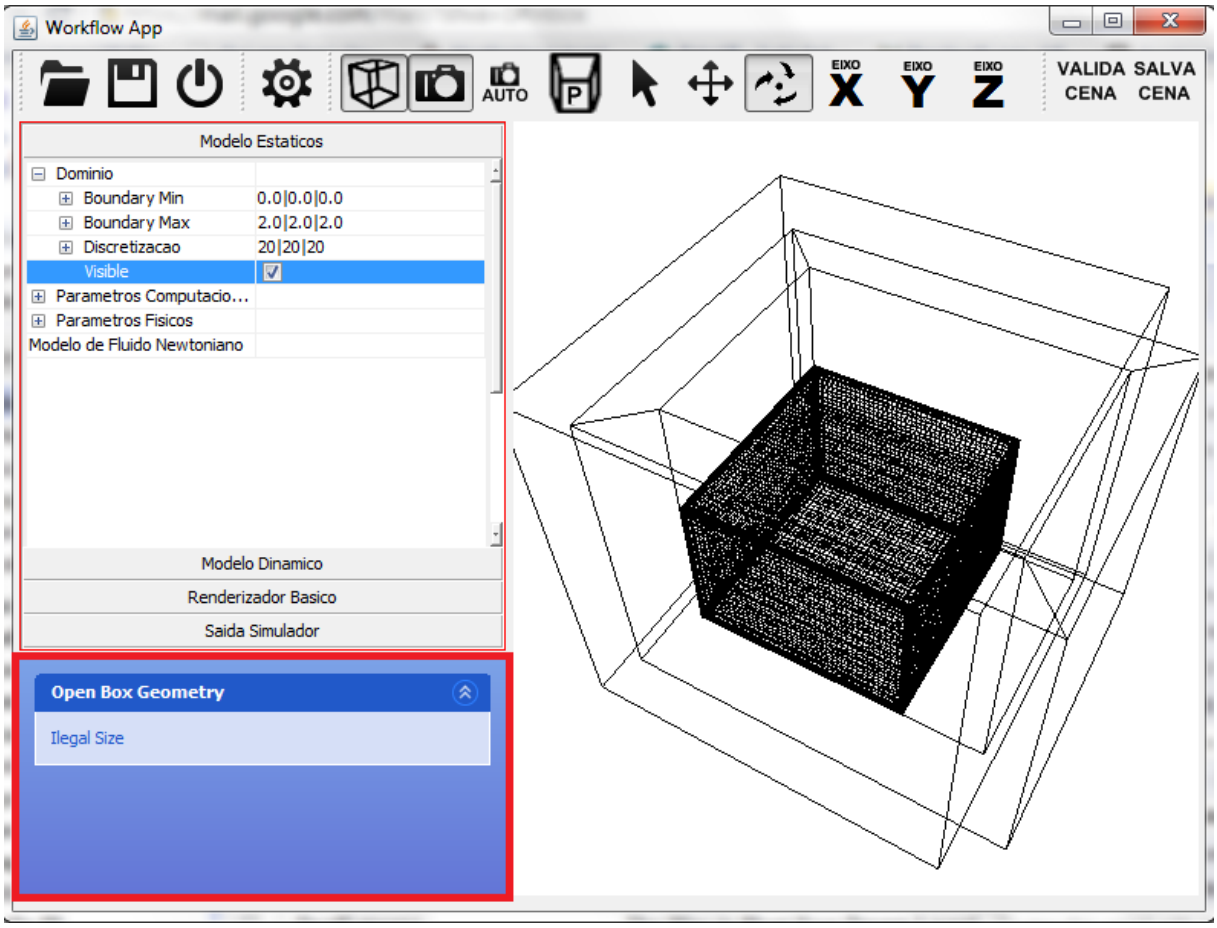


Figura 34: Representação do funcionamento do sub-componente *IValidatable* – Repare que o contêiner *Open Box* (caixa aberta) está fora do domínio. Logo, foi apresentado ao usuário a mensagem "Illegal size", ou seja, "tamanho ilegal", criada pelo sistema de validação através do uso do sub-componente *IValidatable*, que é exibida na caixa destacada em vermelho.

5.4. Execução dos programas reestruturados do Freeflow3D

Como visto ao longo deste capítulo, a nova arquitetura, definida no item 5.2 para reestruturação dos programas do *Freeflow3D,* foi desenvolvida originando fluxos de execuções, componentes, programas e mecanismos auxiliares, tudo para que os novos visualizador e modelador pudessem funcionar sob a nova arquitetura.

O presente item mostra o resultado final da reestruturação, ou seja, apresenta os fluxos de execução guia do visualizador e do modelador sendo executados sob a nova arquitetura, e as suas características e funcionalidades.

5.4.1. Modelador sob a nova arquitetura

Para executar o novo modelador é preciso que o usuário final crie um fluxo de execução de modelagem. Para a realização da referida criação, o usuário pode utilizar o programa *WorkflowDrawer* para desenhar o fluxo de execução, que é iniciado pela implementação do componente *IExecution* para modelagem, o *DefaultModelerExecutionImpl*. Um exemplo de fluxo padrão para modelagem, desenhando no programa *WorkflowDrawer*, pode ser conferido na Figura 35 abaixo:

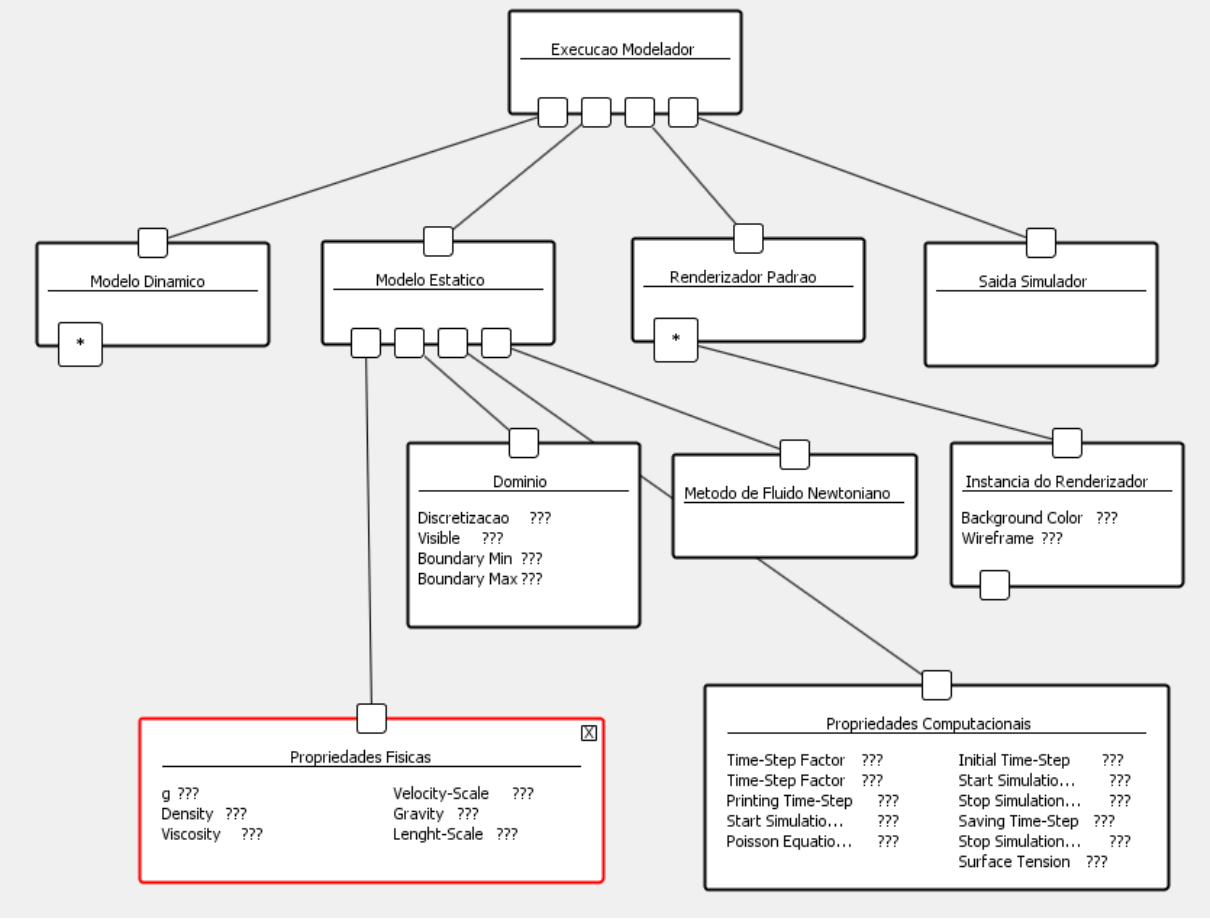


Figura 35: Representação do fluxo de execução da implementação padrão do modelador – Desenhado pelo *WorkflowDrawer* e sem os modelos dinâmicos, que definem a cena.

O usuário pode inserir dados no fluxo de execução através do programa *WorkflowDrawer* ou editando o XML que representa o fluxo de execução. Note na Figura 35 anterior que nenhum modelo foi inserido no fluxo de execução, ou seja, ainda não foi adicionado nenhum fluido, contêiner, entrada ou saída de fluido. Implementações desses componentes podem ser inseridos como "filhos" do componente *IDynamicModel* (implementado por *DefaultDynamicModelImpl*), ou seja, o usuário deve inserir os objetos de acordo com a cena que deseja criar. Para exemplificar o funcionamento do programa, será considerado o fluxo de execução da Figura 35 (anterior) somado a dois modelos, um contêiner aberto (*openbox container*) e um fluido cúbico (*cube fluid*). Veja o fluxo de execução completo do exemplo na Figura 36 (abaixo):

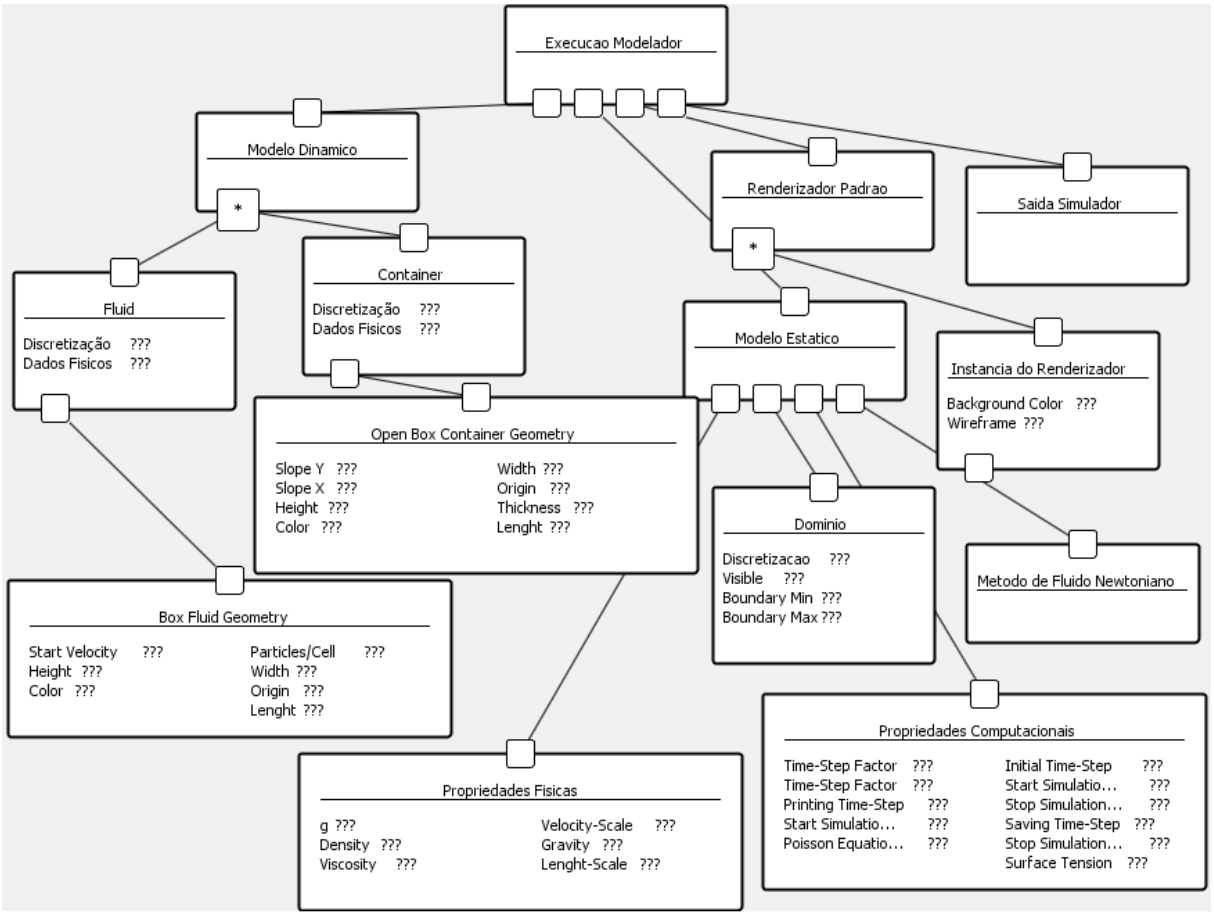


Figura 36: Representação do fluxo de execução completo - Componentes necessários para o funcionamento do modelador somados aos modelos que compõem a cena (dispostos à esquerda na Figura).

O arquivo XML criado representando o fluxo de execução da Figura 36 (anterior), quando executado no programa executor de fluxos, gera o seguinte programa retratado na Figura 37 (abaixo). Observe que do lado esquerdo tem-se a representação do fluxo de execução e de todos os dados de cada componente para possíveis alterações. Ao centro encontra-se a representação 3D da cena definida no fluxo de execução. No topo, encontram-se as barras de ferramentas, com os botões definidos pelos componentes escolhidos.

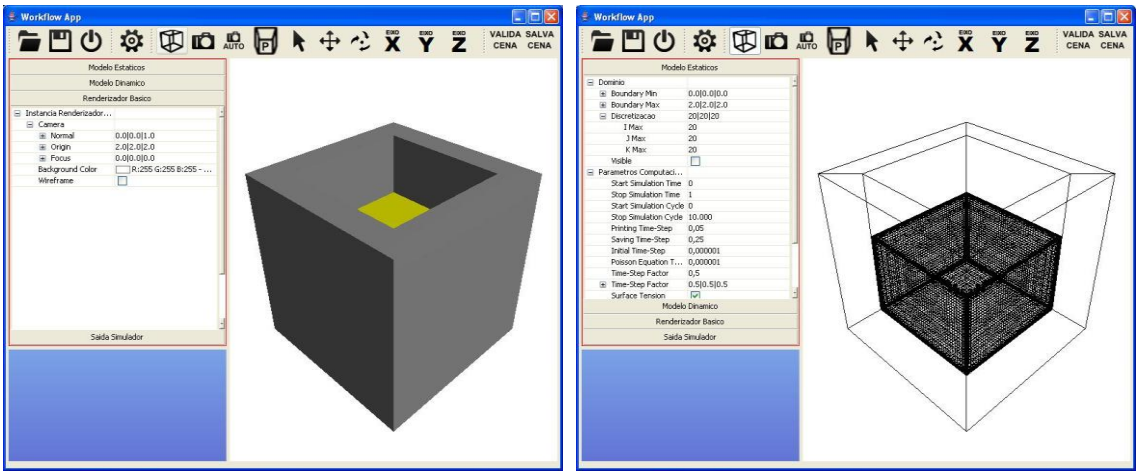


Figura 37: Execução do fluxo de execução para modelagem.

Além das funcionalidades do programa legado, ou seja, as possibilidades que o programa legado possuía de definir os dados iniciais para a simulação, de inserir os modelos e de visualizar o resultado, novas funcionalidades foram criadas no desenvolver do presente trabalho, sendo elas:

- **Interface gráfica criada dinamicamente e facilmente alterável:** com seu funcionamento abordado no item 5.3.5 e 5.3.6, o sistema dinâmico de criação de interface gráfica traz esta nova funcionalidade ao modelador, se comparado com o programa legado. Antes, no programa legado, adicionar um novo parâmetro era uma tarefa complexa, pois era preciso criar todo o código da interface gráfica e ainda adequar seu posicionamento às telas já criadas. No modelador reestruturado, basta alterar o arquivo descritor de implementações do componente adicionando mais uma entrada de dados, após tal adição tanto o dado internamente no componente (através do sistema de estrutura de dados flexível – vide item 5.3.6) como toda a interface gráfica para alterá-lo serão automaticamente criados.

- **Edição de valores em tempo real:** o usuário pode alterar os dados dos componentes em tempo de execução, ou seja, em tempo real, o que é feito através da interface gráfica que representa o fluxo de execução, seus componentes e seus dados. Caso o dado em questão esteja relacionado com a cena (como, por exemplo,

o tamanho do contêiner), a cena é automaticamente atualizada, graças ao funcionamento do padrão de projeto *observer*[17]. Todas as alterações do usuário que possam violar alguma pré-condição para a simulação são necessariamente validadas, como descrito no próximo item.

- **Validação em tempo real:** com a ajuda do sub-componente *IValidatable* (abordado no item 5.3.6), as ações do usuário são validadas em tempo de execução, para garantir que os pré-requisitos da simulação sejam cumpridos. Por exemplo, caso o usuário translade algum objeto para fora do domínio da simulação, uma mensagem de aviso é exibida para ele informando que a cena não se encontra validada. Essa funcionalidade tem como objetivo viabilizar as alterações na cena em tempo de execução, o que não era possível na versão legada.

- ***Picking* (seleção com o mouse):** na versão reestruturada do modelador, o usuário pode selecionar os objetos diretamente na cena com o mouse. Ao clicar na imagem 2D que representa a cena 3D, o ponto do clique é projetado para a cena através das características da câmera, criando um raio com origem no observador da câmera e na direção para o ponto focal da câmera. Caso esse raio intersecte algum objeto na cena, esse objeto é selecionado e sua representação é alterada para informar sua seleção. Com o objeto selecionado, é possível aplicar transformações a ele e, por exemplo, transladá-lo, conforme será visto no próximo item.

- **Translação de objetos (inclusive com o mouse):** no programa legado não era possível transladar os objetos, sendo que sua posição era definida somente no momento de sua criação e qualquer alteração de posição implicava em reiniciar o programa e reinserir todos os dados. Com a reestruturação e a implementação dos componentes e sub-componentes da nova arquitetura, é possível alterar os dados do objeto em tempo real, alterando sua posição na própria cena. Também é possível, na nova versão reestruturada, selecionar os objetos com o mouse (através do *picking)*, escolher um eixo na barra de ferramentas e transladá-lo com o mouse.

- **Manipulação da câmera com o mouse:** a partir da reestruturação, a câmera pode ter seus dados alterados via entrada de dados na interface gráfica, que representa o fluxo de execução, de modo semelhante ao que era feito no programa legado. Porém, a funcionalidade de manipulação da câmera foi criada para ser realizada através do mouse, simulando o comportamento de um *TrackBall* e proporcionando um ajuste mais flexível da posição da câmera.

- **Câmera auto-ajustável à cena:** para facilitar um ponto de partida para a configuração da câmera, foi criada a funcionalidade da câmera auto-ajustável, que enquadra toda a cena no campo de visão da câmera automaticamente. Para execução desta funcionalidade nova, é calculado o retângulo que abrange toda a cena e forma sua fronteira (*Axis Aliged Boundary Box[5]*) e a câmera é posicionada próximo a uma aresta deste retângulo, com seu foco apontado para o centro da caixa (e, conseqüentemente, para o centro da cena).

5.4.2. Visualizador sob a nova arquitetura

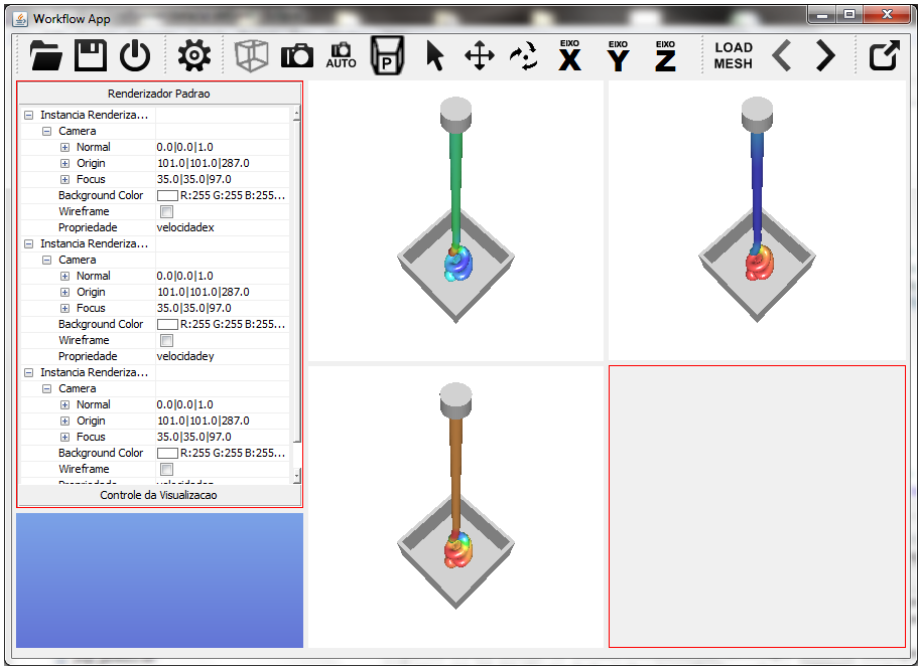


Figura 38: Exemplo de execução de um fluxo de execução para visualização – Note que existem três instâncias de *rendering* exibindo informações diferentes texturizadas no fluido, sendo elas informações sobre a velocidade no eixo x, y e z.

O visualizador reestruturado do *Freeflow3D*, tal como o modelador detalhado acima, funciona baseado em um fluxo de execução guia. O visualizador é mais simples que o modelador e isso se reflete no tamanho do fluxo de execução e em suas funcionalidades. O visualizador do *Freeflow3D* abre os arquivos gerados pela simulação numérica e, por isso, o usuário final não precisa se preocupar em adicionar os modelos na cena como acontece no modelador. O fluxo de execução padrão do visualizador, construído com a ajuda do *WorkflowDrawer*, pode ser conferido na Figura 39 (abaixo):

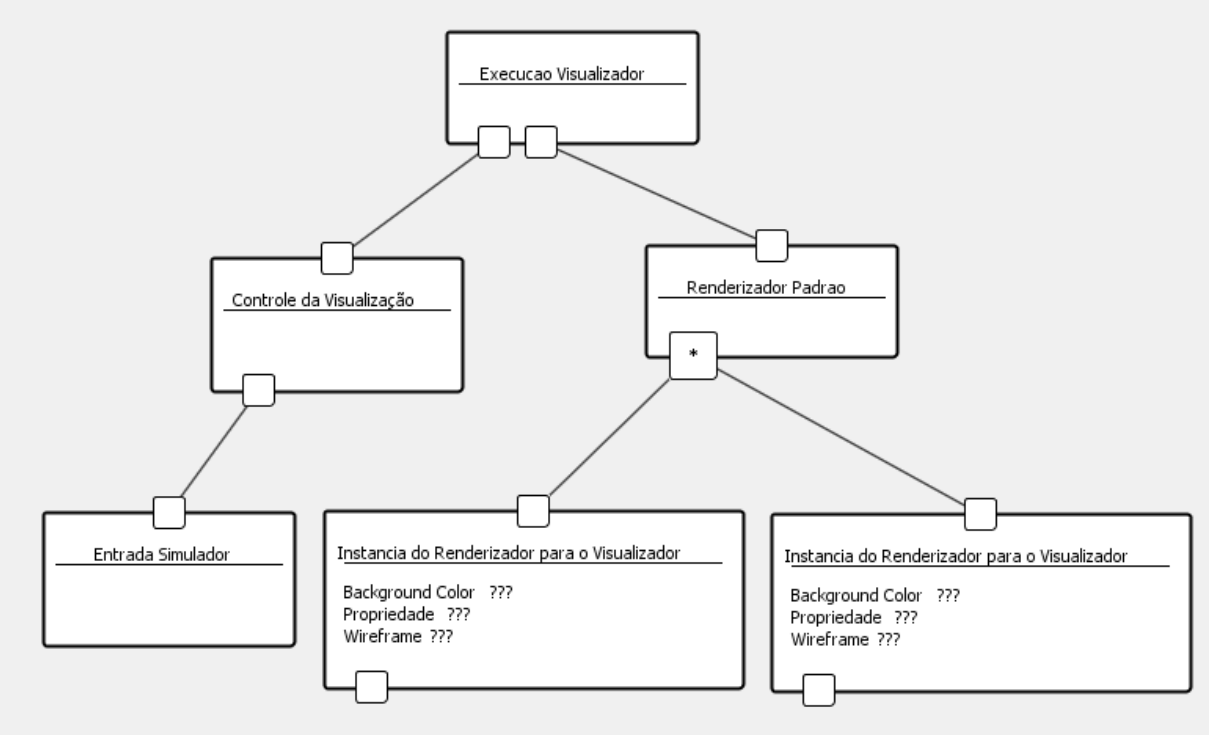


Figura 39: Representação do fluxo de execução da implementação padrão do visualizador – Desenhado pelo *WorkflowDrawer* e com múltiplas instâncias de *rendering*.

Note que na Figura 39 (acima) foram adicionadas mais de uma instância de *rendering*, isso acontece para que se tenham múltiplas visões da cena para análise dos resultados, e também para aplicar sobre a mesma cena dados de propriedades geradas na simulação (exemplos: velocidade, pressão, etc.).

O componente *IVisController* exerce um papel importante na solução da reestruturação do visualizador, mantendo o controle sobre a cena aberta e controlando o avanço e retrocesso nas etapas do resultado da simulação. O fluxo de execução descrito na Figura 39 (acima), ao ser executado pelo *WorkflowRunner*, sobre a simulação de teste KWMAI2005C gera a seguinte representação (vide Figura 40 abaixo):

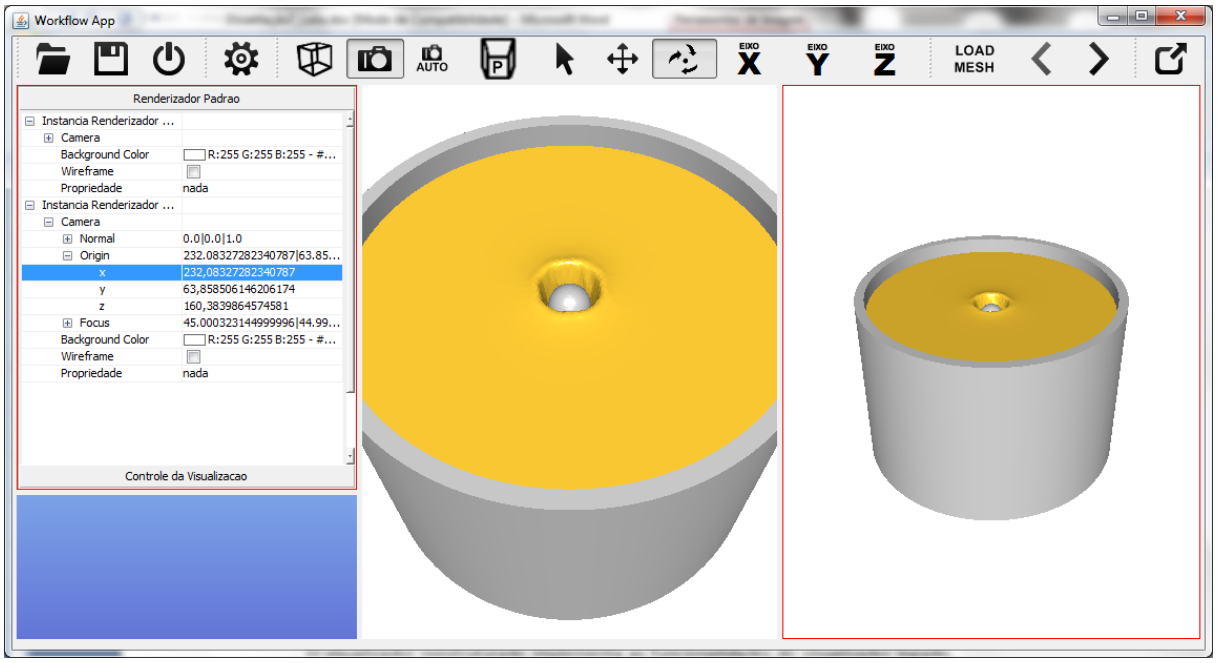


Figura 40: Representação da execução do fluxo de execução de exemplo.

O visualizador reestruturado implementa as funcionalidades do visualizador legado, como a realização do *rendering* das etapas de simulação, abertura e exibição das propriedades calculadas na simulação (velocidade, pressão, viscosidade, entre outras). O novo visualizador também conta com novas funcionalidades, sendo que a maioria já foi descrita no item anterior sobre o modelador, tais como, por exemplo: criação dinâmica de interface gráfica, movimentação da câmera através do mouse, edição de dados com resultado em tempo real, entre outras. Contudo, dentre as novas funcionalidades, é possível levantar também o fato de que foi desenvolvido um componente específico para o visualizador (*DefaultVisRendererInstanceImpl*) que representa uma instância de *rendering* modificada e possibilita a visualização de propriedades diversas em diferentes instâncias de *rendering*, exibidas simultaneamente na tela (como visto na Figura 38).

5.5. Estudo de caso: rendering modular - shaders e a vantagem do uso de componentes

Esse item tem como objetivo demonstrar como o uso de componentes traz vantagens ao programa reestruturado através de um exemplo de uso. O exemplo em questão é a implementação de um componente de *rendering* que possa substituir a implementação padrão trazendo uma nova funcionalidade: o uso de programas *shader*s, como visto no item 3.4.2 do capítulo 3.

Os programas *shaders* substituem a *pipeline* padrão das placas gráficas e podem ser programadas para gerar *renderings* com modelos de iluminação mais complexos, além de efeitos avançados de computação gráfica, como sombras, reflexões, etc. Para adicionar essa funcionalidade, foi implementado um novo componente do tipo *IRendererInstance*, que tem como interface dois campos para informar o caminho do programa *vertex* e *fragment* da linguagem GLSL.

Como a implementação criada (de nome *DefaultVisShaderRendererInstanceImpl*) é baseada no componente básico *IRendererInstance*, é possível trocar a implementação padrão (no caso do visualizador, a *DefaultVisRendererInstanceImpl*) pela nova implementação sem maiores mudanças. Essa é a grande vantagem do uso de componentes nessa arquitetura, ou seja, a troca transparente de implementações de componentes. Contanto que a implementação garanta as funcionalidades básicas e interface definidas pelo componente básico, ela pode substituir as implementações padrão sem nenhum efeito colateral ao funcionamento dos fluxos de execuções. No caso do *DefaultVisShaderRendererInstanceImpl*, ele adiciona uma nova funcionalidade, o uso de programas *shaders*. Outras implementações poderiam, por exemplo, desenvolver uma organização de cena a fim de otimizar o *rendering* e melhorar a performance, e ela não alteraria o funcionamento do programa.

Nesse estudo de caso, o usuário deseja utilizar a nova implementação da instância de *rendering*. Logo, ele altera seu fluxo de execução, trocando a implementação padrão pela implementação com o uso de *shader*, como ilustrado na Figura 41 (abaixo):

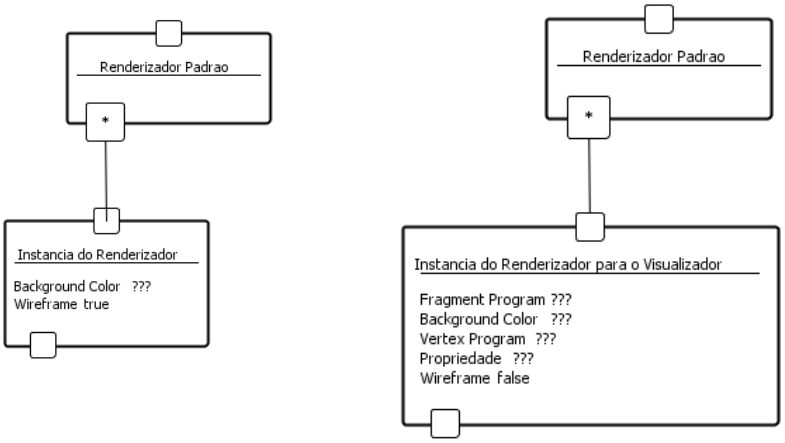


Figura 41: Representação da troca do componente – Uso da implementação com uso de *shaders* em detrimento da implementação padrão da instância do *renderer*.

O próximo passo é configurar o componente com o endereço dos *shaders* que substituirão os programas padrão da placa gráfica alterando sua função de processamento de vértices e coloração de *pixel*. Nesse estudo de caso, foram utilizados dois *shaders* para testes, um que simula uma *rendering* cartunesca (*toon shading*[34]) e outro que executa funções de reflexão e refração de um cubo de texturas (*cube mapping*[35]). Ao executar o fluxo no *WorkflowRunner*, o usuário já confere as mudanças causadas pelo uso de *shaders* (como pode ser visto na Tabela 5 (abaixo), que resume a troca de componentes e exibe os resultados das execuções dos *shaders*).

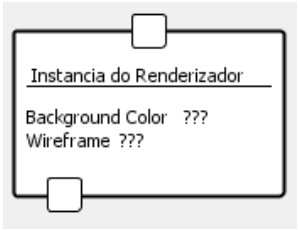
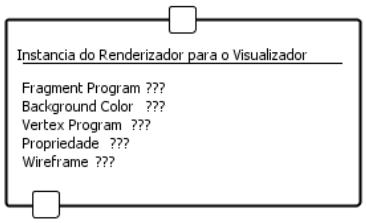
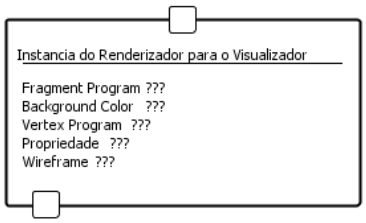
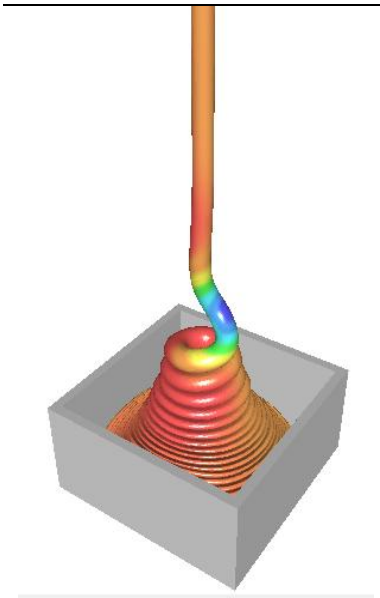
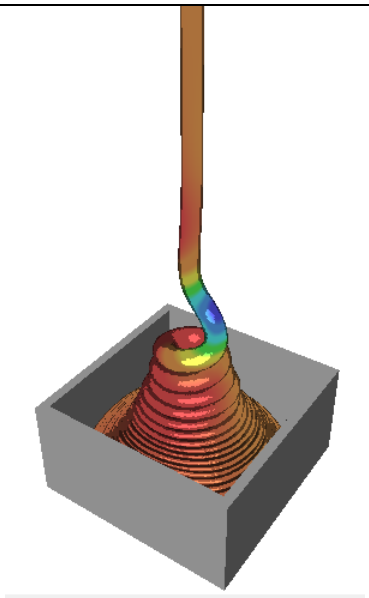
Componente DefaultRendererInstanceImpl	Componente DefaultVisShader RendererInstanceImpl	Componente DefaultVisShader RendererInstanceImpl
Representação Gráfica: 	Representação Gráfica: 	Representação Gráfica: 
Dados: Background Color: 255,255,255 Wireframe: false Propriedade: velocidade x	Dados: Background Color: 255,255,255 Wireframe: false Propriedade: velocidade x Vertex: TOONv.glsl Fragment: TOONf.glsl	Dados: Background Color: 255,255,255 Wireframe: false Propriedade: velocidade x Vertex: CUBEv.glsl Fragment: CUBEf.glsl
		

Tabela 5: Implementação padrão e com *shaders* – Diferenças entre a implementação padrão e a implementação com *shaders*.

Através deste exemplo, fica claro as vantagens do uso do componente e a flexibilidade que esse uso proporciona ao usuário final, que pode escolher novas implementações dos componentes no futuro. Não só o componente de *rendering* pode ter novas implementações, como também todos os outros componentes que compõe os fluxos

de execução. Ao usuário, fica somente a tarefa de escolher ou alterar seus fluxo de execução com o endereço da nova implementação, seja via interface gráfica desenhando no programa de auxílio *WorkflowRunner* ou alterando o arquivo XML que descreve seu fluxo de execução.

6. Resultados Obtidos

Ao longo do presente trabalho, o leitor pode conferir tudo o que foi realizado para reestruturar os programas legados do *Freeflow3D* para modelagem e visualização de simulações de escoamento de fluido. Os resultados obtidos podem ser analisados sob os aspectos quantitativo e qualitativo referentes à aplicação dos conceitos estudados no trabalho realizado.

Quanto à análise dos resultados sob a ótica quantitativa, o item de mais destaque é o grau de esforço empregado na reestruturação dos programas do *Freeflow3D*. A solução desenvolvida contém cerca de 31.000 (trinta e uma mil) linhas de código, distribuídas da seguinte forma: 7.000 (sete mil) linhas de código foram geradas automaticamente (responsáveis pela manipulação de arquivos XML); 10.000 (dez mil) linhas de código resultaram da tradução das funções de baixo nível (que formam o "Núcleo do *Freeflow3D*") da linguagem "C" para "*Java*"; e 14.000 (quatorze mil) linhas de código foram criadas na reestruturação dos programas propriamente ditos, isto é, na definição e aplicação da nova arquitetura, na criação de implementações padrão dos componentes e na criação de programas que suportem o funcionamento da nova arquitetura. Todas essas linhas de código são responsáveis não somente pela reestruturação e reescrita dos programas legados, mas também pelas novas funcionalidades desenvolvidas, são algumas delas que garantem, inclusive, uma maior eficiência no desenvolvimento e na diminuição do código necessário (através da geração automática de interface gráfica, do uso de uma estrutura de dados flexível e expansível através da descrição de arquivos XML, dentre outras funcionalidades).

Já sob os aspectos qualitativos, é possível observar importantes resultados obtidos da aplicação dos conceitos estudados durante o trabalho realizado. Iniciando pela aplicação dos conceitos de Engenharia de *Software*, vê-se que os conceitos de Engenharia de Software Baseada em Componentes (ESBC) foram essenciais para a definição da nova arquitetura do *Freeflow3D*. Essa nova arquitetura, baseada em componentes e fluxos de execução, adaptou os conceitos da ESBC para trazer diversos ganhos à solução, dentre eles: a facilidade de criar

e alterar funcionalidades, agora representadas por componentes semanticamente coesos e de fácil definição, criação e manutenção; a desnecessidade de preocupar-se com a interface gráfica nos futuros desenvolvimentos, já que ela é resolvida automaticamente; e a facilidade de testar novas execuções e alterá-las através do uso de fluxos de execução, dando flexibilidade e produtividade ao usuário final.

Ainda com relação à Engenharia de *Software*, é possível citar também os resultados obtidos com a aplicação das atividades da reengenharia de *software*. Todo o código foi reestruturado a partir da reimplementação das macro-funcionalidades dos programas legados em componentes, que agora contam com um código coeso, eficiente e baseado nas melhores técnicas de programação, obtidas principalmente com o uso da linguagem orientada a objetos (ex.: encapsulamento de dados, alta coesão e baixo acoplamento) e ao uso de padrões de projeto. O armazenamento de dados do *Freeflow3D* também foi reestruturado, se baseando principalmente em descrições na linguagem de marcação XML e provendo maior flexibilidade na entrada de dados pelo usuário final. Por fim, novas funcionalidades foram criadas evidenciando a engenharia progressiva empregada, que durante toda a reestruturação preparou o código criado para absorvê-las.

Na área de Computação Gráfica, obtiveram-se resultados principalmente na modernização do processo de *rendering*, na criação de uma estrutura de *renderer* e gerenciamento de cena e na melhora da interação com o usuário através de uma série de funcionalidades. O processo de *rendering* foi construído através do uso de *Opengl* e, conseqüentemente, acelerado nas placas gráficas, o que garante um melhor desempenho, desempenho esse que pode ser escalado com o uso de placas gráficas mais poderosas. O uso de *Opengl* também garante a possibilidade do uso de novas funcionalidades e efeitos gráficos, dentre eles o uso de materiais e *shaders* programáveis.

As estruturas criadas para representação do *renderer* e do gerenciamento de cena garantem o funcionamento de novas funcionalidades e preparam a solução para receber implementações mais sofisticadas, tais como, por exemplo, os gerenciadores de cena com algoritmos de otimização de *rendering,* através de algoritmos de posicionamento de cena (ex.: *frustrum culling*, etc.). Com essa nova estrutura de *renderer*, foi possível trazer aos novos programas diversas funcionalidades ausentes no programa legado, como a seleção de objetos através do mouse (*picking*), a manipulação de objetos e câmera através do mouse

(inclusive simulando o funcionamento de uma *TrackBall*), dentre outras funcionalidades que foram descritas e detalhadas ao longo do trabalho e que garantem uma experiência ao usuário final mais rica e eficiente.

Em termos de programas desenvolvidos, foram criados dois programas, o *WorkflowDrawer* e *WorkflowRunner*, que juntos garantem o funcionamento do *Freeflow3D* sob a nova arquitetura. Eles auxiliam na criação e executam fluxos de execuções para resolver problemas, tanto de modelagem quanto de visualização no âmbito do *Freeflow3D*. Através dos fluxos de execução customizados pelo usuário final e, conseqüentemente, das implementações de componentes escolhidas, o programa *WorkflowRunner* cria execuções do visualizador e do modelador, o que pode ser conferido nas Figuras 42 e 43 (abaixo).

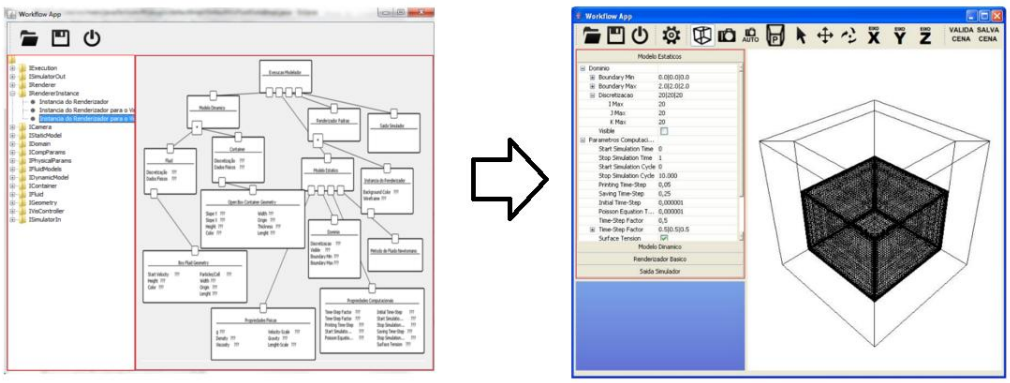


Figura 42: Fluxo de execução desenhado no *WorkflowDrawer* gera a execução do modelador no *WorkflowRunner*.

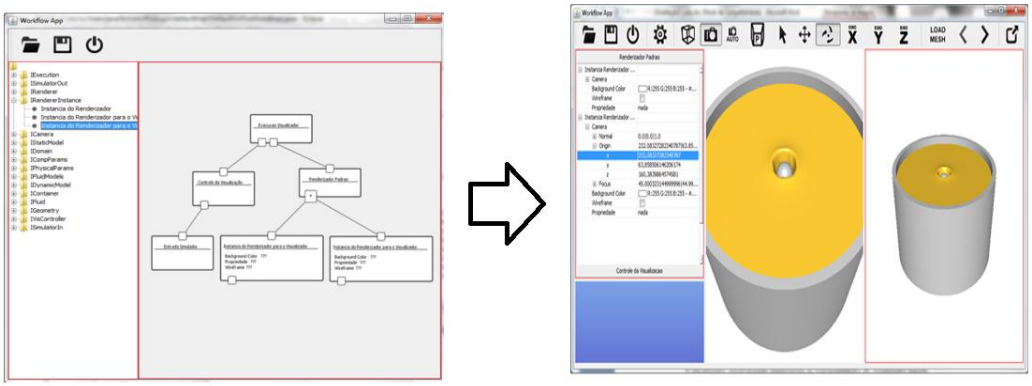


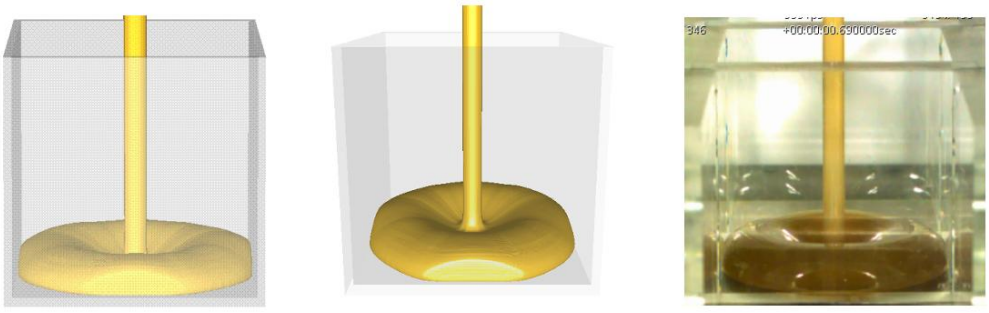
Figura 43: Fluxo de execução desenhado no *WorkflowDrawer* gera a execução do visualizador no *WorkflowRunner*.

Além de todos esses resultados descritos acima, obtidos no desenvolvimento da reestruturação, na aplicação dos conceitos de computação e na criação de programas, tem-se o que se pode dizer serem os resultados mais importantes do trabalho: as melhorias das limitações levantadas da análise do programa legado. Os pontos de melhorias serão recapitulados a seguir e finalizam a análise de resultados obtidos em curso.

Quanto ao código fonte não portável, o uso da linguagem *Java* e *Opengl* proporcionaram a possibilidade dos programas criados durante o presente trabalho serem executados inicialmente em *Linux*, *Windows* e *Mac*.

Vê-se que a dificuldade de manutenção e evolução do código legado foi atacada através da nova arquitetura proposta pelo presente trabalho. O uso de componentes representando cada funcionalidade do programa favorecem a sua manutenção, já que o código da funcionalidade está encapsulado no componente de forma coesa. Ademais, novas funcionalidades podem ser implementadas em novos componentes que têm suas execuções garantidas pela nova arquitetura, componentes estes que são intercambiáveis dentro dos fluxos de execução.

O *renderer* que outrora era executado via *software,* agora é acelerado em *hardware*, produz imagens fidedignas em 16 milhões de cores, em contrapartida às antigas imagens em 256 cores que simulavam 16 milhões de cores através da técnica de *halftone*[3]. Além disso, por ter seu processo de *rendering* acelerado em *hardware* com execução altamente paralelisada, obtiveram-se ganhos de performance que tornaram as visualizações interativas e viabilizaram funcionalidades como a manipulação de objetos e câmera através do mouse. Veja na Figura 44 (abaixo) a comparação entre o resultado do visualizador legado, a nova versão e os resultados reais de escoamento de fluido.



Legado Nova versão Experimento real

Figura 44: Comparação entre resultados – Comparação entre o resultado do visualizador legado, a nova versão e os resultados reais de escoamento de fluido.

Por fim, referente à interface gráfica dos novos programas, esta ganhou diversas melhorias. A entrada de parâmetros não possui mais a limitação de só poder ser alterada após o usuário pressionar *enter* em cada campo. A partir da reestruturação realizada neste trabalho, o uso, em conjunto, dos componentes e da interface gráfica gerada automaticamente, oferece ao usuário telas muito mais coesas e organizadas, favorecendo o bom uso do programa (veja na Figura 44, que evidencia a diferença entre as versões). As supracitadas funcionalidades de interação com a cena através do mouse também favorecem a interface do usuário com os novos programas.

Como visto, todos os resultados esperados foram obtidos com sucesso ao longo do presente trabalho. As limitações dos programas do *Freeflow3D* foram supridas e, nesse processo, foi criado algo talvez mais importante que as funcionalidades reimplementadas ou criadas, que é a nova arquitetura por trás dos programas reestruturados. Essa nova arquitetura, fortemente baseada nos conceitos de engenharia de software, traz um novo paradigma tanto para o desenvolvimento quanto para o uso dos programas do *Freeflow3D*. A nova arquitetura marca o futuro dos programas do *Freeflow3D*, pois viabiliza a evolução do mesmo por facilitar e automatizar o seu desenvolvimento, como também por privilegiar o uso efetivo dos programas pelos usuários finais do *Freeflow3D*.

7. Conclusão

O objetivo inicial do trabalho era a reestruturação do visualizador e modelador do *Freeflow3D*, corrigindo suas principais deficiências e deixando-o portável. Contudo, pode-se concluir que em certo ponto o resultado final evoluiu um passo além do objetivo inicial.

Uma nova arquitetura foi proposta para auxiliar tanto o futuro desenvolvimento dos programas do *Freeflow3D,* como também o seu uso pelo usuário final. Na nova arquitetura, várias tarefas foram automatizadas para que o trabalho necessário, tanto do desenvolvedor quanto do usuário final, fossem reduzidos, viabilizando o uso e a evolução dos programas.

Diversas técnicas de Engenharia de *Software* foram utilizadas para garantir um código melhor, coeso, eficiente e de fácil entendimento e manutenção. A tradução dos programas em componentes favorece essas características e o baixo acoplamento entre os vários módulos dos programas. Já o conceito de fluxo de execução flexibiliza o uso do programa pelo usuário final, que passa a ter a flexibilidade de montar, alterar e reusar fluxos para resolver seus problemas com a utilização dos componentes gerados.

A modernização dos programas almejada como objetivo principal deste trabalho foi atingida. Através do presente trabalho os programas ganharam novas funcionalidades, trazendo uma nova experiência para o usuário, e tornando o uso dos programas mais ágil e mais efetivo. O uso do mouse para manipulação de cena, o uso e reuso dos fluxos de execução, a definição e alteração de parâmetros em tempo de execução, a disponibilidade dos dados de entrada de agrupadas de forma coesa, são os principais exemplos da reengenharia progressiva aplicada nos programas do *Freeflow3D*.

Ainda no que se refere à modernização dos programas, se destacam nos programas reestruturados o uso de uma linguagem orientada a objetos (em contraponto à linguagem estruturada C), o uso da placa gráfica para o *rendering* 3D e a portabilidade alcançada, já que a partir da reestruturação o *Freeflow3D* pode ser executado em *Linux*, *Windows* e *Mac*. Além dos ganhos de produtividade e facilidade de desenvolvimento trazidos pela linguagem orientada a objetos *Java*, o uso do *Opengl* deixa o programa preparado para *rendering* de cenas cada vez mais complexas e com efeitos sofisticados (garantido pela aceleração em

hardware promovida pelas placas gráficas) e a portabilidade, por sua vez, garante que uma gama maior de usuários venha a utilizar o ambiente de *softwares* do *Freeflow3D*.

Não obstante o objetivo inicial ter sido atingido através do presente trabalho, o desenvolvimento dos novos programas continuará após o término deste. Todas as funcionalidades e modelos do programa legado devem ser portados para a nova arquitetura (já que alguns modelos e funcionalidades adicionais não foram portadas por não estarem abrangidas no escopo inicial do trabalho). Ademais, como todo programa, para seu aprimoramento o *Freeflow3D* precisa receber todo o *feedback* do uso dos programas pelos usuários potenciais, e a análise disto resultará em mudanças e melhorias contínuas nos programas reestruturados, fazendo com que eles, de fato, substituam os programas legados.

REFERÊNCIAS

[1] CASTELO, A. F., at al, Freeflow: An Intregated Simulation System for Three-Dimensional Free Surface Flows, Comput. Visualization Sci., 2, pp. 199–210, 2000.

[2] SEGAL, M. and AKELEY, K., The Opengl Graphics System: A Specification (Version 2.0), Silicon Graphics, Inc., October 22, 2004.

[3] GOMES, J. and VELHO, L., Fundamentos da Computação Gráfica, Associação Instituto Nacional de Matemática Pura e Aplicada, Rio de Janeiro, 2003.

[4] WATT, A., 3D Computer Graphics, Third Edition, Addison-Wesley, 2000.

[5] EBERLY, D., 3D Game Engine Architeture: Engineering Real-Time Applications with Wild Magic, USA, Morgan Kaufmann Publishers, 2005

[6] AKENINE-MÖLLER, T. and HAINES, E., Real Time Rendering, Second Edition, A K Peters, 2002.

[7] WOO. M, et al, Opengl Programming Guide – Opengl Architecture Review Board, Third Edition, 5th print, Feb 2000, Addison Wesley.

[8] ROST, R., Opengl Shading Language, First Edition, Pearson Education, 2004.

[9] MARK., W., Cg: A System for Programming Graphics Hardware in a C-like Language, 2001.

[10] VERTEX AND FRAGMENT PROCESSOR - Disponível em: <http://www.tomshardware.com/reviews/Opengl-2,427-6.html>. Acesso em 02 de abril de 2009.

[11] EBERLY, D. Game Physics, First Edition, USA, Morgan Kaufmann Publishers, 2003.

[12] IEEE Standards Collection: Software Engineering, IEEE Standard 610.12, 1990, IEEE, 1993.

[13] JACOBSON, I. at al, Reengineering of old systems to an object-oriented architecture. New York, USA, 1991.

[14] PRESSMAN, R., Software Engineering: A Practitioner's Approach, 6th Edition, 2005, McGraw-Hill.

[15] SNEED, H., Planning the reengineering of legacy systems, IEEE Software, volume 12, 1995.

[16] ALEXANDER, C, et al, Pattern Language. Oxford University Press, New York, 1977.

[17] GAMMA, E., Design Patterns, 1995, Addison-Wesley.

[18] XML Extensible Markup Language – Disponível em: <http://www.w3.org/XML/>. Acesso em: 28 de outubro de 2008.

[19] STOJANOVIĆ, Z. A Method for Component-Based and Service-Oriented Software Systems Engineering, Doctoral Thesis, Delft University of Technology, Netherlands, 2005.

[20] WILLIANS S. and KINDEL, C. The Component Object Model: A Technical Overview. Microsoft Corporation White Paper, 1994.

[21] Gartner Group IT Definitions and Glossary – Disponível em: <http://www.gartner.com/technology/research/it-glossary/>. Acesso em 18 de novembro de 2009.

[22] BROWN, A. W. and SHORT, K., On Components and Objects: The Foundations of Component-Based Development, 5th International Symposium on Assessment of Software Tools (SAST '97), pp.0112, 1997

[23] Flow Based Programming – Disponível em: <http://www.jpaulmorrison.com/fbp/>. Acesso em 18 de novembro de 2009.

[24] IEEE, and ISO/IEC, Systems and Software Engineering - Recommended Practice for Architectural Description of Software-Intensive Systems, ISO/IEC 42010 IEEE Std 1471-2000, c1–24, 2007.

[25] Java Swing - Disponível em: <http://download.oracle.com/javase/tutorial/uiswing/start/about.html>. Acesso em: 14 de fevereiro de 2011.

[26] Java SwingX – Disponível em: <http://swinglabs.org>. Acesso em 14 de fevereiro de 2011.

[27] L2FProd – Disponível em: <http://www.l2fprod.com/>. Acesso em 14 de fevereiro de 2011.

[28] Log4j – Disponível em: <http://logging.apache.org/log4j/>. Acesso em 14 de fevereiro de 2011.

[29] Spring – Disponível em: <http://www.springsource.org/>. Acesso em 14 de fevereiro de 2011.

[30] Castor – Disponível em: <www.castor.org>. Acesso em 14 de fevereiro de 2011.

[31] JOGL – Disponível em: <jogamp.org/jogl/www/>. Acesso em 14 de fevereiro de 2011.

[32] Maven – Disponível em: <maven.apache.org>. Acesso em 14 de fevereiro de 2011.

[33] JNI (Java Native Interface) – Disponível em: <http://download.oracle.com/javase/6/docs/technotes/guides/jni/index.html>. Acesso em 14 de fevereiro de 2011.

[34] Toon Shading em GLSL – Disponível em: <http://www.lighthouse3d.com/tutorials/glsl-tutorial/toon-shading-version-iii/ >. Acesso em 14 de fevereiro de 2011.

[35] Cube Mapping em OGL – Disponível em: <http://developer.nvidia.com/content/cube-map-ogl-tutorial>. Acesso em 14 de fevereiro de 2011.