UNIVERSIDADE FEDERAL DE JUIZ DE FORA
INSTITUTO DE CIÊNCIAS EXATAS
PROGRAMA DE PÓS-GRADUAÇÃO EM MODELAGEM COMPUTACIONAL

Igor Pires dos Santos

Construção de uma ferramenta computacional para simulação hemodinâmica em modelos 1D de árvores arteriais

Juiz de Fora

2022

Igor Pires dos Santos

Construção de uma ferramenta computacional para simulação hemodinâmica em modelos 1D de árvores arteriais

Dissertação apresentada ao Programa de Pós-graduação em Modelagem Computacional da Universidade Federal de Juiz de Fora como requisito parcial à obtenção do título de Mestre em Modelagem Computacional. Área de concentração: Modelagem Computacional

Orientador: Prof. Dr. Rafael Alves Queiroz de Bonfim

Coorientador: Prof. Dr. Ruy Freitas Reis

Juiz de Fora

2022

Ficha catalográfica elaborada através do Modelo Latex do CDC da UFJF
com os dados fornecidos pelo(a) autor(a)

Santos,Igor Pires.
Construção de uma ferramenta computacional para simulação hemodinâmica em modelos 1D de árvores arteriais / Igor Pires dos Santos. – 2022.
126 f. : il.

Orientador: Rafael Alves Queiroz de Bonfim
Coorientador: Ruy Freitas Reis
Dissertação – Universidade Federal de Juiz de Fora, Instituto de Ciências Exatas. Programa de Pós-graduação em Modelagem Computacional, 2022.

1. Árvores Arteriais. 2. Escoamento Pulsátil. 3. Hemodinâmica Computacional. 4. Fábricas Abstratas. 5. Orientação à Objetos. I. Queiroz, Rafael Alves Bonfim de, orient. II. Reis, Ruy Freitas, coorient. III. Título.

Igor Pires dos Santos

Construção de uma ferramenta computacional para simulação hemodinâmica em modelos 1D de árvores arteriais

Dissertação apresentada ao Programa de Pós-Graduação em Modelagem Computacional da Universidade Federal de Juiz de Fora como requisito parcial à obtenção do título de Mestre em Modelagem Computacional. Área de concentração: Modelagem Computacional.

Aprovada em 23 de setembro de 2022.

BANCA EXAMINADORA

Prof. Dr. Rafael Alves Bonfim de Queiroz - Orientador

Universidade Federal de Ouro Preto

Prof. Dr. Ruy Freitas Reis - Coorientador

Universidade Federal de Juiz de Fora

Prof. Dr. João Rafael Alves

Instituto Federal do Mato Grosso

Profa. Dra. Barbara de Melo Quintela

Universidade Federal de Juiz de Fora

Juiz de Fora, 13/09/2022.

Documento assinado eletronicamente por **João Rafael Alves**, **Usuário Externo**, em 23/09/2022, às 16:12, conforme horário oficial de Brasília, com fundamento no § 3º do art. 4º do Decreto nº 10.543, de 13 de novembro de 2020.

Documento assinado eletronicamente por **Ruy Freitas Reis**, **Professor(a)**, em 23/09/2022, às 16:16, conforme horário oficial de Brasília, com fundamento no § 3º do art. 4º do Decreto nº 10.543, de 13 de novembro de 2020.

Documento assinado eletronicamente por **Barbara de Melo Quintela**, **Professor(a)**, em 23/09/2022, às 16:21, conforme horário oficial de Brasília, com fundamento no § 3º do art. 4º do Decreto nº 10.543, de 13 de novembro de 2020.

Documento assinado eletronicamente por **Rafael Alves Bonfim de Queiroz**, **Usuário Externo**, em 26/09/2022, às 15:28, conforme horário oficial de Brasília, com fundamento no § 3º do art. 4º do Decreto nº 10.543, de 13 de novembro de 2020.

A autenticidade deste documento pode ser conferida no Portal do SEI-Ufjf (www2.ufjf.br/SEI) através do ícone Conferência de Documentos, informando o código verificador **0946832** e o código CRC **8AE5BE26**.

Dedico este trabalho primeiramente a Deus, que me possibilitou a vida e consequências dela. A minha família e amigos que nunca deixaram de me amparar nesse processo evolutivo, minha namorada, e a todos que me acompanharam durante essa caminhada.

AGRADECIMENTOS

Agradeço à minha família, pelo encorajamento e apoio, e principalmente aos meus pais, por serem inspiração para os meus estudos.

À minha namorada Luana, pessoa com quem amo compartilhar a vida. Obrigado pelo suporte, carinho, paciência e companhia.

À Taiga e o Matheus, pessoas com quem dividi o meu dia-a-dia acadêmico, muito obrigado pelo suporte e a paciência.

À Adriana Ferreira, que com sua serenidade e magnificência ajudou a formar a pessoa que sou hoje. Infelizmente não foi possível agradecer pessoalmente, então deixo aqui registrado que sem você com certeza eu não estaria aqui hoje, obrigado.

Ao professor Rafael Alves Bonfim de Queiroz, pela excelente orientação, apoio e paciência, sem a qual este trabalho não se realizaria. Ao professor Ruy Freitas Reis, pela também excelente orientação, sua perspectiva possibilitou a melhora deste trabalho em diversos aspectos.

Aos professores dos Departamentos de Ciência da Computação e de Mecânica Aplicada e Computacional pelos seus ensinamentos e aos funcionários do curso, que durante esses anos, contribuíram de algum modo para o nosso enriquecimento pessoal e profissional.

Aos órgãos de fomento CAPES e CNPq que através de seus incentivos permitiram a realização deste trabalho. Em tempos como estes agradeço especialmente por ter tido esta oportunidade e chegar aonde estou, que estas instituições possam preservadas e a ciência prevaleça!

"A vida é curta demais para aprendermos somente com os nossos erros."

(Sebastião Roberto Pereira Ramon)

RESUMO

O escoamento sanguíneo em modelos de árvores arteriais do sistema circulatório é frequentemente utilizado para melhor entender este complexo sistema. Ao realizar estas simulações hemodinâmicas pode ser observado como as ondas de pressão e fluxo são alteradas ao longo do modelo vascular. Desta forma, é possível obter dados sem a realização de exames invasivos levando em conta diferentes cenários hemodinâmicos. Neste trabalho, apresentam-se: (i) um modelo matemático da literatura que descreve o escoamento sanguíneo pulsátil em modelos de árvores arteriais unidimensionais (1D), (ii) uma ferramenta computacional desenvolvida que calcula a pressão e fluxo em cada vaso a partir do modelo matemático . A ferramenta computacional desenvolvida conta com uma estrutura de dados proposta que concilia os conceitos de uma linguagem orientada à objetos e bibliotecas adicionais com o modelo matemático adotado. A estrutura de dados foi utilizada no desenvolvimento de dois ambientes, um console (modo terminal) e uma interface gráfica de usuário. A arquitetura de classes tem como objetivo conciliar as ferramentas disponibilizadas pelas bibliotecas para simular corretamente o modelo matemático escolhido e permitir que outros modelos matemáticos sejam facilmente acoplados posteriormente. A estrutura de dados proposta possibilita que as diversas características de uma árvore arterial possam ser armazenadas, utilizadas e visualizadas em tempo de execução. Os resultados obtidos neste trabalho estão condizentes com dados numéricos relatados na literatura. A ferramenta computacional construída é útil para realizar simulação hemodinâmicas em modelos 1D de árvores arteriais e pode servir como uma plataforma para investigação de modelos computacionais

ABSTRACT

The blood flow in arterial tree models of the circulatory system is frequently studied to better understand this complex system. The hemodynamic simulation shows how the wave changes as it progresses through the geometric model. It is possible to extract analytical data without invasive procedures taking different flow characteristics into account. In this work, the following are presented: (i) an analytical scheme based on physical and mathematical laws to calculate the local characteristics of the pressure and flux wave in one-dimensional (1D) arterial tree's models, (ii) a computational environment developed to simulate and visualize the results of the model's construction and hemodynamic studies. The computational environment is composed of a data structure that integrates the concepts of an object-oriented language and other libraries with the adopted analytical scheme. The proposed data structure allows the computational environment to store, utilize and display different characteristics of an arterial tree. The data structure constructed two environments: (1) a console(terminal mode), and (2) a user interface. The class architecture objectives are to ensure the correct use of the libraries to simulate the mathematical model and to allow other mathematical models to be easily integrated posteriorly. The results produced in this work are consistent with real morphometric data and numeric data related in the literature. The computational tool that was built simulates hemodynamic characteristics in 1D arterial tree models and functions as a platform to investigate computational models.

LISTA DE ILUSTRAÇÕES

Figura 1 – Segmento do modelo geométrico de uma árvore arterial, composto pelo nó proximal A e pelo nó distal B. 28

Figura 2 – Notação usada para identificar cada segmento de vaso (k, j) (figura adaptada de [12]). 30

Figura 3 – Interface gráfica da ferramenta construída. 35

Figura 4 – Representação de classes de um elemento. *WiseElement* é uma classe abstrata base e seus componentes são: *WiseStructure* representa a estrutura contida em um arquivo VTK e DataStructure representa a estrutura de ponteiros e variáveis utilizadas na iteração. 42

Figura 5 – Tipos de elementos. *WiseGraphic*, um gráfico bidimensional. *WiseMesh*, uma malha tridimensional. *WisePoly*, um cubo. *WiseArteryTree*, uma árvore arterial. 43

Figura 6 – Pontos, linhas e células para especificação do modelo geométrico. . . . 44

Figura 7 – Máquina de estados que controla o funcionamento de um elemento inteligente. 45

Figura 8 – Elemento no estado *Warming*, *Raw* e *Cooling*. 45

Figura 9 – Elemento no estado *Cold*. 46

Figura 10 – Elemento no estado *Hot*. 47

Figura 11 – Arquitetura de classes fábrica e fluxo de trabalho do elemento *WiseElement*. A fábrica *WiseElementFactory* é responsável por criar o elementos, a fábrica *WiseIterationFactory* é responsável pela iteração do elemento e a fábrica *GraphicFactory* é responsável por criar as estruturas de visualização. . 48

Figura 12 – Objeto *WiseObject* e todos seu componentes: *WiseObjectFactory*, fábrica responsável pela criação de objetos; *WiseElementFactory*, fábrica responsável pela criação de elementos; *WiseIterationFactory*, fábrica de iteração; *WiseGraphicFactory*, fábrica gráfica; *WiseCollection*, coleção de elementos; *GraphicModel* coleção de objetos gráficos. 50

Figura 13 – Máquina de estados utilizada pelos objetos *WiseObject*. O estado *Ready* indica que o objeto foi criado corretamente e o estado *Set* indica que o objeto teve suas fábricas corretamente adicionadas. Enquanto o objeto estiver iterando ele permanecerá no estado *Go* e quando finalizar irá para o estado *Finished*. O estado *Crashed* é utilizado quando o objeto não funciona corretamente. 51

Figura 14 – Modelo gráfico *GraphicModel* contém uma coleção de objetos gráficos. . 52

Figura 15 – Tipos de objetos gráficos *GraphicObjects*, representando cada tipo de elemento *WiseElement*. O objeto *Graphic* representa um gráfico, o objeto *Mesh* uma malha, o objeto *Poly* um cubo e o objeto *ArteryTree* uma árvore arterial. 53

Figura 16 – Tipos de elementos gráficos *GraphicElements*. *Point*, um ponto. *Line*, uma linha. *Quad*, um quadrado. *Cube*, um cubo. *Cylinder*, um cilindro. *Sphere*, uma esfera. 53
Figura 17 – Projeto *WiseProject* e seus componentes, uma lista de objetos *WiseObject* e uma lista de elementos *WiseElement*. 54
Figura 18 – Fábrica de projeto *WiseProjectFactory* e seus componentes: fábricas de elementos *WiseElementFactories*, fábricas de objetos *WiseObjectFactories*, fábrica de objetos gráficos *Graphic Object Factories*, fábricas de elementos gráficos *GraphicElementFactories* e fábricas de iteração *WiseIterationFactories* . 55
Figura 19 – Todas as fábricas que compõem uma fábrica de projeto. 56
Figura 20 – Modelo de *Threads*. *WiseThreadPool*, responsável por orquestrar o funcionamento das demais *threads*, bem como os objetos contidos em um projeto *WiseProject*. *WiseIO*, *thread* responsável por processos de leitura e escrita. *WiseConsole*, *thread* responsável por interpretar os comandos de texto e traduzir-los na chamada de métodos. *WiseProcessor*, *thread* responsável por realizar o método iterativo de um objeto *WiseObject* 58
Figura 21 – Modelo de *Threads* ao receber uma linha de comando da interface de usuário. 59
Figura 22 – Modelo de Threads ao receber um comando de escrita/leitura. 60
Figura 23 – Máquina de estados utilizadas por trabalhos *WiseJobs*. 62
Figura 24 – Ilustração da interface gráfica da ferramenta computacional. 63
Figura 25 – Estrutura do projeto que compõe o ambiente computacional *InGU*. . . 64
Figura 26 – Captura de tela com a execução do ambiente computacional *InGU* em um console. 64
Figura 27 – Captura de tela com a execução do comando de ajudo no ambiente computacional *InGU* em um console. 66
Figura 28 – Fluxo de execução do ambiente computacional InGU, em azul as atividades de criação de elementos, em verde as atividades de criação de projetos e em laranja as atividades de criação de objetos. 86
Figura 29 – Fluxo de execução do ambiente computacional IGU. 87
Figura 30 – Janela principal ambiente computacional IGU. Da esquerda para a direita: (A) Menu principal do programa; (B) Árvore de projetos e seus elementos; (C) Área de trabalho, no caso mostrando OpenGl *Canvas*; (D) Seleção de abas 88
Figura 31 – Opções do menu principal da ferramenta computacional *IGU* 89
Figura 32 – Parte da janela principal contendo a área de trabalho e as abas que selecionam a interface selecionada, sendo os componentes: (C) Área de trabalho *Canvas*; (D) Seleção de abas 90

Figura 33 – Área de trabalho exibindo o elemento gráfico *Canvas* com a aba *Canvas* selecionada. 90
Figura 34 – Área de trabalho exibindo o elemento gráfico *Console* com a aba *Console* selecionada. 91
Figura 35 – Árvore de projetos, na imagem a ferramenta apresenta um projeto $p1$, um objeto $obj1$ e seus elementos gráficos e elementos. 92
Figura 36 – Ilustra a janela *Canvas* exibida sobre a área de trabalho *Canvas*, a janela exibindo um gráfico obtido como resultado e a área de trabalho exibindo a árvore arterial estudada. 93
Figura 37 – Janela *Jobs* exibida com a aba *Timetable* selecionada, a janela lista os trabalhos recebidos pela estrutura *WiseThreadPool* e seus tempos de execução. . 94
Figura 38 – Janela *Jobs* exibida com a aba *Timetable* selecionada, a janela lista os trabalhos recebidos pela estrutura *WiseThreadPool* e os separa logicamente pelas listas de espera. 95
Figura 39 – Representação do modelo de árvore arterial canina (figura adaptada de [12]). 97
Figura 40 – Amplitude da pressão $|P|$ ao longo da árvore arterial considerando diferentes viscosidade do fluido μ e frequências: (a) f = 3,65 Hz, (b) f = 7,30 Hz. 99
Figura 41 – Amplitude da pressão $|P|$ ao longo da árvore arterial considerando diferentes viscosidade do fluido μ e frequências: (a) f = 10,95 Hz, (b) f = 14,60 Hz. 100
Figura 42 – Amplitude da pressão $|P|$ ao longo da árvore arterial considerando diferentes valores de viscoelasticidade ϕ_0 e frequências: f = 3,65 Hz e f = 7,30 Hz. 101
Figura 43 – Amplitude da pressão $|P|$ ao longo da árvore arterial considerando diferentes valores de viscoelasticidade ϕ_0 e frequências: f = 10,95 Hz e f = 14,60 Hz. 102
Figura 44 – Amplitude da pressão $|P|$ ao longo da árvore arterial considerando diferentes valores de viscoelasticidade ϕ_0 e frequências: (a) f = 3,65 Hz, (b) f = 7,30 Hz. 104
Figura 45 – Amplitude da pressão $|P|$ ao longo da árvore arterial considerando diferentes valores de viscoelasticidade ϕ_0 e frequências: (a) f = 10,95 Hz, (b) f = 14,60 Hz. 105
Figura 46 – Representação do projeto aberto através do arquivo *CMakeLists.txt* na interface *IDE* do *QtCreator*. 106
Figura 47 – Ferramenta computacional *IGU0*. 106

LISTA DE TABELAS

Tabela 1 – Propriedades de cada vaso arterial do modelo. 37
Tabela 2 – Descrição do comando para ajuda. 65
Tabela 3 – Descrição do comando para ler arquivo de comando. 66
Tabela 4 – Descrição do comando para bateria de testes. 67
Tabela 5 – Descrição do comando para listar testes. 67
Tabela 6 – Descrição do comando para executar caso de teste. 68
Tabela 7 – Descrição do comando para testar a igualdade de dois arquivos. 68
Tabela 8 – Descrição do comando para criar projetos. 69
Tabela 9 – Descrição do comando para selecionar projetos. 69
Tabela 10 – Descrição do comando para listar projetos. 69
Tabela 11 – Descrição do comando imprimir projetos. 70
Tabela 12 – Descrição do comando para excluir projetos. 70
Tabela 13 – Descrição do comando para salvar projetos. 70
Tabela 14 – Descrição do comando para carregar projetos. 71
Tabela 15 – Descrição do comando para criar. 71
Tabela 16 – Descrição do comando para clonar elementos. 72
Tabela 17 – Descrição do comando para listar elementos. 72
Tabela 18 – Descrição do comando para imprimir elementos. 72
Tabela 19 – Descrição do comando para excluir elementos. 72
Tabela 20 – Descrição do comando para salvar elementos. 73
Tabela 21 – Descrição do comando para carregar elementos. 73
Tabela 22 – Descrição do comando para listar exportações de um elemento. 74
Tabela 23 – Descrição do comando para exportar elementos. 74
Tabela 24 – Descrição do comando para escalar elementos. 75
Tabela 25 – Descrição do comando para definir parâmetros de elementos. 75
Tabela 26 – Descrição do comando para definir todos os parâmetros de um campo pertencente a um elemento. 76
Tabela 27 – Descrição do comando para listar fábricas de elemento. 77
Tabela 28 – Descrição do comando para listar exemplos contidos em determinada fábrica de elemento. 77
Tabela 29 – Descrição do comando para criar objetos. 78
Tabela 30 – Descrição do comando para clonar objetos. 78
Tabela 31 – Descrição do comando para listar elementos. 79
Tabela 32 – Descrição do comando para imprimir objetos. 79
Tabela 33 – Descrição do comando para excluir objetos inteligentes. 79
Tabela 34 – Descrição do comando para salvar objetos. 80
Tabela 35 – Descrição do comando para carregar objetos. 80

Tabela 36 – Descrição dos comandos de exportação elementos contidos no *Forno* de objetos. 80
Tabela 37 – Descrição do comando para definir de parâmetros do objeto. 81
Tabela 38 – Descrição do comando para definir objeto. 81
Tabela 39 – Descrição do comando para iterar objetos. 82
Tabela 40 – Descrição do comando para listar fábricas de iteração. 82
Tabela 41 – Descrição do comando definir fábricas gráficas. 83
Tabela 42 – Descrição do comando para listar fábricas gráficas de objetos. 83
Tabela 43 – Descrição do comando para definir fábricas gráficas de objeto. 83
Tabela 44 – Descrição do comando para listar objetos gráficos. 84
Tabela 45 – Descrição do comando para listar elementos gráficos *Canvas*. 84
Tabela 46 – Descrição dos comandos que enviam um objeto gráfico para ser exibido em um elemento gráfico da interface de usuário. 84
Tabela 47 – Descrição do comando terminar link gráfico. 85
Tabela 48 – Propriedades dos vasos do modelo de árvore arterial [10, 12] 97
Tabela 49 – Parâmetros de entrada utilizados no teste de carga. 107
Tabela 50 – Tempo de execução média em milissegundos (ms) do ciclo de iteração com diferentes arranjos de *threads*, em negrito os melhores tempos. 108
Tabela 51 – *Speed up* do ciclo de iteração com diferentes arranjos de *threads*, em negrito os maiores aumentos de desempenho. 108
Tabela 52 – *Speed up* do ciclo de iteração com diferentes arranjos de cargas de trabalho w, em negrito os melhores aumentos de desempenho. 109

LISTA DE ACRÔNIMOS

PPGMC	Programa de Pós-Graduação em Modelagem Computacional
UFJF	Universidade Federal de Juiz de Fora
XML	Xtreme Markup Language
IGU	Iterador Gráfico Universal
InGU	Iterador não-Gráfico Universal
VTK	Visualization Tool**k**it

SUMÁRIO

1 INTRODUÇÃO . . . 23
1.1 OBJETIVOS . . . 24
1.2 ORGANIZAÇÃO . . . 25
2 MODELAGEM DO ESCOAMENTO SANGUÍNEO . . . 27
2.1 MODELO MATEMÁTICO . . . 27
2.1.1 CÁLCULO DA PRESSÃO E DO FLUXO SANGUÍNEO . . . 29
2.1.2 CÁLCULO DO COEFICIENTES DE REFLEXÃO E ADMITÂNCIA . 31
2.1.3 CÁLCULO DA IMPEDÂNCIA DE ENTRADA . . . 32
2.1.4 INCORPORAÇÃO DA VISCOSIDADE E VISCOELASTICIDADE NO MODELO . . . 32
3 FERRAMENTA COMPUTACIONAL . . . 35
3.1 ALGORITMOS PARA OS CÁLCULOS HEMODINÂMICOS . . . 36
3.2 ESTRUTURA DE DADOS . . . 40
3.2.1 ELEMENTO DA CLASSE . . . 41
3.2.2 PADRÃO DE PROJETO FÁBRICA . . . 47
3.2.3 OBJETO DA CLASSE . . . 49
3.2.4 OBJETO GRÁFICO DA CLASSE . . . 52
3.2.5 PROJETO DA CLASSE . . . 54
3.2.6 FÁBRICA DE PROJETO DA CLASSE . . . 55
3.2.7 THREADS DA CLASSE *WiseThreadPool* . . . 57
3.2.8 OBJETO DA CLASSE *WiseThreadPool* . . . 61
3.3 USABILIDADE DA FERRAMENTA COMPUTACIONAL . . . 63
3.3.1 CONSOLE . . . 63
3.3.2 CASO DE USO . . . 85
3.3.3 JANELA PRINCIPAL . . . 88
3.3.4 ÁRVORE DE PROJETOS . . . 91
3.3.5 JANELAS . . . 91
4 RESULTADOS OBTIDOS . . . 97
4.1 SIMULAÇÕES HEMODINÂMICAS . . . 97
4.1.1 ESCOAMENTO VISCOSO . . . 98
4.1.2 VASO VISCOELÁSTICO . . . 98
4.1.3 ESCOAMENTO VISCOSO EM VASO VISCOELÁSTICO . . . 98
4.2 ANÁLISES DE DESEMPENHO DA FERRAMENTA COMPUTACIONAL 103
4.2.1 ANÁLISE DE DESEMPENHO DE THREADS . . . 108
4.2.2 ANÁLISE DE CARGA DE TRABALHO . . . 109
5 CONCLUSÕES E TRABALHOS FUTUROS . . . 111
REFERÊNCIAS . . . 113

A PROCESSO DE COMPILAÇÃO 117
A.1 INSTALAR QT 117
A.2 INSTALAR BIBLIOTECAS SECUNDÁRIAS 117
A.3 CLONAR E COMPILAR REPOSITÓRIO 117
B FORMATO DE ARQUIVO DE COMANDOS 119
C FORMATO DE ARQUIVO DE ELEMENTO 121
D FORMATO DE ARQUIVO DE OBJETO 123

1 INTRODUÇÃO

Estudos de simulação hemodinâmica têm sido frequentemente baseados em modelos de árvores arteriais para obter uma melhor compreensão de todos os aspectos relacionados ao escoamento sanguíneo, dentre eles a propagação de ondas e análise do pulso de pressão, passando pelo diagnóstico e inclusive aplicações no planejamento cirúrgico. A representação do sistema cardiovascular através de um modelo puramente 3D que leva em conta a estrutura geométrica exata de todos os vasos não é, no momento, viável computacionalmente, para contornar estes limites computacionais modelos dimensionalmente heterogêneos conhecidos como 0D (zero-dimensional)–1D (unidimensional)–2D (bidimensional)–3D (tridimensional por partes) vêm sendo empregados [13].

Modelos 3D são utilizados para estudar em detalhe a hemodinâmica local de distritos arteriais de interesse, e a geometria destes modelos são provenientes de dados anatômicos obtidos normalmente via reconstrução de imagens médicas de pacientes específicos [26, 33]. Modelos 1D são adotados para representar as artérias de maior calibre e a estrutura geométrica destes modelos pode ser construída a partir de dados anatômicos [6, 14, 31]. Tais modelos são capazes de capturar os efeitos de propagação de ondas [4, 12], a interação das reflexões destas ondas e dar como resultado um pulso de pressão e vazão com significado fisiológico, tanto em artérias centrais quanto periféricas. No entanto, um modelo 1D de toda a árvore arterial sistêmica não é possível devido à falta de dados anatômicos precisos das regiões periféricas. Portanto, a árvore tem que ser truncada em algum nível. Normalmente, este truncamento é feito empregando modelos 0D [23, 31] conhecidos por terminais Windkessel à jusante do modelo 1D para representar o comportamento de distritos arteriais relacionados com o nível de arteríolas e capilares.

Os modelos matemáticos são reconhecidos como boas ferramentas de medição tendo seus resultados comparados com dados *in-vivo* [8]. Para simular corretamente a hemodinâmica corporal, as propriedades viscoelásticas do vaso precisam ser consideradas no comportamento do escoamento cardiovascular. Apesar disso, além da dificuldade de obter dados anatômicos precisos, a quantidade de cálculos necessários e a complexidade matemática e numérica fazem do desenvolvimento de ferramentas confiáveis e práticas para a simulação do sistema cardiovascular humano um dos desafios das próximas décadas [27].

A incidência maior de picos na onda de pressão ao percorrer a aorta já foi documentada como evidência para os efeitos da reflexão em árvores vasculares [21, 22, 24]. Enquanto as áreas de reflexão não podem ser completamente conhecidas ou localizadas, é geralmente aceito que a forma da onda de pressão é modificada significativamente enquanto progride pela aorta, de uma forma que só pode ser explicada por reflexões de onda. Um entendimento mais claro da relação entre modificações e fatores de modificação motiva a busca e desenvolvimento de modelos matemáticos que determinam a forma da onda que o

pulso de pressão toma em cada ponto ao percorrer uma árvore arterial.

Dentro deste contexto, adotou-se neste trabalho o modelo matemático de Duan e Zamir [11, 12] que descreve escoamento sanguíneo pulsátil em árvores arteriais. Estes autores propuseram um modelo relativamente simples para representação da pressão sanguínea e do fluxo em um modelo de árvore arterial. Dentro de cada segmento de vaso, o escoamento sanguíneo foi calculado baseado em uma aproximação de Womersley [37], incluindo a elasticidade da parede, bem como a densidade do sangue e a viscosidade.

O modelo matemático de Duan e Zamir foi escolhido para implementação e simulação computacional por sua capacidade de capturar o pico de pressão existente no escoamento sanguíneo. Este modelo possibilita o cálculo correto das características locais das ondas de pressão e fluxo a medida que elas progridem ao longo de um modelo de árvore 1D e se tornam modificadas por reflexões de onda. Salienta-se que a presença de picos de pressão, a velocidade da onda e a pressão da reflexão de onda podem indicar a existência problemas na circulação periférica [32].

1.1 OBJETIVOS

O principal objetivo deste trabalho é desenvolver uma ferramenta computacional capaz de simular o modelo matemático de Duan e Zamir [12], que descreve um escoamento sanguíneo pulsátil em uma rede arterial com vasos unidimensionais (1D). Os objetivos específicos são:

- simular os efeitos da reflexão de onda no escoamento sanguíneo sobre diferentes condições;
- analisar o desempenho da ferramenta computacional no tocante às threads e carga de trabalho.

Objetivos secundários foram agregados a ferramenta computacional durante o desenvolvimento, facilitando o uso da mesma e o uso em conjunto com outras ferramentas da literatura. Dentre estes objetivos destacam-se:

- visualizar o resultado numérico da simulação hemodinâmica;
- realizar o mesmo experimento com diferentes parâmetros;
- exportar os resultados do escoamento no padrão VTK;
- facilitar futuros experimentos sobre diferentes condições.

1.2 ORGANIZAÇÃO

Os demais capítulos deste trabalho estão organizados como segue. No Capítulo 2, apresenta-se o modelo matemático utilizado para calcular os efeitos do escoamento pulsátil ao atravessar uma árvore arterial sobre três cenários diferentes, o primeiro sobre o efeito da viscoelasticidade dos vasos sanguíneos, em seguida da viscosidade sanguínea e, finalmente, um cenário com ambos os efeitos.

No Capítulo 3, descreve-se a ferramenta computacional desenvolvida em C++, a qual conta com a utilização das bibliotecas Qt/OpenGL. Utilizando propriedades da programação paralela através da comunicação entre *threads*, arquitetura de objetos com objetivo único e utilização da estrutura moderna Fábrica [36].

No Capítulo 4, apresentam-se os resultados numéricos obtidos pela ferramenta computacional proposta.

No Capítulo 5, sumarizam-se as conclusões e salientam-se os próximos passos deste trabalho.

No final do texto, os apêndices contêm informações complementares ao uso da ferramenta. O Apêndice A contém os passos necessários para se compilar a ferramenta computacional. O Apêndice B demonstra o formato esperado de um arquivo de comandos à ser utilizado. Os Apêndices C e D representam os arquivos de saída da ferramenta computacional, um elemento e um objeto inteligente, respectivamente.

2 MODELAGEM DO ESCOAMENTO SANGUÍNEO

Neste capítulo, apresenta-se o modelo matemático de Duan e Zamir [11], que representa o escoamento sanguíneo pulsátil em árvores arteriais. Por fim, apresenta-se um algoritmo que resume os passos dos cálculos realizados para obtenção da pressão e fluxo ao longo da árvore arterial.

O esquema analítico apresentado neste capítulo tem sido empregado por autores para o estudo da hemodinâmica de modelos anatômico e fractal de árvores arteriais [1, 2, 30, 19, 20, 35, 39].

2.1 MODELO MATEMÁTICO

A propagação de ondas em um tubo é governada pelas equações da onda para a pressão $p(x,t)$ e fluxo $q(x,t)$ como seguem:

$$\frac{\partial p}{\partial t} = -\frac{c}{Y}\frac{\partial q}{\partial x}, \quad (2.1)$$

$$\frac{\partial q}{\partial t} = -cY\frac{\partial p}{\partial x}, \quad (2.2)$$

nos quais t é o tempo, x é a coordenada axial ao longo do tubo, c é a velocidade de onda, $Y = \frac{A}{\rho c}$ é a admitância e A é a área da seção transversal do tubo, e ρ é a densidade do fluido. Estas equações são baseadas na linearização das equações de movimento do fluido [10, 22].

Para uma onda harmônica simples, as Equações (2.2) e (2.1) resultam em:

$$p = \bar{p}_0 \exp\left[i\omega\left(t - \frac{x}{c}\right)\right] + R\bar{p}_0 \exp\left[i\omega\left(t - \frac{2L}{c} + \frac{x}{c}\right)\right], \quad (2.3)$$

$$q = Y\left\{\bar{p}_0 \exp\left[i\omega\left(t - \frac{x}{c}\right)\right] - R\bar{p}_0 \exp\left[i\omega\left(t - \frac{2L}{c} + \frac{x}{c}\right)\right]\right\}, \quad (2.4)$$

onde $\omega = 2\pi f$ é a frequência angular, f é frequência em Hertz, L é o comprimento do tubo, $\bar{p}_0$ é a amplitude da onda incidente, R é o coeficiente de reflexão definido pela razão entre as ondas refletidas pelas ondas que chegam no local de reflexão [10, 17] e i é a unidade imaginária ($i^2 = -1$).

O modelo matemático considera que o segmento é composto pelo nó proximal A e pelo nó distal B, considerando o nó proximal o ponto mais próximo da origem do fluxo e o nó distal o ponto mais distante, como apresentado na Figura 1.

As Equações (2.3) e (2.4) para pressão e fluxo são aplicadas em cada segmento de vaso do modelo de árvore arterial, tomando $x = 0$ para o nó proximal e $x = L$ para o nó distal do segmento. Um segmento de vaso é definido pelo intervalo vascular entre dois locais de ramificação [40]. No sistema arterial, as bifurcações são os locais de ramificação mais comuns [38].

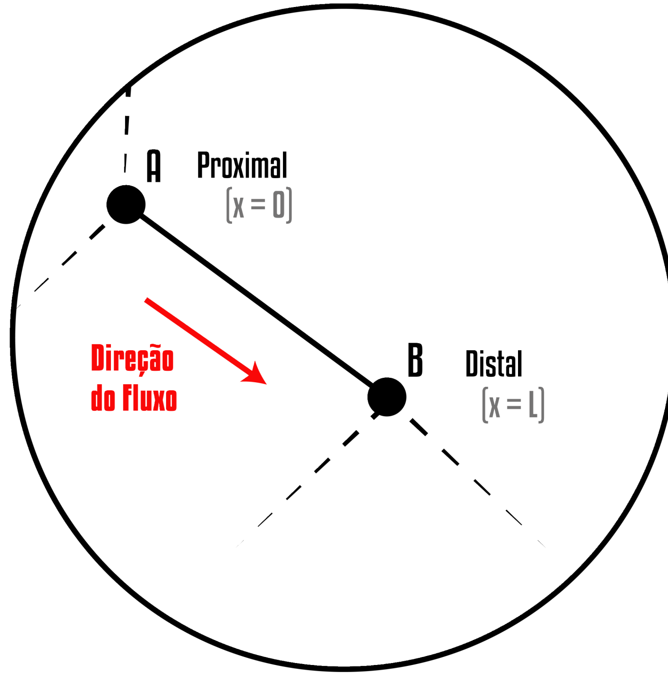


Figura 1 – Segmento do modelo geométrico de uma árvore arterial, composto pelo nó proximal A e pelo nó distal B.

Um segmento de vaso é identificado por (k, j), onde o k representa o nível da geração e j representa a ordem do segmento naquela geração [12], como na Figura 2. Desta forma, a pressão e o fluxo ao longo de um segmento (k, j) do modelo de árvore arterial são dados por:

$$\begin{aligned} p(k,j) &= \bar{p}(k,j)\exp\left[i\omega\left(t-\frac{x(k,j)}{c(k,j)}\right)\right] \\ &+ R(k,j)\bar{p}(k,j)\exp\left[i\omega\left(t-\frac{2L(k,j)}{c(k,j)}+\frac{x(k,j)}{c(k,j)}\right)\right], \end{aligned} \tag{2.5}$$

$$\begin{aligned} q(k,j) &= Y(k,j)\left\{\bar{p}(k,j)\exp\left[i\omega\left(t-\frac{x(k,j)}{c(k,j)}\right)\right]\right. \\ &- \left. R(k,j)\bar{p}(k,j)\exp\left[i\omega\left(t-\frac{2L(k,j)}{c(k,j)}+\frac{x(k,j)}{c(k,j)}\right)\right]\right\}, \end{aligned} \tag{2.6}$$

nos quais $\bar{p}(k, j)$ é a amplitude combinada do grupo de ondas progressivas no segmento (k, j) e $R(k, j)$ é o coeficiente de reflexão no final daquele segmento. O coeficiente de reflexão R é a razão das ondas progressivas pelas ondas atrasadas avaliadas no nó distal $x(k, j) = L(k, j)$.

O grupo de ondas progressivas viaja no sentido positivo de $x(k, j)$, estas são compostas de ondas progressivas vindo de vasos acima deste, bem como, ondas refletidas na junção à montante $x(k, j) = 0$. O grupo de ondas atrasadas viaja no sentido oposto e é composto por ondas vindas de vasos conectados ao nó distal, estas ondas são consideradas a reflexão das ondas progressivas na junção à jusante $x(k, j) = L(k, j)$ e segmentos adjacentes, este grupo de ondas viaja na direção oposta ao fluxo sanguíneo.

As Equações (2.5) e (2.6) descrevem, respectivamente, as ondas de pressão e de fluxo localmente em um segmento (k, j) do modelo de árvore, e localmente na posição $x(k, j)$ dentro deste segmento de vaso. O principal objetivo deste modelo matemático é determinar as duas variáveis desconhecidas: a amplitude da pressão $\bar{p}(k, j)$ e o coeficiente de reflexão $R(k, j)$, que são detalhados na Seção **2.1.1**.

A Figura 2 mostra a notação usada para identificar cada segmento de vaso (k, j), onde k é a geração/nível do vaso e j é um número sequencial dentro daquela geração. Os nós proximal e distal do segmento (k, j) são denotados por A e B na Figura 1, respectivamente. O coeficiente de reflexão $R(k, j)$ do segmento (k, j) está associado ao nó distal B.

Na Equação (2.6), tem-se a admitância característica para cada segmento dada por:

$$Y(k, j) = \frac{A(k, j)}{\rho(k, j)c(k, j)}, \tag{2.7}$$

nos quais $A(k, j)$ é a área da seção transversal do segmento (k, j), $\rho(k, j)$ é a densidade do fluido dentro do vaso e $c(k, j)$ é a velocidade da onda correspondente. A admitância de um segmento é uma medida do quanto o segmento permite o fluxo.

Assumindo um segmento elástico de parede fina, a velocidade da onda $c(k, j)$ é calculada por [10]:

$$c(k, j) = \sqrt{\frac{E(k, j)h(k, j)}{\rho(k, j)d(k, j)}}, \tag{2.8}$$

onde $E(k, j)$ é o módulo de Young, $d(k, j)$ é o diâmetro do segmento (k, j) e $h(k, j)$ é a espessura da parede do segmento, a qual neste estudo é dada por [12]:

$$h(k, j) = 0,05d(k, j). \tag{2.9}$$

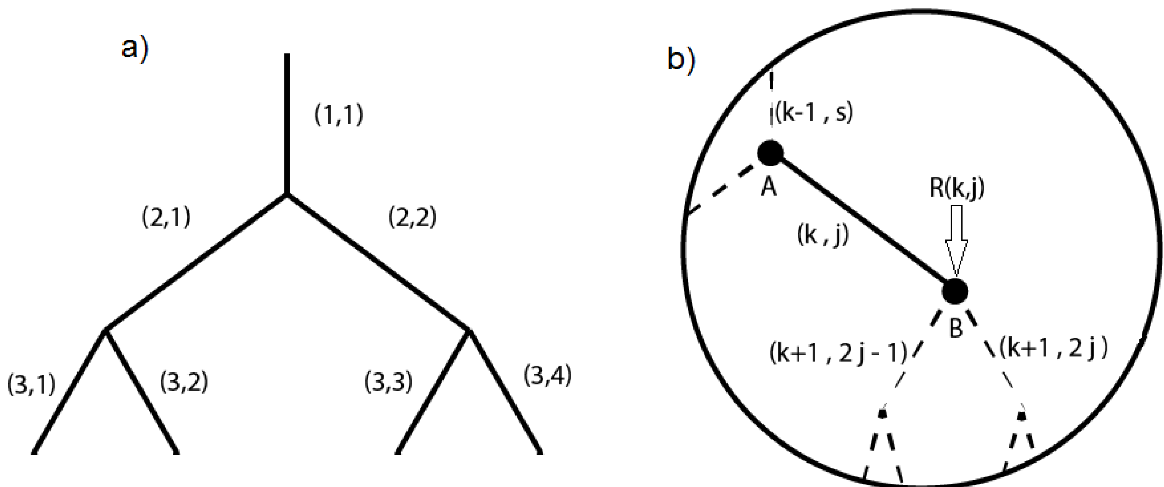


Figura 2 – Notação usada para identificar cada segmento de vaso (k, j) (figura adaptada de [12]).

2.1.1 CÁLCULO DA PRESSÃO E DO FLUXO SANGUÍNEO

Para determinar a pressão $\bar{p}(k,j)$ em um certo segmento (k,j), aplica-se a condição de continuidade de pressão no nó proximal A (ver Figura 2). Escrevendo as componentes progressiva e atrasada da onda como $p_f(k,j)$ e $p_b(k,j)$ respectivamente, a pressão na posição proximal do segmento $x(k,j)=0$ é dada por:

$$[p(k,j)]_A = [p_f(k,j)]_A + [p_b(k,j)]_A\,, \tag{2.10}$$

nos quais as pressões $[p_f(k,j)]_A$ e $[p_b(k,j)]_A$ são expressas por:

$$[p_f(k,j)]_A = \bar{p}(k,j)\exp\left[i\omega t\right], \tag{2.11}$$

$$[p_b(k,j)]_A = R(k,j)\bar{p}(k,j)\exp\left[i\omega\left(t-\frac{2L(k,j)}{c(k,j)}\right)\right]. \tag{2.12}$$

Similarmente, a pressão no segmento pai $(k-1,s)$ pode ser escrita como:

$$p(k-1,s) = p_f(k-1,s) + p_b(k-1,s), \tag{2.13}$$

nos quais s é um número sequencial do segmento pai e as pressões $p_f(k-1,s)$ e $p_b(k-1,s)$ são dadas por:

$$p_f(k-1,s) = \bar{p}(k-1,s)\exp\left[i\omega\left(t-\frac{x(k-1,s)}{c(k-1,s)}\right)\right], \tag{2.14}$$

$$p_b(k-1,s) = R(k-1,s)\bar{p}(k-1,s)\exp\left[i\omega\left(t-\frac{2L(k-1,s)}{c(k-1,s)}+\frac{x(k-1,s)}{c(k-1,s)}\right)\right].$$

No nó distal do vaso superior, $x(k-1,s)=L(k-1,s)$, a pressão é dada por:

$$[p(k-1,s)]_A = [p_f(k-1,s)]_A + [p_b(k-1,s)]_A\,, \tag{2.15}$$

nos quais

$$[p_f(k-1,s)]_A = \bar{p}(k-1,s)\exp\left[i\omega\left(t-\frac{L(k-1,s)}{c(k-1,s)}\right)\right], \tag{2.16}$$

$$[p_b(k-1,s)]_A = R(k-1,s)\bar{p}(k-1,s)\exp\left[i\omega\left(t-\frac{L(k-1,s)}{c(k-1,s)}\right)\right]. \tag{2.17}$$

A condição de continuidade da pressão exige que na junção ela assuma um único valor, portanto

$$[p_f(k-1,s)]_A + [p_b(k-1,s)]_A = [p_f(k,j)]_A + [p_b(k,j)]_A\,. \tag{2.18}$$

Substituindo as Equações (2.11), (2.12), (2.16) e (2.17) na Equação (2.18) e resolvendo para $\bar{p}(k,j)$, resulta em:

$$\bar{p}(k,s) = \frac{\bar{p}(k-1,s)\left[1+R(k-1,s)\right]\exp\left[-\frac{i\omega L(k-1,s)}{c(k-1,s)}\right]}{1+R(k,j)\exp\left[-2i\omega\frac{L(k,j)}{c(k,j)}\right]}. \tag{2.19}$$

Conforme [12], para efeitos de cálculo da pressão e fluxo, adimensionalizam-se as pressões em (2.19) em termos da pressão de entrada $p_0 = \bar{p}_0 \exp[i\omega t]$.

Considerando $P(k,j) = \frac{p(k,j)}{p_0}$ e $\bar{P}(k,j) = \frac{\bar{p}(k,j)}{\bar{p}_0}$, a Equação (2.5) para o cálculo da pressão pode ser expressa de forma adimensionalizada por:

$$\begin{aligned} P(k,j) &= \bar{P}(k,j)\Big\{\exp[-i\beta(k,j)X(k,j)] \\ &+ R(k,j)\exp[-i2\beta(k,j)]\exp[i\beta(k,j)X(k,j)]\Big\}, \end{aligned} \tag{2.20}$$

nos quais $\beta(k,j) = \frac{\omega L(k,j)}{c(k,j)}$ e $X = \frac{x(k,j)}{L(k,j)}$. Similarmente, a Equação (2.6) para o fluxo $q(k,j)$ pode ser obtida de forma adimensionalizada por:

$$\begin{aligned} Q(k,j) &= M(k,j)\bar{P}(k,j)\Big\{\exp[-i\beta(k,j)X(k,j)] \\ &- R(k,j)\exp[-2i\beta(k,j)]\exp[i\beta(k,j)X(k,j)]\Big\}, \end{aligned} \tag{2.21}$$

nos quais $Q(k,j) = \frac{q(k,j)}{q_0}$, $M = \frac{Y(k,j)}{Y(1,1)}$ e $q_0 = Y(1,1)p_0$. O cálculo da admitância $Y(1,1)$ na posição proximal do segmento raiz, ou seja, da artéria de alimentação é apresentado na próxima seção.

2.1.2 CÁLCULO DO COEFICIENTES DE REFLEXÃO E ADMITÂNCIA

Para determinar os coeficientes de reflexão nas junções, consideram-se as duas junções A e B das extremidades de um segmento genérico (k,j) de um modelo de árvore arterial (ver Figura 2). Na posição distal B, o coeficiente de reflexão é definido por [10, 22]:

$$R(k,j) = \frac{Y(k,j) - [Y_e(k+1,2j) + Y_e(k+1,2j-1)]}{Y(k,j) + [Y_e(k+1,2j) + Y_e(k+1,2j-1)]}, \tag{2.22}$$

nos quais $Y_e(k+1,2j-1)$ e $Y_e(k+1,2j)$ são admitâncias efetivas nos segmentos à jusante de B. Estas admitâncias são determinadas pela razão entre o fluxo e pressão naquela posição, que é dada por:

$$Y_e(k+1,s) = \frac{Y(k+1,s)\{1 - R(k+1,s)\exp[-i2\beta(k+1,s)]\}}{1 + R(k+1,s)\exp[-i2\beta(k+1,s)]}, \tag{2.23}$$

nos quais $s = 2j-1$ e $2j$ são os números sequenciais dos dois segmentos filhos e $R(k+1,s)$ é o coeficiente de reflexão na posição distal de cada segmento. Similarmente, $Y_e(k,j)$, a admitância na posição proximal A do segmento (k,j) pode ser dada por:

$$Y_e(k,j) = \frac{Y(k,j)\{1 - R(k,j)\exp[-i2\beta(k,j)]\}}{1 + R(k,j)\exp[-i2\beta(k,j)]}. \tag{2.24}$$

Substituindo $R(k,j)$ da Equação (2.22) em (2.24), obtém-se uma equação para cálculo das admitâncias efetivas ao longo do modelo de árvore arterial:

$$Y_e(k,j) = \frac{Y(k,j)[Y_e(k+1,2j) + Y_e(k+1,2j-1) + iY(k,j)\tan\beta(k,j)]}{Y(k,j) + i[Y_e(k+1,2j) + Y_e(k+1,2j-1)]\tan\beta(k,j)}. \tag{2.25}$$

Em segmentos terminais, pode ser assumido que não ocorrem mais reflexões à jusante das posições distais destes segmentos, portanto a admitância efetiva destes segmentos é igual às suas admitâncias características. Adotando a Equação (2.24), todas as admitâncias efetivas podem ser determinadas percorrendo a árvore a partir dos segmentos terminais até o segmento raiz.

2.1.3 CÁLCULO DA IMPEDÂNCIA DE ENTRADA

A impedância vascular de um modelo de árvore arterial é expresso por

$$z = \frac{p}{q}, \tag{2.26}$$

onde p e q podem ser definidos pelas Equações (2.3) e (2.4), respectivamente. Dados que $p(k,j) = P(k,j)p_0$, $q(k,j) = Q(k,j)q_0$ e $q_0 = Y(1,1)p_0$, em termos de variáveis adimensionalizadas, a impedância pode ser reescrita por

$$Z = \frac{P(k,j)}{Y(1,1)Q(k,j)}. \tag{2.27}$$

A impedância de entrada de um modelo de árvore arterial determina Z na posição proximal do vaso raiz, ou seja, em $x(k,j) = 0$.

Adotando as Equações (2.20) e (2.21) com $x(k,j) = 0$, obtém-se a impedância de entrada:

$$Z = \frac{-1}{Y(1,1)}. \tag{2.28}$$

Em suma, a impedância de entrada em módulo é o inverso da admitância característica da artéria de alimentação.

$$R = Z^{-1} = -Y(1,1). \tag{2.29}$$

2.1.4 INCORPORAÇÃO DA VISCOSIDADE E VISCOELASTICIDADE NO MODELO

A partir do modelo matemático aqui apresentado, os seguintes cenários podem ser investigados nas simulações hemodinâmicas:

- **Cenário 1**: análise do impacto da viscosidade sanguínea ($\mu(k,j)$).
  Os efeitos da viscosidade sanguínea podem ser investigados por substituir a velocidade da onda $c(k,j)$ por uma velocidade da onda complexa:

  $$c_v(k,j) = c(k,j)\sqrt{\epsilon}, \tag{2.30}$$

  onde ϵ é um fator viscoso que corresponde a um tubo elástico com restrições [11]. Seja α o número de Womersley adimensional

  $$\alpha = r(k,j)\sqrt{\frac{\omega\rho(k,j)}{\mu(k,j)}}, \tag{2.31}$$

o fator viscoso ϵ é calculado por:

$$\epsilon = 1 - F_{10}(\alpha), \tag{2.32}$$

onde a função F_{10} é avaliada deste modo:

$$F_{10}(\alpha) = \frac{2}{\alpha\sqrt{i}}(1 + \frac{1}{2\alpha}), \tag{2.33}$$

onde J_p denota a função de Bessel de índice p.

- **Cenário 2**: análise do impacto da viscoelasticidade da parede do vaso (ϕ_0).
  A viscoelasticidade da parede do segmento é incorporada substituindo o módulo de Young estático $E(k, j)$ por um módulo elástico complexo $E_c(k, j)$ no cálculo da velocidade $c(k, j)$ na equação (2.8) da seguinte forma [12]:

  $$E_c(k, j) = |E_c(k, j)| \exp\{i\phi\}, \tag{2.34}$$

  onde ϕ é o ângulo de fase entre a pressão e o deslocamento da parede do segmento [34] expresso por $\phi = \phi_0[1 - \exp(-\omega)]$ e $|E_c(k, j)|$ corresponde ao módulo de Young fornecido para a simulação.

- **Cenário 3**: efeitos da viscosidade sanguínea $\mu(k, j)$ e da viscoelasticidade da parede do segmento ϕ_0 de forma combinada.

  Neste último cenário, utiliza-se a Equação (2.34) para determinar a velocidade da onda $c(k, j)$ (2.8) no modelo. Com este resultado, calcula-se a Equação (2.30) para determinar a velocidade complexa $c_v(k, j)$ a ser considerada no modelo.

3 FERRAMENTA COMPUTACIONAL

Nesta seção apresentam-se detalhes da ferramenta computacional desenvolvida para simulação do escoamento sanguíneo em modelos de árvores arteriais. A principal finalidade desta ferramenta é possibilitar a visualização de estruturas das árvores arteriais e, após a simulação hemodinâmica, dos gráficos que ilustram as curvas de distribuição do fluxo sanguíneo e pressão.

Esta ferramenta foi desenvolvida em C++ utilizando as bibliotecas do Qt 5.15.0 [7] e OpenGL [18], que ajudam na construção da interface gráfica e na exibição de modelos gráficos, como árvores arteriais e gráficos dos resultados. A ferramenta foi disponibilizada em dois ambientes, o primeiro nomeado de *Iterador Gráfico Universal* (IGU), pois em seu modelo de classes qualquer objeto que implemente a classe *WiseObject* (descrito na Seção **3.2.3**) está apto para realizar iterações e desenhar-se através de diretivas OpenGL em um elemento de interface gráfica. O segundo ambiente disponibilizado foi nomeado de *Iterador não-Gráfico Universal* (InGU), pois segue a mesma generalização, entretanto é disponibilizada pelo console, não possuindo interface gráfica.

Buscou-se no desenvolvimento desta ferramenta alto grau de generalização para que seja possível analisar todos os parâmetros de um objeto facilmente e que ela possua grande versatilidade. Além de árvores arteriais, a ferramenta possibilita que outros objetos sejam gerados para visualização, qualquer objeto que implemente a classe *GraphicObject* (descrito na Seção **3.2.4**) pode ser visualizado. A Figura 3 ilustra a ferramenta desenvolvida.

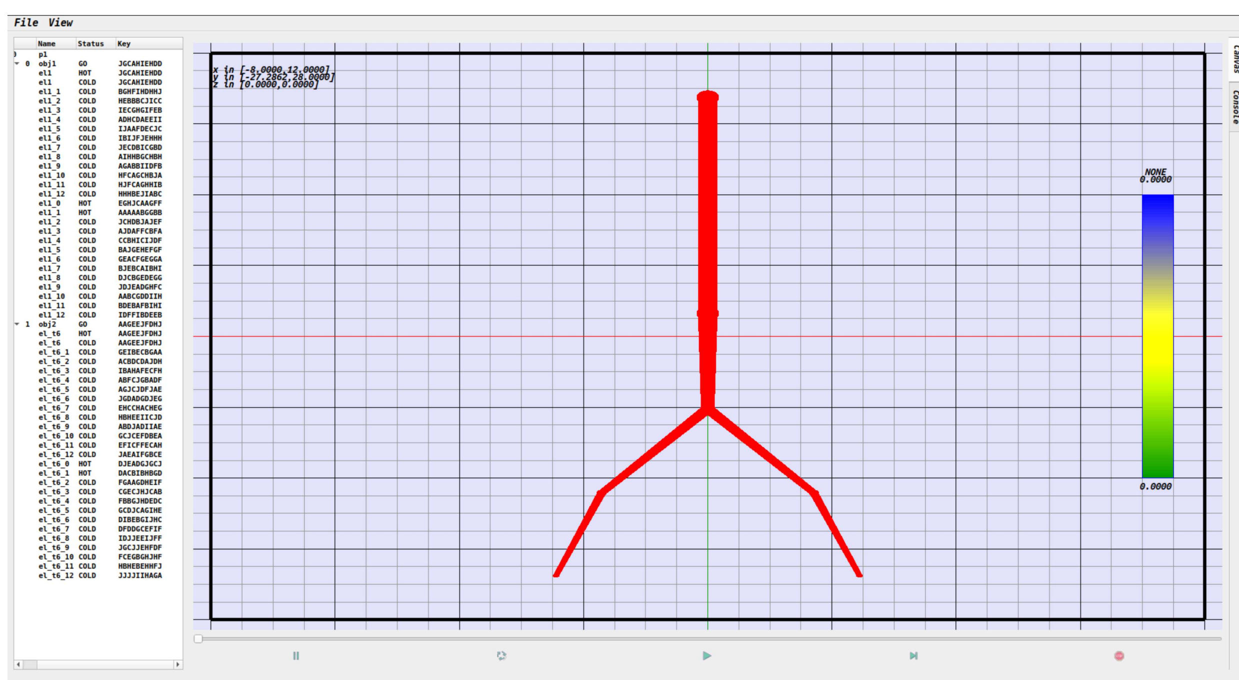


Figura 3 – Interface gráfica da ferramenta construída.

Através da modelagem de classes no paradigma do C++ [25], foi possível realizar

diversas generalizações que ampliam a adaptabilidade dos objetos que podem ser inseridos neste modelo de classes. Na Seção 3.2 é proposto um esquema de classes virtuais, ou abstratas, que foram criadas para facilitar o manuseio dos objetos dentro do ambiente computacional e prover funções básicas.

Uma classe virtual não possui construtor próprio, porque classes virtuais são incompletas. Estas classes possuem métodos virtuais, que por sua vez precisam ser definidos pela classe herdeira. Portanto, uma classe virtual funciona como um conjunto de regras que classes herdeiras devem seguir. Todas as classes que implementem esta classe virtual devem ter sua própria definição dos métodos virtuais, podendo ter funcionamentos distintos, entretanto recebendo os mesmos parâmetros. Adicionado este conceito à estrutura de dados, classes virtuais generalizam os objetos e garantem o seu funcionamento, separando as estruturas de acordo com seu propósito em classes.

Nesta ferramenta computacional ponteiros foram utilizados na organização destas classes virtuais. Quando um ponteiro para uma classe abstrata é definido, é possível que diversas estruturas de dados sejam referências. Ao referenciar um objeto *WiseObject* através de um ponteiro é possível acessar os métodos padrões da classe abstrata que irão acessar a implementação de cada objeto. Todos os objetos irão ter por definição um método que irá construir a estrutura, mas cada objeto irá construir suas próprias estruturas. Portanto, é possível que a ferramenta computacional envie estes dados por referência sem necessariamente reescrevê-los na memória. Este fator foi determinante na escolha da linguagem *C++*, pois um ponteiro é o endereço de memória física, o que permite uma agilidade maior na organização da ferramenta e na utilização destas estruturas pela ferramenta em suas diferentes *threads*[1].

3.1 ALGORITMOS PARA OS CÁLCULOS HEMODINÂMICOS

Nesta seção, detalham-se os algoritmos que descrevem os passos para o cálculo das equações da pressão P e fluxo Q definidas nas Equações (2.20) e (2.21), respectivamente.

Inicialmente, salientam-se que as Equações (2.22) e (2.23) expõem a necessidade de valores $Y_e(k+1, 2j)$ e $Y_e(k+1, 2j-1)$ atrelados aos vasos adjacentes na posição distal do vaso (k, j). Por outro lado, o valor da pressão média do vaso (k, j) calculado usando a Equação (2.19) depende de valores à montante desse vaso, ou seja, do valor $\bar{p}(k-1, s)$. Tendo isto em vista, a estrutura de dados desenvolvida utiliza ponteiros para acessar as propriedades de artérias à montante e a jusante de um vaso do modelo.

Como neste trabalho adotou-se a linguagem de programação C++, os objetos podem se passados para uma outra função por referência. Ao passar um objeto por referência, o endereço de memória é enviado, dando total acesso aos parâmetros do objeto,

[1] Thread é um pequeno programa que trabalha como um subsistema, sendo uma forma de um processo se autodividir em duas ou mais tarefas.

este endereço é chamado de ponteiro. Ao enviar o ponteiro do vaso raiz, é possível acessar todo o modelo de árvore arterial. Cada vaso arterial do modelo contém as propriedades apresentadas na Tabela 1, além de ponteiros para os vasos adjacentes.

Tabela 1 – Propriedades de cada vaso arterial do modelo.

valores conhecidos	**unidade**	**valores calculados**	**unidade**
comprimento (L)	cm	velocidade da onda (c)	cm/s
densidade (ρ)	g/cm^3	beta (β)	–
módulo de Young (E)	g/cms^2	velocidade angular (ω)	rad/s
viscosidade (μ_0)	cm^2/s	espessura da parede (h)	cm
raio (r)	cm	número de Womersley (α)	—
		admitância característica (Y)	cm^4s/g
		admitância efetiva (Y_e)	cm^4s/g
		coeficiente de reflexão (R)	g/cm^4s
		fator viscoso (ϵ)	–
		pressão (P)	–
		pressão média ($\bar{p}$)	g/cms^2
		fluxo (Q)	–

Além das propriedades especificadas na Tabela 1, o vaso possui ainda três ponteiros para os seus vasos adjacentes. Um destes ponteiros, para o vaso adjacente ao seu nó proximal, ou seja, para o seu vaso pai $(k-1, s)$. Os demais ponteiros para os vasos adjacentes ao seu nó distal, ou seja, para seus vasos filhos à esquerda (esq) e à direita (dir) expressos por $(k+1, 2j-1)$ e $(k+1, 2j)$, respectivamente. Este arranjo de ponteiros é equivalente a uma árvore duplamente encadeada, onde através de um único ponteiro para um vaso é possível trafegar a árvore nos dois sentidos. Isto é útil quando é necessário acessar propriedades de outro vaso para se determinarem por exemplo, a pressão média, coeficientes de reflexão e admitâncias de um dado vaso.

Os ponteiros permitem também definir se o vaso é a raiz ou uma folha. Caso o ponteiro para o segmento superior $(k-1, s)$ seja igual ao valor da *flag* nula se trata de um vaso raiz. Caso ambos vasos $(k+1, 2j-1)$ e $(k+1, 2j)$ sejam iguais ao valor da *flag* nula se trata de um vaso folha, isto é, um vaso terminal.

Basicamente, os passos para determinar as ondas de pressão e fluxo que percorrem um modelo de árvore arterial usando o modelo de Duan e Zamir explicado na Seção **2.1.4** são dados por:

1 Cálculo da admitância característica (Y) de cada vaso;
2 Cálculo da admitância efetiva (Y_e) de cada vaso;
3 Cálculo do coeficiente de reflexão (R) de cada vaso;
4 Armazenar valor da admitância característica (Y_r) do vaso raiz, isto é, o valor de $Y(1,1)$;
5 Cálculo da pressão média ($\bar{p}$) em cada vaso;
6 Cálculo das ondas de pressão e fluxo P e Q.

Os passos acima mencionados foram organizados no Algoritmo 2. Esse algoritmo tem como entrada: o modelo de árvore arterial ($\mathcal{MAA}$) com seus vasos caracterizados por suas propriedades, a frequência f, um fator de escala da viscosidade γ_μ, o parâmetro da viscoelasticidade da parede ϕ, a amplitude da pressão média no vaso raiz $\bar{p}_0$, e a quantidade de espaçamento para discretização de um vaso N.

Algoritmo 1: Cálculos hemodinâmicos do modelo de árvore arterial ($\mathcal{MAA}$).

Entrada: $\mathcal{MAA}$, f, γ_μ, ϕ, $\bar{p}_0$, N
1 **início**
2   inicializa vaso v a partir do vaso raiz;
3   **se** *existe(v)* **então**
4   Chama $\mathcal{M}_1$ (v, f, γ_μ, ϕ_0) e armazena Y_r;
5   **fim se**
6   **se** *existe($Y(1,1)$)* **então**
7   Chama $\mathcal{M}_2$(v , $\bar{p}_0$, Y_r, N);
8   **fim se**
9   Retorna $\mathcal{MAA}$ com cada vaso contendo P e Q.
10 **fim**

O Algoritmo 2 proposto tem dois módulos, a saber: $\mathcal{M}_1$ e $\mathcal{M}_2$. O módulo $\mathcal{M}_1$ realiza o cálculo percorrendo o caminho a partir dos vasos finais até o vaso raiz (estratégia do tipo *bottom-up*), visando calcular os valores de admitância efetiva (Y_e), admitância característica (Y) e coeficiente de reflexão (R) de cada vaso. Este módulo é definido pelo Algoritmo 2. Necessitam-se que as admitâncias efetivas dos vasos à justante do vaso k ($Y_e(k+1, 2j-1)$ e $Y_e(k+1, 2j)$) estejam definidas. Em vista disso, verifica-se a existência dos vasos à justante do vaso atual tanto a esquerda (esq) quanto a direita (dir) e caso existam realiza-se uma chamada recursiva. Após o retorno das chamadas recursivas, calculam-se as propriedades do vaso atual. Assim, as propriedades do vasos à jusante do vaso atual serão conhecidas, ou seja, já terão sido determinadas $Y_e(k+1, 2j-1)$ e $Y_e(k+1, 2j)$.

Algoritmo 2: $\mathcal{M}_1$ $(v,\, f,\, \gamma_\mu,\, \phi)$ – Cálculo das admitâncias e coeficiente de reflexão.

1 **início**
2   **se** *existe(v → esq)* **então**
3         $\mathcal{M}_1$ $(v \to esq,\, f,\, \gamma_\mu,\, \phi)$;
4   **fim se**
5   **se** *existe(v → dir)* **então**
6         $\mathcal{M}_1$ $(v \to dir,\, f,\, \gamma_\mu,\, \phi)$;
7   **fim se**
8   Calcula as propriedades c, ω, β;
9   **se** $(\gamma_\mu == 0)$ *e* $(\phi == 0)$ **então**
10   Calcula Y;
11   **fim se**
12   **senão**
13   **se** $(\gamma_\mu\ != \ 0)$ **então**
14   Calcula α, ϵ, E_v, c_v e Y_v;
15   Atualiza $E = E_v$, $c = c_v$ e $Y = Y_v$;
16   **fim se**
17   **se** $(\phi\ != \ 0)$ **então**
18   Calcula E_c;
19   Atualize $E = E_c$;
20   **fim se**
21   **fim se**
22   **se** *(v é um vaso terminal)* **então**
23         $Y_e = Y$ e $R = 0$;
24   **fim se**
25   **senão**
26   Calcula Y_e e R;
27   **fim se**
28 **fim**

O módulo $\mathcal{M}_2$ realiza os cálculos percorrendo o modelo a partir do vaso raiz até os vasos terminais (abordagem do tipo *top-bottom*), visando calcular o valor da pressão média, pressão e fluxo em cada vaso. Este módulo é expresso no Algoritmo 3. Como expresso na Equação (2.19), o valor da pressão média requer o valor da pressão média do vaso à montante $(k-1, s)$. Logo, inicialmente, se o vaso atual é a artéria de alimentação (raiz), neste caso $\bar{p} = \bar{p}_0$. Caso contrário, calcula-se $\bar{p}$ com a Equação (2.19). Em seguida, o valor das ondas de pressão e fluxo são obtidas ao longo de cada vaso 1D, que são discretizados através de N espaçamentos. Por fim, a recursão é enviada aos segmentos inferiores, desta

forma se garante a existência de um valor $\bar{p}(k-1,s)$.

Algoritmo 3: $\mathcal{M}_2$ – Cálculo da pressão e fluxo ao longo de cada vaso.

```
1  M2 (v, p̄0, Yr, N)
2  início
3     se (v é o vaso raiz) então
4        p̄ = p̄0;
5     fim se
6     senão
7        Calcula p̄;
8     fim se
9     Calcula P(Xi) e Q(Xi), onde Xi ∈ [0,1];
10    se existe( v → esq ) então
11       M2 (v → esq, p̄0, Yr, N);
12    fim se
13    se existe(v → dir) então
14       M2 (v → dir, p̄0, Yr, N);
15    fim se
16 fim
```

No Algoritmo 3 tem-se que N denota a quantidade de espaçamento dX no intervalo [0,1] para discretização de um vaso. Assim, $dX = 1/N$ e $X_i = idX(i = 0, 1, \cdots, N)$.

3.2 ESTRUTURA DE DADOS

Nesta seção, apresentam-se detalhes da estrutura de dados elaborada para a ferramenta computacional. Como mencionado na Seção 3.1 , foi empregada uma estrutura para armazenar todas as referências às informações do modelo geométrico da árvore arterial.

Iterar o modelo matemático na ferramenta computacional equivale à aplicar um objeto inteligente a sua fábrica de iteração *WiseIterationFactory* (descrito na Seção **3.2.2**). A fábrica de iteração *DuanAndZamirIterationFactory* acoplada à uma árvore arterial *WiseArterialTree* é equivalente a execução do Algoritmo 10 em todo o modelo geométrico, definindo a onda de pressão P e fluxo Q em todos os segmentos.

A ferramenta computacional foi desenvolvida para ser capaz de armazenar, carregar e iterar o modelo matemático e gerar novas estruturas para visualização. A estrutura de dados proposta para a ferramenta precisa realizar estas operações e armazenar corretamente os dados gerados.

As seções a seguir descrevem a estrutura de dados genérica, tendo como principal objeto de estudo, o escoamento pulsátil através de um modelo de árvore arterial 1D e

a visualização dos seus resultados. Em versões anteriores, a ferramenta computacional definia seus modelos matemáticos, geométricos e de visualização em apenas uma estrutura abstrata, o que acarretou em classes grandes com difícil manutenção e pouca versatilidade. Isto porque o mesmo objeto ficava responsável por diversas funções: instanciar, iterar e desenhar na tela através de diretivas OpenGL. Portanto, na versão atual da ferramenta computacional as classes foram desenvolvidas para ter propósito único, cujo objetivo é garantir que as classe tenham uma função apenas, garantindo uma classe com um número reduzido atributos e métodos, o que facilita a manutenção e entendimento.

Ao longo do desenvolvimento da ferramenta, três principais processos foram identificados: (1) a iteração dos objetos, uma vez carregados estes objetos são atualizados por algoritmos, gerando novas versões; (2) o armazenamento dos objetos, os elementos criados no método de iteração podem ser armazenados e então recuperados; e (3) a exibição dos objetos, os elementos carregados podem ter seus parâmetros visualizados em um elemento de interface gráfica *OpenGL* [18]. O ciclo iterativo irá gerar ao final uma animação, para tanto cada iteração irá gerar um quadro desta animação. O objeto *WiseObject* irá armazenar todos estes quadros, cada quadro representa um elemento *WiseElement*. Cada quadro antes de ser exibido precisa ser processado em diretivas *OpenGL*, o responsável sobre esse processo é o elemento gráfico *GraphicObject*. Igualmente quando se tratam de animações e reproduções de vídeo, somente quadros necessários estarão em memória.

Finalmente, estas estruturas são utilizadas pela ferramenta computacional possibilitando que modelos geométricos sejam carregados, iterados e então exibidos, sem que haja perda dos quadros ou de algum dado. Os elementos principais da estrutura de dados serão descritos nas seções que se seguem.

3.2.1 ELEMENTO DA CLASSE

Elementos são objetos que implementam a classe *WiseElement*. O principal objetivo de um elemento é manter os dados mais recentes de um modelo geométrico e os dados necessários para o método de iteração. No caso do estudo do escoamento pulsátil, considerou-se que uma iteração seja a execução completa do algoritmo apresentado na Seção 3.1 em toda a árvore. Portanto, cada elemento possuirá todas as informações do modelo geométrico de uma árvore arterial, seus segmentos e suas propriedades.

Através das malhas estruturadas e não estruturadas presentes na biblioteca *VTK* (*Visualization ToolKit*) [5] é possível descrever diversas estruturas de dados através de elementos padronizados, como pontos, linhas e células. Utilizando os elementos básicos contidas nestas malhas, a estrutura básica da arquitetura foi criada e está presente na Figura 4.

A Figura 4 mostra que um elemento inteligente é composto por duas outras estruturas: *WiseStructure* - utiliza pontos, linhas, células e campos para determinar

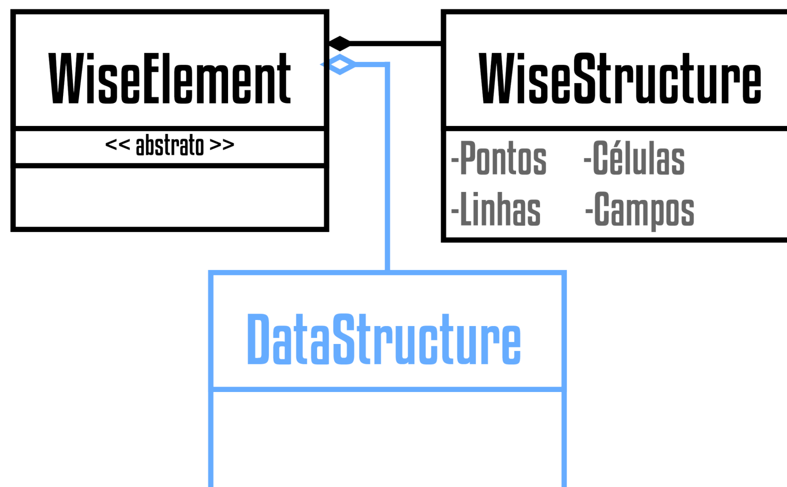


Figura 4 – Representação de classes de um elemento. *WiseElement* é uma classe abstrata base e seus componentes são: *WiseStructure* representa a estrutura contida em um arquivo VTK e DataStructure representa a estrutura de ponteiros e variáveis utilizadas na iteração.

estruturas geométricas; *DataSructure* - representa os dados abstratos específicos de cada elemento. Estas estruturas são equivalentes entre si, isto é feito para que a primeira estrutura mantenha um formato padrão de pontos, linhas, células e campos. Enquanto dados abstratos equivalentes podem ser utilizados, os dados mantidos na estrutura padrão *WiseStructure* são utilizados principalmente na leitura e escrita de objetos, portanto são mantidos como vetores que possuem cadeias de caracteres, os dados presentes nesta estrutura podem ser editados pelo usuário. Enquanto os dados abstratos contém a mesma informação contida em variáveis da linguagem, como números inteiros, vetores, ponteiros e outros.

Cada parâmetro de um elemento representa uma grandeza associada à um componente do modelo geométrico (pontos, linhas e células) ou ainda sobre todo o modelo (campos). Os parâmetros armazenam valores definidos por um nome e uma referência ao modelo geométrico. Os raios r de uma árvore arterial são dados relacionados à um vaso (segmento 1D), que é representado por uma linha. Como só existe um raio por segmento, somente um valor será armazenado. Os valores de pressão P através de um segmento com $X \in [0, 1]$ estão relacionados ao mesmo componente geométrico, mas são representados por uma sequência de valores. A estrutura de parâmetros permite que sejam armazenados um ou múltiplos valores no mesmo local e os acesse da mesma forma. Ao acessar o valor do raio r de um segmento, um vetor é recebido com apenas uma posição e ao acessar a pressão P são esperados N elementos.

Como ilustrado na Figura 5 um elemento é aquele que implementa a classe abstrata *WiseElement*. Os dados abstratos *DataStructure* de cada classe podem ser salvos na estrutura padrão *WiseStructure* e utilizados quando necessário.

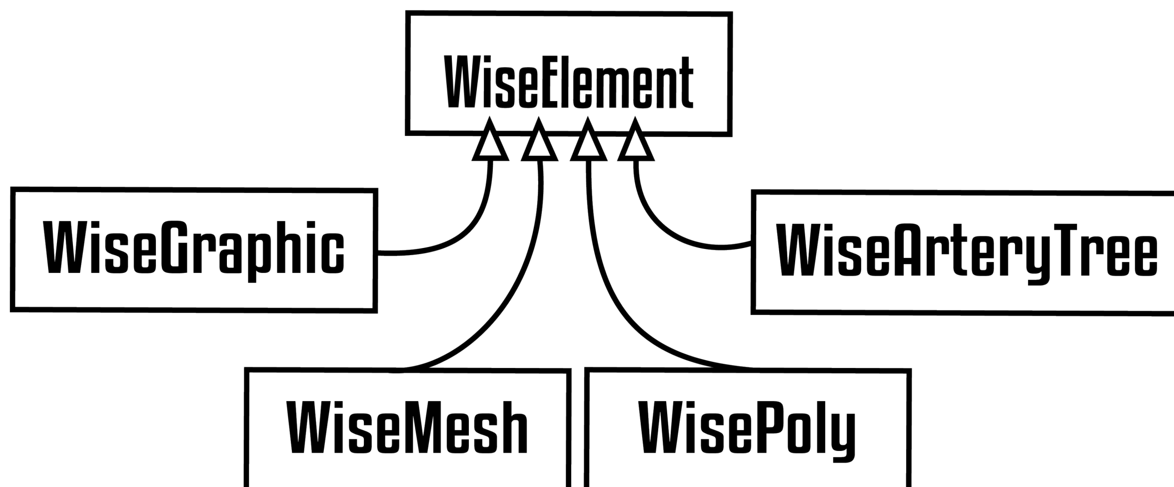


Figura 5 – Tipos de elementos. *WiseGraphic*, um gráfico bidimensional. *WiseMesh*, uma malha tridimensional. *WisePoly*, um cubo. *WiseArteryTree*, uma árvore arterial.

O modelo geométrico de uma árvore arterial foi traduzido para o elemento *WiseArteryTree*. A estrutura deste elemento conta com pontos e linhas que definem os parâmetros do modelo geométrico na estrutura *WiseStructure*. A estrutura abstrata *DataStructure* conta com ponteiros para cada segmento e grandezas físicas armazenadas em pontos flutuantes de precisão dupla.

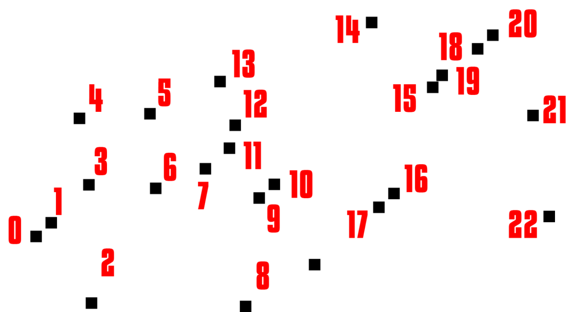
Foram criados ainda outros elementos, como *WiseGraphic* que representa um gráfico, *WiseMesh* representa um plano e o elemento *WisePoly* representa um cubo. A seguir mais detalhes sobre estes elementos.

O elemento *WiseGraphic* foi criado para armazenar dados ao longo de uma dimensão, como a pressão ao longo da árvore arterial. Os elementos *WiseMesh* e *WisePoly* foram criados para armazenar dados ao longo de duas e três dimensões respectivamente, e foram utilizados como exemplo de objetos que podem ser generalizados na ferramenta e visualizados mudando pouco as estruturas já existentes. O elemento *WiseGraphic* é equivalente a um vetor de valores (x, v), o elemento *WiseMesh* é equivalente a uma matriz de valores (x, y, v) e o elemento *WisePoly* é equivalente a uma matriz tridimensional de valores (x, y, z, v), onde v é o valor associado e (x),(x, y) ou (x, y, z) denota uma posição. Através deste modelo, é possível armazenar em um *WiseMesh* o valor v da pressão ao longo de uma árvore arterial na direção x variando o valor da frequência no eixo y.

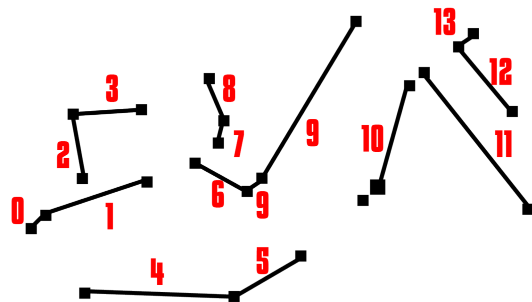
Os elementos servem como estruturas de armazenamento padrão que compõem um outro objeto. O ciclo de manipulação desses elementos se divide em três partes: (1) criação - os objetos podem ser criados a partir de exemplos pré-definidos ou através de um arquivo de entrada *VTK* ou *XML (eXtensible Markup Language)*; (2) iteração - processo em que o elemento com todas as estruturas definidas e consistentes é utilizado por um algoritmo; (3) exibição - este último ciclo utiliza os dados armazenados nesta estrutura e gera um objeto para visualização.

Todo elemento completo possui uma redundância de dados, podendo ser representado por qualquer uma das duas estruturas que o compõe. A estrutura *WiseStructure* utiliza componentes simples para descrever as estruturas e seus parâmetros. Essa estrutura é a que mais se assemelha à encontrada na arquitetura *VTK*. Por utilizar uma quantidade limitada de diretivas o arquivo proveniente dessa estrutura pode ser rapidamente lido, interpretado e até mesmo exportado.

Pontos



Linhas



Células

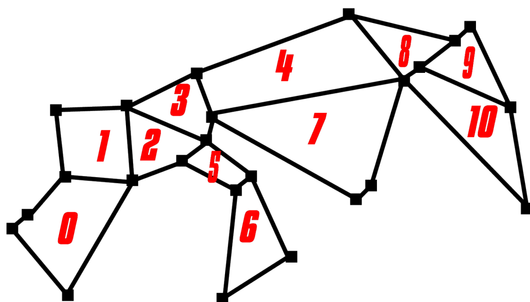


Figura 6 – Pontos, linhas e células para especificação do modelo geométrico.

Os componentes de uma estrutura são pontos, células, linhas e campos. Pontos demarcam uma posição no espaço, células e linhas representam conjuntos de pontos e os campos são dados gerais da estrutura. Um modelo de pontos e linhas é capaz de representar o mesmo modelo geométrico de uma árvore arterial, basta considerar cada ponto uma bifurcação de vasos e cada segmento de vaso uma linha. Através destes componentes é possível armazenar e acessar dados sobre cada segmento. Para dados gerais do modelo, como a frequência f, são utilizados os campos da estrutura.

A classe *WiseElement* foi desenvolvida para construir a estrutura abstrata (*DataStructure*) a partir das informações contidas na estrutura (*WiseStructure*). Desta forma é importante que a estrutura possa ser armazenada e recuperada rapidamente. O elemento é o elemento básico do método de iteração, sendo duplicado antes de ser iterado, gerando novas estruturas idênticas. Imediatamente, a nova estrutura abstrata *DataStructure* é des-

cartada e os dados contidos na nova estrutura *WiseStructure* são armazenados. Portanto, a estrutura abstrata não estará sempre presente. Seguindo este ciclo de vida, todos os elementos obedecem à máquina de estados contida na Figura 7.

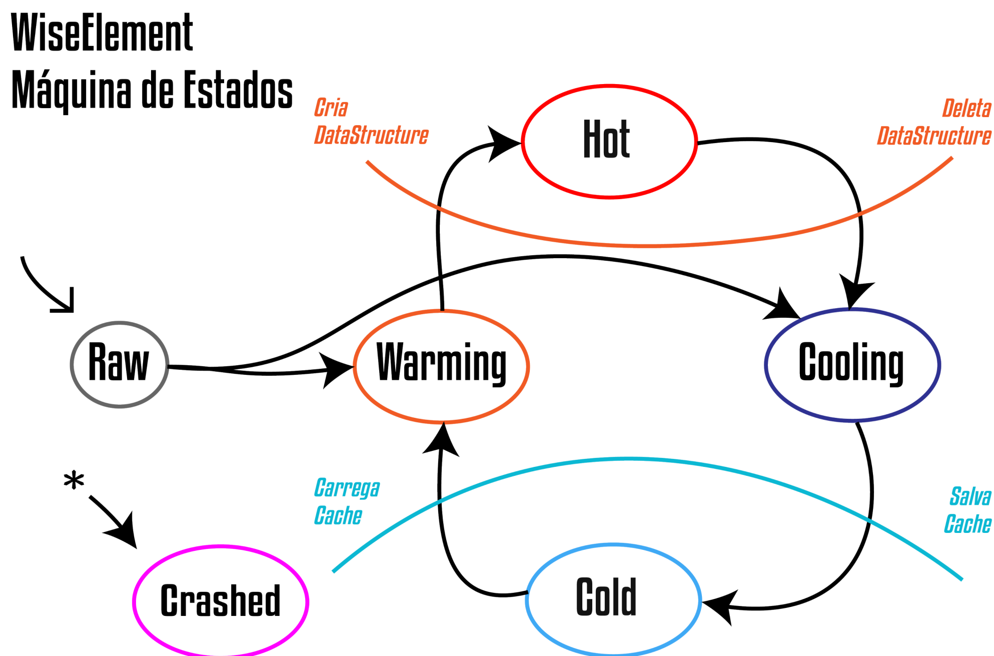


Figura 7 – Máquina de estados que controla o funcionamento de um elemento inteligente.

A máquina de estados dos elementos gerencia o ciclo de vida deles. Inicialmente, os elementos recebem o estado *Raw*, ou cru, que representa um elemento ainda sem estruturas carregadas ou sem consistência garantida. Uma vez que a estrutura é inserida e verificada o elemento muda para o status *Warming* ou *Cooling*, respectivamente esquentando ou esfriando.

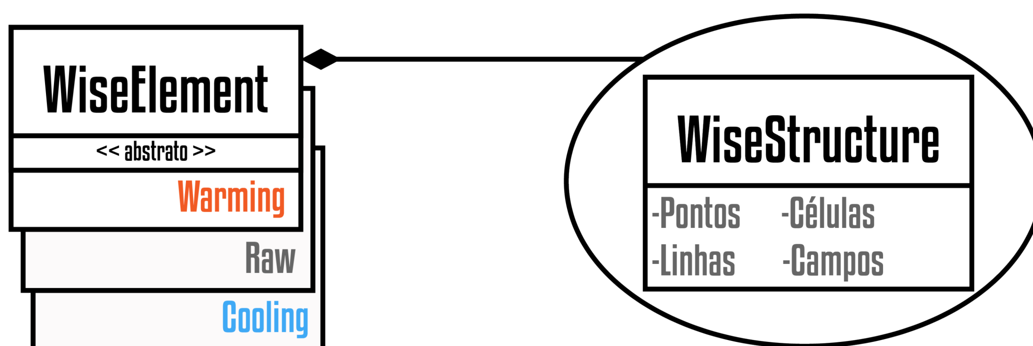


Figura 8 – Elemento no estado *Warming*, *Raw* e *Cooling*.

A Figura 8 mostra as estruturas presentes em elementos no estado *Warming*, *Raw* e *Cooling*. Cada estado possui uma finalidade diferente e representa em que estado estão as informações do elemento. Elementos no estado *Warming* estão aguardando a construção de sua estrutura abstrata, enquanto elementos no estado *Cooling* aguardam que seus

dados sejam salvos em memória. Finalmente, elementos no estado *Raw* indicam que não é esperada a consistência dos dados.

Inicialmente os elementos são criados no estado *Raw*, enquanto os dados do elemento são carregados ele permanecerá neste estado. Ao final do carregamento o objeto trocará de estado para *Warming* ou *Cooling*, que indicam o próximo passo do elemento. Estar no estado *Warming* indica que o objetivo é esquentar o objeto, atingindo o estado *Hot*. Enquanto o estado *Cooling* indica que a estrutura aguarda resfriamento, atingindo o estado *Cold*. Quando os dados abstratos são criados corretamente para o elemento seu estado é definido como *Hot*, neste estado todas as informações do objeto estão em memória, sendo este o estado mais pesado do elemento. Em contrapartida, o estado *Cold* indica que a estrutura foi corretamente armazenada em disco e os elementos neste estado estão em seu estado mais leve, pois possuem em memória apenas o caminho para a estrutura armazenada.

Como ilustrado na Figura 7, o estado *Cold* está associado com o uso da memória para armazenamento de estruturas inteligentes. A estrutura do arquivo é equivalente a uma malha estruturada VTK, mas são efetivamente salvos em um arquivo *XML*. Caso sejam novamente carregados por uma mudança de estado ou deletados o arquivo armazenado é deletado.

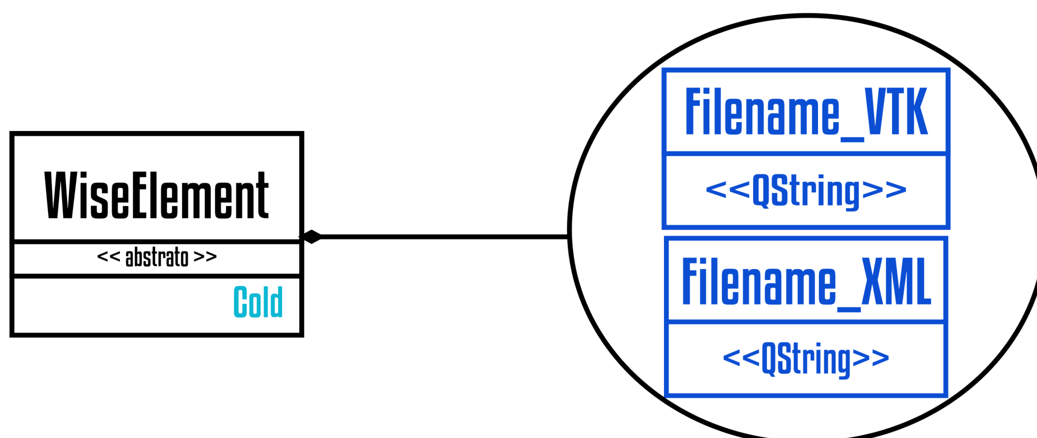


Figura 9 – Elemento no estado *Cold*.

O estado *Crashed* serve para identificar objetos que não tem mais o funcionamento esperado. Durante a troca de estados do elemento é verificado se as estruturas esperadas estão presentes, caso não estejam o objeto é direcionado a este estado.

Finalmente, o estado *Hot* representa os elementos que possuem todas as estruturas presentes. Isto significa que a estrutura *WiseStructure* e a estrutura abstrata equivalente *DataStructure* estão presentes e são consistentes.

Para que um elemento possa ser iterado ele precisa estar no estado *Hot*, porque durante a iteração de um elemento seus dados abstratos são utilizados. A cada passo da

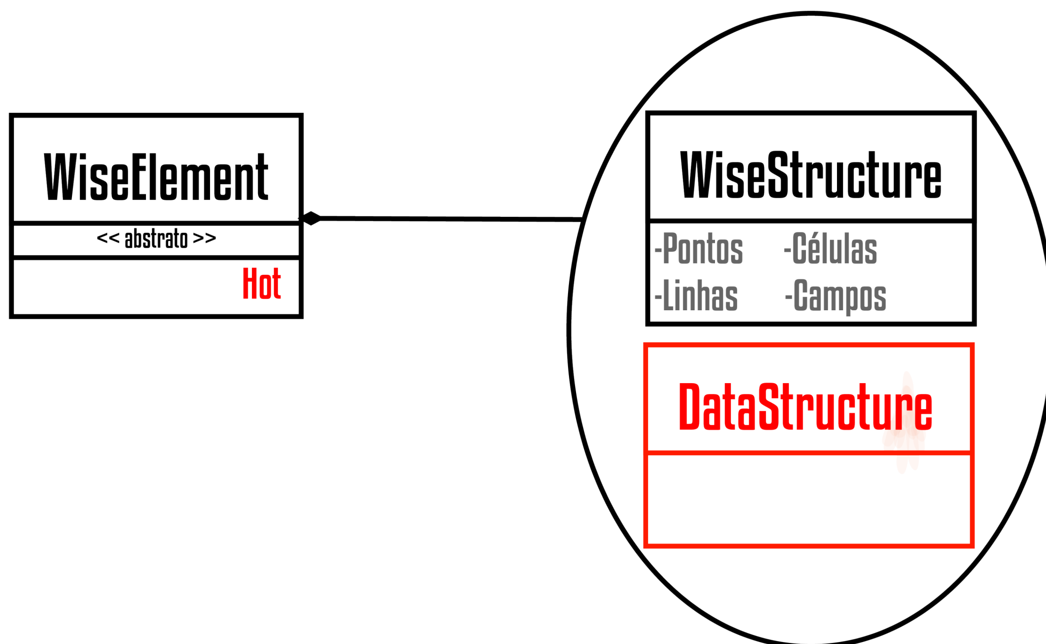


Figura 10 – Elemento no estado *Hot*.

iteração a estrutura *DataStructure* é atualizada, exigindo uma atualização da estrutura *WiseStructure*.

Seguindo o conceito de classes com propósito único, os elementos são responsáveis apenas por gerir os dados contidos na estrutura e abstrata. Portanto, as funções de criar, iterar e visualizar estas estruturas foram divididas em outras classes.

3.2.2 PADRÃO DE PROJETO FÁBRICA

Como mencionado anteriormente, três principais barreiras foram identificadas armazenamento, iteração e visualização. Nesta seção, uma arquitetura de classes que permite a execução de cada passo através do paradigma de Fábricas Dinâmicas [36] é proposta. Esta arquitetura com Fábricas que permitem a criação de instâncias com definições concretas armazenadas como metadados. Isso facilita a adição de novos objetos que podem ser interpretados sem modificar o código da fábrica em si.

A fábrica é uma classe que possui construtores de outras classes. Este tipo de classe permite que os métodos de construção do objeto sejam separados da lógica interna do objeto, sendo útil em classes com uma construção complexa. Sendo estas fábricas também classes virtuais, três principais tipos de fábricas foram idealizadas: (1) uma que cria os elementos, (2) uma que itera estes elementos e (3) uma que cria um elemento gráfico a partir de um elemento.

Primeiramente, a fábrica de elementos *WiseElementFactory* é utilizada para a criação de elementos, essa fábrica é utilizada para criar um elemento a partir de parâmetros pré-definidos. Esse tipo de fábrica é também uma classe virtual, seus métodos definem

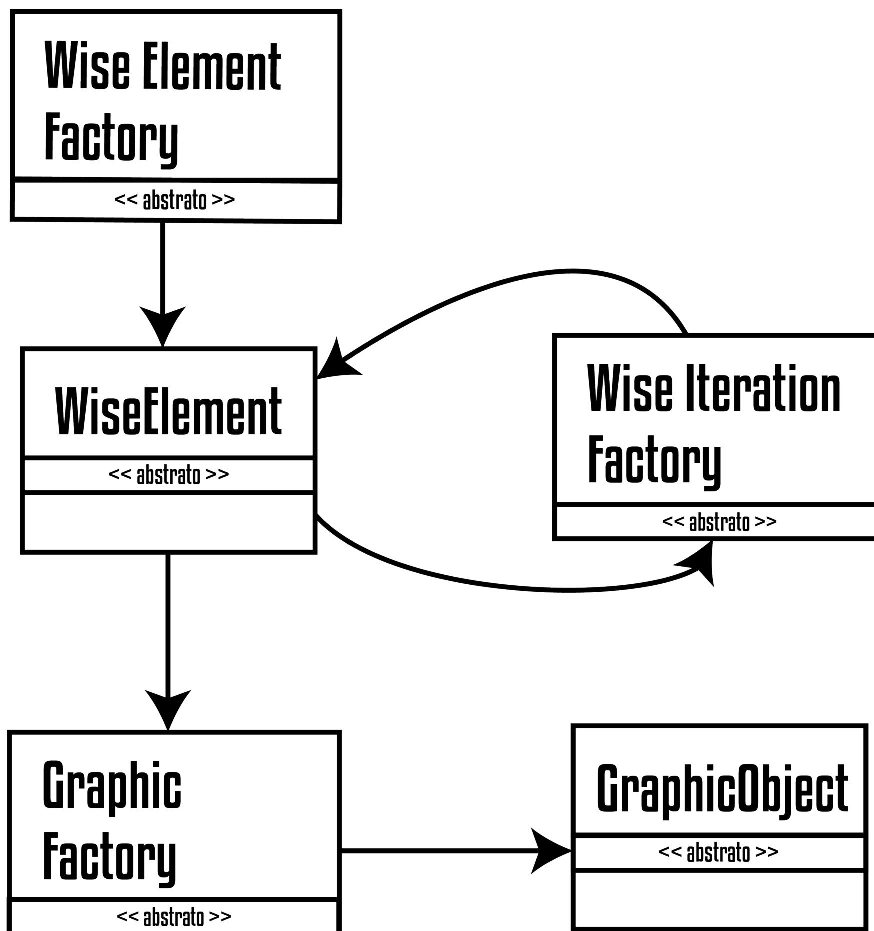


Figura 11 – Arquitetura de classes fábrica e fluxo de trabalho do elemento *WiseElement*. A fábrica *WiseElementFactory* é responsável por criar o elementos, a fábrica *WiseIterationFactory* é responsável pela iteração do elemento e a fábrica *GraphicFactory* é responsável por criar as estruturas de visualização.

que fábricas de elementos possuem três maneiras de executar: criar elementos à partir de uma lista de exemplos; criar elemento a partir de um arquivo *VTK* ; e criar o elemento a partir de um arquivo *XML*.

Os métodos de criação de elementos são selecionados através do nome da fábrica e de um dos três métodos de criação. A ferramenta computacional possui uma lista com todas a fábricas disponíveis e as utiliza quando necessário. Devido à forma como os dados são carregados para cada elemento, é necessário que haja uma fábrica para cada tipo de elemento.

Similarmente, a fábrica *WiseIterationFactory* só pode operar com um tipo específico de elemento. Entretanto, é possível que haja mais de uma fábrica de iteração disponível por tipo de elemento. Desta forma, uma árvore arterial pode ser iterada por diferentes algoritmos de iteração e o mesmo ocorre com os outros tipos de elementos. Estas fábricas são responsáveis por executar algum algoritmo que utilize o tipo de dados do elemento. No caso de uma árvore arterial é possível utilizar uma fábrica que irá executar o modelo

matemático descrito na Seção 3.1 utilizando os ponteiros para segmentos disponíveis em uma *WiseArteryTree*.

Por último, a fábrica *GraphicFactory* irá criar o objeto gráfico correspondente ao elemento. Assim como um elemento um objeto gráfico pode ser salvo em memória. Um elemento é composto por todas as linhas, pontos, células e seus valores associados, enquanto um objeto gráfico contém as representações gráficas destas estruturas e permite a visualização destes valores associados. Os elementos estruturais de uma árvore arterial (seus pontos e linhas) são traduzidos em esferas e cilindros como visto na Figura 3. A cor destas esferas e cilindros representam algum valor associado à estrutura, apenas um valor pode ser exibido por vez ao longo da escala de cores, como o fluxo Q ou a pressão P. Desta forma, os elementos gráficos só exigem que um parâmetro por vez seja armazenado.

3.2.3 OBJETO DA CLASSE

A classe que combina todas as estruturas utilizadas no método de iteração da ferramenta computacional foi nomeada de objeto *WiseObject*. Este objeto preserva todos os passos de de iteração e poupa a quantidade de recursos mantida em memória.

A classe *WiseObject* é composta por um coleção de elementos e objetos gráficos equivalentes entre si. Nessa classe, a presença de uma fábrica gráfica é opcional, possibilitando que objetos sejam iterados sem que alguma estrutura seja disponibilizada para visualização. Utilizando a mesma separação de classes com propósito único, fábricas dinâmicas garantem que um objeto seja criado corretamente. As fábricas presentes na Seção **3.2.2** foram incluídas como propriedades de um objeto, desta forma estes objetos serão compostos por três fábricas: *WiseElementFactory*, *WiseIterationFactory* e *GraphicFactory*.

O objeto possuirá duas estruturas de armazenamento de elementos. O elemento guardado no *Forno* será o objeto utilizado pelo algoritmo a cada iteração, portanto é sempre o mais recente. A cada ciclo iterativo o elemento contido no *Forno* é duplicado e uma nova instância é adicionada ao *Freezer*, estes elementos serão armazenados e recuperados quando necessário.

O ciclo de vida de um objeto consiste na sua criação, a iteração de um modelo matemático e, opcionalmente, a exibição de um modelo gráfico. Objetos são criados com seu primeiro elemento. Ao criar um objeto, sua fábrica adiciona em sua estrutura a fábrica de elementos. Desta forma objetos são capazes de replicar seus elementos. No momento da criação, o primeiro elemento é duplicado e uma instância segue para o *Forno*, enquanto a outra segue para o *Freezer*.

Através do modelo de classes de um objeto presente na Figura 12 é possível identificar todos os componentes presentes em um objeto. O objeto troca seus elementos de estado automaticamente. A coleção de elementos *WiseCollection* irá manter apenas um elemento em memória. O elemento contido no *Forno* que deve permanecer no estado

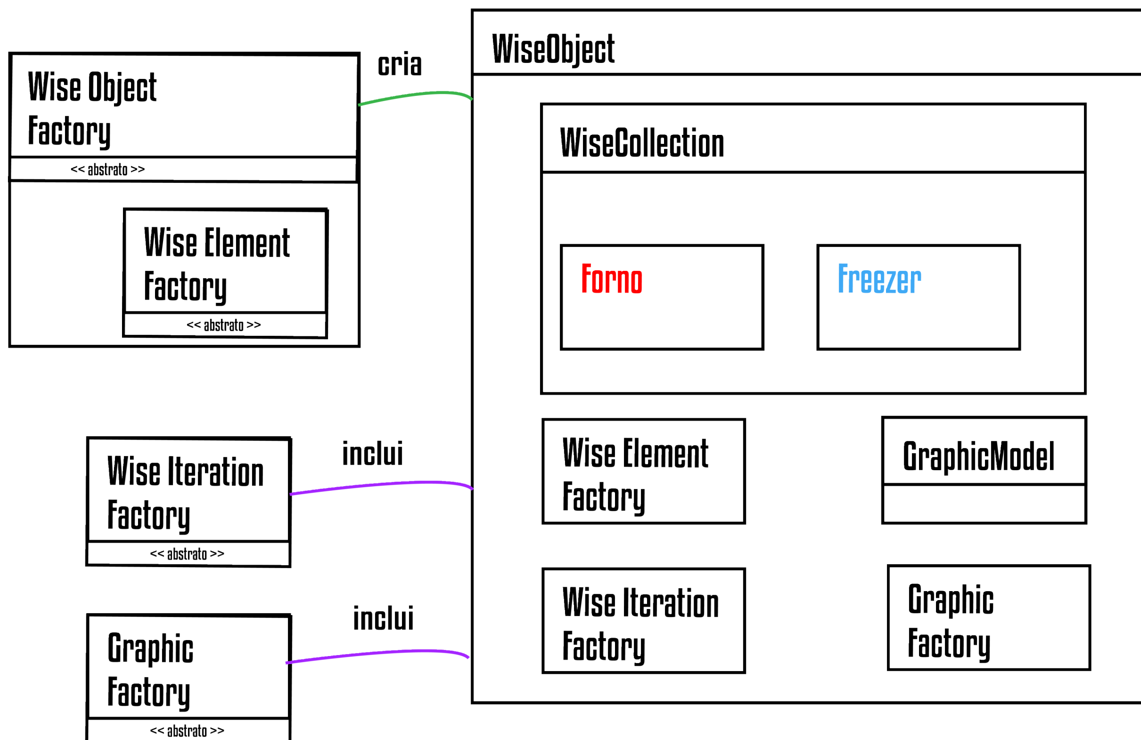


Figura 12 – Objeto *WiseObject* e todos seu componentes: *WiseObjectFactory*, fábrica responsável pela criação de objetos; *WiseElementFactory*, fábrica responsável pela criação de elementos; *WiseIterationFactory*, fábrica de iteração; *WiseGraphicFactory*, fábrica gráfica; *WiseCollection*, coleção de elementos; *GraphicModel* coleção de objetos gráficos.

Hot. Ao mesmo tempo a coleção irá manter um histórico de elementos armazenados no *Freezer* que devem permanecer no estado *Cold*. É possível que o objeto volte à um estado anterior, substituindo o elemento presente no *Forno* com algum estado anterior armazenado no *Freezer*. Dessa forma o elemento utilizado no método iterativo será o objeto recuperado. Ao realizar essa operação, a coleção de elementos irá recuperar a estrutura do elemento, alterando seu estado para *Warming*, em seguida irá recriando seus dados abstratos utilizando a fábrica de elementos disponível, além de alterar seu estado para *Hot*. Estas mudanças de estados são descritas pela máquina de estados do objeto na Figura 13.

As trocas de estado dos objetos são causadas por operações do usuário para preparar o objeto para iteração e para executar o método de iteração. Desta forma, o usuário pode definir quais fábricas serão inseridas no objeto, os parâmetros do modelo geométrico do experimento e quais trocas de estados devem ser executadas.

Como objetos são criados a partir de um elemento, eles possuem os mesmos métodos de criação. Desta forma, as fábricas de objetos *WiseObjectFactory* são compostas por fábricas de elementos *WiseElementFactory*. Finalmente, com este modelo de classes, foram disponibilizadas duas formas de criar objetos inteligentes, utilizando um elemento já existente ou os métodos de criação de elementos disponíveis na fábrica de elementos.

Em seguida, é necessário definir qual será a lógica de iteração do objeto, adicionando

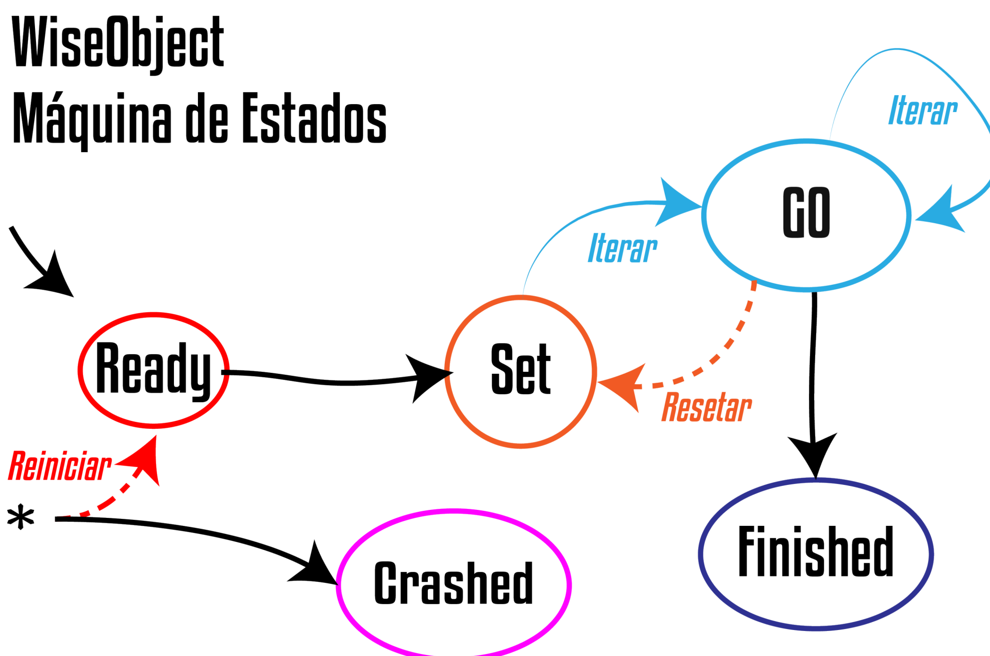


Figura 13 – Máquina de estados utilizada pelos objetos *WiseObject*. O estado *Ready* indica que o objeto foi criado corretamente e o estado *Set* indica que o objeto teve suas fábricas corretamente adicionadas. Enquanto o objeto estiver iterando ele permanecerá no estado *Go* e quando finalizar irá para o estado *Finished*. O estado *Crashed* é utilizado quando o objeto não funciona corretamente.

uma fábrica de iteração *WiseIterationFactory* compatível. Para o caso de escoamento pulsátil através de uma árvore arterial é necessário adicionar a fábrica de iteração correspondente ao algoritmo da Seção 3.1 e em seguida definir os parâmetros desejados, como a frequência f, a viscosidade μ e o ângulo de fase ϕ. Com as alterações concluídas o objeto passa a poder ser iterado, opcionalmente uma fábrica gráfica *GraphicFactory* pode ser inserida. Neste caso, o objeto passará a gerar objetos gráficos *GraphicObject* a cada iteração.

Inicialmente, um objeto é criado no estado *Ready* somente com seu elemento inicial e uma fábrica do tipo *WiseElementFactory*. Neste estado é esperada a inclusão das fábricas de iteração e gráfica. Uma vez que elas estejam corretamente acopladas ao objeto, é possível fazer a troca do estado *Ready* para o estado *Set*. Com a mudança de estado, é adicionado à estrutura *WiseStructure* todos os parâmetros disponibilizados pelas fábricas de iteração e gráfica. Um objeto no estado *Set* indica que o objeto foi corretamente criado, uma fábrica de iteração foi adicionada, possivelmente uma fábrica gráfica, e agora o objeto aguarda alterações nestes parâmetros ou execução do método iterativo.

Com os parâmetros definidos e as fábricas devidamente acopladas, o objeto está pronto para a iteração. O método iterativo de um objeto é representado na transição para o estado *Go*. Uma iteração de uma *WiseArteryTree* representa o cálculo dos valores de pressão e fluxo em toda a árvore arterial. Caso algum erro ocorra durante o processamento

de dados o objeto se desloca para o estado *Crashed*, assim como elementos. É possível também finalizar a execução de um objeto ao enviá-lo para o estado *Finished*, neste estado o objeto não poderá ser iterado novamente. Para que parâmetros da iteração possam ser alterados sem que se perda elementos, é possível que um objeto no estado *Go* retorne para o estado *Set*. Todas estas trocas de estados são gerenciadas pela máquina de estado apresentada na Figura 13.

3.2.4 OBJETO GRÁFICO DA CLASSE

Quando o objeto *WiseObject* estiver corretamente carregado e uma fábrica gráfica *GraphicFactory* for adicionada à sua estrutura o primeiro objeto gráfico *GraphicObject* de sua coleção será criado. Assim como um elemento representa uma iteração, um objeto gráfico representa um ciclo iterativo. Enquanto a estrutura responsável por elementos, *WiseCollection*, é responsável por manter os últimos dados de iteração aquecidos, a coleção de objetos gráficos *GraphicModel* mantém o objeto que está sendo exibido por alguma tela. As coleções tem como objetivo manter seus elementos e realizar trocas de estados quando necessário. No caso dos elementos, somente o último objeto é necessário e o mesmo ocorre no contexto dos objetos gráficos, apenas o objeto exibido em um elemento da interface de usuário e seus vizinhos serão mantidos em memória.

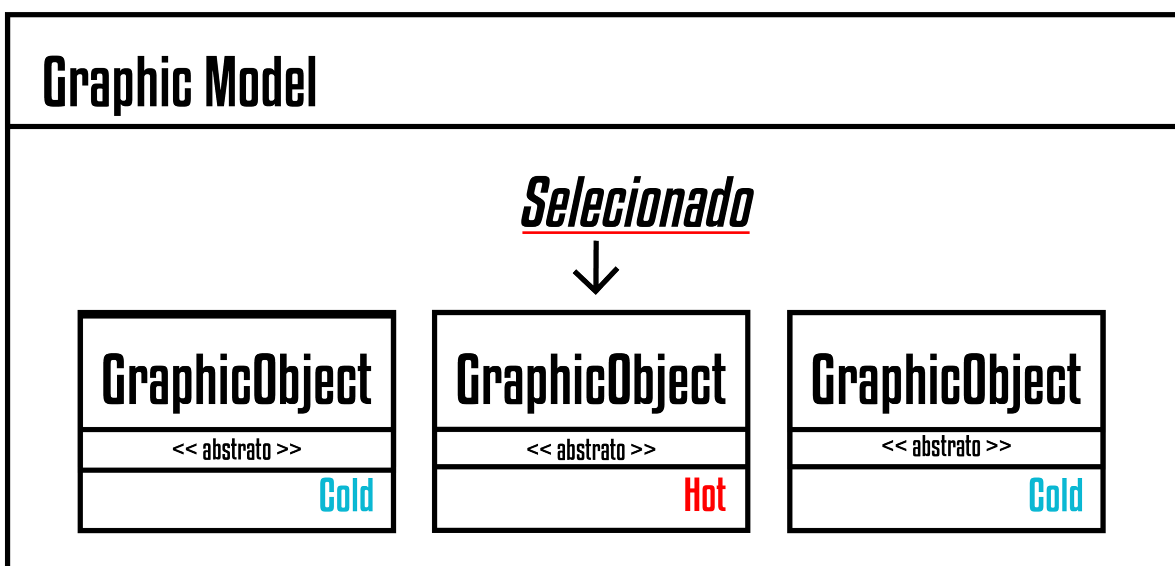


Figura 14 – Modelo gráfico *GraphicModel* contém uma coleção de objetos gráficos.

Na Figura 14 estão os componentes que permitem armazenar todos os quadros da animação final. Quando um objeto *WiseObject* está corretamente configurado com uma instância de fábrica gráfica *GraphicFactory*, seu método iterativo é atrelado à criação de objetos gráficos. Isso permite que o objeto iterado crie um elemento *WiseElement* e um objeto gráfico *GraphicObject* a cada iteração.

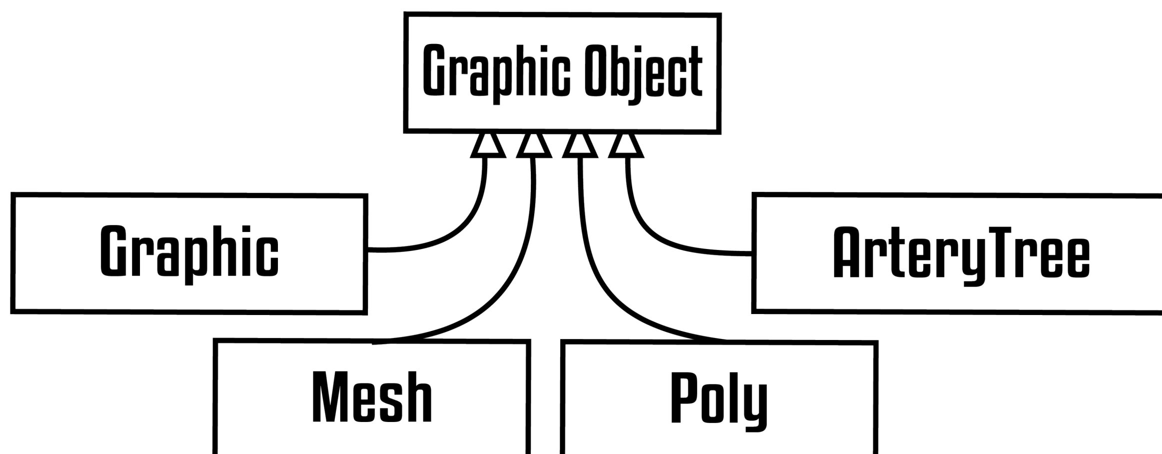


Figura 15 – Tipos de objetos gráficos *GraphicObjects*, representando cada tipo de elemento *WiseElement*. O objeto *Graphic* representa um gráfico, o objeto *Mesh* uma malha, o objeto *Poly* um cubo e o objeto *ArteryTree* uma árvore arterial.

Cada objeto gráfico é desenhado através de diretivas OpenGL. O formato do modelo geométrico é apresentado ao usuário através das formas geométrica, enquanto algum parâmetro do elemento *WiseElement* é visualizado através de uma escala de cores.

Os objetos gráficos contidos na Figura 15 possuem métodos específicos para desenhá-lo utilizando uma escala de cores e formas geométricas padrão, bidimensionais ou tridimensionais. Cada forma geométrica utilizada é representada por elementos gráficos *GraphicElement*. A cada iteração a fábrica utilizará o elemento contido na estrutura *Forno* e irá retirar a informação do elemento mais atual. Cada objeto gráfico possui um valor máximo e mínimo que é vinculado à escala cores. Logo as cores representam os valores armazenados em cada objeto gráfico.

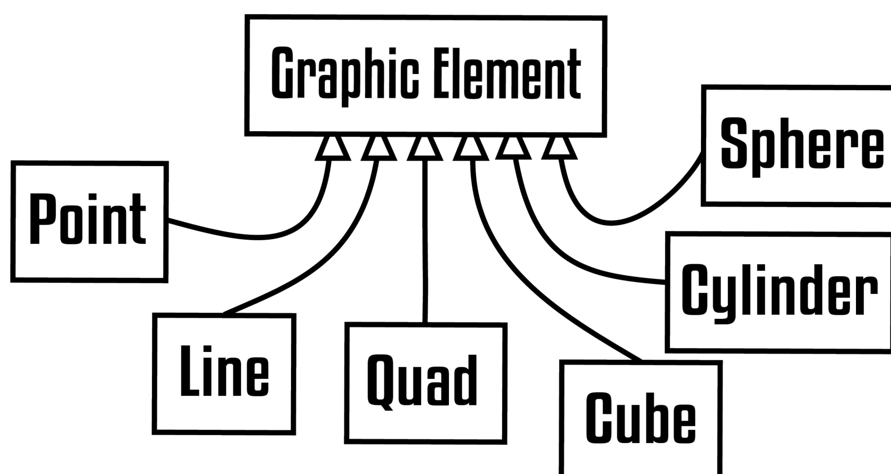


Figura 16 – Tipos de elementos gráficos *GraphicElements*. *Point*, um ponto. *Line*, uma linha. *Quad*, um quadrado. *Cube*, um cubo. *Cylinder*, um cilindro. *Sphere*, uma esfera.

Os elementos gráficos podem ser utilizados por qualquer tipo de objeto gráfico. A representação geométrica de uma árvore arterial é construída utilizando cilindros e esferas,

que representam segmentos de vaso e terminais, respectivamente. Por padrão, os cilindros receberão uma lista de valores da pressão $P(X)$ para $\forall X \in [0, 1]$, enquanto as esferas receberão os valores da pressão $P(X)$ quando $X = 0$ ou $X = 1$, condicionado a escolha do nó distal ($X = 1$) ou o nó proximal ($X = 0$). Assim como os objetos *WiseObject* têm seus tipos definidos pelo tipo de elemento, o objeto gráfico *GraphicObject* tem seu tipo definido pelo elemento que representa com sua fábrica própria.

Nesta estrutura gráfica ficam presentes o modelo geométrico, a escala de cores utilizada e o parâmetro a ser visualizado. Apesar do objeto gráfico *GraphicObject* ser uma redundância dos dados armazenados em um elemento *WiseElement*, ele é menor por conter somente um dos parâmetros armazenado, podendo ser carregado e armazenado mais rapidamente. O objetivo das estruturas *GraphicModel*, *GraphicObject* e *GraphicElement* é permitir que diversos objetos gráficos *GraphicObjects* possam ser armazenados e rapidamente carregados em memória. Os objetos gráficos em sequência representam uma animação que pode ser visualizada através da interface gráfica.

3.2.5 PROJETO DA CLASSE

O projeto *WiseProject* é o escopo de trabalho da ferramenta computacional e são utilizados na organização dos objetos. O primeiro passo para utilizar a ferramenta é criar um projeto que irá conter todos os objetos e elementos criados.

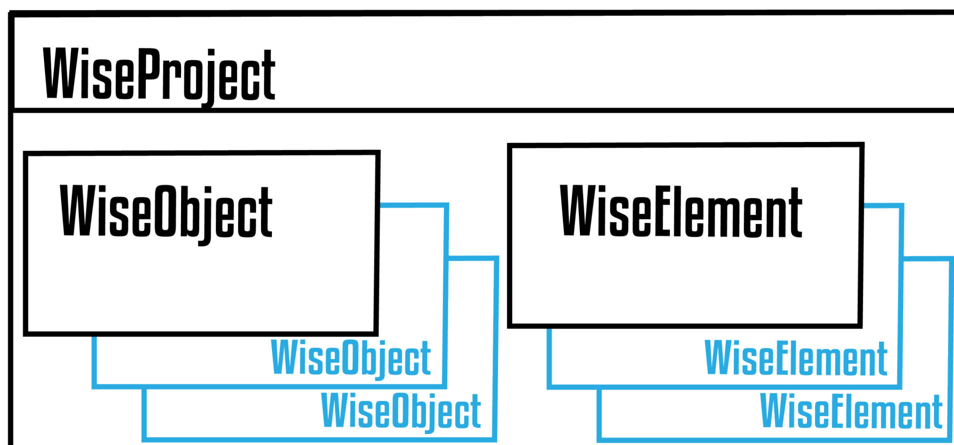


Figura 17 – Projeto *WiseProject* e seus componentes, uma lista de objetos *WiseObject* e uma lista de elementos *WiseElement*.

Como ilustrado na Figura 17, o projeto inteligente é representado por duas coleções, uma de elementos *WiseElement* e outra de objetos *WiseObject*. Os comandos recebidos pela ferramenta computacional terão efeito apenas sobre um projeto e suas coleções. Por exemplo, quando um elemento for utilizado na criação de um objeto eles devem estar no mesmo projeto.

O projeto é uma estrutura organizacional e quando um comando é executado sobre objetos de um projeto, estes objetos são bloqueados até o final da execução do comando.

Enquanto permanecer bloqueado um objeto não pode ser excluído ou alterado por outro comando. Ao excluir um projeto, todas as demandas devem ser finalizadas. Finalmente, as estruturas dos projetos podem ser salvas e carregadas, portanto estas estruturas também servem para armazenar experimentos inteiros e seus resultados.

3.2.6 FÁBRICA DE PROJETO DA CLASSE

O conceito de fábrica foi adicionado ao projeto para padronizar a construção de objetos. Inclusa neste conceito, a fábrica de projetos *WiseProjectFactory* é responsável pela construção de projetos. Diferentemente das outras, a *WiseProjectFactory* contém todas as fábricas suportadas pela ferramenta computacional.

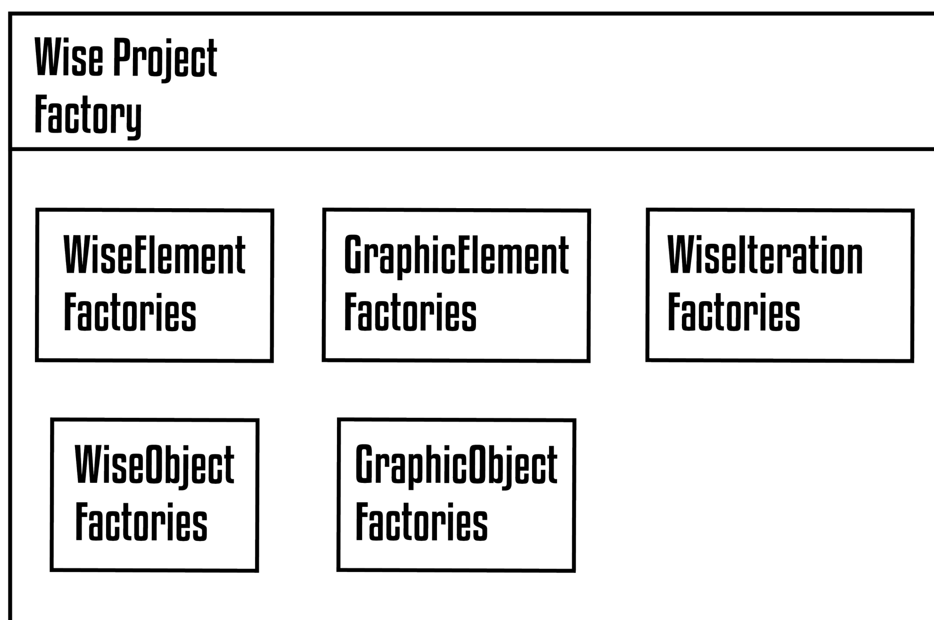
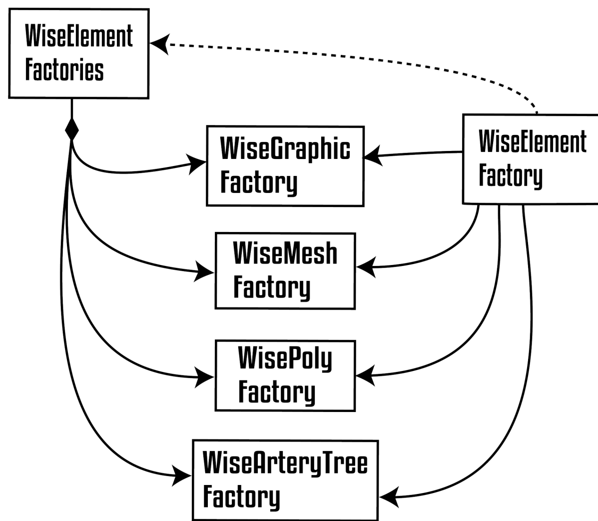


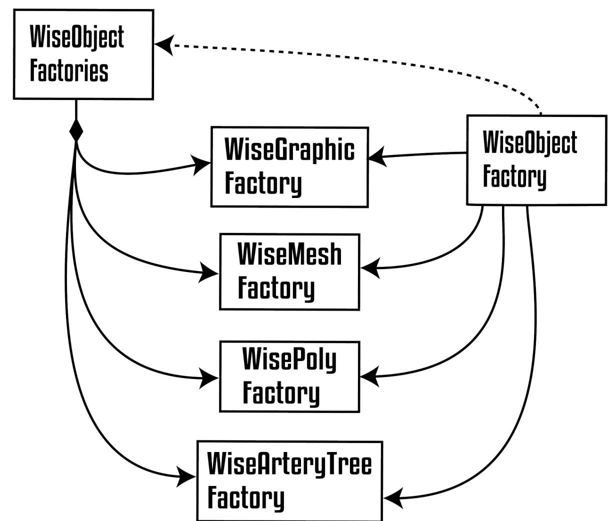
Figura 18 – Fábrica de projeto *WiseProjectFactory* e seus componentes: fábricas de elementos *WiseElementFactories*, fábricas de objetos *WiseObjectFactories*, fábrica de objetos gráficos *Graphic Object Factories*, fábricas de elementos gráficos *GraphicElementFactories* e fábricas de iteração *WiseIterationFactories* .

Como é possível observar na Figura 18, a fábrica de projetos inteligentes é composta de outras coleções de fábricas, agrupadas pelo tipo de estrutura que criam. Nesta classe estão as fábricas de elementos *WiseElement*, objetos *WiseObject*, objetos gráficos *GraphicObject*, elementos gráficos *GraphicElement* e iteração *WiseIterationFactory*. Por exemplo, quando um projeto contendo diversas árvores arteriais e gráficos é carregado ele é construído pela fábrica de projetos. A fábrica de projetos irá reconhecer o formato de cada estrutura e encaminhar para a fábrica correspondente.

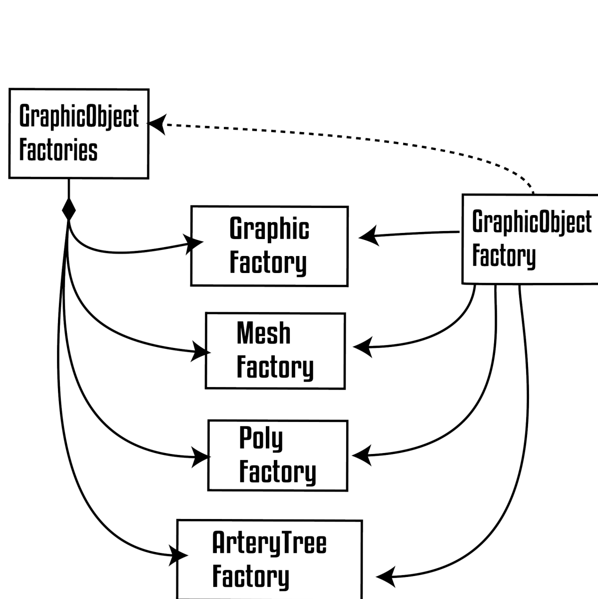
As fábricas de iteração descritas na Figura 19 foram criadas para resolver o modelo matemático descrito na Seção 2.1 e extrair o seu resultado. A fábrica de iteração estática *StaticIterationFactory* é uma fábrica que não altera o objeto inteligente no processo de



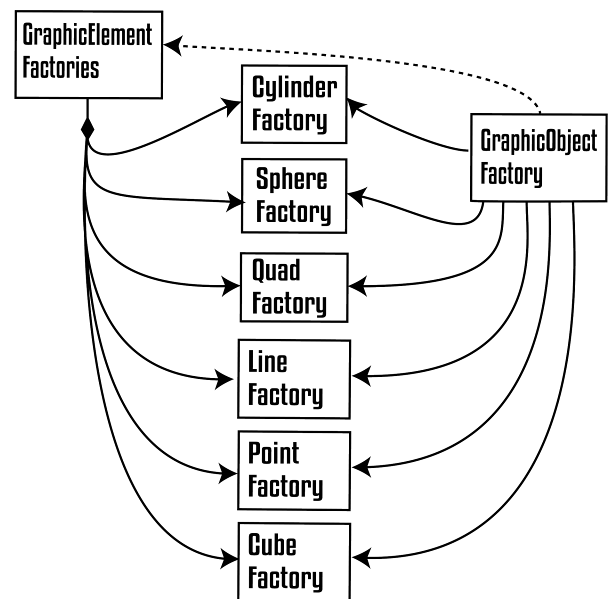
(a) Fábricas de elementos



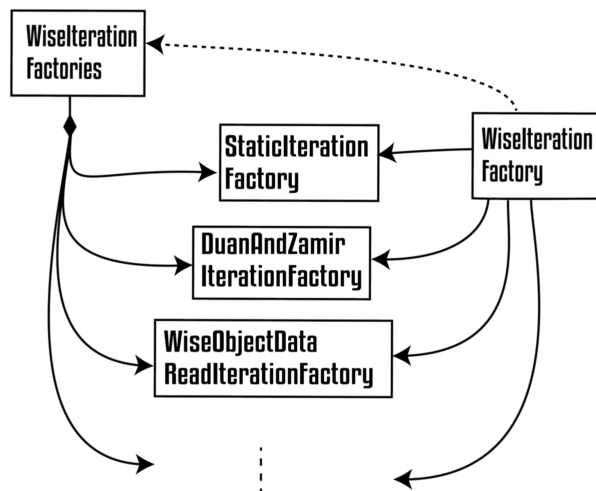
(b) Fábricas de objetos



(c) Fábricas de objetos gráficos



(d) Fábricas de elementos gráficos



(e) Fábricas de iteração

Figura 19 – Todas as fábricas que compõem uma fábrica de projeto.

iteração, esta é a única classe de iteração que pode ser utilizada em mais de um tipo de objeto, podendo ser executada em qualquer tipo de objeto inteligente.

O modelo do escoamento pulsátil proposto está presenta na fábrica de iteração *DuanAndZamirIterationFactory*, esta fábrica é responsável por utilizar a estrutura de uma árvore arterial *WiseArteryTree* e aplicar o modelo matemático. Finalmente, a fábrica de iteração *WiseObjectDataReadIterationFactory* é uma fábrica responsável por armazenar os resultados obtidos a cada iteração e armazenar em uma estrutura do tipo *WiseMesh*.

3.2.7 THREADS DA CLASSE *WiseThreadPool*

Até o momento as estruturas foram construídas utilizando conceitos padrões da linguagem C++, herança de propriedades através do polimorfismo, ponteiros, classes virtuais e fábricas dinâmicas. Nesta seção o conceito de programação paralela e divisão de tarefas acoplado à ferramenta computacional é proposto.

A arquitetura de computadores atual permite aos usuários da aplicação executar processos que irão utilizar os recursos físicos do computador. O processador é a unidade central de processamento que executa milhões de tarefas por segundo. No entanto, executa apenas uma instrução por vez. Quando diversos processos estão em execução no computador o sistema operacional é responsável por balancear o tempo de processamento que cada processo ganha, com isso somos capazes de executar diversos programas ao mesmo tempo, este ciclo é conhecido como escalonamento de processos.

Processadores mais recentes estão equipados com múltiplos núcleos de processamento, o que permite a execução de diversas tarefas ao mesmo tempo. Para que um único processo tire vantagem desta arquitetura ele precisa se reestruturar em *threads*, estes elementos são partes do processo e cada uma possui seu próprio ciclo de execução e é avaliado separadamente no escalonamento de processo. Desta forma, um processo é capaz de se beneficiar de uma arquitetura com múltiplos núcleos. Entretanto, é necessária uma reorganização do processo uma vez que todas as *threads* irão dividir o mesmo espaço de memória. Como cada *thread* possui o seu próprio ciclo de execução, para que possam trocar informações é necessário que haja um tratamento para possibilitar a comunicação assíncrona.

Observou-se que as estruturas possuem comportamentos padronizados e que um objeto é autônomo. Ele sozinho é capaz de se armazenar, reconstruir, iterar e ocasionalmente se desenhar. Imaginando um cenário em que diversos objetos estão iterando, as tarefas foram divididas em três tipos de *Threads*, primeiramente trabalhos de leitura e escrita, em seguida a iteração dos objetos.

Estes objetos estão disponíveis para todas as *threads* através do projeto *WiseProject* e são enviados por referência no trabalho *WiseJob* caso não estejam bloqueados. Desta forma, computadores com múltiplos núcleos podem fazer uso de suas *threads* e permitir

que mais de um objeto realize suas tarefas por vez.

Ao longo da pesquisa a estrutura de classes foi agregada as bibliotecas comuns do Qt 5.15.0 [7], em um primeiro momento, para facilitar a visualização dos resultados através dos elementos gráficos de interface de usuários disponibilizados. Em seguida, permitiu o processamento de elementos de forma paralela. Com isso um modelo que suportasse o processamento paralelo utilizando classes *QThreads*, que possuem tarefas concorrentes, foi construído.

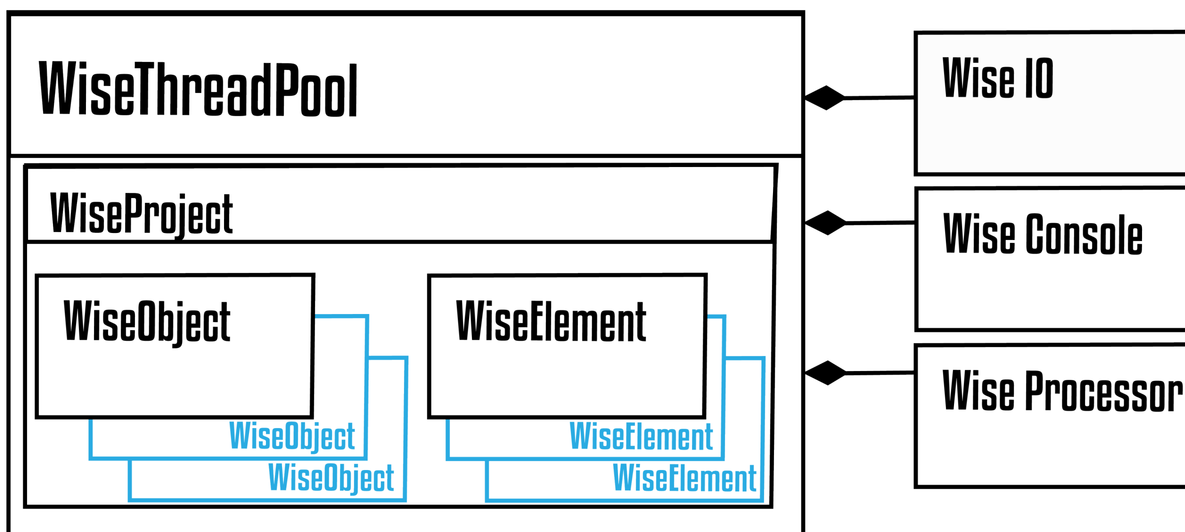


Figura 20 – Modelo de *Threads*. *WiseThreadPool*, responsável por orquestrar o funcionamento das demais *threads*, bem como os objetos contidos em um projeto *WiseProject*. *WiseIO*, *thread* responsável por processos de leitura e escrita. *WiseConsole*, *thread* responsável por interpretar os comandos de texto e traduzir-los na chamada de métodos. *WiseProcessor*, *thread* responsável por realizar o método iterativo de um objeto *WiseObject*

As tarefas se dividem em três principais grupos: tarefas auxiliares, tarefas de leitura/escrita e trabalhos de iteração. O principal trabalho auxiliar é interpretar os comandos recebidos pelo programa e então criar a instância de trabalho *WiseJob*. Sendo os grupos de tarefas mais custosos os de tarefas de escrita/leitura e iteração, eles foram divididos em *threads* específicas chamadas de *WiseIO* e *WiseProcessor*, respectivamente.

Os objetos distribuídos *threads* contidos na Figura 20 possuem um ciclo próprio, portanto executam tarefas assíncronas e precisam de tratamento adequado. O sistema de sinais e fendas disponibilizado pelas bibliotecas comuns do Qt permitem que as *threads* se comuniquem assincronamente através do envio de mensagens. Estas mensagens são chamadas de trabalhos *WiseJobs*, cada uma possuindo sua atividade relacionada e objeto relacionado. Uma vez criados, os trabalhos são alocados a sua respectiva categoria de *thread* passando por um balanceamento executado pelo gerenciador de threads *WiseThreadPool*. Enquanto o processamento é feito, todos os dados relativos ao processamento são bloqueados, isso previne a sobrescrita de dados quando há mais de uma *thread* trabalhando.

O gerenciador de *threads* *WiseThreadPool* é composto por *threads* que executam os trabalhos e por projetos *WiseProject* que criam trabalhos *WiseJobs*. Ao executar um comando de texto, a linha de entrada será recebida pelo gerenciador e o primeiro trabalho *WiseJob* será criado. Em seguida, será interpretado pelas *threads* do grupo de tarefas auxiliares *WiseConsole*, caso seja um trabalho de leitura, escrita ou iteração um trabalho associado será criado e enviado ao gerenciador de *threads*. Caso não seja um trabalho desses tipos ele será executado na *thread* atual e o trabalho finalizado. O console *WiseConsole* funciona como interpretador principal dos comandos e das mensagens, ao receber uma linha de texto este objeto irá analisar seu conteúdo e executar a ação correspondente.

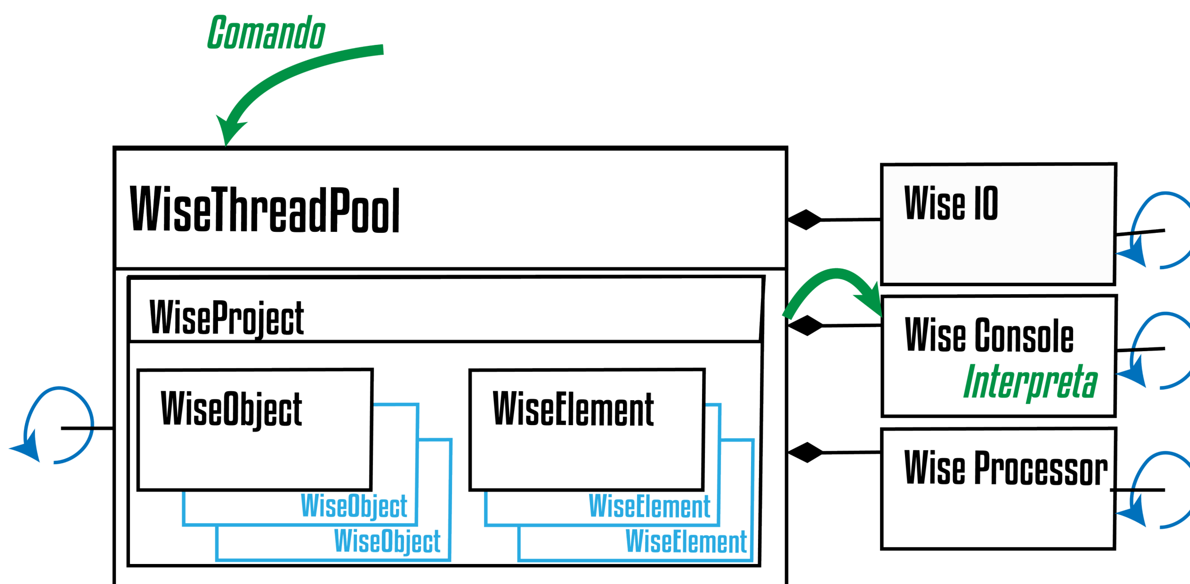


Figura 21 – Modelo de *Threads* ao receber uma linha de comando da interface de usuário.

Quando se trata de um comando de escrita ou leitura, a *thread* *WiseIO* é utilizada. Ao receber esse tipo de comando o console irá listar o trabalho no gerenciador de threads *WiseThreadPool*. O gerenciador irá balancear as requisições entre as *threads* e em seguida o gerenciador de *threads* irá aguardar uma mensagem indicando o final da execução do trabalho. Quando um objeto *WiseObject* é iterado um elemento é criado na estrutura *Freezer*, gerando uma requisição através de um trabalh. Portanto, a *thread* *WiseIO* é muito utilizada no ciclo de vida de um elemento, gerenciando o processo de armazenamento e reconstrução do mesmo. Esta estrutura é que efetivamente aquece e resfria os elementos.

Os trabalhos de iteração funcionam da mesma forma que as de leitura ou escrita, entretanto são enviadas a *threads* *WiseProcessor*. Estas *threads* não possuem lógicas muito complexas e são responsáveis apenas por gerenciar os objetos e executar os métodos requisitados. Por este motivo, os objetos e elementos possuem métodos abstratos e seguem esses conjuntos de regras para que uma estrutura que desconheça o seu funcionamento ou suas estruturas internas sejam capazes de executar métodos padrões.

Mesmo com a arquitetura de *threads* inclusa na ferramenta computacional ainda é

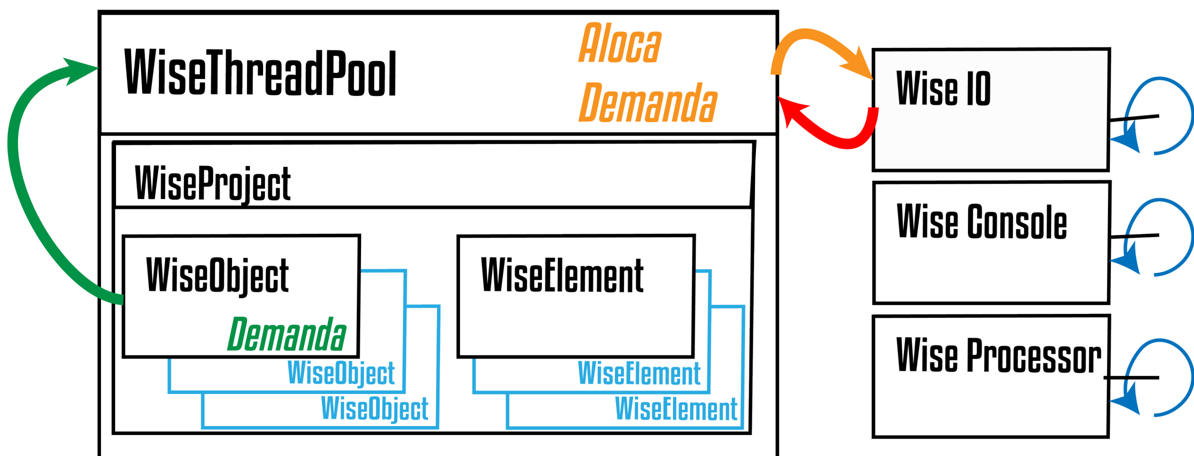


Figura 22 – Modelo de Threads ao receber um comando de escrita/leitura.

possível que ela seja executada em processadores com apenas um núcleo, neste caso mais *threads* não irão tornar a ferramenta mais rápida, pois apesar dos trabalhos ainda serem dividos em *threads* diferentes, elas serão executadas no mesmo núcleo de processamento. Além disto, existe um custo computacional pela comunicação entre threads, portanto com apenas um núcleo a arquitetura de classes divididas em *threads* se torna uma desvantagem. As *threads* têm um impacto maior nas atividades de escrita e leitura, isto porque o objeto está dividido em várias estruturas menores que são armazenadas e acessadas constantemente, principalmente no caso de uma animação gráfica.

Ainda dentro do gerenciador de *threads* existem quatro listas de espera:

- *Pre-Queue (Pre-Q)*: Lista de pré-seleção. Nesta estrutura, os trabalhos são agrupados por grupo de trabalho e ordenados por data de criação;

- *Queue (Q)*: Lista de espera para trabalhos que aguardam sua alocação em *threads* adequadas;

- *Running*: Lista de trabalhos que já foram enviados as suas respectivas *threads* e aguardam resposta;

- *Finished*: Lista de trabalhos que já terminaram o seu ciclo de execução.

No momento de criação os objetos são adicionados à lista *Pre-Q*. A cada ciclo da *thread WiseThreadPool*, os trabalhos da lista *Pre-Queue* são enviados a lista de espera *Queue*, os trabalhos de cada carga de trabalho são selecionados e enviados caso não haja pré-requisitos. A única exceção é a carga de trabalho utilizada pela ferramenta para armazenar e recuperar objetos rapidamente, para isto foi reservada um identificador da carga de trabalho.

Os trabalhos na lista de espera *Queue* são balanceados entre as *threads* disponíveis utilizando uma distribuição uniforme. Uma vez alocados, os trabalhos passam para a lista de trabalhos *Running*, que contém os trabalhos em execução.

Ao final da execução em uma *thread* separada, a estrutura armazena os trabalhos finalizados em uma lista *Finished*, que finalizaram seu processamento em uma *thread* concorrente e uma resposta foi recebida pelo gerenciador de *threads*.

3.2.8 OBJETO DA CLASSE *WiseThreadPool*

Para que a comunicação entre as *threads* *WiseThreads* seja feita de forma padronizada e possa ser estendida futuramente, ao receber alguma demanda, o gerenciador de *thread* *WiseThreadPool* irá criar um objeto do tipo *WiseJob*. Este objeto contém os seguinte parâmetros:

- *ID*: número de identificação único;
- *Workload*: número de carga de trabalho;
- *Arg*: cadeia de caracteres opcional, pode conter parâmetros para a execução do trabalho;
- *Type*: tipo de trabalho;
- *Status*: estado do trabalho;
- *Antecessors*: lista de trabalhos que antecedem este na ordem de chamada;
- *Pre-requisites*: lista de trabalhos que antecedem este na ordem de execução, ou seja, precisam ser finalizados antes;
- *Ponteiros*: o trabalho do tipo *WiseJob* pode ter um ponteiro para as estruturas da ferramenta computacional.

O número de identificação do trabalho é único e incremental, a carga de trabalho representa o grupo em que o trabalho irá executar. Ao executar a leitura de um arquivo de entrada, cada linha do arquivo irá ser interpretada como um comando e adicionada ao mesmo grupo de trabalho e vinculada ao último comando pela lista de antecessores *Antecessors*, o que garante que eles serão executados em ordem.

O tipo *Type* do trabalho é um identificador dado ao mesmo após sua interpretação inicial. Este identificador permite às *threads* determinar qual a composição do objeto e quais parâmetros para execução foram preenchidos, como ponteiros e linhas de entrada.

Os estados de um trabalho são ditados pela sua máquina de estados representada na Figura 23. Os estados desta máquina indicam em qual estrutura da *WiseThreadPool* o

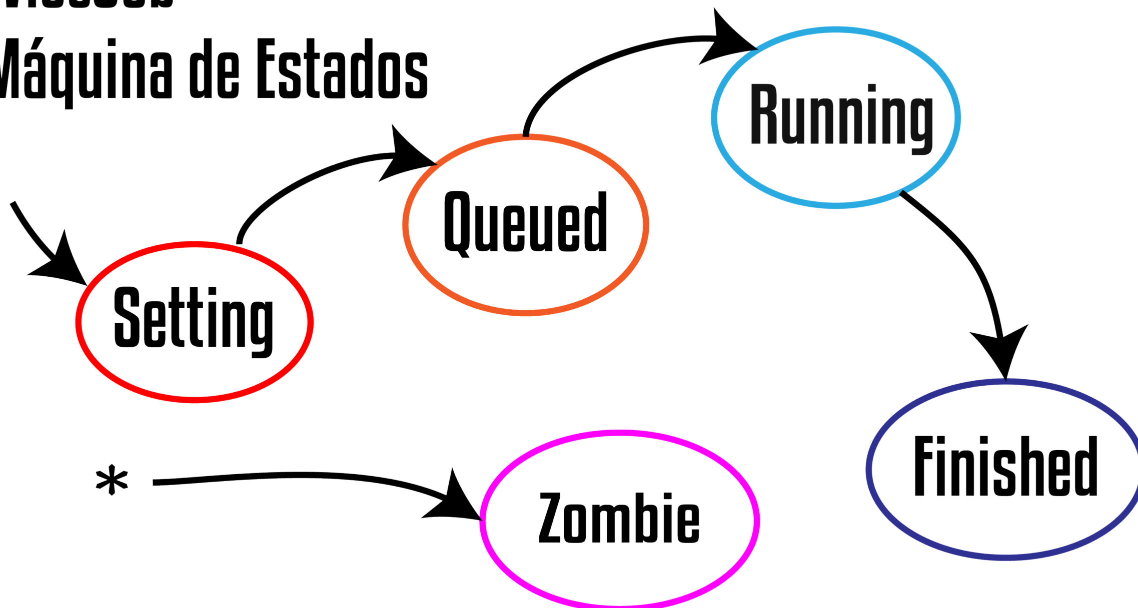


Figura 23 – Máquina de estados utilizadas por trabalhos *WiseJobs*.

trabalho está e se seu funcionamento é adequado. O estado *Setting* indica que o trabalho foi criado e está na lista de seleção *Pre-Q*; o estado *Queued* significa que o trabalho foi adicionado à fila de espera e aguarda execução; o estado *Running* indica que o trabalho está sendo executado; o estado *Finished* indica que o trabalho foi finalizado corretamente; e, o estado *Zombie* indica que o trabalho não foi finalizado corretamente.

No cerne da biblioteca *Qt* está a classe *QObject*, que possui o mecanismo de comunicação entre objetos de sinais e fendas. Para que um objeto possa utilizar desta interface de comunicação é necessário que ele herde as características da classe *QObject* por polimorfismo [7].

Os objetos de trabalho servem como mensagens de comunicação entre as *threads*. Através desta estrutura objetos do tipo *QObject* podem se comunicar e executar a requisição de trabalhos complexos. Isto foi feito para permitir que estruturas *QWidget*, que são elementos gráficos da interface de usuário e herdam da classe *QObject*, pudessem enviar mensagens diretamente ao gerenciador de *threads*. Os objetos *QWidget* são elementos gráficos como o quadro *OpenGL*, uma caixa de texto ou um botão. Todos os objetos desta classe enviam sinais a outro *QObject* ao interagir com o usuário, como o uso de comandos de teclado e/ou mouse. Através da comunicação dos objetos da classe *QObject* é possível que comandos na interface usuário acionem as funcionalidades das *threads* e dos objetos.

3.3 USABILIDADE DA FERRAMENTA COMPUTACIONAL

Nesta seção apresentam-se detalhes da interface escolhida para facilitar o uso da ferramenta computacional. A interface de usuário foi concebida para permitir o controle de todas as estruturas descritas na Seção 3.2 com todos os seus recursos gráficos extraídos. A interface se divide em dois ambientes: um console, que permite uma interação textual e acesso as mesmas funcionalidades de iteração; e uma janela com elementos gráficos capazes de exibir os resultados gráficos obtidos.

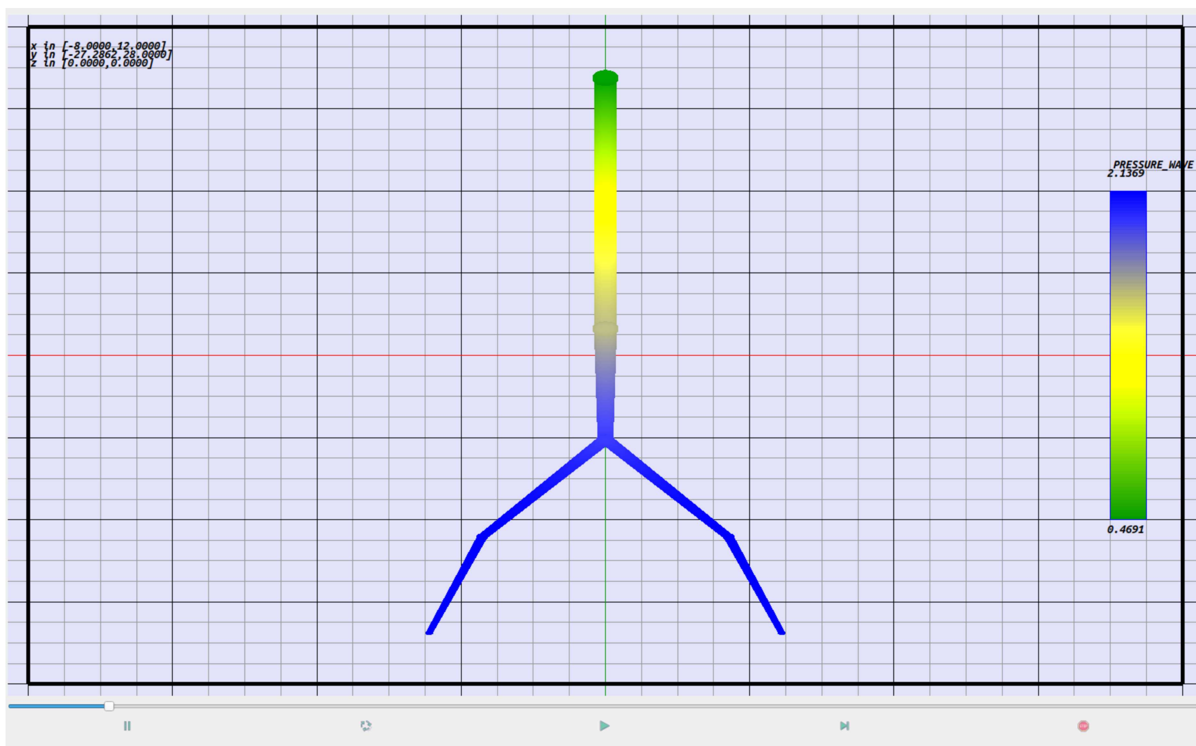


Figura 24 – Ilustração da interface gráfica da ferramenta computacional.

Ambas as interfaces foram construídas para manipular objetos e suas estruturas, por isto ambas possuem instâncias do gerenciador de *threads* descritos na Seção **3.2.7**. O console trata de um envio direto de mensagens para a *thread WiseConsole*, que é efetivamente o console. Os diferentes ambientes da ferramenta computacional são descritos. Cada ambiente de interface representa um projeto *Qt/C++* distinto e podem utilizar as mesmas classes.

3.3.1 CONSOLE

O ambiente de console da ferramenta computacional consiste em um projeto *Qt/C++* sem interface gráfica. O projeto foi intitulado ***I**terador **n**ão-**G**ráfico **U**niversal* (*InGU*), porque o ambiente apesar de não conter os elementos para a visualização dos elementos gráficos, é capaz de criar as estruturas gráficas para visualização futura. No console, as bibliotecas gráficas do OpenGL não foram incluídas no processo de compilação,

portanto ambientes sem recursos gráficos são capazes de compilar e executar a ferramenta computacional através deste ambiente.

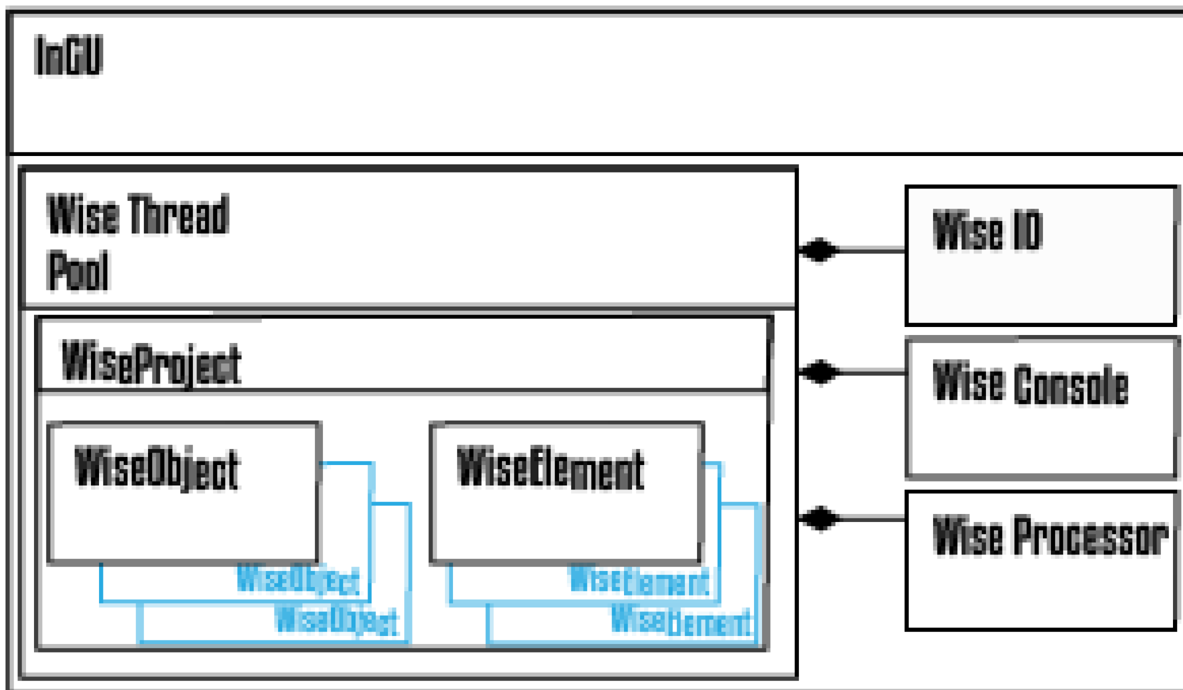


Figura 25 – Estrutura do projeto que compõe o ambiente computacional *InGU*.

No Apêndice A estão as sequências de instruções necessárias para compilar todos os projetos da ferramenta computacional. Ao ser compilado, um executável *InGU* contendo o projeto console é gerado. Ao ser executado, o console padrão do sistema operacional irá ser aberto com o cabeçalho da ferramenta e aguardará entrada de texto.

```
.    /=====================================================================/
.   /====/       IGU (Iterador Gráfico Universal)       /==================/
.  /====/        (or, Universal Graphic Iterator)      /==================/
. /===================================================================/
. /=1.1===============================================================/
. /===================================================================/
. segunda-feira 09/08/2021 19:19:08 313
. /===================================================================/
. [WISE CONSOLE] LOAD invoked
. <p1:WISE_PROJECT> 'WISE_ID [0] (NAME: p1) WISE PROJECT CREATED'
. <el1:WISE_ELEMENT> 'WISE_ID [0] (NAME: el1) WISE ELEMENT CREATED'
. <el1:WISE_OBJECT> 'WISE_ID [0] (NAME: obj1) WISE OBJECT CREATED'
. <el1:WISE_ELEMENT> 'WISE_ID [1] (NAME: el1) WISE ELEMENT CREATED'
. <el1:WISE_ELEMENT> 'WISE_ID [2] (NAME: el1) WISE ELEMENT CREATED'
. <el1:WISE_ELEMENT> 'WISE_ID [3] (NAME: el1) WISE ELEMENT CREATED'
. <el1:WISE_ELEMENT> 'WISE_ID [4] (NAME: el1) WISE ELEMENT CREATED'
. <el1:WISE_ELEMENT> 'WISE_ID [5] (NAME: el1) WISE ELEMENT CREATED'
. <el1:WISE_ELEMENT> 'WISE_ID [6] (NAME: el1) WISE ELEMENT CREATED'
. <el1:WISE_ELEMENT> 'WISE_ID [7] (NAME: el1) WISE ELEMENT CREATED'
. <el1:WISE_ELEMENT> 'WISE_ID [8] (NAME: el1) WISE ELEMENT CREATED'
. <el1:WISE_ELEMENT> 'WISE_ID [9] (NAME: el1) WISE ELEMENT CREATED'
. <el1:WISE_ELEMENT> 'WISE_ID [10] (NAME: el1) WISE ELEMENT CREATED'
. <el1:WISE_ELEMENT> 'WISE_ID [11] (NAME: el1) WISE ELEMENT CREATED'
. <el1:WISE_ELEMENT> 'WISE_ID [12] (NAME: el1) WISE ELEMENT CREATED'
```

Figura 26 – Captura de tela com a execução do ambiente computacional *InGU* em um console.

Para executar um comando, basta inserir uma linha de texto e apertar a tecla *Enter*. Ao capturar a linha de texto, o programa de console irá na verdade redirecionar o

comando para a estrutura *WiseThreadPool*, que por sua vez irá alocar uma *thread* do tipo *WiseConsole* para interpretar a mensagem. Este comportamento é o mesmo apresentado na Seção **3.2.7**. Uma lista de comandos foi disponibilizada e está acessível através do comando *help*. Ao enviar este comando para o console, uma lista com todos as possíveis entradas será exibida. Nas próximas seções, estes comandos, suas entradas, seu escopo e *thread* responsável são descritos. Esta *thread* irá executar a tarefa.

- **Ajuda** (Tabela 2): o comando para ajuda (*help*) é o primeiro comando da interface e foi feito para listar todas as entradas possíveis do programa. Ao receber este comando a thread *WiseConsole* envia o texto pré-definido com todos os comandos.

Linha de Comando	help	
Escopo	nenhum	
Thread Responsável	*WiseConsole*	
Entrada	<vazio>	Nenhuma

Tabela 2 – Descrição do comando para ajuda.

Ao executar o comando, o usuário receberá uma lista de comandos divididos em escopos específicos, como visto na Figura 27. Os escopos foram criados para agrupar comandos por área de atuação, comandos auxiliares, como o comando de ajuda, são os comandos que não alteram as estruturas e exibem informações auxiliares ao usuário.

- **Ler arquivo de entrada** (Tabela 3): o comando para ler arquivo irá receber o endereço de um arquivo local, este arquivo deve conter um comando por linha, um exemplo pode ser encontrado no Apêndice B. Para este tipo de comando, deve-se enviar um *WiseJob* contendo a linha de comando e qual a sequência de trabalhos que será executada. Por se tratar de uma leitura de arquivo de comando, a *thread WiseIO* será responsável por ler o arquivo e criar os trabalhos com os comandos subsequentes. Estes novos comandos irão passar novamente pela interpretação de uma *thread* do tipo *WiseConsole* e em seguida executados.

- **Bateria de testes** (Tabela 4): o ciclo principal de testes da ferramenta se baseia em verificar a consistência de todas as fábricas e do fluxo principal da ferramenta computacional, exceto detalhes gráficos e do processo de iteração. Ao executar a bateria de testes todos os elementos *WiseElement* e objetos *WiseObject* disponíveis

```
.    /================================================================================/
.   /====/                    AJUDA                     /=============================/
. /==============================================================================/
. |help..........................................................................|
. |        Lista todos os comandos da plataforma.                                |
. |SCOPE READ....................................................................|
. |read cmds <file>..............................................................|
. |        <file> Caminho para arquivo de entrada (XML ou VTK).                  |
. |        Lê todas as linhas do arquivo e executa a sequência de comandos.      |
. |SCOPE TEST....................................................................|
. |test..........................................................................|
. |        Realiza todas as baterias de teste.                                   |
. |test.list.....................................................................|
. |        Realiza todas as baterias de teste.                                   |
. |test.<test_id>................................................................|
. |        Realiza um teste específico.                                          |
. |test file <test_id> <file1> <file2>...........................................|
. |        Testa a equidadade de dois arquivos.                                  |
. |..............................................................................|
. |SCOPE PROJECT.................................................................|
. |..............................................................................|
. |project.......................................................................|
. |project create <project_name>                                                 |
. |        <project_name> Nome do projeto à ser criado.                          |
. |       Cria projeto.                                                          |
. |project use <project_name>                                                    |
. |        <project_name> Nome do projeto à ser selecionado.                     |
. |       Seleciona projeto para execução dos próximos comandos.                 |
. |project list                                                                  |
. |       Lista os projetos disponíveis.                                         |
. |project print                                                                 |
. |       Imprime o projeto selecionado.                                         |
. |project delete                                                                |
. |       Deleta o projeto selecionado.                                          |
. |project save <filename>                                                       |
. |       <filename> Arquivo de saída aonde o projeto será salvo.                |
. |       Salva projeto em um arquivo XML.                                       |
. |project load <filename>                                                       |
. |       <filename> Arquivo de entrada de onde o projeto será reconstruido.     |
. |       Carrega um arquivo XML.                                                |
. |..............................................................................|
. |SCOPE ELEMENT.................................................................|
. |..............................................................................|
. |element.......................................................................|
. |element create <type> <example> <name> [ARGS]                                 |
. |       <type> Tipo de elemento à ser criado.                                  |
. |       <example> Nome do exemplo à ser utilizado.                             |
. |       <name> Nome do elemento à ser criado.                                  |
. |       [ARGS] Argumentos extras do método de criação específico.              |
. |       Cria elemento inteligente à partir de um exemplo da fábrica.           |
. |element clone <element_name> <name>                                           |
. |       <element_name> Nome do elemento à ser clonado.                         |
. |       <name> Nome do elemento clone.                                         |
. |       Clona um elemento inteligente.                                         |
. |element list                                                                  |
. |       Lista os elementos do projeto selecionado.                             |
```

Figura 27 – Captura de tela com a execução do comando de ajudo no ambiente computacional *InGU* em um console.

Tabela 3 – Descrição do comando para ler arquivo de comando.

Linha de Comando	read cmds <file>	
Escopo	READ	
Thread Responsável	*WiseIO*	
Entrada	<file>	Caminho para arquivo contendo sequência de comandos

serão testados individualmente. Para isto, uma fábrica de projeto *WiseProjectFactory* irá ser encarregada de criar todos os elementos e objetos disponíveis, sendo executados pela própria ferramenta em tempo de execução.

Estes testes foram utilizados principalmente no momento do desenvolvimento da ferramenta e garantem que as fábricas de elementos e objetos estão funcionando corretamente. Futuramente, caso novas estruturas sejam incluídas, estes testes garantem que todas as classes do projeto foram elaboradas corretamente, com isso todas as estruturas que são carregadas pela ferramenta não perdem informação ao serem armazenadas e recuperadas, processo recorrente na ferramenta computacional. Os comandos deste tipo (Tabela 4) são de responsabilidade da *thread WiseConsole*, que irá criar e executar trabalhos *WiseJob* correspondente a bateria de testes. Ao finalizar estes trabalhos, os resultados são impressos na tela e o trabalho principal finalizado.

Tabela 4 – Descrição do comando para bateria de testes.

Linha de Comando	test	
Escopo	TEST	
Thread Responsável	*WiseConsole*	
Entrada	<vazio>	Nenhuma

- **Lista de testes** (Tabela 5): o comando para listar testes irá enumerar todos os testes possíveis de serem executados pela ferramenta computacional. Assim como o comando de ajuda, este comando irá imprimir no console todos os resultados encontrados.

Tabela 5 – Descrição do comando para listar testes.

Linha de Comando	test list	
Escopo	TEST	
Thread Responsável	*WiseConsole*	
Entrada	<vazio>	Nenhuma

- **Caso de teste** (Tabela 6): Como mencionado anteriormente, os testes são enumerados. Baseando-se nessa lista, é possível selecionar um teste pelo seu número correspondente utilizando o comando para executar caso de teste. Os casos de testes

irão gerar uma sequência de comandos a serem interpretados pelo *WiseConsole*. Os testes são compostos de comandos que irão criar e deletar estruturas que são finalizados na própria *thread WiseConsole*. Entretanto estas estruturas são enviadas para um arquivo externo e então lidas. Estes subcomandos irão ser executados pela *thread WiseIO*. O caso de teste só é dado como concluído quando todos os subcomandos terminam sua execução.

Tabela 6 – Descrição do comando para executar caso de teste.

Linha de Comando	test <test_id>	
Escopo	TEST	
Thread Responsável	*WiseConsole*	
Entrada	<test_id>	Número do teste a ser executado

- **Testar igualdade de arquivos** (Tabela 7): outro comando de teste disponibilizado pela ferramenta computacional é o teste de igualdade de arquivos. Este comando utiliza uma chave numérica *test_id* para salvar o resultado do teste. Tanto o resultado quanto a quantidade de testes executados com a mesma chave são salvos. Por se tratar da leitura de dois arquivos diferentes esse comando é executado por uma *thread* do tipo *WiseIO*.

Tabela 7 – Descrição do comando para testar a igualdade de dois arquivos.

Linha de Comando	test file <test_id> <file1> <file2>	
Escopo	TEST	
Thread Responsável	*WiseIO*	
Entrada	<test_id>	Chave numérica para armazenar resultado
	<file1>	Caminho para o primeiro arquivo
	<file2>	Caminho para o segundo arquivo

- **Criar projeto** (Tabela 8): comando utilizado para criar um novo projeto, o qual recebe como entrada o nome do projeto. Este comando irá utilizar a fábrica de projetos para criar a estrutura vazia do projeto. Ao receber um comando deste a *thread WiseConsole* irá retornar um projeto em branco para ser acoplado à *WiseThreadPool*. Isto é feito para que outras *threads* possam receber o mesmo projeto de forma separada.

Tabela 8 – Descrição do comando para criar projetos.

Linha de Comando	project create <name>	
Escopo	PROJECT	
Thread Responsável	WiseConsole	
Entrada	<name>	Nome do projeto a ser criado

- **Usar projeto** (Tabela 9): para que a maioria dos comandos funcione é necessário que um projeto esteja selecionado, pois é a estrutura do projeto que disponibiliza elementos e objetos. Uma vez que projetos tenham sido criados, eles podem ser selecionados utilizando o comando para usar projetos.

Tabela 9 – Descrição do comando para selecionar projetos.

Linha de Comando	project use <project_name>	
Escopo	PROJECT	
Thread Responsável	*WiseConsole*	
Entrada	<project_name>	Nome do projeto a ser selecionado

- **Listar projetos** (Tabela 10): é possível verificar todos os projetos do ambiente utilizando o comando para listar projetos. Todos os projetos carregados no momento da execução são exibidos na listagem.

Tabela 10 – Descrição do comando para listar projetos.

Linha de Comando	project list	
Escopo	PROJECT	
Thread Responsável	*WiseConsole*	
Entrada	<vazio>	Nenhuma

- **Imprimir projeto** (Tabela 11): o comando para imprimir projetos irá exibir no console a representação do arquivo *XML* do projeto selecionado. Caso o projeto possua elementos e objetos eles também são impressos na estrutura de saída.

Tabela 11 – Descrição do comando imprimir projetos.

Linha de Comando	project print	
Escopo	PROJECT	
Thread Responsável	WiseConsole	
Entrada	<vazio>	Nenhuma

- **Excluir projeto** (Tabela 12): com um projeto selecionado, o comando para exclusão de elementos irá excluir o elemento do projeto recebendo o nome do elemento como parâmetro de entrada. Este comando será interpretado pela *thread WiseConsole*, que notifica o gerenciador de *threads WiseThreadPool*. O gerenciador só irá aguardar até que o projeto não possua trabalhos em execução para que posso excluí-lo.

Tabela 12 – Descrição do comando para excluir projetos.

Linha de Comando	project delete	
Escopo	PROJECT	
Thread Responsável	*WiseConsole*	
Entrada	<vazio>	Nenhuma

- **Salvar projeto** (Tabela 13): o comando para salvar projetos irá extrair do projeto selecionado sua representação em um arquivo *XML*, assim como no processo de impressão do projeto. Este arquivo será impresso em um arquivo externo e por isto é executado pela thread *WiseIO*.

Tabela 13 – Descrição do comando para salvar projetos.

Linha de Comando	project save <file_name>	
Escopo	PROJECT	
Thread Responsável	*WiseIO*	
Entrada	<file_name>	Caminho do arquivo de saída

- **Carregar projeto** (Tabela 14): o comando para carregar projetos irá extrair de um arquivo de entrada no formato *XML* a estrutura de um projeto. A ferramenta computacional irá utilizar os arquivos de entrada diretamente nas estruturas de fábrica relacionadas. Caso o projeto possua elementos e objetos eles também serão reconstruídos no projeto reconstruído.

Tabela 14 – Descrição do comando para carregar projetos.

Linha de Comando	project load <filename>	
Escopo	PROJECT	
Thread Responsável	*WiseIO*	
Entrada	<filename>	Caminho do arquivo de entrada

- **Criar elemento** (Tabela 15): as fábricas de elementos descritas na Seção **3.2.2** são acessadas pelo comando de criar elementos. Este comando recebe como entrada o tipo de elemento, o nome e o exemplo a ser utilizado. Cada fábrica de elementos possui uma lista de exemplos disponíveis, que é visualizada através do comando para listar exemplos de uma fábrica de elementos.

Tabela 15 – Descrição do comando para criar.

Linha de Comando	element create <type> <example> <name> [ARGS]	
Escopo	ELEMENT	
Thread Responsável	*WiseConsole*	
Entrada	<type>	Tipo de elemento à ser criado, seleciona a fábrica de elemento à ser utilizada.
	<example>	Caminho para o primeiro arquivo
	<name>	Caminho para o segundo arquivo
	[ARGS]	Individualmente, as fábricas podem receber parâmetros para a criação de elementos

- **Clonar elemento** (Tabela 16): assim como o comando para criação de elementos descrito na Tabela 15, ao comando para clonagem de elementos também irá acessar as fábricas de elementos. Ao clonar um elemento, o seu tipo é selecionado juntamente com a fábrica correspondente. Após serem selecionados a fábrica receberá como parâmetro o elemento e o nome do novo elemento a ser criado. Portanto, o comando de clonagem de elementos poderá ser utilizando com a entrada do nome do elemento a ser clonado e o nome do novo elemento.

- **Listar elementos** (Tabela 17): com um projeto já selecionado, o comando para listar elementos pode ser utilizado para que o console imprima uma lista com o nome de todos os elementos presentes no projeto.

Tabela 16 – Descrição do comando para clonar elementos.

Linha de Comando	element clone <element_name> <name>	
Escopo	ELEMENT	
Thread Responsável	*WiseConsole*	
Entrada	<element_name> <name>	Nome do elemento a ser clonado. Nome do novo elemento.

Tabela 17 – Descrição do comando para listar elementos.

Linha de Comando	element list	
Escopo	ELEMENT	
Thread Responsável	*WiseConsole*	
Entrada	<vazio>	Nenhuma

- **Imprimir elemento** (Tabela 18): o comando para imprimir elementos irá extrair do projeto selecionado a representação em um arquivo *XML*. A informação textual deste arquivo será impressa como resultado no console. Para que o comando funcione é necessário que um projeto esteja selecionado e que o nome do elemento a ser impresso seja informado.

Tabela 18 – Descrição do comando para imprimir elementos.

Linha de Comando	element print <element_name>	
Escopo	ELEMENT	
Thread Responsável	*WiseConsole*	
Entrada	<element_name>	Nome do elemento à ser impresso

- **Excluir elemento** (Tabela 19): o comando para excluir elementos irá remover os dados do elemento da memória.

Tabela 19 – Descrição do comando para excluir elementos.

Linha de Comando	element delete <element_name>	
Escopo	ELEMENT	
Thread Responsável	*WiseConsole*	
Entrada	<element_name>	Nome do elemento a ser excluído.

- **Salvar elemento** (Tabela 20): os elementos podem ser exportados em dois formatos, um arquivo *XML* e um arquivo *VTK*. Ambos futuramente podem ser utilizados na reconstrução do elemento. Como entrada o comando para salvar elementos receberá o nome do elemento, o tipo de arquivo a ser escrito e o caminho onde ele deve ser salvo. A saída do comando será o arquivo determinado, que a partir dele é possível reconstruir o elemento completo.

Tabela 20 – Descrição do comando para salvar elementos.

Linha de Comando	element save <element_name> <save_type> <filename>	
Escopo	ELEMENT	
Thread Responsável	*WiseIO*	
Entrada	<element_name>	Nome do elemento inteligente a ser salvo.
	<save_type>	Tipo de arquivo a ser exportando, podendo ser *XML* ou *VTK*
	<filename>	Caminho para o arquivo a ser exportado

- **Carregar elemento** (Tabela 21): o comando para carregar elemento irá extrair de um arquivo de entrada no formato *XML* ou *VTK* a estrutura de um elemento. A ferramenta computacional irá utilizar os arquivos de entrada diretamente na fábrica de elemento relacionada. Caso seja um arquivo *XML* não será necessário informar o tipo de elemento a ser construído. Caso contrário, o tipo e o nome são recebidos como parâmetros de entrada.

Tabela 21 – Descrição do comando para carregar elementos.

Linha de Comando	element load <load_type> element load <**VTK**> <filename> <type> <name> element load <**XML**> <filename>	
Escopo	ELEMENT	
Thread Responsável	*WiseIO*	
Entrada	<filename>	Caminho para o arquivo de entrada
	<load_type>	Tipo de arquivo a ser carregado, podendo ser *XML* ou *VTK*
	<type>	Tipo de elemento a ser criado
	<name>	Nome do elemento a ser criado

- **Listar exportações do elemento** (Tabela 22): os elementos podem ser exportados para outros formatos de arquivo, como uma imagem *PNG*, imagem *JPG* ou texto

TXT. Cada tipo de elemento possui uma lista de exportações disponíveis. Assim, um comando foi inserido para acessar a lista de exportações de um certo objeto.

Tabela 22 – Descrição do comando para listar exportações de um elemento.

Linha de Comando	element export_list <element_name>	
Escopo	ELEMENT	
Thread Responsável	*WiseConsole*	
Entrada	<element_name>	Nome do elemento à analisado

- **Exportar elemento** (Tabela 23): a estrutura abstrata dos elementos requer que cada um deles possua o método de exportação, este método pode ser utilizado através do comando de exportar elementos. Salienta-se que os métodos de exportação podem receber parâmetros diferentes. A principal exportação utilizada foi disponibilizada na classe *WiseGraphic*, que propicia a exportação dos gráficos em arquivos de imagem. Desta forma, possibilitou-se a criação de imagens em um ambiente que não possui interface gráfica.

Tabela 23 – Descrição do comando para exportar elementos.

Linha de Comando	element export <element_name> <export_name> [ARGS]	
Escopo	ELEMENT	
Thread Responsável	*WiseIO*	
Entrada	<element_name>	Nome do elemento a ser exportado
	<export_name>	Tipo de exportação a ser realizada
	[ARGS]	Os elementos podem receber parâmetros para sua exportação

- **Escalar elemento** (Tabela 24): os dados inseridos em um elemento podem ser escalados através do comando para escalar elemento. Os valores contidos em pontos, linhas, células e campos podem ser multiplicados por um escalar s. Isso permite que modelos geométricos com parâmetros em unidades fora do padrão possam ser escalados. Utilizando o nome do elemento, o tipo e o nome do parâmetro a ser escalado e o valor do escalar s, o comando para escalar elementos permite ao usuário dimensionar cada parâmetro individualmente.

Tabela 24 – Descrição do comando para escalar elementos.

Linha de Comando	element scale <element_name> <cell_type> <field> <scale>	
Escopo	ELEMENT	
Thread Responsável	*WiseConsole*	
Entrada	<type>	Tipo de elemento à ser criado, seleciona a fábrica de elemento à ser utilizada
	<element_name>	Nome do elemento à ser exportado
	<cell_type>	Tipo de parâmetro à ser escalado (Ponto,Célula,Linha e Campos)
	<field>	Nome do campo à ser escalado
	<scale>	Escalar f utilizado ao escalar elemento

- **Definir parâmetro de elemento** (Tabela 25): os dados inseridos em um elemento podem ser alterados e receber um valor através do comando para definir parâmetro de elemento. As informações contidas em pontos, linhas, células e campos podem ser substituídas por um valor de entrada. Isso permite modelos geométricos tenham seus parâmetros editados. Para alterar o parâmetro de um elemento é necessário adicionar a linha de comando o nome do elemento, o tipo e nome do parâmetro a ser alterado, a posição do valor no vetor do parâmetro e o valor a ser inserido.

Tabela 25 – Descrição do comando para definir parâmetros de elementos.

Linha de Comando	element set_field <element_name> <cell_type> <field_name> <id> <data>	
Escopo	ELEMENT	
Thread Responsável	WiseConsole	
Entrada	<type>	Tipo de elemento a ser criado, seleciona a fábrica de elemento a ser utilizada.
	<element_name>	Nome do elemento a ser definido
	<cell_type>	Tipo de parâmetro a ser escalado (Ponto,Célula,Linha ou Campo)
	<field_name>	Nome do campo a ser escalado
	<id>	Posição do valor no vetor do parâmetro, 0 caso valor único
	<data>	Valor à ser inserido como parâmetro do elemento

- **Definir todos os parâmetros de elemento** (Tabela 26): O comando para definição de todos os valores pertencentes a um parâmetro auxilia no manuseio de parâmetros que possuem múltiplos valores. O comando define todas as posições encontradas

no vetor do parâmetro e as altera para o valor de entrada. Para alterar todos os valores de um determinado parâmetro de um elemento é necessário adicionar a linha de comando o nome do elemento, o tipo e nome do parâmetro a ser alterado e o valor a ser inserido.

Tabela 26 – Descrição do comando para definir todos os parâmetros de um campo pertencente a um elemento.

Linha de Comando	element set_all_field <element_name> <cell_type> <field_name> <data>	
Escopo	ELEMENT	
Thread Responsável	*WiseConsole*	
Entrada	<type>	Tipo de elemento a ser criado, seleciona a fábrica de elemento a ser utilizada
	<element_name>	Nome do elemento à ser exportado
	<cell_type>	Tipo de parâmetro à ser escalado (POINT,LINE,CELL ou FIELD)
	<field_name>	Nome do campo à ser escalado
	<data>	Valor à ser inserido como parâmetro do elemento

- **Listar fábricas de elemento** (Tabela 27): o comando para listar todas as fábricas de elemento *WiseElementFactory* foi criado. O resultado impresso ao executar este comando é a lista de fábricas disponíveis na classe *WiseElementFactories* acoplada na ferramenta computacional.

  Consequentemente, os resultados impressos representam os tipos de elementos suportados pela ferramenta computacional.

Tabela 27 – Descrição do comando para listar fábricas de elemento.

Linha de Comando	element factories list	
Escopo	ELEMENT	
Thread Responsável	*WiseConsole*	
Entrada	<vazio>	Nenhuma

- **Listar exemplos disponíveis de elemento** (Tabela 28): para cada tipo de elemento, ou fábrica listada pelo comando descrito na Tabela 27, existirá uma lista de exemplos disponíveis. Exemplos são elementos pré-definidos que podem ser criados a partir do comando de criar elementos descrito na Tabela 15. Para listar os exemplos disponíveis o comando recebe como parâmetro de entrada a fábrica a ser analisada.

Tabela 28 – Descrição do comando para listar exemplos contidos em determinada fábrica de elemento.

Linha de Comando	element factories examples <factory>	
Escopo	ELEMENT	
Thread Responsável	*WiseConsole*	
Entrada	<factory>	Nome da fábrica de elementos analisada.

- **Criar objeto** (Tabela 29): assim como o comando de criação de elementos, o comando de criação de objetos irá acessar a fábrica de elementos *WiseElementFactory*. É possível criar objetos de duas formas: (1) utilizando um elemento; (2) utilizando os exemplos disponibilizados pela fábrica de elementos. Assim como descrito na Seção 3.2, ao criar um objeto, um elemento é adicionado na estrutura do *Forno*, enquanto um Clone é acoplado ao *Freezer*;

Tabela 29 – Descrição do comando para criar objetos.

Linha de Comando	object create <object_name> <element_name> object create <type> <example> <name> <element_name> [ARGS]	
Escopo	OBJECT	
Thread Responsável	WiseConsole	
Entrada	<object_name>	Nome do objeto a ser criado.
	<element_name>	Nome do elemento a ser utilizado na criação ou do elemento a ser criado a partir do exemplo
	<type>	Tipo de elemento a ser criado
	<name>	Nome do exemplo de elemento a ser criado
	[ARGS]	Individualmente, as fábricas podem receber parâmetros para a criação de elementos

- **Clonar objeto** (Tabela 30): Assim como o elemento, o objeto pode ser clonado em uma fábrica própria. Através do comando de clonar objetos estes métodos podem ser acessados. Basta inserir o nome do objeto a ser clonado. A fábrica correta é acessada verificando o tipo do objeto a ser clonado.

Tabela 30 – Descrição do comando para clonar objetos.

Linha de Comando	object clone <object_name> <name>	
Escopo	OBJECT	
Thread Responsável	WiseConsole	
Entrada	<object_name>	Nome do objeto a ser clonado
	<name>	Nomo do novo elemento

- **Listar objetos** (Tabela 31): com um projeto já selecionado, o comando de listar objetos pode ser utilizado para que o console imprima uma lista com o nome de todos os objetos presentes no projeto.

- **Imprimir objeto** (Tabela 32): o comando de imprimir elementos irá extrair do projeto selecionado a representação em um arquivo *XML*. A informação textual deste arquivo será impressa como resultado no console. Para que o comando funcione é

Tabela 31 – Descrição do comando para listar elementos.

Linha de Comando	object list	
Escopo	OBJECT	
Thread Responsável	*WiseConsole*	
Entrada	<vazio>	Nenhuma

necessário que um projeto esteja selecionado e que o nome do objeto a ser impresso seja informado.

Dentro deste arquivo *XML* estão todas as estruturas descritas na Seção **3.2.3**, ou seja, os elementos que compõe as estruturas do *Forno* e *Freezer* de detalhes do objeto, bem como seu tipo e fábricas utilizadas.

Tabela 32 – Descrição do comando para imprimir objetos.

Linha de Comando	object print <object_name>	
Escopo	OBJECT	
Thread Responsável	*WiseConsole*	
Entrada	<object_name>	Nome do objeto a ser impresso

- **Excluir objeto** (Tabela 33): com um projeto selecionado, o comando de exclusão de objetos irá excluir o objeto do projeto recebendo o nome do objeto como parâmetro de entrada.

Tabela 33 – Descrição do comando para excluir objetos inteligentes.

Linha de Comando	object delete <object_name>	
Escopo	OBJECT	
Thread Responsável	*WiseConsole*	
Entrada	<object_name>	Nome do objeto a ser excluído

- **Salvar objeto** (Tabela 34): O comando de salvar objetos possibilita que a estrutura complexa de um objeto, bem como seus componentes, seja arquivada em um arquivo *XML*. Os objetos salvos podem ser recuperados em seguida. Como entrada, o comando de salvar objetos irá receber o nome do objeto a ser salvo e o caminho para o arquivo de saída.

Tabela 34 – Descrição do comando para salvar objetos.

Linha de Comando	object save <object_name> <filename>	
Escopo	OBJECT	
Thread Responsável	*WiseIO*	
Entrada	<object_name>	Nome do objeto a ser salvo
	<filename>	Caminho para o arquivo de saída

- **Carregar objeto** (Tabela 35): o comando de carregar objetos irá enviar um arquivo de entrada a fábrica de objetos *WiseObjectFactory*, que por sua vez irá reconstruir cada componente do objeto com sua fábrica adequada. Isto significa que a coleção de elementos e objetos gráficos irão reconstruir cada objeto. O comando recebe como entrada o nome do arquivo apenas. No Apêndice C há um exemplo de arquivo *XML* contendo um objeto válido.

Tabela 35 – Descrição do comando para carregar objetos.

Linha de Comando	object load <filename>	
Escopo	OBJECT	
Thread Responsável	*WiseIO*	
Entrada	<filename>	Caminho para o arquivo de entrada

- **Exportar objeto** (Tabela 36): o comando para exportar objetos funciona da mesma forma que o comando de exportar elementos, isto porque o comando irá exportar o elemento contido na estrutura do *Forno*. Portanto, os comandos para listar exportações e o para exportar elemento foram adicionados ao objeto.

Tabela 36 – Descrição dos comandos de exportação elementos contidos no *Forno* de objetos.

Linha de Comando	object export_list <object_name> object export <object_name> <export_type> [ARGS]	
Escopo	OBJECT	
Thread Responsável	*WiseConsole*	
Entrada	<object_name>	Nome do objeto a ser analisado ou exportado
	<export_type>	Tipo de exportação a ser realizada
	[ARGS]	Argumentos de entrada, específicos de cada exportação

- **Definir parâmetros de objeto** (Tabela 37): assim como o comando de exportar objeto, este comando para definir parâmetros atua sobre o elemento contido na estrutura do *Forno*. Ao utilizar comandos de definição de parâmetros em um objeto, o elemento contido na estrutura do *Forno* é afetado. Portanto, caso algum ajuste seja feito no modelo geométrico, as iterações passadas não sofrem mudanças, enquanto qualquer iteração nova recebe a nova informação definida.

Tabela 37 – Descrição do comando para definir de parâmetros do objeto.

Linha de Comando	object set_field <object_name> <cell_type> <field_name> <id> <data> object set_all_field <object_name> <cell_type> <field_name> <data>	
Escopo	OBJECT	
Thread Responsável	*WiseConsole*	
Entrada	<object_name>	Nome do elemento a ser definido
	<cell_type>	Tipo de parâmetro a ser escalado (Ponto,célula,linha ou campo)
	<field_name>	Nome do campo a ser escalado
	<id>	Posição do valor no vetor do parâmetro, 0 caso valor único
	<data>	Valor a ser inserido como parâmetro do elemento

- **Definir objeto** (Tabela 38): o comando para definir um objeto tem ligação direta com o seu processo de iteração. Como mencionado na Seção 3.2, o objeto é criado por padrão no estado *Ready*. Tendo sido corretamente criado e acoplado às fábricas de iteração e, opcionalmente, gráficas, ele pode ser definido mudando para o estado *Set*. O comando para definir um objeto recebe apenas o nome do objeto e retorna uma mensagem de conclusão.

Tabela 38 – Descrição do comando para definir objeto.

Linha de Comando	object set <object_name>	
Escopo	OBJECT	
Thread Responsável	*WiseConsole*	
Entrada	<object_name>	Nome do objeto a ser setado

- **Iterar objeto** (Tabela 39): o comando para iterar objetos executa o ciclo de iteração de um objeto que tenha sido corretamente setado. Os objetos no estado *Set* podem ter seus parâmetros alterados e corretamente configurados, em seguida passam pelo ciclo de iteração. Ao terminarem o ciclo, os objetos mudam para o estado *Go*.

Tabela 39 – Descrição do comando para iterar objetos.

Linha de Comando	object go <object_name>	
Escopo	OBJECT	
Thread Responsável	WiseProcessor	
Entrada	<object_name>	Nome do objeto a ser iterado

- **Listar fábricas de iteração de objeto** (Tabela 40): cada tipo de objeto possuirá uma lista de fábricas de iteração *WiseIterationFactory* que podem ser acopladas a ele. O comando de listar fábricas de iteração irá receber o nome do objeto inteligente a ser analisado, as fábricas de iteração que forem compatíveis com o objeto são escritas no console.

Tabela 40 – Descrição do comando para listar fábricas de iteração.

Linha de Comando	object iteration_factories list <object_name>	
Escopo	OBJECT	
Thread Responsável	*WiseConsole*	
Entrada	<object_name>	Nome do objeto a ser analisado

- **Definir fábrica de iteração de objeto** (Tabela 41): antes que um objeto possa passar para o estado *Set* ele precisar em sua composição uma fábrica de iteração, caso deseje um objeto estático (que não mude durante o ciclo iterativo) a fábrica *StaticIterationFactory* deve ser acoplada. O comando para definir a fábrica de iteração irá receber o nome do objeto que terá a fábrica acoplada e o nome da fábrica a ser adicionada.

- **Listar fábricas gráficas de objeto** (Tabela 42): além de fábricas de iteração, objetos podem ser compostos por fábricas gráficas. O comando para listar fábricas gráficas irá imprimir no console as fábricas gráficas disponíveis para um determinado

Tabela 41 – Descrição do comando definir fábricas gráficas.

Linha de Comando	object iteration_factories set <object_name> <factory_name>	
Escopo	OBJECT	
Thread Responsável	*WiseConsole*	
Entrada	<object_name>	Nome do objeto inteligente que irá receber a fábrica
	<factory_name>	Fábrica de iteração a ser inserido no objeto

objeto. Como dado de entrada deste comando, é recebido irá receber o nome deste objeto.

Tabela 42 – Descrição do comando para listar fábricas gráficas de objetos.

Linha de Comando	object graphic_factories list <object_name>	
Escopo	OBJECT	
Thread Responsável	*WiseConsole*	
Entrada	<object_name>	Nome do objeto a ser analisado

- **Definir fábrica gráfica de objeto** (Tabela 43): uma vez que o objeto tenha sido criado corretamente ele pode ser acoplado a fábricas de iteração ou gráficas. O comando para definir fábrica gráfica irá acoplar uma fábrica gráfica disponível a um objeto compatível. Como dados de entrada, o comando receberá o nome do objeto a ser alterado e o nome da fábrica gráfica a ser adicionada.

Tabela 43 – Descrição do comando para definir fábricas gráficas de objeto.

Linha de Comando	object graphic_factories set <object_name> <factory_name>	
Escopo	OBJECT	
Thread Responsável	*WiseConsole*	
Entrada	<object_name>	Nome do objeto que recebe a fábrica
	<factory_name>	Fábrica gráfica a ser inserido no objeto

- **Listar objetos gráficos** (Tabela 44): uma vez que o objeto gráfico tenha sido corretamente criado e acoplado a uma fábrica gráfica, o seu modelo é disponibilizado em uma lista de objetos gráficos. Essa lista exibe o nome de todos os objetos que podem ser visualizados e está disponível para visualização através de um comando.

Tabela 44 – Descrição do comando para listar objetos gráficos.

Linha de Comando	graphic list	
Escopo	OBJECT	
Thread Responsável	*WiseConsole*	
Entrada	<vazio>	Nenhuma

- **Listar Canvas** (Tabela 45): o comando para listar *Canvas* nomeia todos os elementos gráficos disponíveis. Quando o ambiente *InGU* está sendo executado não existem elementos gráficos disponíveis. Entretanto, como é descrito a seguir, o ambiente computacional *IGU* apresenta exatamente as mesmas funcionalidades com elementos gráficos. Ao executar este comando no ambiente computacional com elementos gráficos, cada objeto e seu nome são enviado como resultado.

Tabela 45 – Descrição do comando para listar elementos gráficos *Canvas*.

Linha de Comando	canvas list	
Escopo	GRAPHIC	
Thread Responsável	*WiseConsole*	
Entrada	<vazio>	Nenhuma

- **Link gráfico** (Tabela 46): tendo o conhecimento dos elementos gráficos disponíveis para desenho e das estruturas aptas a serem desenhadas, é possível que estas estruturas sejam conectadas para que funcionem em conjunto. Ao realizar esta conecção, nomeada de link gráfico, o elemento gráfico passará a exibir o elemento gráfico.

Tabela 46 – Descrição dos comandos que enviam um objeto gráfico para ser exibido em um elemento gráfico da interface de usuário.

Linha de Comando	graphic link <graphic_name> <canvas_name> canvas link <canvas_name> <graphic_name>	
Escopo	GRAPHIC	
Thread Responsável	*WiseConsole*	
Entrada	<graphic_name>	Nome do objeto gráfico a ser linkado com elemento gráfico *Canvas*
	<canvas_name>	Nome do *Canvas* a receber objeto gráfico para desenho

- **Terminar link gráfico** (Tabela 47): uma vez que os elementos gráficos começam a ser exibidos em elementos de interface gráfica eles são desenhados continuamente. Para liberar o elemento de interface gráfica e deixá-lo vazio, o comando de terminar link gráfico deve ser utilizado. Como dado de entrada, este comando recebe o nome do elemento de interface gráfica.

Tabela 47 – Descrição do comando terminar link gráfico.

Linha de Comando	canvas purge <canvas_name>	
Escopo	GRAPHIC	
Thread Responsável	*WiseConsole*	
Entrada	<canvas_name>	Canvas a ser limpo

3.3.2 CASO DE USO

O principal objetivo da ferramenta computacional é iterar modelos utilizando alguma lógica pré-definida por algum algoritmo disponível. Através dos comandos descritos na Seção **3.3.1** é possível que estes modelos sejam criados, alterados, iterados e, opcionalmente, visualizados.

A Figura 28 ilustra a principal sequência de atividades executadas para se iterar um objeto. Com este modelo é possível observar a sequência de passos necessária para se iterar um objeto.

Primeiramente, é necessário criar um projeto e selecioná-lo. Cada atividade representada no fluxograma é executada por um comando correspondente. O mesmo fluxo pode ser observado no Apêndice B. As primeiras linhas de comando representadas neste arquivo são de criação de projetos.

Ao criar um objeto, seus elementos também são criados. Esta funcionalidade está disponível ao usuário uma vez que ele tenha selecionado um projeto. É possível ainda criar um objeto a partir de um elemento. Desta forma, há garantia da igualdade inicial entre os modelos.

Com um objeto criado corretamente é possível adicionar a ele uma fábrica de iteração e uma gráfica. Ao acoplar a fábrica de iteração o método iterativo é liberado, bem como a possibilidade de alterar os parâmetros do objeto.

Também existe a possibilidade de se exportar elementos, para elementos *WiseGraphic* isso significa a exportação de imagem no formato **P**ortable **N**etwork **G**raphics, ou *png*, com seu comando representado na linha 7 do Apêndice B.

Como mencionado anteriormente, a mesma estrutura foi utilizada na construção do ambiente computacional composto por elementos gráficos *IGU*, conforme a Figura 29.

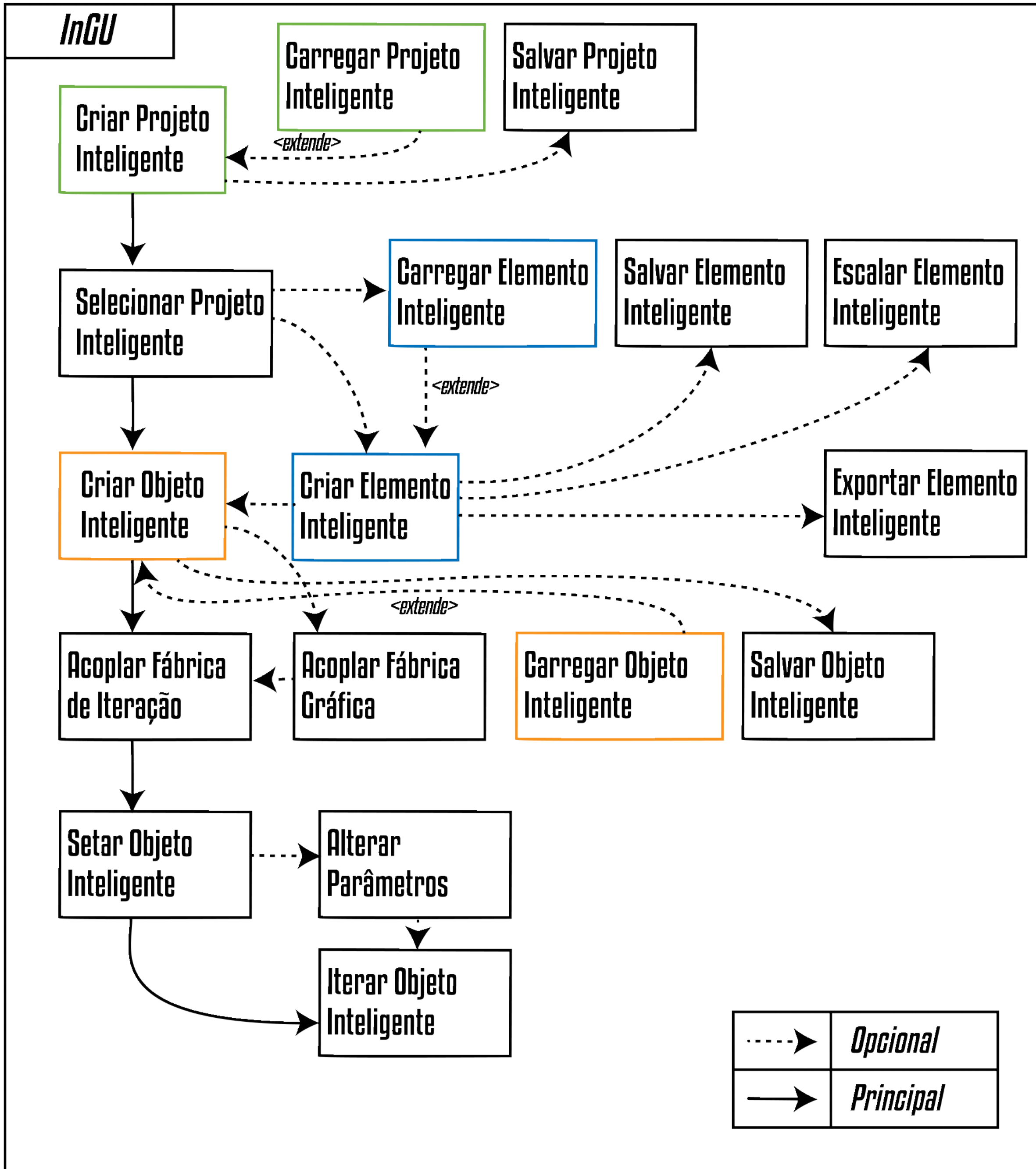


Figura 28 – Fluxo de execução do ambiente computacional InGU, em azul as atividades de criação de elementos, em verde as atividades de criação de projetos e em laranja as atividades de criação de objetos.

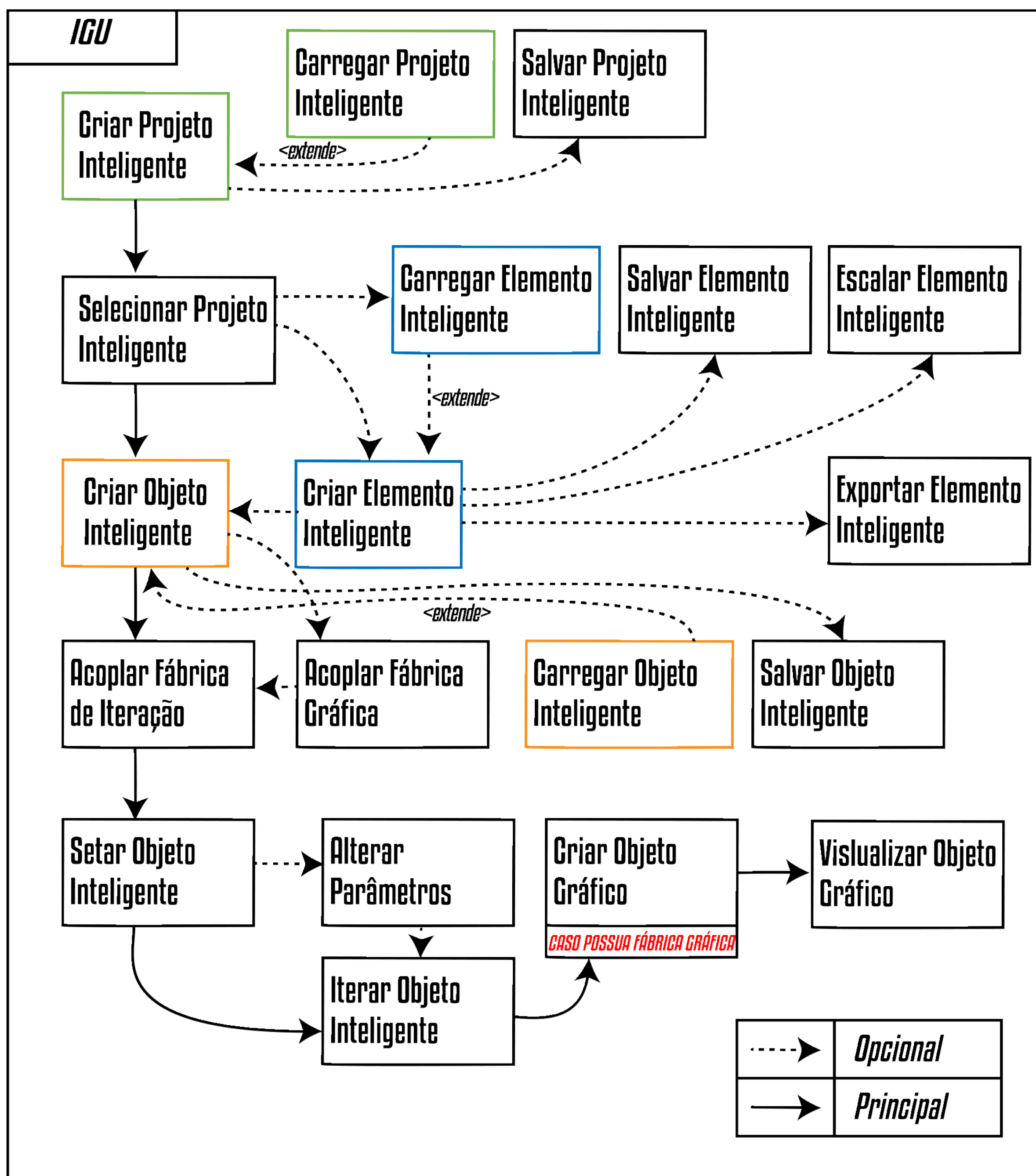


Figura 29 – Fluxo de execução do ambiente computacional IGU.

Desta forma, o principal fluxo de uso da interface gráfica foi desenvolvido. O funcionamento apresentado pela Figura 29 difere do apresentado na Figura 28 apenas na visualização dos objetos gráficos disponibilizada pelos elementos gráficos. Isto garante que ambos os ambientes computacionais sejam alterados em uma eventual atualização de código e que tenham funcionamento idêntico.

3.3.3 JANELA PRINCIPAL

O ambiente com interface gráfica foi nomeado de **I**terador **G**ráfico **U**niversal (IGU), que também está presente como parte de um projeto *Qt/C++*. Diferentemente do InGU, este possui elementos gráficos. O principal elemento gráfico disponibilizado pela interface gráfica é a janela principal, que é um objeto do tipo *QMainWindow*. Estes objetos são disponibilizados pela biblioteca *Qt* e permitem que um projeto *C++* possua uma interface gráfica utilizando diretivas OpenGL e GLUT (*The OpenGL Utility Kit*) [18].

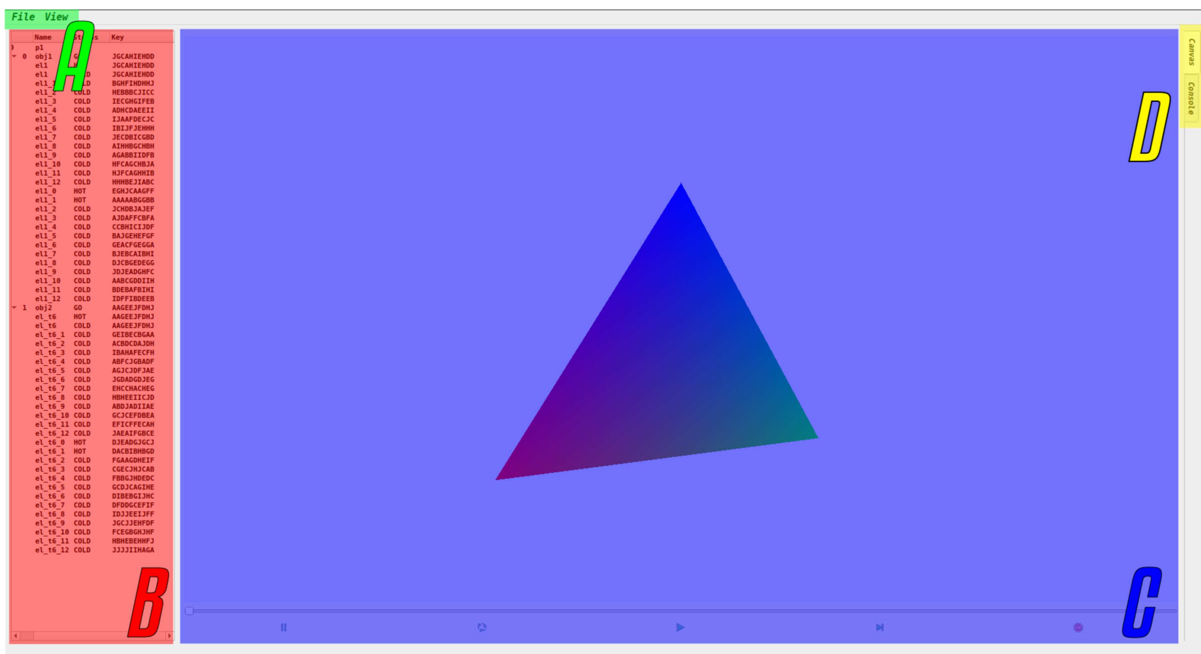


Figura 30 – Janela principal ambiente computacional IGU. Da esquerda para a direita: (A) Menu principal do programa; (B) Árvore de projetos e seus elementos; (C) Área de trabalho, no caso mostrando OpenGl *Canvas*; (D) Seleção de abas

A Figura 30 ilustra o comportamento inicial da ferramenta computacional. Ao ser aberta, o ambiente de trabalho, projetos, objetos e elementos são recuperados de um arquivo contido na mesma parte da ferramenta computacional. Este arquivo é salvo toda vez que a ferramenta é encerrada com projetos ainda no ambiente. Ainda nesta figura estão representados os principais grupos de elementos gráficos.

A seguir, discorrem-se sobre cada grupo de elemento gráfico presente na janela principal da ferramenta e seu funcionamento.

- **Menu**: o primeiro grupo são os elementos que constituem o menu principal da aplicação. Cada opção do menu é representada por um uma linha de texto, ao selecionar a linha uma ação é executada (em detalhe na Fig. 31):
  - **File/Exit**: Fecha o ambiente computacional.
  - **File/Jobs**: Abre a Janela que exibe os trabalhos criados.
  - **View/Open Canvas**: Abre uma nova janela contendo um novo elemento gráfico OpenGL *Canvas*.

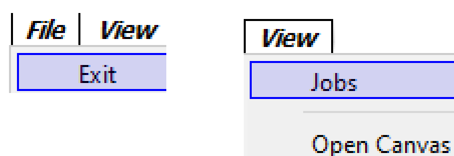


Figura 31 – Opções do menu principal da ferramenta computacional *IGU*

Ao abrir um novo elemento gráfico *Canvas*, um novo nome será listado ao executar o comando para listar canvas. Isso possibilita que mais de um objeto seja exibido ao mesmo tempo em janelas distintas.

- **Abas e área de trabalho**: as abas da janela principal selecionam o elemento de interface gráfica que é exibido na área de trabalho da aplicação.

  Ao selecionar uma das abas, os elementos de interface de usuário da área de trabalho são trocados. Estes elementos são objetos do tipo *QWidget*, que por sua vez são divididos em dois ambientes: o *Canvas* e o *Console*.

  A aba *Console* exibe um ambiente similar ao disponibilizado pelo ambiente *InGU*. Utilizando um elemento de exibição textual longa, uma linha de texto como entrada e um botão, os mesmo comandos disponibilizados na Seção **3.3.1** são acessados por estes elementos gráficos. Para executar uma linha de comando, basta inserir-lá no elemento gráfico e pressionar o botão ou a tecla *Enter*, a resposta é acoplada ao elemento de exibição textual longa.

  A Figura 32 mostra as áreas de trabalho disponibilizadas ao selecionar uma das abas. Através da opção *Canvas*, o elemento gráfico que contém um *QWidget* cuja principal função é uma tela *OpenGL*. Os objetos gráficos, que são resultados de iterações, podem ser visualizados aqui. O funcionamento desta área de trabalho é descrito pela janela *Canvas* e funciona como um reprodutor de vídeo.

  As abas foram construídas para realizar todas as funções da ferramenta computacional com seus elementos gráficos. O ciclo de vida da *thread WiseThreadPool* esta diretamente ligado à interface gráfica. Ao utilizar a interface gráfica para percorrer os elementos gráficos, ou executar a animação, desencadeiam chamadas diretas as *threads*.

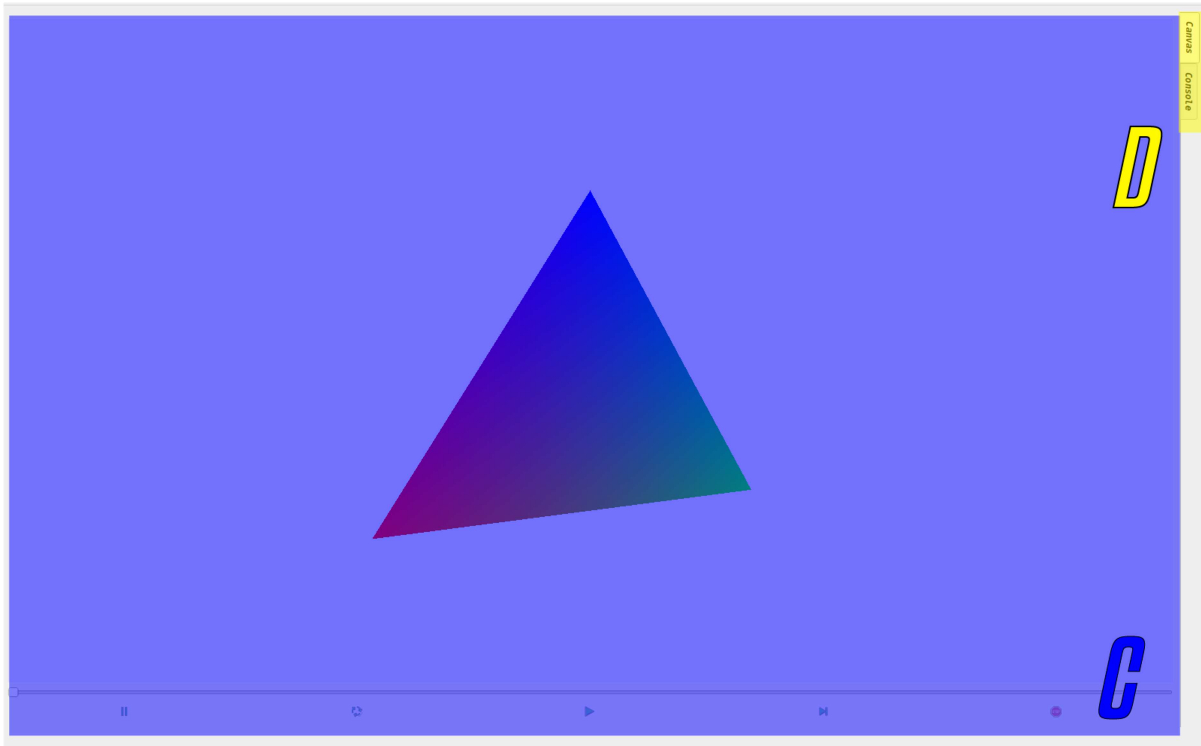


Figura 32 – Parte da janela principal contendo a área de trabalho e as abas que selecionam a interface selecionada, sendo os componentes: (C) Área de trabalho *Canvas*; (D) Seleção de abas

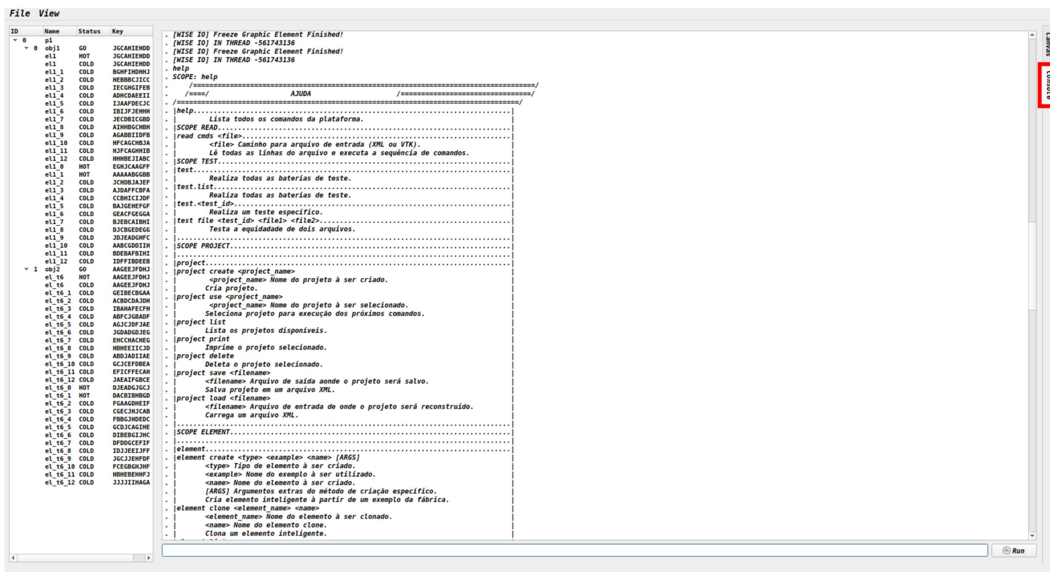


Figura 33 – Área de trabalho exibindo o elemento gráfico *Canvas* com a aba *Canvas* selecionada.

Figura 34 – Área de trabalho exibindo o elemento gráfico *Console* com a aba *Console* selecionada.

3.3.4 ÁRVORE DE PROJETOS

Além da área de trabalho, o elemento gráfico que exibe a árvore de projetos permanece fixo. Este elemento gráfico é um explorador de árvore *QTreeWidget* que exibe todos os projetos carregados e suas estruturas. Esta árvore permite rapidamente verificar os elementos criados, seus nomes, suas chaves únicas e seu estado atual.

A Figura 35 apresenta o elemento gráfico de uma árvore de projetos com objetos carregados. Na imagem está um projeto nomeado *p1*, um objeto *obj1* e diversos elementos e elementos gráficos. Como visto anteriormente quando o objeto está no estado *GO* significa que ele já foi corretamente configurado e iterado. Dentro deste objeto estão os elementos e elementos gráficos. O primeiro elemento *el1* é o elemento contido na estrutura *Forno* e é o único elemento no estado *Hot*. Em seguida, estão os elementos da estrutura *Freezer*, seguidos pelos objetos gráficos.

Através da árvore de projetos é possível observar a troca de estado dos elementos, por exemplo, ao executar a animação, os elementos gráficos são sucessivamente aquecidos. Observa-se que a mesma quantidade de elementos gráficos e elementos, mantendo a consistência do objeto. Cada iteração gera um elemento com o resultado numérico e um elemento gráfico com a representação gráfica.

3.3.5 JANELAS

Nesta seção, são descritas as janelas disponíveis no menu, o qual foi apresentado na Seção **3.3.3**. Existem dois tipos de janelas disponibilizadas pela ferramenta computacional: *Canvas*, que funciona como um reprodutor de vídeo, exibindo elementos gráficos

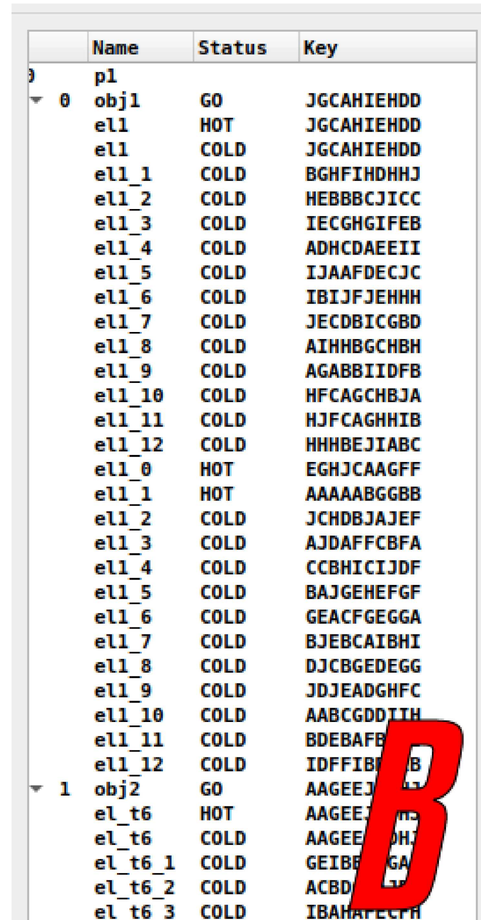
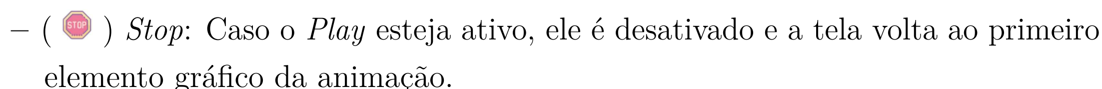


Figura 35 – Árvore de projetos, na imagem a ferramenta apresenta um projeto $p1$, um objeto $obj1$ e seus elementos gráficos e elementos.

sucessivamente; *Jobs*, o qual exibe todas as demandas criadas pelo programa enviadas à *thread* inteligente *WiseThreadTool*. Esta última janela pode ser instanciada apenas uma vez, enquanto diversos *Canvas* podem ser disponibilizados e exibir diferentes objetos.

- **Janela Canvas**: a janela e área de trabalho *Canvas* possibilitam que o usuário selecione o quadro a ser exibido. Cada quadro representa um objeto do tipo *GraphicObject* que é um componente do modelo gráfico *GraphicModel*. O quadro é selecionado através de uma barra de progresso e de botões que alteram a animação feita com o objeto gráfico. Estes botões são:

  - ( ) *Pause*: Pausa a animação do objeto gráfico;
  - ( )*Repeat*: Este botão tem um funcionamento liga e desliga. Ao ser ligado, quando a animação chegar ao último elemento gráfico retornará ao primeiro;
  - ( )*Play*: Este botão também tem um funcionamento liga e desliga. Ao ser ligado, o elemento gráfico irá percorrer por todos os quadros da animação;
  - ( )*Avançar quadro*: Avança um quadro da animação;



– (STOP) *Stop*: Caso o *Play* esteja ativo, ele é desativado e a tela volta ao primeiro elemento gráfico da animação.

Todas as funções do *Canvas* só são disponibilizadas ao usuário depois que um objeto gráfico é ligado a tela *OpenGl*. Quando um objeto é exibido em um *Canvas*, cada quadro exibido altera quais objetos gráficos *GraphicObject* estarão no estado aquecido *Hot*. Isto foi feito utilizando a conexão entre objetos *QObjects* e os elementos gráficos *QWidget*. Desta forma a tela é capaz de acionar diretamente a estrutura do *WiseThreadPool* e selecionar quais elementos devem ser aquecidos e exibidos. A mesma conexão é responsável por enviar a linha de comando da aba *Console* a *WiseThreadPool* para que seja executada e, em seguida, enviar a mensagem de resposta ao elemento textual.

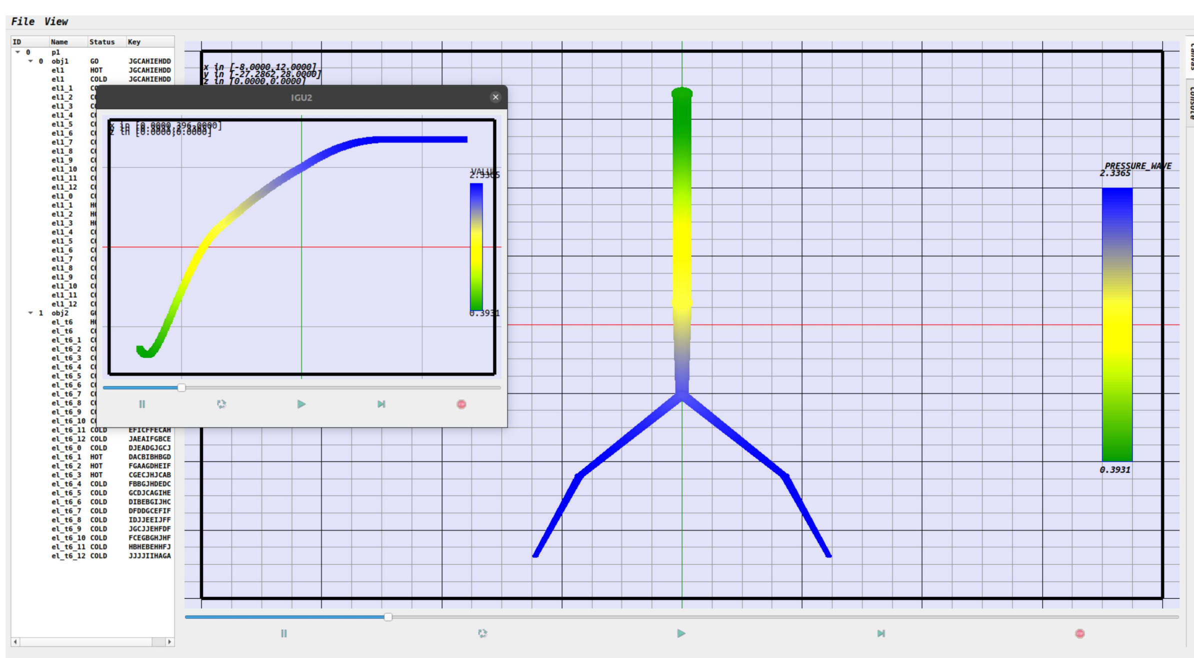
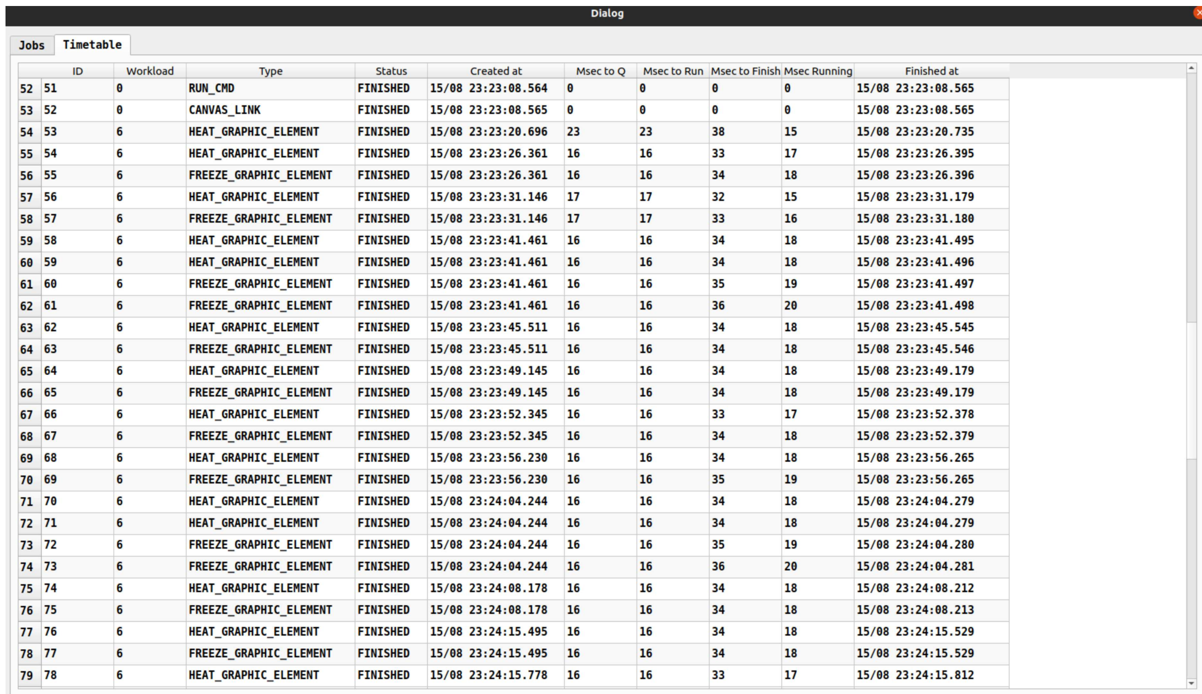


Figura 36 – Ilustra a janela *Canvas* exibida sobre a área de trabalho *Canvas*, a janela exibindo um gráfico obtido como resultado e a área de trabalho exibindo a árvore arterial estudada.

Durante sua execução esta janela irá selecionar o elemento gráfico *GraphicElement* que deve ser exibido, selecionando o elemento através do objeto gráfico *GraphicObject*. Para que o elemento seja exibido corretamente ele precisa estar no estado *Hot*, que representa o momento em que o elemento está corretamente carregado em memória com o modelo geométrico e o parâmetro a ser exibido. Portanto, ao selecionar o elemento o *Canvas* irá criar trabalhos de aquecimento dos elementos. A coleção de objetos gráficos *GraphicModel* é encarregada de enviar estes trabalhos a *thread* inteligente *WiseThreadPool*.

- **Janela Jobs**: a janela *Jobs* exibe uma listagem compreensiva dos processos iniciados

e enviados a *thread WiseThreadPool*. Cada comando executado via *Console* ou elemento de interface gráfica irá gerar processos listados nesta janela.



	ID	Workload	Type	Status	Created at	Msec to Q	Msec to Run	Msec to Finish	Msec Running	Finished at
52	51	0	RUN_CMD	FINISHED	15/08 23:23:08.564	0	0	0	0	15/08 23:23:08.565
53	52	0	CANVAS_LINK	FINISHED	15/08 23:23:08.565	0	0	0	0	15/08 23:23:08.565
54	53	6	HEAT_GRAPHIC_ELEMENT	FINISHED	15/08 23:23:20.696	23	23	38	15	15/08 23:23:20.735
55	54	6	HEAT_GRAPHIC_ELEMENT	FINISHED	15/08 23:23:26.361	16	16	33	17	15/08 23:23:26.395
56	55	6	FREEZE_GRAPHIC_ELEMENT	FINISHED	15/08 23:23:26.361	16	16	34	18	15/08 23:23:26.396
57	56	6	HEAT_GRAPHIC_ELEMENT	FINISHED	15/08 23:23:31.146	17	17	32	15	15/08 23:23:31.179
58	57	6	FREEZE_GRAPHIC_ELEMENT	FINISHED	15/08 23:23:31.146	17	17	33	16	15/08 23:23:31.180
59	58	6	HEAT_GRAPHIC_ELEMENT	FINISHED	15/08 23:23:41.461	16	16	34	18	15/08 23:23:41.495
60	59	6	HEAT_GRAPHIC_ELEMENT	FINISHED	15/08 23:23:41.461	16	16	34	18	15/08 23:23:41.496
61	60	6	FREEZE_GRAPHIC_ELEMENT	FINISHED	15/08 23:23:41.461	16	16	35	19	15/08 23:23:41.497
62	61	6	FREEZE_GRAPHIC_ELEMENT	FINISHED	15/08 23:23:41.461	16	16	36	20	15/08 23:23:41.498
63	62	6	HEAT_GRAPHIC_ELEMENT	FINISHED	15/08 23:23:45.511	16	16	34	18	15/08 23:23:45.545
64	63	6	FREEZE_GRAPHIC_ELEMENT	FINISHED	15/08 23:23:45.511	16	16	34	18	15/08 23:23:45.546
65	64	6	HEAT_GRAPHIC_ELEMENT	FINISHED	15/08 23:23:49.145	16	16	34	18	15/08 23:23:49.179
66	65	6	FREEZE_GRAPHIC_ELEMENT	FINISHED	15/08 23:23:49.145	16	16	34	18	15/08 23:23:49.179
67	66	6	HEAT_GRAPHIC_ELEMENT	FINISHED	15/08 23:23:52.345	16	16	33	17	15/08 23:23:52.378
68	67	6	FREEZE_GRAPHIC_ELEMENT	FINISHED	15/08 23:23:52.345	16	16	34	18	15/08 23:23:52.379
69	68	6	HEAT_GRAPHIC_ELEMENT	FINISHED	15/08 23:23:56.230	16	16	34	18	15/08 23:23:56.265
70	69	6	FREEZE_GRAPHIC_ELEMENT	FINISHED	15/08 23:23:56.230	16	16	35	19	15/08 23:23:56.265
71	70	6	HEAT_GRAPHIC_ELEMENT	FINISHED	15/08 23:24:04.244	16	16	34	18	15/08 23:24:04.279
72	71	6	HEAT_GRAPHIC_ELEMENT	FINISHED	15/08 23:24:04.244	16	16	34	18	15/08 23:24:04.279
73	72	6	FREEZE_GRAPHIC_ELEMENT	FINISHED	15/08 23:24:04.244	16	16	35	19	15/08 23:24:04.280
74	73	6	FREEZE_GRAPHIC_ELEMENT	FINISHED	15/08 23:24:04.244	16	16	36	20	15/08 23:24:04.281
75	74	6	HEAT_GRAPHIC_ELEMENT	FINISHED	15/08 23:24:08.178	16	16	34	18	15/08 23:24:08.212
76	75	6	FREEZE_GRAPHIC_ELEMENT	FINISHED	15/08 23:24:08.178	16	16	34	18	15/08 23:24:08.213
77	76	6	HEAT_GRAPHIC_ELEMENT	FINISHED	15/08 23:24:15.495	16	16	34	18	15/08 23:24:15.529
78	77	6	FREEZE_GRAPHIC_ELEMENT	FINISHED	15/08 23:24:15.495	16	16	34	18	15/08 23:24:15.529
79	78	6	HEAT_GRAPHIC_ELEMENT	FINISHED	15/08 23:24:15.778	16	16	33	17	15/08 23:24:15.812

Figura 37 – Janela *Jobs* exibida com a aba *Timetable* selecionada, a janela lista os trabalhos recebidos pela estrutura *WiseThreadPool* e seus tempos de execução.

A Figura 37 mostra a lista de todos os trabalhos recebidos pela *thread WiseThread* e suas propriedades. Cada linha nesta janela representa um *WiseJob* e as colunas descrevem propriedades de cada trabalho:

- *ID*: Número de identificação;
- *Workload*: Número da carga de trabalho;
- *Status*: Estado do trabalho;
- *Created at*: Data de criação;
- *Msec to Q*: Milisegundos do momento da criação até a chegada a fila;
- *Msec to Run*: Milisegundos do momento da criação até a alocação do trabalho em uma *thread*;
- *Msec to Finish*: Milisegundos do momento da criação até o final de sua execução;
- *Msec Running*: Milisegundos em execução;
- *Finished at*: Data de finalização;

A Figura 38 demonstra as cinco listas exibidas ao selecionar a aba *Timetable* da janela *Jobs*. A primeira lista *All Jobs* lista todos os trabalhos criados e as listas subsequentes listam as listas presentes no gerenciador de threads *WiseThreadPool*

Figura 38 – Janela *Jobs* exibida com a aba *Timetable* selecionada, a janela lista os trabalhos recebidos pela estrutura *WiseThreadPool* e os separa logicamente pelas listas de espera.

descritas na Seção **3.2.8**. Desta forma é possível verificar os trabalhos que aguardam execução, estão sendo executados ou foram finalizados.

4 RESULTADOS OBTIDOS

Neste capítulo, resultados obtidos com a ferramenta computacional construída são apresentados. Os valores de amplitude da pressão $|P|$ foram calculados em toda árvore arterial proposta utilizando a ferramenta computacional.

4.1 SIMULAÇÕES HEMODINÂMICAS

Nesta seção, apresentam-se resultados obtidos através das simulações do modelo matemático de Duan e Zamir [12] usando a ferramenta computacional desenvolvida tanto no ambiente InGU quanto no ambiente IGU. As simulações realizadas aqui tratam da propagação de uma onda harmônica simples ao longo de uma árvore, onde reflexões de onda modificam a amplitude da onda de pressão enquanto ela avança.

A escolha de uma onda harmônica simples neste estudo possibilita investigar os efeitos da frequência, fluido viscoso e viscoelasticidade da parede do segmento de vaso.

Considerou-se também neste estudo um modelo de árvore arterial canina como ilustrado na Figura 39. As propriedades dos segmentos foram escolhidas oriundas dos dados de Fung [15] e são descritas na Tabela 48.

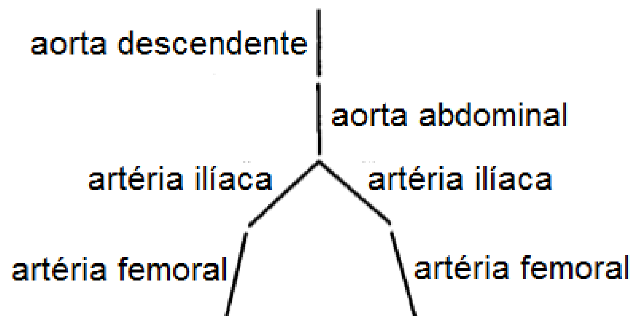


Figura 39 – Representação do modelo de árvore arterial canina (figura adaptada de [12]).

Tabela 48 – Propriedades dos vasos do modelo de árvore arterial [10, 12]

Artéria	Comprimento (cm)	Densidade ρ (g/cm^3)	Viscosidade μ_0 (g/cms)	Diâmetro (cm)	Módulo de Young (dyn/cm^2)
Aorta Descendente	25	$0,960$	0,0385	1,3	4,8 $\times 10^6$
Aorta Abdominal	11	$1,134$	0,0449	0,9	1,0 $\times 10^7$
Ilíaca	12	$1,172$	0,0472	0,6	1,0 $\times 10^7$
Femoral	10	$1,235$	0,0494	0,4	1,0 $\times 10^7$

Nas simulações realizadas, calculou-se a distribuição de amplitude de pressão ao longo da árvore arterial (Figura 39). Os resultados foram obtidos para quatro diferentes

frequências e três diferentes cenários de escoamento/segmento: (i) escoamento viscoso em segmento puramente elástico (cenário 1 da Seção **2.1.4**), (ii) escoamento invíscido em segmento viscoelástico (cenário 2) e (iii) escoamento viscoso em segmento viscoelástico (cenário 3).

Os resultados obtidos nas simulações são mostrados nas Figuras 40, 41, 42, 43, 44 envolvendo a amplitude da pressão ao longo do modelo de árvore arterial. Nestas figuras, o comprimento de cada segmento arterial foi dimensionado para $1,0$, de modo que o comprimento adimensional total da árvore é $4,0$. O comprimento real é 58 cm. A amplitude da pressão também foi escalada pela pressão de entrada $P_o = 1$, e os resultados finais são portanto mostrados em termos de amplitude de pressão adimensional $|P|$ versus a distância adimensional X do início da árvore.

4.1.1 ESCOAMENTO VISCOSO

Nas Figuras 40 e 41, o efeito da viscosidade do fluido é examinado separadamente considerando-se o escoamento em vasos puramente elásticos com quatro valores diferentes de viscosidade do fluido, ou seja, $\mu = 0$; $0,5\mu_0$; $1,0\mu_0$ e $1,5\mu_0$, onde μ_0 é o valor base da viscosidade da Tabela 48. Observa-se que o efeito da viscosidade do fluido é reduzir o aumento global na amplitude da onda de pressão causada pelas reflexões das ondas à medida que a onda se desloca na direção à jusante. Além disso, modera os picos locais na distribuição de pressão.

4.1.2 VASO VISCOELÁSTICO

Nas Figuras 42 e 43, o efeito da viscoelasticidade da parede do vaso é considerado separadamente considerando-se o escoamento invíscido e tomando-se quatro valores diferentes da viscoelasticidade da parede do vaso. O modelo viscoelástico proposto utilizado para fins destes cálculos é apresentado no cenário 2 da Seção **2.1.4**, no qual a viscoelasticidade da parede do vaso é representada por um módulo de Young complexo. Estas figuras mostram os resultados para $\phi_0 = 0^o$, 4°, 8° e 12°. Quando $\phi_0 = 0^o$, tem-se um valor representando uma parede puramente elástica e para $\phi_0 > 0$ tem-se a representação da viscoelasticidade. Nota-se a partir destas figuras que o efeito da viscoelasticidade, como o da viscosidade do fluido, é amortecer o aumento global da amplitude da onda de pressão causada pelas reflexões das ondas à medida que a onda se desloca na direção à jusante, bem como moderar os picos locais na distribuição de pressão.

4.1.3 ESCOAMENTO VISCOSO EM VASO VISCOELÁSTICO

Nas Figuras 44, 45, o efeito da viscoelasticidade da parede do segmento é adicionada ao efeito do escoamento viscoso, adotando dois valores diferentes da viscoelasticidade e dois valores de viscosidade. O modelo utilizado no cenário 3 é a soma dos efeitos apresentados

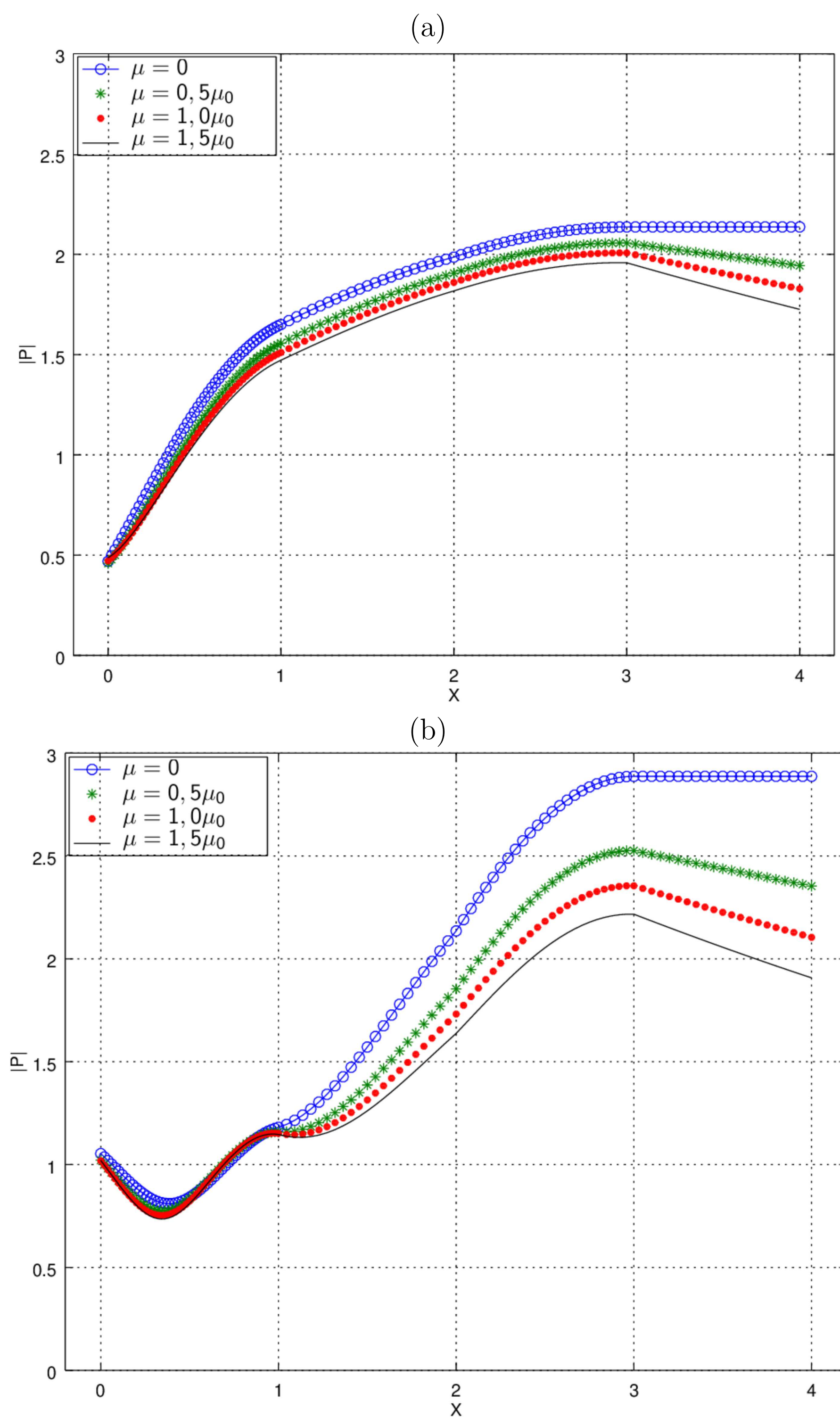
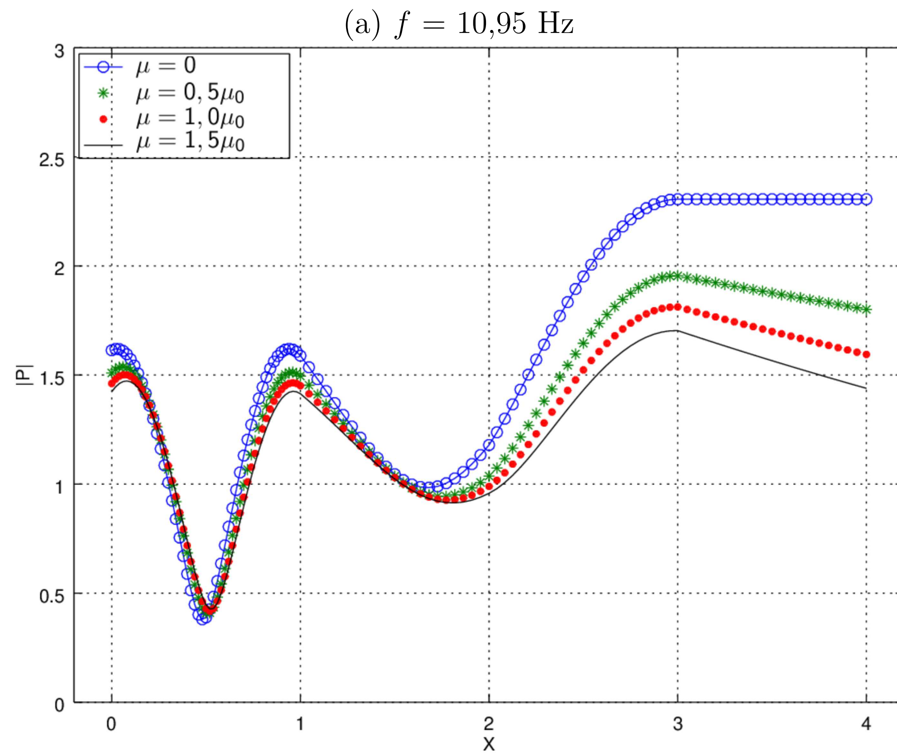


Figura 40 – Amplitude da pressão $|P|$ ao longo da árvore arterial considerando diferentes viscosidade do fluido μ e frequências: (a) f = 3,65 Hz, (b) f = 7,30 Hz.

(a) f = 10,95 Hz



(b) f = 14,60 Hz

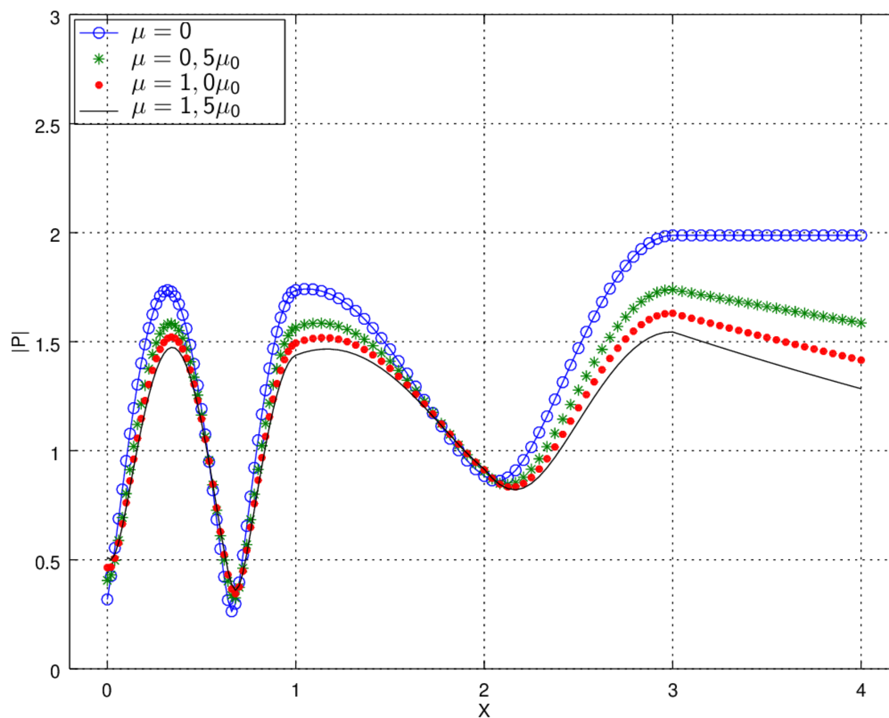


Figura 41 – Amplitude da pressão $|P|$ ao longo da árvore arterial considerando diferentes viscosidade do fluido μ e frequências: (a) f = 10,95 Hz, (b) f = 14,60 Hz.

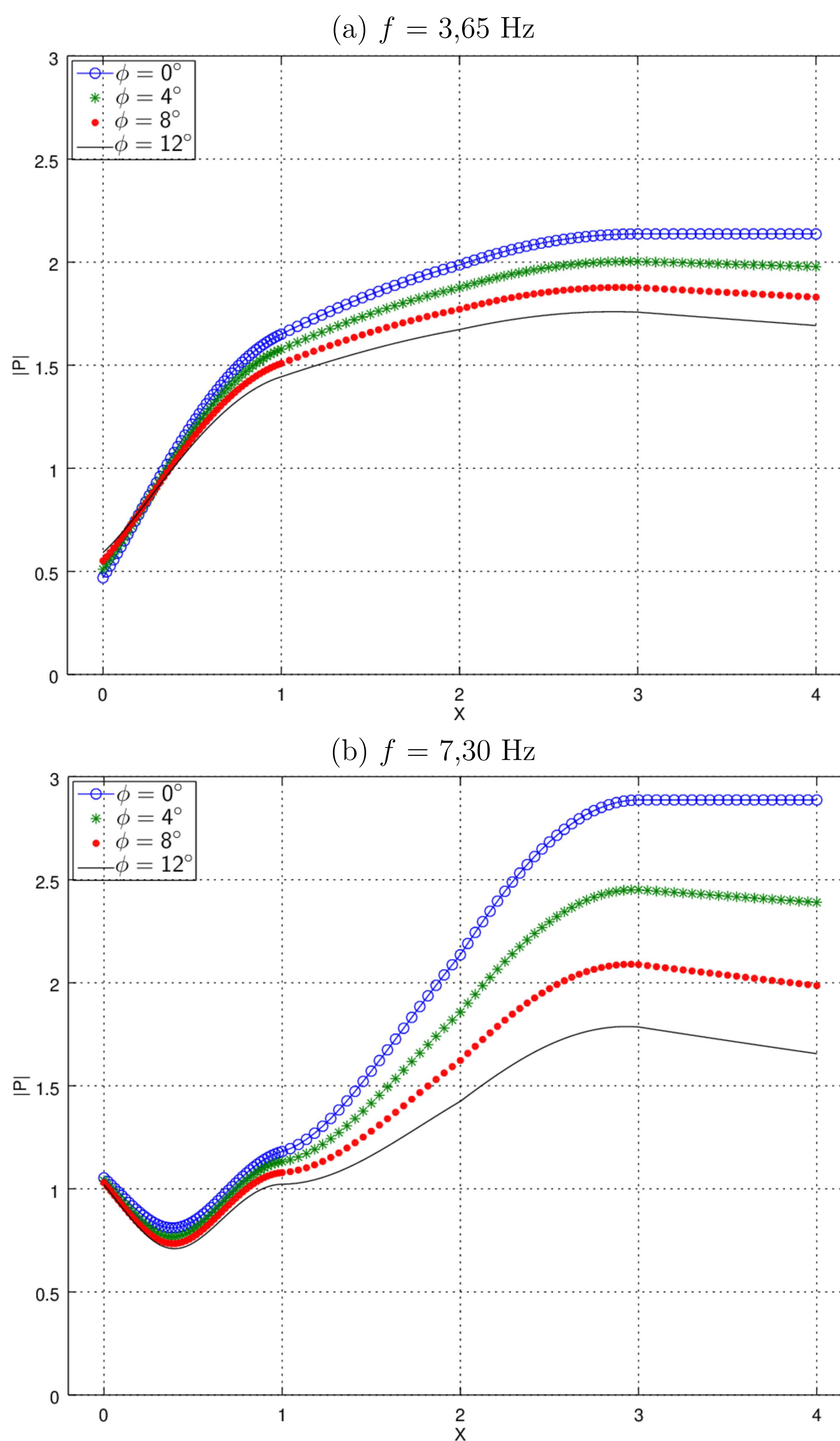


Figura 42 – Amplitude da pressão $|P|$ ao longo da árvore arterial considerando diferentes valores de viscoelasticidade ϕ_0 e frequências: f = 3,65 Hz e f = 7,30 Hz.

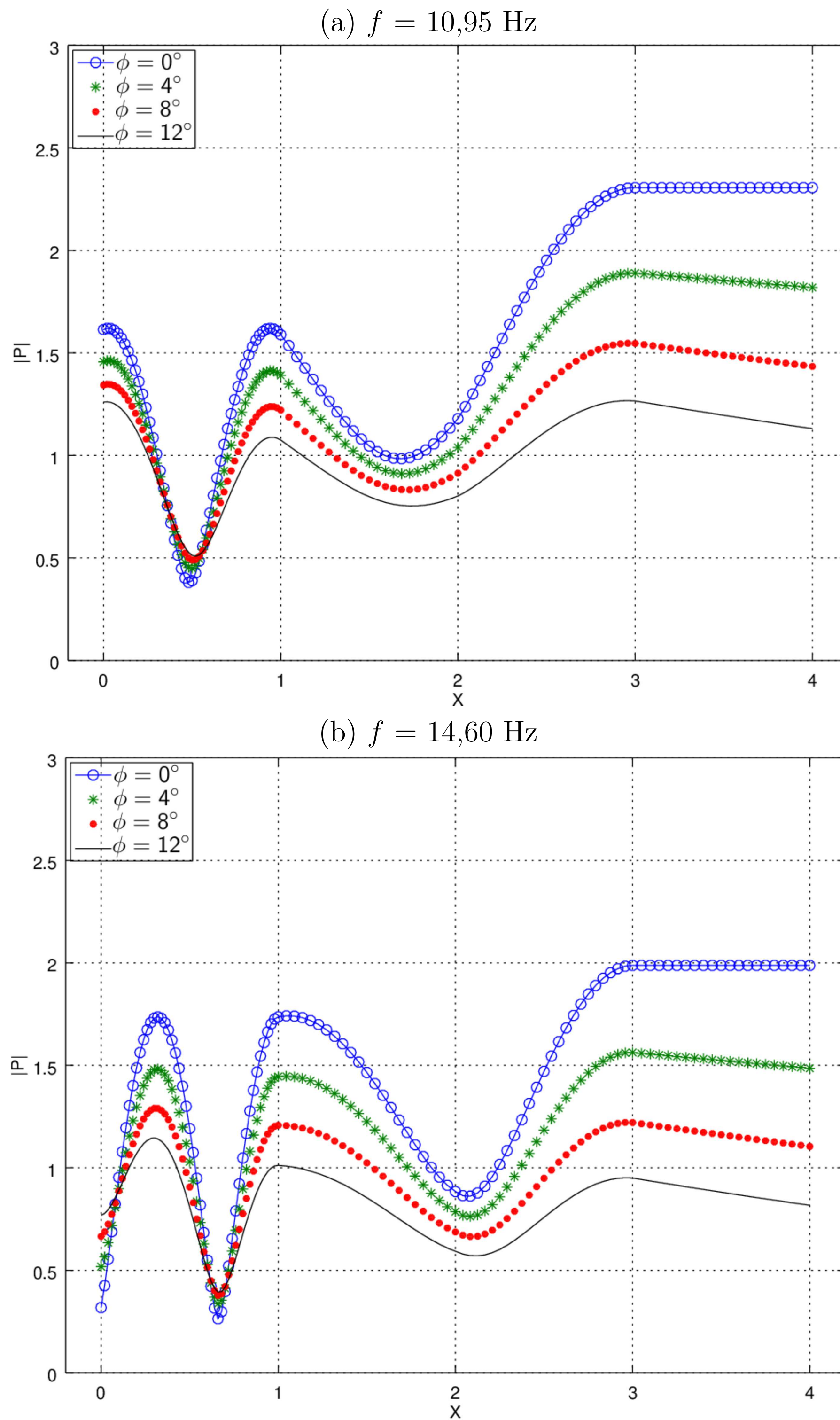


Figura 43 – Amplitude da pressão $|P|$ ao longo da árvore arterial considerando diferentes valores de viscoelasticidade ϕ_0 e frequências: f = 10,95 Hz e f = 14,60 Hz.

na Seção **2.1.4**, adicionando o fator viscoso e o módulo de Young complexo. Estas figuras mostram o resultado para $\phi_0 = 0^o$, 8° e com as viscosidades $\mu = 0$ e $0,5\mu_0$. Ao visualizar os efeitos da viscoelasticidade e viscosidade na amplitude da onda de Pressão P, é observado o amortecimento global da amplitude de onda de pressão, somado à moderação dos picos locais na distribuição de pressão. Entretanto, o efeito somado dos fenômenos causa um amortecimento mais eficaz que o visto nos outros cenários.

4.2 ANÁLISES DE DESEMPENHO DA FERRAMENTA COMPUTACIONAL

Nesta seção, apresentam-se resultados obtidos com a implementação da ferramenta computacional em seus dois ambientes *IGU* e *InGU*. As simulações realizadas aqui tratam da aplicação da ferramenta na obtenção de resultados do escoamento pulsátil em um modelo de árvore arterial. Os resultados de ambas as versões serão apresentados e comparados com a versão anteriormente apresentada.

Na Figura 46, tem-se selecionado o arquivo *CMakeLists.txt* que é o responsável por interpretar o projeto *Qt* e corretamente compilar as bibliotecas necessárias. Esse arquivo é o responsável por dizer quais arquivos são necessários para se compilar o projeto na estrutura *CMake*, no ambiente *Qt* este arquivo contém as instruções para compilar todos os ambientes deste trabalho e um ambiente com a versão *alpha* da ferramenta computacional *IGU0*. A versão *alpha* da ferramenta computacional *IGU* exporta seus resultados em arquivos *VTK*, que podem ser lidos pela nova versão da ferramenta ou ainda utilizados em outro ambiente.

Ao abrir o arquivo do projeto é possível verificar a existência de três projetos à serem compilados: (1) *IGU0*, contendo a versão *alpha* do programa com interface; (2) *IGU*, que é a versão atual da ferramenta com interface gráfica apresentada neste trabalho; e (3) *InGU* que é a versão atual da ferramenta sem uma interface gráfica. Podemos ver estes projetos presentes na Figura 46.

A versão *alpha* não contém o padrão de projeto de Fábricas ou classes de propósito único. Enquanto os ambientes *IGU* e *InGU* utilizam este padrão de projeto e estrutura de classes, utilizando as mesmas classes em ambos ambientes.

Na Figura 47 pode ser observado que a interface gráfica do usuário possui duas telas de *Canvas* e diversos botões na janela principal. O funcionamento desta interface é muito similar ao da nova interface, entretanto está muito mais ligada exclusivamente à simulação do escoamento pulsátil. Dois *Canvas* possibilitam a exibição de uma árvore arterial e seu gráfico ao mesmo tempo com a propriedade hemodinâmica calculada (no exemplo, a pressão ao longo dos vasos), entretanto fixa estes dois elementos na janela principal. Com o esquema de janelas desenvolvido no ambiente *IGU* é possível que diversos objetos gráficos sejam exibidos ao mesmo tempo e estas janelas escaladas independentemente.

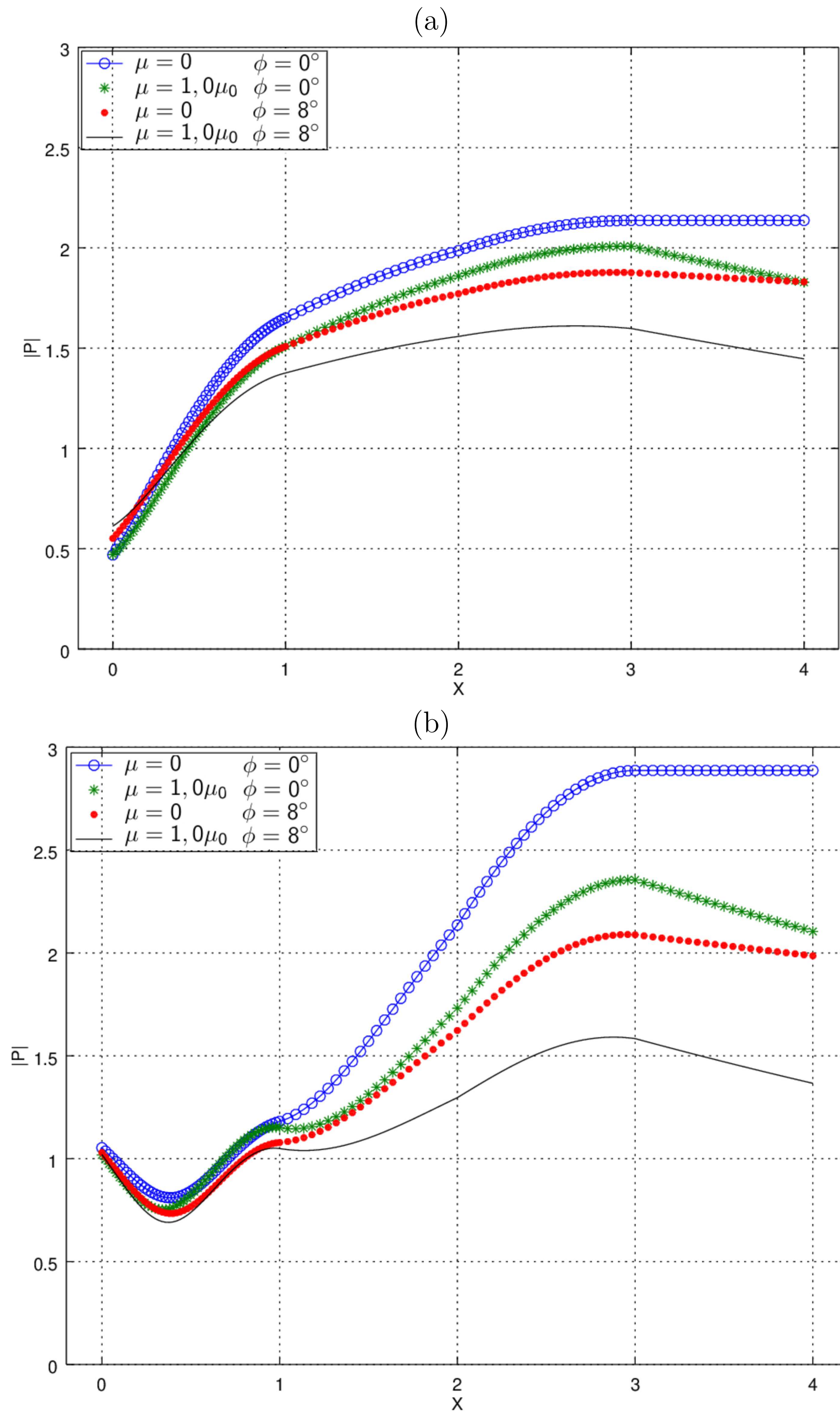
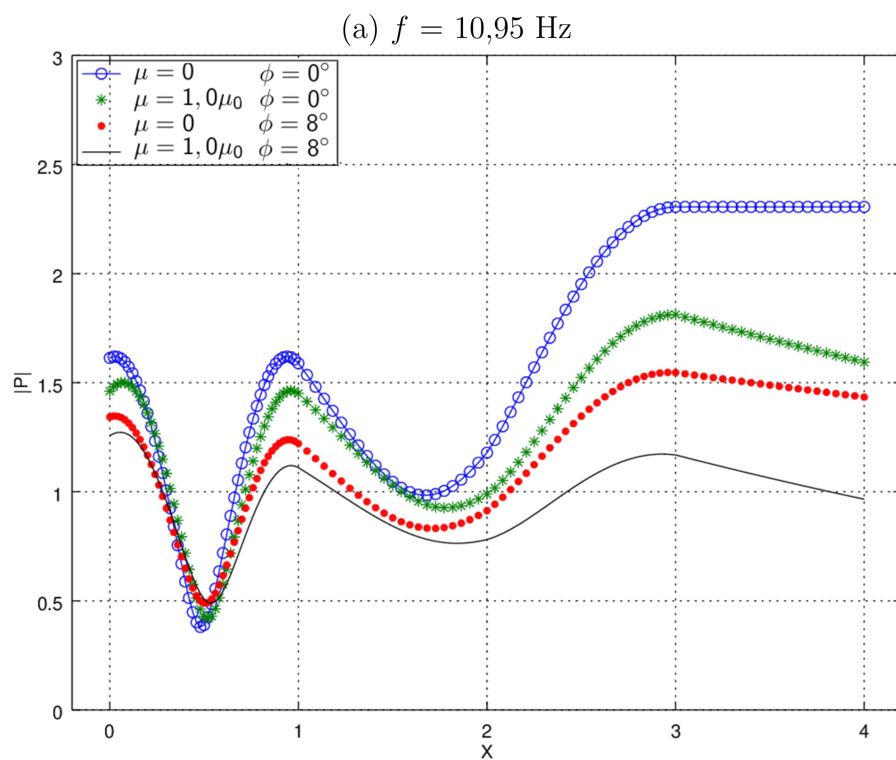


Figura 44 – Amplitude da pressão $|P|$ ao longo da árvore arterial considerando diferentes valores de viscoelasticidade ϕ_0 e frequências: (a) f = 3,65 Hz, (b) f = 7,30 Hz.

(a) f = 10,95 Hz



(b) f = 14,60 Hz

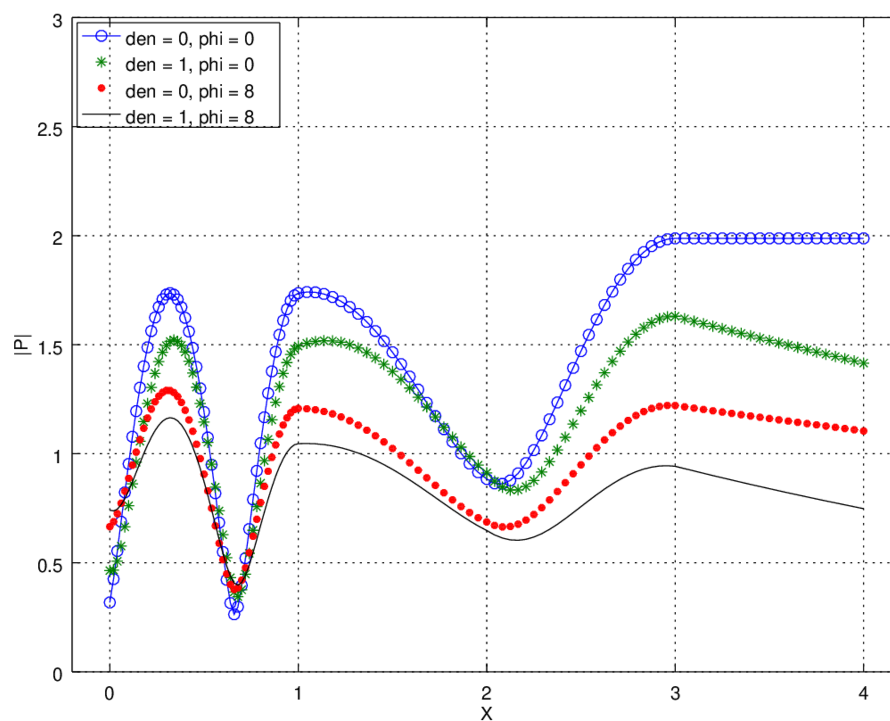


Figura 45 – Amplitude da pressão $|P|$ ao longo da árvore arterial considerando diferentes valores de viscoelasticidade ϕ_0 e frequências: (a) f = 10,95 Hz, (b) f = 14,60 Hz.

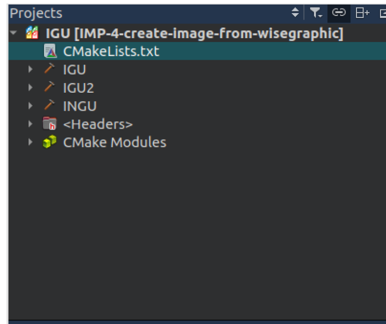


Figura 46 – Representação do projeto aberto através do arquivo *CMakeLists.txt* na interface *IDE* do *QtCreator*.

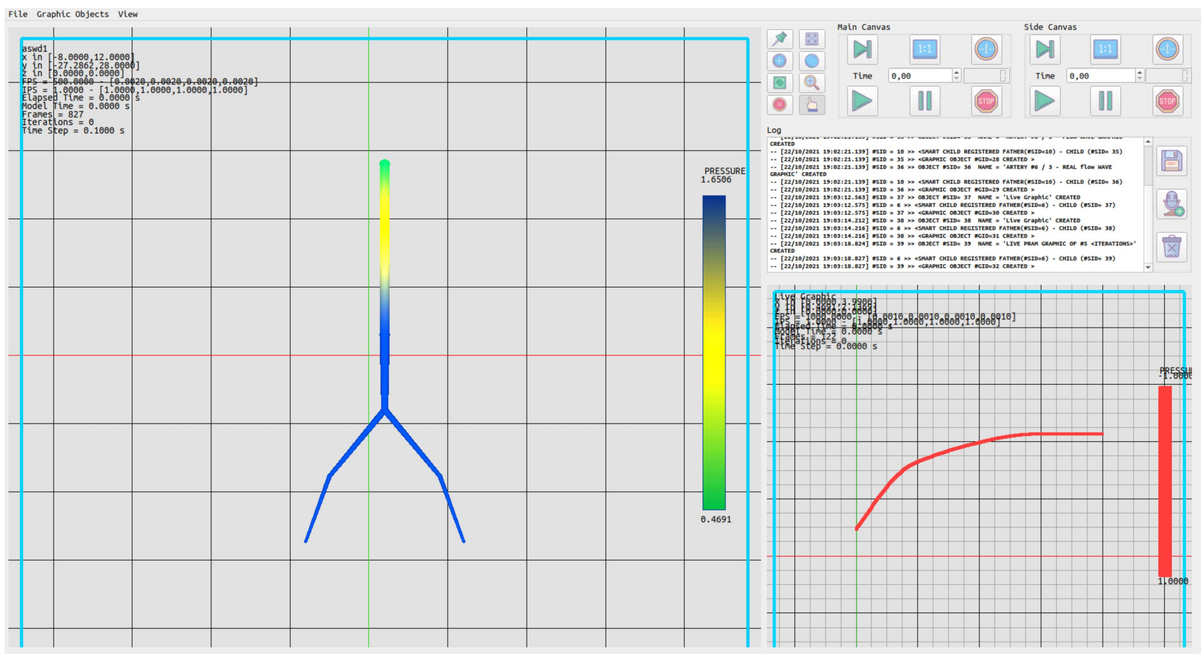


Figura 47 – Ferramenta computacional *IGU0*.

A estrutura de dados do ambiente *IGU0* não apresenta os conceitos de fábrica e é incapaz de armazenar todos os estados do modelo geométrico. Logo, caso se deseje recuperar um estado anterior é necessário reiniciar o experimento.

No ambiente *IGU0*, o processo iterativo dos modelos estava atrelada ao uso da interface de usuário, sem a possibilidade de executar uma sequência de comandos de uma só vez. Portanto, não é possível mensurar claramente o tempo de execução de um trabalho no ambiente *IGU0*, porque o seu processo de configuração requer interação com a interface de usuário. Através dos comandos disponibilizados na Seção **3.3.1**, presentes nos ambientes *IGU* e *InGU*, é possível que objetos inteligentes executem uma bateria de comandos e tenha seu desempenho mensurado e analisado.

Finalmente, como visto na Seção **3.2.7**, a ferramenta (*IGU* e *InGU*) utiliza a *thread WiseProcessor* para processar o objeto *WiseObject*. Este recurso permite que ferramenta execute o ciclo de iteração sem afetas o funcionamento da interface gráfica de usuário ou

bloquear seus processos e funcionalidades.

Para quantificar o desempenho trazido da ferramenta computacional *INGU*, um experimento foi realizado. Este experimento consistiu no cálculo da distribuição da amplitude de pressão ao longo da árvore arterial (ver Figura 39) considerando diferentes parâmetros: (i) viscosidade μ, (ii) ângulo de faze ϕ e (iii) frequência f. Em suma, os três cenários da Seção 4 são aqui contemplados. O experimento foi executado em uma máquina com o processador *Intel Core I9-9900K@3.60GHz* com $4GB@2300MHz$ de memória RAM disponíveis. Para medir o tempo de execução o ambiente computacional *InGU*, por não possuir interface gráfica demonstrou ser o mais rápido durante os testes. O tempo de execução no ambiente *IGU* é em média o dobro do obtido no ambiente *InGU*.

Tabela 49 – Parâmetros de entrada utilizados no teste de carga.

Frequência	$f \in \{3,65; 7,30; 10,95; 14,60\}$
Viscosidade	$\mu \in \{0; 0,5\mu_0; 1,0\mu_0; 1,5\mu_0\}$
Ângulo de Fase	$\phi_0 \in \{0\circ, 4\circ, 8\circ, 12\circ\}$

Na Tabela 49, apresentam-se os parâmetros de entrada utilizados para testar o desempenho da ferramenta *InGU*. O objetivo é realizar 64 iterações combinando os parâmetros de entrada e armazenar o tempo de execução. Cada iteração irá executar o experimento com uma combinação dos três parâmetros de entrada.

O gerenciador de *threads* *WiseThreadPool* é o responsável por gerenciar a quantidade de *threads* utilizada pela ferramenta. Primeiramente, observou-se que a simulação do modelo matemático para um determinado conjunto de parâmetros possui uma média tempo de execução de $30ms$, enquanto uma operação de escrita e leitura demora entre 100 e $300ms$. Como o ciclo de iteração de um objeto *WiseObject* envolve diretamente operações de escrita e leitura, estas foram as *threads* escolhidas para serem duplicadas. *Threads* do tipo *WiseConsole* e *WiseProcessor* requerem uma quantidade maior de trabalhos para que se obtenha algum aumento de desempenho. No experimento escolhido o aumento da quantidade destas *threads* implica diretamente no aumento do tempo de execução devido à comunicação entre elas.

Para executar o teste um arquivo de entrada com uma lista de comandos, assim como o Anexo B, foi confeccionado para criar a estrutura, realizar os cálculos, salvar os resultados e limpar o ambiente. O ambiente computacional *InGU* permite sua execução com parâmetros de entrada. Ao executar o comando "`./INGU read cmds arquivo`"o ambiente computacional executa o comando de leitura de arquivo de entrada e finaliza o programa.

Para que as *threads* aumentem o desempenho, os comando precisam ser divididos em cargas de trabalho. Os comandos recebidos pela ferramenta são executados de forma

sequencial dentro de seus respectivos grupos de trabalho. Utilizando objetos *WiseObject* distintos, grupos de trabalho diferentes podem ser executados simultaneamente. Com isto, o experimento foi dividido em 4 grupos numerados assim $\{1, 2, 4, 8\}$. Para cada grupo, utilizou-se a seguinte quantidade de *threads* $t \in \{1, 2, 4, 8\}$. Como as cargas de trabalho não compartilham objetos *WiseObject*, o uso de mais cargas afeta diretamente a quantidade de memória utilizada pela ferramenta computacional. Os resultados do experimento realizado são apresentados a seguir.

4.2.1 ANÁLISE DE DESEMPENHO DE THREADS

Na Tabela 50 estão registrados os tempos médios de execução do experimento, utilizando 10 amostras.

Tabela 50 – Tempo de execução média em milissegundos (ms) do ciclo de iteração com diferentes arranjos de *threads*, em negrito os melhores tempos.

Cargas de Trabalho	1	2	4	8
1 Thread	7854,4	4475,4	2810,6	2453,6
2 Threads	**5078,8**	**3543,6**	2885,8	2564,4
4 Threads	5232,4	3286	**2239,8**	**1959,6**
8 Threads	7677	4545,4	2887,8	2452,6

Com estes tempos de execução, construiu-se a Tabela 51 contendo o cálculo do ganho de performance ao duplicar o número de *threads*. O ganho de desempenho *speed up* S_th é dado por:

$$S_th = \frac{t_{th-1}}{t_{th}}, \tag{4.1}$$

onde $th \in \{1, 2, 4, 8\}$ indica a quantidade de *threads*.

Tabela 51 – *Speed up* do ciclo de iteração com diferentes arranjos de *threads*, em negrito os maiores aumentos de desempenho.

Cargas de Trabalho	1	2	4	8
2 Threads	**1,54650**	**1,26295**	0,97394	0,95679
4 Threads	0,970641	1,07839	**1,28841**	**1,30863**
8 Threads	0,68156	0,72292	0,77560	0,79898

Ao duplicar o número de *threads* é esperado um *speed up* de 2, entretanto a comunicação entre as *threads* não permite que este número seja alcançado. Após uma quantidade de *threads* este número passa a cair com a adição de novas *threads*. Observa-se

também na Tabela 51 que reorganizar o experimento em cargas de trabalho aumenta a velocidade da execução até mesmo em sistema com apenas uma *thread*.

Os resultados obtidos com 2 e 4 *threads* foram os melhores. Com esta quantidade de *threads* e com o aumento da quantidade de cargas de trabalho é possível melhorar significativamente o desempenho da ferramenta. Com 8 *threads*, a ferramenta apresenta um tempo médio de execução baixo mesmo com uma quantidade maior de trabalhos.

4.2.2 ANÁLISE DE CARGA DE TRABALHO

Com os mesmos resultados de tempo apresentados na Tabela 50, foi calculada a melhora de desempenho pela utilização de uma quantidade maior de cargas de trabalho (w). O ganho de desempenho *speed up* S_w apresentado na Tabela 52 é dado por:

$$S_w = \frac{t_{w-1}}{t_w}, \tag{4.2}$$

onde $w \in \{1, 2, 4, 8\}$ denota a quantidade cargas de trabalho.

Tabela 52 – *Speed up* do ciclo de iteração com diferentes arranjos de cargas de trabalho w, em negrito os melhores aumentos de desempenho.

Cargas de Trabalho	2	4	8
1 Thread	**1,755**	**1,5923**	**1,1455**
2 Threads	1,4332	1,2279	1,1253
4 Threads	1,5923	1,4671	1,143
8 Threads	1,689	1,574	1,1774

Ao duplicar o número de cargas de trabalho w é observada uma melhora expressiva no ambiente executado com uma *thread*, mostrando que a ferramenta é capaz de tirar proveito dos recursos da máquina mesmo que o cenário não disponha de diversos núcleos de processamento disponíveis. Diferentemente do *speed up* calculado no aumento de *threads*, neste caso há sempre uma melhora de desempenho.

5 CONCLUSÕES E TRABALHOS FUTUROS

Este trabalho apresenta uma ferramenta computacional construída para simulação hemodinâmica de modelos de árvores arteriais 1D. Ela contempla o ambiente de interface gráfica (IGU) e de console ($InGU$). Simulações hemodinâmicas e análise de desempenho foram realizadas para verificar o potencial da ferramenta.

Em relação a hemodinâmica, os resultados apresentados na Seção 4.1 estão de acordo com aqueles obtidos por Duan e Zamir[11] considerando a propagação de uma onda harmônica simples nos três cenários abordados nas simulações.

As curvas mostram a ocorrência de picos de pressão ao longo de um modelo árvore arterial. Além disso, como a viscosidade sanguínea, frequência e viscoelasticidade da parede do vaso afetam a onda de pressão.

No tocante ao desempenho computacional, foi calculado ganho de desempenho (*speed up*) tanto levando em conta o número de threads quanto à carga de trabalho. Percebeu-se que a divisão das tarefas em *threads* aumenta o desempenho mas como este aumento também significa um aumento de comunicação entre as *threads*, limitando o ganho de desempenho. Enquanto a divisão de tarefas em cargas de trabalho permite um ganho considerável, mas requer uma quantidade grande de memória disponível.

A ferramenta construída está disponibilizada no repositório de código aberto Bitbucket:

- **Link para Repositório**: **http://bit.ly/2KwZ4np**.

As informações necessárias para compilar a ferramenta computacional e seus ambientes estão inclusas no Apêndice A e na página inicial do repositório.

Como trabalhos futuros, destacam-se:

1. Analisar hemodinamicamente modelos de árvores gerados no contexto do método CCO (*Constrained Constructive Optimization*) [16, 28, 29, 9, 3];
2. Investigar a influência da escolha parâmetros na resposta do modelo matemático de Duan e Zamir, tais como: módulo de Young e espessura do vaso;
3. Analisar o resultado hemodinâmico utilizando um outro esquema iterativo;
4. Calibrar estrutura de *threads* para suportar experimentos maiores e poussir um *overhead* de comunicação menor.

Durante a realização deste trabalho, foi publicado um artigo com resultados obtidos nesta pesquisa [30].

REFERÊNCIAS

1 Khaled Ben Abdessalem and Ridha Ben Saleh. A new formula for predicting the position of severe arterial stenosis. *Comput Methods Biomech Biomed Engin.*, (10):1096–1103, 2017.

2 H Alderson and M Zamir. Effects of stent stiffness on local haemodynamics with particular reference to wave reflections. *J Biomech.*, (10):339–48, 2004.

3 P. F. B. Anjos. *Um algoritmo baseado em otimização para construção de modelos de árvores arteriais com nexo em hemodinâmica computacional.* PhD thesis, Universidade Federal de Juiz de Fora, 2021.

4 M. Anliker, R.L. Rockwell, and E. Odgen. Nonlinear analysis of flow pulses and shock waves in arteries. *ZAMP*, (22):217–246, 1971.

5 L. S. Avila. Vtk user's guide, 2010.

6 A.P. Avolio. Multi-branched model of the human arterial system. *Med. Biol. Eng. Comput*, (18):709–718, 1980.

7 Benajmin Baka. *Getting Started with Qt 5: Introduction to programming Qt 5 for cross-platform application development.* Packt Publishing, 2019.

8 Giulia Bertaglia, Adrián Navas-Montilla, Alessandro Valiani, Manuel Ignacio Monge García, Javier Murillo, and Valerio Caleffi. Computational hemodynamics in arteries with the one-dimensional augmented fluid-structure interaction system: viscoelastic parameters estimation and comparison with in-vivo data. *Journal of Biomechanics*, 100:109595, 2020.

9 P.F. Brito, L.D.M. Meneses, R.W. Santos, and R.A.B Queiroz. Automatic construction of 3d models of arterial tree incorporating the fahraeus-lindqvist effect. *Revista Eletrônica Paulista de Matemática*, (10):38–49, 2017.

10 B. Duan and M. Zamir. Biodynamics: Circulation. *New York: Springer-Verlag*, 1984.

11 B. Duan and M. Zamir. Effect of dispersion of vessel diameters and lengths in stochastic networks. i. modeling of microcirculatory flow. *Microvascular Research*, (31):203–222, 1986.

12 B. Duan and M. Zamir. Pressure peaking in pulsatile flow through arterial tree structures. *Annals of Biomedical Engineering*, (23):794–803, 1995.

13 L. Formaggia, J.F. Gerbeau, F. Nobile, and A. Quarteroni. Computer methods in applied mechanics and engineering. *Revista Eletrônica Paulista de Matemática*, (191):561–582, 2001.

14 L. Formaggia, D. Lamponi, and A. Quarteroni. One-dimensional models for blood flow in arteries. *Journal of Engineering Mathematics*, (47):251–276, 2003.

15 Yuan-cheng Fung. *Biomechanics: mechanical properties of living tissues.* Springer Science and Business Media, 2013.

16 R. Karch, F. Neumann, M. Neumann, and W. Schreiner. A three-dimensional model for arterial tree representation, generated by constrained constructive optimization. *Computers in biology and medicine*, (29):19–38, 1999.

17 G. Karreman. Some contributions to the mathematical biology of blood circulation. reflection of pressure waves in the arterial system. *Bull. Math. Biophys.*, (14):327–350, 1952.

18 Group Khronos. Opengl, 2019.

19 Natalya N. Kizilova. Pulse wave reflections in branching arterial networks and pulse diagnosis methods. *Journal of the Chinese Institute of Engineers*, 26(6):869–880, 2003.

20 Natalya N. Kizilova. Reflection of pulse waves and resonance characteristics of arterial beds. *Fluid Dynamics*, 38(5):1573–8507, 2003.

21 N.T. Kouchoukos, L.C. Sheppard, and D. A. McDonald. Estimation of stroke volume in the dog by a pulse contour method. *Circ. Res.*, (26):611–623, 1970.

22 M. Lighthill. Mathematical biofluidmechanics. *Philadelphia: Society for Industrial and Applied Mathematics*, 1975.

23 R.E. Mates, F.J. Klocke, and J.M. Canty. Coronary capacitance. *Progress in Cardiovascular Diseases*, (31):1–15, 1988.

24 D. A. McDonald. Blood flow in arteries. *Baltimore: Williams and Wilkins*, 1974.

25 Alan Parker. *Algorithms and Data Structures in C++*. Routledge, 2018.

26 C.S. Peskin. Flow patterns around heart valves: a numerical method. *J. Comput. Phys.*, (10):252–271, 1972.

27 Alfio Quarteroni and Luca Formaggia. Mathematical modelling and numerical simulation of the cardiovascular system. In *Computational Models for the Human Body*, volume 12 of *Handbook of Numerical Analysis*, pages 3–127. Elsevier, 2004.

28 R.A.B. Queiroz. *Construção automática de modelos de árvores circulatórias e suas aplicações em hemodinâmica computacional*. PhD thesis, Laboratório Nacional de Computação Científica, 2013.

29 R.A.B. Queiroz, P.J. Blanco, R.A. Feijó, and J.N Ulysses. Ifmbe proceedings. 49ed.: Springer international publishing. *Baltimore: Williams and Wilkins*, pages 884–887, 2015.

30 I. P. Santos, R. F. Reis, and R. A. B. Queiroz. Simulation of pulsatile flow through arterial tree models. *Revista Mundi: Engenharia e Gestão*, 5(6):292–01–292–14, 2020.

31 N. Stergiopulos, D.F. Young, and T. R. Rogge. Computer simulation of arterial flow with applications to arterial and aortic stenoses. *Journal of Biomechanics*, (25):1477–1488, 1992.

32 Takamichi Takahashi, Hirofumi Tomiyama, Victor Aboyans, Kento Kumai, Hiroki Nakano, Masatsune Fujii, Kazuki Shiina, Chisa Matsumoto, Akira Yamashina, and Taishiro Chikamori. Association of pulse wave velocity and pressure wave reflection with the ankle-brachial pressure index in japanese men not suffering from peripheral artery disease. *Atherosclerosis*, 317:29–35, 2021.

33 C.A. Taylor, T.J.R. Hughes, and C.K. Zarins. Finite element modeling of three-dimensional pulsatile flow in abdominal aorta: relevance to atherosclerosis. *Annals of Biomedical Engineering*, (26):975–987, 1998.

34 M.G. Taylor. The input impedance of an assembly of randomly branching elastic tubes. *Biophys. J*, (6):29–51, 1966.

35 Carl-Johan Thore. Pressure estimation in the systemic arteries using a transfer function, 2007.

36 L. Welicki, J. W. Yoder, and R. Wirfs-Brock. The dynamic factory pattern, 2008.

37 J.R. Womersley. Xxiv. oscillatory motion of a viscous liquid in a thin-walled elastic tube—i: The linear approximation for long waves. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, 46(373):199–221, 1955.

38 M. Zamir. Optimality principles in arterial branching. *J. Theor. Biol.*, (62):227–251, 1976.

39 M Zamir. Mechanics of blood supply to the heart: wave reflection effects in a right coronary artery. *Proceedings of the Royal Society*, 265(1394):439–444, 1998.

40 M. Zamir and S. Phipps. Network analysis of an arterial tree. *American Journal of Physiology*, (21):25–34, 1988.

A PROCESSO DE COMPILAÇÃO

A.1 INSTALAR QT

Rodar os comandos para baixar o arquivo da versão da biblioteca Qt e para efetuar sua instalação.

- wget http://download.qt.io/official_releases/qt/5.15/5.15.0/qt-opensource-linux-x64-5.15.0.run
- chmod +x qt-opensource-linux-x64-5.15.0.run
- ./qt-opensource-linux-x64-5.15.0.run

Uma outra alternativa são os comandos para buscar as mesma bibliotecas via *apt-get*:

- sudo apt-get install qtbase5-dev
- sudo apt-get install qtdeclarative5-dev

Instalar a biblioteca *Qt* de acordo com as instruções dadas pelo guia de instalação. Depois, instale *Qt Creator* caso se deseje alterar o código-fonte.

A.2 INSTALAR BIBLIOTECAS SECUNDÁRIAS

Outras bibliotecas menores também são necessárias para compilar corretamente a ferramenta computacional, os comandos abaixo instalam todas elas:

- sudo apt-get install build-essential
- sudo apt-get install libfontconfig1
- sudo apt-get install mesa-common-dev
- sudo apt-get install libglu1-mesa-dev -y
- sudo apt-get install freeglut3-dev

A.3 CLONAR E COMPILAR REPOSITÓRIO

O repositório pode ser clonado com o seguinte comando:

git clone https://MrBlackPower@bitbucket.org/MrBlackPower/igu.git

Para compilar o projeto corretamente através da *IDE QtCreator*, as seguintes instruções devem ser seguidas:

1. Abrir pasta do projeto *Project*
    - Selecionar arquivo CMakeLists.txt;
    - Configurar o projeto para ser compilado;
    - Escolher o ambiente computacional desejado e rodar o projeto.

Caso apareceça a mensagem *cannot find -lGL*, rode os seguintes comandos:

- sudo rm /usr/lib/x86_64-linux-gnu/libGL.so
- sudo ln -s /usr/lib/libGL.so.1 /usr/lib/x86_64-linux-gnu/libGL.so

B FORMATO DE ARQUIVO DE COMANDOS

Abaixo, seguem as chamadas para realização dos cálculos hemodinâmicos do modelo de árvore arterial ($\mathcal{MAA}$) em comandos que a ferramenta computacional é capaz de processar.

```
object create artery_tree DUAN_AND_ZAMIR obj_t5 el_t5
object iteration_factories set obj_t5 DUAN_AND_ZAMIR
object set obj_t5
object set_field obj_t5 FIELD FREQUENCY 0 3.65
object set_field obj_t5 FIELD ARTERY_CHOOSE_MODE 0 HIGHEST
object go obj_t5
object create graphic BLANK obj_t6 el_t6
object iteration_factories set obj_t6 OBJECT_READ
object set obj_t6
object set_field obj_t6 FIELD READ_ELEMENT_NAME 0 obj_t5
object set_field obj_t6 FIELD READ_FIELD 0 PRESSURE_WAVE
object set_field obj_t6 FIELD READ_CELL_TYPE 0 FIELD
object go obj_t6
object export obj_t6 IMAGE_PNG pressure_3_65.png
object set_field obj_t6 FIELD READ_FIELD 0 FLOW_WAVE
object go obj_t6
object export obj_t6 IMAGE_PNG flow_3_65.png
object set_field obj_t5 FIELD FREQUENCY 0 7.30
object go obj_t5
object set_field obj_t6 FIELD READ_FIELD 0 PRESSURE_WAVE
object set_field obj_t6 FIELD READ_CELL_TYPE 0 FIELD
object go obj_t6
object export obj_t6 IMAGE_PNG pressure_7_30.png
object set_field obj_t6 FIELD READ_FIELD 0 FLOW_WAVE
object go obj_t6
```

```
1 object export obj_t6 IMAGE_PNG flow_7_30.png
2 object set_field obj_t5 FIELD FREQUENCY 0 10.95
3 object go obj_t5
4 object set_field obj_t6 FIELD READ_FIELD 0 PRESSURE_WAVE
5 object set_field obj_t6 FIELD READ_CELL_TYPE 0 FIELD
6 object go obj_t6
7 object export obj_t6 IMAGE_PNG pressure_10_95.png
8 object set_field obj_t6 FIELD READ_FIELD 0 FLOW_WAVE
9 object go obj_t6
10 object export obj_t6 IMAGE_PNG flow_10_95.png
11 object set_field obj_t5 FIELD FREQUENCY 0 14.60
12 object go obj_t5
13 object set_field obj_t6 FIELD READ_FIELD 0 PRESSURE_WAVE
14 object set_field obj_t6 FIELD READ_CELL_TYPE 0 FIELD
15 object go obj_t6
16 object export obj_t6 IMAGE_PNG pressure_14_60.png
17 object set_field obj_t6 FIELD READ_FIELD 0 FLOW_WAVE
18 object go obj_t6
19 object export obj_t6 IMAGE_PNG flow_14_60.png
20 object delete obj_t6
21 object delete obj_t5
```

C FORMATO DE ARQUIVO DE ELEMENTO

```
1  <WISE_ELEMENT>
2    <!-- XML for Wise Element el1 -->
3    <ELEMENT_PARAMS NAME="el1" FILENAME_VTK="wise_el1_ACJHBFGCGG_0_0.vtk"
         FILENAME_XML="wise_el1_ACJHBFGCGG_0_0.xml" TYPE="GRAPHIC"
         ELEMENT_KEY="ACJHBFGCGG" ID="0" DT="1.000000" MODEL_TIME="0.000000"
         INSTANCE="0" STATUS="HOT" />
4    <WISE_STRUCTURE>
5      <!-- XML for Wise Object el1 -->
6      <STRUCTURE_PARAMETERS NAME="el1" STRUCTURE_ID="0" STRUCTURE_KEY="
           DGJJGJBJAH" N_POINTS="101" N_CELLS="0" N_LINES="0" N_FIELDS="0"
           MAX_X="360" MAX_Y="100" MAX_Z="0" MIN_X="0" MIN_Y="-1" MIN_Z="0"
           IMPORTED="0" IMPORT_TYPE="3" IMPORT_FILENAME="UNTITLED" />
7      <STRUCTURE>
8        <POINTS>
9        <POINT ID="0" X="0.0000000000" Y="0.0000000000" Z="0.0000000000" />
10        ...
11       <POINT ID="100" X="360.0000000000" Y="0.0000000000" Z="0.0000000000
             " />
12       </POINTS>
13       <LINES />
14       <CELLS />
15     </STRUCTURE>
16     <DATA>
17       <POINT_DATA>
18         <INFO NAME="VALUE" TYPE="DOUBLE">
19         <DATA VALUE="0.0000000000" />
20          ...
21         <DATA VALUE="0.0000000000" />
22         </INFO>
23       </POINT_DATA>
24       <CELL_DATA />
25       <LINE_DATA />
26       <FIELD />
27     </DATA>
28   </WISE_STRUCTURE>
29 </WISE_ELEMENT
```

D FORMATO DE ARQUIVO DE OBJETO

```
1  <WISE_OBJECT>
2  <!-- XML for Wise Object obj1 -->
3  <OBJECT_PARAMS NAME="obj1" STATUS="GO" TYPE="ARTERY_TREE" DT="1.000000"
       MODEL_TIME="0.000000" FRAMES="0" ITERATIONS="0" ITERATION_FACTORY_SET=
       "1" ITERATION_FACTORY="DUAN_AND_ZAMIR" GRAPHIC_FACTORY_SET="1"
       GRAPHIC_FACTORY="GRAPHIC_ARTERY_TREE_FACTORY" GRAPHIC_MODEL_SET="1" />
4  <WISE_COLLECTION>
5  <COLLECTION_PARAMS ID="0" N="13" />
6  <WISE_OVEN>
7  <WISE_ELEMENT>
8  <!-- XML for Wise Element el1 -->
9  <ELEMENT_PARAMS NAME="el1" FILENAME_VTK="wise_el1_JGCAHIEHDD_0_0.vtk"
       FILENAME_XML="wise_el1_JGCAHIEHDD_0_0.xml" TYPE="ARTERY_TREE"
       ELEMENT_KEY="JGCAHIEHDD" ID="0" DT="1.000000" MODEL_TIME="12.000000"
       ITERATION="0" STATUS="HOT" />
10 <WISE_STRUCTURE>
11 <!-- XML for Wise Object el1 -->
12 <STRUCTURE_PARAMETERS NAME="el1" STRUCTURE_ID="0" STRUCTURE_KEY="
       HDAGBBBCJC" N_POINTS="7" N_CELLS="0" N_LINES="0" N_FIELDS="0" MAX_X="
       12" MAX_Y="28" MAX_Z="0" MIN_X="-8" MIN_Y="-27" MIN_Z="0" IMPORTED="0"
        IMPORT_TYPE="3" IMPORT_FILENAME="UNTITLED" />
13 <STRUCTURE>
14 <POINTS>
15 <POINT ID="0" X="2.0000000000" Y="28.0000000000" Z="0.0000000000" />
16 <POINT ID="1" X="2.0000000000" Y="3.0000000000" Z="0.0000000000" />
17 <POINT ID="2" X="2.0000000000" Y="-8.0000000000" Z="0.0000000000" />
18 <POINT ID="3" X="-5.0000000000" Y="-17.7467940000" Z="0.0000000000" />
19 <POINT ID="4" X="-8.0000000000" Y="-27.2861860000" Z="0.0000000000" />
20 <POINT ID="5" X="9.0000000000" Y="-17.7467940000" Z="0.0000000000" />
21 <POINT ID="6" X="12.0000000000" Y="-27.2861860000" Z="0.0000000000" />
22 </POINTS>
23 <LINES>
24 <LINE ID="0" A="0" B="1" />
25 <LINE ID="1" A="1" B="2" />
26 <LINE ID="2" A="2" B="3" />
27 <LINE ID="3" A="3" B="4" />
28 <LINE ID="4" A="2" B="5" />
29 <LINE ID="5" A="5" B="6" />
30 </LINES>
31 <CELLS />
32 </STRUCTURE>
33 <DATA>
34 <POINT_DATA />
35 <CELL_DATA>
36 <INFO NAME="RADIUS" TYPE="DOUBLE">
```

```
37 <DATA VALUE="0.6500000000" />
38 ...
39 <DATA VALUE="0.2000000000" />
40 </INFO>
41 <INFO NAME="DENSITY" TYPE="DOUBLE">
42 <DATA VALUE="0.9600000000" />
43 ...
44 <DATA VALUE="1.2350000000" />
45 </INFO>
46 <INFO NAME="YOUNG_MODULUS" TYPE="DOUBLE">
47 <DATA VALUE="4800000.0000000000" />
48 ...
49 <DATA VALUE="10000000.0000000000" />
50 </INFO>
51 <INFO NAME="VISCOSITY" TYPE="DOUBLE">
52 <DATA VALUE="0.0385000000" />
53 ...
54 <DATA VALUE="0.0494000000" />
55 </INFO>
56 <INFO NAME="ALPHA" TYPE="DOUBLE">
57 <DATA VALUE="-1.0000000000" />
58 ...
59 <DATA VALUE="-1.0000000000" />
60 </INFO>
61 <INFO NAME="WAVE_SPEED" TYPE="DOUBLE">
62 <DATA VALUE="500.0000000000" />
63 ...
64 <DATA VALUE="636.2847629758" />
65 </INFO>
66 <INFO NAME="WALL_THICKNESS" TYPE="DOUBLE">
67 <DATA VALUE="0.0650000000" />
68 ...
69 <DATA VALUE="0.0200000000" />
70 </INFO>
71 <INFO NAME="LENGTH" TYPE="DOUBLE">
72 <DATA VALUE="25.0000000000" />
73 ...
74 <DATA VALUE="9.9999999865" />
75 </INFO>
76 <INFO NAME="PHI" TYPE="DOUBLE">
77 <DATA VALUE="0.0000000000" />
78 ...
79 <DATA VALUE="0.0000000000" />
80 </INFO>
81 <INFO NAME="ANGULAR_FREQUENCY" TYPE="DOUBLE">
82 <DATA VALUE="86.0010988920" />
83 ...
```

```
84  <DATA VALUE="86.0010988920" />
85  </INFO>
86  <INFO NAME="BETA_ABS" TYPE="DOUBLE">
87  <DATA VALUE="4.3000549446" />
88  ...
89  <DATA VALUE="1.3516133621" />
90  </INFO>
91  <INFO NAME="BETA_REAL" TYPE="DOUBLE">
92  <DATA VALUE="4.3000549446" />
93  ...
94  <DATA VALUE="1.3516133621" />
95  </INFO>
96  <INFO NAME="BETA_IMAGE" TYPE="DOUBLE">
97  <DATA VALUE="0.0000000000" />
98  ...
99  <DATA VALUE="0.0000000000" />
100 </INFO>
101 <INFO NAME="VISCOUS_WAVE_SPEED_ABS" TYPE="DOUBLE">
102 <DATA VALUE="1.0000000000" />
103 ...
104 <DATA VALUE="1.0000000000" />
105 </INFO>
106 <INFO NAME="VISCOUS_WAVE_SPEED_REAL" TYPE="DOUBLE">
107 <DATA VALUE="-1.0000000000" />
108 ...
109 <DATA VALUE="-1.0000000000" />
110 </INFO>
111 <INFO NAME="VISCOUS_WAVE_SPEED_IMAGE" TYPE="DOUBLE">
112 <DATA VALUE="0.0000000000" />
113 ...
114 <DATA VALUE="0.0000000000" />
115 </INFO>
116 <INFO NAME="VISCOUS_YOUNG_MODULUS_ABS" TYPE="DOUBLE">
117 <DATA VALUE="1.0000000000" />
118 ...
119 <DATA VALUE="1.0000000000" />
120 </INFO>
121 <INFO NAME="VISCOUS_YOUNG_MODULUS_REAL" TYPE="DOUBLE">
122 <DATA VALUE="-1.0000000000" />
123 ...
124 <DATA VALUE="-1.0000000000" />
125 </INFO>
126 <INFO NAME="VISCOUS_YOUNG_MODULUS_IMAGE" TYPE="DOUBLE">
127 <DATA VALUE="0.0000000000" />
128 ...
129 <DATA VALUE="0.0000000000" />
130 </INFO>
```

```
131 <INFO NAME="FLOW_ABS" TYPE="DOUBLE">
132 <DATA VALUE="0.2712788317" />
133 ...
134 <DATA VALUE="0.1154357112" />
135 </INFO>
136 <INFO NAME="FLOW_REAL" TYPE="DOUBLE">
137 <DATA VALUE="−0.0642456722" />
138 ...
139 <DATA VALUE="−0.0862382822" />
140 </INFO>
141 <INFO NAME="FLOW_IMAGE" TYPE="DOUBLE">
142 <DATA VALUE="0.2635615642" />
143 ...
144 <DATA VALUE="−0.0767356639" />
145 </INFO>
146 <INFO NAME="FLOW_WAVE" TYPE="VECTOR">
147 <DATA VALUE="0.0043418095 0.0042507071 0.0041518920 0.0040455728
        0.0039319766 0.0038113495 0.0036839567 0.0035500842 0.0034100397
        0.0032641549 0.0031127882 0.0029563287 0.0027952022 0.0026298798
        0.0024608902 0.0022888384 0.0021144345 0.0019385376 0.0017622264
        0.0015869140 0.0014145382 0.0012478823 0.0010911088 0.0009505832
        0.0008358491 0.0007597172 0.0007349176 0.0007666319 0.0008483353
        0.0009669300 0.0011099295 0.0012682282 0.0014357897 0.0016086623
        0.0017841919 0.0019605202 0.0021362844 0.0023104383 0.0024821444
        0.0026507080 0.0028155347 0.0029761038 0.0031319507 0.0032826544
        0.0034278298 0.0035671222 0.0037002030 0.0038267672 0.0039465316
        0.0040592334 0.0041646288 0.0042624931 0.0043526199 0.0044348206
        0.0045089248 0.0045747794 0.0046322496 0.0046812177 0.0047215841
        0.0047532669 0.0047762019 0.0047903426 0.0047956605 0.0047921449
        0.0047798028 0.0047586593 0.0047287569 0.0046901564 0.0046429360
        0.0045871919 0.0045230378 0.0044506052 0.0043700434 0.0042815193
        0.0041852177 0.0040813415 0.0039701115 0.0038517676 0.0037265686
        0.0035947936 0.0034567429 0.0033127400 0.0031631342 0.0030083040
        0.0028486623 0.0026846642 0.0025168177 0.0023457001 0.0021719832
        0.0019964706 0.0018201591 0.0016443352 0.0014707365 0.0013018221
        0.0011412251 0.0009944780 0.0008699883 0.0007796313 0.0007368490
        0.0007501554" />
148 ...
149 </WISE_OBJECT>
```