ADRIANO DA SILVA FERREIRA

DESENVOLVIMENTO DE UM AMBIENTE COMPUTACIONAL PARA UM SIMULADOR ELETROMAGNÉTICO BASEADO NO MÉTODO FDTD

CAMPINAS
2013

UNIVERSIDADE ESTADUAL DE CAMPINAS
FACULDADE DE ENGENHARIA ELÉTRICA E DE COMPUTAÇÃO

ADRIANO DA SILVA FERREIRA

DESENVOLVIMENTO DE UM AMBIENTE COMPUTACIONAL PARA UM SIMULADOR ELETROMAGNÉTICO BASEADO NO MÉTODO FDTD

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Engenharia Elétrica da Faculdade de Engenharia Elétrica e de Computação da Universidade Estadual de Campinas para obtenção do título de Mestre em Engenharia Elétrica, na área de Telecomunicações e Telemática.

Orientador: PROF. DR. HUGO ENRIQUE HERNANDEZ FIGUEROA
Co-orientador: PROFa. DRa. MARLI DE FREITAS GOMES HERNANDEZ

ESTE EXEMPLAR CORRESPONDE À VERSÃO FINAL DA DISSERTAÇÃO
DEFENDIDA PELO ALUNO ADRIANO DA SILVA FERREIRA
E ORIENTADO PELO PROF. DR. HUGO ENRIQUE HERNANDEZ FIGUEROA

Assinatura do Orientador

_____

CAMPINAS
2013

Ficha catalográfica
Universidade Estadual de Campinas
Biblioteca da Área de Engenharia e Arquitetura
Luciana Pietrosanto Milla - CRB 8/8129

F413d

Ferreira, Adriano da Silva, 1984-
Desenvolvimento de um ambiente computacional para um simulador eletromagnético baseado no método FDTD / Adriano da Silva Ferreira. – Campinas, SP : [s.n.], 2013.

Orientador: Hugo Enrique Hernandez Figueroa.
Coorientador: Marli de Freitas Gomes Hernandez.
Dissertação (mestrado) – Universidade Estadual de Campinas, Faculdade de Engenharia Elétrica e de Computação.

1. Eletromagnetismo - Computação. 2. Diferenças finitas. 3. Computação gráfica. 4. Visualização de informação. 5. Software livre. I. Hernandez Figueroa, Hugo Enrique. II. Hernandez, Marli de Freitas Gomes,1959-. III. Universidade Estadual de Campinas. Faculdade de Engenharia Elétrica e de Computação. IV. Título.

Informações para Biblioteca Digital

Título em outro idioma: Development of a computational environment for an electromagnetic simulator based on FDTD method
Palavras-chave em inglês:
Eletromagnetism - Computing
Finite differences
Computer graphics
Information visualization
Free software
Área de concentração: Telecomunicações e Telemática
Titulação: Mestre em Engenharia Elétrica
Banca examinadora:
Hugo Enrique Hernandez Figueroa [Orientador]
Bernardo de Barros Correira Kyotoku
Lucas Heitzmann Gabrielli
Data de defesa: 06-12-2013
Programa de Pós-Graduação: Engenharia Elétrica

COMISSÃO JULGADORA - TESE DE MESTRADO

Candidato: Adriano da Silva Ferreira

Data da Defesa: 6 de dezembro de 2013

Título da Tese: "Desenvolvimento de um Ambiente Computacional para um Simulador Eletromagnético Baseado no Método FDTD"

Prof. Dr. Hugo Enrique Hernandez Figueroa (Presidente): _____

Dr. Bernardo de Barros Correia Kyotoku: _____

Dr. Lucas Heitzmann Gabrielli: _____

RESUMO

Este trabalho tem por objetivo desenvolver um ambiente computacional livre para o *software* MEEP (MIT *Electromagnetic Equation Propagation*), um simulador eletromagnético de código aberto baseado no método das Diferenças Finitas no Domínio do Tempo (*Finite-Difference Time-Domain* – FDTD).

Este ambiente computacional foi implementado sob o paradigma de Programação Orientada à Objetos, através da linguagem de programação Java, e estruturado em Pré-processamento (configuração do cenário da simulação eletromagnética), Processamento (aplicação do FDTD no cenário configurado) e Pós-processamento (análise e visualização dos resultados da simulação), com o propósito de prover novas funcionalidades para o Pré-processamento e o Pós-processamento do MEEP.

Através do VTK (*Visualization Toolkit*), um *software* de Visualização e Computação Gráfica Tridimensional, buscou-se implementar requisitos geométricos e gráficos ausentes no Pré-processamento do MEEP com o desenvolvimento de funcionalidades para leitura de arquivos gráficos e construção de modelos geométricos de cristais fotônicos pré-definidos, e com o desenvolvimento de mecanismos eficientes de geração, interação e visualização de objetos gráficos. Utilizou-se, do VTK, funcionalidades para leituras de arquivos STL (*Standard Tessellation Language*) e *Wavefront* OBJ, ambos fornecedores de geometrias, e implementou-se um importador de arquivos CAD (*Computer-Aided Design*), de formato DXF (*Drawing Exchange Format*), que fornece dados de geometrias e de materiais.

O objetivo é prover uma interface gráfica de usuário para possibilitar e facilitar, através da inserção automática e visualização de objetos gráficos, a configuração de cenários mais elaborados nas simulações eletromagnéticas, em oposição ao mecanismo de *scripting* em linguagens de programação específicas disponíveis nas versões atuais do MEEP.

Após executar o estágio de Processamento, o MEEP gera dois arquivos de saída, ambos em formato HDF5, que representam a geometria discretizada e o resultado dos

cálculos da simulação. *Softwares* como MATLAB®, HDFView ou H5utils (ferramenta livre desenvolvida pelo próprio grupo do MEEP) podem manipular arquivos HDF5. Entretanto, essas ferramentas carecem de funcionalidades que permitem análises mais elaboradas e detalhadas em Pós-processamento, oferecendo mecanismos muito simples e limitados para a realização de cortes em geometrias e animação de simulações. Neste sentido, objetiva-se, com este ambiente computacional, contribuir com um Pós-processamento capaz de realizar cortes através de planos posicionados, arbitrariamente, em geometrias tridimensionais simuladas e gerar animações bidimensionais de propagação de ondas eletromagnéticas em simulações bidimensionais e tridimensionais, tomando, como base, os *softwares* H5utils e VTK.

ABSTRACT

This work aims the development of a free computational environment for the software MEEP (MIT Electromagnetic Equation Propagation), an open-source electromagnetic simulator based on Finite-Difference Time-Domain (FDTD) method.

This computational environment has been implemented under the Object Oriented Programming paradigm using Java programming language, and it is structured in Pre-processing (configuration of electromagnetic simulation scenario), Processing (execution of the FDTD on defined scenario) and Post-processing (analysis and visualization of simulation results), with the purpose of providing new functionalities of Pre-processing and Post-processing for the MEEP.

Through the VTK (Visualization Toolkit), a 3D Visualization and Computer Graphics software system, we have implemented geometrical and graphical requirements missing in MEEP´s Pre-processing with the development of features for reading graphics files and for building of geometrical pre-defined photonic crystals models, and with the development of efficient mechanisms for generation, interaction and displaying of graphics objects. We used VTK features for reading STL (Standard Tessellation Language) and Wavefront OBJ files, both suppliers of geometries, and we implemented a CAD file importer (Computer-Aided Design), based on DXF format (Drawing Exchange Format), which provides geometrical and material data.

The goal is to provide a graphical user interface to enable and facilitate, through automatic insertion and visualization of graphical objects, the configuration of more elaborate scenarios in electromagnetic simulations, as opposed to the scripting engine in specific programming languages available in the current versions of MEEP.

After executing the Processing stage, MEEP generates two output files, both in HDF5 format, which represent the discretized geometry and the calculation result of the simulation. Softwares such as MATLAB®, HDFView or H5utils (free tool developed by the MEEP´s group) can manipulate HDF5 files. However, these tools lack features that allow more elaborate and detailed analysis in Post-processing, offering very simple and

limited mechanisms for making cuts in geometries and animation simulations. In this sense, the objective is, with this computing environment, to contribute with a Post-processing system able to perform cuts through arbitrary plans in three-dimensional simulated geometries and to generate two-dimensional animations of propagation of electromagnetic waves in two and three-dimensional simulations, based on H5utils and VTK softwares.

SUMÁRIO

Dedico este trabalho à minha mãe, Neusa, e ao meu pai, Valdecir, que sempre acreditaram nas minhas escolhas e sempre foram fundamentais para a realização desta caminhada.

AGRADECIMENTOS

Agradeço o professor Hugo pela liberdade dada ao desenvolvimento das idéias e do trabalho e pelos desafios propostos; à orientação objetiva e, acima de tudo, a oportunidade dada.

Agradeço a professora Marli, que acompanhou e orientou meus projetos desde a graduação e que foi fundamental para a concretização deste trabalho.

Agradeço meu pai, Valdecir, e minha mãe Neusa, que sempre foram meu alicerce, e meus irmãos Alex e Daniela por todo apoio dado. Agradeço meus sobrinhos Pietra, Lorenzo e Davi, e nosso pequeno cachorro Tigrão (*in memorian*), pela inspiração.

Agradeço a Jéssica Patrícia, que sempre me apoiou e me ajudou no que foi possível, seja fornecendo um lugar para pousar em sua república – nas noite longas de laboratório – seja no companheirismo constante.

Agradeço os amigos de laboratório do DECOM, em especial, Igor, Carlos Henrique, Maicon, Zady, Roger e Angelo, pelas sugestões e pela contribuição direta no desenvolvimento deste trabalho.

Agradeço os amigos Ralf, Adenilton, Daniel e Estêvão que, pelo simples bate-papo e curiosidade pelo trabalho, fizeram surgir dúvidas e idéias que foram aplicadas neste projeto. Sou grato, também, aos amigos de banda Marco Aurélio, Rodrigo e Edmur, que a cada ensaio ajudaram a descontrair, com a música, os momentos mais difíceis.

Por fim, agradeço o CNPq pelo apoio financeiro.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

API	*Application Programming Interface*
ASCII	*American Standard Code for Information Interchange*
AVI	*Audio Video Interleave*
AWT	*Abstract Window Toolkit*
CAD	*Computer-Aided Design*
CEPOF	Centro de Pesquisa em Óptica e Fotônica
CPqD	Centro de Pesquisa e Desenvolvimento em Telecomunicações
CST	*Computer Simulation Technology*
DAO	*Data Access Object*
DECOM	Departamento de Comunicações
DXF	*Drawing Exchange Format*
FDTD	*Finite-Difference Time-Domain*
FEEC	Faculdade de Engenharia Elétrica e de Computação
FEM	*Finite Element Method*
FOTONICOM	INCT de Fotônica para Comunicações Ópticas
FPS	*Frames per Second*
GDS	*Graphic Database System*
GPL	*General Public License*
GPU	*Graphics Processing Unit*
GUI	*Graphical User Interface*
HDF	*Hierarchical Data Format*
HSVA	*Hue-Saturation-Value-Alpha*
INCT	Institutos Nacionais de Ciência e Tecnologia
JPEG	*Joint Photographic Experts Group*
MEEP	MIT *Electromagnetic Equation Propagation*
MIT	*Massachusetts Institute of Technology*
MPB	MIT *Photonic Bands*

MMTL	*Multilayer Multiconductor Transmission Line*
MPEG	*Moving Picture Experts Group*
MPI	*Message Passing Interface*
NCSA	*National Center for Supercomputing Applications*
OBJ	*Object*
OPC	*Optimized Photonic Crystal*
PCF	*Photonic Crystal Fiber*
PML	*Perfectly Matched Layer*
PNG	*Portable Network Graphics*
RGBA	*Red-Green-Blue-Alpha*
STL	*Stereo Lithography*
TCL	*Tool Command Language*
TXT	*Text*
UCS	*User Coordinate System*
UML	*Unified Modeling Language*
UNICAMP	Universidade Estadual de Campinas
VTK	*Visualization Toolkit*
XML	*Extensible Markup Language*

CAPÍTULO 1

INTRODUÇÃO

1.1 CONSIDERAÇÕES INICIAIS

Nos dias atuais, diversas áreas do conhecimento fazem uso de recursos computacionais para possibilitar, viabilizar, facilitar e manter o desenvolvimento de estudos, pesquisas, ferramentas e produtos. A diversidade de recursos computacionais é extremamente ampla e estende-se desde o suporte físico, baseado em *hardware*, à técnicas avançadas de análise, processamento, comunicação e reconhecimento de padrões em dados, baseadas em programas de computador.

Não diferente, as áreas das ciências físicas e da engenharia utilizam suporte constante de mecanismos computacionais para realização de estudos inerentes às suas áreas. Dentre as diversas frentes de estudos, destaca-se a necessidade de pesquisas em fenômenos não-observáveis pelo olho humano e que tem profundo impacto na obtenção de uma ciência consistente dos mesmos. Além do fato de não serem observáveis, esses fenômenos podem apresentar natureza complexa, de forma que a realização de qualquer pesquisa, representação ou reprodução desses seja inviável através de mecanismos puramente humanos.

Neste contexto, a simulação computacional oferece um grande potencial para concretizar os diversos trabalhos referentes à reprodução desses fenômenos, efetuando os cálculos que os regem em tempo finito ou, em certos casos, em tempo real, e tornando-os observáveis aos humanos.

O MEEP [1], que é o *software* base deste trabalho, está de acordo com essa contextualização. Seu propósito é modelar sistemas eletromagnéticos através de seu pacote de simulação. Em outras palavras, o MEEP busca obter soluções para as equações diferenciais parciais de Maxwell, com o propósito de investigar o fenômeno de guiamento

de ondas eletromagnéticas [2].

Para obtenção das soluções das equações de Maxwell, o MEEP implementa o método numérico FDTD, introduzido por Yee em 1966 [3]. Este método caracteriza-se por dividir ou discretizar o espaço e o tempo em grades regulares para obtenção das soluções das equações de Maxwell no domínio do tempo [1, 2, 3, 4, 5].

Outros algoritmos ou técnicas numéricas mais sofisticadas, como o FEM (Método dos Elementos Finitos) [6] podem ser empregados no eletromagnetismo computacional. Uma discussão sobre quais métodos são melhores para simulação eletromagnética não é abordada neste trabalho, embora seja possível afirmar que nenhuma técnica é ótima para todos os problemas e que os métodos numéricos disponíveis no eletromagnetismo computacional possuem suas vantagens e desvantagens. O FDTD é descrito como um método simples, robusto e de propósito geral, além de ser altamente paralelizável para execução em *clusters* de computadores ou supercomputadores [1, 7], o que torna-o interessante para resolução de problemas grandes. Dentre suas desvantagens, destaca-se a aproximação simplória da solução ou da discretização em relação à outros métodos, como o FEM [8].

Existem diversos programas comerciais que suportam o método FDTD, mas que possuem alto custo de aquisição. Neste contexto, o MEEP é vantajoso por ser um programa livre e que, portanto, pode ser utilizado sem nenhum custo. Além disso, é um sistema flexível, pois possui código-fonte aberto, o que possibilita sua modificação e, sobretudo, a extensão de suas funcionalidades. Esse aspecto é particularmente interessante, uma vez que as pesquisas frequentemente demandam a flexibilidade presente em um código-fonte aberto [1]. O MEEP pode ser usado conforme a licença GNU-GPL (Licença Pública Geral) [9].

Para operar o MEEP, o usuário do sistema deve escrever um programa arbitrário que controle a simulação [1]. Primordialmente, o pacote de simulação ofereceu dois mecanismos de operação: uma interface de baixo nível na linguagem de programação C++ e uma interface de alto nível, baseada em *scripting* na linguagem funcional *Scheme*, implementada pela biblioteca externa GNU *Guile* [10]. Posteriormente, realizou-se um novo esforço para implementar uma interface entre o MEEP e a linguagem de programação

Python [11], com o propósito de integrar o simulador ao ecossistema dessa linguagem e usufruir de suas bibliotecas, sobretudo as de visualização. O projeto do MEEP para C++ e *Scheme* pode ser acessado em [12] e o do MEEP-Python em [13].

Evidentemente, existem diferenças entre todas as versões do MEEP. A versão em C++ é a de nível de programação mais baixo, pois há a necessidade de se desenvolver mais funções para operação do simulador. Um bom exemplo é a necessidade de implementação das funções que definem as geometrias e seus respectivos materiais, que são responsáveis pelo fornecimento das características eletromagnéticas contidas no cenário de simulação. Além disso, o usuário deverá se preocupar com os mecanismos de baixo nível do C++, principalmente aqueles que envolvem a manipulação de endereçamentos de memória. Por outro lado, a utilização do MEEP na versão em C++ é mais flexível, pois esta linguagem oferece os mecanismos de extensão de funcionalidades através do paradigma de Programação Orientada à Objetos [14, 15], além do usuário ter acesso direto à interface exposta pelo núcleo do sistema MEEP.

Na versão em *Scheme*, a simulação é definida em termos de expressões de alto nível [11], eliminando a preocupação com detalhes de programação e sintaxe presentes no C++. As funções ou *scripts*, de aspecto intuitivo e amigável ao usuário, garantem acessibilidade aos meios de operação do MEEP e disponibilizam cinco geometrias pré-definidas para utilização na simulação: esfera, cilindro, cone, bloco e elipsoide [16], que podem ser combinadas entre si para formar geometrias mais complexas. Como este tipo de linguagem funcional propõe, o conjunto de expressões do programa de simulação aproxima-se de um texto. A desvantagem é que o usuário perde a flexibilidade presente na versão em C++.

A versão em Python utiliza o modelo de orientação à objetos e as chamadas de funções têm características semelhantes à interface em C++ [11], isto é, as assinaturas das funções são semelhantes em ambas versões. A principal diferença do MEEP-Python é em relação à definição da geometria da simulação. Embora tenham disponibilizado mais de uma maneira de identificar o objeto gráfico e seus materiais no cenário de simulação, os autores indicam aquela que consiste na definição de coordenadas de polígonos em matrizes para representar a geometria, e na definição de materiais de acordo com a dimensão da

simulação, isto é, no caso bidimensional, os materiais são diretamente relacionados aos polígonos e, no caso tridimensional, formam camadas com determinadas espessuras para posteriormente serem relacionados aos polígonos criados. O intuito é utilizar o mecanismo de varredura do domínio de simulação efetuado pelo MEEP e validar qual material está presente numa determinada posição do cenário. Esta validação é realizada pelo algoritmo *winding number* [17], que verifica o material contido em um determinado polígono ou em uma determinada camada de acordo com a localização espacial estabelecida pelo mecanismo de varredura [18].

A principal vantagem dessa versão é que o MEEP foi estendido ao ecossistema Python, possibilitando a utilização das bibliotecas presentes nesta linguagem, principalmente aquelas que auxiliam o processo de simulação, como as bibliotecas que disponibilizam funcionalidades de visualização.

1.2 FORMULAÇÃO DO PROBLEMA

Pode-se dizer que o MEEP é um pacote popular, pois foi baixado mais de 10 mil vezes e referenciado em mais de 100 publicações em jornais desde que se tornou disponível [1]. Entretanto, notam-se alguns esforços para viabilização do programa.

A disponibilidade da versão em linguagem funcional *Scheme* busca facilitar a criação da simulação, tanto em relação à elaboração do código da simulação quanto na inclusão de geometrias no cenário. O usuário beneficia-se com a eliminação dos detalhes de programação presentes na linguagem C++ e com a disponibilidade de geometrias primitivas implementadas, o que facilita a configuração do Pré-processamento e possibilita a elaboração de geometrias com maior grau de complexidade. No entanto, o usuário precisará dominar a linguagem *Scheme* e não terá à disposição elementos geométricos normalmente encontrados em objetos gráficos, como triângulos e polígonos. Além disso, o usuário deverá exportar a geometria em sua forma discretizada – com os respectivos materiais – caso queira visualizá-la antes da etapa de Processamento e validar se configurou

corretamente o cenário da simulação.

Essa tentativa de viabilização torna-se mais notória na última versão MEEP-Python: a implementação do algoritmo *winding number* objetiva mapear analiticamente qualquer geometria para o MEEP. Ainda assim, a criação de geometrias complexas pode gerar uma quantidade enorme de código, pois os valores de coordenadas e de materiais presentes nas geometrias devem ser explicitados em matrizes. Isto significa que a configuração de qualquer geometria com muitos pontos exige muito esforço e atenção do usuário do sistema, além de aumentar a probabilidade de se cometer erros na inserção de todos os dados. Nesta versão, o usuário tem a opção de visualizar o cenário de simulação, mas deverá programar no código de simulação a chamada para uma biblioteca externa de visualização disponível em Python.

Em relação ao Pós-processamento, fica a critério do usuário definir a ferramenta para abrir o arquivo HDF5 [19] resultante da simulação. O grupo do MEEP disponibiliza o H5utils [20], uma ferramenta para conversão do formato HDF5 em outros tipos de arquivos, como PNG (arquivo de imagem), VTK (arquivo de versão mais antiga do *Visualization Toolkit*), TXT (arquivo de texto) e V5D (arquivo do programa livre Vis5D+ [21]). Outras ferramentas podem ser utilizadas para abrir o arquivo HDF5, como o MATLAB® [22] e o HDFView [23]. De qualquer forma, o usuário necessitará de uma ferramenta externa para abrir os arquivos correspondentes ao Pós-processamento. Além disso, essas ferramentas não disponibilizam funcionalidades para análises mais elaboradas e detalhadas em Pós-processamento, como a definição de planos arbitrários para cortes em volumes e a geração de animações automatizadas de simulações. As melhores opções podem ser o Matplotlib [24], para visualização 2D, e o Mayavi2 para 3D [11, 25], disponíveis no Python.

De uma maneira geral, compreende-se que o MEEP apresenta as seguintes limitações:

- Dificuldades na criação, inserção, manipulação, interação e visualização de geometrias complexas;
- Carência de funcionalidades de visualização para Pós-processamento;

- Ausência de uma interface gráfica que isente o usuário da necessidade de conhecer as linguagens pelas quais o MEEP é operado;
- Dependência de ferramentas e bibliotecas externas para realização de todas as etapas da simulação;
- Ausência de um sistema integrado que permita o gerenciamento de simulações;
- Ausência de um sistema de alto nível, modular, reusável e extensível, que permita expandir o MEEP para outros tipos de sistemas, como as aplicações *Web*.

1.3 OBJETIVOS DO TRABALHO

Este trabalho tem o propósito principal de contribuir com as etapas de Pré e Pós-processamento do simulador eletromagnético MEEP, buscando suprir os problemas levantados no tópico anterior. Objetiva-se, também, prover um ambiente computacional que forneça geometrias de dispositivos fotônicos de acordo com as demandas dos seguintes grupos: Departamento de Microondas e Óptica da FEEC-UNICAMP, FOTONICOM, CEPOF e CPqD.

Em termos de funcionalidades, têm-se os seguintes objetivos:

- Criação de importadores e leitores de dados gráficos para geração de geometrias no ambiente computacional;
- Implementação de geometrias baseadas em dispositivos fotônicos;
- Criação de uma interface de computação gráfica para visualização dos artefatos pertencentes à etapa de Pré-processamento, como geometrias, fontes eletromagnéticas e cenários completos de simulações;
- Criação de mecanismos de mapeamento de geometrias do ambiente computacional para processamento no MEEP;
- Desenvolvimento de mecanismos de interação com o programa H5utils e com o VTK para geração de volumes, cortes e filmes com perspectivas bidimensionais,

para análise de resultados em Pós-processamento;

- Desenvolvimento de um *software* livre, integrado, modular, reusável e extensível, com o propósito de flexibilizar o ambiente computacional para novas implementações e torná-lo portável para outros tipos de sistemas, como os *Web*.

1.4 ESTRUTURA DO TRABALHO

No Capítulo 2, são mostrados os mecanismos atuais utilizados pelo MEEP para realização do estágio de Pré-processamento e são introduzidos os conceitos básicos e os elementos de visualização e computação gráfica utilizados no desenvolvimento do Pré-processamento deste trabalho. Esse capítulo também descreve os formatos de arquivos gráficos DXF, STL e OBJ e apresenta os leitores de arquivos STL e OBJ disponíveis no *framework* do VTK. Além disso, são mostrados os recursos utilizados do VTK para o desenvolvimento dos dispositivos de cristais fotônicos disponíveis e para a definição da estratégia de discretização das geometrias e seus materiais para posterior processamento no MEEP.

No Capítulo 3, são apresentados os conceitos de visualização e os recursos utilizados do VTK e do H5utils para implementação de todas funcionalidades presentes no Pós-processamento deste trabalho.

No Capítulo 4, é mostrado o projeto do sistema. São descritos os componentes implementados e a relação entre eles. É apresentada também a integração do ambiente computacional com o MEEP na versão C++.

A validação das funcionalidades implementadas e os resultados obtidos são apresentados no Capítulo 5, através da realização de simulações no ambiente computacional implementado.

As conclusões sobre o desenvolvimento deste ambiente computacional, suas contribuições, os trabalhos em andamento e algumas sugestões de trabalhos que podem agregar contribuições futuras à esta pesquisa são apresentadas no Capítulo 6.

CAPÍTULO 2

PRÉ-PROCESSAMENTO

Este capítulo descreve os recursos atualmente disponíveis no MEEP para realização do estágio de Pré-processamento e introduz os mecanismos de visualização e computação gráfica utilizados neste ambiente computacional para a implementação de funcionalidades relacionadas à este estágio das simulações eletromagnéticas. São apresentadas a descrição do arquivo CAD-DXF e seu importador, a integração dos leitores de arquivos gráficos STL e OBJ, os mecanismos de implementação dos dispositivos de cristais fotônicos e a estratégia de mapeamento de geometrias e respectivos materiais para processamento no MEEP.

2.1 INTRODUÇÃO

Para realizar simulações eletromagnéticas no MEEP, o usuário deve definir, ao menos, quatro elementos principais: o domínio computacional, que corresponde ao espaço finito da simulação; a fonte eletromagnética, geradora das ondas eletromagnéticas; o campo eletromagnético a ser verificado (elétrico ou magnético); e os materiais contidos no domínio, que correspondem àqueles presentes nos objetos geométricos – que podem ser estruturas guiantes ou não guiantes de onda – e àqueles não pertencentes às geometrias, contidos no espaço e normalmente representados pelo ar. A Figura 2.1 ilustra um exemplo de um domínio de simulação configurado.

O domínio computacional, a fonte eletromagnética e o campo eletromagnético possuem parâmetros bem definidos e podem ser facilmente configurados. No entanto, a inserção de geometrias e materiais torna-se difícil na medida que os objetos tornam-se customizados. O usuário do sistema é responsável por definir o objeto gráfico da

simulação.

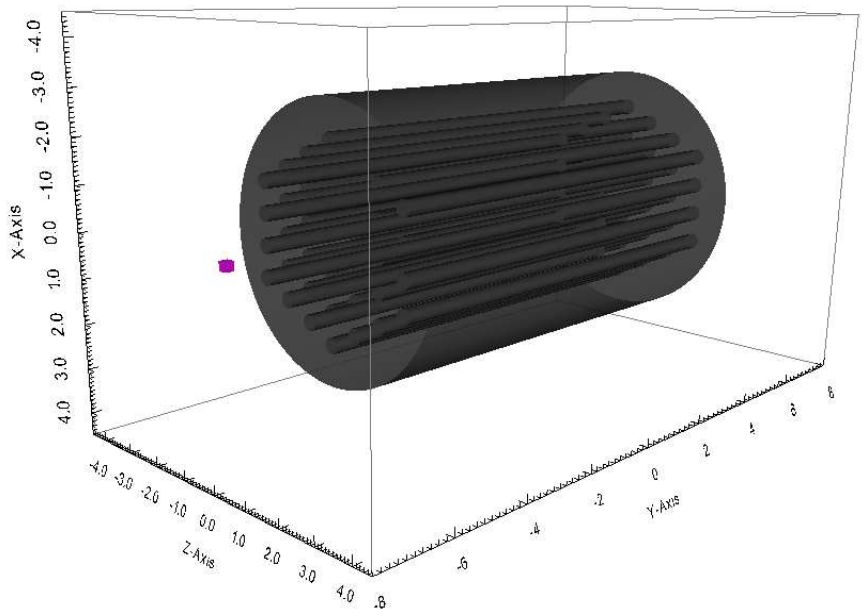


Figura 2.1 – Domínio de simulação configurado: objeto cilíndrico (guia de onda); caixa menor roxa (fonte); caixa transparente maior (domínio computacional).

2.2 INSERÇÃO DE GEOMETRIAS E MATERIAIS NO MEEP

Para cada versão atual do MEEP existe uma maneira de inserir geometrias e materiais.

No pacote de simulação na versão em C++, que é organizado em termos de classes de objetos (este conceito está presente no Paradigma de Programação Orientação à Objetos), a classe *structure* [26] é a responsável por receber o mecanismo de mapeamento de geometrias e materiais, que pode ser realizado de duas formas: através de um ponteiro de função ou da classe *material_function* [26]. Ambos mecanismos de mapeamento de geometrias e respectivas propriedades de materiais operam em função das posições determinadas pela discretização do subespaço de simulação, definida por parâmetros informados pelo usuário. O esquema de discretização do MEEP será melhor comentado no tópico **2.8**, mas pode ser visto como uma grade computacional que divide uniformemente o

domínio em linhas (caso unidimensional), quadrados (caso bidimensional) e cubos (caso tridimensional), conforme a resolução definida pelo usuário da simulação.

Embora forneça um conjunto de funções para realização de uma interface com o núcleo do MEEP, a classe *material_function* não oferece uma implementação para mapear geometrias na simulação. Como documentado no código-fonte do sistema (disponível no arquivo de cabeçalho *meep.hpp*), *material_function* é base para implementação de classes que serão definidas pelo usuário. Comporta-se, assim, como uma interface, pois explicita, através de seus métodos e dados, o padrão de funções (tipos de parâmetros de entrada e tipo de retorno) exigidos pelo MEEP. O método *eps* é o procedimento utilizado por *material_function* para amostrar as características de permissividade elétrica – ε – nos materiais das geometrias e do espaço livre contidos no domínio de simulação. Este projeto suporta atualmente apenas as características de permissividade elétrica, embora o MEEP seja capaz de incluir as características de permeabilidade magnética – μ – de materiais (a permeabilidade magnética estará presente nas versões subsequentes deste projeto).

Equivalente ao método *eps* de *material_function*, o ponteiro de função *eps*, referenciado pela classe *structure*, também não possui uma implementação de mapeamento definida.

O MEEP opera ambos mecanismos através de chamadas consecutivas da função *eps*, atribuindo novos valores ao parâmetro *vec* [26] – a classe responsável por carregar dados de coordenadas de um ponto – para retornar um determinado material de acordo com o posicionamento indicado em *vec*. As chamadas à *eps* são realizadas até que todos os pontos da grade computacional tenham sido percorridos. Fica, portanto, a cargo do usuário desenvolver o mecanismo que mapeia, corretamente, as geometrias e respectivos materiais baseado nas informações de localização fornecidas por *vec*.

Pode-se dizer que as geometrias que são definidas em função de coordenadas são mais fáceis de mapear, pois é possível delimitá-las através de suas funções, isto é, existem meios de determinar quais valores de *vec* pertencem ou não à região do objeto em simulação. Se, por exemplo, o usuário quiser definir um objeto circular, basta representá-lo em sua forma analítica, ou seja, através da função $x^2 + y^2 = r^2$ (a soma dos quadrados das

coordenadas é igual ao quadrado do raio do círculo). Assim, é possível determinar quais pontos de coordenadas *x* e *y* fazem parte do círculo ou não. Os pontos que resultam em valores menores ou iguais ao quadrado do raio do círculo são considerados pertencentes à ele e os restantes ao espaço livre do domínio computacional. Nos casos de geometrias um pouco mais elaboradas, o usuário pode definir mais de uma classe *structure* na simulação e, portanto, mais de uma função de mapeamento, possibilitando a diminuição da complexidade de definição de geometrias em apenas uma função. A Figura 2.2 apresenta a forma da função do círculo em C++ que pode ser passada à classe *structure* para ser processada pelo MEEP.

```
// teste 2: dielétrico do sílicio em um círculo sólido
double epsCircle(const vec &p)
{
    double currentVec[3] = {p.x(), p.y(), 0.};

    double value = ((currentVec[0] - circleCenter[0]) * (currentVec[0] - circleCenter[0])) +
                   ((currentVec[1] - circleCenter[1]) * (currentVec[1] - circleCenter[1]));

    if (value <= (radius * radius)) // está dentro ou na fontreira do círculo
        return 12.0; // valor dielétrico do silício

    return 1.0; // valor dielétrico do ar
}
```

Figura 2.2 – Função em C++ que define uma região circular: o objeto *vec* possui a posição corrente; a variável *value* é usada para validar se o ponto está dentro do círculo com raio representado pela variável *radius.*

A versão do pacote de simulação em *Scheme* oferece mais mecanismos de inserção de geometrias. São definidos, por padrão, os seguintes objetos gráficos para esta versão: círculo, cone, cilindro, esfera e bloco, que são configurados através de seus parâmetros. A Figura 2.3 ilustra, como exemplo, o *script* para a construção de um bloco.

```
(set! geometry (list
                (make block (center 0 0) (size infinity 1 infinity)
                      (material (make dielectric (epsilon 12))))))
```

Figura 2.3 – *Script* em *Scheme* para a construção de um bloco no MEEP [27].

O usuário que opta por utilizar o MEEP via *Scheme* é favorecido pela facilidade de

inserção das geometrias padrões disponíveis, podendo combiná-las para criação de objetos mais elaborados. Ainda assim, existem objetos que não podem ser formados pelas geometrias padrões disponíveis ou pela combinação entre elas, limitando, assim, a elaboração de simulações com determinados objetos gráficos customizados. O usuário pode, por exemplo, ter dificuldade de gerar objetos triangulares ou poligonais.

A versão MEEP-Python oferece outros mecanismos de inserção de geometrias. Dentre os disponíveis, os autores recomendam utilizar aquele que é baseado na criação de polígonos. Este método opera da seguinte maneira: define-se uma matriz com os valores de coordenadas para cada polígono que forma a geometria a ser simulada, e relaciona-se os materiais à geometria de acordo com a dimensão do domínio de simulação, isto é, no caso bidimensional, os materiais são diretamente associados à cada polígono e, no caso tridimensional, os materiais são definidos com certas espessuras para formarem camadas e serem posteriormente relacionados aos polígonos [18].

Nesta versão, é utilizado o mecanismo de varredura da grade computacional, efetuado pelo MEEP, para validar os materiais contidos na geometria. Baseado nas posições determinadas pela varredura, o algoritmo *winding number* verifica, no caso bidimensional, o polígono na qual a posição corrente repousa ou, no caso tridimensional, a camada de material na qual se encontra a posição atual. A Figura 2.4 ilustra um exemplo de um Pré-processamento configurado na versão MEEP-Python.

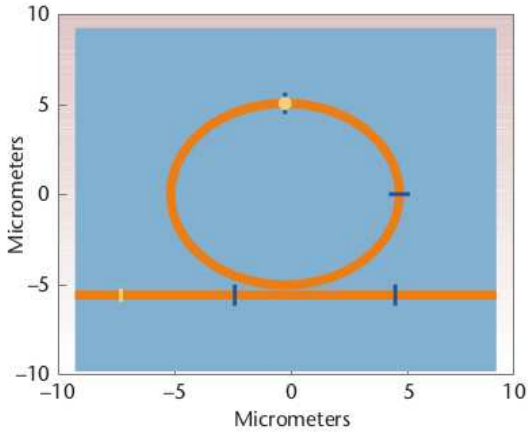


Figura 2.4 – Exemplo de um Pré-processamento configurado no MEEP-Python para simulação 2D [11].

2.3 *VISUALIZATION TOOLKIT* - VTK

O VTK é um sistema de *software* orientado à objetos e de código aberto para computação gráfica, visualização e processamento de imagens [28]. Seu *framework* (conjunto de objetos e funcionalidades) é amplo e contem representações de dados estruturados e não-estruturados como pontos estruturados e não-estruturados, polígonos, imagens, volumes e *grids* estruturadas, retilineares e não estruturadas; algoritmos de visualização, que incluem métodos para dados escalares, dados de vetores, tensores, texturas e dados volumétricos; e técnicas avançadas de modelagem como redução de polígonos, modelagem implícita, suavização de malhas, cortes, contornos e triangularização [29].

O VTK possibilita a integração de dados com outras aplicações através da implementação de exportadores e importadores de dados e oferece mecanismos de leitura e escrita de dados em arquivos e banco de dados. Seu sistema é multiplataforma e pode ser executado nos sistemas operacionais Linux, Unix, Mac e Windows.

As descrições anteriormente apresentadas compõem as justificativas de escolha e uso do sistema VTK no ambiente computacional desenvolvido neste projeto de pesquisa.

O VTK consiste de duas partes: uma biblioteca de classes compiladas em C++ e as interfaces ou camadas interpretadoras, conhecidas como *wrappers*, que permitem a manipulação das classes compiladas através das linguagens Java, TCL e Python, conforme mostra a Figura 2.5.

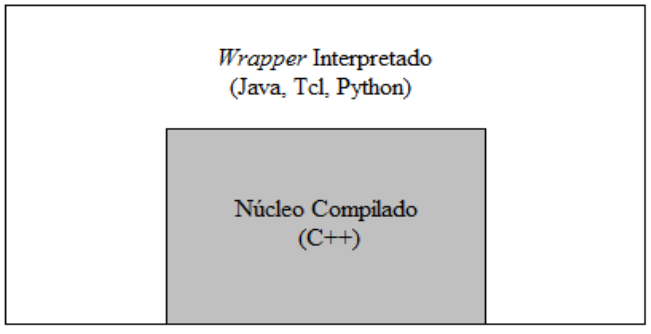


Figura 2.5 – Núcleo compilado em C++ e as linguagens interpretadas no VTK [30].

Neste projeto, utilizou-se principalmente o VTK em Java, a linguagem base deste ambiente computacional. A versão C++ do VTK também foi utilizada, quando desenvolveu-se o suporte para mapeamento de geometrias na versão C++ do MEEP.

O VTK é constituído de dois subsistemas principais: o *pipeline* de visualização e o modelo gráfico, que serão detalhados nos tópicos seguintes.

2.3.1 *Pipeline* de Visualização

A visualização é o processo de transformação de dados em imagens que eficientemente e precisamente transmitem informações sobre estes dados [30]. No VTK, o processo pela qual os dados fluem para serem transformados é conhecido como *Pipeline* de Visualização. O *pipeline* consiste de objetos de dados (representação), objetos que processam os dados e de conexões entre os objetos de maneira que exista um fluxo direcionado da informação.

2.3.1.1 Objetos de Dados

Os objetos de dados representam a informação e são referidos como *Dataset*. O *Dataset* consiste de atributos e de uma estrutura, dividida em duas partes: geometria e topologia. A geometria corresponde à informações de posição no espaço tridimensional e é representada por pontos, enquanto a topologia, representada por células, corresponde à propriedades que não variam em transformações geométricas, como a rotação e a translação [30]. Os atributos de dados são informações que caracterizam o objeto *Dataset*, incluindo dados escalares, vetores, tensores e textura. A Figura 2.6 ilustra a arquitetura do *Dataset*. Os diferentes tipos de *Dataset* disponíveis no VTK são mostrados na Figura 2.7.

As células que formam a estrutura topológica do *Dataset* são os blocos fundamentais do objeto gráfico e são classificadas em lineares e não-lineares, respectivamente ilustradas na Figura 2.8.

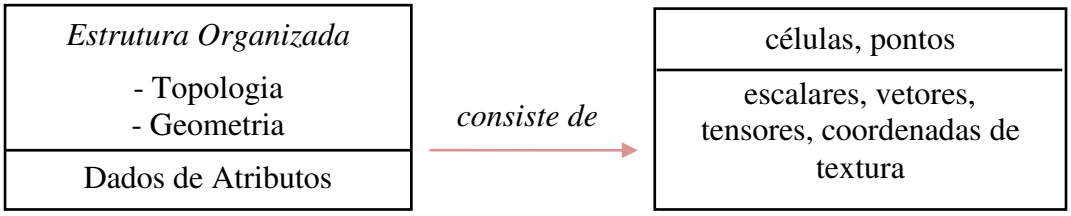


Figura 2.6 – Arquitetura do *Dataset* [30].

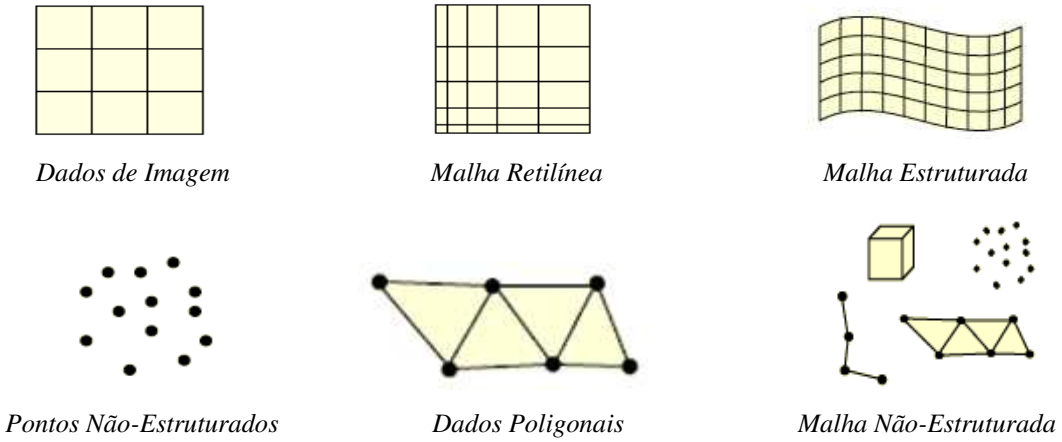


Figura 2.7 – Tipos de *Dataset*. A malha não-estruturada abrange todos tipos de células [30].

Um ponto importante relativo à definição de uma célula é a necessidade de associação correta entre os dados geométricos, isto é, uma lista de pontos deve relacionar-se com uma lista de índices para formar a topologia da célula, conforme mostra a Figura 2.9. A lista deve possuir uma ordenação específica dos índices de forma que sua associação à lista de pontos defina corretamente a topologia da célula.

Os dados de atributos, que correspondem às características presentes na geometria, podem ser associados à pontos, células, arestas e faces, entre outros, e são representados por tipos específicos baseados nas formas de dados mais comuns encontradas nos algoritmos de visualização [30], conforme ilustra a Figura 2.10.

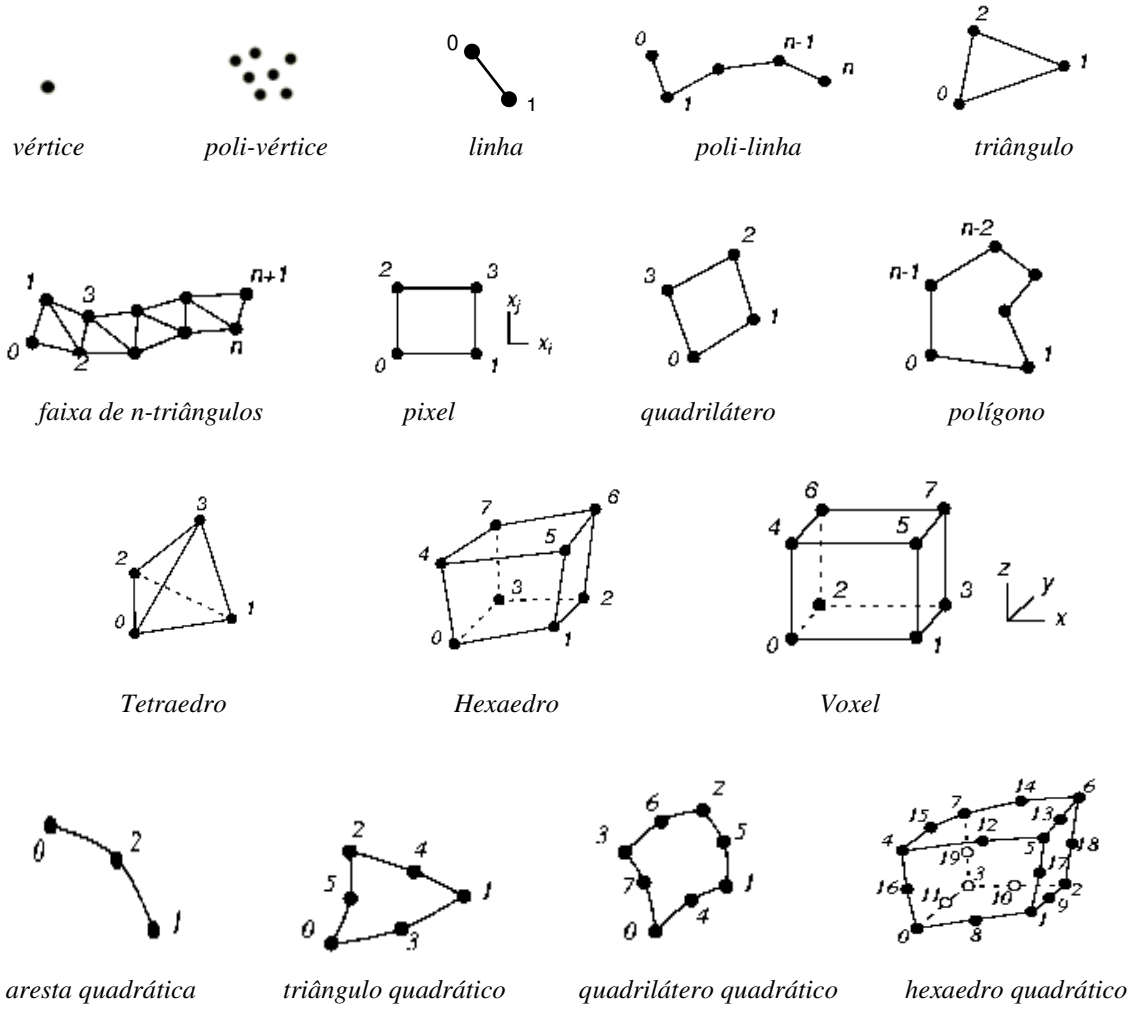


Figura 2.8 – Alguns tipos de células lineares e não-lineares no VTK [30].

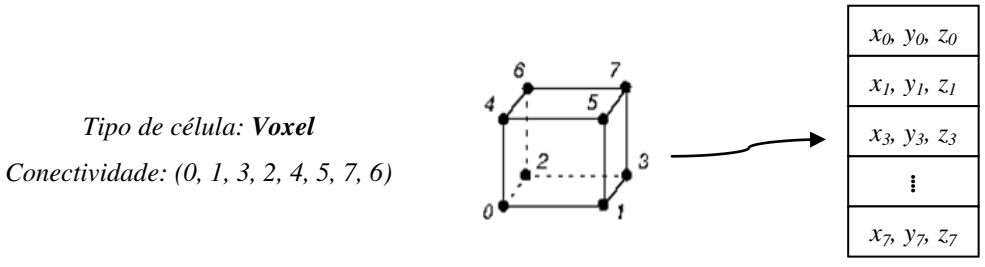


Figura 2.9 – Topologia da célula definida pela lista de índices associados à lista de coordenadas de pontos [30].

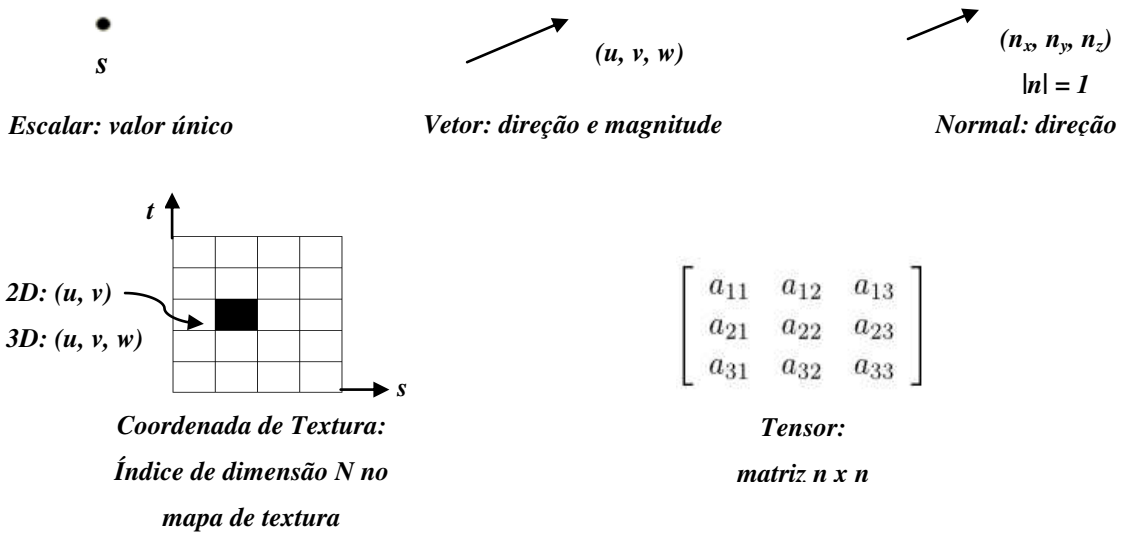


Figura 2.10 – Dados de atributos no VTK [30].

2.3.1.2 Objetos de Processos

Objetos de processos operam em dados de entrada para gerar dados de saída [28]. Estes, por sua vez, podem gerar novos dados de entrada ou serem transformados, ganhando um novo aspecto.

No VTK, os objetos de processo são classificados em objetos fontes, filtros e mapeadores. Os fontes geram dados sem nenhuma entrada ou são capazes de ler dados externos utilizando, para isso, objetos de leitura. Os filtros exigem pelo menos um objeto de dados como entrada e geram um ou mais dados de saída. O processo de triangularização é um exemplo de filtro no VTK. Os mapeadores consomem dados de entrada e não geram outros dados de saída, terminando, assim, o fluxo de dados do *pipeline* de visualização. Estes mapeadores convertem os dados para as primitivas gráficas, isto é, comunica seus dados com o subsistema gráfico. Os mapeadores podem escrever seus dados em arquivos, através de objetos de escrita.

A Figura 2.11 ilustra uma visão geral da execução do *pipeline* de visualização através da conexão entre os objetos de processos.

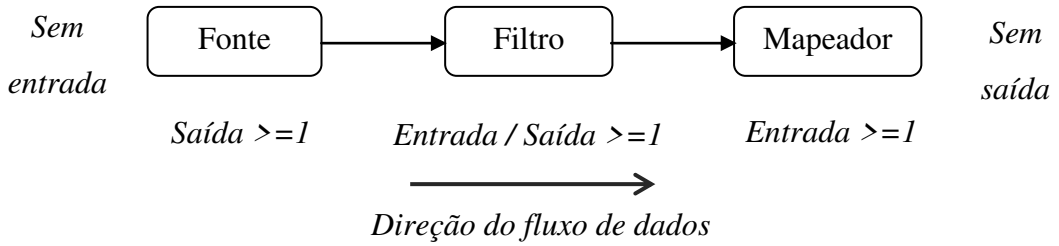


Figura 2.11 – Visão geral do *pipeline* de visualização [28].

2.3.2 Modelo Gráfico

O modelo gráfico envolve os objetos responsáveis pela renderização de uma cena.

Para visualizar os dados, é necessária a abertura de uma janela na tela do computador. A classe *vtkRenderWindow*, disponível no VTK, é a responsável por representar a janela que aparecerá na interface gráfica do usuário. Ela carrega os renderizadores, responsáveis pela produção das imagens, e gerencia a janela no dispositivo de saída, além de ser responsável por manter informações referentes às características da janela de saída, como a posição, o tamanho e o nome da janela [30].

A classe *vtkRender* tem a responsabilidade de coordenar luzes, câmeras e atores presentes nas cenas em execução. Quando instanciada, o objeto em execução cria uma câmera e as luzes por padrão, caso estes não sejam especificados. No entanto, ao menos um ator deve ser definido. A classe *vtkRender* define também a cor da luz ambiente e do fundo da imagem.

Para que as instâncias da classe *vtkRender* apareçam na interface, elas precisam ser relacionadas com o objeto *vtkRenderWindow*. Quando uma instância do renderizador (*vtkRender*) é definido no *vtkRenderWindow*, sua área de ocupação na janela de saída deve ser indicada em um *viewport*, uma região retangular onde ficam as instâncias do *vtkRender*. O *viewport* recebe os valores da coordenada onde deseja-se colocar o *vtkRender*. Estes valores de coordenadas são normalizados, isto é, possuem intervalo de 0 a 1, em ambos eixos *x* e *y*. Se houver apenas uma instância do *vtkRender*, ele se estenderá por toda a região

do *viewport*, ocupando toda a janela de saída. Neste caso, não é necessário indicar o valor do *viewport*, pois a definição é feita por padrão.

Para controlar a câmera da cena, o VTK possui a classe *vtkCamera*. Os atributos referentes a posição, ponto focal e localização são definidos nesta classe, assim como as propriedades de elevação, azimute e *zoom* dos objetos gráficos. O renderizador encarrega-se de criar uma instância do *vtkCamera* caso esta não for definida.

Os objetos gráficos da cena são representados pela classe *vtkActor*. Esta classe combina as propriedades do objeto como tipo, cor, sombreamento, definição geométrica e orientação no sistema de coordenadas global [30]. As transformações de rotação, translação e mudança de escala são realizadas no objeto *vktActor*. As propriedades do ator são definidas na classe *vtkProperty*, responsável pela aparência da cena. Cada ator cria, por padrão, uma instância de *vtkProperty*, mas é possível associar outras instâncias do *vtkProperty* com o ator ou atores da cena, assim diferentes atores podem compartilhar as mesmas propriedades de aparência. As instâncias da classe *vtkActor* mantêm, além dos objetos de transformação e de propriedades, instâncias da classe *vtkMapper*. Como mencionado anteriormente, a classe *vtkMapper* termina o *pipeline* de visualização e é a ponte para o subsistema gráfico. Todo objeto *vtkActor* definido deve ter associado um *vtkMapper*. A Figura 2.12 apresenta um exemplo da janela renderizada na interface gráfica do usuário.

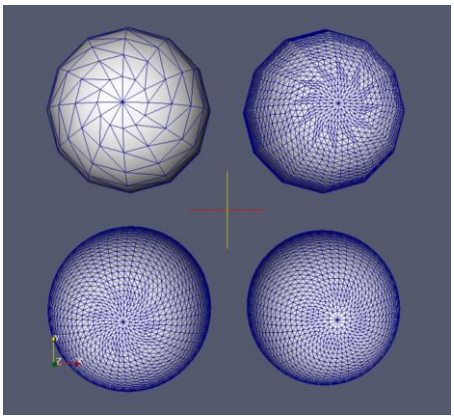


Figura 2.12 – Janela renderizada no VTK: cada objeto gráfico esférico está contido em um renderizador (*vtkRender*) e é representado por um *vtkActor*. As posições são definidas pelo *viewport* [31].

2.4 ARQUIVO *DRAWING EXCHANGE FORMAT* - DXF

O arquivo DXF é um tipo original do AUTOCAD® [32], um *software* proprietário desenvolvido pela companhia AutoDesk® [33] para criação de desenhos CAD (*Computer-Aided Design*) – desenhos auxiliados por computador. Apesar de pertencer à um sistema proprietário, o arquivo DXF pode ser usado livremente.

O formato DXF organiza as informações contidas no arquivo do desenho de modo a representar os dados através de marcas, isto é, cada elemento de dados no arquivo é indexado por um número inteiro chamado de grupo de código. O valor do grupo de código indica o tipo e o significado do respectivo elemento de dados.

O arquivo DXF possui dois formatos: o ASCII (*American Standard Code for Information Interchange*) e o binário.

2.4.1 Estrutura do Arquivo DXF

O arquivo DXF é organizado em seções, da seguinte maneira [34]:

- Seção ***HEADER***: contem informações gerais do desenho, onde cada parâmetro desta seção tem um nome e um valor associado;
- Seção ***TABLES***: contem definições de tipo de linha (*LTYPE*), camada (*LAYER*), estilo de texto da tabela (*STYLE*), tabela de visão (*VIEW*), tabela do sistema de coordenadas do usuário (*UCS*), tabela de configuração da visão (*VPORT*), tabela de estilo de dimensão (*DIMSTYLE*) e tabela de identificação da aplicação (*APPID*);
- Seção ***BLOCKS***: contem entidades de definição de blocos descrevendo cada entidade que compõe cada bloco no desenho;
- Seção ***ENTITIES***: contem as entidades desenhadas, incluindo quaisquer referências a blocos;
- ***END OF FILE*** (fim de arquivo).

Os vários grupos que compõem o formato DXF ocupam duas linhas cada. A primeira linha é o código de grupo (representado pelo inteiro sem sinal) e a segunda é um valor associado ao código de grupo. A função específica do grupo depende da variável corrente, do item de tabela ou da entidade a ser lida [35]. A seguir, são listados alguns dos principais códigos de grupo.

- Código **0**: identifica o começo de uma entidade, entrada de tabela ou separador de arquivo;
- Código **2**: usado para identificar uma seção do arquivo DXF ou um nome de uma tabela;
- Código **6**: nome do tipo de linha;
- Código **7**: nome do estilo de texto;
- Código **8**: nome da camada;
- Código **10**: coordenada *x* primária (ponto inicial de uma linha, centro de um círculo, etc.);
- Código **20**: coordenada *y* primária;
- Código **30**: coordenada *z* primária;
- Código **50 - 58**: ângulos;
- Código **62**: número de cor;
- Código **210**, **220**, **230**: coordenadas *x*, *y*, *z* da direção de deslocamento;
- Código **1000**: *string* em ASCII;
- Código **1011**, **1021**, **1031**: coordenadas *x*, *y*, *z* da posição do cenário 3D, onde se encontram as entidades;
- Código **1012**, **1022**, **1032**: coordenadas *x*, *y*, *z* do deslocamento do cenário;
- Código **1013**, **1023**, **1033**: coordenadas *x*, *y*, *z* da direção do cenário.

O código de grupo **999** indica que a linha seguinte é um comentário na forma de *string*.

A seguir, é apresentada a seção *ENTITIES*, responsável por fornecer as informações de geometria e materiais.

2.4.1.1 Seção *ENTITIES*

As entidades aparecem tanto na seção *BLOCKS* quanto na seção *ENTITIES* do arquivo DXF. As definições de cada entidade são equivalentes nestas duas seções.

Cada entidade começa com um código de grupo 0 (zero), identificando o tipo da entidade, e contem um código de grupo 8 que fornece o nome da camada na qual está. Além disso, cada entidade tem associada a ela os parâmetros de elevação, espessura, tipo de linha e cor. A seguinte estrutura de código é utilizada para a leitura desses parâmetros das entidades:

- Código **6**: seguida da indicação do nome do tipo de linha;
- Código **38**: seguida da indicação da elevação da entidade (se for um valor diferente de zero). Contudo, esta parte pode ser suprimida por coordenadas *z* aplicadas aos pontos da entidade;
- Código **39**: precede a indicação da espessura da entidade;
- Código **62**: precede a cor da entidade;
- Códigos **210**, **220**, **230**: esses parâmetros indicam, respectivamente, as coordenadas *x*, *y*, *z* da direção de extrusão.

A seguir, é mostrado o formato de algumas das principais entidades que aparecerão no arquivo, ou seja, dos códigos de grupo que aparecerão em sua leitura:

- ***LINE***: 10, 20, 30 (ponto inicial), 11, 21, 31 (ponto final);
- ***POINT***: 10, 20, 30 (ponto);
- ***CIRCLE***: 10, 20, 30 (centro), 40 (raio);
- ***SOLID*** e ***3DFACE***: quatro pontos definem os quatro cantos do sólido: (10, 20, 30), (11, 21, 31), (12, 22, 32), (13, 23, 33). Se apenas três pontos forem encontrados durante a leitura (sólido triangular), o terceiro e o quarto pontos serão o mesmo [35].

A Figura 2.13 ilustra um exemplo básico da estrutura de grupos de códigos para a entidade *3DFACE*.

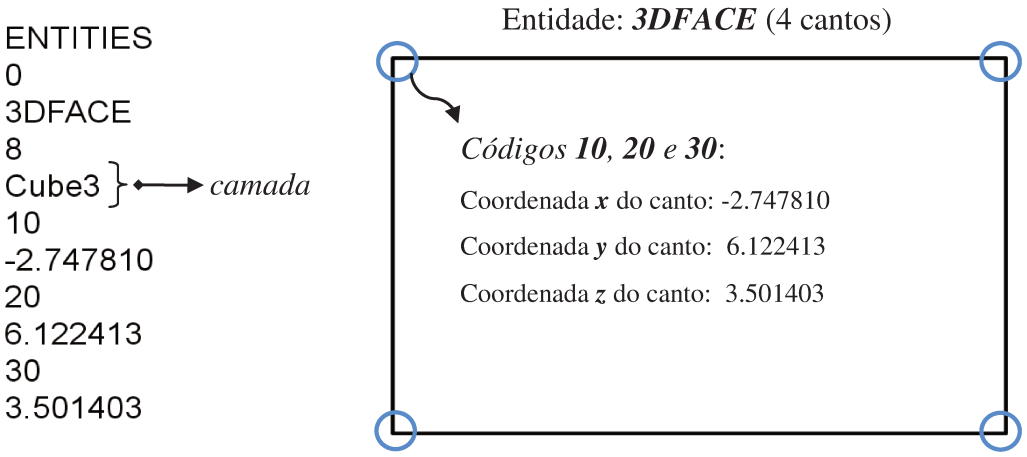


Figura 2.13 – Estrutura básica de código de grupo em uma entidade (*3DFACE*) do arquivo DXF.

A Figura 2.14 apresenta um exemplo de um objeto gráfico disponível em DXF.

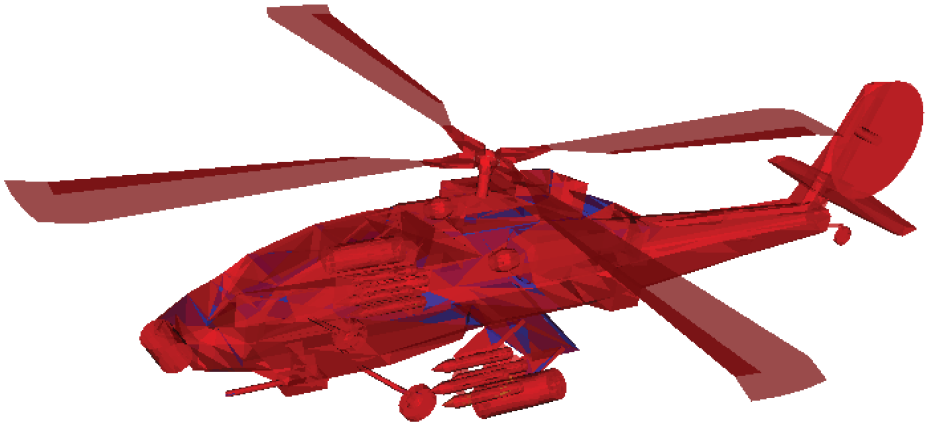


Figura 2.14 – Visualização de um objeto gráfico (helicóptero) armazenado em arquivo DXF.

2.5 ARQUIVO *STEREO-LITHOGRAPHY* - STL

O arquivo de extensão *.stl* é composto de dados relativos à superfícies de geometrias e sua organização é em termos de faces (*facets*), que são estruturas poligonais representadas na forma de triângulos. Uma superfície geométrica é inteiramente formada, portanto, por

um conjunto de triângulos.

O conteúdo do arquivo STL descreve as faces através de coordenadas relacionadas ao vetor normal e aos vértices do triângulo, ordenados no sentido anti-horário. Cada triângulo da geometria pode dividir dois de seus vértices com os triângulos adjacentes. O vetor normal tem direção voltada para o exterior do objeto [36].

Existem dois formatos para arquivos STL: ASCII (textual) e binário. Por ocupar menor volume de armazenamento em arquivo, a versão binária é a mais utilizada.

O VTK implementa a classe *vtkSTLReader*, um dos objetos fontes de leitura de arquivos presentes em sua biblioteca. A classe *vtkSTLReader* lê os dados de um arquivo STL e, assim como outros objetos de leitura, gera um objeto de dados no *pipeline* de visualização do VTK (tópico **2.3.1**) [28]. Os dois formatos do STL, ASCII e binário, são suportados no VTK.

Para efetuar leituras de arquivos com extensão *.stl* utilizando a classe *vtkSTLReader*, deve-se chamar o método *SetFileName*, que recebe o nome e o caminho de diretório do arquivo.

A Figura 2.15 apresenta um exemplo de um objeto gráfico disponível em STL.

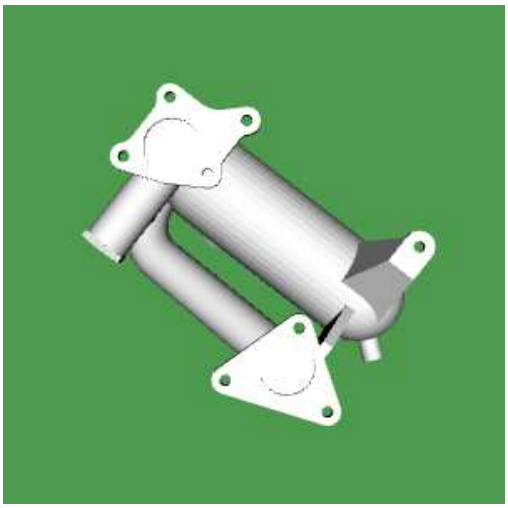


Figura 2.15 – Visualização de objeto gráfico armazenado em formato STL [37].

2.6 ARQUIVOS *WAVEFRONT* - OBJ

O tipo de arquivo *.obj* define geometrias e outras propriedades de objetos para o visualizador avançado Wavefront® [38]. Assim como os arquivos STL, os arquivos OBJ podem ser formatados em código ASCII ou binário.

De uma maneira geral, o arquivo *.obj* define objetos poligonais, como linhas, pontos e faces, e objetos de forma livre, como curvas e superfícies. As faces são construídas de modo análogo às faces presentes em STL. No entanto, os vértices das faces do arquivo OBJ podem ser de tipos diferentes, distinguindo-se entre vértices de geometria (coordenadas de uma face geométrica), vértices de textura (coordenadas de faces texturais), vértices de vetores normais e vértices de parâmetros de espaço (objetos de forma livre) [38].

O VTK implementa a classe *vtkOBJReader* e fornece, assim como a classe *vtkSTLReader*, o método *SetFileName*. O parâmetro deste método é do tipo ponteiro para um vetor de *char* (caracteres), que carregará o caminho e o nome do arquivo de extensão *.obj* (ou *.stl*). O objeto *vtkOBJReader* gera, como saída, dados poligonais.

A Figura 2.16 apresenta um exemplo de um objeto gráfico disponível em OBJ.

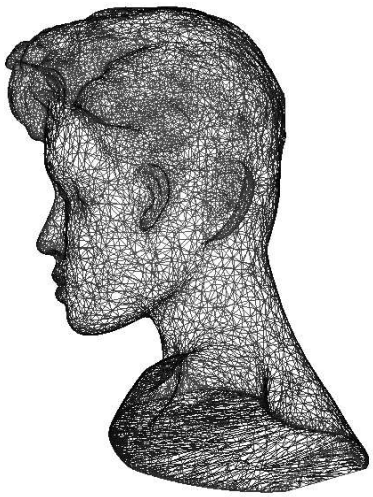


Figura 2.16 – Visualização de um objeto gráfico armazenado em formato OBJ.

2.7 CONSTRUÇÃO DE DISPOSITIVOS DE CRISTAIS FOTÔNICOS COM VTK

Uma das propostas de Pré-processamento desta pesquisa é a realização de simulações que envolvam propagação de ondas em dispositivos de cristais fotônicos. Pesquisadores do DECOM e grupos relacionados, como FOTONICOM, CePOF e CPqD, têm desenvolvido estudos e pesquisas relacionados à este tipo de dispositivo. A construção de dispositivos de cristais fotônicos otimizados para acoplamento de luz [39, 40] é um exemplo deste tipo de trabalho. O grupo do MEEP também tem interesse em pesquisas relativas à fotônica e faz uso de seu sistema para realização de simulações, em conjunto com outros pacotes de *softwares* também desenvolvidos pelo grupo, como o MPB (MIT *Photonic Bands*) [41].

Os dispositivos de cristais fotônicos possuem importantes aplicações nos sistemas de tecnologia de ondas de luz [42], como a óptica. De um modo bem geral, estes dispositivos são aqueles cuja operação é baseada na interação de fótons em um sólido. Os fótons, responsáveis pela "condução de informação", são as partículas elementares ou fundamentais da luz.

O *Visualization Toolkit* provê um conjunto de funcionalidades que possibilitam a criação de objetos com características sólidas, baseadas em operações booleanas que são aplicadas à objetos construídos por funções implícitas ou que são implementadas em filtros de objetos poligonais, representados pelo *Dataset vtkPolyData* (classe de objetos). Neste projeto, ambas abordagens foram testadas, e suas descrições são apresentadas nos tópicos subsequentes.

Embora o VTK possibilite a caracterização de objetos em formas sólidas através da união, diferença e intersecção de objetos gráficos primitivos, as geometrias resultantes não representam sólidos reais, mas superfícies de polígonos. Para adequar essas geometrias à sólidos, utilizou-se a técnica de traçamento de raios, descrita no tópico **2.8**.

2.7.1 Funções Implícitas, Operações Booleanas e Junção de Geometrias no VTK

Neste trabalho, utilizou-se funções implícitas para a modelagem de geometrias de dispositivos de cristais fotônicos. Estas funções possuem a seguinte forma: $F(x, y, z) = c$. Os parâmetros x, y e z representam as coordenadas de um determinado ponto e o valor c uma constante arbitrária [30].

As funções implícitas são ferramentas convenientes para descrever formas geométricas comuns, como planos, esferas, cilindros, cones e outros. Além disso, estas funções são capazes de separar o espaço euclidiano em três regiões distintas: dentro, fora ou sobre a função implícita. Um exemplo é a função $F(x, y, z) = x^2 + y^2 + z^2 - R^2$, uma equação que representa uma esfera de raio R. Quando $F(x, y, z) = 0$, a região do espaço é a superfície da esfera; quando $F(x, y, z) > 0$, a região corresponde à parte exterior da esfera; por fim, quando $F(x, y, z) < 0$, a região definida é o interior do objeto. Qualquer ponto pode ser classificado de acordo com a região utilizando a função implícita da esfera [30].

A modelagem de objetos pode ser realizada por uma única função implícita ou pela combinação entre elas. A aplicação de uma técnica de contorno sobre o *Dataset* amostrado pela função gera, como saída, uma iso-superfície em representação poligonal [30].

Para criar objetos complexos, as funções implícitas podem ser combinadas pelos operadores booleanos correspondentes à união, intersecção e diferença, disponíveis no VTK [30]. A Figura 2.17 ilustra um exemplo de combinação booleana.

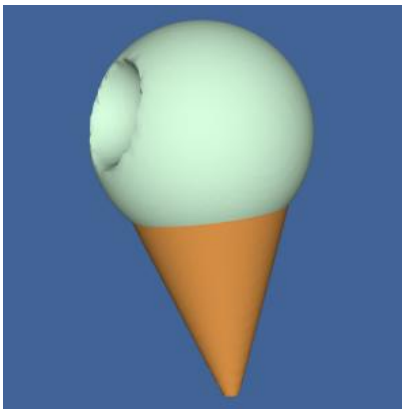


Figura 2.17 – Geometria gerada pela diferença de duas esferas e união com um cone [28].

O VTK provê outro mecanismo de combinação booleana, através do filtro *vtkPolyDataBooleanFilter*. Neste caso, não é necessário definir geometrias através de funções implícitas. As operações booleanas serão diretamente aplicadas ao *Dataset* que, neste caso, deve ser do tipo *vtkPolyData*. Ou seja, a aplicação deste filtro é efetivada apenas em geometrias poligonais.

2.8 MAPEAMENTO DE GEOMETRIAS E MATERIAIS NO MEEP

Para realizar a computação de propagação do campo eletromagnético, o FDTD necessita de uma malha de células que represente o cenário de simulação discretizado. No MEEP, as células da malha discretizada são uniformes, com extensões de mesmo tamanho nas dimensões *x*, *y* e *z*, formando quadrados / *pixels* (espaço bidimensional) ou cubos / *voxels* (espaço tridimensional).

Para determinar o material contido nas células, é necessária a execução de um algoritmo de traçamento de raios que efetue a varredura dos objetos gráficos presentes no domínio computacional. Esses objetos devem possuir volumes definidos [43], de forma que se estabeleça duas regiões: a interior e a exterior ao objeto. O domínio computacional representa a área total onde os objetos gráficos estão inseridos.

O procedimento utilizado pelo algoritmo consiste em mapear toda a região do domínio computacional com células uniformes. Como exemplo, foi utilizado um domínio bidimensional, ilustrado na Figura 2.18 (a). Nesta técnica, os raios são lançados de forma a encontrar os pontos de intersecção com o(s) objeto(s) gráfico(s) contido(s) no domínio, como ilustrado na Figura 2.18 (b).

Os raios originam-se de um ponto inicial em *x* (podendo se iniciar em *y* ou *z*) e viajam paralelos à este eixo. Os pontos finais dos raios lançados são representados pelas posições atribuídas ao objeto *vec* do MEEP. A quantidade de raios lançados é determinada pela quantidade de posições (em *vec*) que estão dentro dos limites da geometria, ou seja, os pontos em *vec* que caem fora dos limites inferiores e superiores do objeto gráfico (*x*, *y* e *z*

iniciais e finais – disponibilizados pelo VTK) são considerados pontos externos à geometria. Esse critério evita cálculos desnecessários de intersecções, computacionalmente mais caras que a validação que determina se a posição corrente está dentro ou fora dos limites de extensão da geometria.

Para cada raio criado, adota-se o seguinte critério: o raio deve parar no ponto denotado por *vec*; se o número de intersecções entre o raio e o objeto gráfico for ímpar, considera-se que a célula correspondente está dentro do objeto, caso contrário considera-se que a célula está na região externa ao objeto [6]. Na Figura 2.18 (b), o raio $\mathbf{R_2}$ para em um ponto (meramente ilustrativo) determinado por *vec*. Mesmo que na célula correspondente exista uma sub-região do objeto gráfico, o número de intersecções calculadas é par. Portanto, esta célula fará parte da região externa ao objeto (algumas técnicas, como o *subpixel smothing*, empregado pelo MEEP em [4], podem melhorar a precisão do mapeamento nestas células que contem uma sub-região da geometria a ser mapeada).

Para a célula considerada interna ao volume, deve-se atribuir o material que compõe aquela sub-região interseccionada. Os valores de materiais das geometrias são baseados nos dados de atributos das células interseccionadas do *Dataset* (descrito com mais detalhes no tópico **4.10**).

A Figura 2.19 ilustra a malha gerada após a execução do algoritmo de cruzamento de raios. É possível notar que uma resolução maior do domínio computacional pode refinar a malha, obtendo-se, assim, uma representação mais aproximada do objeto gráfico original.

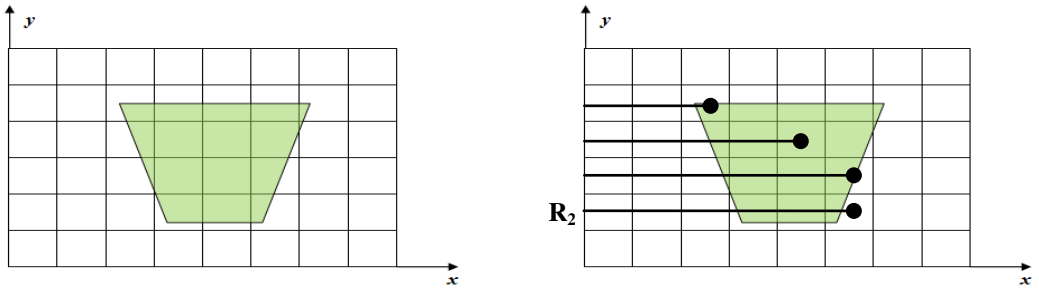


Figura 2.18 – Mapeamento do domínio computacional na *grid* cartesiana (a) e o cruzamento de raios (b).

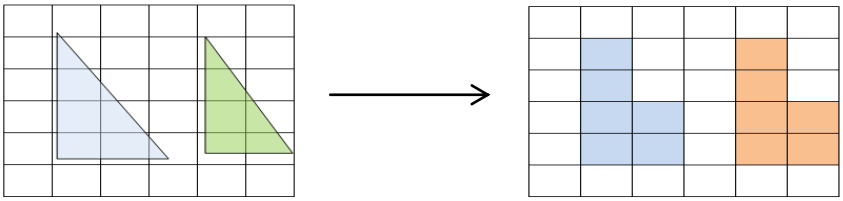


Figura 2.19 – Discretização da geometria por cruzamento de raios e definição de seus materiais.

CAPÍTULO 3

PÓS-PROCESSAMENTO

Este capítulo apresenta os conceitos de visualização e os recursos utilizados do VTK e do H5utils para implementação das funcionalidades gráficas presentes no Pós-processamento deste trabalho. São mostrados os mecanismos do utilitário H5topng, disponível no pacote H5utils, usados para obtenção de imagens no formato PNG a partir dos arquivos de tipo HDF5, criados pelo MEEP para o armazenamento dos resultados dos cálculos do FDTD na etapa de Processamento. São apresentadas, também, a geração de filmes bidimensionais pelo *software* Mencoder [44] e as funcionalidades do VTK para obtenção de planos de cortes em volumes, a partir das imagens obtidas pelo H5topng.

3.1 INTRODUÇÃO

Em geral, a execução das simulações nos *softwares* de simulação é dividida nos estágios de Pré-processamento, Processamento e Pós-processamento. Após a configuração do Pré-processamento, é realizada a etapa de Processamento, caracterizada pela execução dos cálculos da simulação. Esta computação – realizada no Processamento – normalmente tem custo computacional alto, pois se realiza por processos iterativos sobre equações computacionalmente caras, que devem gerar aproximações de soluções que atinjam um erro mínimo ou tolerável, o que dificulta ou inviabiliza a criação de sistemas capazes de processar simulações científicas em tempo real. Desta maneira, os programas de simulação realizam os cálculos da aplicação em questão – seja ela física, química, biológica, geográfica, etc. – e escrevem ou armazenam os resultados dos cálculos em tipos de arquivos que possam ser posteriormente traduzidos ou transformados em visualizações gráficas de simulação.

Após a aplicação da etapa de Processamento no cenário criado no Pré-processamento, o MEEP escreve os resultados dos cálculos do FDTD em HDF5, uma biblioteca que fornece um modelo de dados e um formato de arquivo livre para armazenamento e gerenciamento de dados. Desenvolvido pelo NCSA (*National Center for Supercomputing Applications* – Universidade de Illinois – Urbana-Champaign) para executar a entrada e saída de dados de forma flexível e eficiente em arquivos complexos e com grande volume de dados [19], os arquivos de extensão *.h5* armazenam os dados em matrizes, que podem assumir forma multidimensional. Para o método FDTD, que discretiza o espaço em uma malha uniforme de linhas (1D), retangular (2D) ou cúbica (3D), e o tempo (que pode assumir a 2ª, 3ª e ou 4ª dimensão), o armazenamento matricial torna-se interessante, pois cada unidade da malha geométrica e dos cálculos de propagação de ondas é guardada no respectivo elemento da matriz no arquivo *.h5*. Em outras palavras, a geometria – que pode ser representada por uma malha unidimensional, bidimensional ou tridimensional – e os cálculos de propagação de ondas – que podem assumir a forma de uma malha quadridimensional (espaço tridimensional mais o tempo) – no FDTD, têm as células de suas respectivas malhas associadas a cada elemento matricial presente no arquivo HDF5.

O MEEP gera duas saídas em *.h5*, que representam a geometria discretizada e os dados de propagação de ondas, ambos resultantes da etapa de Processamento. A quantidade de elementos nas matrizes de ambos arquivos depende da configuração do domínio computacional efetuado pelo usuário do sistema, que define a extensão de cada dimensão da simulação, a resolução – que é o fator de divisão do domínio computacional, incluindo a geometria da simulação – e a largura da PML (*Perfectly Matched Layer* [45]) para geração da malha. Em termos gerais, as PML´s são camadas definidas em todos os lados do domínio computacional para absorção das ondas que alcançam os limites do espaço finito (domínio).

Uma vez definida a resolução e os comprimentos do domínio computacional e da PML, o MEEP discretiza o cenário configurado, multiplicando cada unidade de comprimento do domínio e da PML pela resolução definida, isto é, uma única unidade de

comprimento do cenário completo irá conter uma quantidade de células de malha igual ao valor definido para a resolução. Assim, quanto maior for o valor da resolução, maior será a quantidade de células da malha discreta, sendo que se a resolução tender ao infinito, a discretização se aproximará cada vez mais do espaço real ou contínuo. Entretanto, deve se considerar que a capacidade de processamento e armazenamento em memória dos computadores atuais definirá o limiar máximo de discretização, pois uma malha com um número muito grande de células exigirá processadores e memórias dificilmente encontrados ou indisponíveis atualmente.

Os arquivos HDF5 resultantes da simulação no MEEP possuem, portanto, matrizes com a mesma dimensão das malhas geradas na discretização do cenário de simulação. Para verificar as dimensões de um arquivo *.h5*, pode-se utilizar o programa H5ls [47], que possui a funcionalidade de imprimir as dimensões de uma matriz contida em um arquivo HDF5.

Um cenário de Pré-processamento configurado em um domínio bidimensional de extensão igual à 2 no eixo *x* e 2 no eixo *y*, largura de PML igual à 0.5 e resolução igual a 10, irá gerar uma malha retangular de dimensão 25 x 25, pois o produto da resolução pelas extensões totais nos eixos *x* (2 de extensão mais 0.5 de PML) e *y* (2 de extensão mais 0.5 de PML) efetua uma quantidade de 25 *pixels* em cada dimensão. A Figura 3.1 ilustra um exemplo de Pré-processamento baseado nessas configurações. As dimensões do cenário gerado foram impressas pelo H5ls e podem ser averiguadas na Figura 3.2.

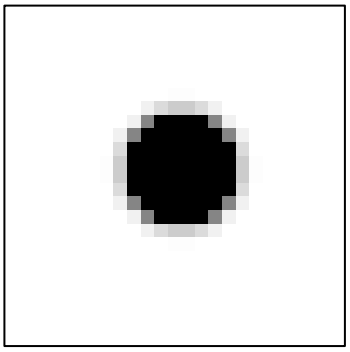


Figura 3.1 – Cenário de Pré-processamento discretizado em uma malha de 25 x 25 *pixels*.

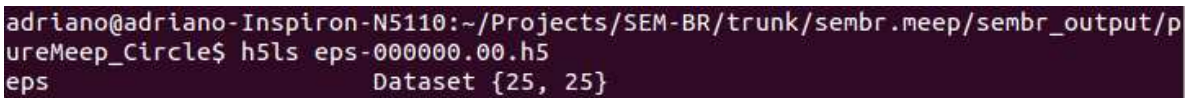


Figura 3.2 – Arquivo *eps-000000.00.h5* correspondente ao cenário de simulação (25 x 25 *pixels*).

Adicionalmente, o arquivo *.h5* corresponde aos cálculos de propagação de ondas inclui a dimensão tempo em sua matriz, que correspondente aos passos de tempo – definido pelo usuário – discretizados. O FDTD, caracterizado por resolver as equações de Maxwell no domínio do tempo, define uma quantidade de passos de tempo como um parâmetro para o intervalo de tempo finito da propagação de ondas na simulação.

Diferentemente de como ocorre na discretização do espaço de simulação, a discretização dos passos de tempo é efetuada através da divisão de cada unidade de tempo pelo fator de Courant [46], multiplicada pela resolução. No MEEP, o fator de Courant tem um valor padrão igual à 0.5 [46]. Caso fosse realizada uma simulação com 10 passos de tempo no domínio 2D anteriormente mencionado (2.5 x 2.5 e resolução igual a 10), o arquivo HDF5 resultante dos cálculos do FDTD teria duzentos elementos na dimensão relativa ao tempo, pois o produto entre a resolução e o resultado da divisão entre os passos de tempo pelo fator de Courant resulta em 200 (10 x 10 / 0.5 = 200).

A Figura 3.3 ilustra a execução do H5ls para obtenção das dimensões da matriz de geometria (arquivo *eps-000000.00.h5*), e a Figura 3.4 ilustra a execução do H5ls para impressão das dimensões da matriz com cálculos de campo elétrico resultantes do FDTD (arquivo *ez-000000.00.h5*). Ambos arquivos foram criados pelo MEEP em uma simulação 3D, que gerou uma matriz tridimensional com 11 elementos nas dimensões *x*, *y*, *z*, para a geometria discretizada, e uma outra matriz 4D com 11 elementos nas dimensões *x*, *y* e *z* e 200 elementos na dimensão *t* (tempo), para os cálculos de propagação de campo elétrico.

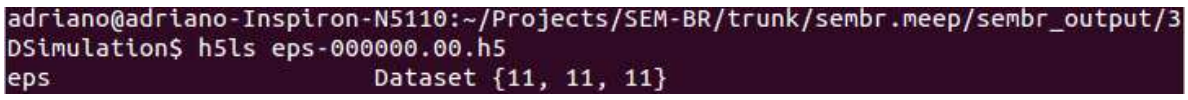


Figura 3.3 – *Script* h5ls: cenário de simulação com 11 x 11 x 11 elementos (eixos *x*, *y*, *z*).

```
adriano@adriano-Inspiron-N5110:~/Projects/SEM-BR/trunk/sembr.meep/sembr_output/3
DSimulation$ h5ls ez-000000.00.h5
ez                       Dataset {11, 11, 11, 200/Inf}
```

Figura 3.4 – *Script* h5ls: campos eletromagnéticos de diferentes passos de tempo. Matriz de 11 x 11 x 11 x 200 elementos (eixos *x*, *y*, *z* e o tempo).

3.2 UTILITÁRIO H5TOPNG

O H5topng, presente no pacote H5utils, permite obter imagens bidimensionais no formato PNG (*Portable Network Graphics*) a partir de arquivos no formato *.h5*. As imagens 2D obtidas representam fatias das matrizes presentes no *.h5*, e são formadas pela atribuição de cores aos elementos de dados correspondentes à determinada fatia da matriz. O H5topng é um programa que pode ser executado via comando ou *shell script* [48].

Basicamente, o H5topng oferece uma coleção de mapas de cores [49], que são passados no comando de conversão de dados em imagem. Os dados das matrizes são convertidos conforme a rampa de cores presentes nos mapas disponíveis pelo H5utils, que instala os mapas de cores no diretório */H5utils/colormaps* (sistema operacional Linux).

Cada mapa de cor é constituído por uma tabela de triplas ordenadas que representa a respectiva rampa de cores. Neste projeto, o mapa de cores definido é o *dkbluered*, que atribui níveis da cor azul ao campo eletromagnético negativo, níveis da cor vermelha ao positivo, branco ao neutro e preto à região ausente de campo eletromagnético.

O H5topng pode gerar imagens para cada fatia ou camada matricial presente no respectivo *.h5*. Por exemplo, um arquivo HDF5 que armazena uma matriz de dimensão 20 x 20 pode ser convertido em até 20 imagens correspondentes à primeira dimensão e outras 20 imagens correspondentes à segunda dimensão. O H5topng está restrito, portanto, em gerar imagens de camadas ou fatias que correspondam à planos ortogonais aos eixos das dimensões definidas. Imagens de planos transversais ou inclinados no espaço de simulação não podem ser obtidos com o H5topng.

O exemplo da Figura 3.5 apresenta um comando típico do H5topng. Para indicar a fatia a ser convertida em imagem, o utilitário oferece quatro variáveis, relativas às

dimensões *x*, *y*, *z* e *t* (tempo). Para as simulações 2D, os arquivos de matrizes bidimensionais podem ser diretamente convertidos em imagens. Assim, não há a necessidade de indicar ao comando a variável *x* ou *y* e a respectiva fatia de corte.

```
adriano@adriano-Inspiron-N5110:~/Projects/SEM-BR/trunk/sembr.meep/sembr_output/3
DSimulation$ h5topng -z 5 -c gray eps-000000.00.h5
```

Figura 3.5 – Conversão da geometria presente num arquivo *.h5* (fatia) em imagem PNG: a variável *z* indica a fatia de corte e a variável *c* indica a rampa de cor (neste caso, *gray* (cinza)) utilizada pelo H5topng.

Se a quantidade de passos de tempo foi definida pelo usuário na simulação, a variável *t* pode ser informada no comando de conversão do arquivo *.h5*, correspondente aos cálculos de propagação de onda, em imagem. Neste caso, obtém-se a fatia de tempo que determina o perfil do campo eletromagnético propagado no intervalo entre tempo inicial e o tempo determinado no comando. Nas simulações tridimensionais, deve-se indicar ao comando ao menos uma dimensão para obtenção de uma imagem bidimensional. Para os arquivos *.h5* de matrizes tridimensionais, é necessário indicar a fatia de corte espacial (Figura 3.5); para os que armazenam matrizes quadridimensionais, devem ser indicadas as fatias de corte espacial (*x*, *y* ou *z*) e temporal (*t*), conforme é mostrado na Figura 3.6.

```
adriano@adriano-Inspiron-N5110:~/Projects/SEM-BR/trunk/sembr.meep/sembr_output/3
DSimulation$ h5topng -y 5 -t 100 -Zc -dkbluered ez-000000.00.h5
```

Figura 3.6 – Fatia de uma matriz 4D presente num arquivo HDF5: especifica-se as fatias do eixo de corte (neste caso, eixo *y*) e do tempo (variável *t*) para obtenção de uma imagem através do H5topng.

O H5topng oferece a opção de conversão de várias fatias em imagens num único comando, indicando-se à variável dimensão a fatia inicial, final e um passo para obtenção das fatias que serão convertidas, em notação similar ao MATLAB® (*fatia_inicial:passo:fatia_final)* [48]. Se, por exemplo, deseja-se obter os perfis de campo eletromagnético dos primeiros 20 passos de tempo em uma matriz com 200 elementos na dimensão tempo, deve-se indicar à variável *t* o plano inicial de índice zero, o plano final de índice 19, e um passo de 1 fatia por vez, conforme ilustra a Figura 3.7. Esse procedimento pode ser aplicado em todas as dimensões de uma matriz presente em um arquivo HDF5.

```
adriano@adriano-Inspiron-N5110:~/Projects/SEM-BR/trunk/sembr.meep/sembr_output/3
DSimulation$ h5topng -y 5 -t 0:1:19 -Zc -dkbluered ez-000000.00.h5
```

Figura 3.7 – Obtenção de 20 imagens (fatias 0 à 19) em um único comando do H5topng: a variável *t* (neste caso) indica todos os planos de cortes.

O H5topng permite a criação de imagens que mostram a geometria de um arquivo *.h5* sobre o perfil eletromagnético de outro arquivo *.h5*. Em um único comando, indica-se a fatia de corte espacial para uma mesma região – em ambas matrizes – e o(s) passo(s) de tempo a ser(em) obtido(s). A sobreposição das imagens deve ser indicada pela opção *-a*, que define o mapa de cores que a geometria utilizará na sobreposição. Por padrão, a geometria que sobrepõe o perfil de campo eletromagnético é transparente, com fator de opacidade igual à trinta por cento (30%). A opção de comando *-A* deve complementar a opção *-a*, fornecendo o nome do arquivo *.h5* que contem a geometria a ser sobreposta. A Figura 3.8 mostra um comando do H5topng que sobrepõe as fatias adjacentes obtidas de matrizes de dois arquivos *.h5* distintos.

```
adriano@adriano-Inspiron-N5110:~/Projects/SEM-BR/trunk/sembr.meep/sembr_output/3
DSimulation$ h5topng -z 6 -t 100 -Zc -dkbluered -a yarg -A eps-000000.00.h5 ez-0
00000.00.h5
```

Figura 3.8 – Sobreposição entre a geometria (*eps-000000.00.h5*) e o perfil de campo correspondente (*ez-000000.00.h5*) através do H5topng.

As imagens de campos eletromagnéticos disponíveis no Pós-processamento deste projeto fazem uso da sobreposição da geometria disponível no respectivo HDF5.

3.3 GERAÇÃO DE FILMES 2D COM O MENCODER

As simulações bidimensionais podem ser visualizadas através de uma animação ou um filme 2D. Para isso, o usuário do MEEP precisa definir a quantidade de passos de tempo de propagação do campo eletromagnético e indicar quais fatias de dados de propagação de ondas serão armazenadas no arquivo HDF5. Através do H5topng, pode-se

gerar imagens bidimensionais que retratam a geometria simulada sob diferentes perfis de campo eletromagnético, através da conversão das fatias matriciais da dimensão tempo em imagens de campo eletromagnético, para diferentes instantes de propagação.

O Mencoder é um *software* codificador de filmes capaz de criar um arquivo de filme a partir das imagens PNG geradas pelo H5topng. Sua função é converter ou codificar a informação de um formato de arquivo em um outro formato. O Mencoder suporta a codificação de alguns tipos de imagens, como JPEG (*Joint Photographic Experts Group*) e PNG, em alguns tipos de filmes, como o AVI (*Audio Video Interleaved*) e MPEG *(Moving Picture Experts Group).*

Para operar o Mencoder, deve-se utilizar o *shell script* para inserção das expressões de comando. Os parâmetros de conversão das imagens em filme incluem o trecho padrão presente em todos nomes de arquivos a serem inseridos no filme, a altura, a largura e a quantidade de imagens exibidas por segundo (fps – *frames per second*), o formato original das imagens e o formato de filme a ser convertido. A Figura 3.9 ilustra um comando do Mencoder para conversão de imagens PNG em um vídeo MPEG.

```
adriano@adriano-Inspiron-N5110:~/Projects/SEM-BR/trunk/sembr.meep/sembr_output/3
DSimulation$ mencoder mf://ez-000000.00.t*.png -mf w=1280:h=720:fps=10:type=png
-ovc copy -oac copy -o ez.mpg
```

Figura 3.9 – Exemplo de um comando do Mencoder utilizado para conversão de imagens PNG em um vídeo MPEG: a opção *–mf* especifica a altura, a largura e o tipo da imagem, além da taxa de exibição de imagens por segundo.

3.4 VISUALIZAÇÃO DE PÓS-PROCESSAMENTO COM O VTK

As funcionalidades utilizadas do VTK para a implementação do Pós-processamento incluem a geração de volumes – a partir do empilhamento de imagens – e a configuração de planos para corte de volumes criados. Essas funcionalidades aplicam-se, portanto, à simulações tridimensionais.

O H5topng permite obter imagens PNG para todas fatias presentes em um arquivo

HDF5. Para formar um volume, todas as imagens geradas devem ser empilhadas em ordem crescente, de acordo com a indexação das fatias na matriz do HDF5. As fatias convertidas em imagens, para geração de um único volume, devem corresponder à configuração espacial de um mesmo passo de tempo. Volumes criados a partir de imagens de perfil de campo eletromagnético de diferentes passos de tempo geram informações e visualizações erradas, pois serão formados pela sobreposição de imagens que correspondem à diferentes instantes de propagação de onda ou diferentes perfis de campo eletromagnético.

3.4.1 Geração de Volumes com o *vtkPNGReader*

O VTK provê um conjunto de objetos-fonte para leitura de diferentes tipos de imagens, que podem ser configurados para gerar volumes a partir da combinação de imagens em pilhas. Para os arquivos de imagem PNG, o VTK disponibiliza a classe *vtkPNGReader* [28], que é uma especialização da classe base *vtkImageReader2*.

A geração de um volume pelo objeto *vtkPNGReader* pode ser feita através do fornecimento dos nomes de arquivos das imagens que serão empilhadas. A classe *vtkPNGReader* herda da super-classe *vtkImageReader2* o método *SetFileNames*, que recebe um parâmetro do tipo *vtkStringArray*. O objeto da classe *vtkStringArray* deve inserir todos os nomes de arquivos de imagens antes de ser passado, como argumento, à função *SetFileNames*.

Após carregar e empilhar as imagens na forma de volume, o objeto *vtkPNGReader* disponibiliza o *Dataset vtkImageData*, que fornece os dados de geometria, topologia e atributos ao *pipeline* de visualização. O *vtkImageData* é definido como um objeto de dados estruturados pelo VTK, implementado para suportar imagens bidimensionais (formada por *pixels*) e tridimensionais (formada por *voxels*) e caracterizado por representar as coordenadas de pontos (geometria) em uma grade regularmente retangular (topologia) e paralela ao sistema de coordenadas global (espaço cartesiano) [30]. Os atributos de dados presentes na instância do *vtkImageData*, referenciado pelo objeto *vtkPNGReader*, são

representados pelos valores escalares de cores gerados na execução do H5topng.

3.4.2 Mapeamento de Atributos de Dados em Cores

As cores apresentadas no ambiente gráfico do VTK normalmente são definidas pelo mapeamento de dados escalares em cores, em um processo similar ao realizado pelo H5topng. Para isso, o VTK disponibiliza o objeto *vtkLookupTable* [28], responsável em fornecer os mecanismos de mapeamento de dados em cores. Através do *vtkLookupTable*, é possível configurar a quantidade, os valores e o espaço ou rampa de cores – como o RGBA *(Red-Green-Blue-Alpha)* ou HSVA *(Hue-Saturation-Value-Alpha)* – e definir os limiares inferior e superior de mapeamento dos dados escalares. Os valores de escalares menores que o limiar inferior são mapeados para o valor mínimo da rampa de cores e os maiores para o valor máximo [28].

O mapeamento de dados escalares em cores é realizado por objetos do tipo *vtkMapper*, que utilizam a configuração do objeto *vtkLookupTable* especificado e os atributos de dados do *Dataset* para efetuar o mapeamento em primitivas gráficas. Uma instância da classe *vtkLookupTable* é criada pelo mapeador por padrão, caso não seja especificada. O mapeador genérico de um *Dataset (vtkDataSet)* é o objeto representado pela classe base *vtkDataSetMapper*.

Os atributos de dados escalares que representam diretamente os valores de cores em um objeto *vtkDataSet*, não necessitam ser mapeados através de uma instância do *vtkLookupTable*. Neste caso, o mapeador é configurado para realizar o mapeamento padrão, tratando os escalares como tipo de dados *unsigned char* e utilizando os valores diretamente como cores [28].

A definição das cores das imagens carregadas no *vtkPNGReader* pelo H5topng elimina a necessidade de utilização da classe *vtkLookupTable*. O mapeador, representado pela instância da classe *vtkDataSetMapper*, é configurado para o modo de coloração padrão, mapeando diretamente os dados escalares da instância *vtkImageData*, presente no

objeto *vtkPNGReader*, como valores de cores. O método disponível na classe *vtkDataSetMapper* que configura o mapeamento direto de dados escalares em cores é o *SetColorModeToDefault*.

3.4.3 *Widgets* Tridimensionais

Uma das maneiras de prover meios de interação entre o sistema gráfico e o usuário é através dos *Widgets* 3D [28], disponíveis no *framework* do VTK. No Pós-processamento deste trabalho, foram utilizados os tipos de objetos *vtkImplicitPlaneWidget2* e *vtkImagePlaneWidget*. Junto com os objetos de interação *(VTK Interactors)* [28], os *Widgets* 3D formam o conjunto de objetos do VTK para interação gráfica com o usuário.

Em termos gerais, os mecanismos de interação gráfica fornecidos pelo VTK são executados a partir da observação de eventos realizados pelo usuário na janela gráfica. Na tentativa de interagir com o sistema gráfico, o usuário maneja um dispositivo de entrada do computador – como o *mouse* e o teclado – e envia comandos específicos que são captados, na forma de eventos, pelo objeto de interação na janela de renderização.

O objeto de interação *vtkRenderWindowInteractor*, presente na janela de renderização representada pelo objeto *vtkRenderWindow*, é o responsável em captar as mensagens dos dispositivos de entrada e invocar respostas aos eventos solicitados. Objetos do tipo *vtkInteractorObserver* observam os eventos invocados e respondem às solicitações interagindo com os atores da cena. As classes que compõem o conjunto de *Widgets* 3D no VTK são especializações da super-classe *vtkInteractorObserver.*

A principal diferença entre os objetos de interação *(VTK Interactors)* e os *Widgets* 3D é em relação à representação gráfica. Os objetos de interação não são visíveis na janela de renderização; para manipulá-los, o usuário deve conhecer os comandos e executá-los a partir dos dispositivos de entrada. Os *Widgets* 3D são representados graficamente e podem ser diretamente manuseados, facilitando a interação com o usuário e permitindo tipos de interações mais complexas.

3.4.4 Planos de Cortes de Volumes com o *vtkImplicitPlaneWidget2* e o *vtkCutter*

Uma das maneiras de se definir planos pelo VTK é através da classe *vtkImplicitPlaneWidget2* [28]. Este *Widget* 3D orienta e posiciona um plano ilimitado sobre uma região na forma de caixa. O plano é perfurado por uma flecha perpendicular, que permite seu manuseio durante a interação com o usuário. Objetos do tipo *vtkImplicitPlaneWidget2* são tipicamente usados para realização de cortes [28].

Para utilizar um objeto *vtkImplicitPlaneWidget2*, é necessário configurar um objeto da classe *vtkImplicitPlaneRepresentation*, responsável em definir a representação de um plano infinito para o objeto *vtkImplicitPlaneWidget2*. As propriedades de origem e normal do plano e a propriedade de posição inicial de um objeto *vtkImplicitPlaneWidget2* são definidas no objeto *vtkImplicitPlaneRepresentation*, através dos métodos *SetOrigin*, *SetNormal* (ambas funções recebem parâmetros de coordenadas tridimensionais) e *PlaceWidget* (recebe as coordenadas dos limites de extensão de uma região na forma de caixa), respectivamente. Antes de ser executado, um objeto *vtkImplicitPlaneWidget2* deve ser relacionado, ainda, à instância do objeto de interação *vtkRenderWindowInteractor*, através do método *SetInteractor*, e ser invocado para execução através do método *On*.

Ao interagir graficamente com um objeto do tipo *vtkImplicitPlaneWidget2*, o usuário pode orientar e posicionar o plano em outras regiões. A posição atual do plano pode ser obtida através da chamada do método *UpdatePlacement*, disponível na classe *vtkImplicitPlaneRepresentation*. A atualização de localização atribui novos valores às propriedades de origem e normal do plano representado pelo objeto *vtkImplicitPlaneRepresentation*.

A atualização das informações de posicionamento de um plano, de forma interativa, propicia a realização de cortes arbitrários em um ator delimitado pela região do objeto *vtkImplicitPlaneWidget2*. Uma das maneiras de se realizar cortes no VTK é através da classe *vtkCutter* [28].

Para efetuar cortes em atores de uma cena, um objeto da classe *vtkCutter* necessita receber informações do plano de corte e do *Dataset* a ser cortado. O plano recebido pela

classe *vtkCutter* é do tipo *vtkPlane*, caracterizado por ser uma função implícita no VTK *(vtkImplicitFunction)*, isto é, uma função real do tipo $w = F(x, y, z)$. Um objeto *vtkPlane* pode ser configurado com os dados de origem e normal de um plano – através dos métodos *SetOrigin* e *SetNormal* – e passado, como parâmetro, ao método *SetCutFunction* da classe *vtkCutter*.

O *Dataset vtkImageData*, criado pelo objeto *vtkPNGReader*, pode ser conectado à um objeto *vtkCutter* e ser cortado pelo plano posicionado arbitrariamente pelo usuário. O objeto *vtkCutter* gera, como saída, um plano de imagem na forma de *Dataset* do tipo *vtkPolyData* (dados poligonais). A imagem adquirida é resultado da interpolação realizada sobre os dados de atributos da instância *vtkImageData* presentes na região de corte [28, 30].

3.4.5 Planos de Cortes de Volumes com o *vtkImagePlaneWidget*

A visualização de dados em volumes pode ser realizada pela classe *vtkImagePlaneWidget* [28], um tipo de *widget* 3D do VTK que permite a manipulação de planos de cortes em volumes e a visualização de dados de atributos em forma de imagem sobre a região planar de corte. Os planos, ortogonais ao eixos *x*, *y* e *z*, podem percorrer a extensão total do volume e posicionar-se sobre qualquer imagem presente na pilha. O ator, representado pelo objeto gráfico (volume) em interação com o objeto do tipo *vtkImagePlaneWidget*, não aparece completamente na cena; ele é visível somente através das imagens geradas sobre os planos de cortes.

As cores, representadas pelos dados de atributos das imagens que formam o volume, não são diretamente mapeadas pelo objeto *vtkImagePlaneWidget*. O mapeamento deve ser realizado pelo objeto da classe *vtkImageToColors*, que recebe a instância do *vtkImageData* corrente – como entrada – para posteriormente ser passado como argumento ao método *SetColorMap*, disponível na classe *vtkImagePlaneWidget*.

Um objeto do tipo *vtkImagePlaneWidget* deve ser relacionado, através do método *SetInteractor*, ao objeto de interação *vtkRenderWindowInteractor*, instanciado na cena

gráfica, e ser conectado ao *Dataset (vtkDataSet)* que representa o volume da interação. Para iniciar a execução, o objeto *vtkImagePlaneWidget* deve ser configurado para o modo de interação, através do método *InteractionOn*, e invocado para execução pelo método *On*.

3.4.6 Legenda de Dados de Campos Eletromagnéticos com o *vtkScalarBarActor*

Para relacionar dados escalares com cores, o VTK disponibiliza a classe *vtkScalarBarActor* [28]. A classe, representada graficamente por um ator, forma uma legenda constituída de três partes: barra de cores, título e valores de dados escalares mapeados.

A barra de cores é configurada por um objeto *vtkLookupTable*, que pode fornecer a rampa de cores a ser representada na legenda. Os valores disponíveis nas rampas de cores do H5utils podem ser atribuídos ao *vtkLookupTable*, através do método *SetTableValue*. A instância da classe *vtkScalarBarActor* recebe o objeto *vtkLookupTable* pelo método *SetLookupTable*.

Os dados de campo eletromagnético podem ser adquiridos na execução do H5topng, incluindo-se a opção *-v* (saída verbosa) ao comando de conversão. Através dos dados de campo, pode-se obter o menor e o maior valor de campo eletromagnético da simulação e utilizá-los na configuração do método *SetRange* da classe *vtkLookupTable*, que relacionará o intervalo de dados do campo eletromagnético com as cores da barra de forma automática.

CAPÍTULO 4

PROJETO DO AMBIENTE COMPUTACIONAL

Este capítulo apresenta o projeto do ambiente computacional de simulação eletromagnética desenvolvido neste trabalho. A arquitetura do sistema é mostrada e descrita pelos componentes de *software* implementados. São apresentadas, também, as principais funcionalidades e relações de dependência dos pacotes Java, implementados nos componentes deste sistema, bem como a integração entre o ambiente computacional e o MEEP. Os desenhos do *software* são representados em diagramas UML [50], que fornece todos os meios de formalização de sistemas computacionais.

4.1 INTRODUÇÃO

Existe, atualmente, uma lista grande de simuladores eletromagnéticos disponíveis. Alguns desses simuladores proveem suporte a variados tipos de funcionalidades nas diferentes etapas de processamento de simulação – Pré-processamento, Processamento e Pós-processamento – e oferecem recursos sofisticados que aprimoram, cada vez mais, as simulações. *Softwares* como o COMSOL *Multiphysics*® [51] e o CST *Studio Suite*® [52] são exemplos de simuladores conhecidos na academia e na indústria por disponibilizarem um conjunto grande e diversificado de recursos para elaboração de simulações mais complexas e de grande porte. Esses sistemas são comerciais e, apesar de serem mais completos em relação aos não-comerciais, demandam alto custo de aquisição. Deste modo, a instalação deste tipo de *software* torna-se, em certos casos, difícil ou inviável.

Existem, também, opções de programas de simulação eletromagnética livres e de código aberto, como o emGine [53], MMTL *Electromagnetic Simulator* [54], MEEP e outros. Por não serem comerciais, estes programas podem ser adquiridos gratuitamente e,

por possuírem código aberto, podem ser alterados por outros desenvolvedores. Contudo, esses programas oferecem funcionalidades mais simples em relação àqueles comerciais.

Para a implementação deste ambiente computacional, considerou-se os critérios de aquisição gratuita e de extensão de *softwares*, com a finalidade de manter um sistema apropriadamente extensível e com acesso livre de uso. Esses critérios alinham-se com as características dos grupos de pesquisa do DECOM-UNICAMP, que constantemente efetuam pesquisas através de simulações eletromagnéticas e, por isto, demandam códigos extensíveis para implementação de funcionalidades específicas e liberdade de acesso. Dentre as opções de simuladores eletromagnéticos livres e de código aberto, optou-se pelo MEEP, devido à três fatores principais: referência como simulador de código aberto, aplicabilidade em Fotônica e processamento paralelo.

O MEEP, por ser um sistema de código aberto e orientado à objetos, é originalmente extensível. Programas de código aberto podem ter seus códigos-fonte vistos, modificados e estendidos sob as normas de uma licença de *software* livre, como a GPL, utilizada pelo MEEP. Sistemas orientados à objetos têm, como uma das características fundamentais, a capacidade de conceber meios de abstração de funcionalidades, de forma a torná-las base de outras funcionalidades por meio da extensão.

Os grupos de pesquisa relacionados ao DECOM, como o CEPOF e o FOTONICOM, têm desenvolvido pesquisas na área de Fotônica, o que torna o MEEP de grande utilidade, pois sua aplicabilidade pode ser estendida com o MPB *(*MIT *Photonic Bands),* um pacote complementar ao MEEP para simulações eletromagnéticas em Fotônica. Além disso, ambos sistemas foram desenvolvidos para suportar processamento paralelo em *clusters* de computadores através do MPI *(Message Passing Interface)* [55].

Este projeto visa criar um envoltório de funcionalidades em torno do MEEP, com o propósito de estender suas capacidades funcionais e torná-lo apto à realização de simulações eletromagnéticas mais elaboradas e de grande porte.

4.2 DEFINIÇÃO DO PROJETO DE *SOFTWARE*

Os critérios de extensibilidade de sistema motivam, diretamente, a definição de um projeto estruturado em módulos, que encapsulam unidades funcionais capazes de serem reusáveis – eliminando o retrabalho de implementação de funções – e extensíveis – abstraindo funções para torná-las base ou interface de outras funcionalidades específicas.

O paradigma de Programação Orientada à Objetos pode disponibilizar os meios de sistematização modular. A unidade básica do paradigma – o objeto – é uma instância de uma estrutura de programação conhecida como classe, responsável em abstrair um conjunto de objetos com aspectos similares. A classe busca catalogar funções e dados semanticamente relacionados e encapsulá-los em uma estrutura. Em outras palavras, a classe define o comportamento de seus objetos por meio de funções e os estados por meio de atributos de dados.

As classes de objetos podem estabelecer relação com outras classes, de modo que o relacionamento entre elas pode gerar funcionalidades de alto nível, criando estruturas lógicas mais abstratas e globais. Uma ou mais classes de objetos podem ser ligadas para formar um componente de sistema, uma unidade modular independente que provê um conjunto de operações por meio de uma interface.

O ambiente computacional deste projeto é definido em termos de componentes de *software*. Uma simulação eletromagnética pode envolver um conjunto de funcionalidades de diferentes aspectos de computação, como as unidades de cálculo numérico, de computação gráfica, de banco de dados, de persistência de dados, de entidades de dados, de leitores de arquivos, de relatórios, de interface gráfica de usuário, entre outros. Codificar um sistema de muitas características induz sua modularização, sem a qual torna-se inviável o gerenciamento do mesmo. Os aspectos de reuso, manutenção e extensão tornam-se possíveis quando o sistema é organizado de forma modular.

Os componentes deste sistema buscam prover funcionalidades definidas conforme as similaridades de características operacionais. Para efetivar os requisitos propostos ao ambiente computacional, as funcionalidades implementadas são modularizadas nos

seguintes componentes: apresentação (interface de visualização gráfica e de usuário), visualização e computação gráfica, leitor DXF, geometria (interface entre dados de arquivos gráficos e VTK), processamento (modularização do MEEP), Pós-processamento (modularização do H5topng e Mencoder), persistência de dados e entidades de dados de sistema. Dessa forma, o *software* é estruturado em multicamadas ou *n*-camadas [56], cada qual representada por um componente. As camadas devem ser flexíveis de modo que a alteração de uma delas não afete o funcionamento das outras camadas.

O ambiente computacional atual é uma aplicação *desktop*, cuja execução é realizada localmente em um computador de forma autônoma e cuja instalação é efetivada para cada máquina. Entretanto, a portabilidade da aplicação *desktop* para a *Web*, por exemplo, pode ser realizada pela mudança do componente que comporta a interface gráfica *desktop* por um componente que comporta uma interface *Web*.

A definição e relação dos componentes são apresentadas detalhadamente a seguir, assim como a descrição das técnicas de Orientação à Objetos utilizadas. Antes, serão apresentados os sistemas que dão suporte à este ambiente computacional.

4.3 SISTEMAS DE SUPORTE AO AMBIENTE COMPUTACIONAL

Assim como o MEEP, a primeira versão do ambiente computacional foi desenvolvida no sistema operacional Unix. Para instalar o ambiente em outros sistemas operacionais, deve-se, primeiramente, compilar o MEEP para outros sistemas operacionais. Não há uma versão do MEEP para Windows. A opção pode ser a utilização do Cygwin [57], que fornecerá a interface Linux ao MEEP instalado no Windows.

A codificação do sistema é praticamente baseada na linguagem Java. O Java provê um amplo *framework* para desenvolvimento em diversas frentes de sistemas, como os *Web*, *Mobile*, *Desktop*, aplicações distribuídas e embutidas, Java 3D, entre outros. O Java é atualmente um sistema da Oracle® [58], mas que pode ser usado livremente. Esses fatores são parte daqueles que influenciaram a escolha desta linguagem.

O grupo do DECOM possui outros trabalhos baseados em Java [59], e a experiência na codificação de sistemas com esta linguagem também influenciou sua escolha. Além disso, o VTK possui a classe *vtkPanel*, que pode ser diretamente acoplada ao componente *JPanel* do Java e ser fixada na interface gráfica do usuário. Esta opção não é disponível, por exemplo, na versão C++ do VTK.

Como mencionado neste trabalho, o VTK é o *framework* base de visualização e computação gráfica deste ambiente computacional. Por ser um sistema robusto, consistente e amplo, que fornece inúmeras funcionalidades, desde a visualização gráfica até a computação geométrica – além de fornecer, amplamente, tutoriais, livros e listas de e-*mails* que dão suporte à codificação – o VTK foi escolhido como sistema de Visualização e Computação Gráfica.

O MEEP é o núcleo deste ambiente computacional. As funcionalidades implementadas buscam a integração com a interface do MEEP, assim como as entidades de sistema desenvolvidas buscam formar um mapa dos objetos de dados do MEEP. O H5utils é a referência de utilização após o processamento do MEEP, e pode prover o suporte básico para execução de um Pós-processamento mais elaborado, conforme demonstra este trabalho no próximo capítulo.

A primeira versão deste sistema, desenvolvido sobre o Linux, utiliza aqueles programas que são executados nativamente via comandos. O Mencoder é um *software* executável via comandos e, por esta facilidade, é o programa que gera filmes 2D no ambiente computacional. Para executar o filme, utiliza-se o *software* VLC, que tem suporte em Java através da biblioteca VLC Java *Library* [61]. O componente executa a aplicação VLC e exibe o filme gerado pelo Mencoder.

Para auxiliar no gerenciamento do ambiente computacional, todos fontes foram armazenados no repositório do Google *Code* [61], um controlador de versão de *softwares* que gera versões automáticas para as submissões de código novo e alterado. Com o sistema em repositório, muitos desenvolvedores podem trabalhar ao mesmo tempo, em um sistema que bloqueia arquivos em edição (por usuário), controlado pelo próprio repositório. O ambiente computacional pode ser adquirido em [62], somente em modo de leitura. Para

alterá-lo, o desenvolvedor deve ser cadastrado pelo administrador do repositório.

4.4 COMPONENTE DE INTERFACE GRÁFICA DE USUÁRIO

O componente de interface gráfica de usuário, também conhecida como GUI *(Grafical User Interface)*, encapsula as rotinas que envolvem a interação direta com o usuário do sistema. A GUI é organizada em diferentes janelas de interface de usuário, que permitem organizar e separar as diferentes funcionalidades que o sistema disponibiliza.

Todos os componentes gráficos disponíveis na GUI são fornecidos pelo *Swing Application Framework* [63], uma API *(Application Programming Interface)* para interface gráfica que provê componentes mais sofisticados em relação ao antigo conjunto de ferramentas AWT *(Abstract Window Toolkit)* [64] do Java. O *Swing* – como é normalmente referenciado – provê um conjunto de componentes, como janelas, botões, painéis, campos de texto e de marca, menus, barras de progressão, tabelas, entre outros, para dar suporte à construção da GUI.

Para ilustrar as principais operações realizadas na interface gráfica de usuário, pode-se utilizar o diagrama da UML conhecido como Casos de Uso [56]. Através dele, é possível visualizar a interação entre usuário – representado por um ator – e as funções acessíveis do sistema que serão operadas pelo ator, representada em balões. A Figura 4.1 mostra o caso de uso com as operações predominantes nesta versão do ambiente computacional.

As operações disponíveis no ambiente computacional estão divididas, atualmente, em sete janelas diferentes: a janela principal, que comporta todas operações de simulação; a janela inicial, que permite ao usuário criar ou abrir um projeto de simulação; a janela de inserção e configuração da geometria básica representada pelo cubo; a janela de inserção e configuração da geometria básica representada pela esfera; a janela de escolha do tipo de Pós-processamento; a janela de configuração de Pós-processamento; e a janela de criação de um projeto novo.

A janela inicial, comportada pela classe *InitialWindow.java* e ilustrada na Figura

4.2, permite ao usuário criar um novo projeto de simulação ou abrir um já existente. Nesta janela, o ambiente computacional também pode ser fechado.

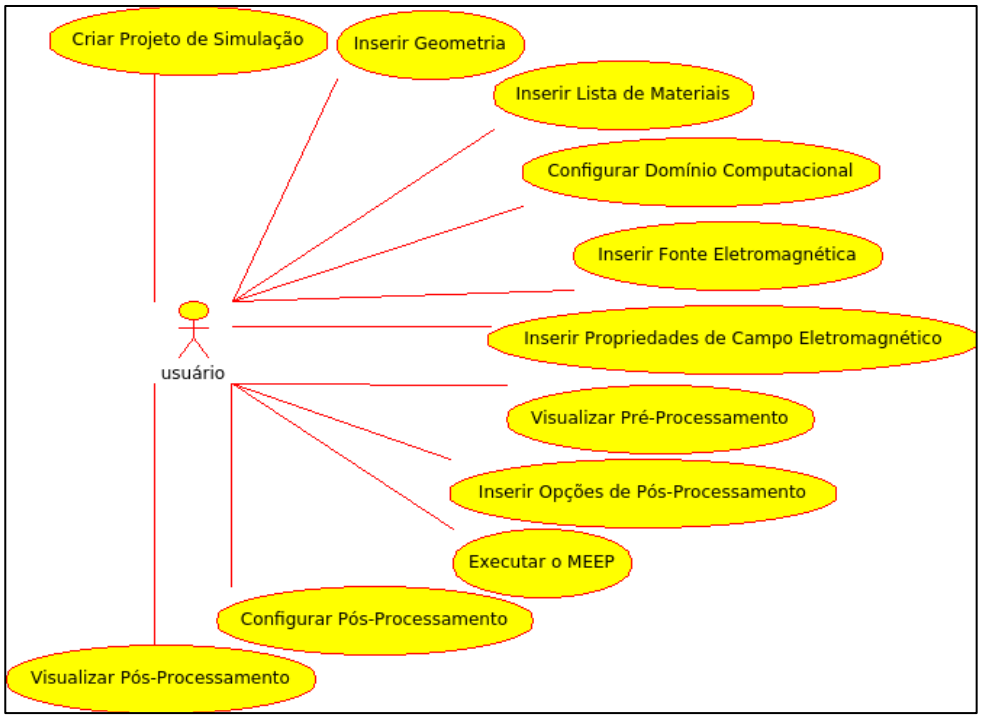


Figura 4.1 – Casos de uso com as principais operações do ambiente computacional.

Figura 4.2 – Janela inicial do ambiente.

Todas as operações que o usuário pode executar no simulador são acessíveis por botões, exceto as operações de interação com os objetos gráficos, que são realizadas via *mouse* ou teclado. Essas operações são orientadas à eventos, isto é, a ação do usuário sobre os objetos – botões e objetos gráficos – invoca funções na forma de eventos.

Para criar um projeto novo, o usuário deve pressionar o botão indicado em *"Novo projeto de simulação"*, que abrirá, em seguida, a janela de criação de um novo projeto. Esta janela está acoplada dentro da janela inicial, conforme mostra a Figura 4.3. Depois de definir o nome e o diretório de armazenamento do projeto, o usuário poderá efetivar a criação de um projeto novo pressionando o botão *"Confirmar"*.

Figura 4.3 – Janela de criação de um novo projeto acoplada à janela inicial.

Se optar por abrir um projeto criado anteriormente, o usuário deve pressionar o ícone indicado por *"Abrir projeto de simulação"*. A aplicação abrirá uma janela de diretórios para que o usuário possa escolher o caminho do projeto, conforme apresenta a Figura 4.4. O usuário deve selecionar a pasta que possui o nome do projeto escolhido e pressionar *"Open"*.

As operações que criam ou abrem um projeto de simulação direcionam, posteriormente, a aplicação para a janela principal do simulador. Para fechar a aplicação, o usuário deve pressionar o botão indicado por *"Sair do simulador"*.

Figura 4.4 – Escolha de um projeto de simulação existente na janela de diretórios.

Para operar a simulação, o ambiente computacional provê a janela principal, comportada na classe *MainPresentation.java* e ilustrada na Figura 4.5.

No Pré-processamento, o usuário deve fornecer as seguintes entradas para prover os dados adequados para processamento do MEEP: geometria e materiais, domínio computacional de simulação, largura de PML, resolução, configuração de fonte e de propriedades de campo eletromagnético. O diagrama de Casos de Uso da Figura 4.6 ilustra as operações que o usuário deve realizar para efetivar a configuração do Pré-processamento.

Os principais botões de acesso, utilizados na janela principal para inserir geometrias em Pré-Processamento, são ilustrados na Figura 4.7.

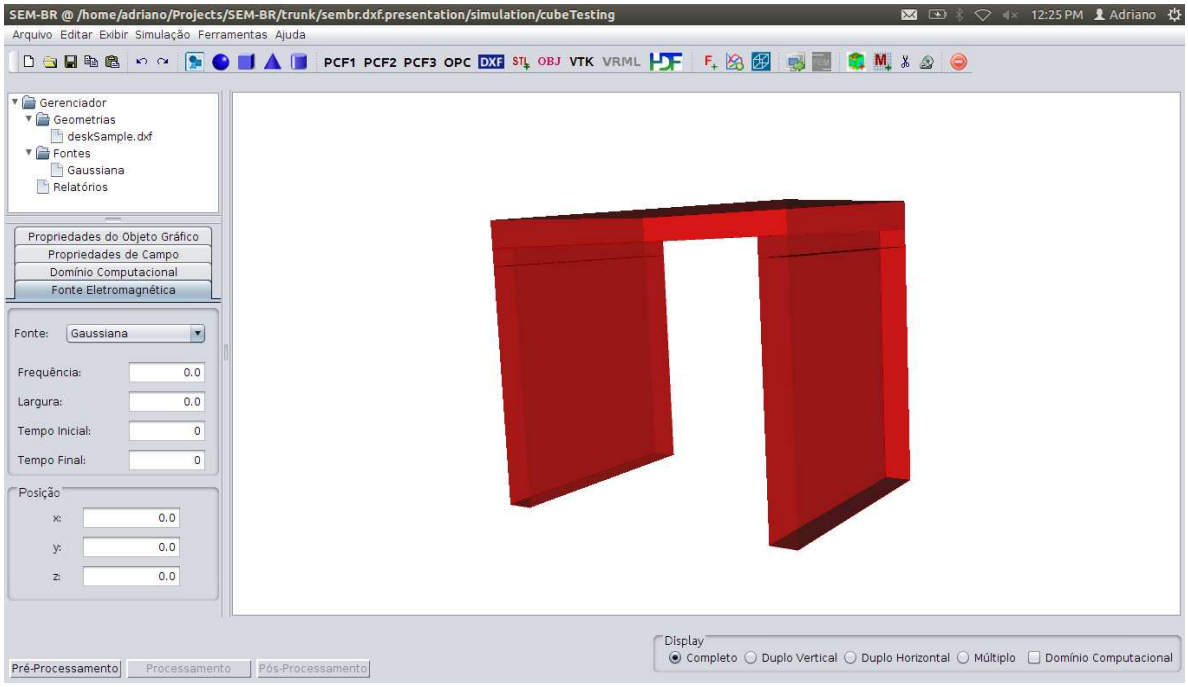


Figura 4.5 – Janela principal do ambiente.

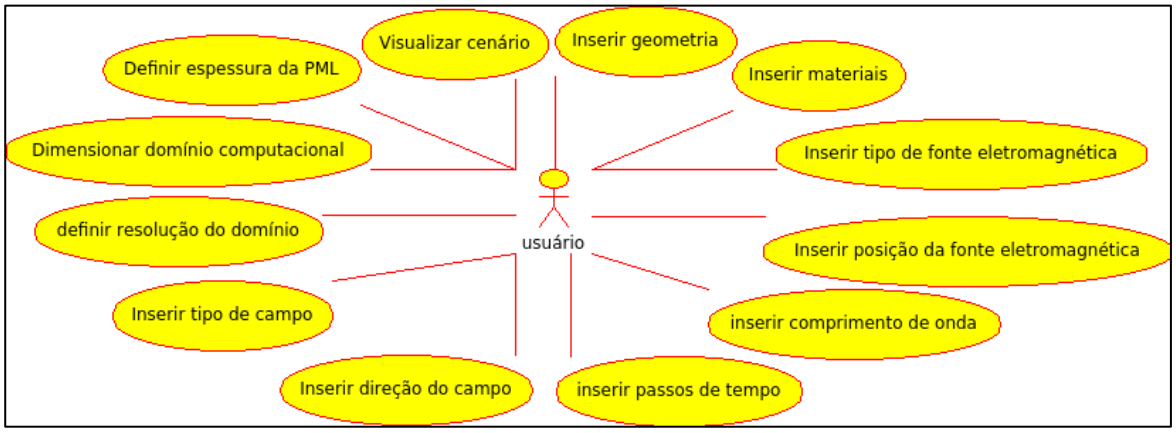


Figura 4.6 – Diagrama de Casos de uso referente à configuração do Pré-processamento.

Figura 4.7 – Botões para inserção de Geometrias, respectivamente: Adicionar Esfera, Cubo, Cone, Cilindro, PCF1, PCF2, PCF3, OPC, DXF, STL, OBJ, VTK.

Conforme o usuário insere os artefatos de simulação, um modelo na forma de árvore de diretório (componente *JTree*) é atualizado para ilustrar as escolhas do usuário. O modelo de árvore de diretório é ilustrado na Figura 4.8.

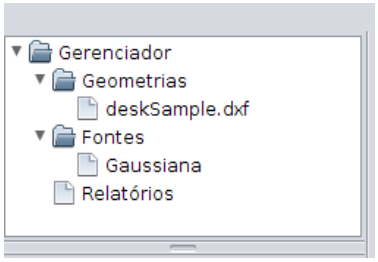


Figura 4.8 – Modelo de árvore de diretórios disponível na tela principal.

As janelas de inserção do cubo e da esfera, ilustradas nas Figuras 4.9 (a) e 4.9 (b), podem ser abertas através dos botões *"Adicionar Cubo"* e *"Adicionar Esfera"* (Figura 4.7). O cubo é configurado pelos limites de coordenadas dos eixos nas três dimensões; a esfera é definida pelo comprimento do raio e pelas coordenadas do centro.

(a) (b)

Figura 4.9 – (a) Janela para criação do cubo; (b) janela de criação da esfera.

Os outros modos de inserção de geometrias são acessíveis pelos diferentes botões disponíveis. Para introduzir geometrias de arquivos DXF, STL, OBJ e VTK (Figura 4.7), o usuário deve pressionar os respectivos botões rotulados com os nomes das extensões

disponíveis. Uma caixa de escolha de diretório será aberta para que o usuário procure pelos arquivos.

Para o Pré-processamento, existem ainda as opções de geometrias de cristais fotônicos pré-definidos. Para isso, o usuário deve escolher os diferente modelos, rotulados pelos seguintes botões (Figura 4.7): PCF1, PCF2, PCF3 *(Photonic Crystal Fiber)* e OPC *(Optimized Photonic Crystal)*. Após pressionar o botão do cristal fotônico escolhido, a respectiva geometria será carregada e visualizada na GUI.

As propriedades de domínio computacional, fonte eletromagnética, campo eletromagnético e objeto gráfico estão disponíveis em um painel de abas (componente *JTabbedPane*), cada qual em uma aba distinta. Os materiais estão disponíveis na aba de propriedades do objeto gráfico, através de uma tabela (componente *JTable*) que recebe os nomes de materiais e os valores correspondentes à propriedade dielétrica dos respectivos materiais (índice de refração). O conjunto de abas, com destaque à aba de propriedades do objeto gráfico, é apresentado na Figura 4.10.

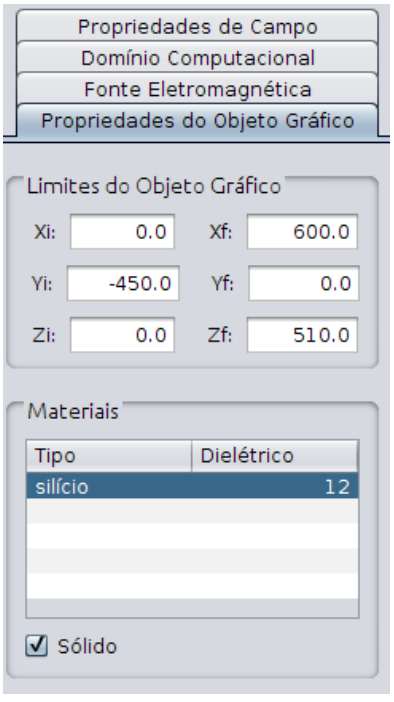


Figura 4.10 – Abas de configuração de Pré-processamento. Na aba de propriedades do objeto gráfico, é inserida a lista de materiais (tabela inferior).

As opções de Pós-processamento – apresentadas na janela de relatórios de Pós-processamento da Figura 4.11 – devem ser escolhidas na fase do Pré-processamento, para que o MEEP realize e armazene a simulação corretamente.

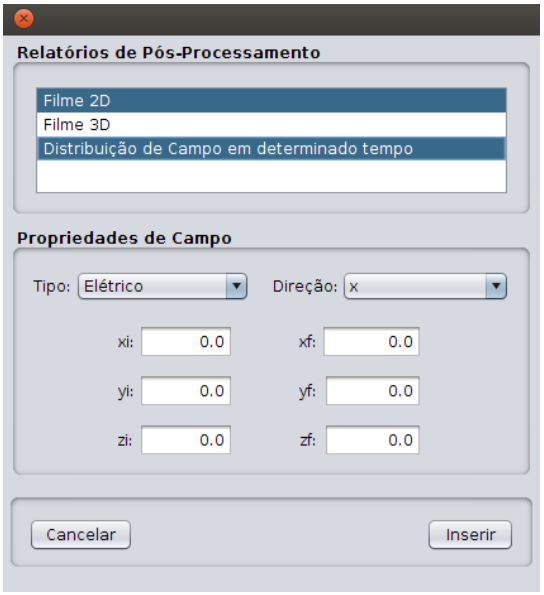


Figura 4.11 – Janela para escolha de Pós-processamento.

A janela de opções de Pós-processamento pode ser acessada pelo botão ilustrado na Figura 4.12 (a). Após configurar o Pré-processamento, o usuário pode visualizar o cenário criado pressionando o botão *"Visualizar Domínio Computacional"*, ilustrado na Figura 4.12 (b).

(a) (b)

Figura 4.12 – Ícones de acesso às janelas de opções de Pós-processamento (a) e visualização do cenário de simulação (b).

A visualização do domínio configurado apresenta um cenário que contem a geometria, uma caixa pequena roxa (representando a fonte), uma caixa exterior e transparente mostrando os limites do domínio computacional e uma régua exterior ao

domínio, de dois ou três eixos (simulação 2D e 3D, respectivamente), com uma diferença de distância do domínio igual à espessura da PML. A Figura 4.13 ilustra um cenário completo de simulação.

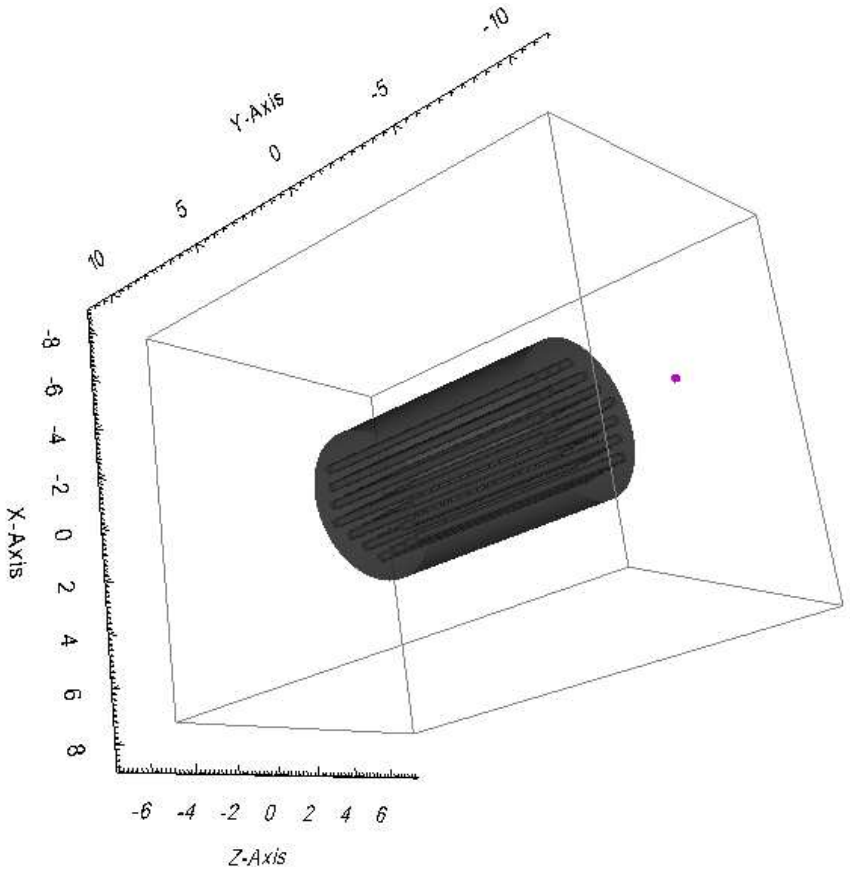
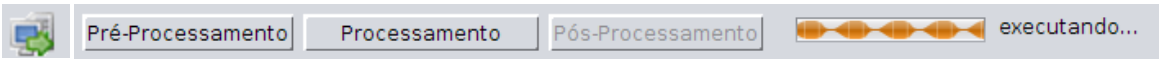


Figura 4.13 – Exemplo de um cenário de simulação tridimensional no ambiente computacional.

Se o cenário configurado corresponder àquele desejado pelo usuário, ele pode efetivar a execução da etapa de Processamento pressionando o botão *"Simular FDTD"*, ilustrado na Figura 4.14 (a). O *status* da simulação, apresentado na Figura 4.14 (b), pode ser verificado nas marcas (componentes *JLabel*), que indica(m) o(s) estágio(s) de processamento já executados. Os estágios ainda não executados tem suas marcas desabilitadas. O MEEP será instanciado e aparecerá, na GUI, uma barra de progresso – representada pelo componente *JprogressBar* – informando ao usuário que o processamento está em andamento, conforme mostra a Figura 4.14 (b).



(a) (b)

Figura 4.14 – (a) Botão que executa o MEEP; (b) Marcas que indicam o estágio de simulação e a barra de progresso indicando que o MEEP está em execução.

A barra de progresso é executada, juntamente com o MEEP, em uma *thread* separada. *Threads* [7] abrem uma nova unidade de execução a partir do processo responsável pela aplicação que está sendo executada. O processo sempre instancia um programa em uma *thread*; a criação de uma nova *thread* indica que o processo possui duas *threads*. Essa característica é particularmente interessante na GUI deste ambiente computacional, pois, ao prover mecanismos de *multithreading* [7] através da classe *SwingWorker* [63], a execução do FDTD pelo MEEP é realizada em uma *thread* separada daquela que executa o ambiente e, portanto, o usuário pode utilizar a janela principal enquanto o MEEP é executado. Se este mecanismo *multithreading* não fosse utilizado, a janela principal permaneceria travada durante a execução do MEEP. O diagrama de atividades da UML, mostrado na Figura 4.15, pode ilustrar o fluxo de atividades do ambiente computacional na execução da etapa de processamento. Pode-se notar que depois de efetivar o Processamento na atividade *"Simular FDTD"*, o sistema abre duas outras atividades (*"Manter execução da tela principal"* e *"Executar MEEP"*) que se realizam em paralelo. Posteriormente, o fluxo é juntado e apenas uma *thread* permanece em execução. A implementação atual do ambiente computacional não suporta, entretanto, a execução de múltiplas simulações (esta capacidade pode ser desenvolvida nas versões subsequentes deste sistema).

Depois de aplicar o FDTD no cenário configurado, o usuário pode configurar o Pós-processamento anteriormente escolhido, clicando sobre a marca *"Pós-processamento"* (Figura 4.14 (b)), que é habilitada após a execução do MEEP. O simulador abrirá a janela de configuração de Pós-processamento, mostrada na Figura 4.16. Se o usuário optou por visualizar o perfil de campo eletromagnético em um determinado tempo, ele deverá gerar apenas imagens do volume de um determinado passo de tempo. Na opção de filme 2D, o

usuário poderá definir o intervalo de passos de tempo, além da largura, altura e quantidade de *frames* por segundo do filme que será gerado.

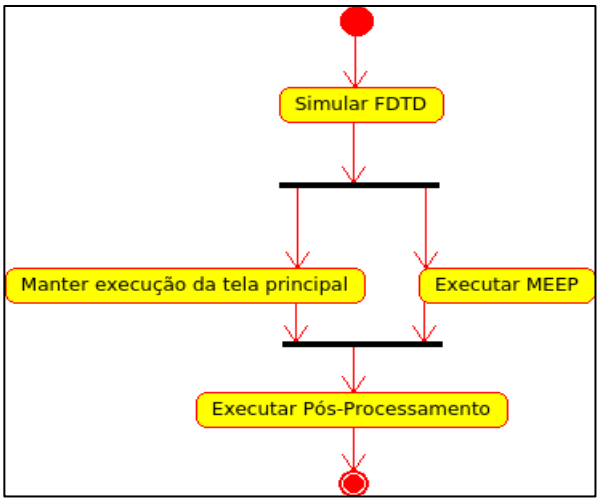


Figura 4.15 – Diagrama de Atividades que representa a etapa de Processamento do ambiente.

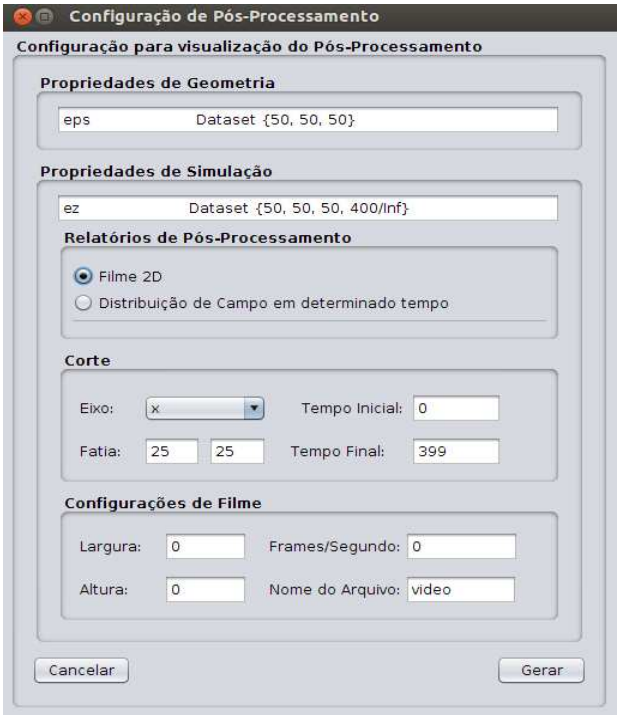


Figura 4.16 – Tela de configuração de Pós-processamento do ambiente computacional.

Para filmes 2D, o *software* VLC será instanciado sobre o simulador e exibirá a animação após as imagens serem geradas. Os filmes criados (e salvos) podem ser revistos através do botão *"Filme 2D"*. Neste caso, o usuário escolherá o filme na lista de diretórios. Para análise de volumes, o usuário pode visualizar os cortes ortogonais pressionando o botão *"Visualizar Cortes Ortogonais"*, ou escolher um plano de corte desejado através do botão *"Escolher Plano de Corte"*. No último caso, depois de definir interativamente o plano de corte sobre o volume, o usuário pode gerar o plano de corte pressionando o botão *"Efetuar Corte".* A Figura 4.17 apresenta os botões de operação do Pós-processamento.

Figura 4.17 – Operações de Pós-processamento, respectivamente: *Visualizar Cortes Ortogonais*; *Escolher Plano de Corte*, *Efetuar Corte* e *Filme 2D*.

O usuário pode gerenciar os projetos de simulação através das opções de botão *"Salvar"* e "*Deletar*", respectivamente ilustrados na Figura 4.18 (c) e 4.18 (d). A primeira opção guarda o projeto e o *status* da simulação; a última remove o projeto do diretório em que foi criado e limpa toda a janela principal. Neste caso, o usuário pode criar um novo projeto pressionando o botão *"Novo Projeto"* (Figura 4.18 (a)) – a janela de novo projeto aparecerá sobre a janela principal – ou abrir um projeto existente pressionando *"Abrir Projeto"* (Figura 4.18 (b)), que abrirá a janela com a lista de diretórios para escolha de um projeto já existente.

(a) (b) (c) (d)

Figura 4.18 – Botões (a) *"Novo Projeto"*, (b) *"Abrir Projeto"*, (c) *"Salvar Projeto"* e (d) *"Deletar Projeto"*.

Os componentes gráficos específicos para criação da GUI são encapsulados em bibliotecas que são invocadas apenas pelo componente que comporta a camada de apresentação (GUI), assim como as funções programadas para a interface gráfica. Essas funções fazem processamentos básicos, que envolvem a aquisição de dados inseridos e

exibição de dados solicitados pelo usuário. A GUI pode trabalhar a validação dos dados para evitar inconsistências no momento de processá-los em outros módulos. No entanto, o trabalho sobre os dados para geração de informação é realizado nos outros componentes do ambiente computacional.

4.5 COMPONENTE DE VISUALIZAÇÃO E COMPUTAÇÃO GRÁFICA

O componente de visualização e computação gráfica, responsável em modular todas funcionalidades utilizadas do VTK neste ambiente computacional, é composto de pacotes que organizam as operações de acordo com suas características funcionais. O projeto atual possui os seguintes pacotes: entrada de dados de fornecedores de geometrias e criação de objetos de dados, processamento do *pipeline* de visualização e modelo gráfico, leitor de arquivos gráficos e escritor de arquivos de documentos XML [65], nativos do VTK para armazenamento de geometrias de Pré-processamento.

4.5.1 Pacote *sembr.vtk.data.input*

O pacote que cria objetos de dados e interpreta geometrias importadas recebe o nome *sembr.vtk.data.input*. Com exceção das geometrias de arquivos STL, OBJ e VTK, todos objetos de dados para geometrias são gerados neste pacote, como a esfera, o cubo, os cristais fotônicos e os objetos gráficos de arquivos DXF.

Assim como o VTK utiliza a palavra "*source*" (fonte) para compor parte dos nomes de classes que representam geometrias pré-programadas, o ambiente computacional também utiliza a assinatura "*source*" para designar a classe que utiliza os objetos geométricos previamente programados. Para isso, implementou-se uma classe base abstrata chamada *SourceBase*, com o propósito de distinguir os objetos formados por geometrias programadas e os objetos formados diretamente da leitura de um arquivo gráfico. Por ser abstrata, a classe *SourceBase* não pode ser instanciada como um objeto. Sua função é

fornecer um conjunto de funcionalidades padrões que serão utilizadas pelas classes filhas.

Como todas as geometrias criadas são parte da simulação, elas devem ter suporte para serem salvas ou armazenadas no projeto de simulação corrente. Para isso, este componente provê a interface *ISavable* – presente no pacote de persistência *sembr.vtk.persistence* – que fornece um modelo de persistência ou armazenamento àqueles objetos que precisam gravar dados. O diagrama de Classes [56] da Figura 4.19 mostra a hierarquia de classes para o pacote *sembr.vtk.data.input*. Por conveniência de ilustração, os métodos das sub-classes, disponíveis em grande quantidade, foram omitidos.

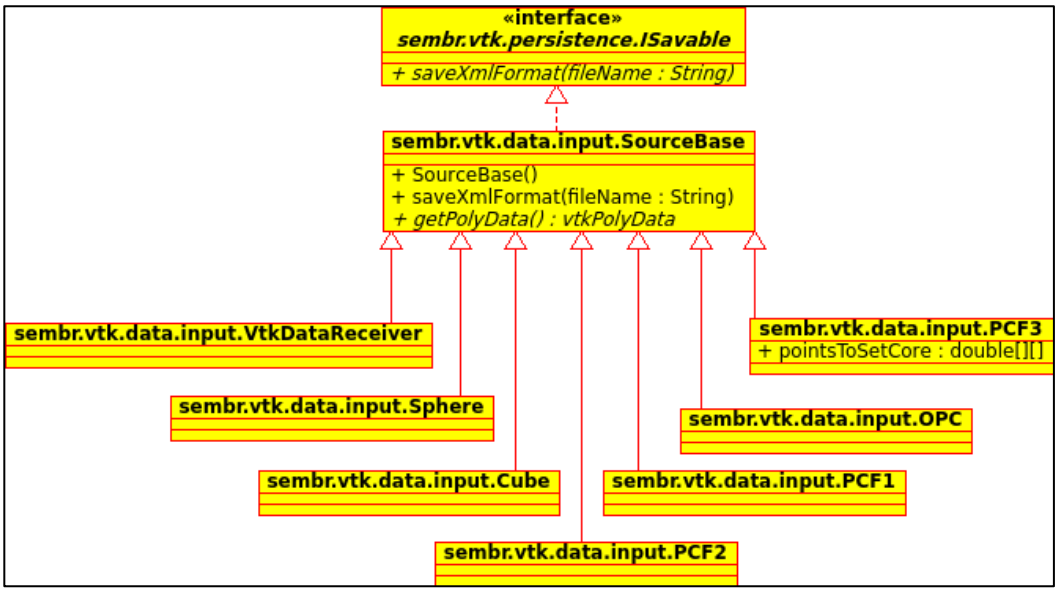


Figura 4.19 – Relações de dependência entre as classes do pacote *sembr.vtk.data.input.*

As classes filhas *Cube*, *Sphere*, *PCF1*, *PCF2*, *PCF3*, *OPC* e *VTKDataReceiver* representam, respectivamente, as seguintes opções de geometrias: cubo, esfera, fibras de cristal fotônico *(PCF1*, *PCF2*, *PCF3)*, cristal fotônico otimizado *(OPC)* e DXF. Elas herdam e implementam os métodos *saveXmlFormat* – pertencente à interface *ISavable* e implementado na super-classe *SourceBase* – para armazenamento de documentos XML e *getPolyData* – pertencente à classe *SourceBase* – que deve retornar a forma poligonal (*Dataset vtkPolyData)* da geometria. Esse desenho hierárquico atribui dois aspectos importantes às geometrias que são utilizadas para simulação: elas podem ser salvas no

projeto criado pelo usuário e sempre assumem a forma básica poligonal utilizada pelo VTK na renderização.

A classe *Cube* possui a propriedade *vtkCubeSource* como objeto de representação de um cubo. Da mesma forma, a classe *Sphere* possui a propriedade *vtkSphereSource* para representar uma esfera. As classes *vtkCubeSource* e *vtkSphereSource* representam objetos-fonte no *framework* do VTK por serem programaticamente disponíveis.

As geometrias das fibras de cristal fotônico e do dispositivo de cristal fotônico otimizado são formadas pela combinação de geometrias simples, realizadas a partir de operações booleanas ou de junção de geometrias (tópico **2.7.1**). Ambas abordagens foram testadas para verificação da complexidade de programação e da qualidade de resultados.

As geometrias das classes *PCF1* e *PCF3* são formadas pela aplicação de operações booleanas sobre cilindros criados pela função implícita do cilindro *(vtkCylinder)*. Os mecanismos booleanos do VTK – diferença, união e intersecção – operam apenas as geometrias geradas pelas funções de forma $w = F(x, y, z)$. Todos cilindros são anexados pela classe *vtkImplicitBoolean*, que subtrai os cilindros de menor raio (representação dos furos da fibra de cristal fotônico) de um cilindro externo (fibra) através do método *SetOperationTypeToDifference*. A diferença entre o cilindro maior e o conjunto de cilindros internos menores gera uma representação de fibra que apresenta furos cilíndricos internos, configurados em posições que formam um padrão hexagonal.

Todos os cilindros gerados por funções implícitas têm altura infinita. Para representá-los com altura definida, é necessário cortá-los. A classe *vtkPlane* pode gerar uma representação planar para a realização de cortes. Um objeto do tipo *vtkPlane* é gerado pela função implícita do plano e, por isto, pode ser adicionado na instância da classe *vtkImplicitBoolean*. Deve-se criar dois planos que cortam transversalmente as extremidades dos cilindros.

As operações booleanas aplicadas sobre os cilindros pela classe *vtkImplicitBoolean* não geram um *Dataset*. Para gerá-lo, deve-se utilizar a classe *vtkSampleFunction*, que receberá o modelo da função gerada na instância do *vtkImplicitBoolean* e criará um *Dataset* do tipo *vtkImageData*. Como a classe *vtkImageData* representa uma geometria em forma de

grade uniforme retangular, ela deve ser passada por um mecanismo de contorno para obtenção da superfície (forma) geométrica. Para isso, a saída de um objeto do tipo *vtkSampleFunction* deve ser conectada à entrada de um objeto do tipo *vtkContourFilter*, um filtro que aplica o contorno sobre a região geométrica e gera, posteriormente, um *Dataset* poligonal como saída (*vtkPolyData*). A Figura 4.20 apresenta as geometrias geradas pelas classes *PCF1* e *PCF3*.

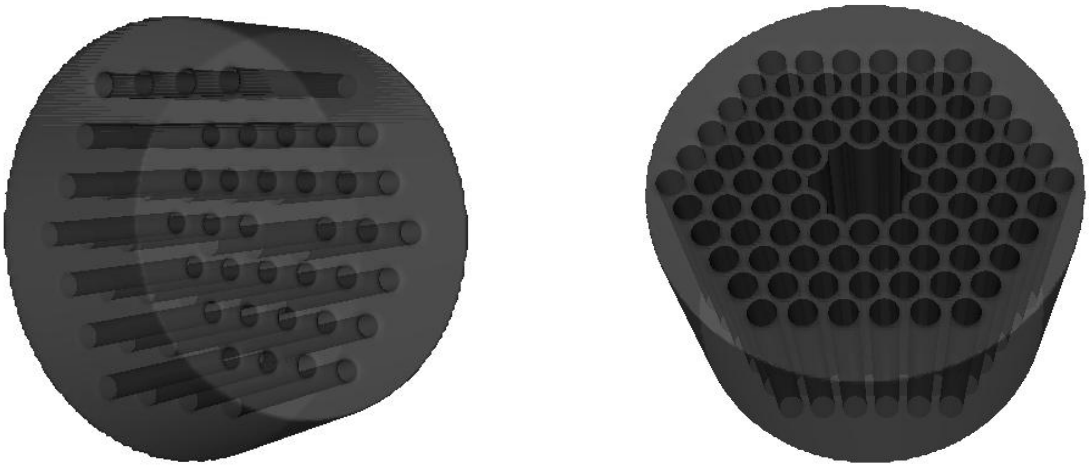


Figura 4.20 – (a) *PCF1* e (b) *PCF3*, obtidas pela aplicação de operações booleanas em geometrias.

As geometrias das classes *PCF2* e *OPC* são formadas por objetos-fonte, que são combinados pela classe *vtkAppendPolyData*. A classe *PCF2* é composta por cilindros criados pela classe *vtkCylinderSource*, que são juntados no objeto *vtkAppendPolyData* para assumir a forma de uma fibra composta de cilindros internos sólidos. A junção efetuada na instância da classe *vtkAppendPolyData* gera uma geometria completa na forma de um *Dataset* poligonal *(vtkPolyData)*. A Figura 4.21 apresenta a geometria obtida em *PCF2*.

A geometria da classe *OPC* é formada por caixas e furos cilíndricos. A classe *vtkCubeSource* é utilizada para formar três caixas de diferentes tamanhos, juntadas pela instância da classe *vtkAppendPolyData*. Os cilindros que formam os furos na maior caixa (central) são instâncias da classe *vtkCylinderSource* e são juntados no objeto *vtkAppendPolyData*, antes de perfurarem a caixa central. Os buracos realizados pelos cilindros são similares àqueles definidos na geometria gerada pela classe *PCF1*, isto é, são

ocos. Entretanto, a classe *OPC* não utiliza as operações booleanas disponíveis na classe *vtkImplicitBoolean*. Por ser formada de objetos-fonte de superfícies poligonais (e não por objetos gerados por funções implícitas), a geometria de *OPC* é perfurada pelo filtro *vtkBooleanOperationPolyDataFilter*, que aplica mecanismos booleanos sobre objetos poligonais. O método da classe *vtkBooleanOperationPolyDataFilter* que subtrai o conjunto de cilindros da caixa é o *SetOperationToDifference*. A Figura 4.22 apresenta o dispositivo fotônico obtido.

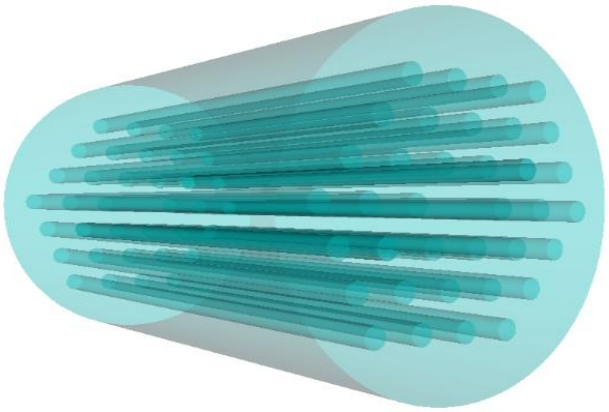


Figura 4.21 – *PCF2* obtida pela junção de geometrias básicas (cilindros).

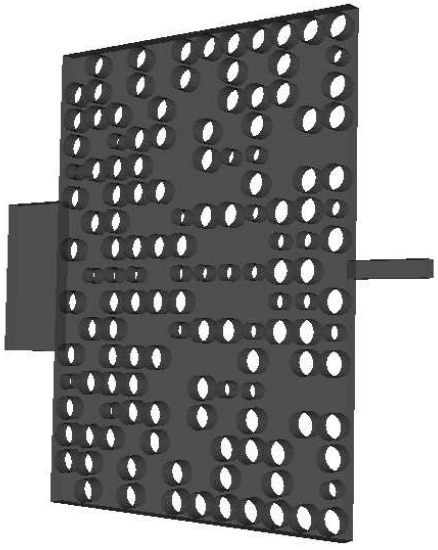


Figura 4.22 – *OPC* obtido pela aplicação de operações booleanas em geometrias.

A classe *vtkDataReceiver* forma geometrias complexas a partir de um conjunto de geometrias primitivas, que são juntadas pela classe *vtkAppendPolyData*, assim como nas classes *PCF2* e *OPC*. As formas geométricas primitivas são adquiridas pela propriedade *IDataSet*, uma interface que provê um padrão para fornecimento de geometrias simples ao VTK. Desenvolvida para criar uma ponte entre o conjunto geométrico amplo do DXF e o VTK, a interface *IDataSet* fornece o conjunto de métodos que devem ser implementados para criar geometrias no VTK. Modeladores de geometrias que disponibilizam arquivos de dados gráficos compostos de formas como faces, círculos, linhas, poli-linhas e poli-faces podem ser importados para o VTK através da realização da interface *IDataSet*.

A classe *vtkDataReceiver* recebe as diferentes formas geométricas primitivas e aplica o algoritmo correto para formar cada geometria no VTK. Na interface *IDataSet*, as geometrias estão disponíveis, separadamente, em coletores de formas, isto é, faces, círculos, linhas, poli-linhas e poli-faces são agrupadas em diferentes coletores. Esta organização permite que a classe *vtkDataReceiver* execute, apenas uma vez, o algoritmo correto de construção das geometrias primitivas presentes em cada coletor de forma. Cada forma geométrica, baseada nas coordenadas dos pontos, possui um arranjo diferente de indexação de pontos para ser construída, conforme descreve o tópico **2.3.1.1**. Depois de gerar os conjuntos de formas geométricas primitivas, a classe *vtkDataReceiver* utiliza a classe *vtkAppendPolyData* para anexá-los e formar a geometria completa em forma de *Dataset* poligonal *(vtkPolyData)*.

4.5.2 Pacote *sembr.vtk.reader*

O pacote de leituras de arquivos gráficos é organizado por classes que utilizam os seguintes leitores de arquivos gráficos e de imagens, nativos do VTK: *vtkSTLReader*, *vtkOBJReader* e *vtkPNGReader*. A representação gráfica é gerada pela interpretação efetuada no respectivo leitor que, após carregar o arquivo correspondente, disponibiliza a forma geométrica.

As classes pertencentes à esse pacote são especializações da super-classe *ReaderBase*. Assim como a classe pai *SourceBase* – presente no pacote *sembr.vtk.data.input* – *ReaderBase* implementa a interface *ISavable*, disponível no pacote *sembr.vtk.persistence*. Dessa forma, as classes que geram geometrias através da leitura de arquivos podem ser persistidas ou armazenadas por meio da implementação do método *saveXmlFormat*. A hierarquia de classes presentes nesse pacote é ilustrada na Figura 4.23. Por conveniência de ilustração, os métodos das sub-classes, disponíveis em grande quantidade, foram omitidos.

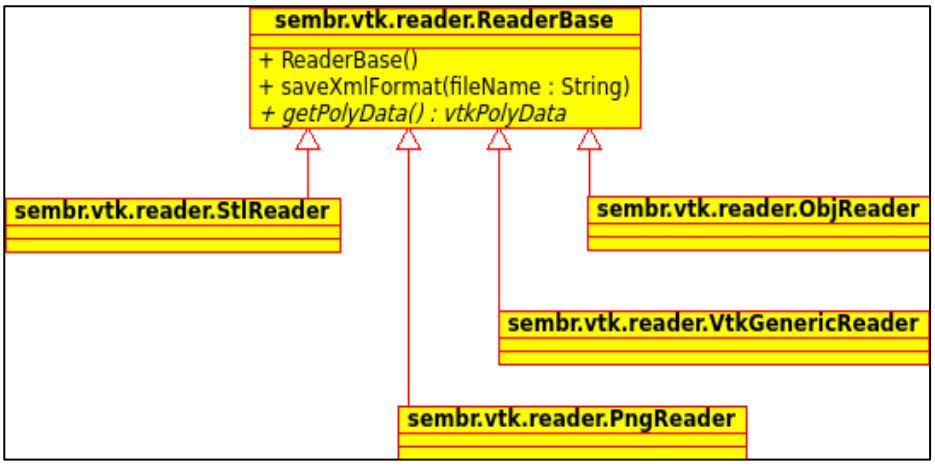


Figura 4.23 – Hierarquia geral de classes no pacote *sembr.vtk.reader.*

As sub-classes *STLReader* e *OBJReader* geram representações geométricas de arquivos gráficos STL e OBJ, a partir das propriedades *vtkSTLReader* e *vtkOBJReader*, respectivamente. As saídas destas propriedades retornam a forma poligonal *(Dataset vtkPolyData)* das geometrias presentes nos respectivos arquivos e foram utilizadas, portanto, para implementar o método abstrato *getPolyData*, herdado da super-classe *ReaderBase* pelas sub-classes *STLReader* e *OBJReader*.

A sub-classe *VtkGenericReader* gera representações geométricas de arquivos de extensão *.vtp*, caracterizados em armazenar dados poligonais em formato XML. A propriedade do tipo *vtkXMLGenericDataObjectReader*, pertencente à sub-classe *VtkGenericReader*, realiza a leitura do arquivo *.vtp* e gera um objeto de dados *(*classe

vtkDataObject) como saída. Como o *Dataset* poligonal *vtkPolyData* é sub-classe de *vtkDataObject*, a saída da instância da classe *vtkXMLGenericDataObjectReader* pode ser convertida em *vtkPolyData* e utilizada, portanto, na implementação do método abstrato *getPolyData,* presente na classe *VtkGenericReader* pela herança estabelecida com *ReaderBase*.

A sub-classe *PNGReader* é executada apenas no estágio de Pós-processamento, assumindo características diferentes dos leitores anteriormente mencionados. Sua operação principal é gerar volumes (a partir do empilhamento de imagens de perfil de campo eletromagnético) para realização de cortes. Para isso, a sub-classe possui as seguintes propriedades: *vtkPNGReader* (empilhamento de imagens PNG), *vtkImplicitPlaneWidget2* (plano definido arbitrariamente) e *vtkImagePlaneWidget* (planos de visualização ortogonais), conforme descritos nos respectivos tópicos **3.4.1**, **3.4.4** e **3.4.5**.

A sub-classe *PNGReader* recebe os nomes de arquivos de imagens PNG através da propriedade *vtkStringArray*, que é passada para a instância da classe *vtkPNGReader*, responsável em carregar as imagens em uma pilha. Como a saída de *vtkPNGReader* é um *Dataset* do tipo *vtkImageData*, o método *getPolyData* herdado deve ser implementado de maneira diferente, pois seu tipo de retorno é o *Dataset vtkPolyData*. Como a sub-classe *PNGReader* utiliza o filtro *vtkOutlineFilter* para criar uma caixa, de faces transparentes, como representação do domínio computacional da simulação, utilizou-se a saída de *vtkOutlineFilter*, que é um *Dataset* do tipo *vtkPolyData*, para implementar o método *getPolyData*.

A sub-classe *PNGReader* possui, como propriedade, a classe *vtkScalarBarActor* para representar a barra de escalares. Para configurá-la, a classe *PNGReader* recebe, como parâmetros, o componente do campo eletromagnético desejado (Ex, Ey, Ez, Hx, Hy, Hz) e os valores mínimo e máximo do campo eletromagnético. A rampa de cores é fornecida pela tabela de *lookup (vtkLookupTable)*, configurada no método *getLookupTable*, implementado na classe *PNGReader*.

4.5.3 Pacote *sembr.vtk.writer*

O pacote para escritas de arquivos organiza as funcionalidades que efetuam a persistência de arquivos gráficos do VTK. As super-classes dos pacotes que geram geometrias – *sembr.vtk.data.input* e *sembr.vtk.reader* – importam o pacote *sembr.vtk.writer* para realizar o armazenamento de suas geometrias em arquivos gráficos baseados no formato XML.

A classe *VtkXmlDataSetWriter* é a responsável em escrever os dados gráficos em arquivos. Para salvar um arquivo, a classe *VtkXmlDataSetWriter* recorre à propriedade *vtkXMLDataSetWriter*, uma classe genérica, nativa do VTK, que pode armazenar os diferentes *Datasets* do VTK em documentos XML, como o *vtkPolyData*, o *vtkImageData*, o *vtkUnstructuredGrid*, entre outros. A classe *VtkXmlDataSetWriter* atual suporta o armazenamento de *Datasets* do tipo *vtkPolyData* e *vtkImageData*, que são os tipos de *Datasets* utilizados no ambiente computacional. O diagrama da Figura 4.24 ilustra o diagrama da classe *VtkXmlDataSetWriter* e a agregação realizada com a classe *vtkXMLDataSetWriter*.

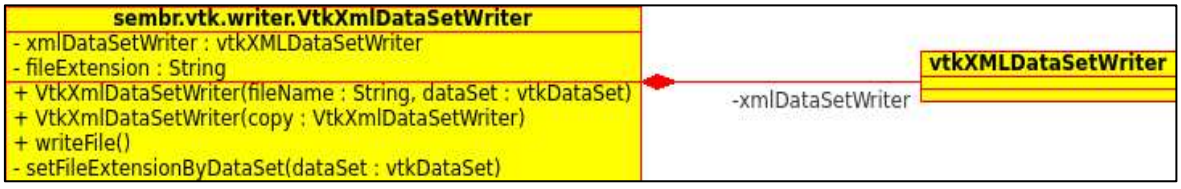


Figura 4.24 – Diagrama da classe *VtkXmlDataSetWriter* e a agregação realizada com a classe v*tkXMLDataSetWriter*.

Para decidir qual tipo de arquivo armazenar, a classe *VtkXmlDataSetWriter* analisa a tipagem do *Dataset* recebido como parâmetro e configura sua propriedade de extensão de arquivo, chamada *fileExtension*, do tipo *String* (caracteres), através do método *setFileExtensionByDataSet* (Figura 4.24). A propriedade é concatenada com o nome do arquivo, recebido como parâmetro, para gerar o nome completo do arquivo que será armazenado. Para tornar possível a visualização dos dados armazenados nos arquivos de

tipo *.vtp* ou *.vti*, a classe *vtkXMLDataSetWriter* deve ser configurada para o formato de armazenamento ASCII (codificação em caracteres), através do método *SetDataModeToAscii*.

4.5.4 Pacote *sembr.vtk.visualizing*

As geometrias geradas na forma de *Dataset* devem ser conectadas no fluxo de trabalho *(workflow)* do VTK, de modo que o *pipeline* de visualização e o modelo gráfico sejam executados para geração dos objetos gráficos. O pacote de visualização provê os meios de mapear *Datasets* no modelo gráfico, criando os atores, renderizadores e a janela de renderização da cena gráfica, através das classes *VtkVisualizationManager* e *VtkMultRenderPanel*. O diagrama de classes de ambas, assim como a agregação entre elas, é apresentada na Figura 4.25.

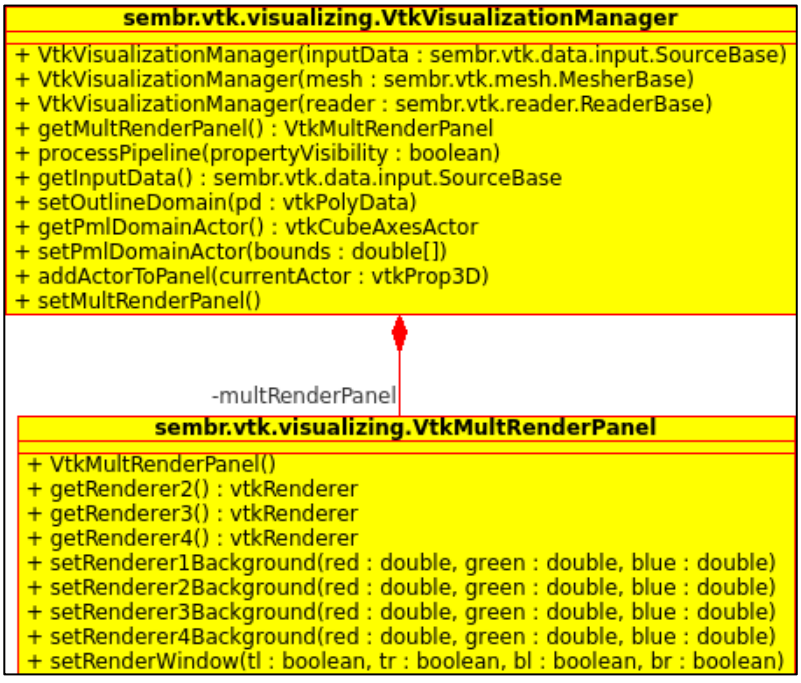


Figura 4.25 – Diagramas das classes *VtkVisualizationManager* e *VtkMultRenderPanel* e a agregação realizada entre elas.

A classe *vtkVisualizationManager* pode receber, como parâmetro, as classes bases *SourceBase* e *ReaderBase*, que fornecem as geometrias em forma de *Dataset vtkPolyData*, através do método *getPolyData.* Um objeto de tipo *vtkPolyDataMapper* mapeia os *Datasets* recebidos (finalizando o *pipeline* de visualização) e conecta-se ao modelo gráfico através de um objeto da classe *vtkActor*, que define a geometria em um objeto gráfico visualizável. Além do ator que representa a geometria, a classe *vtkVisualizationManager* possui atores para representar o domínio computacional e o domínio da PML.

O domínio computacional é representado pela propriedade de tipo *vtkOutlineFilter*, que gera uma saída em formato poligonal e é conectado à um objeto de tipo *vtkPolyDataMapper* que, por sua vez, é passado à um ator. O ator que representa o domínio da PML é do tipo *vtkCubeAxesActor*, que não necessita ser formado geometricamente e, por isso, não recebe um mapeador. A instância da classe *vtkCubeAxesActor* precisa apenas dos limites de coordenadas do domínio da PML para ser corretamente dimensionada.

Para gerar a cena, é utilizada a sub-classe *vtkMultRenderPanel*, uma especialização da classe *vtkPanel*, que possui as propriedades *vtkRender* (renderizador) e *vtkRenderWindow* (janela de renderização). Os atores, câmeras e luzes são adicionados aos renderizadores, que posteriormente são adicionados à janela de renderização. A classe *vtkMultRenderPanel* é especializada em fornecer quatro renderizadores, utilizando o renderizador disponível na classe pai e implementando outros três. Essa característica atribui a capacidade de visualização de até quatro renderizadores na janela de renderização, que são posicionados pela configuração do *viewport* através de coordenadas normalizadas (de 0 a 1).

Assim como as câmeras, os atores de objetos gráficos, de domínio computacional e de PML, criados na classe *vtkVisualizationManager*, são adicionados aos renderizadores da classe *vtkMultRenderPanel*, que são posteriormente passados à janela de renderização para criação da cena gráfica. A Figura 4.26 apresenta os resultados obtidos com múltiplos renderizadores.

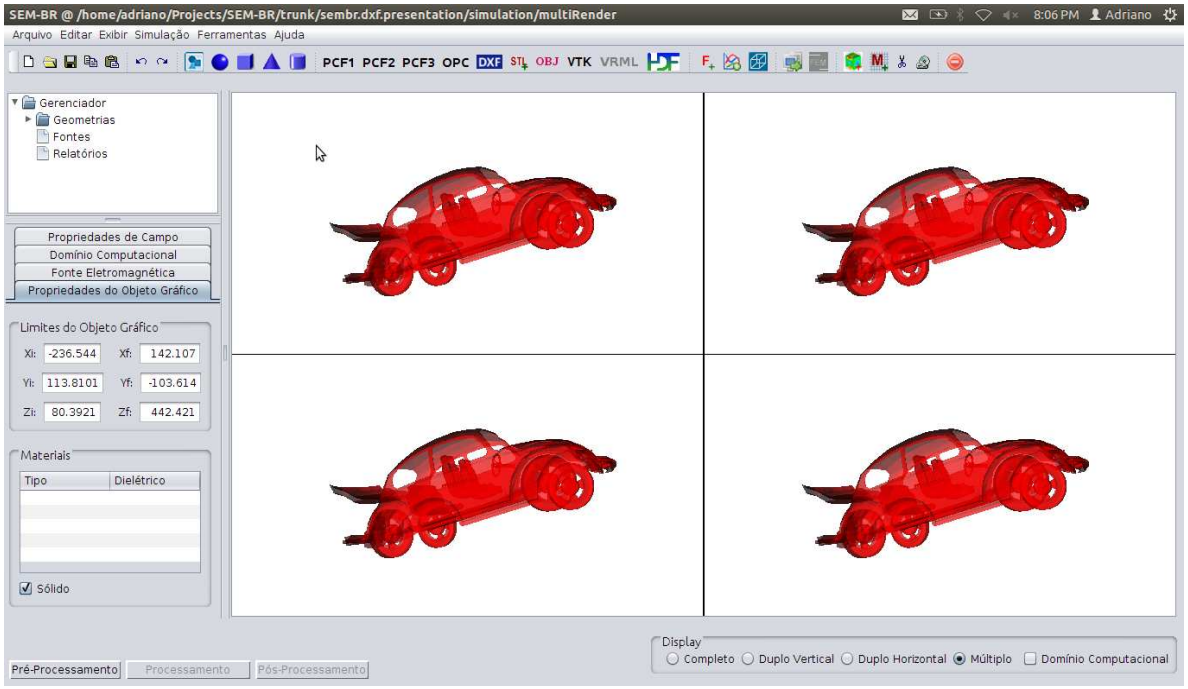


Figura 4.26 – Múltiplos renderizadores disponibilizados pela classe *vtkMultRenderPanel*.

4.6 COMPONENTE DXF-AUTOCAD

O componente de importação de arquivos DXF-Autocad é estruturado, basicamente, em classes que representam os leitores de arquivos ASCII e binário, as seções do arquivo DXF, os grupos de códigos que definem parâmetros e delimitadores de seções e as classes de formas geométricas. A representação do arquivo DXF é feita pela classe *DxfFile*, composta de classes para as seções (*DxfHeader*, *DxfTables*, *DxfBlocks*, *DxfEntities*) e para os grupos (*DxfGroups*).

A classe *DxfFile* é propriedade da classe *DxfReader*, responsável em estabelecer uma interface com os componentes que utilizam o serviço do componente DXF. Ao receber o caminho e o nome de um arquivo DXF, o objeto de tipo *DxfReader* cria uma instância da classe *DxfFile*, que inicia o processo de leitura do arquivo através do seu método *read*. Neste, um objeto de tipo *DxfGroups* é instanciado para obter e validar códigos de grupos, com a finalidade de especificar o objeto de seção (*DxfHeader*, *DxfTables*, *DxfBlocks*,

DxfEntities) que será instanciado dentro do objeto *DxfFile* corrente.

Para obter e validar os códigos de grupos do arquivo DXF, a classe *DxfGroups* disponibiliza o método *read* e uma propriedade do tipo *DedicatedDxfFileInputStream*, que efetua leituras em grupos de duas linhas – código de grupo e respectivo valor – através do seu método *read2lines*. Neste, a classe *DedicatedDxfFileInputStream* referencia sua propriedade *DxfStreamReader*, uma super-classe interna abstrata que generaliza os aspectos comuns entre as classes de arquivos ASCII (*AsciiDxfStreamReader*) e binário (*BinaryDxfStreamReader*). A classe *DxfStreamReader* fornece o método abstrato *readChunk*, que é chamado na execução do método *read2lines*. O método *readChunck* obtem, do arquivo, os dados de código de grupo e respectivo valor. O diagrama de sequência [56], ilustrado na Figura 4.27, ilustra a sucessão de chamadas de métodos para obtenção dos dados de código de grupo e respectivo valor nos referidos objetos.

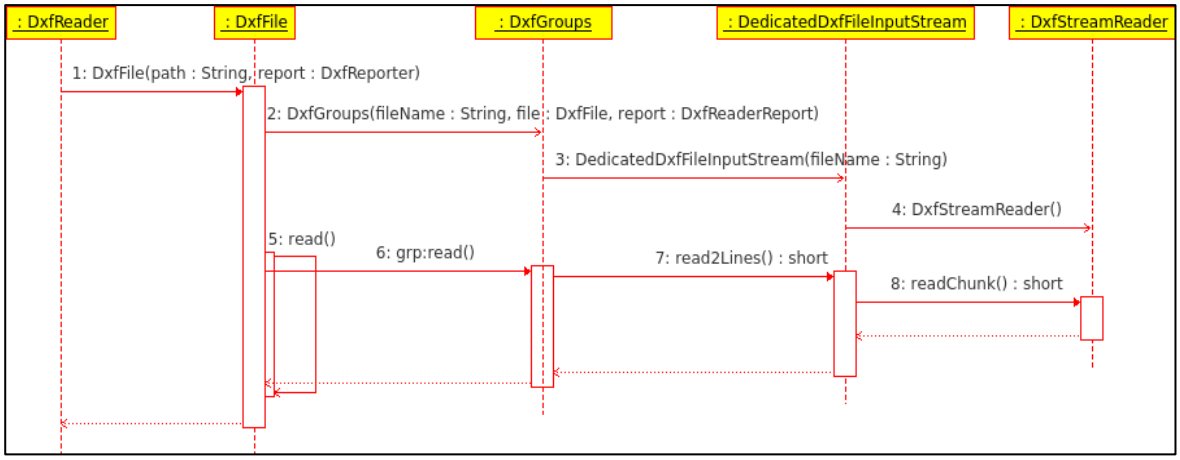


Figura 4.27 – Diagrama geral de sequência das principais operações para carregamento de um arquivo DXF no ambiente computacional.

Com base no código de grupo obtido, a classe *DxfGroups* determina o tipo de seção que a instância da classe *DxfFile* deve validar, em seu método *read*, para criar o objeto de seção adequado. Ao ser instanciado, o objeto de seção recebe o objeto *DxfGroups* corrente (que contem a estrutura de código referente à determinada seção) e realiza o carregamento dos dados referentes à sua seção.

Os objetos de seções que fornecem dados de geometrias e materiais são definidos pelas classes *DxfEntities* e *DxfBlocks*. Ambas encapsulam vetores do tipo *DxfEntity*, uma super-classe que generaliza as formas geométricas e os respectivos materiais.

As formas geométricas suportadas neste projeto são representadas nas seguintes sub-classes: *Dxf3DFACE* (face tridimensional com três ou quatro pontos), *DxfPOLYLINE* (poli-faces ou poli-linhas), *DxfLWPOLYLINE* (poli-linhas bidimensionais), *DxfLINE* (linha) e *DxfCIRCLE* (círculo). A herança estabelecida entre a classe base *DxfEntity* e as formas geométricas implementadas é apresentada no diagrama de classes da Figura 4.28.

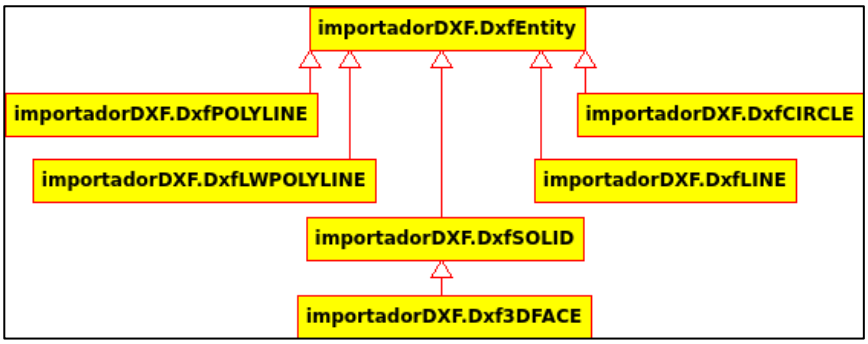


Figura 4.28 – Herança de classes entre a classe base *DxfEntity* e as formas geométricas implementadas.

Antes de ser visualizada, a geometria carregada no objeto *DxfFile* passa pelo componente de geometrias para adequar-se à interface requerida pelo componente VTK, conforme ilustra o diagrama de componentes [56] da Figura 4.29. Neste diagrama, a linha sólida indica que o componente provê uma interface; a seta tracejada indica que o componente implementa uma interface; os nomes *DxfReader* e *IDataSet* denominam interfaces de conexão.

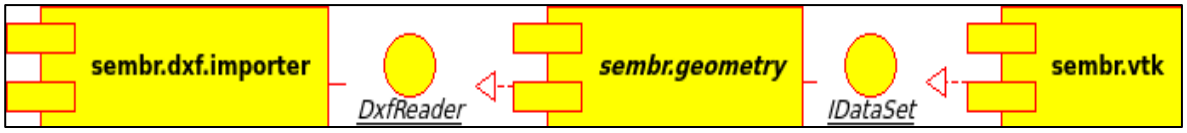


Figura 4.29 – Relação entre os componentes importador DXF, VTK e geometrias.

4.7 COMPONENTE DE GEOMETRIAS

O componente de geometrias (*sembr.geometry*) é responsável em prover uma interface que estabeleça um padrão de fornecimento de geometrias e materiais entre arquivos de dados gráficos e o VTK. O componente separa os provedores de geometrias dos mecanismos de visualização e assegura que os dados geométricos e de materiais sejam adequadamente transferidos ao VTK para a criação correta dos objetos gráficos. Desta forma, o componente de geometrias possibilita que novos leitores ou importadores geométricos sejam acoplados ao ambiente computacional, sem a necessidade de implementação de funcionalidades específicas de visualização. A implementação da interface exposta pelo componente de geometrias garante que os dados geométricos e de materiais sejam apropriadamente processados e visualizados pelo VTK. A interface fornecida pelo componente de geometrias recebe o nome de *IDataSet*.

O componente de geometrias é dividido em três pacotes: *sembr.geometry.interfacing*, *sembr.geometry.shapes* e *sembr.geometry.utils*. Os detalhes de cada pacote são apresentados nos tópicos a seguir.

4.7.1 Pacote *sembr.geometry.interfacing*

Este pacote disponibiliza as interfaces *IDataSet* e *IDataAttribute*, além das classes que as implementam. As formas geométricas e os materiais são providos aos componentes externos pela interface *IDataSet*. As operações para obtenção de materiais são padronizadas pela interface *IDataAttribute*. O diagrama de classes da Figura 4.30 ilustra a relação entre as interfaces e classes desse pacote.

As formas geométricas são representadas pela classe abstrata *Shape*, que contem uma lista de elementos de uma geometria básica. Sua finalidade é armazenar uma lista de um único tipo geométrico, de forma que listas de tipos geométricos diferentes estejam relacionadas à diferentes objetos *Shape*. A interface *IDataAttribute* é implementada pela

classe *Shape*, de modo que seja possível relacionar o elemento geométrico básico com seu respectivo material. Para estabelecer um padrão de operações que são comuns entre todas as formas geométricas, a classe *Shape* utiliza a técnica *Generics* [66], de modo que a lista de elementos geométricos possa ser composta de um tipo qualquer (linha, face, poli-linhas, círculo, etc.) e ser processada de forma correta, independentemente do tipo que a compõe.

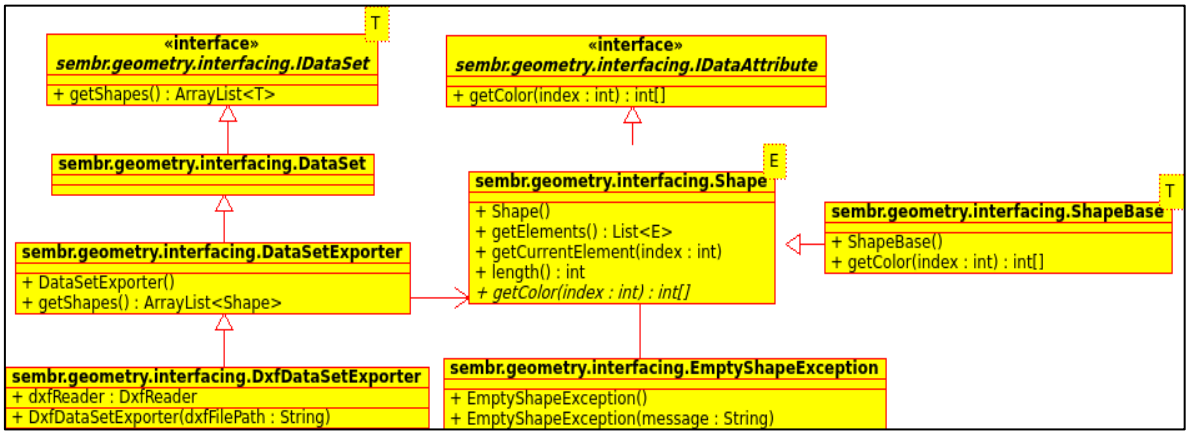


Figura 4.30 – Diagrama de classes para o pacote *sembr.geometry.interfacing*.

A super-classe *ShapeBase* herda as características da classe *Shape* e realiza a interface *IDataAttribute*, através da implementação do método *getColor*. Caracterizada pela técnica *Generics*, a classe *ShapeBase* determina que a lista de elementos de um tipo geométrico seja formada por um objeto de tipo *dxfEntity*, a forma genérica de geometrias importadas de um arquivo DXF. Assim, a classe *ShapeBase* pode formar uma lista de elementos com os seguintes tipos geométricos: face tridimensional com três ou quatro pontos, poli-faces ou poli-linhas tridimensionais, poli-linhas bidimensionais, linha e círculo.

A classe *DataSetExporter* agrega uma lista de objetos do tipo *Shape*, de modo que um objeto gráfico composto de diferentes formas geométricas possa ser armazenado nos diferentes objetos *Shape* contidos na lista. A classe *DataSetExporter* é base para a classe *DxfDataSetExporter*, que tem a finalidade de receber a interface *DxfReader* (componente *sembr.dxf.importer*) e analisar o conteúdo das entidades *(dxfEntity)* do arquivo DXF, para separar as formas geométricas e carregá-las nos diferentes objetos *Shape*.

4.7.2 Pacote *sembr.geometry.shapes*

Este pacote realiza a implementação das diferentes formas geométricas suportadas no ambiente computacional. As classes são vistas como coleções de formas e, por isso, herdam as características da classe abstrata *Shape*. As classes que compõem este pacote, atualmente, são: *FaceCollector*, *CircleCollector*, *LWPolylineCollector*, *LineCollector*, *PolyfaceCollector* e *PolylineCollector*. Cada uma destas classes contêm uma lista do seu tipo específico, disponibilizado na super-classe *Shape*.

Os métodos dos coletores garantem a descrição adequada dos dados geométricos para o componente do VTK, através do mapeamento das geometrias básicas importadas em células. Cada coletor de forma aplica o algoritmo adequado para associar uma lista de pontos com uma lista específica de índices de pontos, de modo a formar a topologia de uma célula do VTK.

Para armazenar todas as formas geométricas possíveis importadas de um arquivo, é disponibilizada a classe *ShapesCollection*, que agrega todos os coletores implementados. A geometria completa carregada de um arquivo é, portanto, acessada pela instância da classe *ShapesCollection*, que fornece métodos de acesso às formas geométricas encontradas. A Figura 4.31 ilustra o diagrama de classes para este pacote.

4.7.3 Pacote *sembr.geometry.utils*

Os dados utilizados para dar suporte à construção das geometrias estão disponíveis no pacote *sembr.geometry.utils*. O pacote disponibiliza a classe *ColorMap*, que contêm o modelo de cores RGB *(Red-Green-Blue)* e é ilustrada na Figura 4.32. A classe *ColorMap* é utilizada, atualmente, para mapear as cores disponíveis nas entidades (*dxfEntity*) carregadas de um arquivo DXF.

Este pacote pode fornecer, futuramente, outros modelos de cores, como o HSV (*Hue-Saturation-Value*) e o mapeamento de materiais com base em valores de cores.

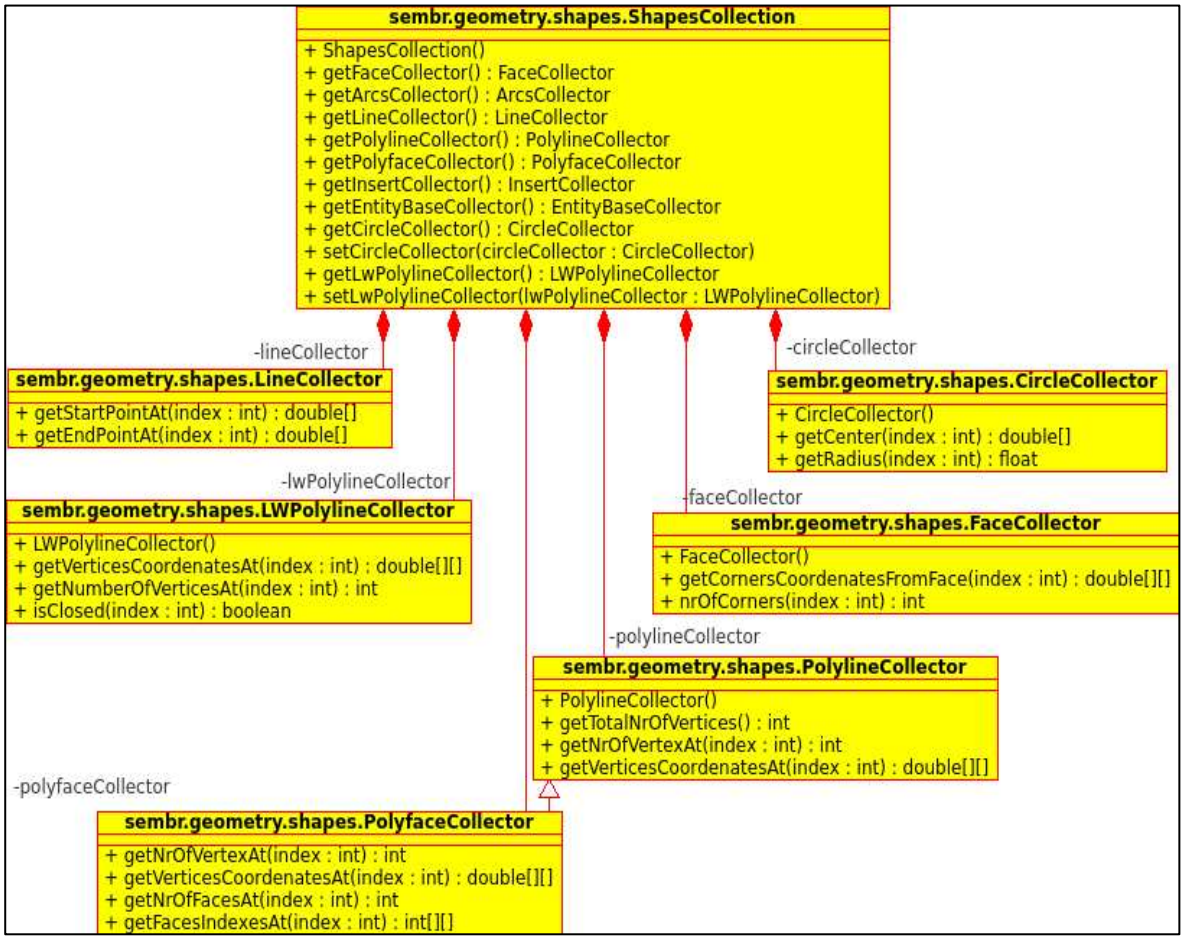


Figura 4.31 – Diagrama de classes para o pacote *sembr.geometry.shapes*.

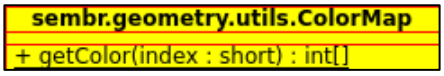


Figura 4.32 – Diagrama de classe para o pacote *sembr.geometry.utils*.

4.8 COMPONENTE DE ENTIDADES

As entidades formam as classes de dados essenciais ao funcionamento do sistema computacional. Nas aplicações multicamadas, as entidades são responsáveis em transportar a informação entre as diversas camadas. Elas estruturam e refletem a relação de dados de modo a estabelecer uma semântica para o sistema.

Geralmente, as entidades programadas são um reflexo direto do modelo de banco de dados de um sistema. As bases de dados relacionais são compostas de tabelas (linhas e colunas), que se associam à outras tabelas conforme regras de relacionamento. As colunas de tabelas representam o tipo de dado que é armazenado; as linhas contêm o conjunto de dados (tuplas) armazenado conforme a estrutura imposta pelas colunas.

A relação dos dados nas tabelas normalmente é refletida no sistema que utiliza o banco de dados, de modo que os dados possam ser recuperados, adquiridos, processados e armazenados corretamente. As entidades são responsáveis em mapear as tabelas, de forma a representar a estrutura de dados do banco de dados no sistema implementado. Assim, as entidades tornam-se classes compostas de propriedades que são geralmente definidas pelas colunas (tipo de dados) das tabelas.

Embora não possua um banco de dados definido, o ambiente computacional deste projeto contêm um componente de entidades, que alimenta todas as outras camadas (componentes) da aplicação. As entidades deste sistema refletem, basicamente, os dados necessários para realização do processamento no MEEP e os dados para gerenciamento das simulações. O diagrama de classes da Figura 4.33 ilustra a estrutura de entidades implementada (exceto a entidade representada pela classe *ProjectSettings*, que é destacada na Figura 4.34).

Os dados adquiridos na GUI são armazenados em entidades e transferidos, através de arquivos, para o componente de processamento para formar a configuração da simulação para o MEEP. Se o usuário optar por salvar o projeto, as entidades serão persistidas em arquivos e podem ser recuperadas pela aplicação.

A entidade responsável por conter os dados referentes ao projeto de simulação (nome do projeto, diretório, *status* do projeto, etc.) é denotada pela classe *ProjectSettings* (Figura 4.34).

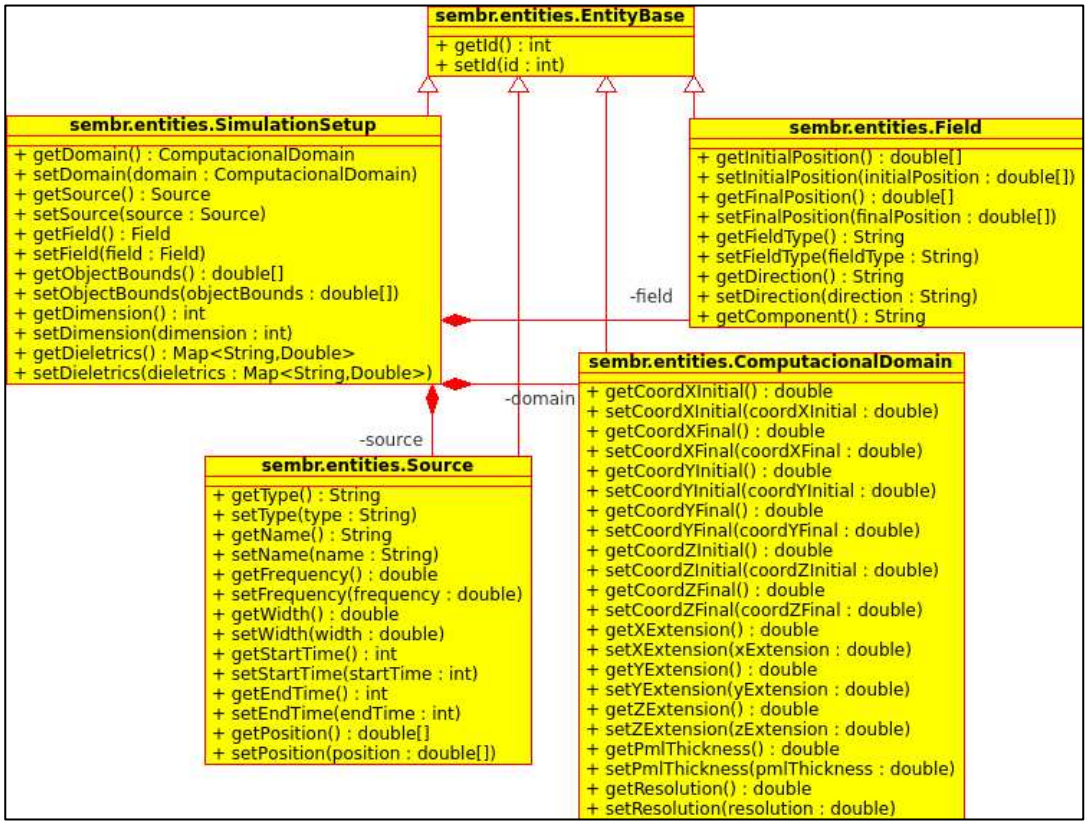


Figura 4.33 – Diagrama da herança de classes para a estrutura de entidades do ambiente computacional.

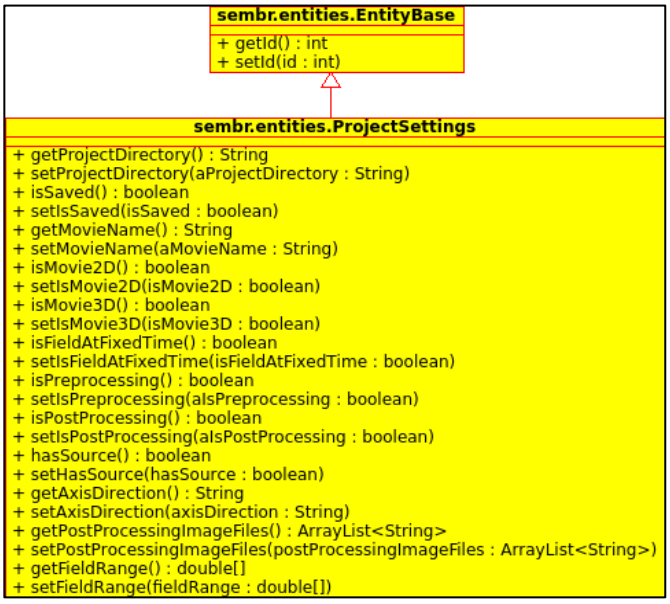


Figura 4.34 – Diagrama da classe (entidade) *ProjectSettings*.

4.9 COMPONENTE DE PERSISTÊNCIA DE DADOS

O componente *sembr.dao* é responsável em prover os mecanismos de persistência de dados, encapsulando e separando os mecanismos de armazenamento e recuperação da informação dos outros componentes do sistema. O acrônimo "*dao*" tem o significado de "*data access objects*" (objetos de acesso à dados).

O ambiente computacional armazena e recupera os dados em arquivos XML, que são mantidos nos diretórios do projeto da simulação corrente. A escolha de utilização de arquivos XML é motivada pelo uso padronizado deste formato, como nos serviços *Web* (*Web Services*) e no armazenamento de geometrias pelo VTK.

O componente é formado por dois pacotes: *sembr.dao.reader* e *sembr.dao.writer.* O pacote de escritas (*sembr.dao.writer*) é formado pela classe *XmlFileWriter*, que recebe as entidades do sistema e os grava, em forma de arquivo XML, no diretório do projeto de simulação recebido pelo parâmetro *path* (diretório de armazenamento). Por outro lado, o pacote de leituras (*sembr.dao.reader*), formado pela classe *XmlFileReader*, efetua leituras de arquivos XML e popula as entidades com os dados carregados destes arquivos. O diagrama de classes da Figura 4.35 mostra as classes de persistência atuais.

Figura 4.35 – Diagrama de classes para o componente *sembr.dao*.

O ambiente computacional guarda três tipos de arquivos XML diferentes: *projets_settings.xml* (configurações de projeto providas pela entidade *ProjectSettings*), *setup.xml* (configurações de processamento providas pelas entidades disponíveis na classe *SimulationSetup*) e *sembrPolyData.vtp* (*Dataset* do VTK). O arquivo de configurações de projeto é armazenado no diretório raiz do projeto de simulação. Os arquivos do *Dataset* e das configurações de processamento são colocados no diretório */pre_processing*.

4.10 COMPONENTE DO MEEP

A execução do MEEP é realizada no componente *sembr.meep*, que é basicamente estruturado por classes que dão suporte à configuração do processamento no MEEP. Cada nova versão do componente *sembr.meep* deve ser compilado para gerar um arquivo executável (*sembr.meep*). A função *main*, disponível no arquivo *main.cpp*, é invocada sempre que o executável é instanciado.

Para flexibilizar a configuração, as entidades do ambiente computacional – programadas em Java – foram mapeadas em C++ para abastecer o MEEP com os dados fornecidos pelo usuário. As entidades são carregadas na classe *XmlFileReader*, que efetua a leitura do arquivo *setup.xml* e carrega as propriedades das entidades com os dados lidos do arquivo.

A integração entre o ambiente computacional e o MEEP é, portanto, realizada pela transferência de arquivos e instanciação do executável gerado, que carrega os arquivos XML armazenados pelo ambiente computacional e alimenta o MEEP com os dados carregados nas entidades.

A ligação entre as entidades do ambiente computacional e a interface do MEEP é realizada na classe *MeepExecutor*, que agrega as classes disponíveis na interface do MEEP para configuração da simulação. A Figura 4.36 ilustra o diagrama da classe *MeepExecutor*.

A classe *MeepExecutor* cria os objetos do MEEP (*grid_volume*, *structure*, *src_time*, *component*, *fields*, *h5_file* [26]) com base nas configurações da propriedade *ProjectSettings*, populada pelo leitor *XmlFileReader*. O parâmetro em forma de ponteiro de função recebido pela classe *MeepExecutor* corresponde à função de mapeamento de geometrias e materiais, que podem ser de dois tipos: bidimensionais (função *epsilon2D*) e tridimensionais (função *epsilon3D*).

As funções de mapeamento *epsilon2D* e *epsilon3D*, contidas no arquivo *main.cpp*, retornam o material específico de uma determinada posição (bidimensional ou tridimensional), determinada pelo mecanismo de varredura do MEEP. Elas recebem um parâmetro do tipo *meep::vec*, que contem os dados de coordenadas da posição de varredura

corrente.

Figura 4.36 – Diagrama da classe *MeepExecutor*.

Dentro das funções de mapeamento, um *Dataset* (geometria) do VTK tem seus dados mapeados pela técnica de cruzamento de raios (tópico **2.8**). Um objeto de tipo *vtkOBBTree* recebe o *Dataset* e calcula as intersecções dos raios que cruzam o domínio computacional. Para os raios que cruzaram o *Dataset* e definiram a região do objeto gráfico, o objeto *vtkOBBTree* obtêm os índices das células da geometria interseccionadas pelos raios. Com base nos índices, é possível obter o material que compõe uma célula do *Dataset*, através dos dados de atributos.

Com o mapeamento geométrico estabelecido pelas funções *epsilon2D* e *epsilon3D*, o MEEP é capaz de gerar a representação discretizada da geometria e utilizá-la para o processamento do FDTD.

4.11 COMPONENTE DE PROCESSAMENTO

O componente *sembr.processing* é composto pela classe *SembrProcessing*, responsável em invocar o executável *sembr.meep* e passar o diretório dos arquivos XML do projeto de simulação. O diagrama da classe *SembrProcessing* é ilustrada na Figura 4.37.

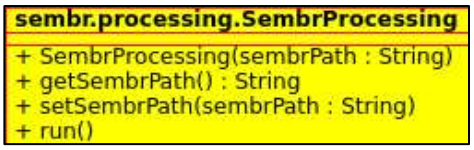


Figura 4.37 – Diagrama da classe *SembrProcessing*.

Este componente é chamado pela GUI em uma *thread* a parte, conforme descrito no tópico **4.4**, sendo processado paralelamente à instância do ambiente computacional. Uma ilustração alternativa é apresentada na Figura 4.38, que exibe a comunicação entre o ambiente computacional (em Java) e o MEEP (em C++), a troca de arquivos XML e a instanciação da *thread* que executa o Processamento pela chamada do MEEP.

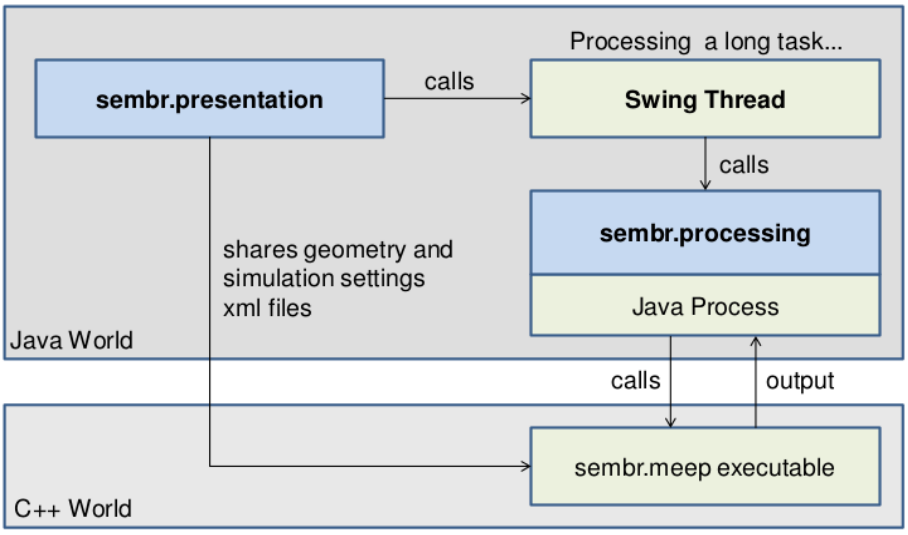


Figura 4.38 – Interação entre o ambiente computacional e o MEEP.

4.12 COMPONENTE DE PÓS-PROCESSAMENTO

O componente *sembr.postprocessing* encapsula as funcionalidades para realização do Pós-processamento. As operações do componente estão disponíveis na classe *SembrPostprocessing* (ilustrada na Figura 4.39), que realiza, através de comandos (*scripts*), as chamadas dos seguintes *softwares*: H5ls, H5topng e Mencoder.

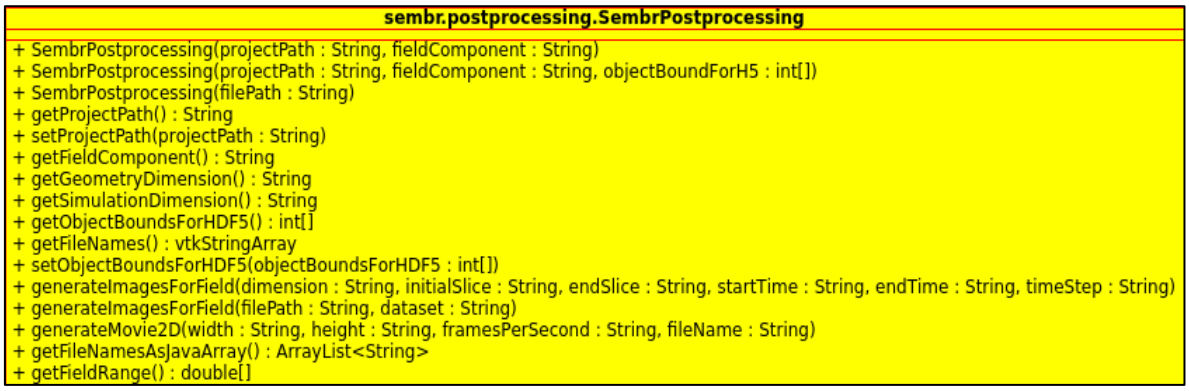


Figura 4.39 – Diagrama da classe *SembrPostprocessing*.

As informações providas pela janela de Pós-processamento na GUI configuram os *scripts* de execução dos *softwares* suportes. Na execução do H5topng, as informações de saída são armazenadas na propriedade *processingOutput*, que guarda os dados de campos eletromagnéticos para serem posteriormente utilizados na criação de valores relacionados à barra de cores.

O componente de Pós-processamento é executado em uma *thread* a parte, pois a geração das imagens para criação dos volumes pode ser um processo lento. O ambiente computacional fica livre para uso neste caso.

CAPÍTULO 5

VALIDAÇÕES E RESULTADOS

Este capítulo apresenta os resultados obtidos com o projeto do ambiente computacional. Para testar e validar os requisitos estabelecidos, são ilustradas as funcionalidades de Pré-processamento e Pós-processamento implementadas, assim como os mecanismos de usabilidade para geração de simulações. Com a finalidade de testar a consistência da implementação através de simulações, os tópicos iniciais 5.1, 5.2 e 5.3 visam comparar simulações igualmente configuradas no ambiente computacional e no MEEP-C++. Os tópicos subsequentes apresentam projetos de simulações em guias de onda mais complexos, modelados no ambiente computacional, que não foram implementados nas versões originais do MEEP devido à dificuldade ou inviabilidade de geração dos mesmos nestas versões.

5.1 INTRODUÇÃO

As simulações geradas neste trabalho visam, sobretudo, validar a capacidade do ambiente computacional em criar soluções semelhantes àquelas geradas pela execução no MEEP nas versões originais – sobre a mesma configuração e conjunto de parâmetros – e demonstrar que o sistema implementado atende os requisitos de criação de simulações mais complexas, de forma fácil e interativa.

O ambiente computacional é comparado à versão em C++ do MEEP, visto que os objetos geométricos simples são facilmente implementados nesta versão e os complexos são trabalhosos para implementar em qualquer versionamento do MEEP.

É importante notar que valores numéricos de parâmetros correspondentes a algum tipo de tamanho no MEEP são adimensionais, isto é, as unidades de tamanho são

invariantes em escala [46], isentando o usuário da necessidade de definir unidades de tamanho (nanômetro, micrometro, milímetro, centímetro, etc) em variáveis de dimensões e comprimento.

5.2 GUIA DE ONDA RETANGULAR: TESTE COMPARATIVO ENTRE O MEEP-C++ E O AMBIENTE COMPUTACIONAL

Para iniciar os testes que verificam a consistência da implementação elaborada no ambiente computacional, este tópico apresenta uma simulação bidimensional básica encontrada no tutorial do MEEP [27], com o propósito de comparar o resultado de ambas simulações, baseadas nos mesmos parâmetros, e averiguar se o ambiente computacional é capaz de mapear o guia de onda e executar a simulação assim como é realizado no MEEP original.

Embora a versão do código de simulação presente no tutorial esteja originalmente disponível na linguagem *Scheme*, foi implementada uma versão idêntica no MEEP C++. O cenário da simulação, presente nesse tutorial, foi configurado segundo a relação de dados da Tabela 5.1.

Tabela 5.1 – Parâmetros de configuração e valores utilizados na simulação do guia de onda retangular.

Domínio computacional	16 x 8 (eixos *x* e *y*, respectivamente)
Largura da PML	1
Modelo de guia de onda	retangular
Dimensões do guia de onda	infinito x 1 (eixos *x* e *y*, respectivamente)
Centro (C) do guia de onda	C(0, 0) (eixos *x* e *y*, respectivamente)
Constante dielétrica do guia de onda	12 (silício)
Constante dielétrica externa ao guia de onda	1 (ar)
Fonte	Contínua

Posição (P) da fonte	P(-7, 0) (eixos *x* e *y*, respectivamente)
Frequência da fonte	0.15
Comprimento de onda	6.67
Propriedades de campo	Elétrico, na direção do eixo *z* (componente *Ez*)
Resolução do cenário de simulação	10
Tempo de execução	200 passos de tempo

Após executar o código, o MEEP gerou os arquivos HDF5 que representam o guia de onda retangular e os cálculos do campo elétrico. Ambos arquivos foram convertidos em imagens pelo H5utils. A imagem do guia de onda é mostrado na Figura 5.1.

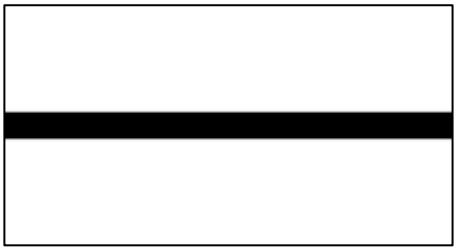
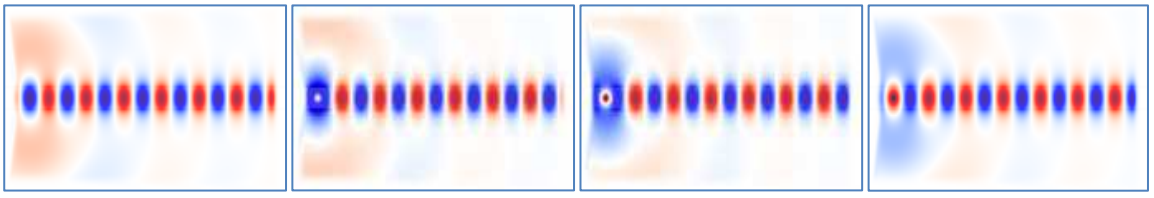


Figura 5.1 – Guia de onda retangular gerado pelo MEEP C++.

A Figura 5.2 ilustra imagens dos perfis de campo elétrico (direção no eixo *z*), produzidas pelo MEEP-C++ para as fatias de tempo 3000, 3020, 3040 e 3060, respectivamente.



(a) (b) (c) (d)

Figura 5.2 – Perfis de campo elétrico para o guia de onda retangular nas fatias de tempo (a) 3000, (b) 3020, (c) 3040 e (d) 3060.

Para testar o ambiente computacional, configurou-se uma simulação idêntica a apresentada anteriormente, com o propósito de validar, principalmente, a capacidade de mapeamento da geometria – através do método de traçamento de raios – e a configuração correta da simulação, através da integração realizada entre o ambiente computacional e o MEEP.

Antes de iniciar a configuração da simulação, o usuário deve criar um projeto de simulação no ambiente computacional. O projeto organizará e armazenará todos os artefatos de simulação (arquivos de geometria, de configuração e de Pós-processamento) em um determinado diretório. Para isso, o usuário deve definir um nome de projeto e um caminho de diretório, conforme mostrado no tópico **4.4** e na Figura 4.3.

Um guia de onda retangular, projetado no AUTOCAD® e armazenado em formato DXF, foi importado para o ambiente computacional. Após ler o arquivo, o sistema apresentou automaticamente a visualização do objeto gráfico importado, conforme ilustra a Figura 5.3.

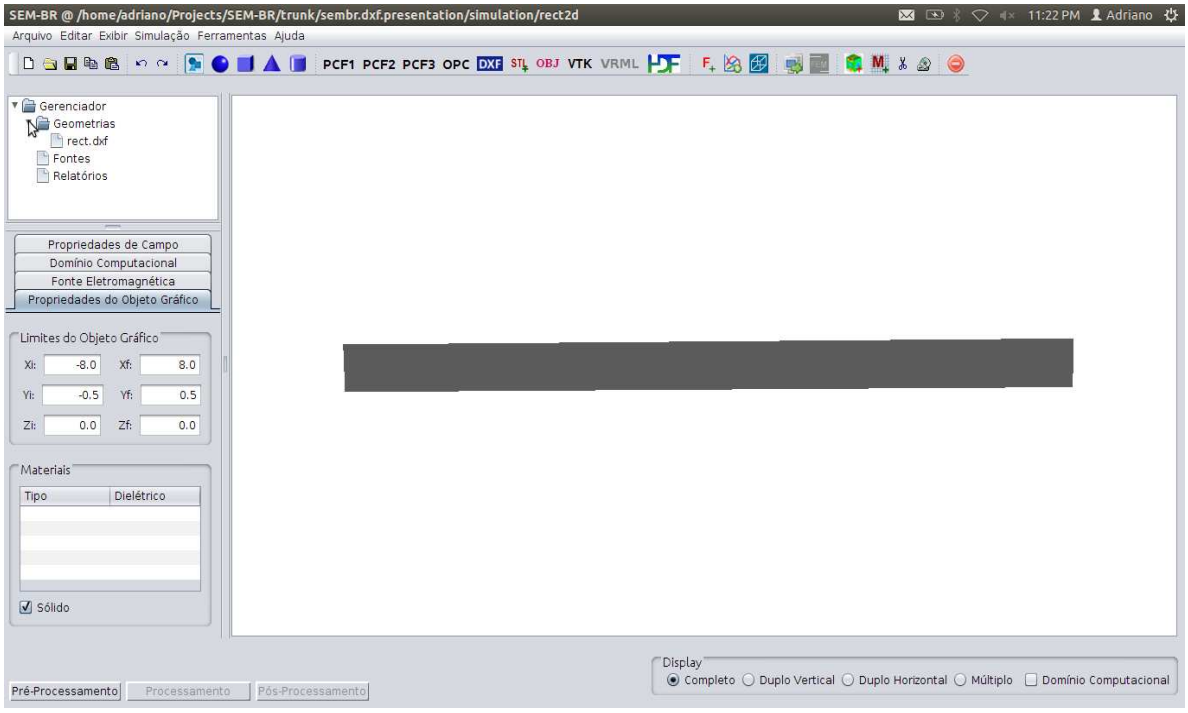
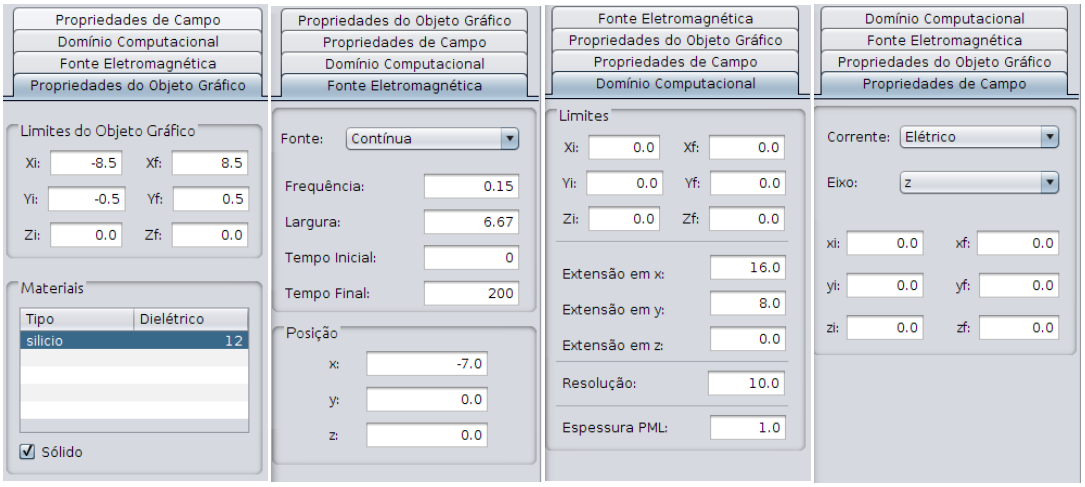


Figura 5.3 – Geometria importada e visualização do nome do arquivo na árvore, à esquerda.

A configuração da simulação é realizada na GUI, através do conjunto de abas disponível à esquerda da tela principal. Cada aba permite a configuração de uma das seguintes propriedades, indicadas no topo de cada aba: campo eletromagnético, fonte eletromagnética, domínio computacional e objeto gráfico. A Figura 5.4 apresenta as abas com os parâmetros configurados para a simulação do guia de onda retangular.



(a) (b) (c) (d)

Figura 5.4 – Abas com as configurações de simulação definidas: (a) Propriedades do Objeto Gráfico; (b) Fonte Eletromagnética; (c) Domínio Computacional; (d) Propriedades de Campo.

O tipo de campo e o eixo de direção são as propriedades que definem a configuração de campo; o tipo de fonte, a frequência, o comprimento de onda, a quantidade de passos de tempo e a posição espacial são as propriedades que definem a configuração da fonte eletromagnética; as extensões nos eixos *x*, *y*, e *z*, a espessura da PML e a resolução definem a configuração do domínio computacional; e a lista de materiais e os limites de extensão definem as propriedades do objeto gráfico.

Após configurar a simulação, o usuário pode visualizar o cenário através do botão "Visualizar Domínio Computacional". A Figura 5.5 apresenta o cenário do domínio de simulação construído para o guia de onda retangular.

Antes de executar o MEEP sobre o cenário configurado, o usuário deve escolher o tipo de Pós-processamento desejado. O botão "Adicionar Relatório" dá acesso a janela de

escolha do tipo de Pós-processamento, conforme ilustrado na Figura 4.12 (a).

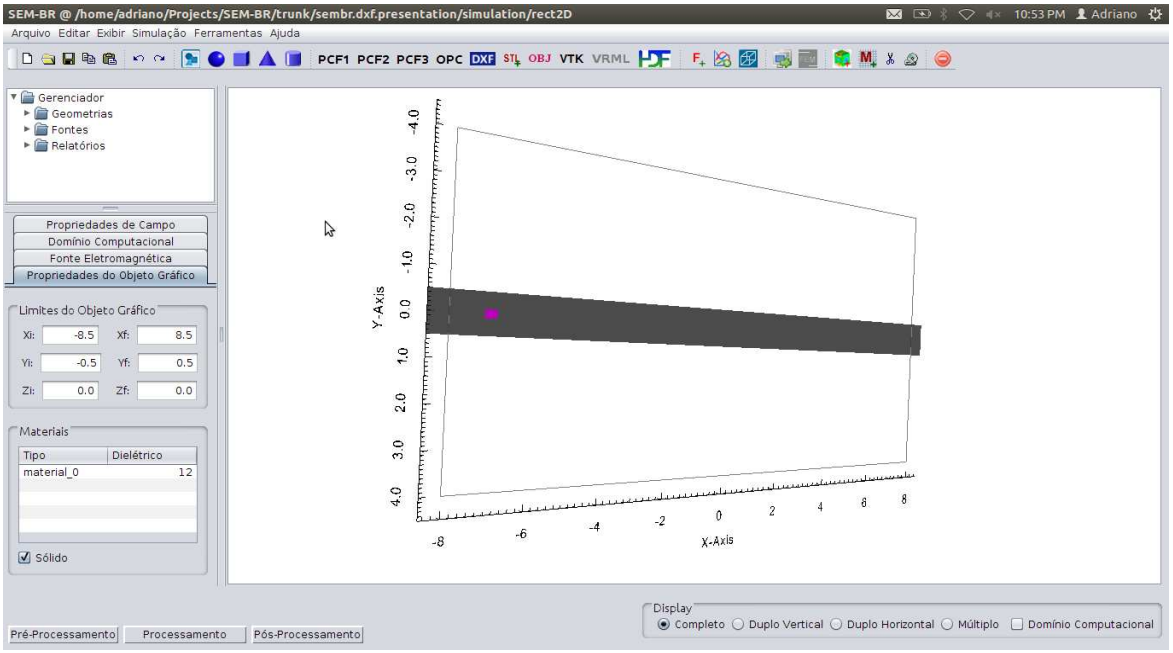


Figura 5.5 – Visualização do cenário gerado para simulação do guia de onda retangular.

Como a simulação é bidimensional, escolheu-se a opção “Filme 2D”, que permite a execução de um vídeo com a propagação do campo no intervalo de tempo definido. Ao inserir o relatório, a árvore de diretórios ilustra a opção de Pós-processamento escolhida pelo usuário, conforme mostra a Figura 5.6.

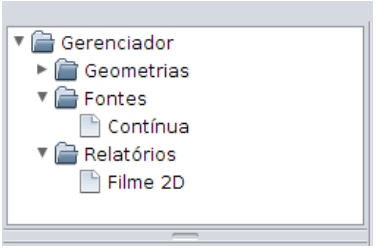


Figura 5.6 – Visualização do relatório de Pós-processamento escolhido pelo usuário.

Depois de definir a configuração da simulação e o relatório de Pós-processamento, o usuário está apto a executar o MEEP através do botão “Simular FDTD”. Durante a

realização da simulação, a barra de progresso permanecerá visível para denotar que a simulação está em execução, conforme ilustra a Figura 4.14 (b). Ao término da execução do MEEP, a barra indica que a simulação foi efetuada, conforme mostra a Figura 5.7.

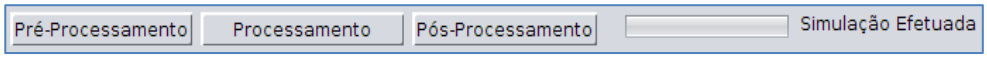


Figura 5.7 – Barra de progresso indicando que o MEEP foi executado.

Para analisar o Pós-processamento escolhido, o usuário deve clicar na marca *(label)* de indicação de "Pós-processamento", na parte inferior da GUI. O ambiente computacional abrirá a janela de configuração de Pós-processamento – ilustrada na Figura 5.8 – que disponibilizará apenas o(s) campo(s) de configuração relativo(s) à(s) opção(ões) de relatório definido(s) pelo usuário. Para a opção de filmes bidimensionais, o usuário pode configurar o intervalo entre os passos de tempo para obtenção das fatias de imagens ou *frames* que formarão o vídeo, as dimensões e o nome do vídeo.

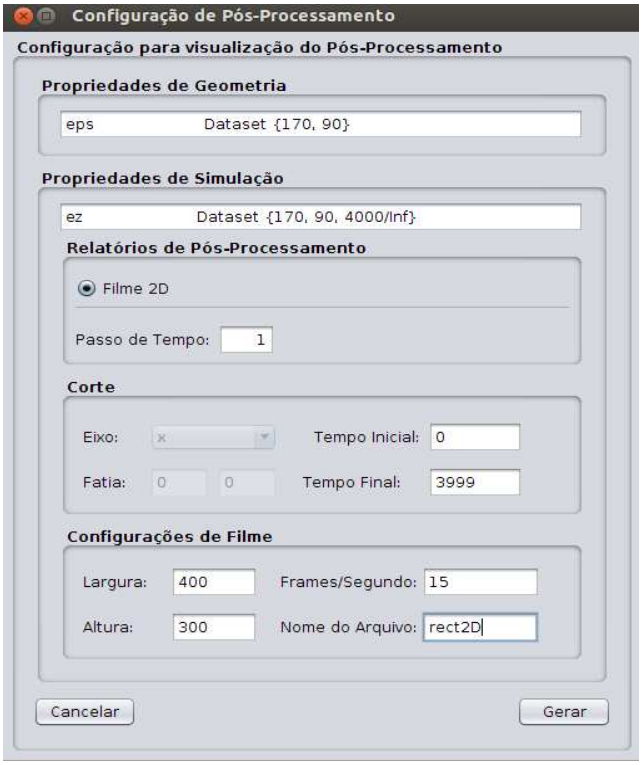


Figura 5.8 – Janela de configuração de Pós-processamento para a simulação do guia de onda retangular.

Após definir as propriedades do filme 2D, o usuário pode clicar no botão “Gerar” da janela de configuração de Pós-processamento, para criar o respectivo vídeo. Ao fechar a janela de configuração, o ambiente computacional abrirá e executará o vídeo da simulação na janela principal, conforme apresenta a Figura 5.9.

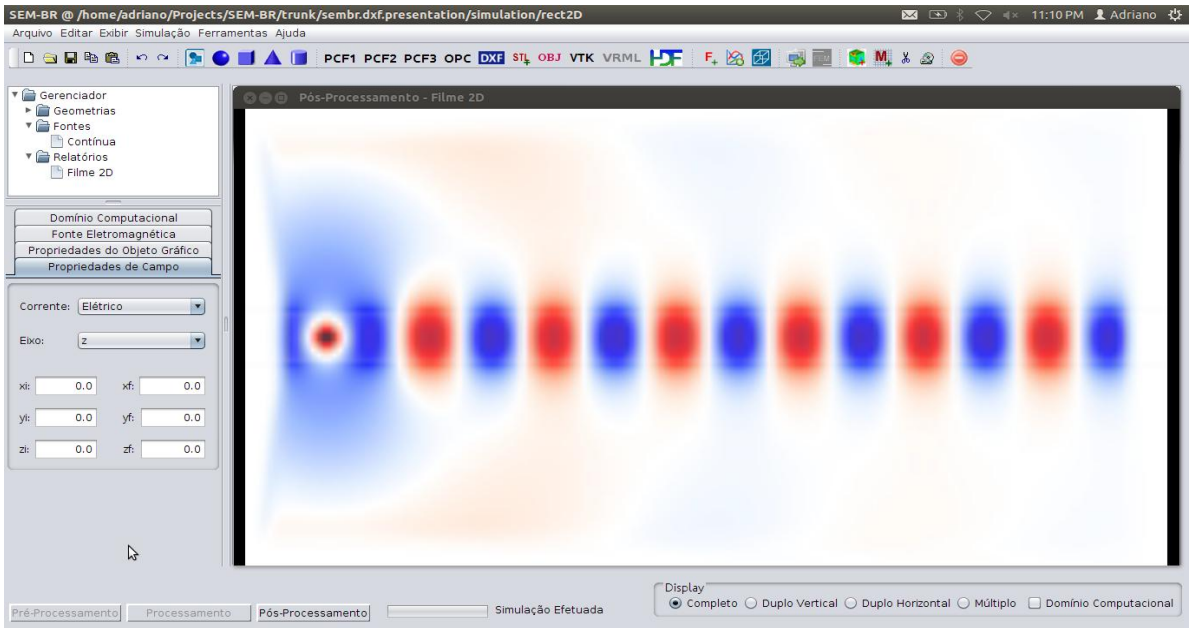


Figura 5.9 – Ilustração do momento de execução do vídeo da simulação realizada no guia de onda retangular.

Afim de validar a equivalência entre o mapeamento da geometria retangular no ambiente computacional e o mapeamento realizado no MEEP original, utilizou-se a ferramenta *wdiff* [67], disponível no sistema operacional Unix/Linux, capaz de verificar a diferença de conteúdo entre dois arquivos. A comparação efetuada pela ferramenta *wdiff* é realizada linha a linha, sendo que cada uma delas é interpretada como uma única palavra.

Os arquivos HDF5 que armazenam as geometrias retangulares, geradas pelas simulações, foram convertidos para o formato de arquivo de texto (extensão *.txt)*, através do utilitário H5totxt – presente no pacote do H5utils – com o propósito de comparar apenas o conteúdo gerado no mapeamento realizado pelo MEEP. A ferramenta *wdiff* aplicada diretamente aos arquivos HDF5 das geometrias não indica diferença de conteúdo entre eles, contudo o número de linhas – palavras – presentes em cada arquivo é diferente,

provavelmente relacionado à divergência entre o conteúdo de metadados presentes entre eles.

Os arquivos de texto obtidos na conversão, correspondentes às geometrias geradas em ambas simulações, possuem a matriz de valores das constantes dielétricas contidas em cada célula – neste caso, 170 x 90 células, resultantes da discretização do espaço de simulação. A aplicação do utilitário *wdiff* nos arquivos de texto retorna a mesma quantidade de linhas em ambos arquivos, que condiz com o número de linhas das matrizes de valores de dielétricos obtidas (neste caso, 170 linhas).

A Figura 5.10 ilustra a saída do comando *wdiff* utilizado, que compara o arquivo *rect_eps_sembr.txt* (gerado pelo ambiente computacional) com o arquivo *rect_eps_pureMeep.txt* (produzido pelo MEEP original). O resultado da comparação aponta que os arquivos tem conteúdo cem por cento em comum (indicado em "100% *common*") e zero por cento de mudança ou diferença (indicado em "0% *changed*"), o que mostra que os arquivos possuem conteúdo idêntico.

```
rect_eps_sembr.txt: 170 words  170 100% common  0 0% deleted  0 0% changed
/home/adriano/Projects/SEM-BR/trunk/sembr.meep/sembr_output/pureMeep_Rectangule/
ez/rect_eps_pureMeep.txt: 170 words  170 100% common  0 0% inserted  0 0% change
d
```

Figura 5.10 – Saída do comando *wdiff* (Unix) utilizado para validar a equidade entre os arquivos de geometria produzidos pelo ambiente computacional e o MEEP original. Os arquivos possuem 170 palavras (linhas) idênticas. A opção *–s* do *wdiff* retorna estatísticas (*statistics*) de arquivos.

O mesmo procedimento foi aplicado aos arquivos com os cálculos de campo eletromagnético. A Figura 5.11 mostra a utilização do comando *wdiff*, que compara o arquivo *ez_sembr.txt* (criado pela conversão do arquivo *.h5*, correspondente à simulação gerada pelo ambiente computacional) com o arquivo *ez_pureMeep.txt* (produzido pela conversão do arquivo *.h5* correspondente à simulação executada no MEEP original). A saída do comando aponta que os arquivos possuem conteúdo cem por cento em comum, o que indica que os arquivos não possuem diferenças e que, portanto, os cálculos de campo foram devidamente computados sobre a geometria retangular mapeada.

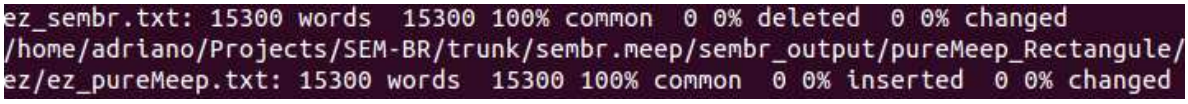


Figura 5.11 – Saída do comando *wdiff* (Unix) utilizado para validar a equidade entre os arquivos de campo elétrico produzidos pelo ambiente computacional e o MEEP original. Os arquivos possuem 15300 palavras (linhas) idênticas.

5.3 OBJETO GEOMÉTRICO CIRCULAR: TESTE COMPARATIVO ENTRE O MEEP-C++ E O AMBIENTE COMPUTACIONAL

O propósito de gerar uma simulação em uma única geometria circular no ambiente computacional é, sobretudo, validar sua capacidade de mapear este tipo geométrico para execução no MEEP. As geometrias circulares servem como um parâmetro interessante para validação da discretização realizada no traçamento de raios, uma vez que sua forma apresenta uma curvatura constante. Um traçamento de raio erroneamente executado pode gerar uma discretização que não represente uma forma razoavelmente aproximada ou tolerável da geometria original. Deve-se notar que a discretização sempre irá gerar uma aproximação da geometria, ao menos que sua resolução tenda ao infinito; a forma geométrica discreta, no FDTD implementado pelo MEEP, tem representação em "forma de escada", isto é, forma de *pixel* no 2D e forma de *voxel* no 3D.

Ademais, formas circulares são comumente usadas como geometrias básicas em modelos geométricos mais complexos, como as fibras de cristais fotônicos – tópicos **5.5** e **5.6** – e os acopladores fotônicos *micro-to-nano* – tópico **4.5.1** – o que justifica uma validação adequada da discretização gerada pelo algoritmo de traçamento de raios implementado.

Como no teste do tópico **5.2**, a simulação foi implementada no MEEP C++, configurada conforme a relação de parâmetros apresentada na Tabela 5.2.

Tabela 5.2 – Parâmetros de configuração e valores utilizados na simulação do objeto geométrico circular.

Domínio computacional	14 x 6 (eixos *x* e *y*, respectivamente)
Largura da PML	1
Modelo de geometria	Circular
Dimensões da geometria	Raio igual à 1
Centro (C) da geometria	C(1, 0) (eixos *x* e *y*, respectivamente)
Constante dielétrica da geometria	12 (silício)
Constante dielétrica externa à geometria	1 (ar)
Fonte	Contínua
Posição (P) da fonte	P(-2, 0) (eixos *x* e *y*, respectivamente)
Frequência da fonte	0.25
Comprimento de onda	4.0
Propriedades de campo	Elétrico, na direção do eixo *z* (componente *Ez*)
Resolução do cenário de simulação	30
Tempo de execução	100 passos de tempo

Uma simulação similar foi gerada no ambiente computacional e armazenada em um diretório escolhido pelo usuário. O acesso ao projeto de simulação pode ser feito pela tela inicial. O usuário pode clicar no botão "Abrir Projeto de Simulação" e selecionar o diretório raiz com o nome do projeto simulação, conforme apresenta a Figura 4.4. Similarmente, o usuário pode abrir um projeto através da tela principal clicando no botão "Abrir", mostrado na Figura 4.18 (b). Neste caso, o ambiente computacional alertará o usuário a salvar o projeto corrente – atualmente aberto – antes de abrir o projeto de simulação desejado. Ambas opções abrirão o projeto escolhido conforme a última configuração salva pelo usuário. O cenário de Pré-processamento definido pode ser visualizado na Figura 5.12.

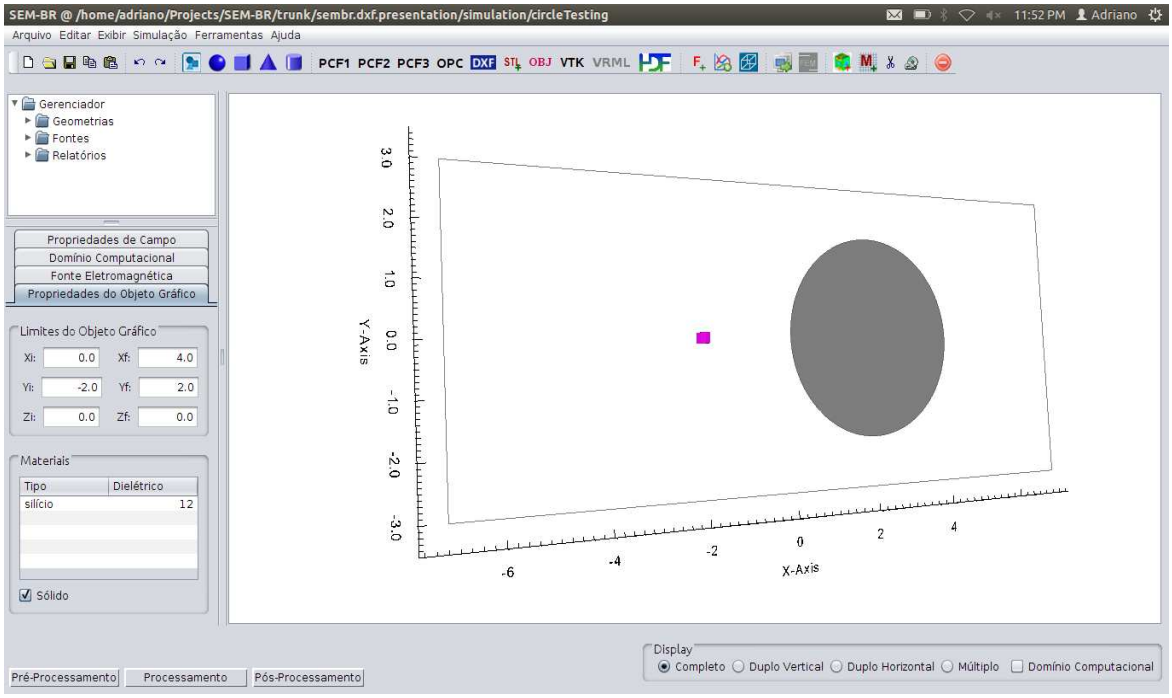


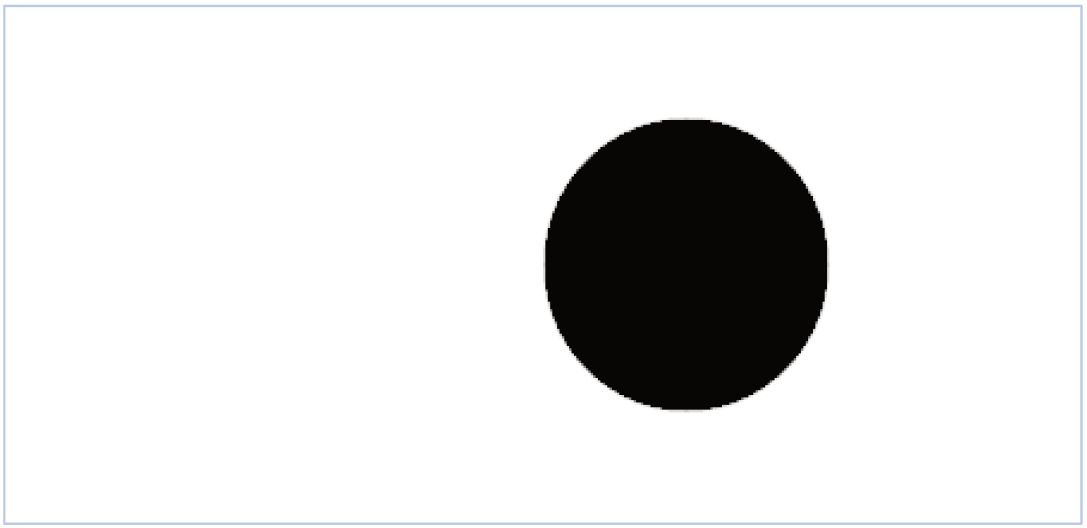
Figura 5.12 – Cenário de Pré-processamento para a simulação com o objeto geométrico circular.

A utilização do comando *wdiff* entre a geometria mapeada pelo ambiente computacional e o MEEP C++ acusa diferença entre as discretizações, como esperado, uma vez que é difícil obter exatidão entre ambos mapeamentos, devido a diferença da estratégia de mapeamento (o *wdiff* apontou diferença de 19% entre linhas correspondentes dos dois arquivos – deve-se considerar, entretanto, que a diferença de um único caracter em uma linha apontará divergência entre as linhas adjacentes). No MEEP C++, o vetor posição retorna a constante dielétrica presente na localização espacial; no ambiente computacional, o VTK computa a intersecção do raio traçado e o ambiente computacional valida a presença ou a ausência de um objeto em determinada posição.

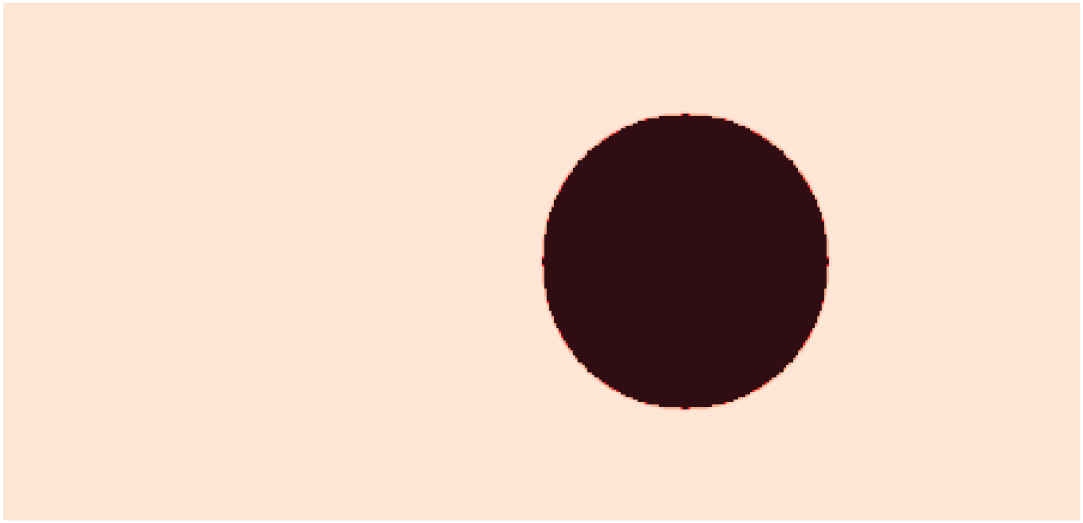
Pode-se validar, visualmente, que os mapeamentos são consistentes e equivalentes, conforme ilustra a Figura 5.13. As imagens das discretizações realizadas no MEEP C++ e no ambiente computacional estão notavelmente aproximadas, com pequenas diferenças nas bordas das geometrias.

Escolheu-se o filme bidimensional como opção de Pós-processamento para a simulação com o objeto geométrico circular. A animação, já armazenada no diretório de

Pós-processamento do projeto de simulação, pode ser acessada pela opção “Filme 2D”, conforme apresentado na Figura 4.17.



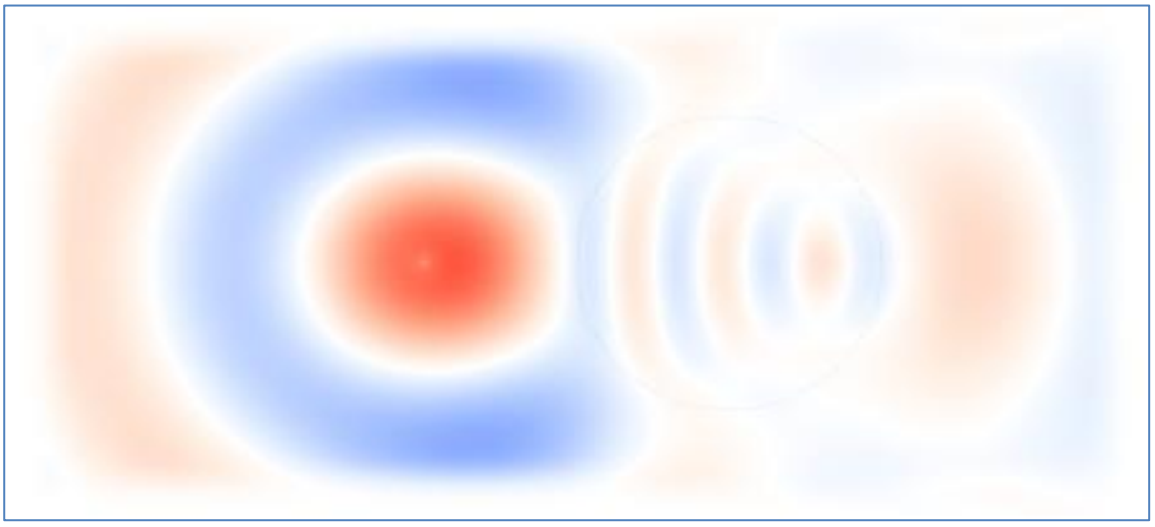
(a)



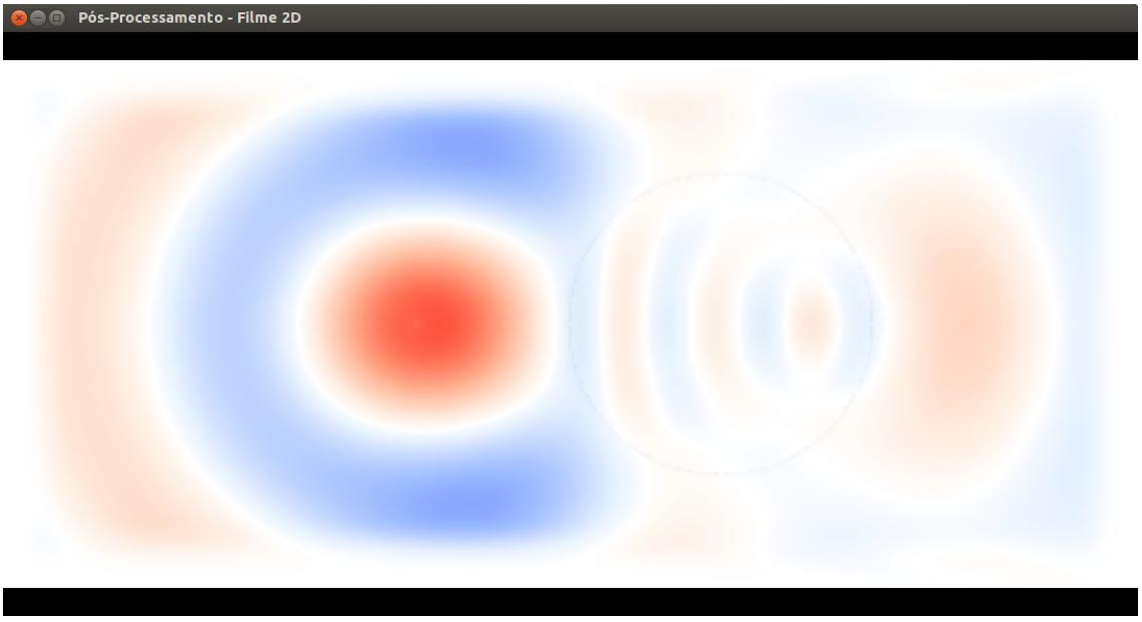
(b)

Figura 5.13 – Comparação visual entre as geometrias circulares geradas (a) no MEEP e (b) no ambiente computacional.

A Figura 5.14 ilustra o resultado do Pós-processamento com uma ilustração da execução do vídeo produzido, em comparação com uma imagem gerada diretamente do H5utils – para a simulação do MEEP-C++ – para uma mesma fatia de tempo.



(a)



(b)

Figura 5.14 – Pós-processamento para o objeto geométrico circular: (a) imagem obtida para a simulação no MEEP-C++ e (b) ilustração da execução do filme 2D para a simulação gerada no ambiente.

5.4 OBJETO GEOMÉTRICO ESFÉRICO: TESTE COMPARATIVO ENTRE O MEEP-C++ E O AMBIENTE COMPUTACIONAL

Este projeto de simulação tem finalidade similar àquela apresentada no tópico anterior (**5.3**): validar a consistência de discretização em geometrias circulares realizada no ambiente computacional, em relação ao mapeamento realizado diretamente no MEEP C++. Além disso, o projeto do ambiente computacional foi estruturado para suportar funcionalidades específicas de Pós-processamento em objetos tridimensionais, como as opções de planos de cortes interativos, que serão inicialmente mostrados neste tópico.

Um objeto gráfico esférico é facilmente gerado no MEEP-C++, apenas estendendo-se a função do círculo – utilizada no tópico anterior – à função da esfera com a introdução do eixo *z*. Os valores para os parâmetros de configuração do cenário de simulação são expressos na Tabela 5.3.

Tabela 5.3 – Configuração do cenário de simulação para a geometria esférica.

Domínio computacional	6 x 4 x 3 (eixos *x*, *y* e *z*, respectivamente)
Largura da PML	0.5
Modelo de geometria	esférico
Dimensões da geometria	Raio igual à 1
Centro (C) da geometria	C(1, 0, 0) (eixos *x*, *y* e *z*, respectivamente)
Constante dielétrica da geometria	2.25 (sílica) [27]
Constante dielétrica externa à geometria	1 (ar)
Fonte	contínua
Posição (P) da fonte	P(-2, 0, 0) (eixos *x*, *y* e *z*, respectivamente)
Frequência da fonte	0.15
Comprimento de onda	6.67
Propriedades de campo	Elétrico, na direção do eixo *x* (componente *Ex*)

Resolução do cenário de simulação	30
Tempo de execução	100 passos de tempo

O cenário de Pré-processamento pode ser visualizado na Figura 5.15. Pode-se notar, com evidência, o domínio de simulação tridimensional configurado antes da realização da simulação.

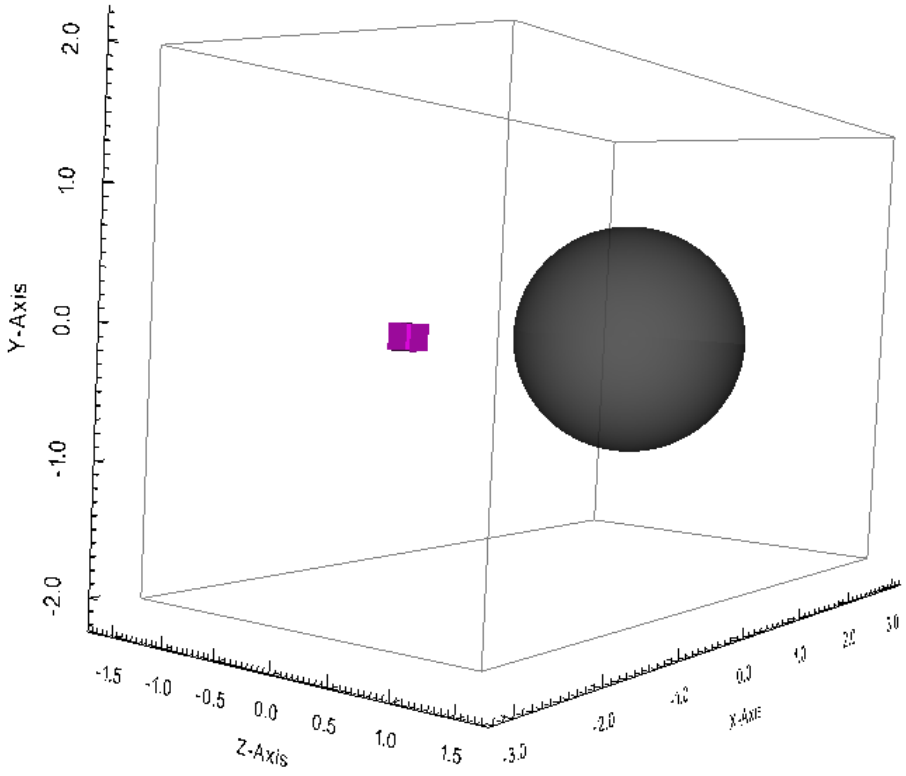
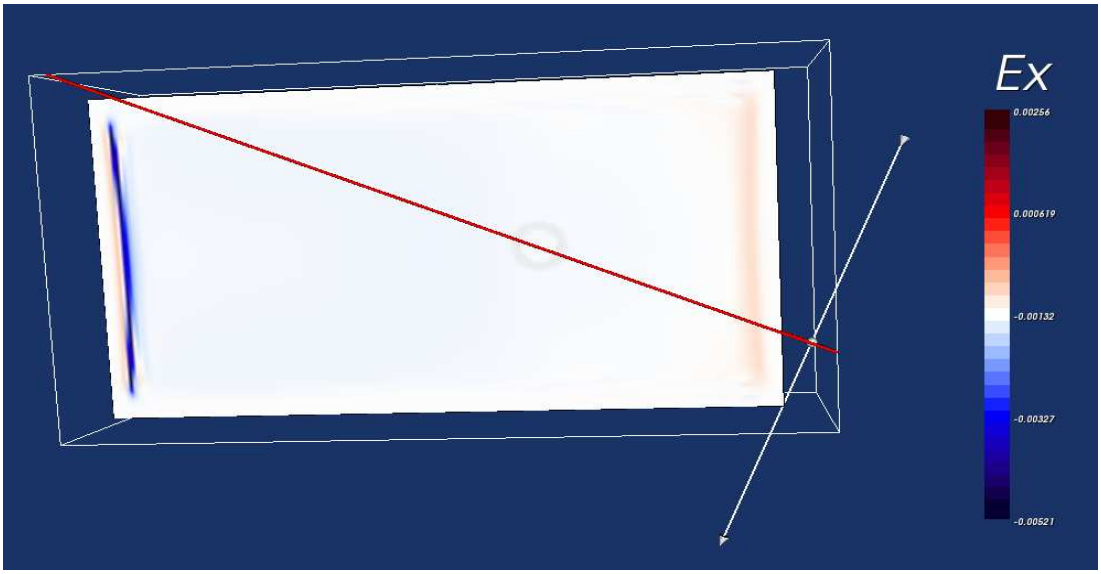


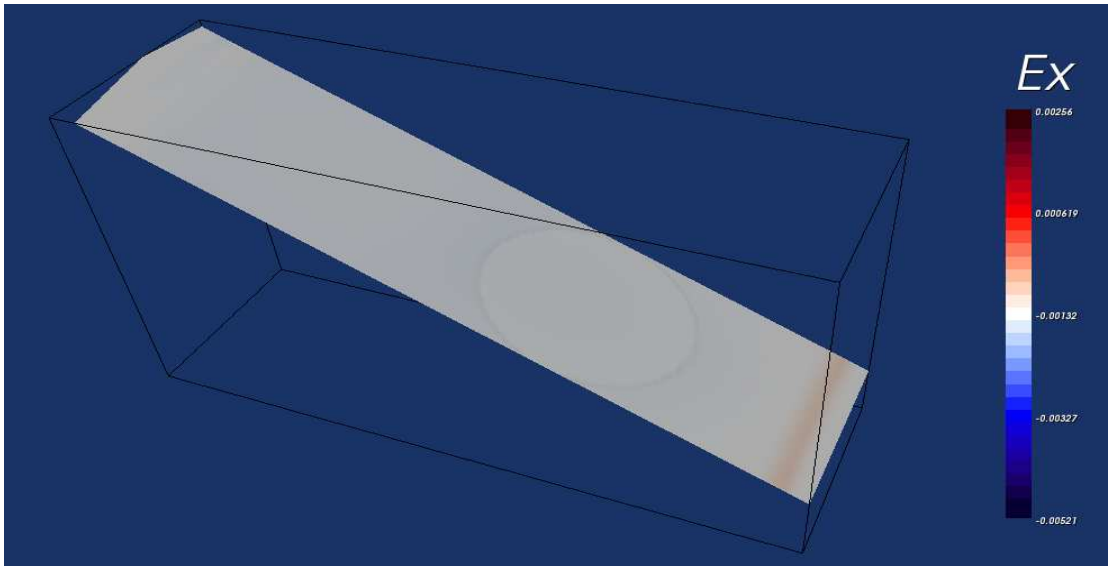
Figura 5.15 – Cenário de Pré-processamento para o objeto geométrico esférico no ambiente computacional.

O usuário pode optar em visualizar um filme bidimensional da simulação 3D. Para isso, deverá escolher um plano de corte para visualização. As outras funcionalidades permitem a realização de cortes arbitrários ou posicionamento de planos ortogonais para análise de perfil de campo. Para realizar um corte arbitrário e analisar o plano resultante, o usuário deve, primeiramente, clicar na opção "Escolher plano de corte", conforme apresentado na Figura 4.17. O ambiente computacional abrirá o domínio simulado, com um

plano que pode ser interativamente posicionado, conforme apresenta a Figura 5.16 (a). Depois de escolher a posição do plano, o usuário pode clicar na opção “Corte” (Figura 4.17), e automaticamente aparecerá o plano escolhido, com a distribuição do campo na região de corte, conforme ilustra a Figura 5.16 (b).



(a)



(b)

Figura 5.16 – (a) Plano interativo para corte sobre domínio simulado e (b) perfil de campo sobre o corte efetuado.

O usuário pode escolher também a opção “Visualizar Cortes Ortogonais” (Figura 4.17) para obter três planos ortogonais que são interativamente deslocados ou rotacionados por todo domínio de simulação, conforme ilustra a Figura 5.17.

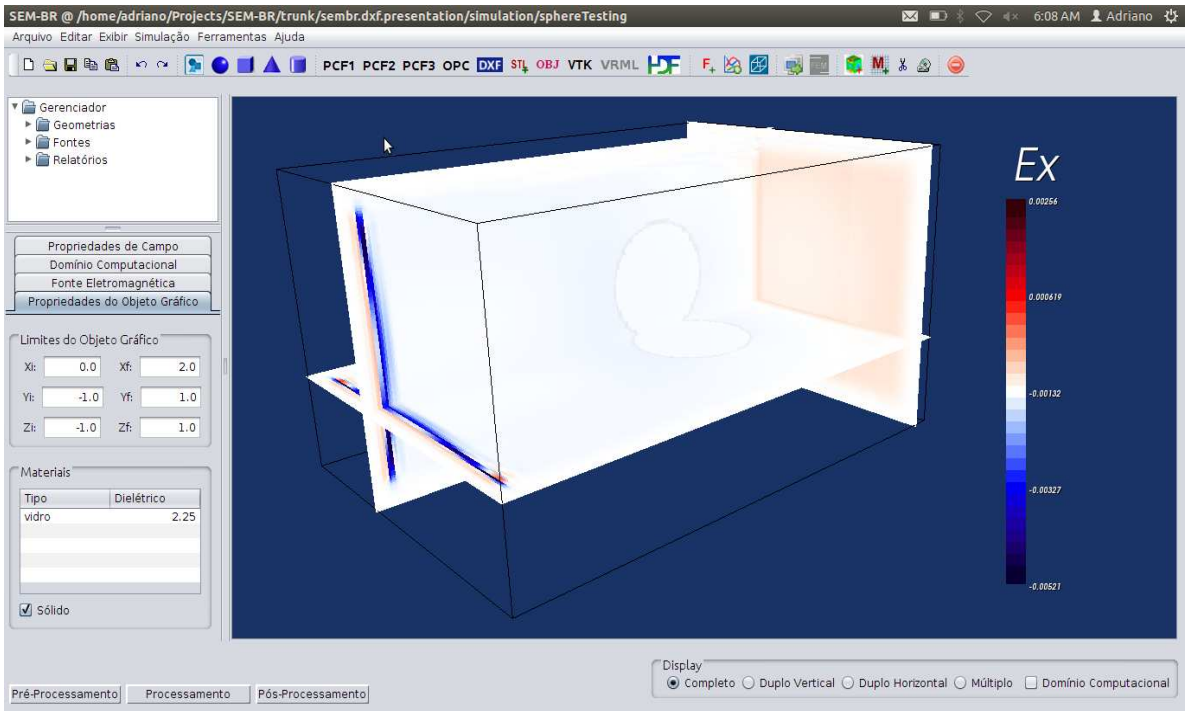


Figura 5.17 – Planos ortogonais que podem se deslocar por todo cenário do projeto de simulação do objeto geométrico esférico.

5.5 FIBRA DE CRISTAL FOTÔNICO – PRIMEIRO EXEMPLO

Com o propósito de validar modelos de guias de ondas mais complexos, este tópico – assim como o tópico **5.6** – apresenta um modelo de Fibra de Cristal Fotônico – PCF (*Photonic Crystal Fiber*) – com um núcleo de sílica puro e perfurações em modelo hexagonal, sem preenchimento com material específico, compostos apenas de ar, como ilustra a Figura 5.18. Este é o modelo representado e implementado pela classe *PCF1* (tópico **4.5.1**).

Evidentemente, este modelo de PCF pode ser modelado nas versões do MEEP

original, uma vez que esta geometria é composta, basicamente, de cilindros. Entretanto, o usuário terá um esforço extra para implementar todo código, uma vez que o posicionamento espacial de todo o conjunto de cilindros e a visualização constante para averiguar se o modelo foi construído corretamente são feitos manualmente.

Figura 5.18 – Modelo de PCF (*PCF1*) implementado para simulação no ambiente computacional [68].

No ambiente computacional, o modelo geométrico da PCF foi construído em forma de objeto através do VTK. Neste sentido, o modelo geométrico está parametrizável e pode ser diretamente conectado à uma interface gráfica para receber apenas os valores que ajustam os parâmetros da geometria, como a posição, altura e raios dos furos (cilindros em modelo triangular) e da fibra (cilindro maior). Ademais, o usuário pode averiguar, instantaneamente, a visualização da geometria criada através do VTK.

Diante da execução bem-sucedida de simulações pelo ambiente computacional nos comparativos com o MEEP-C++ – presenciado nos tópicos anteriores – e da dificuldade de geração da PCF no MEEP original, a simulação da PCF foi realizada apenas no ambiente computacional. A configuração desta simulação assumiu parâmetros hipotéticos, uma vez que o trabalho em [68] disponibilizou apenas os modelos de PCF e não exemplos de simulação efetuadas nas respectivas geometrias. Contudo, a finalidade primordial desta simulação é validar a correta geração de dispositivos complexos para processamento no

MEEP através do ambiente computacional.

A configuração da simulação realizada nesta PCF é dada na Tabela 5.4.

Tabela 5.4 – Configuração do cenário de simulação para a fibra de cristal fotônico *PCF1*.

Domínio computacional	7 x 8 x 7 (eixos *x*, *y* e *z*, respectivamente)
Largura da PML	1.0
Modelo de guia de onda	Fibra de cristal fotônico – *PCF1*
Limites espaciais iniciais (*i*) e finais (*f*) da PCF	(-3, 3, -1.55, 1.55, -3, 3) – (*xi, xf, yi, yf, zi, zf*)
Centro (C) do guia de onda	C(0, 0, 0) (eixos *x*, *y* e *z*, respectivamente)
Constante dielétrica do guia de onda	2.25 (sílica)
Constante dielétrica externa ao guia de onda	1 (ar)
Fonte	contínua
Posição (P) da fonte	P(0, -3.1, 0) (eixos *x*, *y* e *z*, respectivamente)
Frequência da fonte	0.6452
Comprimento de onda	1.55
Propriedades de campo	Elétrico, na direção do eixo *z* (componente *Ez*)
Resolução do cenário de simulação	20
Tempo de execução	77 passos de tempo

O Pré-processamento obtido pode ser conferido na Figura 5.19.

Para análise de Pós-processamento, escolheu-se ambas opções de cortes. Primeiramente, definiu-se um plano de corte arbitrário, através da opção "Escolher plano de corte" (Figura 4.17), como ilustra a Figura 5.20 (a). O corte efetuado através da opção "Corte" (Figura 4.17), com a distribuição do perfil de campo elétrico sobre o plano resultante, pode ser averiguado na Figura 5.20 (b).

Ao clicar na opção "Visualizar Cortes Ortogonais", o ambiente computacional gerou a seguinte visualização, ilustrada na Figura 5.21, para diferentes localizações dos

planos interativamente posicionados pelo dispositivo *mouse*.

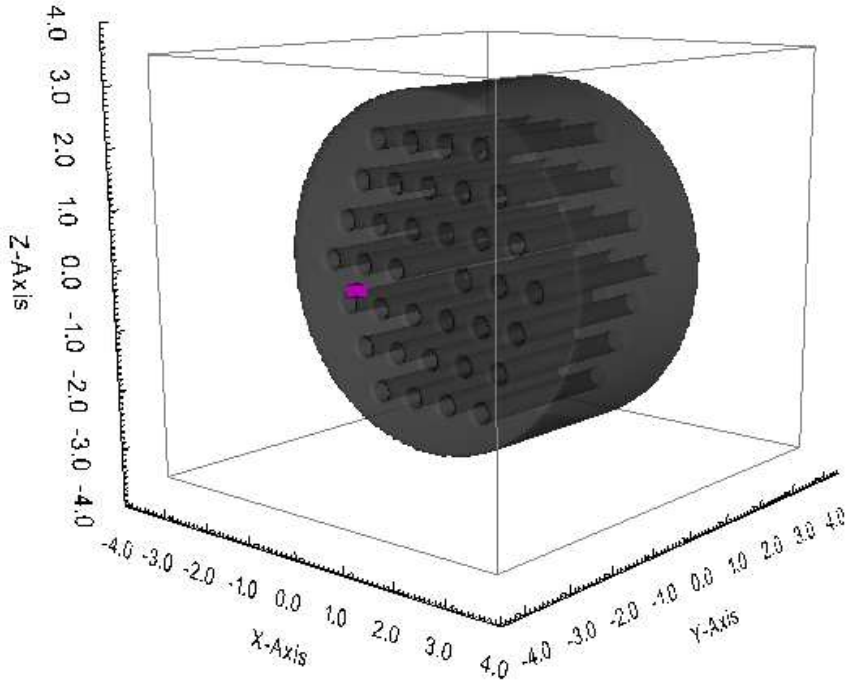
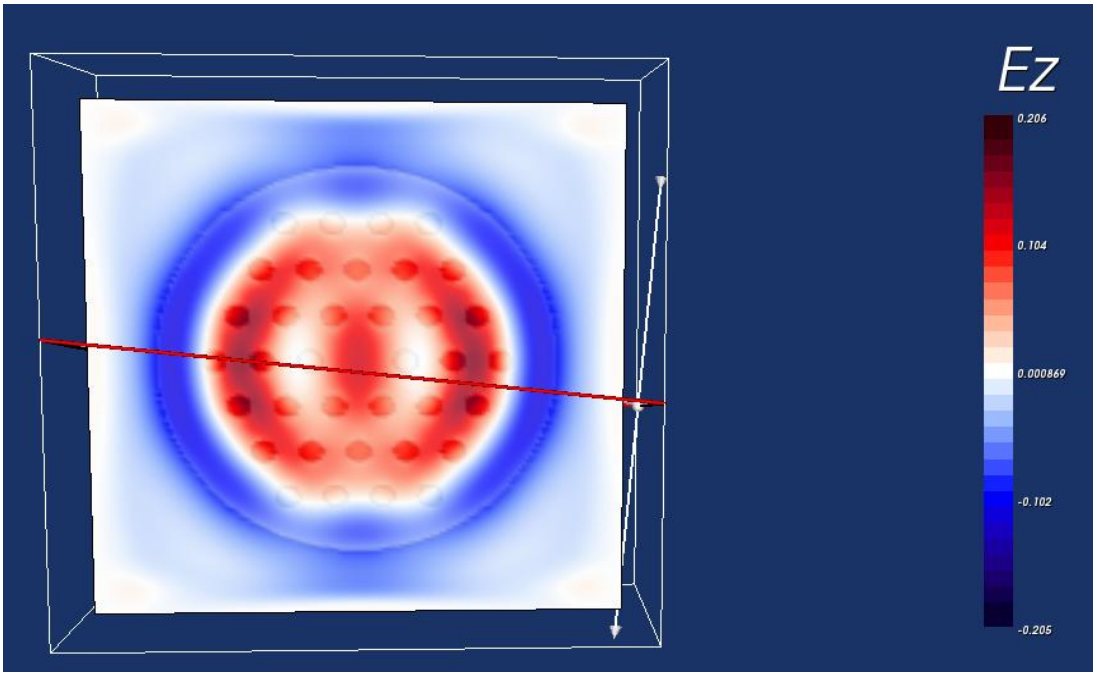
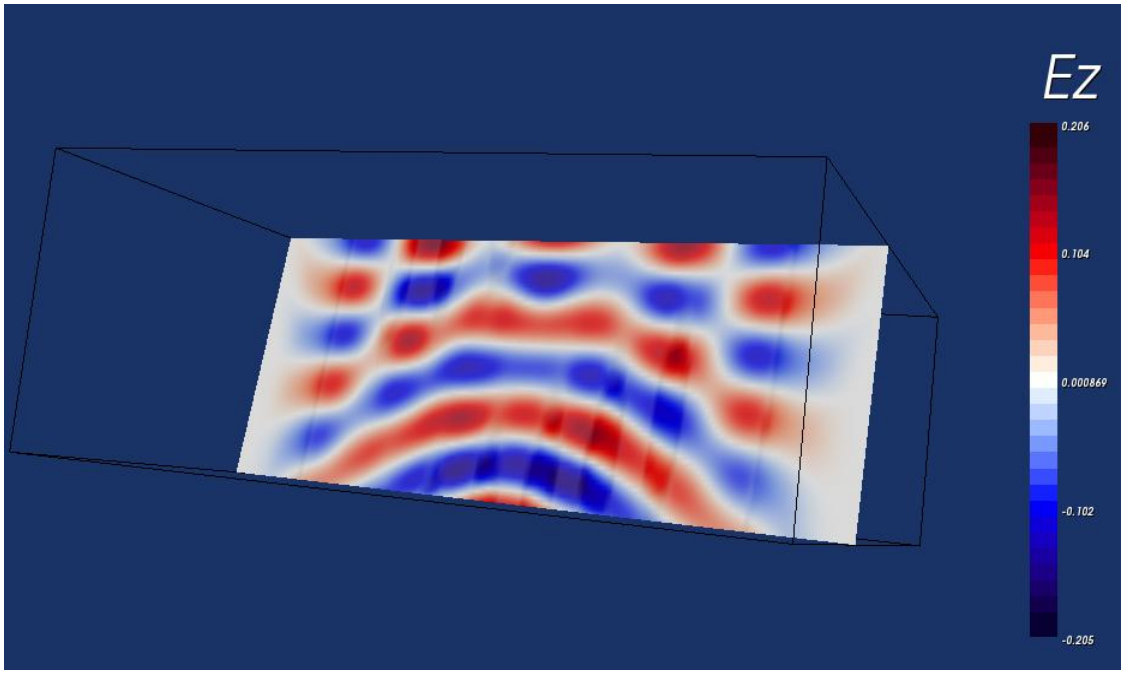


Figura 5.19 – Cenário de Pré-processamento para o guia de onda *PCF1*.



(a)



(b)

Figura 5.20 – (a) Plano de corte posicionado sobre o domínio simulado e (b) o perfil de campo sobre o corte efetuado.

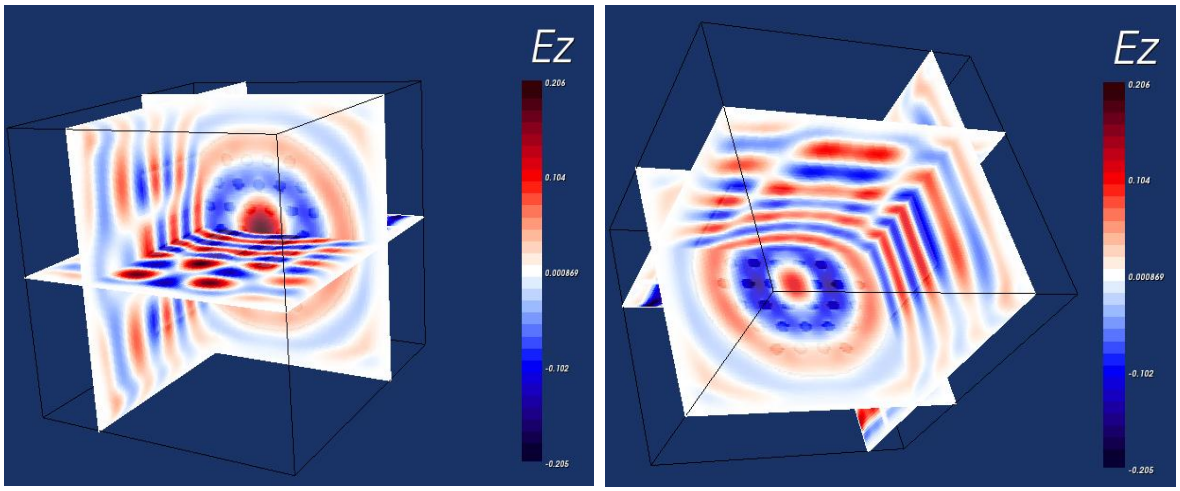


Figura 5.21 – Planos ortogonais posicionados interativamente, em diferentes lugares, sobre o domínio onde a *PCF1* foi simulada.

5.6 FIBRA DE CRISTAL FOTÔNICO – SEGUNDO EXEMPLO

O modelo de PCF mostrado neste tópico foi baseado em [69] e é apresentado na Figura 5.22.

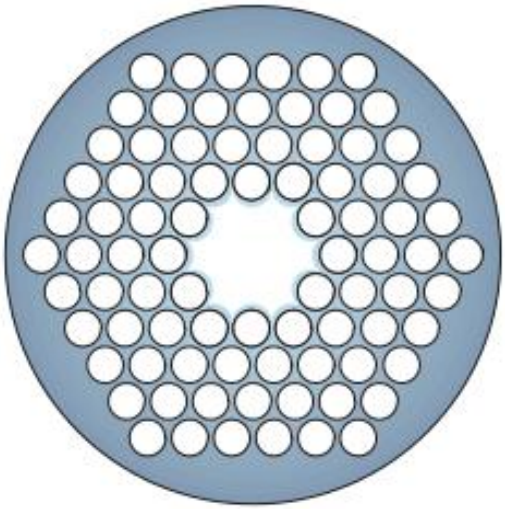


Figura 5.22 – Modelo de PCF (*PCF3*) implementado para simulação no ambiente computacional [69].

Esta fibra de cristal fotônico apresenta núcleo diferenciado, de modo que podemos classificá-la como uma geometria complexa. Considera-se que para um usuário definir este tipo geométrico nas versões originais do MEEP, haverá um grande esforço. Contudo, a PCF é desenvolvida através do VTK sem grande dificuldades, além do que o modelo está, assim como a PCF do tópico anterior, parametrizável através de um objeto de classe.

A configuração desta simulação assumiu parâmetros hipotéticos, uma vez que o trabalho em [69] disponibilizou apenas os modelos de PCF e não exemplos de simulação efetuados nas respectivas geometrias. Contudo, a finalidade primordial desta simulação é validar a correta geração de dispositivos complexos para processamento no MEEP através do ambiente computacional, assim como no tópico anterior.

A configuração do cenário deste projeto de simulação é sumarizado na Tabela 5.5.

Tabela 5.5 – Configuração do cenário de simulação para a fibra de cristal fotônico *PCF3*.

Domínio computacional	8 x 10 x 8 (eixos *x*, *y* e *z*, respectivamente)
Largura da PML	1.0
Modelo de guia de onda	Fibra de cristal fotônico – *PCF3*
Limites espaciais iniciais (*i*) e finais (*f*) da PCF	(-3, 3, -2.5, 2.5, -3, 3) – (*xi, xf, yi, yf, zi, zf*)
Centro (C) do guia de onda	C(0, 0, 0) (eixos *x*, *y* e *z*, respectivamente)
Constante dielétrica do guia de onda	12 (silício)
Constante dielétrica externa ao guia de onda	1 (ar)
Fonte	contínua
Posição (P) da fonte	P(0, -4.05, 0) (eixos *x*, *y* e *z*, respectivamente)
Frequência da fonte	0.6452
Comprimento de onda	1.55
Propriedades de campo	Elétrico, na direção do eixo *z* (componente *Ez*)
Resolução do cenário de simulação	30
Tempo de execução	155 passos de tempo

O cenário de Pré-processamento configurado para o guia de onda *PCF3* pode ser visto na Figura 5.23.

Para averiguar se o modelo foi devidamente mapeado pelo ambiente computacional, utilizou-se diretamente o H5utils para obter-se um corte transversal – perpendicular ao eixo *y* – no meio da PCF. A Figura 5.24 ilustra a geometria mapeada.

Para análise de Pós-processamento, definiu-se um plano de corte arbitrário, através da opção "Escolher plano de corte", ilustrada na Figura 5.25 (a). O corte efetuado através da opção "Corte", com a distribuição do perfil de campo elétrico sobre o plano resultante, pode ser averiguado na Figura 5.25 (b).

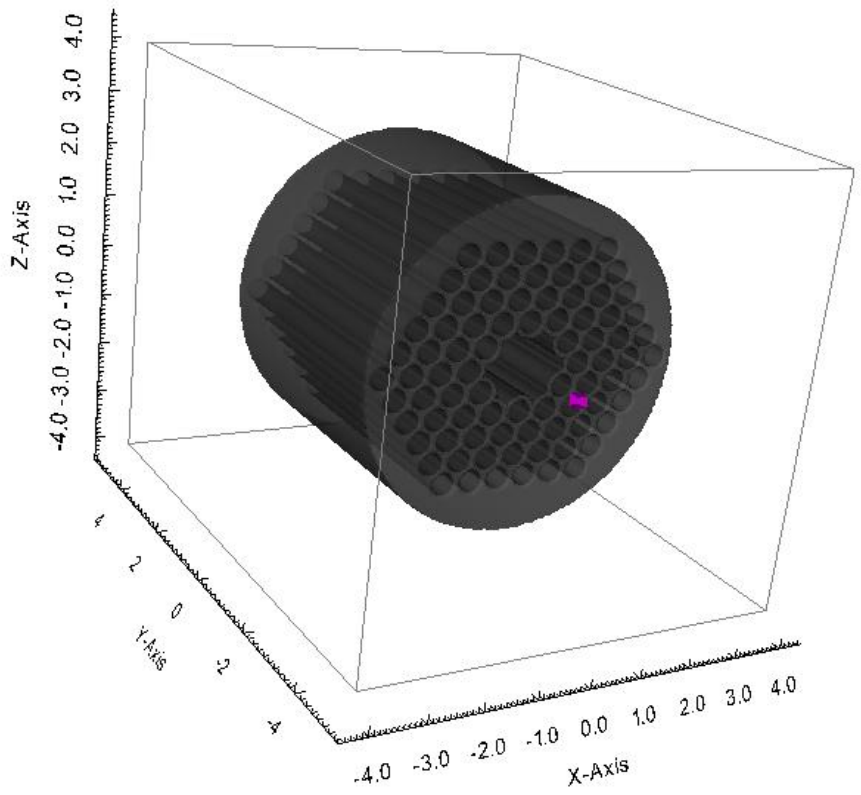


Figura 5.23 – Cenário de Pré-processamento para o guia de onda *PCF3*.

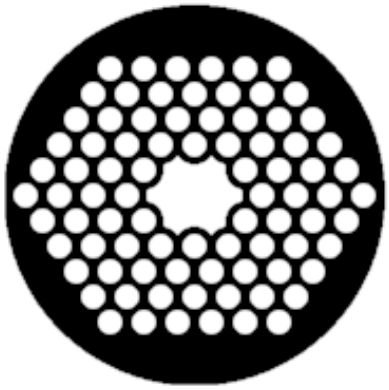
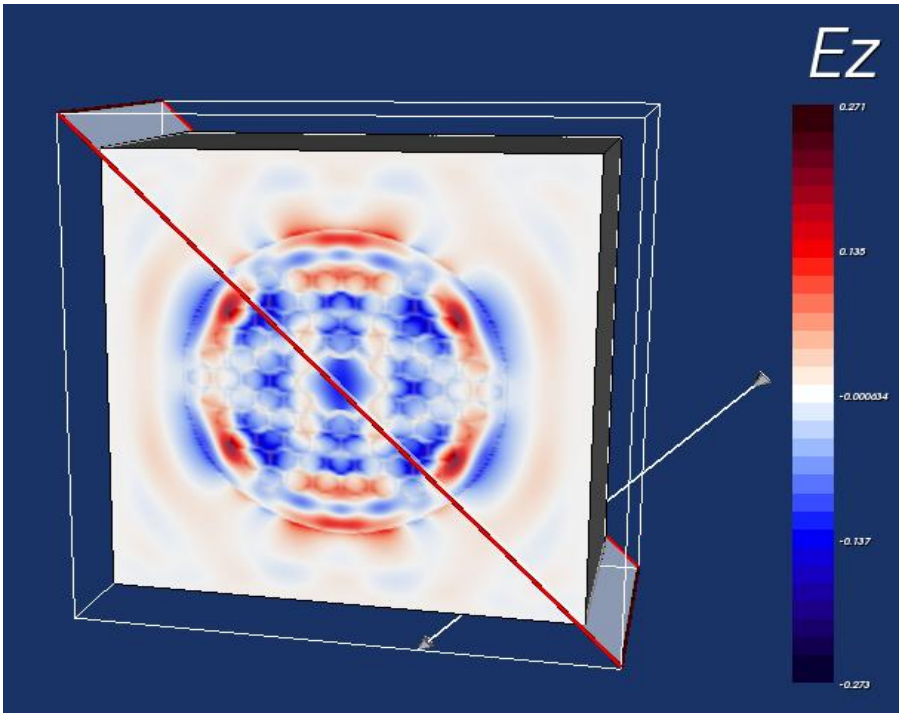
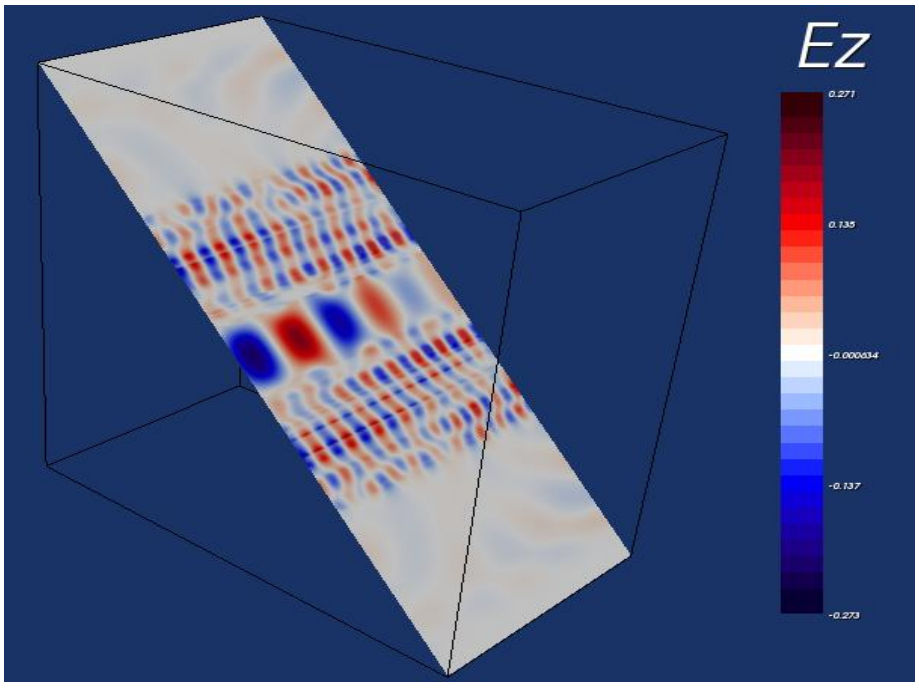


Figura 5.24 – Validação do mapeamento da *PCF3* através do H5utils.



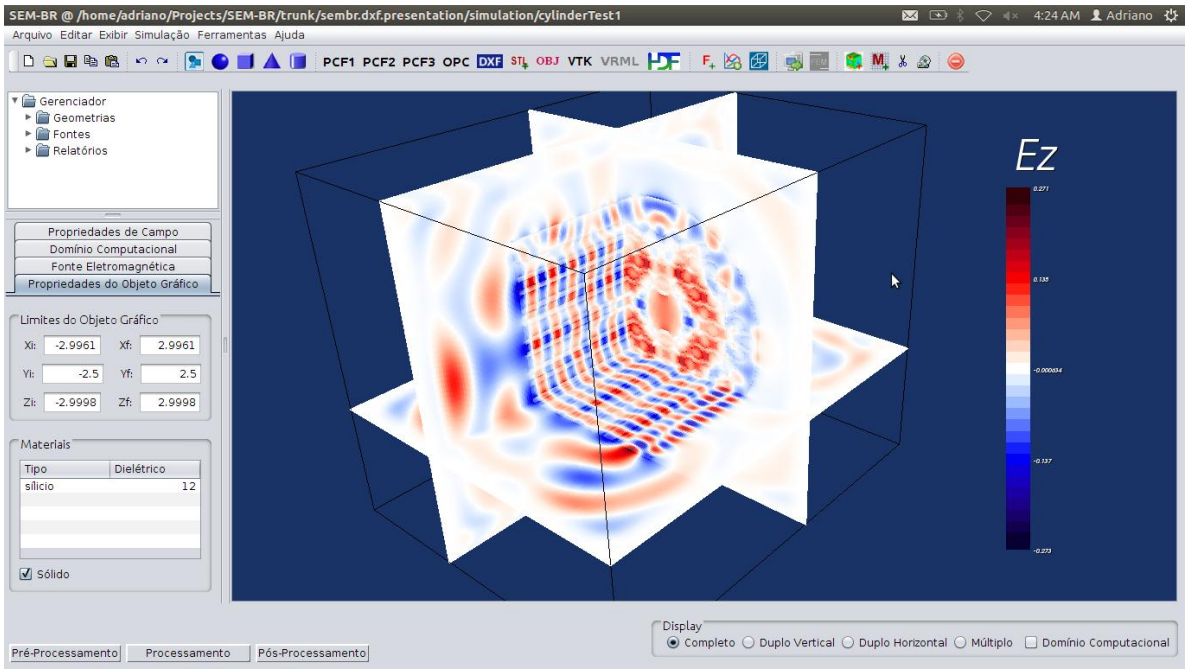
(a)



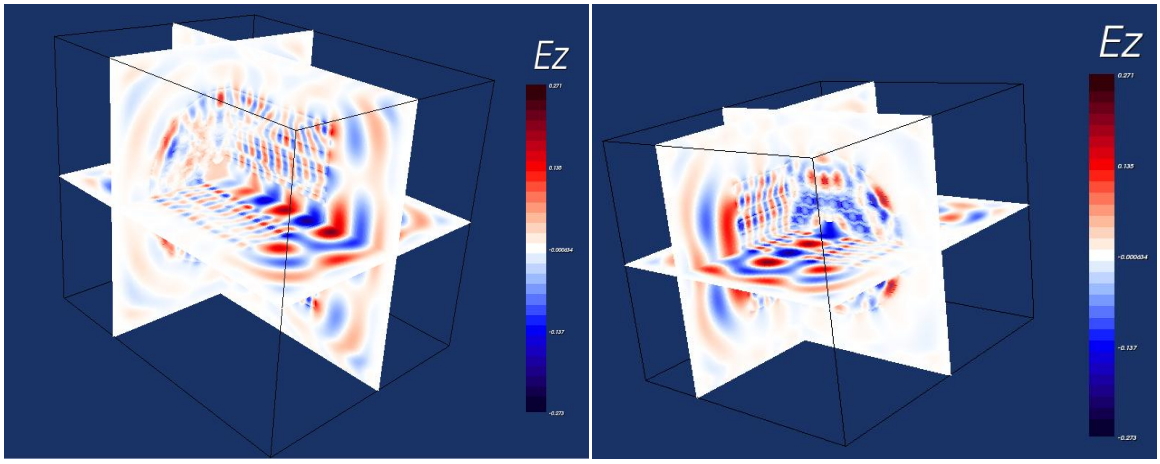
(b)

Figura 5.25 – (a) Plano de corte posicionado sobre o domínio simulado para a *PCF3* e (b) o perfil de campo sobre o corte efetuado.

Ao clicar na opção “Visualizar Cortes Ortogonais”, o ambiente computacional gerou a seguinte visualização, ilustrada nas Figuras 5.26 (a), (b) e (c), para diferentes localizações dos planos, interativamente posicionados pelo dispositivo *mouse*.



(a)



(b) (c)

Figura 5.26 – Planos ortogonais posicionados interativamente, em diferentes lugares – (a), (b) e (c) – sobre o domínio onde a *PCF3* foi simulada.

CAPÍTULO 6

CONCLUSÕES

Este capítulo descreve as conclusões acerca do trabalho desenvolvido no projeto do ambiente computacional. São apresentadas as principais vantagens, desvantagens, os meios de aprimoramento, os trabalhos em andamento e os possíveis trabalhos futuros de continuidade deste *software*, assim como as considerações finais da pesquisa.

6.1 INTRODUÇÃO

Diante do projeto apresentado neste trabalho, pode-se inferir que o ambiente computacional têm potencialidade para tornar-se, juntamente com o MEEP, um simulador de qualidade e de extensa utilidade, aplicável nas áreas de pesquisas e educacionais.

O conjunto de requisitos propostos no Capítulo 1 formou uma proposta de projeto que visou, sobretudo, a criação de um envoltório de funcionalidades propício à viabilizar a utilização do MEEP. A implementação das referidas funcionalidades formou um sistema consistente, testado sobre um conjunto de exigências que o validou sob todas propostas apresentadas.

Em termos de Pré-processamento, conclui-se que o ambiente computacional agregou diversas funcionalidades úteis ao MEEP e, consequentemente, à seus usuários. A capacidade de inserção de geometrias em simulações tornou-se, de certa forma, generalizada, uma vez que o ambiente computacional provê a importação de três tipos diferentes de arquivos gráficos – DXF, STL e OBJ – além de criar modelos gráficos básicos e fotônicos por intermédio do VTK. A capacidade de uma interação objetiva e de execução rápida com os objetos gráficos propicia a utilização deste sistema de Pré-processamento, pois o usuário pode manipular os objetos visualizados em tempo real, seja na interação via

mouse e teclado ou na visualização de superfícies ou *wireframes* [28] dos mesmos. Por fim, a possibilidade de visualização do cenário de simulação – antes da execução do FDTD sobre a cena criada – proporciona, ao usuário, um panorama do que será simulado e previne-o de enviar uma configuração de simulação errônea ao MEEP.

Com relação ao Pós-processamento desenvolvido, conclui-se que o ambiente computacional traz uma nova fronteira de funcionalidades para análises das simulações executadas. As funções disponíveis no H5utils e no VTK formam uma base firme para o desenvolvimento de um Pós-processamento rico. As ferramentas para geração de filmes bidimensionais automáticos e, sobretudo, para a análise de perfis de campos eletromagnéticos em qualquer lugar do domínio computacional simulado, através dos planos de corte e de visualização, formam um conjunto de funcionalidades muito úteis e que estão além das possibilidades de Pós-processamento nas versões originais do MEEP. Assim como no estágio de Pré-processamento, o usuário pode interagir objetivamente – e em tempo real – com os objetos de visualização gerados nesta etapa de processamento.

A modularização do ambiente computacional é um ponto importante quando se trata de um sistema de *software*. A criação de diferentes componentes viabiliza a extensão do sistema, facilita sua integração com outros *softwares* e simplifica sua manutenção. Como este sistema está em constante desenvolvimento e sabendo que outros pacotes de *softwares* relativos à simulações eletromagnéticas podem ser integrados ao ambiente computacional, destaca-se que a modularização do ambiente computacional é de suma importância, sem a qual o sistema poderia não ser estendível, dificultando ou inviabilizando a criação de um sistema funcional completo.

Ainda que o ambiente implementado possua vantagens sobre o MEEP original – como descrito nos parágrafos anteriores – é possível expor suas desvantagens. Apesar da solução de traçamento de raios ser genérica, sua execução é normalmente mais lenta se comparada ao mapeamento de geometrias realizado pelo MEEP original, o que é esperado, pois o traçador de raios do ambiente deve calcular, através do VTK, as intersecções dos raios com o objeto gráfico antes de determinar a constante dielétrica presente em uma determinada localização. A Tabela 6.1 sumariza os tempos de mapeamento de geometrias

entre o MEEP e o ambiente computacional para as comparações efetuadas nos tópicos 5.2, 5.3 e 5.4 (objeto geométrico retangular, circular e esférico, respectivamente).

Tabela 6.1 – Tempos de execução (em segundos) para o mapeamento de geometrias entre o MEEP-C++ e o ambiente computacional.

	MEEP	ambiente computacional
geometria retangular	0.00481009	0.065556
geometria circular	0.0445681	1.74066
geometria esférica	1.31066	536.141

Não obstante, é muito provável que esta solução seja melhorada, uma vez que o próprio VTK oferece outros meios de se calcular interseções que podem ser mais rápidos que a solução atual. Se as novas abordagens mantiverem a consistência atual de mapeamento de geometrias, elas poderão substituir a solução corrente. Um outro meio de acelerar o traçamento de raios é a utilização de computação paralela, visto que a composição da grade de dielétricos (mapeamento geométrico) do objeto gráfico é feita independentemente entre as células da grade. Esse aspecto é fundamental na criação de uma solução paralela eficiente, pois a independência de processamento dos dados em um domínio fragmentado para paralelização influencia drasticamente o desempenho da solução paralelizada [7].

Ademais, considera-se que um sistema que agrega uma série de outros *softwares* – como é o caso do ambiente computacional – oferece maior dificuldade de instalação. Todavia, um pacote de instalação pode automatizar a instalação do ambiente computacional. Em um futuro próximo, pretende-se criar um conjunto de pacotes de instalação para diferentes sistemas operacionais, fazendo com que o ambiente computacional torne-se um *software* multiplataforma.

6.2 TRABALHOS EM ANDAMENTO E FUTUROS

As atividades relacionadas ao desenvolvimento do ambiente computacional, que se encontram em andamento, são as seguintes:

- Instalação e configuração do ambiente e do MEEP no *cluster* disponível no DECOM-FEEC-UNICAMP;
- Integração do Pós-processamento do MIT-MPB através de arquivos HDF5;
- Testes de geração de volumes, em Pós-processamento, através de funções de *ray casting* [30] disponíveis no VTK.

Para agregar novas funcionalidades ao sistema desenvolvido e aperfeiçoar as já existentes, é descrita, abaixo, uma relação de trabalhos futuros que podem ser realizados para compor as novas versões do ambiente computacional:

- Integração total de todas as funcionalidades disponíveis na interface do MEEP com o ambiente computacional;
- Integração total de todas as funcionalidades disponíveis no H5topng;
- Integração da solução FDTD, paralelizada em GPU (*Graphics Processing Unit*), desenvolvida pelo grupo do DECOM-FEEC-UNICAMP [70];
- Criação de pacotes de instalação automática do ambiente computacional;
- Desenvolvimento de um editor de geometrias baseado nas funcionalidades disponíveis no VTK e utilizadas nos dispositivos fotônicos, isto é, automatização da inserção e combinação (operações booleanas) de geometrias básicas para formação de geometrias complexas em Pré-processamento;
- Integração e testes de componentes desenvolvidos em Java para leitura de arquivos GDSII [71], amplamente utilizados na construção de dispositivos fotônicos;
- Inclusão de funcionalidades de Pós-processamento para análise de curvas de nível em campos eletromagnéticos;
- Inclusão de funcionalidades de Pós-processamento para análise de campos

vetoriais em perfis de campos eletromagnéticos;

- Integração de outras técnicas numéricas utilizadas na modelagem de sistemas eletromagnéticos, como o FEM;
- Integração e disponibilização de sistemas de otimização (estratégias evolutivas [40] e redes neurais) empregados na construção de dispositivos fotônicos eficientes. Os dispositivos otimizados poderão ser automaticamente visualizados pelo ambiente computacional;
- Implementação e integração de um sistema *Web* de visualização no ambiente computacional;
- Criação de um sistema de Banco de Dados;
- Inclusão de um sistema de administração de usuário de sistemas;
- Implementação de recursos de tradução automática de línguas baseado em perfis de usuários de diferentes nacionalidades.

6.3 CONSIDERAÇÕES FINAIS

O ambiente computacional completo, incluindo todos os artefatos de sistema – arquivos de código-fonte, instaladores, documentação, entre outros – podem ser obtidos no repositório atual, através do endereço disponível em [62].

Este sistema será devidamente adequado à licença GPL-2 para atender todas as normas de distribuição de *softwares* livres.

REFERÊNCIAS

[1] Oskooi, A. F.; Roundy, D.; Ibanescu, M.; Bermel, P.; Joannopoulos, J. D.; Johnson, S. G. **MEEP**: A flexible free-software package for electromagnetic simulations by the FDTD method. *Computer Physics Communications* 181, pp. 687-702, 2010.

[2] Taflove, A.; Hagness, S. C. **Computational Electrodynamics**: The Finite Difference Time-Domain Method. 3rd. Edition, *Artech House*, 2005.

[3] Yee, K. S. **Numerical solution of initial boundary value problems involving Maxwell´s equations in isotropic media**, *IEEE Trans. Antennas Propragat.* 14, pp. 302-307, 1966.

[4] Farjadpour, A.; Roundy, D.; Rodriguez, A.; Ibanescu, M.; Bermel, P.; Joannopoulos, J. D.; Johnson, S. G.; Burr, G. **Improving accuracy by subpixel smoothing in FDTD**. *Optics Letters* 31 (20), pp. 2972-2974, 2006.

[5] Oh, C.; Escuti, M. J. **Time-domain analysis of periodic anisotropic media at oblique incidence**: an efficient FDTD implementation. *Optics Express* 14 (24), pp. 11870-11884, 2006.

[6] Jin, J. **The Finite Element Method in Electromagnetics**. 2nd. Edition, *Wiley-IEEE Press*, 2002.

[7] Pacheco, P. **An Introduction to Parallel Programming**. *Morgan Kaufmann Publishers Inc*., 2011.

[8] Mishrikey, M. **Analysis and Design of Metamaterials**. *Tese de Doutorado*, ETH-Zürich, Zurique, Suíça, 2010.

[9] **GNU General Public License**. Disponível em: http://www.gnu.org/licenses/gpl.html. Acesso em: 06 março 2013.

[10] **GNU Guile**. Disponível em: http://www.gnu.org/software/guile/. Acesso em: 20 nov. 2013.

[11] Lambert, E.; Fiers, M.; Nizamov, S.; Tassaert, M; Johnson, S. G.; Bienstman, P.; Bogaerts, W. **Python Bindings for the Open Electromagnetic Simulator Meep**. *Computing in Science & Engineering* 13 (3), pp. 53-65, 2011.

[12] **Meep**. Disponível em: http://ab-initio.mit.edu/wiki/index.php/Meep. Acesso em: 20 nov. 2013.

[13] **Python-Meep**. Disponível em: https://launchpad.net/python-meep. Acesso em: 20

nov. 2013.

[14] **C++**. Disponível em: http://www.cplusplus.com/. Acesso em: 21 nov. 2013.

[15] Deitel, P. J. **C++**: How to program. 8th. Edition, *Prentice-Hall, Inc.*, 2011.

[16] **Meep Reference**. Disponível em: http://ab-initio.mit.edu/wiki/index.php/Meep_Reference#geometric-object. Acesso em: 21 nov. 2013.

[17] O´Rourke, J. **Computational Geometry in C**. *Cambridge University Press*, 1998.

[18] **Python Meep Documentation**. Disponível em: http://claudia.intec.ugent.be/software/python-meep/python_meep_documentation. Acesso em: 21 nov. 2013.

[19] **HDF**. Disponível em: http://www.hdfgroup.org/HDF5/. Acesso em: 21 nov. 2013.

[20] **H5utils**. Disponível em: http://ab-initio.mit.edu/wiki/index.php/H5utils. Acesso em: 21 nov. 2013.

[21] **Vis5D+**. Disponível em: http://vis5d.sourceforge.net/. Acesso em: 21 nov. 2013.

[22] **MATLAB®**. Disponível em: http://www.mathworks.com/products/matlab/. Acesso em: 21 nov. 2013.

[23] **HDFView**. Disponível em: http://www.hdfgroup.org/hdf-java-html/hdfview/. Acesso em: 21 nov. 2013.

[24] **Matplotlib**.Disponível em: http://matplotlib.org/. Acesso em: 21 nov. 2013.

[25] **Mayavi2**. Disponível em: http://code.enthought.com/projects/mayavi/. Acesso em: 21 nov. 2013.

[26] **Meep C-plus-plus Reference**. Disponível em: http://ab-initio.mit.edu/wiki/index.php/Meep_C-plus-plus_Reference. Acesso em: 21 nov. 2013

[27] **Meep Tutorial**. Disponível em: http://ab-initio.mit.edu/wiki/index.php/Meep_Tutorial. Acesso em: 21 nov. 2013.

[28] Kitware, Inc. **The VTK User´s Guide**. 11th Edition, *Kitware, Inc.*, 2010.

[29] **VTK – Visualization Toolkit**. Disponível em: http://www.vtk.org/. Acesso em: 21 nov. 2013.

[30] Schoroeder, W.; Martin, K.; Lorensen, W. **The Visualization Toolkit** - An Object-Oriented Approach to 3D Graphics. 4th. Edition, *Prentice-Hall, Inc.*, 2006.

[31] **VTK Examples**. Disponível em: http://www.vtk.org/Wiki/VTK/Examples. Acesso em: 21 nov. 2013.

[32] **AUTOCAD®**. Disponível em: http://www.autodesk.com/products/autodesk-autocad. Acesso em: 21 nov. 2013.

[33] **AUTODESK®**. Disponível em: http://www.autodesk.com/. Acesso em: 21 nov. 2013.

[34] **DXF Reference**. Disponível em: http://images.autodesk.com/adsk/files/autocad_2012_pdf_dxf-reference_enu.pdf. Acesso em: 21 nov. 2013.

[35] **DXF Specification**. Disponível em: http://www.martinreddy.net/gfx/3d/DXF12.spec. Acesso em: 21 nov. 2013.

[36] **STL File Format**. Disponível em: http://mech.fsv.cvut.cz/~dr/papers/Lisbon04/node2.html. Acesso em: 21 nov. 2013.

[37] **VTK C++ Examples**. Disponível em: http://www.vtk.org/Wiki/VTK/Examples/Cxx. Acesso em: 21 nov. 2013.

[38] **Object Files Specification**. Disponível em: http://www.martinreddy.net/gfx/3d/OBJ.spec. Acesso em: 21 nov. 2013.

[39] Santos, C. H. S.; Gonçalves, M. S.; Hernández-Figueroa, H. E. **Designing Novel Photonic Devices by Bio-Inspired Computing**. *IEEE Photonics Technology Letter* 22 (15), pp. 1177-1179, 2010.

[40] Santos, C. H. S.; Gonçalves, M. S.; Hernández-Figueroa, H. E. **Evolutionary Strategy Algorithm Applied to Optimize Micro-to-nano Coupler Devices**. *International Microwave & Optoelectronics Conference (IMOC)*, IEEE-SBMO, Natal, Rio Grande do Norte, 2011.

[41] **MIT Photonic Bands**. Disponível em: http://ab-initio.mit.edu/wiki/index.php/MIT_Photonic_Bands. Acesso em: 21 nov. 2013.

[42] Chuang, S. L. **Physics of Photonic Devices**. *John Wiley & Sons*, 2009.

[43] Picanço, P. R. **Desenvolvimento de uma interface integrada para o projeto e análise de antenas utilizando o método das diferenças finitas no domínio do tempo (FDTD)**. *Dissertação de Mestrado*, Universidade de Brasília, Brasília, 2006.

[44] **Mencoder**. Disponível em: http://www.mplayerhq.hu. Acesso em: 22 nov. 2013.

[45] Berenger, J. **A Perfectly Matched Layer for the Absorption of Electromagnetic Waves**. *Journal of Computational Physics 114* (2), pp. 185-200, 1994.

[46] **Meep Introduction**. Disponível em: http://ab-initio.mit.edu/wiki/index.php/Meep_Introduction. Acesso em: 22 nov. 2013.

[47] **HDF5 Tools**. Disponível em: http://www.hdfgroup.org/HDF5/doc/RM/Tools.html. Acesso em: 22 nov. 2013.

[48] **H5topng**. Disponível em: http://ab-initio.mit.edu/H5utils/h5topng-man.html. Acesso em: 22 nov. 2013.

[49] **Color Table in H5topng**. Disponível em: http://ab-initio.mit.edu/wiki/index.php/Color_tables_in_h5topng. Acesso em: 22 nov. 2013.

[50] **UML**. Disponível em: http://www.uml.org/. Acesso em: 22 nov. 2013.

[51] **COMSOL®**. Disponível em: http://www.comsol.com. Acesso em: 22 nov. 2013.

[52] **CST®**. Disponível em: https://www.cst.com/. Acesso em: 22 nov. 2013.

[53] **EmGine**. Disponível em: http://www.petr-lorenz.com/emgine/. Acesso em: 22 nov.2013.

[54] **MMTL**. Disponível em: http://mmtl.sourceforge.net/. Acesso em: 22 nov. 2013.

[55] **MPI Forum**. Disponível em: http://www.mpi-forum.org/. Acesso em: 22 nov. 2013.

[56] Fowler, M. **UML Distilled.** 3rd. Edition, *Addison-Wesley*, 2003.

[57] **Cygwin**. Disponível em: http://www.cygwin.com/. Acesso em: 22 nov. 2013.

[58] **Oracle®**. Disponível em: http://www.oracle.com/. Acesso em: 22 nov. 2013.

[59] Santos, C. H. S.; Gonçalves, M. S.; Ambrósio, L. A.; Buck, R. M.; Freitas, I. J. F.; Ng, J.; Hernández-Figueroa, H. E.; Hernández, M. G. F. **New Three-dimensional Multiplatform Electromagnetic Simulator to Analyze Biological Effects**. *8th*. *Mediterranean Microwave Symposium IEEE*, Damascus, Syria, 2008.

[60] **VLCJ**. Disponível em: http://www.capricasoftware.co.uk/projects/vlcj/index.html. Acesso em: 22 nov. 2013.

[61] **Google Code**. Disponível em: https://code.google.com/. Acesso em: 22 nov. 2013.

[62] **SEM-BR**. Disponível em: https://code.google.com/p/sembr-geometric-visualization/. Acesso em: 22 nov. 2013.

[63] **Swing Application Framework**. Disponível em: http://docs.oracle.com/javase/tutorial/uiswing/components/. Acesso em: 22 nov. 2013.

[64] **AWT**. Disponível em: http://docs.oracle.com/javase/7/docs/technotes/guides/awt/. Acesso em: 22 nov. 2013.

[65] **XML – Extensible Markup Language**. Disponível em: http://www.w3.org/XML/. Acesso em: 22 nov. 2013.

[66] Tulach, J. **Practical API Design**: Confessions of a Java Framework Architect. *Apress*, 2008.

[67] **Wdiff command**. Disponível em: https://www.gnu.org/software/wdiff/manual/html_node/wdiff.html. Acesso em: 22 nov. 2013.

[68] **RP Photonics Encyclopedia**. Disponível em: http://www.rp-photonics.com/photonic_crystal_fibers.html. Acesso em: 25 nov. 2013.

[69] Knight, C. J. **Photonic Crystal Fibers**. *Nature* 424, pp. 847-851, 2003.

[70] Faria, M.; Ferrera, A.; Santos, C. H. S.; Salazar, Z. C.; Hernández-Figueroa, H. E. **One and Two Dimensional Devices Electromagnetic Simulation Using Parallelism on GPUs**. *Microwave & Optoelectronics Conference (IMOC)*, IEEE-SBMO, Natal, Rio Grande do Norte, 2011.

[71] **Java GDSII API (JGDS)**. Disponível em: http://jgds.sourceforge.net/. Acesso em: 22 nov. 2013.