

UNIVERSIDADE ESTADUAL DO NORTE FLUMINENSE
LABORATÓRIO DE ENGENHARIA E EXPLORAÇÃO DE PETRÓLEO

SIMULADOR DE DIFUSÃO TÉRMICA 3D
TRABALHO DE CONCLUSÃO DE CURSO

Versão 1:
Nicholas de Almeida Pinto
André Duarte Bueno
Guilherme Rodrigues Lima

MACAÉ - RJ
Abril - 2022

Contents

1	Introdução	1
1.1	Escopo do problema	1
1.2	Objetivos	3
1.3	Metodologia utilizada	4
2	Concepção	5
2.1	Características gerais	5
2.2	Especificação	6
2.3	Requisitos	7
2.3.1	Requisitos funcionais	7
2.3.2	Requisitos não funcionais	8
2.4	Casos de uso	9
2.4.1	Diagrama de caso de uso geral	9
2.4.2	Diagrama de caso de uso específico	10
3	Elaboração	12
3.1	Análise de domínio	12
3.2	Formulação modelos teóricos	13
3.2.1	Termos e Unidades	13
3.2.2	Formulação teórica	14
3.2.3	Condição de fronteira	18
3.2.4	Demonstrações matemáticas	20
3.2.5	Condutividade térmica variável	21
3.3	Formulação modelos computacionais	22
3.3.1	O que é processamento paralelo?	22
3.3.2	<i>Processamento paralelo com múltiplas-threads - multi-thread</i>	24
3.3.3	Renderização 3D	25
3.4	Identificação de pacotes	29
3.5	Diagrama de pacotes	29
4	Análise Orientada a Objeto	31
4.1	Dicionário das classes	31

4.2	Diagrama de sequência	33
4.2.1	Diagrama de sequência - cenário geral	33
4.2.2	Diagrama de sequência - adicionando novo material/correlação . . .	34
4.2.3	Diagrama de sequência - análise de resultados	35
4.3	Diagrama de comunicação	35
4.4	Diagrama de máquina de estado	36
4.5	Diagrama de atividades	37
5	Projeto	39
5.1	Projeto do sistema	39
5.2	Diagrama de componentes	39
5.3	Diagrama de implementação	40
6	Ciclos de Planejamento/Detalhamento	42
6.1	Versão 0.1 - Uso modo terminal e biblioteca SFML para saída gráfica . . .	42
6.2	Versão 0.2 - Adição de visualização para os tipos de materiais	43
6.3	Versão 0.3 - Adição de atalhos na tela	43
6.4	Versão 0.4 - Mudança para biblioteca Qt	45
6.5	Versão 0.5 - Melhorias na interface gráfica - usabilidade	46
7	Ciclos Construção - Implementação	48
7.1	Versão 0.3 - Código fonte - SFML	48
7.2	Versão 0.5 - Código fonte - Qt	48
8	Teste	103
8.1	Validação do simulador	103
8.2	Injeção de calor em reservatório - comparação com outro simulador . . .	108
8.3	Injeção de calor em reservatório - modelo five-spot	110
8.4	Injeção de calor em reservatório - modelo 1	113
8.5	Injeção de calor em reservatório - modelo 2	116
8.6	Resfriamento de processadores	119
9	Documentação	121
9.1	Documentação do usuário	121
9.2	Documentação do desenvolvedor	121
10	Manual do Usuário	123
10.1	Instalação	123
10.1.1	Dependências	123
10.2	Interface gráfica	123
10.3	Como adicionar materiais	125
10.3.1	Método da correlação ou constante	126

10.3.2 Método de interpolação	126
10.4 Como gerar relatório em PDF	127
11 Disciplinas Relacionadas a Transferência de Calor	132

List of Figures

1.1	Etapas para o desenvolvimento do software - <i>projeto de engenharia</i>	4
2.1	Diagrama de caso de uso geral	10
2.2	Diagrama de caso de uso específico: adição de novo material e interpolação	11
3.1	Tipos de malha, (a) bloco-centrado e (b) ponto-distribuído	14
3.2	Representação de condutividade térmica em série	17
3.3	Malha utilizada para calcular a temperatura de um ponto, onde cada ponto é o centro dos blocos	18
3.4	Análise da fronteira de Neumann	19
3.5	Processamento serial	23
3.6	Processamento paralelo	23
3.7	Ilustração dos três casos de paralelismo implementados para duas camadas com 9 células cada, e um processador com duas <i>thread</i>	25
3.8	(a) Observador alinhado com uma das faces do cubo. (b) observador não está alinhado e não foram removidas arestas ocultas. O cérebro consegue interpretar que é um objeto 3D, mas fica confuso entre os casos (c) e (d) .	26
3.9	(a) o cubo está com ângulos nulos. (b) os ângulos α e β estão com valor de 0.1 radianos	27
3.10	Mesmo desenho da Figura 3.9, mas agora renderizando a partir de triângulos	27
3.11	(a) mostra um caso onde a normal é na direção do leitor e (b) mostra um caso onde a normal é para dentro da folha	28
3.12	Diagrama de Pacotes	30
4.1	Diagrama de classes	33
4.2	Diagrama de sequência geral	34
4.3	Diagrama de sequência mostrando a importação de um novo material . . .	34
4.4	Diagrama de sequência mostrando a análise de resultados	35
4.5	Diagrama de comunicação	36
4.6	Diagrama de máquina de estado para a classe CSimuladorTemperatura . .	37
4.7	Diagrama de atividades para o algoritmo de renderização CRender3D::Renderizacao() .	38
5.1	Diagrama de componentes	40

5.2	Diagrama de implementação	41
6.1	Versão 0.1, simples e utilizando a biblioteca <i>SFML</i>	42
6.2	Versão 0.2, simples, mas contendo uma segunda janela dos materiais	43
6.3	Versão 0.3, completa e complexa, mas muito lenta	44
6.4	<i>Qt Creator</i>	45
6.5	Versão 0.4, inicial e incompleta, mas utilizando a biblioteca <i>Qt</i>	46
6.6	Versão 0.5, final. Na direita é apresentado a visualização 3D do objeto desenhado	47
8.1	Aplicação do problema unidimensional no simulador. (a) é no tempo inicial e (b) depois de 100 segundos.	104
8.2	Comparação da solução da equação de calor com o resultado do simulador .	107
8.3	Resultados da simulação com renderização 3D	109
8.4	Gráficos da simulação para o tempo de 3.600 segundos	110
8.5	Resultados da simulação do primeiro modelo <i>five-spot</i> após 3.600 segundos	111
8.6	Gráficos da simulação do primeiro modelo <i>five-spot</i>	111
8.7	Resultados da simulação do segundo modelo <i>five-spot</i> após 460.800 segundos	112
8.8	Gráficos da simulação do segundo modelo <i>five-spot</i>	113
8.9	Modelo 1 de injeção térmica em reservatório	114
8.10	Modelo 1 de injeção térmica em reservatório após 4.000 segundos	115
8.11	Graficos mostrando a variação de temperatura na região próxima ao poço .	115
8.12	Tempo inicial da simulação. Na esquerda, o cinza representa o poço, azul a água e o amarelo, arenito. Na direita, é mostrado as temperaturas	116
8.13	Evolução da simulação. Tempo de 610 segundos	117
8.14	Tempo final de 7.180 segundos	118
8.15	Gráficos do tempo final de 7.180 segundos	118
8.16	Interior de um <i>notebook</i> , apresentando o <i>heatpipe</i> , que é a barra de cobre que cruza a GPU e CPU, e resfria na ventoinha	119
8.17	Simulação do sistema de resfriamento do notebook após chegar ao período permanente	120
8.18	Interior de um <i>notebook</i> , apresentando o <i>heatpipe</i> , que é a barra de cobre que cruza a GPU e CPU, e resfria na ventoinha	120
9.1	Logo e documentação do <i>software</i>	122
9.2	Código fonte da classe CSimuladorTemperatura, no Doxygen	122
10.1	Imagen da Interface Gráfica	124
10.2	Como adicionar um material no simulador. Primeiro seleciona Arquivo, Import material. Uma janela será aberta, para o usuário escolher o material.	126
11.1	Principais disciplinas do curso relacionadas a Transferência de Calor	133

List of Tables

2.1	Características básicas do programa	6
2.2	Exemplo de caso de uso	9
8.1	Tabela com as propriedades termofísicas do modelo de validação	107
8.2	Tabela com propriedades termofísicas [Dong, McCartney e Lu 2015]	108
8.3	Tabela com propriedades termofísicas do arenito com água	108
8.4	Tabela com as propriedades termofísicas do modelo 1 - Arenito	113
8.5	Tabela com as propriedades termofísicas do modelo 1 - Ferro	114
8.6	Tabela com as propriedades termofísicas do modelo do <i>notebook</i> - Cobre . .	119

Chapter 1

Introdução

1.1 Escopo do problema

Transferência de calor é uma das principais áreas da física¹ e da engenharia. Seu comportamento foi especulado desde os primeiros filósofos, Aristóteles formula a teoria da matéria ser constituída por quatro elementos: o ar, a água, a terra e o fogo. Os estudos de transferência de calor tiveram grande ascensão no período de 1600 e 1800, sendo, posteriormente, dominados. Estes estudos contaram com a contribuição científica de diversos grandes nomes da física, como Newton e Fourier, [FOURIER 1822].

Qualquer definição de engenharia irá falar da transformação dos materiais usando energia.

"Aplicação da ciência e matemática através da qual as propriedades da matéria e as fontes de energia são tornadas úteis às pessoas (Dic. Merriam-Webster, 2001)."

É por isso que todos os cursos de engenharia incluem, no seu ciclo básico um conjunto de disciplinas de matemática, física e química que possibilitam o entendimento dos problemas associados a temperatura, o efeito da temperatura nos materiais e nos fenômenos estudados. Entre as suas disciplinas obrigatórias, podemos citar as disciplinas de materiais, física, termodinâmica², transferência de calor, algoritmos/programação e cálculo numérico. São disciplinas necessárias para garantir os conhecimentos mínimos para lidar com o desenvolvimento de novos produtos.

Muitos destes conceitos são apresentados para casos simplificados, unidimensionais e para funções bem comportadas. Isto permite ao aluno encontrar as soluções, plotar gráficos e analisar o comportamento dos sistemas em estudo.

Na prática, soluções analíticas para a condução de calor em regime transiente são obtidas para casos unidimensionais, com condições de contorno e iniciais bem definidas,

¹Disponível em <<http://lattes.cnpq.br/documents/11871/24930/TabeladeAreasdoConhecimento.pdf/d192ff6b-3e0a-4074-a74d-c280521bd5f7>>

²opcional em alguns cursos de engenharia

e com propriedades dos materiais constantes.

Para resolver problemas com objetos em duas ou três dimensões com geometria complexa e em regime transiente é necessário, na maior parte dos casos, a utilização da modelagem numérica computacional. A mesma demanda conhecimentos associados ao cálculo numérico, algoritmos, modelagem computacional e desenvolvimento de software. Por isso os cursos de engenharia incluem também disciplinas vinculadas às ciências da computação.

Na indústria do petróleo os conhecimentos dos fenômenos da transferência de calor são fundamentais, seja na etapa de exploração ou na etapa de produção.

Na exploração é buscado um óleo maturado sob temperaturas entre 65°C e 165°C, pois acima de 180°C, é propiciado a formação de gases leves, e acima de 210°C, a formação de grafite, [THOMAS 2004]. Essas temperaturas são obtidas devido ao soterramento da matéria orgânica, e pela proximidade com o manto terrestre.

Na produção, são utilizados trocadores de calor nas plataformas para auxiliar nas separações e no resfriamento do óleo para armazenamento.

Na engenharia de reservatórios, é utilizado vapor de água para aquecer o petróleo no reservatório como método de recuperação avançada [Rosa, Carvalho e Xavier 2006], com o objetivo de diminuir a viscosidade do óleo e facilitar seu escoamento.

Desta forma, torna-se clara a importância do estudo da condução ou difusão de calor em regime transiente, em objetos 3D com geometria complexa, e constituído por diversos materiais com propriedades físicas variáveis.

Conforme a complexidade do problema for evoluindo, os cálculos exigem cada vez mais poder de processamento dos computadores, os quais, atualmente, possuem diversos núcleos lógicos capazes de resolverem cálculos independentemente, acelerando as simulações. Para permitir as divisões das tarefas para os processadores, é utilizada uma linguagem de programação com capacidade de programação paralela ou concorrente.

Neste trabalho utilizaremos métodos de interpolação, solução de matrizes. Métodos iterativos para aproximar as soluções das equações matriciais, tornando desnecessário os processos de inversão de matriz e multiplicação de matriz, etapas normais para o método BTCS (*Backward Time, Centered Space*). Em especial no simulador desenvolvido, como é permitido objetos com formas genéricas, seria impossível resolver o problema por inversão de matriz, pois grandes regiões da matriz teriam valores nulos, consequentemente, seus determinantes seriam zero e impossível de inverter.

Disciplina optativa Programação Paralela e Concorrente I, divisão dos cálculos da modelagem em operações que podem ser resolvidas independentemente e separadamente, e direcionar cada operação para um núcleo do processador, este processo é chamado de *multithreading*, e que acelera em muitas vezes a velocidade das simulações.

Por fim, o ensino de engenharia pode ser aperfeiçoado com a utilização de *softwares* livres, que abordem álgebra, modelagem numérica, programação orientada ao objeto, além do problema físico em si.

1.2 Objetivos

O objetivo deste projeto de engenharia é desenvolver um programa educacional que simula o processo de condução tridimensional de calor, no regime transiente, utilizando métodos numéricos, programação orientada ao objeto com a linguagem C++, mecanismos de paralelismos e *multithreading*, além de renderização 3D.

A finalidade deste projeto de engenharia é desenvolver um *software* capaz de resolver problemas de condução de calor em objetos 3D, constituídos por qualquer materiais com condutividade térmica dependente da temperatura. O software terá interface de usuário amigável e renderização 3D, permitindo a visualização do problema.

Os principais tópicos envolvidos neste projeto são o desenvolvimento de:

- Banco de dados de propriedades térmicas: A partir da literatura, definir padrões para as curvas de propriedades térmicas em função da temperatura e então montar uma estrutura de diretórios com propriedades de materiais conhecidos, normalmente utilizados em engenharia de petróleo. As propriedades térmicas poderão ser obtidas em laboratório, adicionadas ao *software* e utilizados para simulação. As propriedades podem ser calculadas por métodos de correlação ou por interpolação linear.
- Transferência de calor: entender as equações físicas, como o calor é propagado em objetos com diversos formas e composto de diferentes materiais, com propriedades termofísicas dependentes da temperatura.
- Modelagem numérica: solução da equação diferencial da condução ou difusão de calor por meio de diferenças finitas, com o método implícito BTCS, e com condições de contorno de Neumann que podem ser aplicadas em todo o sistema, permitindo geometrias 3D complexas. Além disso, devido à complexidade de encontrar uma solução utilizando sistemas matriciais, é utilizado um método iterativo para obter o resultado aproximado do problema. Por meio da utilização e combinação dessas ferramentas, é possível resolver qualquer problema de condução de calor, especialmente a solução simultânea de casos isolados, e objetos com grandes regiões vazias.
- Programação em C++: por meio da orientação o objeto em C++, o problema pode ser dividido em classes, paradigma este que melhora o controle e organização das etapas de desenvolvimento do software, além de facilitar futuras adaptações e incrementos no simulador. Além disso, essa linguagem possui bibliotecas que auxiliam a utilização de paralelismos e *multithreading*.
- Interface do usuário: com integração total ao simulador, a interface gráfica permite que o usuário tenha liberdade para modificar as propriedades da simulação. E com a implementação de renderização 3D, é possível visualizar o objeto por diferentes ângulos.

1.3 Metodologia utilizada

O software a ser desenvolvido utiliza a metodologia de engenharia de software apresentada pelo Prof. André Bueno na disciplina de programação e ilustrado na Figura 1.1. Note que o “Ciclo de Concepção e Análise” é composto por diversas partes representadas neste trabalho em diferentes capítulos. Os “Ciclos de Planejamento/Detalhamento” e “Ciclo Construção”, envolvem a construção das diferentes versões do software e serão brevemente descritos no capítulo de Projeto.

Esta metodologia é utilizada nas disciplinas:

- LEP01447 : Programação Orientada a Objeto em C++
- LEP01446 : Programação Prática

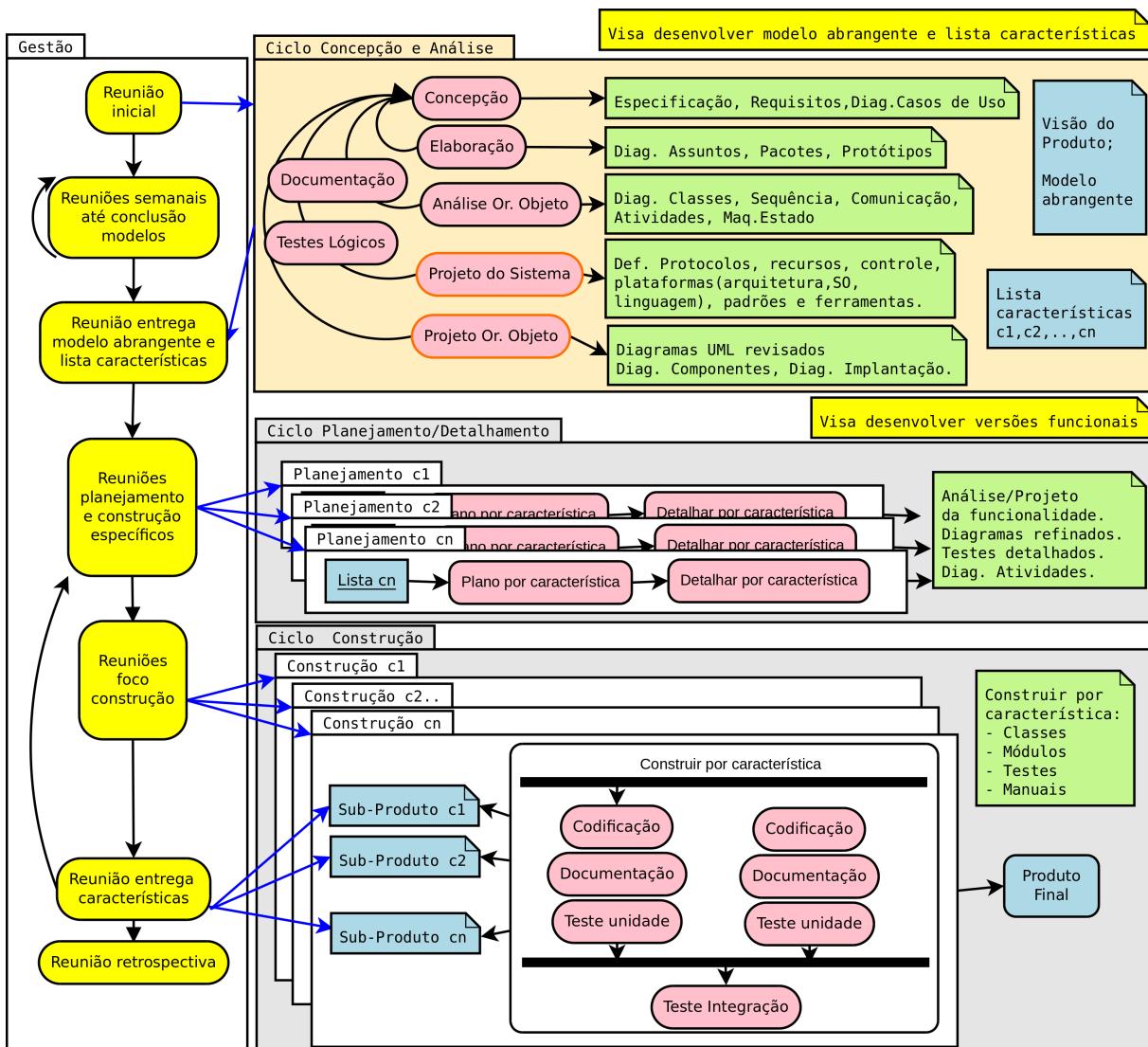


Figure 1.1: Etapas para o desenvolvimento do software - *projeto de engenharia*

Chapter 2

Concepção

Apresenta-se neste capítulo do projeto de engenharia a concepção, a especificação, requisitos e casos de uso do sistema a ser modelado e desenvolvido.

2.1 Características gerais

O software denominado *Simulador de Difusão Térmica 3D (SDT-3D)*, será desenvolvido utilizando o paradigma da orientação a objeto usando a linguagem de alto desempenho C++, o mesmo será capaz de simular a condução de calor em objetos tridimensionais, com formas e superfícies complexas definidas pelo usuário. O usuário pode definir quais materiais constituem o objeto, e qual o método utilizado para calcular as propriedades termofísicas para cada material. Também é permitido a renderização 3D do objeto, o uso de processamento paralelo, e os resultados podem ser salvos em arquivo no formato pdf.

A equação diferencial de condução de calor é modelada por diferenças finitas, especificamente pelo método BTCS (*Backward Time Centered Space*), um método implícito e incondicionalmente estável. As condições iniciais são inseridas pelo usuário, e as condições de contorno externas são definidas por regiões que não trocam calor com o meio externo, isto é, fronteiras adiabáticas.

Table 2.1: Características básicas do programa

Nome	Simulador de difusão térmica 3D - <i>SDT-3D</i>
Componentes principais	Banco de dados com métodos de correlação e interpolação para propriedades termofísicas. Método numérico implícito BTCS. Interface gráfica para desenho da simulação. Renderização 3D, processamento paralelo e saída gráfica e pdf.
Missão	Simulador de transferência de calor em objetos 3D com superfícies complexas, formado por materiais com propriedades dependentes da temperatura. Visando auxiliar no ensino das diversas disciplinas abrangidas por este trabalho, como cálculo numérico, modelagem computacional, programação orientada a objeto em C++, física e matemática.

2.2 Especificação

Deseja-se desenvolver um *software* com interface gráfica amigável ao usuário, onde seja possível desenhar objetos tridimensionais, por meio de perfis 2D. Para cada objeto a ser incluído na simulação o usuário escolhe, a partir de um banco de dados, as propriedades do material.

A simulação é governada pela equação da difusão térmica, a qual é modelada por diferenças finitas, pelo método BTCS, com fronteiras adiabáticas.

Na dinâmica de desenho dos objetos a serem simulados, o usuário deverá escolher:

- o tipo de material e suas propriedades termofísicas a partir de um banco de dados.
- a temperatura inicial do objeto.
- para cada plano 2D a posição do objeto e suas dimensões
- cada camada do objeto poderá ser construída a partir de um banco de objetos geométricos simples, incluindo círculo, elipse, retângulo, quadrado.
-

O usuário terá a liberdade para utilizar um dentre três métodos para obter as propriedades dos materiais: propriedades constantes, correlação e interpolação.

Após os desenhos pelo usuário do sistema a ser simulado é necessário definir:

- um ponto de monitoramento da temperatura.

- tipo de condição de contorno?
- valor de erro aceitável

Finalmente é disparada a simulação em si. A cada etapa da simulação o simulador irá calcular a temperatura em cada ponto, e só irá passar para o próximo passo de tempo se o erro entre iterações for menor que o valor aceitável definido pelo usuário. Finalizado um passo de tempo, a janela gráfica que tem a solução, distribuição de temperaturas, será atualizada. O simulador também irá plotar gráficos com os novos valores calculados (eixo x e y para o ponto de monitoramento selecionado pelo usuário).

Outras características desejadas incluem:

- possibilidade de alterar, a qualquer momento, o desenho do problema, inserindo novos objetos.
- possibilidade de alterar, a qualquer momento, o ponto de monitoramento.

2.3 Requisitos

Apresenta-se a seguir os requisitos funcionais e não funcionais.

2.3.1 Requisitos funcionais

Apresenta-se a seguir os requisitos funcionais.

RF-01	O programa deve ter uma interface gráfica amigável.
--------------	---

RF-02	O usuário tem a liberdade de desenhar qualquer objeto 3D.
--------------	---

RF-03	O usuário tem a liberdade de desenhar qualquer escolher a temperatura em cada ponto.
--------------	--

RF-04	O usuário tem a liberdade de escolher o material em cada ponto do objeto, juntamente com o método para obter a condutividade térmica.
--------------	---

RF-05	Novos materiais podem ser adicionados no banco de dados pelo usuário
--------------	--

RF-06	A condutividade térmica pode ser calculados por correlação ou interpolação linear.
--------------	--

RF-07	O usuário poderá escolher um ponto de estudo, cuja temperatura será monitorada ao longo do tempo, juntamente com todas as linhas cardeais partindo desse ponto.
--------------	---

RF-08	O ponto de estudo poderá ser alterado durante a execução da simulação
RF-09	O usuário poderá escolher uma região de fonte ou sumidouro.
RF-10	O usuário poderá salvar e/ou carregar dados da simulação.
RF-11	O usuário poderá salvar os resultados da simulação em um arquivo pdf.
RF-12	O usuário poderá comparar as propriedades termofísicas dos materiais.
RF-12	O usuário poderá acompanhar a evolução da temperatura em uma superfície 2D em todo intervalo de tempo.
RF-13	O usuário poderá visualizar o objeto 3D desenhado em uma janela específica.
RF-14	O usuário pode definir as propriedades físicas da simulação, como intervalo de tempo e espaço.
RF-15	O desenho dos objetos será feito em camadas 2D a partir de geometrias bidimensionais usuais como circulo, quadrado, retângulo, elipse.

2.3.2 Requisitos não funcionais

RNF-01	Os cálculos devem ser feitos utilizando-se o método numérico de diferenças finitas BTCS.
RNF-02	O programa deverá ser multi-plataforma, podendo ser executado em <i>Windows</i> , <i>GNU/Linux</i> ou <i>Mac</i> .
RNF-03	A performance do programa pode ser alterada com a mudança do modelo de paralelismo.
RNF-04	A linguagem a ser utilizada é C++. ¹
RNF-05	A interface gráfica deve ser desenvolvida pela biblioteca multiplataforma Qt (link para site).
RNF-05	Os gráficos devem ser gerados usando a biblioteca QCustom Plot ² .

RNF-06	O usuário poderá comparar os valores das propriedades termofísicas em função da temperatura de qualquer material por meio de um gráfico na interface de usuário.
---------------	--

2.4 Casos de uso

Nesta seção iremos mostrar alguns casos de uso do software a ser desenvolvido.

2.4.1 Diagrama de caso de uso geral

O diagrama de caso de uso geral da Figura 2.1 mostra o usuário desenhando um objeto com material padrão do simulador, escolhendo um ponto de estudo, executando a simulação, analisando os resultados e salvando o objeto e os resultados em pdf.

Table 2.2: Exemplo de caso de uso

Nome do caso de uso:	Simulação da distribuição da temperatura
Resumo/descrição:	Cálculo da distribuição de temperatura em determinadas condições.
Etapas:	<ol style="list-style-type: none"> 1. Escolher o material e sua temperatura inicial 2. Desenhar o objeto; 3. Escolher um ponto de estudo; 4. Executar a simulação; 5. Analisar resultados; 6. Salvar resultados em pdf.
Cenários alternativos:	Um cenário alternativo envolve uma entrada de propriedades de um metal obtidas em laboratório, escolher se essas propriedades vão ser calculadas por correlação ou interpolação.

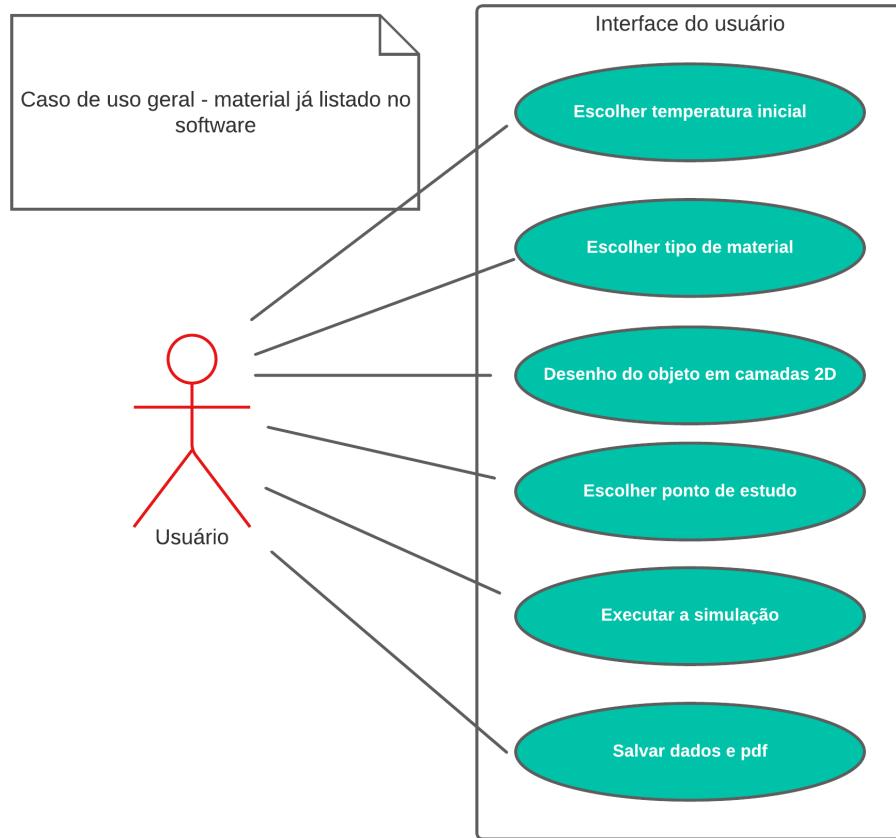


Figure 2.1: Diagrama de caso de uso geral

2.4.2 Diagrama de caso de uso específico

O caso de uso específico da Figura 2.2 mostra um cenário onde o usuário quer utilizar os valores da condutividade térmica obtidos em laboratório. Ele deve montar um arquivo .txt com esses valores (a forma de criar esse arquivo é descrito no Apêndice B), e carregar no simulador (RF-05).

O usuário terá a liberdade de comparar seu material com outros padrões do simulador, e escolhê-lo para o desenho do objeto.

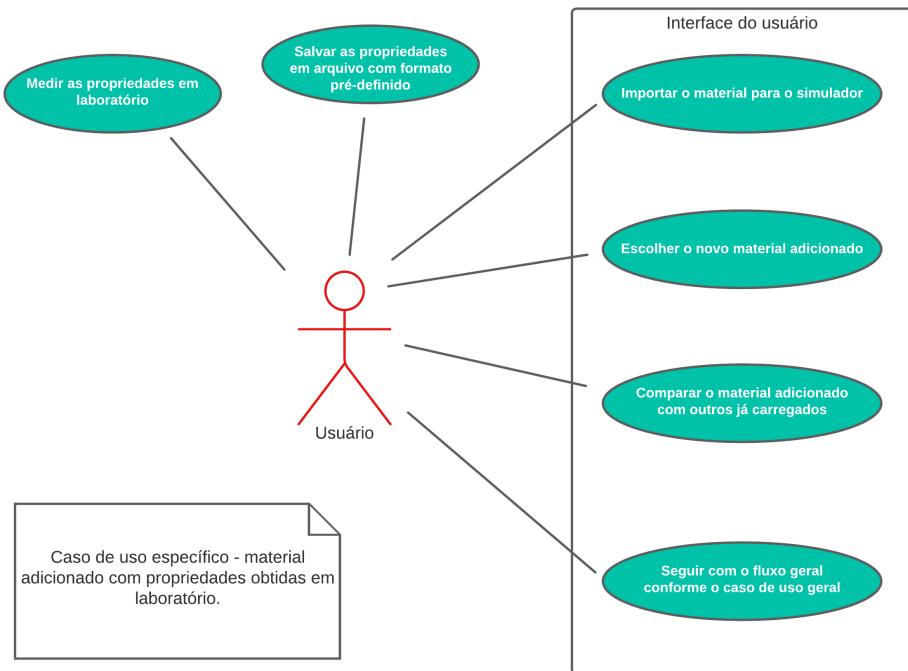


Figure 2.2: Diagrama de caso de uso específico: adição de novo material e interpolação

Chapter 3

Elaboração

Neste capítulo é apresentada a elaboração do programa, constituído pelo desenvolvimento teórico, modelagem numérica, identificação de pacotes e algoritmos adicionais relacionados ao *software* desenvolvido.

3.1 Análise de domínio

A análise de domínio, como parte da elaboração, tem o objetivo de entender e delimitar conceitos fundamentais, sob os quais o *software* é construído [BUENO 2003].

O presente trabalho envolve quatro conceitos fundamentais:

1. Transferência de calor:

Transferência de calor é uma das áreas clássicas de fenomenologia da física. É responsável por tratar das três formas possíveis de transferência de calor: condução, convecção e radiação. Este projeto trata especificamente da condução de calor.

A condução só pode ocorrer em meio material (fluidos ou sólidos), e sem que haja movimento do próprio meio, característica da convecção ([NUSSENZVEIG 2014]).

2. Modelagem numérica:

Métodos numéricos são algoritmos desenvolvidos com ajuda da matemática para resolver problemas complexos da natureza. São utilizados quando uma solução analítica é difícil de ser obtida, ou com condições de contorno complexas.

3. Programação:

O paradigma orientado ao objeto é um dos principais paradigmas da programação, utilizado especialmente na construção de grandes *softwares* devido à portabilidade, organização e delimitação de assuntos. C++ é uma das linguagens mais utilizadas atualmente, por ser mais rápida com muito suporte, por permitir a orientação ao objeto e disponibilizar muitas bibliotecas prontas (como Qt, QCustomPlot).

Os computadores atuais disponibilizam diversos núcleos de processamento, possibilitando o uso de recursos de programação paralela e ou concorrente. C++ fornece a abstração de *threads* que possibilitam, de forma simplificada, o uso de processamento paralelo.

4. Renderização 3D:

Renderização 3D é uma área com grande ascensão na indústria de jogos e *softwares* de engenharia profissional, torna prático que usuários consigam visualizar o objeto sob qualquer ótica. É necessário a utilização de vários conceitos da álgebra linear.

3.2 Formulação modelos teóricos

Apresenta-se a seguir os termos e unidades utilizados, a formulação teórica, condições de contorno, demonstrações e considerações sobre condutividade térmica variável.

3.2.1 Termos e Unidades

Os principais termos e suas unidades utilizadas neste projeto estão listadas abaixo:

- Dados relativos ao material:

- c_p - calor específico a pressão constante [$J/kg \cdot K$];
- k - condutividade térmica [$W/m \cdot K$];
- ρ - massa específica [kg/m^3].

- Dados relativos ao objeto

- $\Delta x, \Delta y$ - distância entre os centros dos blocos, valor inicial: $1px=0.0026\text{ m}$ [m];
- Δz - distância entre perfis, valor inicial: 0.05 m [m];
- T - temperatura no nodo [K];

- Variáveis usadas na simulação:

- i - posição do nodo em relação ao eixo x;
- k - posição do nodo em relação ao eixo y;
- g - qual *grid*/perfil está sendo analisado;
- t - tempo atual;
- n - índice do passo de tempo;
- ν - número da iteração.

3.2.2 Formulação teórica

A taxa de transferência de calor foi modelado empiricamente por Jean B. J. Fourier em 1822 ([FOURIER 1822]). Posteriormente a teoria foi aprimorada até chegar na equação geral da difusão de calor Eq. (3.1). O desenvolvimento teórico para chegar nesta equação, pode ser acompanhado detalhadamente no [Incropora 2008].

Portanto, a seguir é apresentada a equação geral da difusão de calor em meios tridimensionais em coordenadas cartesianas cartesianas:

$$\frac{\partial}{\partial x} \left(k \frac{\partial T}{\partial x} \right) + \frac{\partial}{\partial y} \left(k \frac{\partial T}{\partial y} \right) + \frac{\partial}{\partial z} \left(k \frac{\partial T}{\partial z} \right) = \rho c_p \frac{\partial T}{\partial t} \quad (3.1)$$

Para resolver a equação geral da difusão térmica, será utilizado o método implícito de diferenças finitas BTCS, com malha em formato bloco centrado.

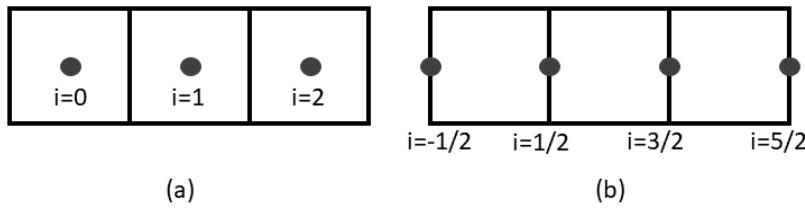


Figure 3.1: Tipos de malha, (a) bloco-centrado e (b) ponto-distribuído

Conforme a Figura 3.1, existem dois tipos principais de malha: bloco-centrado, onde os pontos analisados estão nos centros de cada bloco, e ponto-distribuído, onde os pontos analisados estão nas fronteiras de cada bloco.

Com esses conceitos em mente, a equação geral é modelada por diferenças finitas, mantendo a condutividade térmica dentro da derivada espacial. Inicialmente, será modelado somente a derivada externa:

$$\frac{\partial}{\partial x} \left(k \frac{\partial T}{\partial x} \right) = \frac{(k \frac{\partial T}{\partial x})_{i-\frac{1}{2},j,k} - (k \frac{\partial T}{\partial x})_{i+\frac{1}{2},j,k}}{\Delta x} \quad (3.2)$$

Modelando as derivadas internas:

$$\frac{\partial}{\partial x} \left(k \frac{\partial T}{\partial x} \right) = \frac{k_{i-\frac{1}{2},j,k} \left(\frac{T_{i-1,j,k} - T_{i,j,k}}{\Delta x} \right) - k_{i+\frac{1}{2},j,k} \left(\frac{T_{i,j,k} - T_{i+1,j,k}}{\Delta x} \right)}{\Delta x} \quad (3.3)$$

Com um pouco de álgebra:

$$\frac{\partial}{\partial x} \left(k \frac{\partial T}{\partial x} \right) = \frac{k_{i-\frac{1}{2},j,k} (T_{i-1,j,k} - T_{i,j,k}) - k_{i+\frac{1}{2},j,k} (T_{i,j,k} - T_{i+1,j,k})}{\Delta x^2} \quad (3.4)$$

Chegando na modelagem final para a derivada espacial ao longo do x:

$$\frac{\partial}{\partial x} \left(k \frac{\partial T}{\partial x} \right) = \frac{k_{i-\frac{1}{2},j,k} T_{i-1,j,k} - \left(k_{i-\frac{1}{2},j,k} + k_{i+\frac{1}{2},j,k} \right) T_{i,j,k} + k_{i+\frac{1}{2},j,k} T_{i+1,j,k}}{\Delta x^2} \quad (3.5)$$

Como as outras dimensões são simétricas:

$$\frac{\partial}{\partial y} \left(k \frac{\partial T}{\partial y} \right) = \frac{k_{i,j-\frac{1}{2},k} T_{i,j-1,k} - \left(k_{i,j-\frac{1}{2},k} + k_{i,j+\frac{1}{2},k} \right) T_{i,j,k} + k_{i,j+\frac{1}{2},k} T_{i,j+1,k}}{\Delta y^2} \quad (3.6)$$

$$\frac{\partial}{\partial z} \left(k \frac{\partial T}{\partial z} \right) = \frac{k_{i,j,k-\frac{1}{2}} T_{i,j,k-1} - \left(k_{i,j,k-\frac{1}{2}} + k_{i,j,k+\frac{1}{2}} \right) T_{i,j,k} + k_{i,j,k+\frac{1}{2}} T_{i,j,k+1}}{\Delta z^2} \quad (3.7)$$

A derivada temporal é atrasada no tempo:

$$\frac{\partial T}{\partial t} = \frac{T_{i,j,k}^n - T_{i,j,k}^{n-1}}{\Delta t} \quad (3.8)$$

Substituindo as diferenças finitas na equação geral:

$$\begin{aligned} & \frac{k_{i-\frac{1}{2},j,k} T_{i-1,j,k} - \left(k_{i-\frac{1}{2},j,k} + k_{i+\frac{1}{2},j,k} \right) T_{i,j,k} + k_{i+\frac{1}{2},j,k} T_{i+1,j,k}}{\Delta x^2} + \\ & \frac{k_{i,j-\frac{1}{2},k} T_{i,j-1,k} - \left(k_{i,j-\frac{1}{2},k} + k_{i,j+\frac{1}{2},k} \right) T_{i,j,k} + k_{i,j+\frac{1}{2},k} T_{i,j+1,k}}{\Delta y^2} + \\ & \frac{k_{i,j,k-\frac{1}{2}} T_{i,j,k-1} - \left(k_{i,j,k-\frac{1}{2}} + k_{i,j,k+\frac{1}{2}} \right) T_{i,j,k} + k_{i,j,k+\frac{1}{2}} T_{i,j,k+1}}{\Delta z^2} = \\ & \frac{T_{i,j,k}^n - T_{i,j,k}^{n-1}}{\Delta t} \end{aligned} \quad (3.9)$$

Onde a malha é homogênea na superfície, mas não entre os perfis, ou seja, $\Delta x = \Delta y \neq \Delta z$. Substituindo:

$$\begin{aligned} & \frac{k_{i-\frac{1}{2},j,k} T_{i-1,j,k} - \left(k_{i-\frac{1}{2},j,k} + k_{i+\frac{1}{2},j,k} \right) T_{i,j,k} + k_{i+\frac{1}{2},j,k} T_{i+1,j,k}}{\Delta x^2} + \\ & \frac{k_{i,j-\frac{1}{2},k} T_{i,j-1,k} - \left(k_{i,j-\frac{1}{2},k} + k_{i,j+\frac{1}{2},k} \right) T_{i,j,k} + k_{i,j+\frac{1}{2},k} T_{i,j+1,k}}{\Delta y^2} + \\ & \frac{k_{i,j,k-\frac{1}{2}} T_{i,j,k-1} - \left(k_{i,j,k-\frac{1}{2}} + k_{i,j,k+\frac{1}{2}} \right) T_{i,j,k} + k_{i,j,k+\frac{1}{2}} T_{i,j,k+1}}{\Delta z^2} = \\ & c_p \rho \frac{T_{i,j,k}^n - T_{i,j,k}^{n-1}}{\Delta t} \end{aligned} \quad (3.10)$$

Multiplicando pelo múltiplo comum:

$$\begin{aligned} & \Delta z^2 \left(k_{i-\frac{1}{2},j,k}^{n+1} T_{i-1,j,k}^{n+1} - \left(k_{i-\frac{1}{2},j,k}^{n+1} + k_{i+\frac{1}{2},j,k}^{n+1} \right) T_{i,j,k}^{n+1} + k_{i+\frac{1}{2},j,k}^{n+1} T_{i+1,j,k}^{n+1} \right) + \\ & \Delta z^2 \left(k_{i,j-\frac{1}{2},k}^{n+1} T_{i,j-1,k}^{n+1} - \left(k_{i,j-\frac{1}{2},k}^{n+1} + k_{i,j+\frac{1}{2},k}^{n+1} \right) T_{i,j,k}^{n+1} + k_{i,j+\frac{1}{2},k}^{n+1} T_{i,j+1,k}^{n+1} \right) + \\ & \Delta x^2 \left(k_{i,j,k-\frac{1}{2}}^{n+1} T_{i,j,k-1}^{n+1} - \left(k_{i,j,k-\frac{1}{2}}^{n+1} + k_{i,j,k+\frac{1}{2}}^{n+1} \right) T_{i,j,k}^{n+1} + k_{i,j,k+\frac{1}{2}}^{n+1} T_{i,j,k+1}^{n+1} \right) = \\ & \frac{\Delta z^2 \Delta x^2 c_p \rho}{\Delta t} T_{i,j,k}^n - \frac{\Delta z^2 \Delta x^2 c_p \rho}{\Delta t} T_{i,j,k}^{n-1} \end{aligned} \quad (3.11)$$

Como a equação acima é complexa para ser reorganizada e resolvida por equações matriciais, será utilizado aproximações para resolver esse problema, ideia similar ao Método do Ponto Fixo (MPF). As condições de parada são: diferença entre iterações menor que $0,5^\circ\text{C}$, máximo de iterações igual a 1.000, e mínimo de 800 iterações. Para resolver o sistema de equações, será isolado uma das temperaturas para calcular a iteração $\nu + 1$:

$$\begin{aligned} T_{i,j}^{\nu+1} = & C_1 \frac{\Delta z^2 \Delta x^2 c_p \rho}{\Delta t} T_{i,j}^{n-1} + \\ & C_1 \Delta z^2 \left(k_{i-\frac{1}{2},j,k}^n T_{i-1,j,k}^{\nu} + k_{i+\frac{1}{2},j,k}^n T_{i+1,j,k}^{\nu} \right) + \\ & C_1 \Delta z^2 \left(k_{i,j-\frac{1}{2},k}^n T_{i,j-1,k}^{\nu} + k_{i,j+\frac{1}{2},k}^n T_{i,j+1,k}^{\nu} \right) + \\ & C_1 \Delta x^2 \left(k_{i,j,k-\frac{1}{2}}^n T_{i,j,k-1}^{\nu} + k_{i,j,k+\frac{1}{2}}^n T_{i,j,k+1}^{\nu} \right) \end{aligned} \quad (3.12)$$

Onde C_1 é definido por:

$$\frac{1}{C_1} = \frac{\Delta z^2 \Delta x^2 c_p \rho}{\Delta t} + \frac{\Delta z^2 \left(k_{i-\frac{1}{2},j,k}^n + k_{i+\frac{1}{2},j,k}^n \right)}{\Delta z^2 \left(k_{i,j-\frac{1}{2},k}^n + k_{i,j+\frac{1}{2},k}^n \right)} + \Delta x^2 \left(k_{i,j,k-\frac{1}{2}}^n + k_{i,j,k+\frac{1}{2}}^n \right) \quad (3.13)$$

Agora, é necessário definir o cálculo das condutividades térmicas nas fronteiras. Para solucionar esse problema, pode-se resolver pela taxa de condução ([RESNICK 2009]), esse modelo de solução é similar ao cálculo da permeabilidade de rochas em série [Rosa, Carvalho e Xavier 2014]. Também pode-se utilizar da analogia com a resistência elétrica ([NUSSENZVEIG 2014]), ([Incropera 2008]). Para o presente trabalho, será demonstrado a condutividade térmica equivalente pela taxa de condução térmica:

$$q_x = -kA \frac{dT}{dx} = -\frac{kA}{L} \Delta T \quad (3.14)$$

Onde q_x é a taxa de condução térmica.

Isolando a diferença de temperatura:

$$\Delta T = -\frac{L q_x}{k A} \quad (3.15)$$

A Figura 3.2 mostra o caso de condutividades térmicas em série. A diferença de temperatura entre a esquerda (0) e a direita (2), é a soma das diferenças nesse meio, ou seja:

$$T_0 - T_2 = (T_0 - T_1) + (T_1 - T_2) \quad (3.16)$$

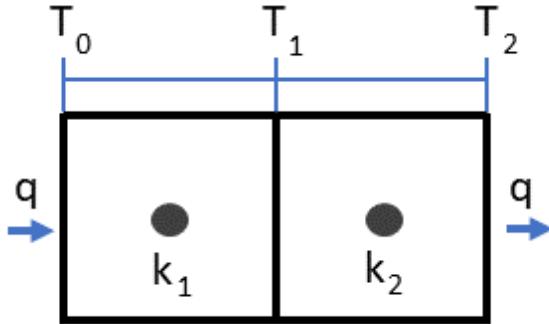


Figure 3.2: Representação de condutividade térmica em série

Logo,

$$\Delta T_t = \Delta T_1 + \Delta T_2 \quad (3.17)$$

Como não há fontes de calor, a taxa de calor (q) que entra no sistema, é igual ao que sai. O comprimento total do sistema é $2L$, então:

$$-\frac{2Lq}{k_r A} = -\frac{Lq}{k_1 A} - \frac{Lq}{k_2 A} \quad (3.18)$$

Com alguns ajustes algébricos:

$$\frac{2}{k_r} = \frac{1}{k_1} + \frac{1}{k_2} \quad (3.19)$$

Ou, simplesmente:

$$k_r = \frac{2k_1 k_2}{k_1 + k_2} \quad (3.20)$$

A Equação 3.20 mostra a condutividade térmica equivalente em série, para materiais com mesmo comprimento. Essa solução, com a condição de comprimento igual, é a mesma apresentada nas referências anteriores.

É importante analisar a célula computacional, ou a região que é observada quando a temperatura é calculada em um ponto específico. Para isso, é apresentada a Figura 3.3, onde a esquerda é o tempo anterior $t=n-1$, e o ponto calculado está no tempo presente $t=n$. Para calcular a temperatura no ponto vermelho, é utilizado o mesmo ponto, mas no tempo anterior, e uma célula em cada sentido.

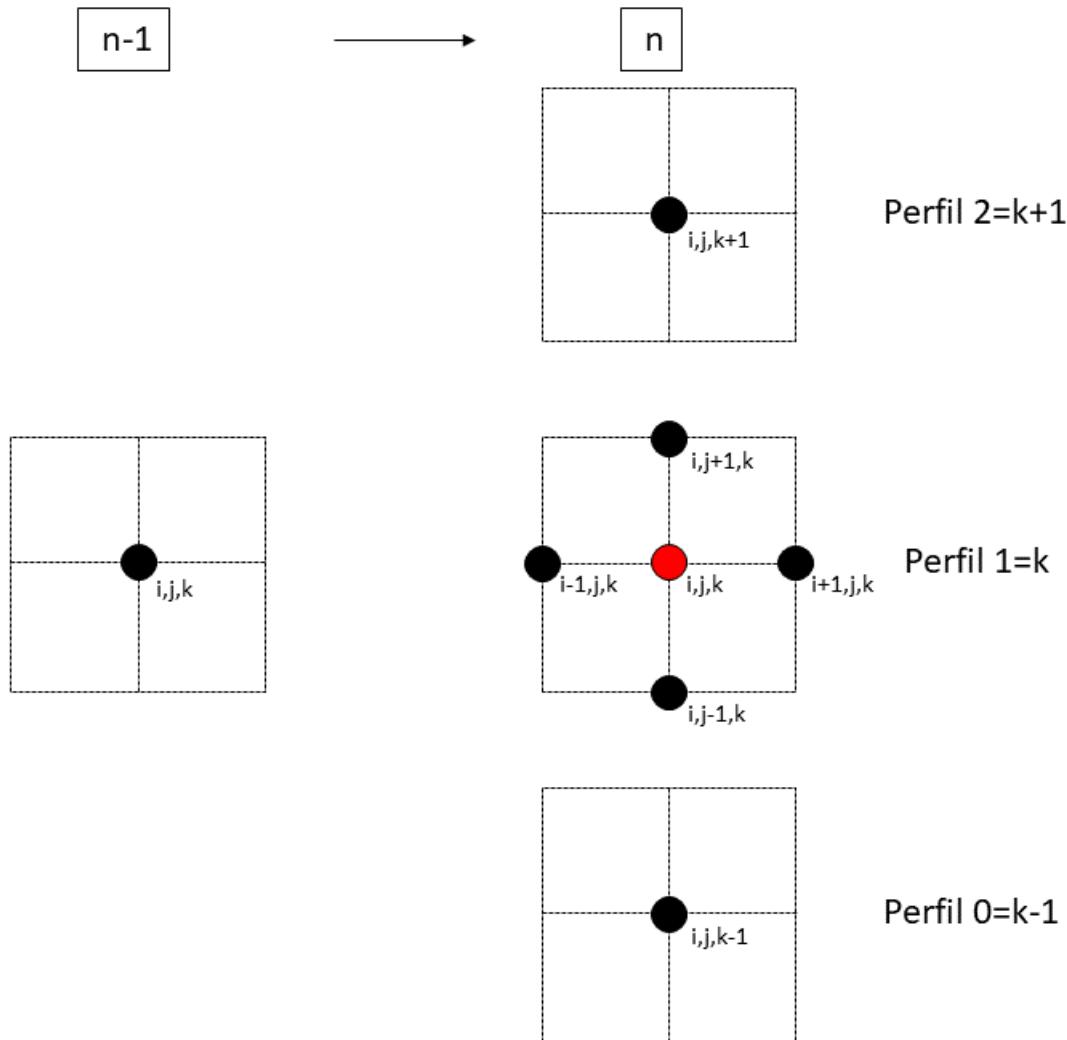


Figure 3.3: Malha utilizada para calcular a temperatura de um ponto, onde cada ponto é o centro dos blocos

A seguir, é resolvida a última etapa da modelagem do problema, a modelagem da condição de fronteira de Neumann.

3.2.3 Condição de fronteira

Condição de fronteira, como o próprio nome diz, é a condição onde estão os limites materiais do objeto. Nessa região, a condução térmica é diferente do interior do objeto, pois não poderá conduzir calor em todos os sentidos, mas só onde existir material adjacente.

A condição de contorno de Neumann define a taxa de troca de calor com o meio externo, no trabalho desenvolvido essa taxa será sempre nula, ou seja, a região estudada não troca calor com o meio externo. Na Figura 3.4, a fronteira está na reta vermelha e, como o método modelado utilizaria o ponto à esquerda, é necessário encontrar um substituto real para esse termo.

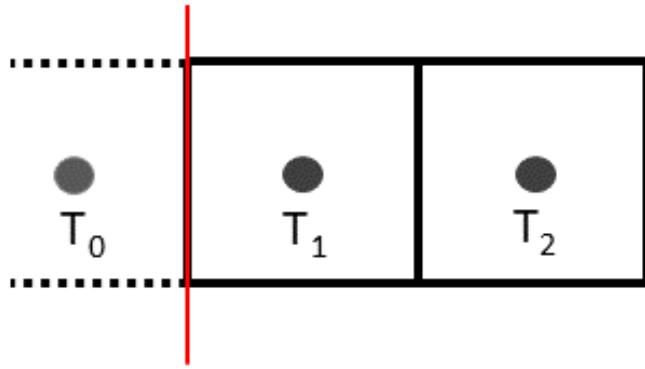


Figure 3.4: Análise da fronteira de Neumann

Por isso, é importante modelar a condição de contorno, que pode ser modelada com diferenças finitas centradas como:

$$k \frac{\partial T}{\partial x}_{i-\frac{1}{2},j,k} = \frac{T_{i,j,k}^n - T_{i-1,j,k}^n}{\Delta x} = 0 \quad (3.21)$$

A equação acima possui duas soluções:

$$\begin{cases} k_{i-\frac{1}{2},j,k} = 0 \\ \frac{\partial T}{\partial x}_{i-\frac{1}{2},j,k} = 0 \end{cases} \quad (3.22)$$

Resolvendo a linha de baixo:

$$\frac{\partial T}{\partial x}_{i-\frac{1}{2},j,k} = \frac{T_{i,j,k}^n - T_{i-1,j,k}^n}{\Delta x} = 0 \quad (3.23)$$

$$T_{i-1,j,k}^n = T_{i,j,k}^n \quad (3.24)$$

Todas as seis fronteiras são simétricas, então:

$$\begin{aligned} T_{i-1,j,k}^n &= T_{i,j,k}^n \\ T_{i+1,j,k}^n &= T_{i,j,k}^n \\ T_{i,j-1,k}^n &= T_{i,j,k}^n \\ T_{i,j+1,k}^n &= T_{i,j,k}^n \\ T_{i,j,k-1}^n &= T_{i,j,k}^n \\ T_{i,j,k+1}^n &= T_{i,j,k}^n \end{aligned} \quad (3.25)$$

As equações encontradas na Eq. 3.25 dizem que, se existir uma fronteira, a temperatura inexistente deve ser substituída pela temperatura do próprio ponto. De forma alternativa, como mostrado na Eq. 3.22, a condutividade térmica na fronteira deve ser zero. Quaisquer dentre as duas opções resolvem o problema da condição de contorno de Neumann.

3.2.4 Demonstrações matemáticas

Nesta parte, serão analisados dois casos para validar as modelagens. Primeiro, será utilizado um objeto formado por uma única célula isolada no espaço. Posteriormente, será analisado o caso do objeto constituído por um único material, mas bidimensional.

Começando pelo objeto de única célula, em todas as suas seis fronteiras devem ser aplicadas as condições de contorno de Neumann. Fisicamente, é esperado que o objeto, por estar isolado, não varie sua temperatura interna ao longo do tempo. Então, partindo da equação geral:

Partindo da Eq. (3.12) e, como demonstrado na Eq. 3.22, quando houver fronteira a condutividade térmica na fronteira é zero:

$$\begin{aligned} T_{i,j,k}^{\nu+1} = & C_1 \frac{\Delta z^2 \Delta x^2 c_p \rho}{\Delta t} T_{i,j,k}^{n-1} + \\ & C_1 \Delta z^2 \left(\begin{array}{c} k_{j-\frac{1}{2},j,k}^n T_{i-1,j,k}^n + k_{j+\frac{1}{2},j,k}^n T_{i+1,j,k}^n \\ \nearrow^0 \quad \searrow^0 \\ \end{array} \right) + \\ & C_1 \Delta z^2 \left(\begin{array}{c} k_{i,j-\frac{1}{2},k}^n T_{i,j-1,k}^n + k_{i,j+\frac{1}{2},k}^n T_{i,j+1,k}^n \\ \nearrow^0 \quad \searrow^0 \\ \end{array} \right) + \\ & C_1 \Delta x^2 \left(\begin{array}{c} k_{i,j,k-\frac{1}{2}}^n T_{i,j,k-1}^n + k_{i,j,k+\frac{1}{2}}^n T_{i,j,k+1}^n \\ \nearrow^0 \quad \searrow^0 \\ \end{array} \right) \end{aligned} \quad (3.26)$$

$$\begin{aligned} \frac{1}{C_1} = & \frac{\Delta z^2 \Delta x^2 c_p \rho}{\Delta t} + \Delta z^2 \left(\begin{array}{c} k_{i-\frac{1}{2},j,k}^n + k_{i+\frac{1}{2},j,k}^n \\ \nearrow^0 \quad \searrow^0 \\ \end{array} \right) + \\ & \Delta z^2 \left(\begin{array}{c} k_{j-\frac{1}{2},k}^n + k_{j+\frac{1}{2},k}^n \\ \nearrow^0 \quad \searrow^0 \\ \end{array} \right) + \Delta x^2 \left(\begin{array}{c} k_{i,j,k-\frac{1}{2}}^n + k_{i,j,k+\frac{1}{2}}^n \\ \nearrow^0 \quad \searrow^0 \\ \end{array} \right) \end{aligned} \quad (3.27)$$

Resultando em:

$$T_{i,j,k}^{\nu+1} = C_1 \frac{\Delta z^2 \Delta x^2 c_p \rho}{\Delta t} T_{i,j,k}^{n-1} \quad (3.28)$$

$$\frac{1}{C_1} = \frac{\Delta z^2 \Delta x^2 c_p \rho}{\Delta t} \quad (3.29)$$

Logo:

$$T_{i,j,k}^{\nu+1} = \frac{\Delta t}{\Delta z^2 \Delta x^2 c_p \rho} \frac{\Delta z^2 \Delta x^2 c_p \rho}{\Delta t} T_{i,j,k}^{n-1} \quad (3.30)$$

$$T_{i,j,k}^{\nu+1} = T_{i,j,k}^{\nu-1} \quad (3.31)$$

Mostrando que a temperatura não varia com o tempo.

Para a segunda demonstração, onde o objeto é constituído pelo mesmo material e mesma condutividade térmica, mas somente bidimensional.

O índice k referente à terceira dimensão continuará aparecendo nas equações abaixo para manter a ideia do algoritmo. Como só existe um valor para essa dimensão, pode-se considerar o valor fixo de 0.

Partindo da Eq. (3.12) e substituindo todas as condutividades térmicas nas interfaces por k, e simplificando para bidimensional:

$$\begin{aligned} T_{i,j,k}^n = & \\ C_1 \frac{\Delta z^2 \Delta x^2 c_p \rho}{\Delta t} T_{i,j,k}^{n-1} + & \\ C_1 \Delta z^2 \left(k T_{i-1,j,k}^n + k T_{i+1,j,k}^n \right) + & \\ C_1 \Delta z^2 \left(k T_{i,j-1,k}^n + k T_{i,j+1,k}^n \right) & \end{aligned} \quad (3.32)$$

$$\frac{1}{C_1} = \frac{\Delta z^2 \Delta x^2 c_p \rho}{\Delta t} + \Delta z^2 (k + k) + \Delta z^2 (k + k) \quad (3.33)$$

Com alguns ajustes:

$$\begin{aligned} \frac{1}{C_1} T_{i,j,k}^n = & \\ \frac{\Delta x^2 c_p \rho}{\Delta t} T_{i,j,k}^{n-1} + & \\ k \left(T_{i-1,j,k}^n + T_{i+1,j,k}^n \right) + & \\ k \left(T_{i,j-1,k}^n + T_{i,j+1,k}^n \right) & \end{aligned} \quad (3.34)$$

$$\frac{1}{C_1} = \frac{\Delta x^2 c_p \rho}{\Delta t} + 2k + 2k \quad (3.35)$$

Logo:

$$\begin{aligned} \left(\frac{\Delta x^2 c_p \rho}{k \Delta t} + 2 + 2 \right) T_{i,j,k}^n = & \\ \frac{\Delta x^2 c_p \rho}{k \Delta t} T_{i,j,k}^{n-1} + & \\ \left(T_{i-1,j,k}^n + T_{i+1,j,k}^n \right) + & \\ \left(T_{i,j-1,k}^n + T_{i,j+1,k}^n \right) & \end{aligned} \quad (3.36)$$

Chegando na equação:

$$\begin{aligned} \frac{c_p \rho}{k} \frac{T_{i,j,k}^n - T_{i,j,k}^{n-1}}{\Delta t} = & \\ \frac{T_{i-1,j,k}^n - 2T_{i,j,k}^n + T_{i+1,j,k}^n}{\Delta x^2} + & \\ \frac{T_{i,j-1,k}^n - 2T_{i,j,k}^n + T_{i,j+1,k}^n}{\Delta x^2} & \end{aligned} \quad (3.37)$$

A Eq. (3.37) representa o método implícito BTCS para um sistema homogêneo bidimensional, conforme [Incropora 2008].

3.2.5 Condutividade térmica variável

A condutividade térmica, no programas desenvolvido, pode variar com o espaço, pelo objeto ser constituído por mais de um material, com condutividade térmica distinta.

Contudo, pode também variar com a temperatura e, consequentemente, com o tempo.

Serão fornecidas aos usuários, três opções para calcular essas condutividades térmicas:

- Valores constantes;
- Correlação;
- Interpolação.

Para o primeiro caso, como o nome diz, a condutividade térmica será constante ao longo de todo o tempo, variando somente com a posição.

No segundo caso, será utilizado os modelos de correlação do *handbook Thermophysical Properties* [Valencia e Quested 2008]. O modelo proposto, é calculado, em geral, como:

$$k = C_0 + C_1 T - C_2 T^2 \quad (3.38)$$

onde C_0 , C_1 e C_2 são constantes da correlação, específicas para cada material.

O terceiro caso, é o cálculo pela interpolação e, como o nome diz, calcula a condutividade térmica pela interpolação linear entre valores obtidos em laboratório.

3.3 Formulação modelos computacionais

Apresenta-se a seguir considerações sobre o que é processamento paralelo, parallelismos usando múltiplas *threads* e informações sobre renderização 3D.

3.3.1 O que é processamento paralelo?

O processamento paralelo consiste em dividir uma tarefa em suas partes independentes e na execução de cada uma dessas partes em diferentes processadores ou núcleos.

A Figura 3.5 mostra um processamento serial e a Figura 3.6 sua versão paralelizada.

Note que os processadores podem estar numa mesma máquina, em um cluster ou em máquinas distribuídas.

Para poder aproveitar o poder de processamento dos novos processadores é preciso que os programas sejam desenvolvidos utilizando processamento paralelo.

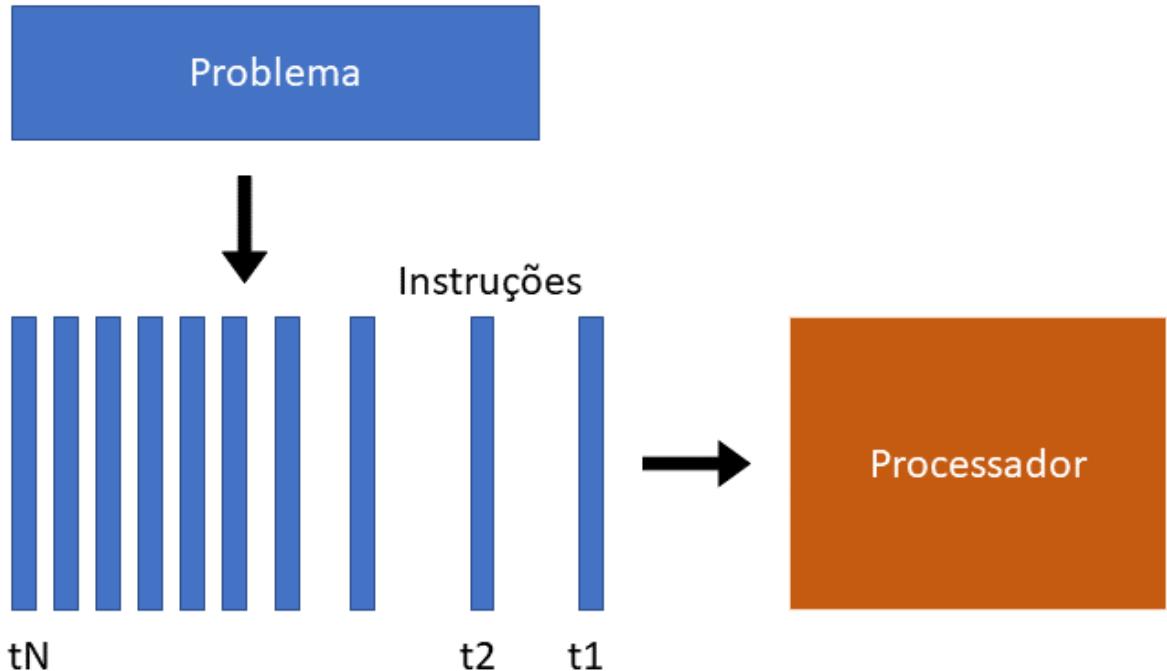


Figure 3.5: Processamento serial

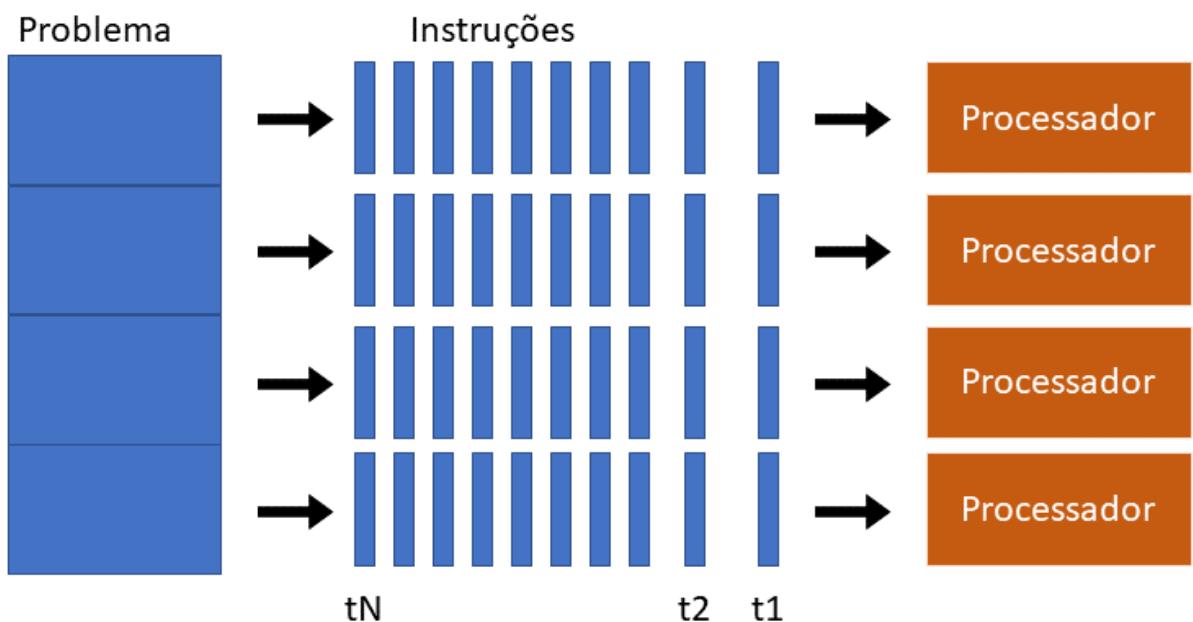


Figure 3.6: Processamento paralelo

Note que no processamento paralelo o acoplamento é mais forte, exigindo sistemas de comunicação mais velozes (como barramentos e/ou *switches* de alta velocidade). O controle dos recursos do sistema é mais refinado.

- **Vantagens:**

- Melhor uso dos recursos disponibilizados; Maior velocidade na realização das tarefas; Maior confiabilidade.

- **Desvantagens:**

- O controle dos recursos é de responsabilidade do programador.

3.3.2 *Processamento paralelo com múltiplas-threads - multi-thread*

Os chips dos processadores atuais são constituídos por vários processadores menores, o que permite que um mesmo processador consiga realizar tarefas (distintas ou iguais) nos processadores menores. A ideia é separar tarefas distintas, para que um processador não fique envolvido em uma única tarefa.

Uma analogia para melhorar a explicação sobre processamento paralelo é a da aplicação de um filtro sobre uma imagem bidimensional. O filtro a ser aplicado é exatamente o mesmo em toda imagem (mesmo algoritmo), sendo possível dividir a imagem em partes e entregar cada parte para um processador (*thread*) diferente executar.

Similarmente ao cenário acima, foram implementados três casos de paralelismo, por questão de didática.

1. Sem paralelismo: uma única *thread* do processador resolve todos os cálculos.
2. Paralelismo por *grid*: cada *thread* resolve uma camada do objeto. Possui certa otimização em relação ao anterior, mas, se só existir objeto em uma camada, outras threads ficam ociosas.
3. Paralelismo total: todas as *threads* do processador resolvem os cálculos de todo o objeto 3D, intercalando a posição com base no número da *thread*.

A Figura 3.7 ilustra melhor esses três casos.

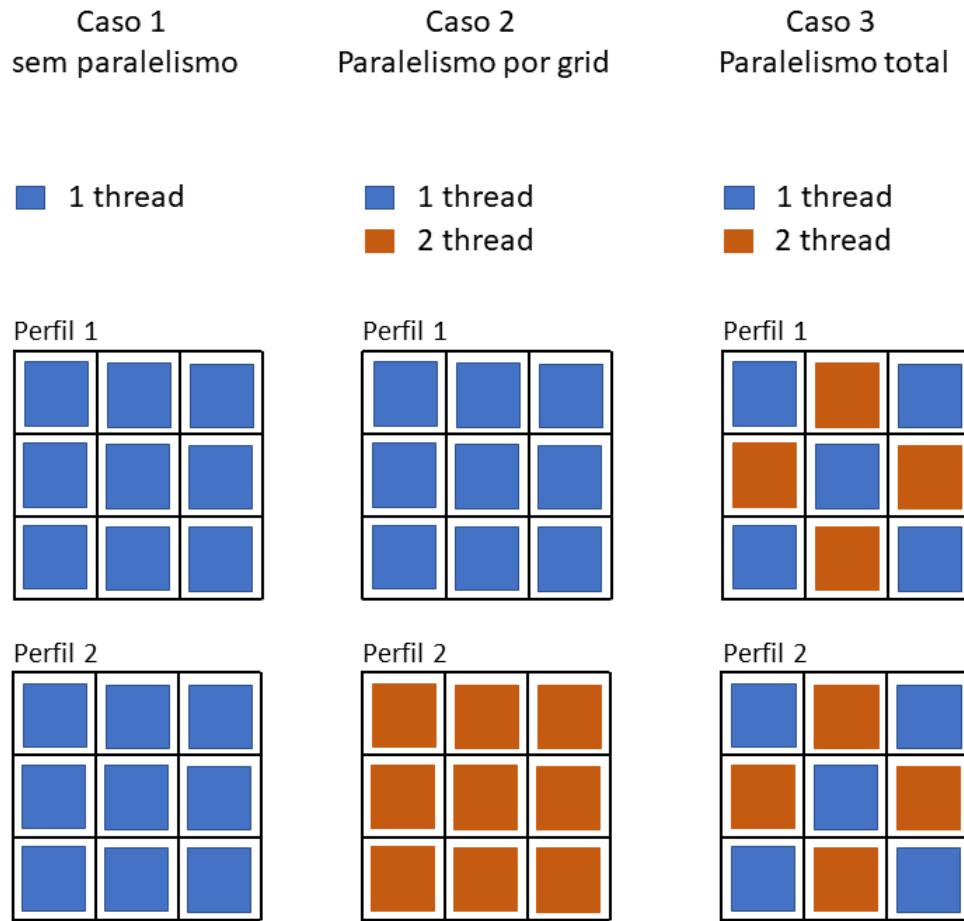


Figure 3.7: Ilustração dos três casos de paralelismo implementados para duas camadas com 9 células cada, e um processador com duas *thread*

O algoritmo utilizado para o caso 3 é:

```
for(int i = NUM_THREAD; i < size; i+=MAX_THREADS)
```

Esse algoritmo diz que a *thread* “i”, deverá começar a resolver as equações na posição “i”. Quando finalizar, deve pular para a posição “i + números de *thread*”.

3.3.3 Renderização 3D

Após o usuário desenhar algum objeto no *software*, pode ser de interesse observar como seria em renderização 3D. Portanto, foram implementados algoritmos para essa renderização.

Inicialmente, é interessante observar a complexidade da renderização: um objeto 3D deve ser apresentado em uma tela 2D, com a ilusão de ótica que é um objeto com profundidade. Por exemplo, um cubo com arestas de tamanho 1 cm é mostrado nos quatro casos da figura abaixo:

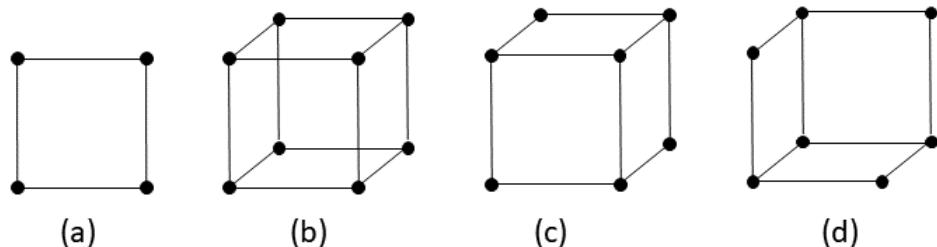


Figure 3.8: (a) Observador alinhado com uma das faces do cubo. (b) observador não está alinhado e não foram removidas arestas ocultas. O cérebro consegue interpretar que é um objeto 3D, mas fica confuso entre os casos (c) e (d)

Todos cantos do cubo da Figura 3.8 estão na mesma posição, o que mudou foi o ângulo do observador com o objeto.

Portanto, tendo definidos os pontos das arestas, seus respectivos vetores são multiplicados pela matriz de rotação [Herter e Lott] mostrada na Eq. 3.39, a qual permite rotacionar qualquer ponto a partir dos três ângulos do observador.

$$R(\alpha, \beta, \gamma) = \begin{bmatrix} \cos(\gamma)\cos(\beta) & \cos(\gamma)\sin(\beta)\sin(\alpha) - \sin(\gamma)\cos(\alpha) & \cos(\gamma)\sin(\beta)\sin(\alpha) + \sin(\gamma)\cos(\alpha) \\ \sin(\gamma)\cos(\beta) & \sin(\gamma)\sin(\beta)\sin(\alpha) + \cos(\gamma)\cos(\alpha) & \sin(\gamma)\sin(\beta)\cos(\alpha) - \cos(\gamma)\sin(\alpha) \\ -\sin(\beta) & \cos(\beta) * \sin(\alpha) & \cos(\beta) * \cos(\alpha) \end{bmatrix} \quad (3.39)$$

onde α , β e γ são os Ângulos de Euler.

Ou seja, inicialmente, um cubo de aresta 3 cm, com uma distância da origem de 1 cm, pode ser mostrado na tela (monitor) com os pontos do caso (a) da Figura 3.9, onde o observador está alinhado com o objeto.

Conforme desejado, o objeto pode mudar seu ângulo com o observador, como no caso (b), onde os ângulos α e β passaram a ter o valor de 0.1 radianos. Não foi só os pontos de trás do cubo que aparecem (e mudaram seus valores), mas todos os pontos foram modificados.

Além disso, a aresta possui valor ligeiramente menor que 3, pois não é mais “de frente” que o observador está olhando, mas ligeiramente de lado. Mesmo que o objeto cubo tenha aresta de 3 centímetros.

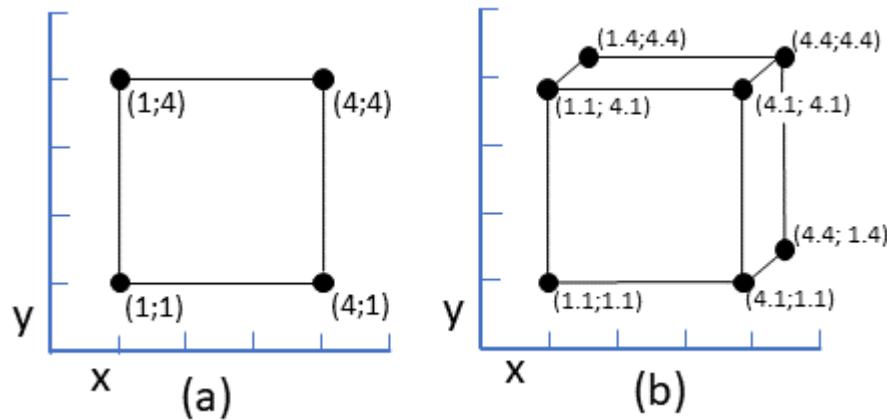


Figure 3.9: (a) o cubo está com ângulos nulos. (b) os ângulos α e β estão com valor de 0.1 radianos

Nos desenhos do simulador, cada pixel da figura, é uma célula com propriedades que serão calculadas, possuindo material, temperatura e volume. Como o usuário pode desenhar por pixel, a renderização 3D deve partir do princípio que cada pixel é um potencial objeto que deve ser renderizado.

Inicialmente, essa conclusão pode ficar vaga, pois todas as células do simulador devem ser renderizadas, mas, quando a simulação fica grande, é numerosa a quantidade de objetos renderizando ao mesmo tempo, tornando muito lenta a apresentação. Então algumas considerações são feitas no algoritmo para otimizar a renderização.

Primeiro, é desejável desenhar triângulos, e não pontos ou retas, por 2 motivos: geometria simples, possui normal e a biblioteca do Qt consegue desenhar e preencher a área com qualquer cor escolhida.

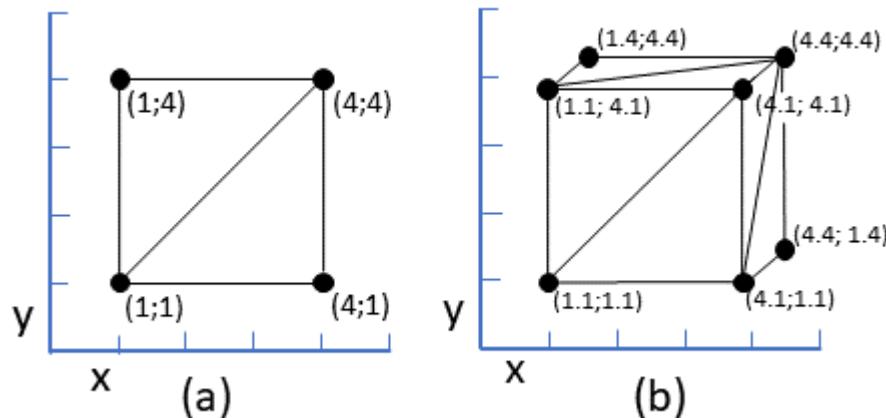


Figure 3.10: Mesmo desenho da Figura 3.9, mas agora renderizando a partir de triângulos

O segundo motivo apresentado, é o mais importante dos três. Um triângulo possui três pontos, podendo ser reduzido para dois vetores (subtraindo o ponto de origem dos

outros dois pontos) e permite-se calcular a normal dessa superfície. Com isso, são obtidos os vetores $\mathbf{a} = \{a_1, a_2, a_3\}$ e $\mathbf{b} = \{b_1, b_2, b_3\}$ permitindo a realização do produto vetorial:

$$\mathbf{a} \times \mathbf{b} = \begin{bmatrix} \mathbf{i} & \mathbf{j} & \mathbf{k} \\ a_1 & a_2 & a_3 \\ b_1 & b_2 & b_3 \end{bmatrix} \quad (3.40)$$

Ou simplesmente:

$$\mathbf{a} \times \mathbf{b} = (a_2 b_3 - a_3 b_2) \mathbf{i} - (a_1 b_3 - a_3 b_1) \mathbf{j} + (a_1 b_2 - a_2 b_1) \mathbf{k} \quad (3.41)$$

Utilizando a Regra da Mão Direita¹, é possível entender a utilidade da equação 3.41: o caso (a) da figura 3.11, mostra uma normal saindo do papel, em direção ao olho do leitor, logo, é um triângulo que deve ser renderizado. O caso (b) possui uma normal no sentido contrário, e não faz sentido desenhar esse triângulo, pois está na parte de trás do objeto.

¹

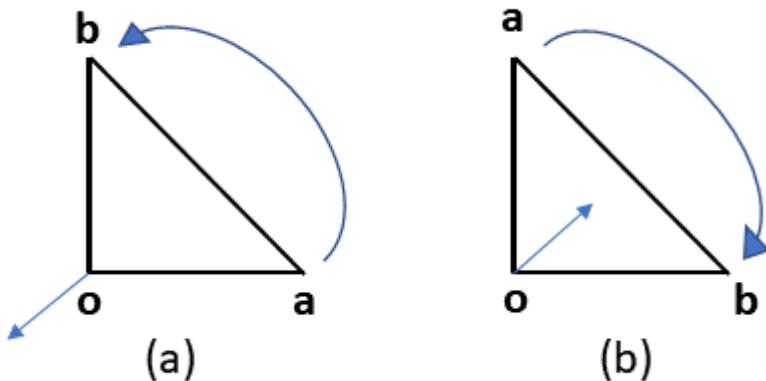


Figure 3.11: (a) mostra um caso onde a normal é na direção do leitor e (b) mostra um caso onde a normal é para dentro da folha

Essa simples operação condicional do valor positivo (apontando para o olho do observador) e negativo (apontando para dentro da tela) de \mathbf{j} da normal, reduz a quantidade de objetos que devem ser renderizados, e otimiza o software em duas vezes.

Uma outra condição implementada é a de avaliar se o objeto está em contato com outro objeto. Na superfície de contato, existem 4 triângulos (2 de cada objeto), e eles não devem ser renderizados, pois estão no interior do objeto maior. Para esse cenário, pode-se pensar em blocos de montar: enquanto eles não estão juntos, é possível observar todas as superfícies do bloco mas, quando eles são encaixados, essa superfície de contato entre eles fica oculta.

¹Para utilizar a Regra da Mão Direita, posicione o dedo polegar sobre o ponto \mathbf{o} , e estique o indicador para o ponto \mathbf{a} , agora, feche o indicador no sentido do ponto \mathbf{b} (seta curvada mostra o sentido que a ponta do indicador deve realizar). No caso (a) da figura, o dedo polegar fica no sentido para fora do papel, e o caso (b), para dentro.

Por fim, antes de renderizar os numerosos triângulos, eles são colocadas em ordem crescente com o valor de j da normal. Isso serve para ser desenhado primeiro o que está atrás, e depois desenhar o que está na frente, sobrescrevendo áreas que deveriam estar ocultas, evitando a criação de figuras confusas como no caso (b) da Figura 3.8. É uma técnica lenta, mas de fácil implementação.

3.4 Identificação de pacotes

- Pacote de malhas: organiza o objeto desenhado em vetores, facilita o acesso do simulador às propriedades de cada célula.
- Pacote de simulação: nele está presente o coração do simulador: o *solver* da equação da temperatura, discretizada por métodos numéricos, e resolvida por método iterativo.
- Pacote de interpolação: utilizado para realizar interpolação com propriedades termofísicas dos materiais, é acessado pelo simulador, e retorna as propriedades do material.
- Pacote de correlação: mesma função da linha acima, mas para método de correlação.
- Pacote de interface ao usuário: utilização da biblioteca Qt, para criar interface gráfica amigável. Fornece um ambiente onde o usuário pode enviar comandos para o simulador de maneira fácil, e apresenta os resultados.
- Pacote de gráficos: utilização da biblioteca *qcustomplot*, para montar os melhores gráficos para o problema. É solicitado ao pacote de malhas os resultados da temperatura. Está presente junto com o pacote de interface.

3.5 Diagrama de pacotes

O diagrama de pacotes é apresentado na Figura 3.12.

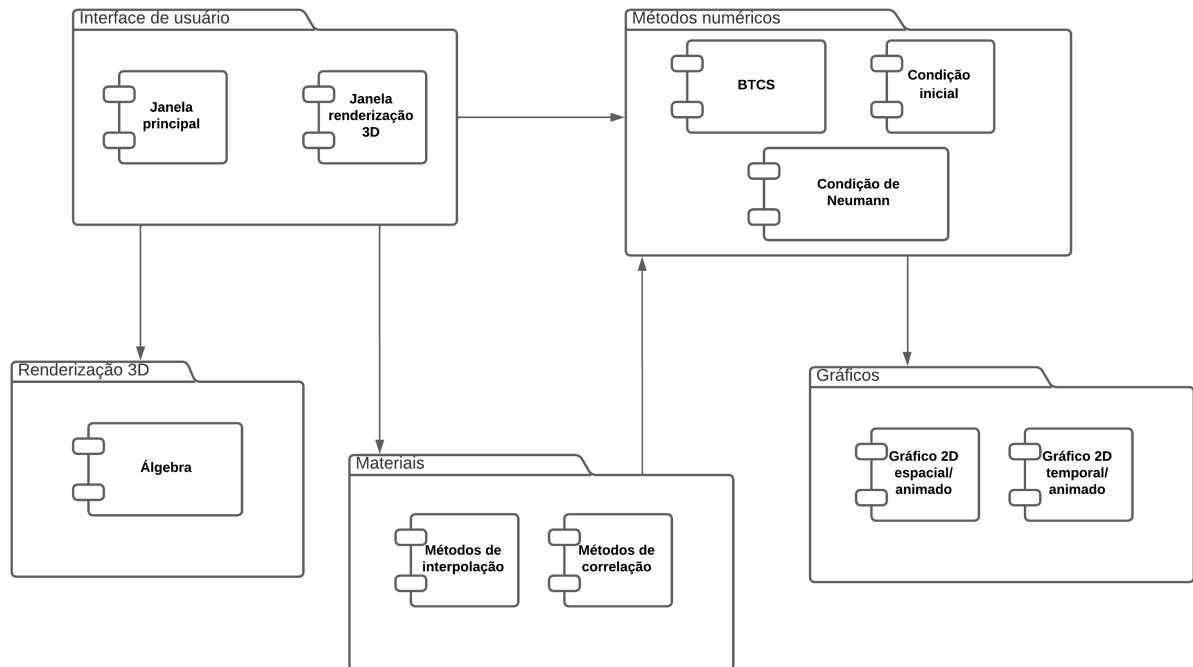


Figure 3.12: Diagrama de Pacotes

Chapter 4

Análise Orientada a Objeto

Neste capítulo serão apresentados os objetos desenvolvidos no projeto, suas relações, atributos e métodos. Depois de uma breve explicação sobre cada objeto, serão apresentados diagramas UML (Linguagem de Modelagem Unificada), para auxiliar no entendimento do *software* e suas relações. Serão apresentados os diagramas de classes, de sequência, de comunicação, de máquina de estado e de atividades.

4.1 Dicionário das classes

O *software* é constituído por 10 classes, onde duas classes de interpolação foram implementadas pelo professor André Duarte Bueno no curso de C++. Utilizar esse código pronto, mostra que o simulador está apto a adições de métodos para o cálculo das propriedades termofísicas.

1. **MainWindow:** Classe responsável pela janela principal. Consegue obter os valores adicionados pelo usuário e enviar para a classe do simulador. Permite ao usuário desenhar o objeto a ser simulado, setar suas propriedades, setar as propriedades numéricas e disparar a simulação. Apresenta os resultados na medida em que os mesmos são realizados, possibilitando parar a simulação para ver resultados parciais.
2. **CRender3D:** Classe responsável por apresentar o objeto em renderização 3D. É criada a partir da classe mainwindow.h e recebe valores do simulador. Possui toda a álgebra necessária para a renderização.
3. **CSimuladorTemperatura:** Classe responsável por organizar as células do objeto a ser simulado e por resolver o sistema numérico do problema da difusão térmica.
4. **CGrid:** Classe responsável por organizar as células do objeto em *grids*, importante para organizar as células e facilitar a utilização pela classe CSimuladorTemperatura.
5. **CCell:** Classe responsável por armazenar informações da célula, se ela está ativa ou não, se é fonte de calor ou não, qual o tipo de material e qual sua temperatura.

6. **CMaterial:** Classe virtual responsável por prover os valores das propriedades termofísicas dos materiais. É chamada pelo CSimuladorTemperatura e é sobreescrita por CMaterialCorrelacao ou CMaterialInterpolacao.
7. **CMaterialCorrelacao:** Classe responsável por calcular os valores das propriedades termofísicas como função da temperatura, a propriedade é determinada a partir de correlações lidas do disco.
8. **CMaterialInterpolacao:** Classe responsável por calcular os valores das propriedades termofísicas como função da temperatura, a propriedade é determinada utilizando método de interpolação linear a partir de dados experimentais armazenados em disco.
9. **CReta:** Representa uma reta, dado um valor de x calcula y. Usada no cálculo da interpolação linear.
10. **CSegmentoReta:** Representa um segmento de reta com intervalos definidos. .

Uma classe externa foi utilizada no simulador:

1. **QCustomPlot:** Classe responsável por gerar gráficos apresentados pelo mainwindow, obtido em (<https://www.qcustomplot.com/>).
2. Também foram utilizadas diversas classes da biblioteca Qt (<https://www.qt.io/>).

O diagrama de classes é apresentado na Figura 4.1. Ele tem como objetivo apresentar todas as classes, seus atributos, métodos, heranças e relações entre as classes.

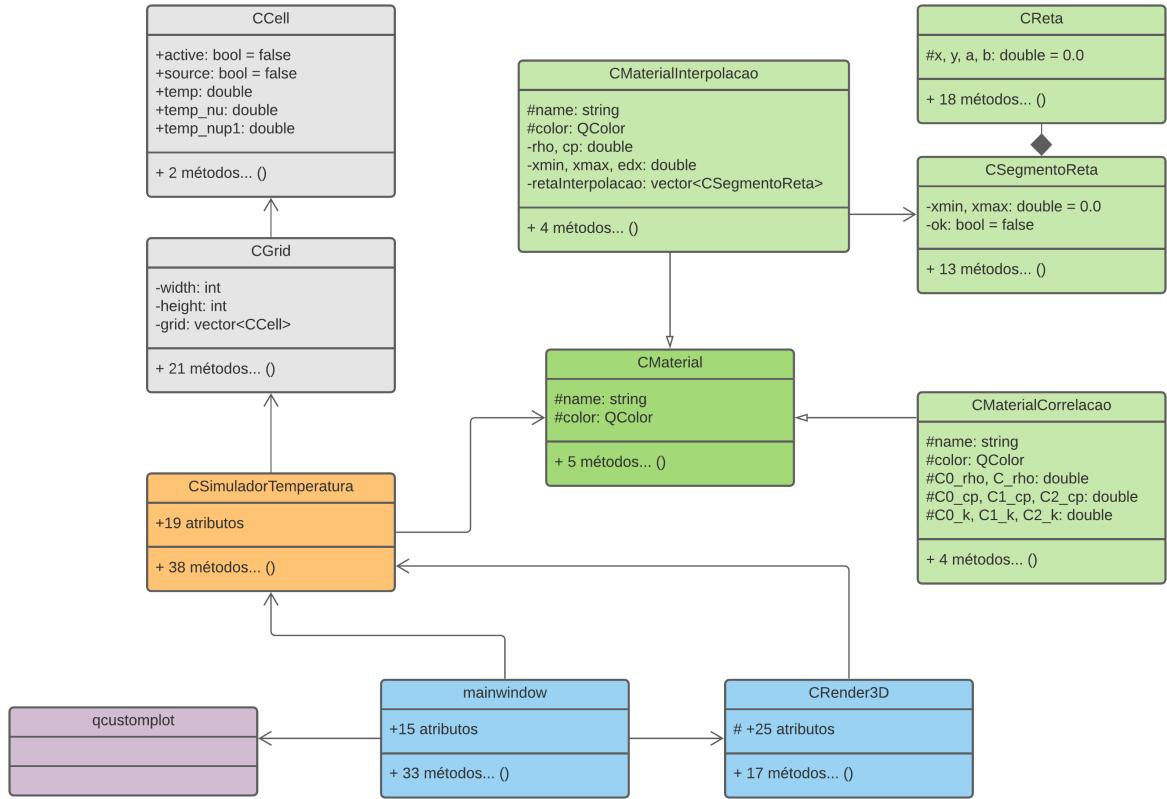


Figure 4.1: Diagrama de classes

4.2 Diagrama de sequência

O diagrama de sequência enfatiza a troca de eventos e mensagens e sua ordem temporal, representando um cenário de uso do software. Contém informações sobre o fluxo de controle do software. Costuma ser montado a partir de um diagrama de caso de uso e estabelece o relacionamento dos atores (usuários e sistemas externos) com alguns objetos do sistema.

4.2.1 Diagrama de sequência - cenário geral

A seguir, é apresentado o diagrama de sequência geral na Figura 4.2, conforme o exemplo do caso de uso da Figura 2.1.

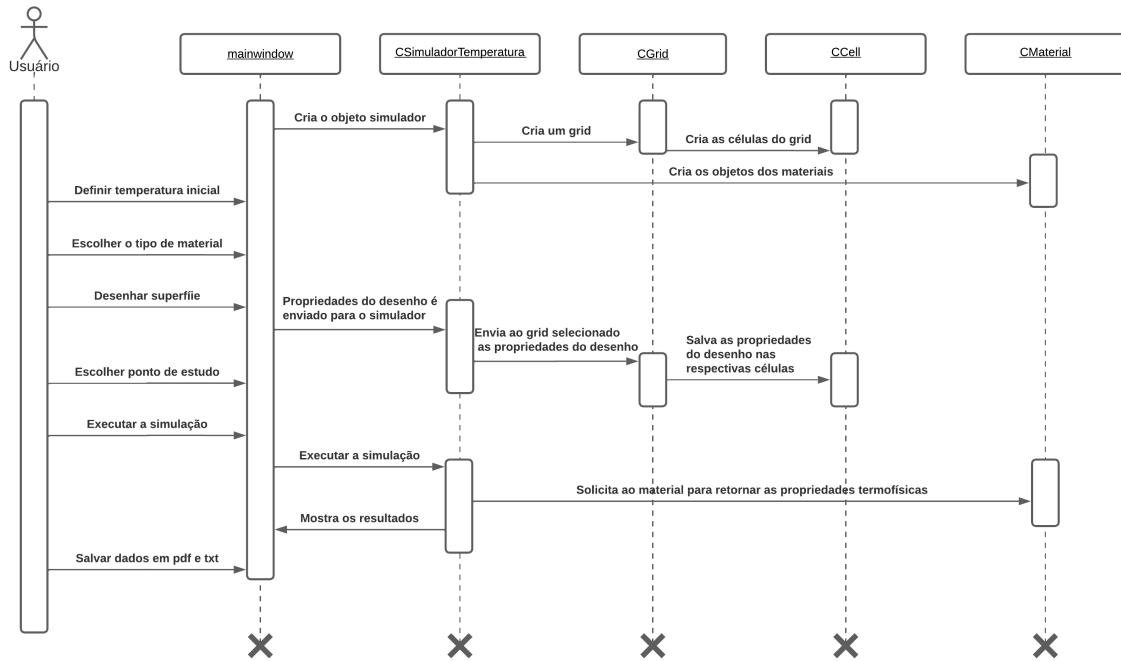


Figure 4.2: Diagrama de sequência geral

4.2.2 Diagrama de sequência - adicionando novo material/correlação

Uma atividade corriqueira é adicionar ao simulador um novo material. Neste caso os dados do material são salvos em um arquivo de disco, em formato pré-definido e então lidos pelo simulador. . O diagrama de sequência da Figura 4.3 mostra este cenário.

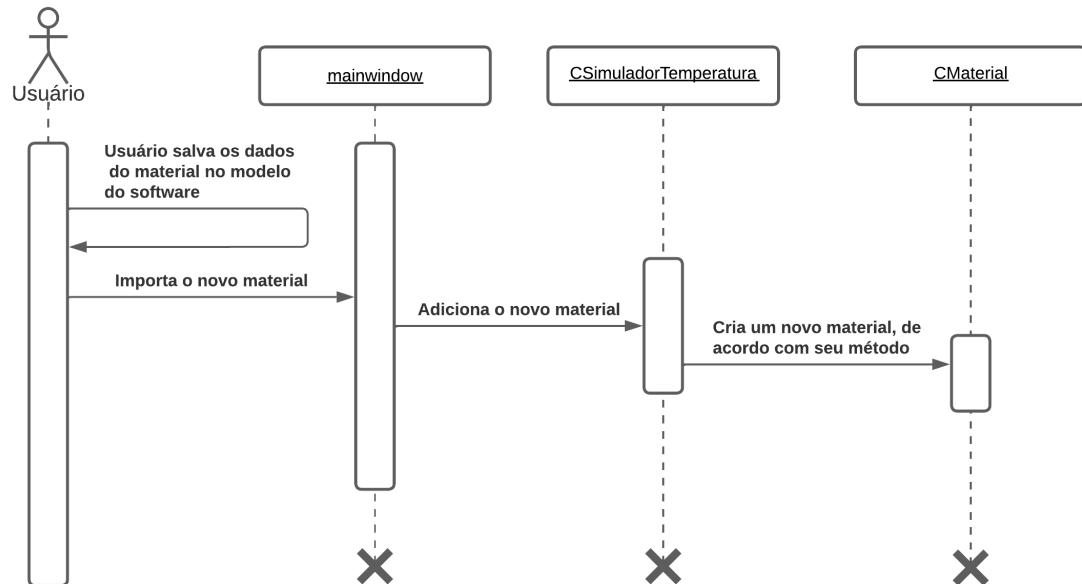


Figure 4.3: Diagrama de sequência mostrando a importação de um novo material

4.2.3 Diagrama de sequência - análise de resultados

A análise de resultados é a etapa final da simulação, e é realizada por meio dos gráficos. A sequência de atualização é mostrada na Figura 4.4.

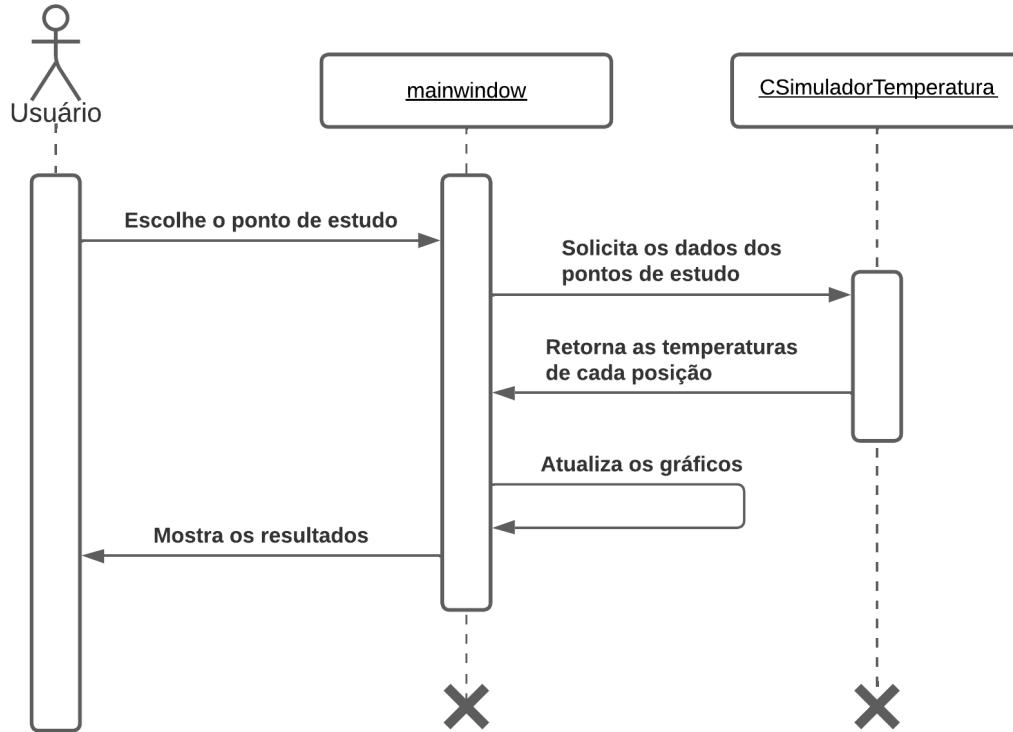


Figure 4.4: Diagrama de sequência mostrando a análise de resultados

4.3 Diagrama de comunicação

O diagrama de comunicação tem como objetivo apresentar as interações dos objetos, juntamente com sua sequência de processos.

O diagrama de comunicação do caso de uso geral é apresentado na Figura 4.5.

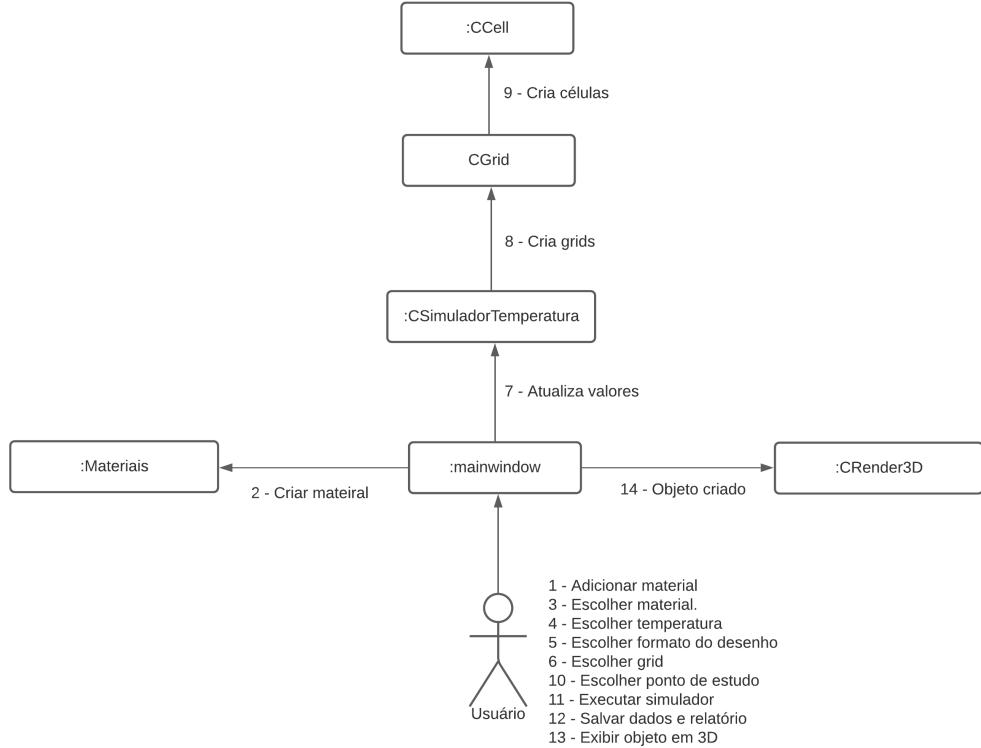


Figure 4.5: Diagrama de comunicação

4.4 Diagrama de máquina de estado

O diagrama de máquina de estado descreve os estados de uma classe desde o momento de sua criação, até sua destruição. A Figura 4.6 mostra a máquina de estado para a classe CSimuladorTemperatura.

Inicialmente, os dados são recebidos pela classe CSimuladorTemperatura, seus atributos são atualizados, e é iniciado o método BTCS. Conforme a opção de paralelismo, ele pode seguir para uma dentre as três sequências, chegando na comparação com a iteração anterior.

Como o método resolve os sistemas de equações de maneira iterativa, é necessário a comparação entre os valores calculados, com a iteração anterior, para avaliar se os valores convergiram. Se não convergiu, ele retorna para os cálculos, se convergiu, ele atualiza as temperaturas nos *grids*, e finaliza suas ações.

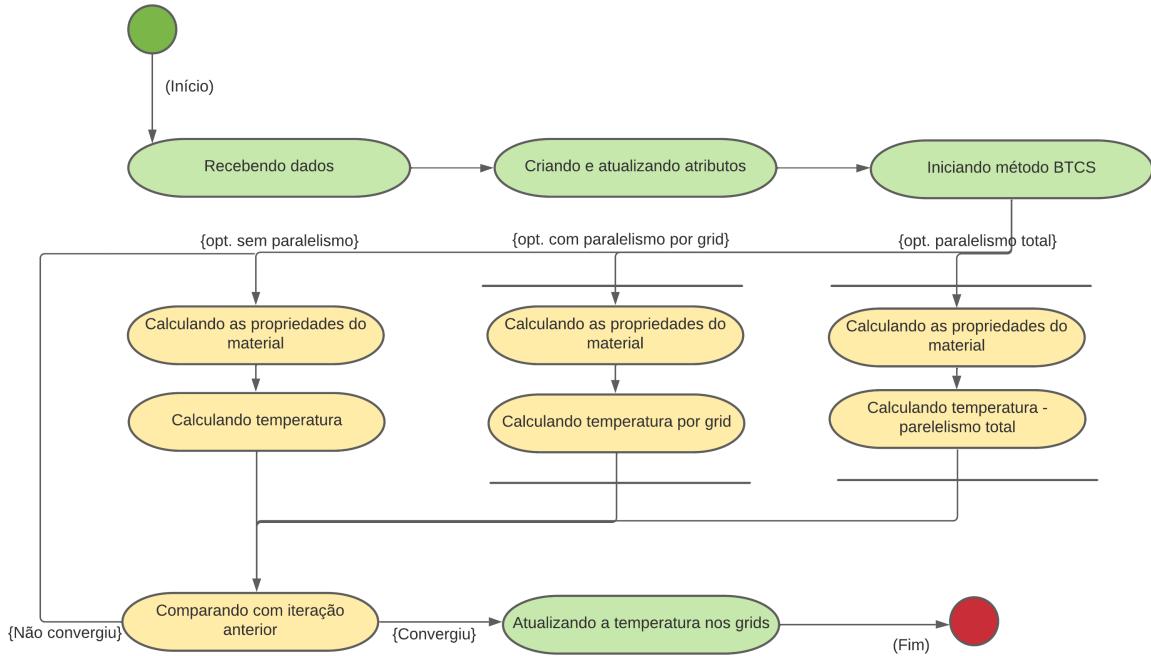


Figure 4.6: Diagrama de máquina de estado para a classe CSimuladorTemperatura

4.5 Diagrama de atividades

No diagrama de atividades apresentado na Figura 4.7, é mostrado em detalhes uma atividade específica. Para o presente caso, será apresentado o diagrama de atividades do algoritmo de renderização da classe CRender3D, devido à complexidade de renderização 3D.

Inicialmente, essa classe recebe os dados do simulador, as posições e os atributos das células. Com isso, são criadas matrizes com os pontos de triângulos para cada respectivo ponto, se a superfície desse objeto for possível de se observar. Ou seja, se existir uma superfície em contato com essa outra superfície, nenhuma das duas serão criadas, pois estarão dentro do objeto.

A partir desse ponto, são obtidos os valores dos ângulos e calculado a matriz rotacionada dos pontos dos triângulos e, em seguida, obtidas suas normas. Se positivo, esse valor é guardado em uma matriz, a qual é ordenada em ordem crescente.

Com todos os valores calculados e ordenados, a classe renderiza na tela o objeto 3D. Conforme condições do usuário, o objeto pode concluir suas atividades, ou calcular novos valores para outra renderização.

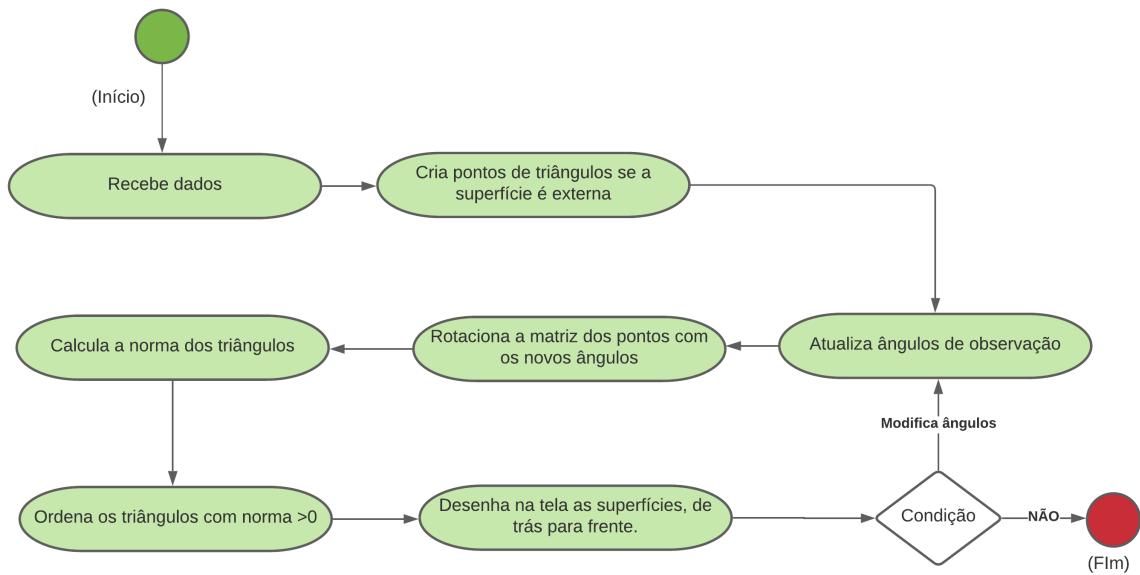


Figure 4.7: Diagrama de atividades para o algoritmo de renderização CRender3D::Renderizacao()

Chapter 5

Projeto

Neste capítulo são apresentadas questões relacionadas ao desenvolvimento do projeto, como ambiente de desenvolvimento e bibliotecas gráficas, comentados juntamente com a evolução de versões. Também são apresentados os diagramas de componentes e de implantação.

5.1 Projeto do sistema

O paradigma de programação selecionado foi o orientado o objeto.

A linguagem de programação selecionada foi C++ pelos seguintes motivos:

- Linguagem de programação de alto desempenho, adequada para cálculos numéricos intensivos.
- Amplo suporte a orientação a objeto e diversos vínculos com UML.
- Dezenas de bibliotecas de suporte prontas, incluindo interface gráfica amigável (biblioteca Qt), gráficos (biblioteca QCustomPlot) e bibliotecas para gerar saídas em pdf.
- Programação de alto nível com vários graus de abstração.
- Diversos ambientes de desenvolvimento - IDEs, compiladores, debugers, profilers.
- Compiladores e ferramentas gratuítas disponíveis, facilitando o uso por alunos.

Iremos apresentar a seguir os diagramas de componentes e de implementação.

5.2 Diagrama de componentes

O diagrama de componentes mostra as relações entre todos os componentes do *software* e é apresentado na Figura 5.1

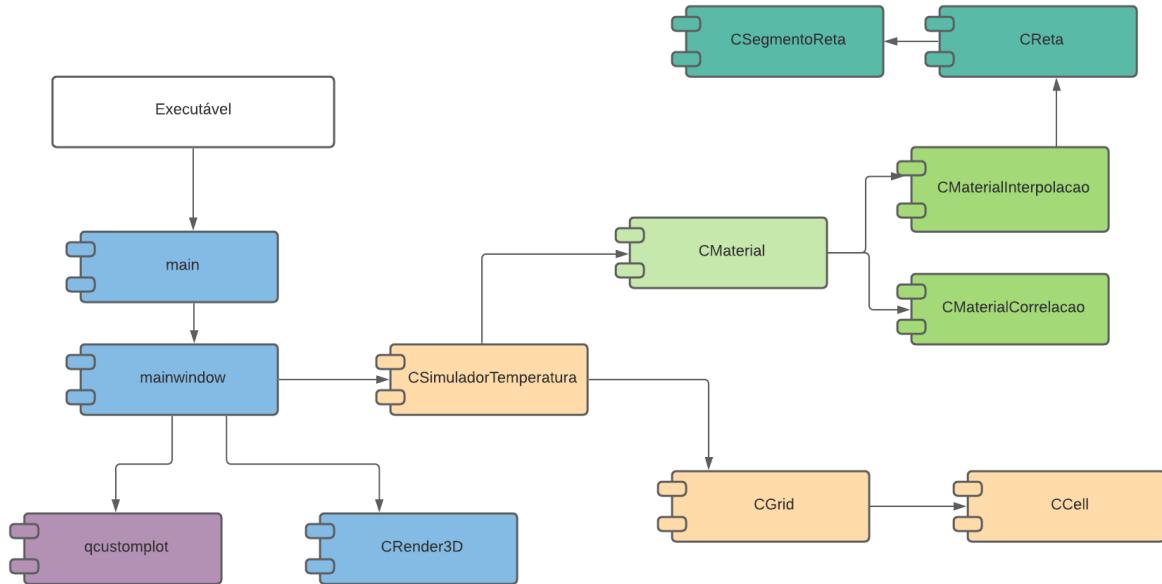


Figure 5.1: Diagrama de componentes

Começando pela esquerda na Figura 5.1, temos o executável e o main. Os outros dois componentes em azul criam janelas para o usuário se comunicar com o *software*. O componente em roxo, é a biblioteca de gráficos utilizados pelo *software*.

Os componentes em verde, são responsáveis por calcular as propriedades dos materiais, por interpolação ou correlação.

Por fim, os componentes em laranja claro são responsáveis pela simulação da difusão térmica.

5.3 Diagrama de implementação

O diagrama de implementação é um diagrama que apresenta as relações entre sistema e *hardware*, mostrando quais equipamentos são necessários para que o *software* seja executado corretamente.

Na Figura 5.2, são mostrados quais equipamentos o *software* SDT-3D utiliza. No caso, são utilizados os periféricos de um *desktop* ou *notebook*, o processador, memória RAM e memória de longo prazo, como um HD ou SSD.

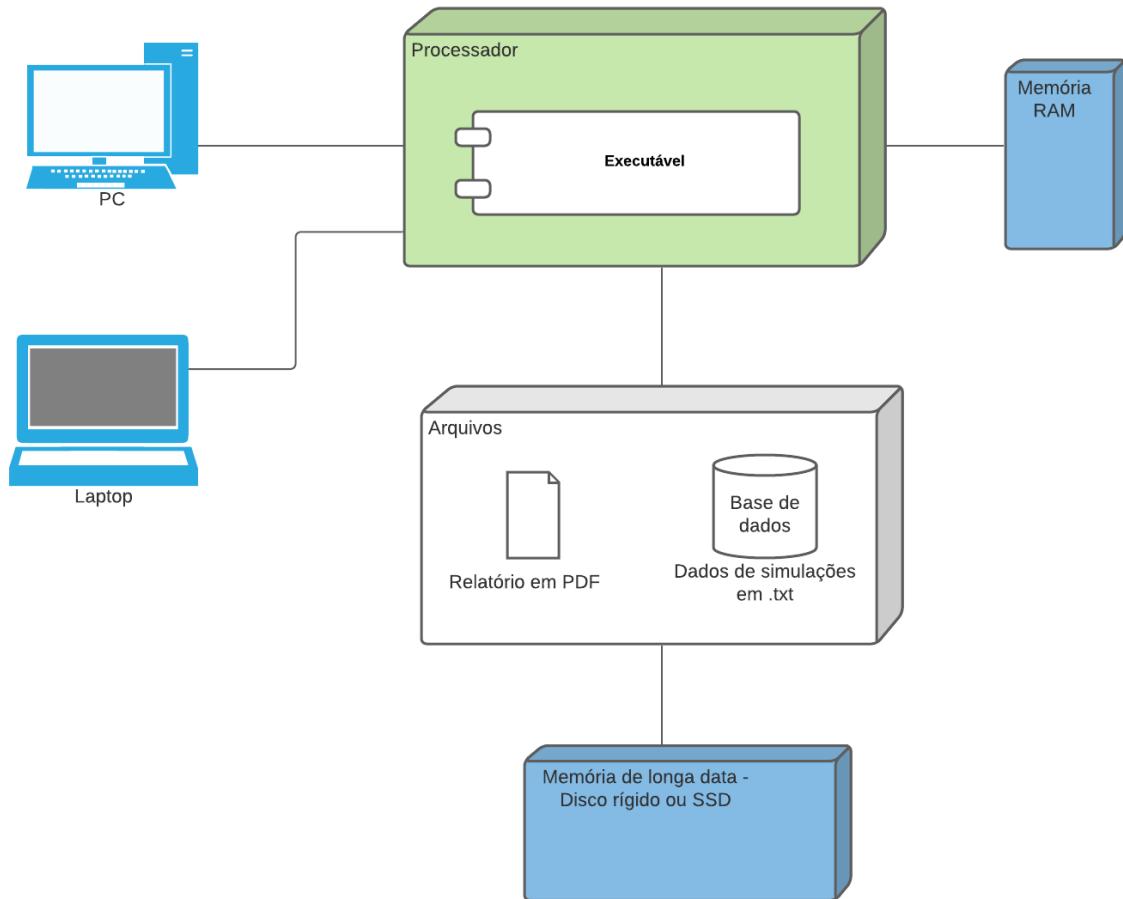


Figure 5.2: Diagrama de implementação

O software foi desenvolvido utilizando o padrão apresentado pelo Prof. André Bueno na disciplina de programação e ilustrado na Figura 1.1. Note que o “Ciclo de Concepção e Análise” é composto pelo que foi apresentado nos capítulos anteriores e neste capítulo. Os “Ciclos de Planejamento/Detalhamento” e “Ciclo Construção”, envolvem a construção das diferentes versões do software, e serão apresentados nos próximos capítulos.

Chapter 6

Ciclos de Planejamento/Detalhamento

6.1 Versão 0.1 - Uso modo terminal e biblioteca SFML para saída gráfica

Na primeira versão foi utilizado um sistema de entrada de dados via terminal e saída de dados gráfica. A biblioteca *SFML*¹ foi utilizada para a criação de janelas para o usuário. Esta versão foi desenvolvida usando ambiente de desenvolvimento *Visual Studio*, tudo isso no sistema operacional *Windows 10*.

Note que é um *software* simples, com uma mistura de janela-terminal. O usuário podia desenhar e simular, mas não tinha muita liberdade para escolher e adicionar materiais. Veja Figura 6.1.

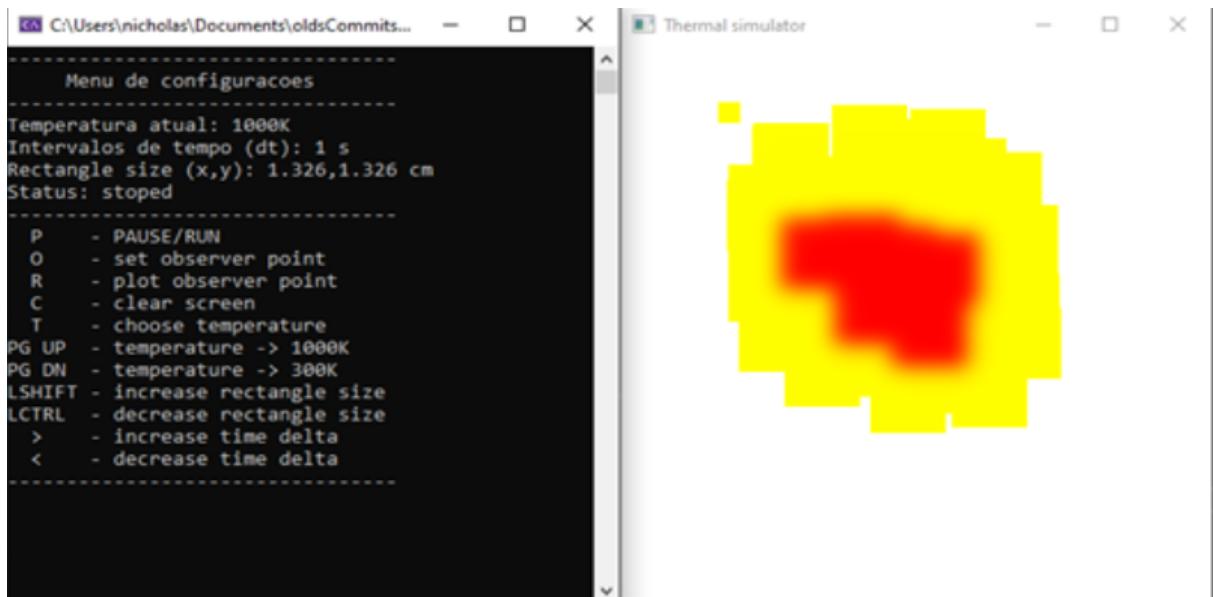


Figure 6.1: Versão 0.1, simples e utilizando a biblioteca *SFML*

¹*Simple and Fast Multimedia Library* é uma biblioteca de interface gráfica multi-linguagem, ver em <<https://www.sfml-dev.org/>>.

6.2 Versão 0.2 - Adição de visualização para os tipos de materiais

Nesta versão foram implementadas as seguintes melhorias:

- Criada uma segunda janela, a qual replica o desenho com as cores do material escolhido.
- Opção de navegação entre perfis, tornando o simulador tridimensional.
- Opção de escolher paralelismo por *grids* ou sem paralelismo.
- Foram realizadas otimizações e organizações de código.
- Adição da opção da temperatura ser uma fonte/sumidouro de calor (temperatura constante no ponto analisado).

Veja Figura 6.2.

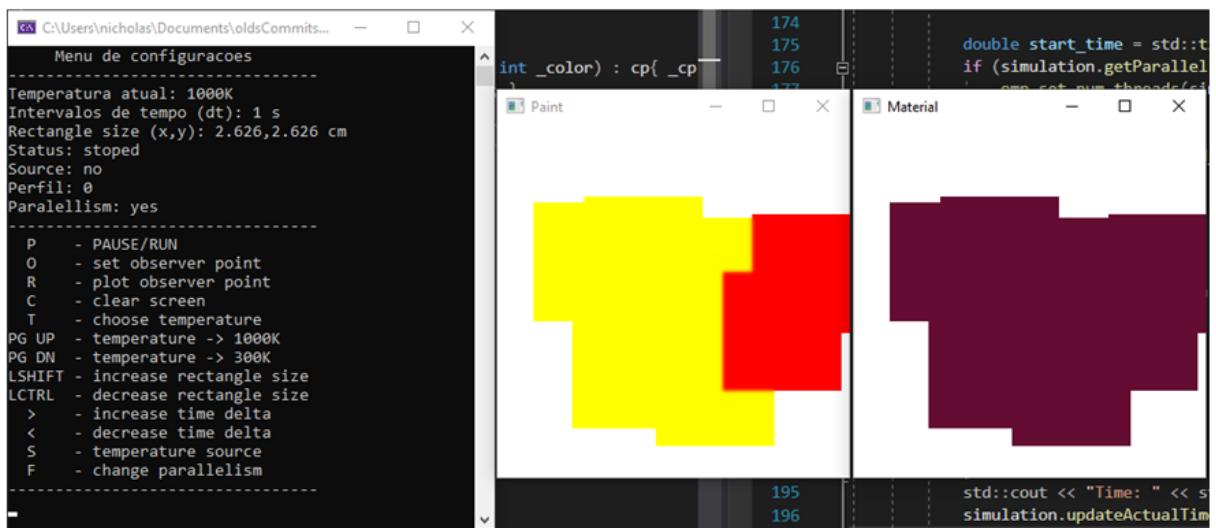


Figure 6.2: Versão 0.2, simples, mas contendo uma segunda janela dos materiais

6.3 Versão 0.3 - Adição de atalhos na tela

Esta foi a última versão montada utilizando a biblioteca *SFML*. Foi uma versão importantíssima para o aprendizado, pois o usuário não desenhava diretamente no *software*, mas era enviado uma lista de propriedades do desenho para o *grid* e, quando o desenho era atualizado, a biblioteca utilizava os valores do *grid*. Isso permitiu juntar as duas janelas.

Foram implementadas as seguintes melhorias:

- Adição de instruções de uso incluindo teclas de atalho.

- Junção das três janelas anteriores (terminal, janela de desenho e janela de materiais) em uma única janela.
- Implementado a opção do material variar com a temperatura, em versão básica de teste. Essa melhoria se desdobrou na opção de escolher materiais com condutividade térmica calculada por valor constante, correlação ou interpolação.

Esta versão é ilustrada na Figura 6.3.

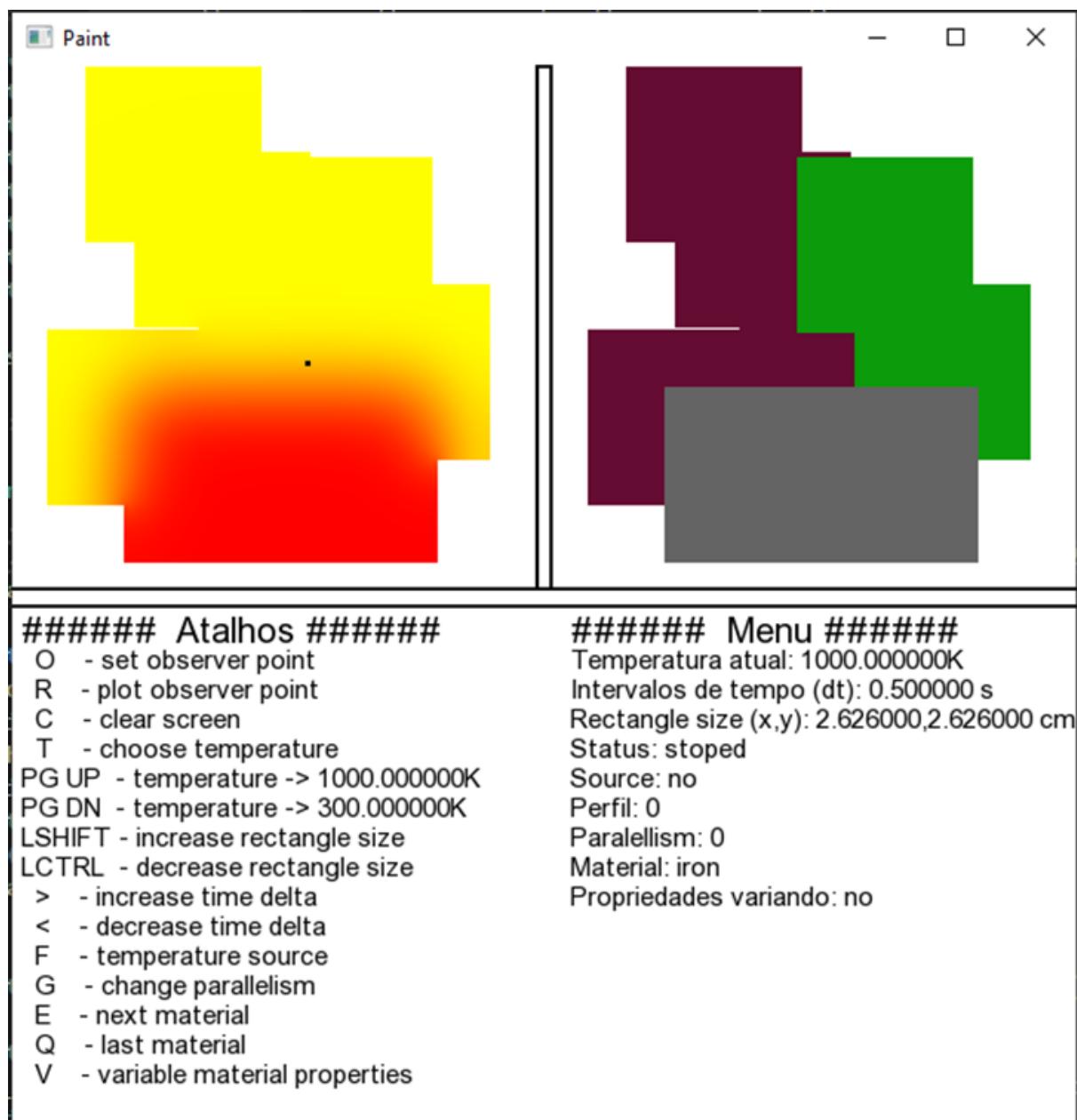


Figure 6.3: Versão 0.3, completa e complexa, mas muito lenta

6.4 Versão 0.4 - Mudança para biblioteca Qt

Durante o desenvolvimento das versões anteriores, foram identificados problemas com a dificuldade de lidar com mudanças na interface, quando se usa a biblioteca *SFML*. Também foi identificado problema de desempenho.

Então surgiu essa necessidade de mudança e a biblioteca gráfica *Qt* (<https://www.qt.io/>), foi escolhida por ser:

- mais completa.
- mais rápida.
- de amplo uso em programas de engenharia.
- com versões livres.
- multiplataforma.

Para utilizar as ferramentas fornecidas por essa biblioteca, foi migrado do editor de texto *Visual Studio* para o *Qt Creator*, a Figura 6.4 apresenta o ambiente de trabalho do *Qt Creator*.

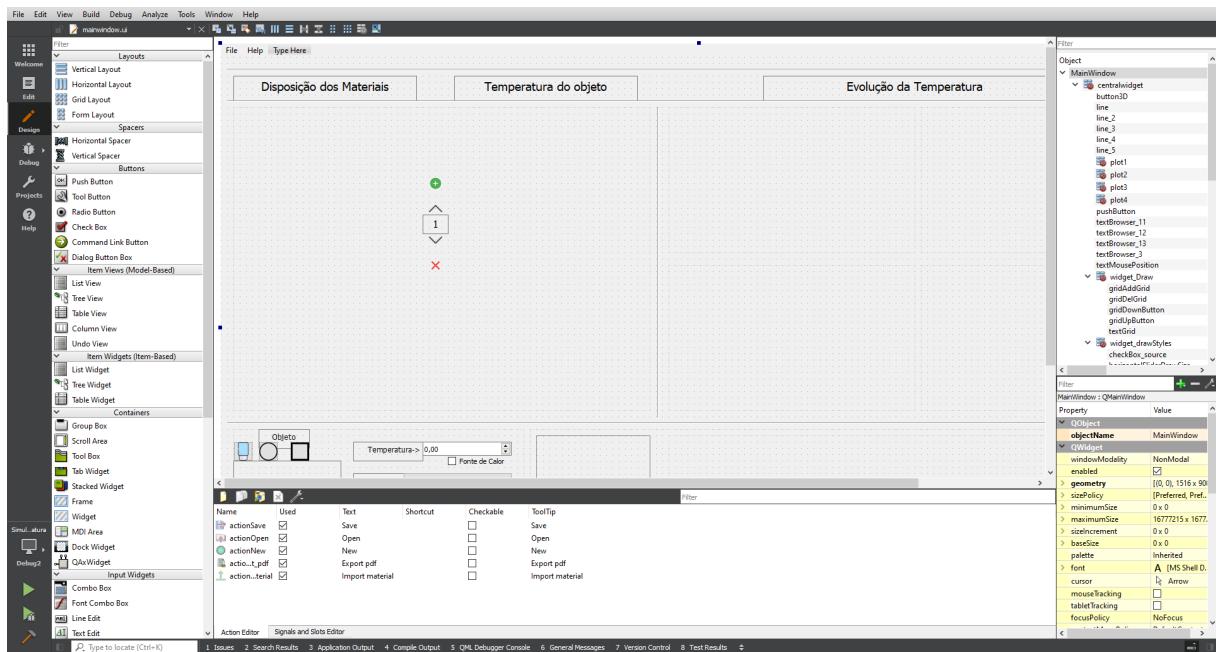


Figure 6.4: *Qt Creator*

Como o *software* foi todo programado utilizando o paradigma da orientação a objeto, esta migração foi muito rápida. Tivemos de modificar, quase que somente, a classe da janela.

Foram implementadas as seguintes melhorias:

- Adição na interface gráfica da entrada de dados da temperatura.

- Botão para selecionar o tipo de material.
- Definição de 4 áreas para saída gráfica.

A versão 0.4 do software é apresentada na Figura 6.5.

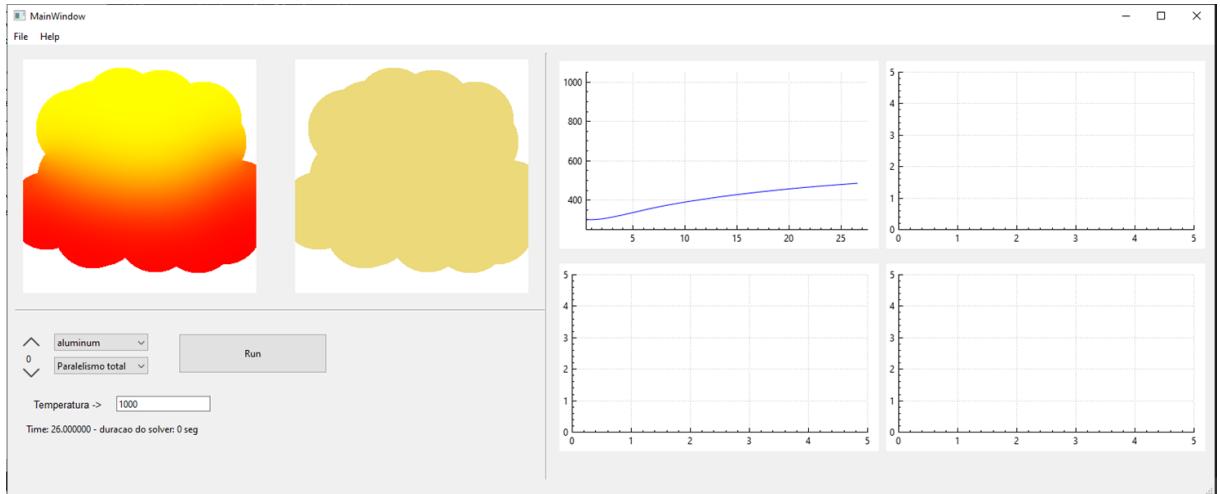


Figure 6.5: Versão 0.4, inicial e incompleta, mas utilizando a biblioteca *Qt*

6.5 Versão 0.5 - Melhorias na interface gráfica - usabilidade

A curva de evolução do *software* dentro do *Qt Creator* foi exponencial, permitindo a criação da versão final apresentada na figura 6.6.

Foram implementadas as seguintes melhorias:

- Duas áreas que apresentam os cortes desenhados.
- Quatro gráficos com valores da temperatura ao longo do tempo ou espaço.
- No canto inferior esquerdo, mostram opções para a simulação ou criação do objeto.
- Na direita, são mostradas as propriedades termofísicas de vários materiais como função da temperatura.
- Janela mostrando o objeto em renderização 3D.

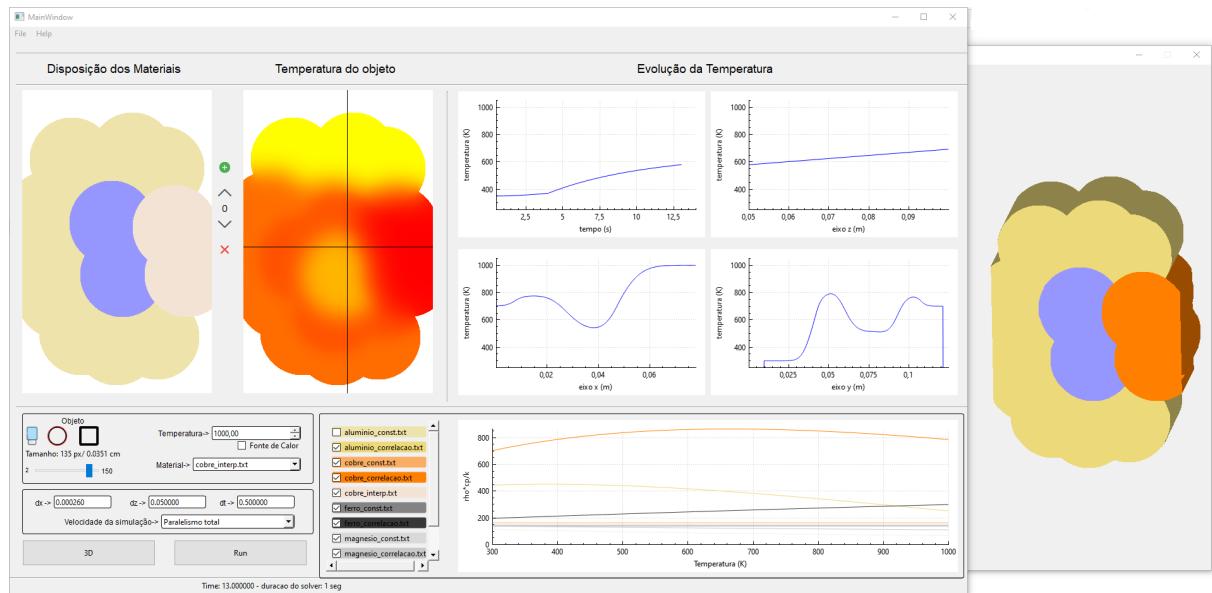


Figure 6.6: Versão 0.5, final. Na direita é apresentado a visualização 3D do objeto desenhado

Chapter 7

Ciclos Construção - Implementação

Neste capítulo, são apresentados os códigos fonte implementados.

7.1 Versão 0.3 - Código fonte - SFML

Como visto na seção 6.3, a versão 0.3 foi a última desenvolvida utilizando a biblioteca *SFML*, apresenta-se a seguir o código da última versão usando *SFML*.

Apresenta-se apenas a parte de interface gráfica na listagem ??.

7.2 Versão 0.5 - Código fonte - Qt

Apresenta-se a seguir um conjunto de classes (arquivos .h e .cpp) além do programa *main*.

Apresenta-se na listagem 7.1 o arquivo com código da função *main*.

Listing 7.1: Arquivo de implementação da função main

```
1 #include "mainwindow.h"
2 #include <QApplication>
3
4 int main(int argc, char *argv[])
5 {
6     QApplication a(argc, argv);
7     MainWindow w;
8     w.show();
9     return a.exec();
10 }
```

Apresenta-se na listagem 7.2 o arquivo de cabeçalho da classe *MainWindow*.

Listing 7.2: Arquivo de implementação da classe *MainWindow*

```
1 #ifndef MAINWINDOW_H
```

```
2#define MAINWINDOW_H
3
4
5#include <QDir>                                /// Biblioteca que permite acessar
6                                         diretórios.
7#include <QImage>                               /// desenhar pixels
8#include <QColor>                                /// escolher a cor dos pixels
9#include <string>
10#include <iostream>
11#include < QPainter>                            /// desenhar pixels
12#include < QPrinter>                            /// Biblioteca que habilita a
13                                         geração de pdf.
14#include <QPainter>                            /// Biblioteca que auxilia a
15                                         geração do pdf.
16#include <QPdfWriter>
17#include < QMainWindow>
18#include <QMouseEvent>                          /// pegar acoes/posicao do mouse
19#include "CRender3D.h"
20#include "ui_mainwindow.h"
21#include "CSimuladorTemperatura.h"
22
23
24QT_BEGIN_NAMESPACE
25namespace Ui { class MainWindow; }
26QT_END_NAMESPACE
27
28class MainWindow : public QMainWindow {
29    Q_OBJECT
30
31public:
32    MainWindow(QWidget *parent = nullptr);
33    ~MainWindow();
34
35private:
36    QDir dir;
37    Ui::MainWindow *ui;
38    QPoint m_mousePos;
39    QPixmap pixmap;
40    QImage *mImage;
```

```
41     QWidget* checkboxes;
42     QVBoxLayout* layout;
43     std::vector<QCheckBox*> myCheckbox;
44     CSimuladorTemperatura *simulador;
45     std::string drawFormat = "circulo";
46
47     int timerId;
48     int parallelType = 2;
49     int size_x = 300, size_y = 480;
50     int currentGrid = 0;
51     int space_between_draws = 50;
52     int left_margin = 20, up_margin = 140;
53     bool runningSimulator = false;
54     bool eraserActivated = false;
55     QPoint studyPoint = QPoint(0,0);
56     int studyGrid;
57     std::vector<bool> selectedMateriails;
58     QVector<double> time, temperature;
59
60 protected:
61     void start_buttons();
62     void mousePressEvent(QMouseEvent *event) override;
63     void printPosition();
64     void printDrawSize();
65     void paintEvent(QPaintEvent *e) override;
66     QImage paint(int grid);
67
68     QColor calcRGB(double temperatura);
69     void runSimulator();
70     void timerEvent(QTimerEvent *e) override;
71
72 private slots:
73     void on_pushButton_clicked();
74     void on_gridDownButton_clicked();
75     void on_gridUpButton_clicked();
76
77     void createWidgetProps();
78
79     void makePlot1();
80     void makePlot2();
81     void makePlot3();
82     void makePlot4();
```

```

83     void makePlotMatProps();
84     bool checkChangeMaterialsState();
85     void on_actionSave_triggered();
86     void on_actionOpen_triggered();
87     void on_actionNew_triggered();
88     void on_actionExport_pdf_triggered();
89     QString save_pdf(QString file_name);
90     void on_buttonCircle_clicked();
91     void on_buttonSquare_clicked();
92     void on_actionImport_material_triggered();
93     void on_gridAddGrid_clicked();
94     void on_gridDelGrid_clicked();
95     void on_buttonEraser_clicked();
96     void on_button3D_clicked();
97 };
98 #endif

```

Apresenta-se na listagem 7.3 implementação da classe mainwindow.

Listing 7.3: Arquivo de implementação da classe MainWindow

```

1 #include "mainwindow.h"
2
3 MainWindow::MainWindow(QWidget *parent)
4     : QMainWindow(parent), ui(new Ui::MainWindow)
5 {
6     up_margin = 100;
7     simulador = new CSimuladorTemperatura();
8     simulador->resetSize(size_x, size_y);
9     ui->setupUi(this);
10    mImage = new QImage(size_x*2+space_between_draws, size_y, QImage
11                         ::Format_ARGB32_Premultiplied);
12    timerId = startTimer(20);
13
14    ui->plot1->addGraph();
15    ui->plot2->addGraph();
16    ui->plot3->addGraph();
17    ui->plot4->addGraph();
18    ui->plot_MatProps->addGraph();
19    ui->plot1->xAxis->setLabel("tempo\u00b3(s)");
20    ui->plot1->yAxis->setLabel("temperatura\u00b3(K)");
21    ui->plot2->xAxis->setLabel("eixo\u00b3z\u00b3(m)");
22    ui->plot2->yAxis->setLabel("temperatura\u00b3(K)");
23    ui->plot3->xAxis->setLabel("eixo\u00b3x\u00b3(m)");

```

```

23     ui->plot3->yAxis->setLabel("temperatura (K)");
24     ui->plot4->xAxis->setLabel("eixo y (m)");
25     ui->plot4->yAxis->setLabel("temperatura (K)");
26     ui->plot_MatProps->xAxis->setLabel("Temperatura (K)");
27     ui->plot_MatProps->yAxis->setLabel("rho*cp/k");
28
29     for(unsigned int i = 0; i < simulador->getMateriais().size(); i ++
30         )
31         ui->plot_MatProps->addGraph();
32     start_buttons();
33 }
34 MainWindow::~MainWindow() {
35     delete mImage;
36     delete simulador;
37     delete ui;
38 }
39
40 void MainWindow::mousePressEvent(QMouseEvent *event) {
41     if (event->buttons() == Qt::LeftButton){
42         std::string actualMaterial = ui->material_comboBox->
43             currentText().toStdString();
44         double temperature = ui->spinBox_Temperature->value();
45         bool isSource = ui->checkBox_source->checkState();
46         int size = ui->horizontalSliderDrawSize->value();
47         simulador->setActualTemperature(temperature); /**
48             importante para atualizar Tmin/Tmax
49
50         if (drawFormat == "circulo")
51             simulador->grid[currentGrid]->draw_cir(event->pos().x()
52                 -left_margin-size_x-space_between_draws, event->pos()
53                 () .y()-up_margin, size/2, temperature, isSource,
54                 simulador->getMaterial(actualMaterial),
55                 eraserActivated);
56         else
57             simulador->grid[currentGrid]->draw_rec(event->pos().x()
58                 -left_margin-size_x-space_between_draws, event->pos()
59                 () .y()-up_margin, size, temperature, isSource,
60                 simulador->getMaterial(actualMaterial),
61                 eraserActivated);
62     }
63     else if (event->buttons() == Qt::RightButton){

```

```

54         int x = event->pos().x() - left_margin - size_x -
55             space_between_draws;
56         int y = event->pos().y() - up_margin;
57         if (x >= 0 && x < size_x && y >= 0 && y < size_y){
58             studyPoint = QPoint(x, y);
59             studyGrid = currentGrid;
60             time.clear();
61             temperature.clear();
62         }
63     update();
64 }
65
66 void MainWindow::printPosition(){
67     int x = QWidget::mapFromParent(QCursor::pos()).x() -
68         left_margin - size_x - space_between_draws;
69     int y = QWidget::mapFromParent(QCursor::pos()).y() - up_margin;
70     QWidget::mapFromParent(QCursor::pos()).x();
71     std::string txt;
72     if ((x>0) && (x<size_x) && (y>0) && (y<size_y))
73         if (!simulador->grid[currentGrid]->operator()(x, y)->active
74             )
75             txt = "(" + std::to_string(x) + ", " + std::to_string(y
76             ) + ")";
77     else
78         txt = "(" + std::to_string(x) + ", " + std::to_string(y
79             ) + ") T: " +
80             std::to_string(simulador->grid[currentGrid]->
81                 operator()(x, y)->temp) + " K " +
82             simulador-
83             >grid[currentGrid]->operator()(x, y)->
84             material->getName();
85     else
86         txt = "";
87
88     ui->textMousePosition->setText(QString::fromStdString(txt));
89 }
90
91 void MainWindow::printDrawSize(){
92     int size = ui->horizontalSliderDrawSize->value();
93     ui->textDrawSize->setText("Tamanho: " + QString::number(size) + "
94         px/cm" + QString::number(size * simulador->getDelta_x()) + " cm"
95         );

```

```
86 }  
87  
88 void MainWindow::start_buttons(){  
89     // adicionar borda em widget  
90     ui->widget_props->setStyleSheet("border-width:1;"  
91                                         "border-radius:3;"  
92                                         "border-style:solid;"  
93                                         "border-color:rgb(10,10,10)");  
94  
95     ui->widget_simulator_deltas->setStyleSheet("border-width:1;"  
96                                         "border-radius:3;"  
97                                         "border-style:  
98                                         solid;"  
99                                         "border-color:rgb  
100                                         (10,10,10));  
101  
102     ui->widget_drawStyles->setStyleSheet("border-width:1;"  
103                                         "border-radius:3;"  
104                                         "border-style:  
105                                         solid;"  
106                                         "border-color:rgb  
107                                         (10,10,10));  
108  
109     /// remover borda das caixas de texto  
110     ui->textBrowser_3->setFrameStyle(QFrame::NoFrame);  
111     ui->textBrowser_4->setFrameStyle(QFrame::NoFrame);  
112     ui->textBrowser_5->setFrameStyle(QFrame::NoFrame);  
113     ui->textBrowser_6->setFrameStyle(QFrame::NoFrame);  
114     ui->textBrowser_7->setFrameStyle(QFrame::NoFrame);  
115     ui->textBrowser_8->setFrameStyle(QFrame::NoFrame);  
116     ui->textBrowser_9->setFrameStyle(QFrame::NoFrame);  
117     ui->textBrowser_10->setFrameStyle(QFrame::NoFrame);  
118     ui->textBrowser_11->setFrameStyle(QFrame::NoFrame);  
119     ui->textBrowser_12->setFrameStyle(QFrame::NoFrame);
```

```
121     ui->textBrowser_13->setFrameStyle(QFrame::NoFrame);
122     ui->textBrowser_14->setFrameStyle(QFrame::NoFrame);
123     ui->textBrowser_16->setFrameStyle(QFrame::NoFrame);
124     ui->textMousePosition->setFrameStyle(QFrame::NoFrame);
125     ui->textDrawSize->setFrameStyle(QFrame::NoFrame);

126
127     /// spinBox temperatura
128     ui->spinBox_Temperature->setSingleStep(50);
129     ui->spinBox_Temperature->setMaximum(2000);
130     ui->spinBox_Temperature->setValue(300);

131
132     /// texto do grid
133     ui->textGrid->setFrameStyle(QFrame::NoFrame);
134     ui->textGrid->setText(QString::fromStdString(std::to_string(
135         currentGrid)));
136     QFont f = ui->textGrid->font();
137     f.setPixelSize(16);
138     ui->textGrid->setFont(f);
139     ui->textGrid->setAlignment(Qt::AlignCenter);

140     /// lista de materiais
141     std::vector<std::string> materiais = simulador->getMateriais();
142     for (unsigned int i = 0; i < materiais.size(); i++)
143         ui->material_comboBox->addItem(QString::fromStdString(
144             materiais[i]));

145     ui->horizontalSliderDrawSize->setMinimum(2);
146     ui->horizontalSliderDrawSize->setMaximum(150);
147     ui->horizontalSliderDrawSize->setValue(50);

148
149     /// lista de paralelismo
150     ui->parallel_comboBox->addItem("Paralelismo total");
151     ui->parallel_comboBox->addItem("Sem paralelismo");
152     ui->parallel_comboBox->addItem("Paralelismo por grid");

153
154     ui->input_dt->setText(QString::fromStdString(std::to_string(
155         simulador->getDelta_t())));
156     ui->input_dx->setText(QString::fromStdString(std::to_string(
157         simulador->getDelta_x())));
158     ui->input_dz->setText(QString::fromStdString(std::to_string(
159         simulador->getDelta_z())));
```

157

```
158     createWidgetProps();
159 }
160
161 void MainWindow::createWidgetProps(){
162     /// scroll com os materiais para o gráfico
163     std::vector<std::string> materiais = simulador->getMateriais();
164     checkboxes = new QWidget(ui->scrollArea);
165     layout = new QVBoxLayout(checkboxes);
166     myCheckbox.resize(materiais.size());
167     selectedMateriails.resize(materiais.size(), false);
168     QString qss;
169     for(unsigned int i = 0; i < materiais.size(); i++){
170         myCheckbox[i] = new QCheckBox(QString::fromStdString(
171             materiais[i]), checkboxes);
172         qss = QString("background-color:\u00e7%1").arg(simulador->
173             getColor(materiais[i]).name(QColor::HexArgb));
174         myCheckbox[i]->setStyleSheet(qss);
175         layout->addWidget(myCheckbox[i]);
176     }
177     ui->scrollArea->setWidget(checkboxes);
178     makePlotMatProps();
179 }
180
181 void MainWindow::paintEvent(QPaintEvent *e) {
182     QPainter painter(this);
183     *mImage = paint(currentGrid);
184     painter.drawImage(left_margin, up_margin, *mImage);
185     e->accept();
186 }
187
188 QImage MainWindow::paint(int grid) {
189     QImage img = QImage(size_x*2+space_between_draws, size_y, QImage
190         ::Format_ARGB32_Premultiplied);
191
192     /// desenho da temperatura
193     for (int i = 0; i < size_x; i++){
194         for (int k = 0; k < size_y; k++){
195             if (!simulador->grid[grid]->operator()(i, k)->active)
196                 img.setPixelColor(i+size_x+space_between_draws, k,
197                     QColor::fromRgb(255,255,255));
198             else
199                 img.setPixelColor(i+size_x+space_between_draws, k,
```

```

                calcRGB(simulador->grid[grid]->operator()(i, k)
                        ->temp));
196         }
197     }
198
199     if ((studyPoint.x() > 0 && studyPoint.x() < size_x) &&
200         studyPoint.y() > 0 || studyPoint.y() < size_y) && grid ==
201         studyGrid){
202         for(int i = 0; i < size_x; i++)
203             img.setPixelColor(i+size_x+space_between_draws,
204                               studyPoint.y(), QColor::fromRgb(0,0,0));
205         for(int i = 0; i < size_y; i++)
206             img.setPixelColor(studyPoint.x()+size_x+
207                               space_between_draws, i, QColor::fromRgb(0,0,0));
208     }
209
210     /// desenho dos materiais
211     for (int i = 0; i < size_x; i++){
212         for (int k = 0; k < size_y; k++){
213             if (!simulador->grid[grid]->operator()(i, k)->active)
214                 img.setPixelColor(i,k, QColor::fromRgb(255,255,255)
215                                     );
216             else
217                 img.setPixelColor(i,k, simulador->grid[grid]->
218                               operator()(i, k)->material->getColor());
219         }
220     }
221     return img;
222 }
223
224 QColor MainWindow::calcRGB(double temperatura){
225     double maxTemp = simulador->getTmax();
226     double minTemp = simulador->getTmin();
227     return QColor::fromRgb(255, (maxTemp - temperatura)*255/(
228                         maxTemp - minTemp), 0, 255);
229 }
230
231 void MainWindow::runSimulator(){
232     simulador->setDelta_t(std::stod(ui->input_dt->text() .
233                               toStdString()));
234     simulador->setDelta_x(std::stod(ui->input_dx->text() .
235                               toStdString()));

```

```
227     simulador->setDelta_z(std::stod(ui->input_dz->text() .  
228         toStdString()));  
  
229     time_t start_time = std::time(0);  
230     std::string type = ui->parallel_comboBox->currentText().  
231         toStdString();  
232     if(type == "Semparalelismo")  
233         simulador->run_sem_paralelismo();  
234     if(type=="Paralelismoporgrid")  
235         simulador->run_paralelismo_por_grid();  
236     if(type=="Paralelismototal")  
237         simulador->run_paralelismo_total();  
238     time.append((time.size()+1)*simulador->getDelta_t());  
  
239     std::string result = "Time:" + std::to_string(time.size()  
240         -1]) + " u-duracao do solver:" + std::to_string(std::time  
241         (0) - start_time) + " useg";  
242     ui->textBrowser_3->setText(QString::fromStdString(result));  
  
243     update();  
244     makePlot1();  
245     makePlot2();  
246     makePlot3();  
247     makePlot4();  
248 }  
  
249 void MainWindow::timerEvent(QTimerEvent *e){  
250     Q_UNUSED(e);  
251     if (runningSimulator)  
252         runSimulator();  
253     makePlotMatProps();  
254     printPosition();  
255     printDrawSize();  
256 }  
  
257  
258 void MainWindow::on_pushButton_clicked()  
259 {  
260     runningSimulator = runningSimulator?false:true;  
261 }  
  
262  
263 void MainWindow::on_gridDownButton_clicked()  
264 {
```

```
265     currentGrid--;
266     if (currentGrid < 0)
267         currentGrid = 0;
268     /// texto do grid
269     ui->textGrid->setText(QString::fromStdString(std::to_string(
270         currentGrid)));
270     ui->textGrid->setAlignment(Qt::AlignCenter);
271     update();
272 }
273
274 void MainWindow::on_gridUpButton_clicked()
275 {
276     currentGrid++;
277     if (currentGrid > simulador->getNGRIDES()-1)
278         currentGrid = simulador->getNGRIDES()-1;
279     /// texto do grid
280     ui->textGrid->setText(QString::fromStdString(std::to_string(
281         currentGrid)));
282     ui->textGrid->setAlignment(Qt::AlignCenter);
283     update();
284 }
285 void MainWindow::makePlot1(){
286     temperature.append(simulador->grid[studyGrid]->operator()(
287         studyPoint.x(), studyPoint.y())->temp);
288
289     ui->plot1->graph(0)->setData(time,temperature);
290     ui->plot1->xAxis->setRange(time[0], time[time.size()-1]+1);
291     ui->plot1->yAxis->setRange(simulador->getTmin()-50, simulador->
292         getTmax()+50);
293     ui->plot1->replot();
294     ui->plot1->update();
295 }
296 void MainWindow::makePlot2(){
297     QVector<double> temperature_z(simulador->getNGRIDES());
298     QVector<double> labor_z(simulador->getNGRIDES());
299     for (int i = 0; i < simulador->getNGRIDES(); i++){
300         labor_z[i] = simulador->getDelta_z()*(i+1);
301         temperature_z[i] = simulador->grid[i]->operator()(
302             studyPoint.x(), studyPoint.y())->temp;
303     }
```

```
302     ui->plot2->graph(0)->setData(labor_z,temperature_z);
303     ui->plot2->xAxis->setRange(labor_z[0], labor_z[labor_z.size()
304                                     -1]);
305     ui->plot2->yAxis->setRange(simulador->getTmin()-50, simulador->
306                                     getTmax()+50);
307     ui->plot2->replot();
308     ui->plot2->update();
309 }
310 void MainWindow::makePlot3(){
311     QVector<double> temperature_x(size_x);
312     QVector<double> labor_x(size_x);
313     std::ofstream file(dir.absolutePath().toStdString()+"\\\
314                               save_results\\horizontal_"+std::to_string(time[time.size()
315                                     -1]+1)+".dat");
316     for (int i = 0; i < size_x; i++){
317         labor_x[i] = simulador->getDelta_x()*(i+1);
318         temperature_x[i] = simulador->grid[studyGrid]->operator()(i
319                         , studyPoint.y())->temp;
320         file << labor_x[i] << ";" << temperature_x[i] << std::endl
321                         ;
322     }
323     file.close();
324     ui->plot3->graph(0)->setData(labor_x,temperature_x);
325     ui->plot3->xAxis->setRange(labor_x[0], labor_x[size_x-1]);
326     ui->plot3->yAxis->setRange(simulador->getTmin()-50, simulador->
327                                     getTmax()+50);
328     ui->plot3->replot();
329     ui->plot3->update();
330 }
331 void MainWindow::makePlot4(){
332     QVector<double> temperature_y(size_y);
333     QVector<double> labor_y(size_y);
334     std::ofstream file(dir.absolutePath().toStdString()+"\\\
335                               save_results\\vertical"+std::to_string(time[time.size()
336                                     -1]+1)+".dat");
337     for (int i = 0; i < size_y; i++){
338         labor_y[i] = simulador->getDelta_x()*(i+1);
339         temperature_y[i] = simulador->grid[studyGrid]->operator()(i
340                         , studyPoint.x(), i)->temp;
```

```

334         file << labor_y[i] << ";" << temperature_y[i] << std::endl
335         ;
336     }
337     file.close();
338     ui->plot4->graph(0)->setData(labor_y,temperature_y);
339     ui->plot4->xAxis->setRange(labor_y[0], labor_y[size_y-1]);
340     ui->plot4->yAxis->setRange(simulador->getTmin()-50, simulador->
341                                     getTmax()+50);
342     ui->plot4->replot();
343     ui->plot4->update();
344 }
345
346 void MainWindow::makePlotMatProps(){
347     bool changeState = checkChangeMaterialsState();
348     if (!changeState)
349         return;
350     int nPoints = 100;
351     QVector<double> props(nPoints);
352     QVector<double> temperature_x(nPoints);
353     std::vector<std::string> materiais = simulador->getMateriais();
354     double max_props = 700;
355
356     double dT = (simulador->getTmax() - simulador->getTmin())/
357               nPoints-1;
358     for (unsigned int mat = 0; mat < materiais.size(); mat++){
359         if (selectedMaterials[mat]){
360             for (int i = 0; i < nPoints; i++){
361                 temperature_x[i] = dT*i + simulador->getTmin();
362                 props[i] = simulador->getProps(temperature_x[i],
363                                                 materiais[mat]);
364             }
365             ui->plot_MatProps->graph(mat)->setPen(QPen(simulador->
366                                                 getColor(materiais[mat])));
367             ui->plot_MatProps->graph(mat)->setData(temperature_x,props)
368             ;
369             for (int i = 0; i < nPoints; i++)
370                 max_props = max_props < props[i]? props[i] : max_props;
371                 // aqui ajusto o ylabel
372         }else{
373             ui->plot_MatProps->graph(mat)->data()->clear();
374         }
375     }

```

```
369     ui->plot_MatProps->xAxis->setRange(temperature_x[0] ,
370                                         temperature_x[nPoints-1]);
371
372     ui->plot_MatProps->replot();
373     ui->plot_MatProps->update();
374 }
375
376 bool MainWindow::checkChangeMaterialsState(){
377     bool change = false;
378     bool temp = false;
379     for (unsigned int i = 0; i<selectedMateriails.size(); i++){
380         temp = myCheckbox[i]->checkState();
381         if (!(selectedMateriails[i] == temp)){
382             change = true;
383             selectedMateriails[i] = temp;
384         }
385     }
386     return change;
387 }
388
389 void MainWindow::on_actionSave_triggered()
390 {
391     QDir dir; QString path = dir.absolutePath();
392     QString file_name = QFileDialog::getSaveFileName(this, "Save\u00e1\u00e7\u00f5\u00e3o\u00e1\u00e7\u00f5\u00e3o",
393             file", path+ "// save", tr("Dados (*.dat)"));
394     std::string txt = simulador->saveGrid(file_name.toStdString());
395     ui->textBrowser_3->setText(QString::fromStdString(txt));
396
397
398 void MainWindow::on_actionOpen_triggered()
399 {
400     QDir dir; QString path = dir.absolutePath();
401     QString file_name = QFileDialog::getOpenFileName(this, "Open\u00e1\u00e7\u00f5\u00e3o\u00e1\u00e7\u00f5\u00e3o",
402             file", path+ "// save", tr("Dados (*.dat)"));
403     std::string txt = simulador->openGrid(file_name.toStdString());
404     ui->textBrowser_3->setText(QString::fromStdString(txt));
405
406 void MainWindow::on_actionNew_triggered()
407 {
```

```
408     simulador->resetGrid();
409     update();
410 }
411
412
413 void MainWindow::on_actionExport_pdf_triggered()
414 {
415     QString file_name = QFileDialog::getSaveFileName(this, "Save report as", "C://Users", tr("Dados (*.pdf)"));
416     QString txt = save_pdf(file_name);
417     ui->textBrowser_3->setText(txt);
418 }
419
420 void MainWindow::on_actionImport_material_triggered() {
421     QString file_name = QFileDialog::getOpenFileName(this, "Open a file", "C://Users//nicholas//Desktop//ProjetoEngenharia//Projeto-TCC-SimuladorDifusaoTermica//SimuladorTemperatura//materiais", tr("Dados (*.constante, *.correlacao, *.interpolacao")));
422     std::string name = simulador->openMaterial(file_name.toStdString());
423     ui->textBrowser_3->setText(QString::fromStdString("Material " + name + " carregado!"));
424     ui->material_comboBox->addItem(QString::fromStdString(name));
425
426     createWidgetProps();
427 }
428
429 void MainWindow::on_buttonCircle_clicked()
430 {
431
432     ui->widget_buttonCircle->setStyleSheet("border-width: 1px;" "border-radius: 15px;" "border-style: solid;" "border-color: #rgb(255,0,0);");
433
434
435
436     ui->widget_buttonSquare->setStyleSheet("border-width: 0px;" "border-radius: 0px;" "border-style: solid;"
```

```
439                                         "border-color:rgb  
440                                         (255,0,0));  
441     drawFormat = "circulo";  
442 }  
443  
444 void MainWindow::on_buttonSquare_clicked()  
445 {  
446     ui->widget_buttonSquare->setStyleSheet("border-width:1px;  
447                                         border-radius:0px;  
448                                         border-style:solid;  
449                                         border-color:rgb  
450                                         (255,0,0));  
451     ui->widget_buttonCircle->setStyleSheet("border-width:0px;  
452                                         border-radius:15px;  
453                                         border-style:solid;  
454                                         border-color:rgb  
455                                         (255,0,0));  
456 }  
457  
458  
459 void MainWindow::on_buttonEraser_clicked()  
460 {  
461     if (eraserActivated){  
462         ui->widget_eraser->setStyleSheet("border-width:0px;  
463                                         border-radius:0px;  
464                                         border-style:solid;  
465                                         border-color:rgb  
466                                         (255,0,0));  
467         eraserActivated = false;  
468     }  
469     else{  
470         ui->widget_eraser->setStyleSheet("border-width:1px;  
471                                         border-radius:5px;  
472                                         border-style:solid;  
473                                         border-color:rgb  
474                                         (255,170,100));
```

```
473         eraserActivated = true;
474     }
475 }
476
477 QString MainWindow::save_pdf(QString file_name){
478
479     QPdfWriter writer(file_name);
480     writer.setPageSize(QPageSize::A4);
481     writer.setPageMargins(QMargins(30, 30, 30, 30));
482
483     QPrinter pdf;
484     pdf.setOutputFormat(QPrinter::PdfFormat);
485     pdf.setOutputFileName(file_name);
486
487     QPainter painterPDF(this);
488     if (!painterPDF.begin(&pdf))           //Se não conseguir abrir o
489         arquivo PDF ele não executa o resto.
490         return "Erro ao abrir PDF";
491
492     painterPDF.setFont(QFont("Arial", 8));
493     painterPDF.drawText(40,140, "==> PROPRIADES DO GRID<==");
494     painterPDF.drawText(40,160, "Delta_x:" + QString::number(
495         simulador->getDelta_x())+" m");
496     painterPDF.drawText(40,180, "Delta_z:" + QString::number(
497         simulador->getDelta_z())+" m");
498     painterPDF.drawText(40,200, "Delta_t:" + QString::number(
499         simulador->getDelta_t())+" s");
500
501     painterPDF.drawText(40,240, "Largura total horizontal:" +
502         QString::number(simulador->getDelta_x()*size_x)+" m");
503     painterPDF.drawText(40,260, "Largura total vertical:" +
504         QString::number(simulador->getDelta_x()*size_y)+" m");
505     painterPDF.drawText(40,280, "Largura total entre perfis (eixo z)：" +
506         QString::number(simulador->getDelta_z()*simulador->
507             getNGrids())+" m");
508
509     painterPDF.drawText(400,140, "==> PROPRIADES DA SIMULAÇÃO<==");
510
511     painterPDF.drawText(400,160, "Temperatura máxima:" + QString::
```

```

        number(simulador->getTmax())+" K");
506    painterPDF.drawText(400,180, "Temperatura mínima:" + QString::
        number(simulador->getTmin())+" K");
507    painterPDF.drawText(400,200, "Tempo máximo:" + QString::number
        (time[time.size()-1])+" s");
508
509    painterPDF.drawText(400,240, "Tipo de paralelismo:" + ui->
        parallel_comboBox->currentText());
510    painterPDF.drawText(400,260, "Coordenada do ponto de estudo(x,
        y,z):" + QString::number(studyPoint.x()*simulador->
        getDelta_x())+", "+QString::number(studyPoint.y()*simulador->
        getDelta_x())+", "+QString::number(studyGrid*simulador->
        getDelta_z()));
511
512    /// print dos 4 desenhos
513    painterPDF.setPen(Qt::blue);
514    painterPDF.setRenderHint(QPainter::LosslessImageRendering);
515    int startDraw_x = 40;
516    int startDraw_y = 300;
517    int space_draw_x = 40;
518    int space_draw_y = 30;
519    int d = 5;
520    painterPDF.setFont(QFont("Arial", 8));
521
522    painterPDF.drawPixmap(startDraw_x, startDraw_y, (size_x*2+
        space_between_draws)/2, size_y/2, ui->plot1->toPixmap());
523    QRect retangulo5(startDraw_x-d, startDraw_y-d, (size_x*2+
        space_between_draws)/2+2*d, size_y/2+2*d);
524    painterPDF.drawRoundedRect(retangulo5, 2.0, 2.0);
525
526    painterPDF.drawPixmap((size_x*2+space_between_draws)/2+
        startDraw_x+space_draw_x, startDraw_y, (size_x*2+
        space_between_draws)/2, size_y/2, ui->plot2->toPixmap());
527    QRect retangulo6((size_x*2+space_between_draws)/2+startDraw_x+
        space_draw_x-d, startDraw_y-d, (size_x*2+space_between_draws
        )/2+2*d, size_y/2+2*d);
528    painterPDF.drawRoundedRect(retangulo6, 2.0, 2.0);
529
530    painterPDF.drawPixmap(startDraw_x, size_y/2+startDraw_y+
        space_draw_y, (size_x*2+space_between_draws)/2, size_y/2, ui
        ->plot3->toPixmap());
531    QRect retangulo7(startDraw_x-d, size_y/2+startDraw_y+

```

```
    space_draw_y-d, (size_x*2+space_between_draws)/2+2*d, size_y
    /2+2*d);

532    painterPDF.drawRoundedRect(retangulo7, 2.0, 2.0);

533

534    painterPDF.drawPixmap((size_x*2+space_between_draws)/2+
        startDraw_x+space_draw_x, size_y/2+startDraw_y+space_draw_y,
        (size_x*2+space_between_draws)/2, size_y/2, ui->plot4->
        toPixmap());

535    QRect retangulo8((size_x*2+space_between_draws)/2+startDraw_x+
        space_draw_x-d, size_y/2+startDraw_y+space_draw_y-d, (size_x
        *2+space_between_draws)/2+2*d, size_y/2+2*d);

536    painterPDF.drawRoundedRect(retangulo8, 2.0, 2.0);

537

538    painterPDF.drawPixmap(startDraw_x, size_y+startDraw_y+
        space_draw_y*2, (size_x*2+space_between_draws*2), size_y/2,
        ui->widget_props->grab());

539

540

541    startDraw_y = 100;
542    space_draw_y = 50;

543

544    for (int i = 0; i<simulador->getNGRIDs(); i++){
545        if (i%6 == 0){
546            startDraw_y = 100;
547            writer.newPage();
548            pdf.newPage();
549        }
550        if (i%2 == 0){
551            painterPDF.drawText(startDraw_x+size_x/2, startDraw_y-d
                -8, "Grid\u25a1"+QString::number(i));
552            painterPDF.drawPixmap(startDraw_x, startDraw_y, (size_x
                *2+space_between_draws)/2, size_y/2, QPixmap::
                fromImage(paint(i)));
553            QRect retangulo1(startDraw_x-d, startDraw_y-d, (size_x
                *2+space_between_draws)/2+2*d, size_y/2+2*d);
554            painterPDF.drawRoundedRect(retangulo1, 2.0, 2.0);
555        }
556        else {
557            painterPDF.drawText(startDraw_x+space_draw_x+size_x+
                size_x/2+4*d, startDraw_y-d-8, "Grid\u25a1"+QString::
                number(i));
558            painterPDF.drawPixmap((size_x*2+space_between_draws)/2+
```

```
    startDraw_x+space_draw_x , startDraw_y , (size_x*2+
    space_between_draws)/2, size_y/2, QPixmap::fromImage
    (paint(i)));

559     QRect retangulo2((size_x*2+space_between_draws)/2+
    startDraw_x+space_draw_x-d, startDraw_y-d, (size_x
    *2+space_between_draws)/2+2*d, size_y/2+2*d);
560     painterPDF.drawRoundedRect(retangulo2, 2.0, 2.0);
561     startDraw_y+=size_y/2+space_draw_y;
562 }
563 }
564 return "PDF salvo!";
565 }

566

567

568 void MainWindow::on_gridAddGrid_clicked()
569 {
570     simulador->addGrid();
571     currentGrid = simulador->getNGrids()-1;
572
573     /// texto do grid
574     ui->textGrid->setText(QString::fromStdString(std::to_string(
575         currentGrid)));
576     ui->textGrid->setAlignment(Qt::AlignCenter);
577     update();
578 }
579

580 void MainWindow::on_gridDelGrid_clicked()
581 {
582     if (simulador->getNGrids() > 1){
583         simulador->delGrid(currentGrid);
584         currentGrid = currentGrid==0? 0:currentGrid-1;
585     }
586
587     /// texto do grid
588     ui->textGrid->setText(QString::fromStdString(std::to_string(
589         currentGrid)));
590     ui->textGrid->setAlignment(Qt::AlignCenter);
591     update();
592 }
593 void MainWindow::on_button3D_clicked(){
```

```
594     CRender3D *newWindow = new CRender3D(simulador);  
595     // CRender3D *newWindow = new CRender3D();  
596     newWindow->show();  
597 }
```

Apresenta-se na listagem 7.4 o arquivo de cabeçalho da classe CRender3D.

Listing 7.4: Arquivo de implementação da classe CRender3D

```
1 #ifndef CRENDER3D_H  
2 #define CRENDER3D_H  
3  
4 #include <math.h>  
5 #include <QVector>  
6 #include < QPainter >  
7 #include <algorithm>  
8 #include < QMainWindow >  
9 #include < QPaintEvent >  
10 #include < QMouseEvent >  
11  
12 #include "CSimuladorTemperatura.h"  
13  
14 QT_BEGIN_NAMESPACE  
15 namespace Ui { class CRender3D; }  
16 QT_END_NAMESPACE  
17  
18 class CRender3D : public QMainWindow  
19 {  
20     Q_OBJECT  
21  
22 public:  
23     CRender3D( QWidget *parent = nullptr );  
24     CRender3D( CSimuladorTemperatura *simulador, QWidget *parent =  
25                 nullptr );  
26     ~CRender3D();  
27 protected:  
28     void paintEvent( QPaintEvent *event ) override;  
29     QVector3D rotate( QVector3D a );  
30     QColor getRGB( double x, double min, double max );  
31  
32     void timerEvent( QTimerEvent *e ) override;  
33     void keyPressEvent( QKeyEvent *event ) override;  
34     void mousePressEvent( QMouseEvent *e ) override;
```

```

35     void mouseReleaseEvent(QMouseEvent *e) override;
36     void mouseMoveEvent(QMouseEvent *e) override;
37
38     void minimizeAngles();
39     void createPoints();
40     void createTriangles();
41
42     QVector<bool> edges(int i, int j, int g);
43     QVector<QVector3D> createCube(QVector3D point);
44     QVector3D produtoVetorial(QVector3D origem, QVector3D a,
45                               QVector3D b);
46
47 private:
48     int size;
49     int timerId;
50     QImage *mImage;
51     QPoint mousePos;
52     int size_x, size_y;
53     int margin_x = 250;
54     int margin_y = 250;
55     double angle_x = 0.0;
56     double angle_y = 0.0;
57     double angle_z = 0.0;
58     double distance = 1.0;
59     bool mousePress = false;
60     bool corMaterial = false;
61     const float PI = 3.141592;
62     double dx = 1, dy = 1, dz = 2;
63     CSimuladorTemperatura *simulador;
64     QVector<QVector3D> drawCube;
65     QVector<QVector3D> triangles;
66     QVector<QColor> colorsMaterial;
67     QVector<QColor> colorsTemperature;
68     QVector<QVector<QVector3D>> cube;
69     QVector<QVector<bool>> activeEdges;
70
71 };
72 #endif // MAINWINDOW_H

```

Apresenta-se na listagem 7.5 implementação da classe CRender3D.

Listing 7.5: Arquivo de implementação da função `main()`

```
1 #include "CRender3D.h"
2
3 CRender3D::CRender3D(QWidget *parent)
4     : QMainWindow(parent)
5 {
6     //ui->setupUi(this);
7     this->setFixedSize(800,800);
8     this->adjustSize();
9     size_x = 500;
10    mImage = new QImage(size_x, size_y, QImage::Format_ARGB32_Premultiplied);
11    timerId = startTimer(0);
12
13    QVector3D point(0,0,0);
14    cube.push_back(createCube(point));
15
16    createTriangles();
17    drawCube.resize(8);
18    update();
19}
20
21 CRender3D::CRender3D(CSimuladorTemperatura *_simulador, QWidget *
22                         parent)
23     : QMainWindow(parent)
24 {
25     simulador = _simulador;
26     this->setFixedSize(800,800);
27     this->adjustSize();
28     size_x = 500;
29     mImage = new QImage(size_x, size_y, QImage::Format_ARGB32_Premultiplied);
30     timerId = startTimer(0);
31     margin_x = 400;//simulador->getWidth();
32     margin_y = 400;//simulador->getHeight();
33     std::cout<<"criando cubos"<<std::endl;
34     dx = 1;//simulador->getDelta_x();
35     dy = dx;
36     dz = 1*simulador->getDelta_z()/simulador->getDelta_x();
37     double maxTemp = simulador->getTmax();
38     double minTemp = simulador->getTmin();
39     for(int g = 0; g<simulador->getNGRIDs(); g++){
40         for(int i = 0; i < simulador->grid[g]->getWidth(); i++) {
```

```

40         for(int j = 0; j < simulador->grid[g]->getHeight(); j
41             ++){
42             if (simulador->grid[g]->operator()(i,j)->active){
43                 cube.push_back(createCube(QVector3D(i,j,(g+1)*
44                     dz)));
45                 activeEdges.push_back(edges(i,j,g));
46                 colorsMaterial.push_back(simulador->
47                     operator()(i,j)->material->getColor());
48                 colorsTemperature.push_back(getRGB(simulador->
49                     grid[g]->operator()(i,j)->temp, minTemp,
50                     maxTemp));
51             }
52         }
53     }
54
55     std::cout<<"cubos criados"<<std::endl;
56     createTriangles();
57     drawCube.resize(8);
58     update();
59 }
60
61
62
63 QVector<bool> CRender3D::edges(int i, int j, int g){
64     QVector<bool> actives(12, true);
65     int max_i = simulador->getWidth()-1;
66     int max_j = simulador->getHeight()-1;
67     int max_g = simulador->grid.size()-1;
68
69
70     if (g > 0){
71         if (simulador->grid[g-1]->operator()(i,j)->active){
72             actives[0] = false;
73             actives[1] = false;
74         }
75     }
76     if (i < max_i){

```

```
77         if (simulador->grid[g]->operator()(i+1,j)->active){
78             actives[2] = false;
79             actives[3] = false;
80         }
81     }
82     if (i > 0){
83         if (simulador->grid[g]->operator()(i-1,j)->active){
84             actives[4] = false;
85             actives[5] = false;
86         }
87     }
88     if (j < max_j){
89         if (simulador->grid[g]->operator()(i,j+1)->active){
90             actives[6] = false;
91             actives[7] = false;
92         }
93     }
94     if (g < max_g){
95         if (simulador->grid[g+1]->operator()(i,j)->active){
96             actives[8] = false;
97             actives[9] = false;
98         }
99     }
100    if (j > 0){
101        if (simulador->grid[g]->operator()(i,j-1)->active){
102            actives[10] = false;
103            actives[11] = false;
104        }
105    }
106    return actives;
107}
108
109void CRender3D::createTriangles(){
110    triangles.resize(12);
111    triangles[0] = QVector3D( 0,1,2);
112    triangles[1] = QVector3D( 4,2,1);
113
114    triangles[2] = QVector3D( 1,5,4);
115    triangles[3] = QVector3D( 7,4,5);
116
117    triangles[4] = QVector3D( 6,3,2);
118    triangles[5] = QVector3D( 0,2,3);
```

```
119
120     triangles[6] = QVector3D( 4,7,2);
121     triangles[7] = QVector3D( 6,2,7);
122
123     triangles[8] = QVector3D( 6,7,3);
124     triangles[9] = QVector3D( 5,3,7);
125
126     triangles[10] = QVector3D( 1,0,5);
127     triangles[11] = QVector3D( 3,5,0);
128 }
129
130 QVector<QVector3D> CRender3D::createCube(QVector3D point){
131     double x = point.x(), y = point.y(), z = point.z();
132
133     QVector<QVector3D> cube(8);
134     cube[0] = QVector3D( x-dx/2.0, y-dy/2.0, z-dz/2.0);
135     cube[1] = QVector3D( x+dx/2.0, y-dy/2.0, z-dz/2.0);
136     cube[2] = QVector3D( x-dx/2.0, y+dy/2.0, z-dz/2.0);
137     cube[3] = QVector3D( x-dx/2.0, y-dy/2.0, z+dz/2.0);
138     cube[4] = QVector3D( x+dx/2.0, y+dy/2.0, z-dz/2.0);
139     cube[5] = QVector3D( x+dx/2.0, y-dy/2.0, z+dz/2.0);
140     cube[6] = QVector3D( x-dx/2.0, y+dy/2.0, z+dz/2.0);
141     cube[7] = QVector3D( x+dx/2.0, y+dy/2.0, z+dz/2.0);
142     return cube;
143 }
144
145 void CRender3D::keyPressEvent(QKeyEvent *event){
146     if (event->key() == Qt::Key_Up){
147         margin_y+=30.0f;
148     }
149     else if (event->key() == Qt::Key_Down){
150         margin_y-=30.0f;
151     }
152     else if (event->key() == Qt::Key_Left){
153         margin_x+=30.0f;
154     }
155     else if (event->key() == Qt::Key_Right){
156         margin_x-=30.0f;
157     }
158     else if (event->key() == Qt::Key_PageUp){
159         distance*=1.1;
160     }
```

```
161     else if (event->key() == Qt::Key_PageDown){
162         distance *=0.9;
163     }
164     else if (event->key() == Qt::Key_W){
165         angle_x -=0.1;
166     }
167     else if (event->key() == Qt::Key_S){
168         angle_x +=0.1;
169     }
170     else if (event->key() == Qt::Key_D){
171         angle_y -=0.1;
172     }
173     else if (event->key() == Qt::Key_A){
174         angle_y +=0.1;
175     }
176     else if (event->key() == Qt::Key_Space){
177         corMaterial = corMaterial ? false:true;
178     }
179     update();
180 }
181
182 void CRender3D::mousePressEvent(QMouseEvent *e){
183     mousePos = e->pos();
184     mousePress = true;
185     update();
186 }
187 void CRender3D::mouseReleaseEvent(QMouseEvent *e){
188     angle_y -= (e->pos().x() - mousePos.x());
189     angle_x -= (e->pos().y() - mousePos.y());
190     mousePress = false;
191     update();
192 }
193
194 void CRender3D::mouseMoveEvent(QMouseEvent *e){
195     if (mousePress){
196         angle_y -= (e->pos().x() - mousePos.x())/60.0;
197         angle_x += (e->pos().y() - mousePos.y())/60.0;
198         mousePos = e->pos();
199     }
200     update();
201 }
```

```
203 void CRender3D::minimizeAngles(){
204     if(angle_x > 2.0f*PI)
205         angle_x = 0.0f;
206     if(angle_x < 0.0f)
207         angle_x = 2.0f*PI;
208
209     if(angle_y > 2.0f*PI)
210         angle_y = 0.0f;
211     if(angle_y < 0.0f)
212         angle_y = 2.0f*PI;
213
214     if(angle_z > 2.0f*PI)
215         angle_z = 0.0f;
216     if(angle_z < 0.0f)
217         angle_z = 2.0f*PI;
218 }
219
220 void CRender3D::paintEvent(QPaintEvent *e) {
221
222     //QPolygon triangle;
223
224     QPainter painter(this);
225     minimizeAngles();
226     QVector<QPolygon> triangulosDesenho;
227     QVector<QColor> coresDesenho;
228     QVector<std::pair<int, double>> pos_norm;
229     QVector<QColor> color;
230     if (corMaterial)
231         color = colorsMaterial;
232     else
233         color = colorsTemperature;
234     double prodVet;
235     int a, b, c;
236     int count = 0;
237     for(int cb = 0; cb < cube.size(); cb++){
238         for(int i = 0; i < 8; i++)
239             drawCube[i] = rotate(cube[cb][i]);
240
241         for(int r = 0; r < 12; r++){
242             if(activeEdges[cb][r]){
243                 a = triangles[r].x();
244                 b = triangles[r].y();
```

```

245         c = triangles[r].z();
246         prodVet = produtoVetorial(drawCube[a], drawCube[b],
247                                     drawCube[c]).z();
248         if(prodVet > 0){
249             pos_norm.push_back(std::pair(count, prodVet));
250             count++;
251             if(r == 0 || r == 1 || r == 8 || r == 9) /**
252                 fronteiras de g
253                 coresDesenho.push_back(QColor(color[cb].red()
254                                             , color[cb].green(), color[cb].blue(),
255                                             255));
256             else
257                 coresDesenho.push_back(QColor(QColor(color[
258                     cb].red()*0.6, color[cb].green()*0.6,
259                     color[cb].blue()*0.6, 255)));
260
261             QPolygon pol;
262             pol << QPoint(drawCube[a].x(),drawCube[a].y())
263                 << QPoint(drawCube[b].x(),drawCube[b].y())
264                 << QPoint(drawCube[c].x(),drawCube[c].y());
265             triangulosDesenho.push_back(pol);
266         }
267     }
268 }
269
270 /// organizo conforme a profundidade
271 std::sort(pos_norm.begin(), pos_norm.end(), [](auto &left, auto
272 &right) {
273     return left.second > right.second;
274 });
275
276 /// desenho na tela
277 int pos;
278 painter.setPen(QColor(0,0,0,0));
279 for(int i = 0; i<triangulosDesenho.size(); i++){
280     pos = pos_norm[i].first;
281     painter.setBrush(coresDesenho[pos]);
282     painter.drawPolygon(triangulosDesenho[pos]);
283 }
284
285 painter.drawImage(0,0, *mImage);

```

```
280     e->accept();  
281 }  
282  
283 QColor CRender3D::getRGB(double x, double min, double max){  
284     return QColor::fromRgb(255, (max - x)*255/(max - min), 0, 255);  
285 }  
286  
287 void CRender3D::timerEvent(QTimerEvent *e){  
288     //angle_x-=0.05;  
289     //angle_y+=0.05;  
290     update();  
291     Q_UNUSED(e);  
292 }  
293  
294 QVector3D CRender3D::rotate(QVector3D a){  
295     double A[3] = {a.x(), a.y(), a.z()};  
296     double rotation[3][3];  
297     double result[3] = {0,0,0};  
298  
299     /// rotation in x  
300     rotation[0][0] = cos(angle_z)*cos(angle_y);  
301     rotation[0][1] = cos(angle_z)*sin(angle_y)*sin(angle_x)-sin(  
            angle_z)*cos(angle_x);  
302     rotation[0][2] = cos(angle_z)*sin(angle_y)*cos(angle_x)+sin(  
            angle_z)*sin(angle_x);  
303  
304     rotation[1][0] = sin(angle_z)*cos(angle_y);  
305     rotation[1][1] = sin(angle_z)*sin(angle_y)*sin(angle_x)+cos(  
            angle_z)*cos(angle_x);  
306     rotation[1][2] = sin(angle_z)*sin(angle_y)*cos(angle_x)-cos(  
            angle_z)*sin(angle_x);  
307  
308     rotation[2][0] = -sin(angle_y);  
309     rotation[2][1] = cos(angle_y)*sin(angle_x);  
310     rotation[2][2] = cos(angle_y)*cos(angle_x);  
311  
312     for(int i = 0;i<3; i++)  
313         for(int j = 0;j<3; j++)  
314             result[i]+=A[j]*rotation[i][j];  
315  
316     return QVector3D((result[0]+margin_x-200)*distance,(result[1]+  
            margin_y-200)*distance,result[2]*distance);
```

```

317 }
318
319 QVector3D CRender3D::produtoVetorial(QVector3D origem, QVector3D a,
320     QVector3D b){
321     QVector3D ax = a - origem;
322     QVector3D bx = b - origem;
323     return QVector3D(ax.y()*bx.z()-ax.z()*bx.y(), -ax.x()*bx.z()+ax
324         .z()*bx.x(), ax.x()*bx.y()-ax.y()*bx.x());
325 }
```

Apresenta-se na listagem 7.6 o arquivo de cabeçalho da classe CSimuladorTemperatura.

Listing 7.6: Arquivo de implementação da classe CSimuladorTemperatura

```

1 #ifndef CSIMULADORTEMPERATURA_H
2 #define CSIMULADORTEMPERATURA_H
3
4 #include <map>
5 #include <QDir>
6 #include <omp.h>
7 #include <QPoint>
8 #include <fstream>
9 #include <iomanip>
10 #include <QDirIterator>
11
12 #include "CGrid.h"
13 #include "CMaterial.h"
14 #include "CMaterialCorrelacao.h"
15 #include "CMaterialInterpolacao.h"
16
17 class CSimuladorTemperatura {
18 private:
19     //int parallel = 0;
20     QDir dir;
21     int MAX_THREADS = omp_get_max_threads() - 4;
22     int width, height;
23     bool materialPropertiesStatus = true;
24     int NGRIDS = 1;
25     const double MIN_ERRO = 0.05;
26     const int MAX_ITERATION = 500, MIN_ITERATION = 50;
27     double delta_x = 2.6e-4, delta_t = 5.0e-1, delta_z = 0.05;
28
29     double Tmax = 400, Tmin = 300;
30 }
```

```
31     double actualTemperature = 300;
32     double actual_time = 0.0;
33     std::map<std::string, CMaterial*> materiais;
34     std::vector<std::string> name_materiais;
35
36 public:
37     std::vector<CGrid*> grid;
38 public:
39     // ----- FUNCOES DE CRIACAO -----
40     CSimuladorTemperatura() { createListOfMaterials(); }
41
42     void resetSize(int width, int height);
43     void resetGrid();
44
45     void createListOfMaterials();
46     CMaterial* chooseMaterialType(std::string name, std::string type);
47
48     void addGrid();
49     void delGrid(int _grid);
50
51     // ----- FUNCOES DO SOLVER -----
52     void run_sem_paralelismo();
53     void run_paralelismo_por_grid();
54     void run_paralelismo_total();
55     void solverByGrid(int g);
56     void solverByThread(int thread_num);
57     double calculatePointIteration(int x, int y, int g);
58
59     std::string saveGrid(std::string nameFile);
60     std::string openGrid(std::string nameFile);
61     std::string openMaterial(std::string nameFile);
62
63     // ----- FUNCOES SET -----
64     void setActualTemperature(double newTemperature);
65     void changeMaterialPropertiesStatus();
66     void setDelta_t(double _delta_t) { delta_t = _delta_t; }
67     void setDelta_x(double _delta_x) { delta_x = _delta_x; }
68     void setDelta_z(double _delta_z) { delta_z = _delta_z; }
69
70     // ----- FUNCOES GET -----
71     int getWidth(){return width;}
```

```

72     int getHeight(){return height;}
73     double getProps(double temperature, std::string material);
74     QColor getColor(std::string material);
75     int getNGrids() { return NGRIDS; }
76     bool getMaterialStatus() { return materialPropertiesStatus; }
77     double maxTemp();
78     double minTemp();
79     double get_ActualTemperature() { return actualTemperature; }

80
81     double getMaxT() { return Tmax; }
82     double getMinT() { return Tmin; }

83
84     double getDelta_t() { return delta_t; }
85     double getDelta_x() { return delta_x; }
86     double getDelta_z() { return delta_z; }
87     double getTime() { return actual_time; }

88
89     CMaterial* getMaterial(std::string mat) { return materiais[mat];
90         }
91
92     std::vector<std::string> getMateriais() { return name_materiais;
93         }
94 };
95 #endif

```

Apresenta-se na listagem 7.7 implementação da classe CSimuladorTemperatura.

Listing 7.7: Arquivo de implementação da função main()

```

1 #include "CSimuladorTemperatura.h"
2
3 void CSimuladorTemperatura::resetSize(int width, int height) {
4     grid.resize(NGRIDS);
5     this->width = width;
6     this->height = height;
7     for (int i = 0; i < NGRIDS; i++)
8         grid[i] = new CGrid(width, height, 0.0);
9 }
10
11 void CSimuladorTemperatura::resetGrid() {
12     for (int i = 0; i < NGRIDS; i++)
13         grid[i]->resetGrid(0.0);
14 }
15

```

```

16 void CSimuladorTemperatura::createListOfMaterials() {
17     /**
18      std::string matName;
19      QDirIterator it(dir.absolutePath()+"//materiais", {"*.
20          correlacao", "*.constante", "*.interpolacao"}, QDir::Files,
21          QDirIterator::Subdirectories);
22      while (it.hasNext()) {
23          it.next();
24          matName = it.fileName().toString();
25          QFileInfo fi(it.fileName());
26          std::string type = fi.suffix().toString();
27          materiais[matName] = chooseMaterialType(matName, type);
28      }
29      for(auto const& imap: materiais)
30          name_materiais.push_back(imap.first);
31 }
32
33 CMaterial* CSimuladorTemperatura::chooseMaterialType(std::string
34     name, std::string type){
35     std::ifstream file(dir.absolutePath().toString()+"/materiais
36         //"+name);
37
38     if (type == "correlacao" || type == "constante")
39         return new CMaterialCorrelacao(name);
40     else if (type == "interpolacao")
41         return new CMaterialInterpolacao(name);
42 }
43
44 void CSimuladorTemperatura::addGrid(){
45     NGRIDS++;
46     grid.push_back(new CGrid(width, height, 0.0));
47 }
48
49 void CSimuladorTemperatura::delGrid(int _grid){
50     NGRIDS--;
51     grid.erase(grid.begin() + _grid);
52 }
53
54 std::string CSimuladorTemperatura::openMaterial(std::string
55     nameFile){
56     std::ifstream file(nameFile);
57 }
```

```
53     std::string type;
54     std::string name;
55     std::getline(file, type);
56     std::getline(file, name);
57     std::cout<<name<<std::endl;
58
59     file.close();
60     if (type == "correlacao")
61         materiais[name] = new CMaterialCorrelacao(nameFile);
62     else
63         materiais[name] = new CMaterialInterpolacao(nameFile);
64     name_materiais.push_back(name);
65     return name;
66 }
67
68
69 void CSimuladorTemperatura::run_sem_paralelismo() {
70     for (int g = 0; g < NGRIDS; g++){
71         grid[g]->startIteration();
72         solverByGrid(g);
73     }
74 }
75
76 void CSimuladorTemperatura::run_parallelismo_por_grid() {
77     omp_set_num_threads(NGRIDES);
78     #pragma omp parallel
79     {
80         grid[omp_get_thread_num()]->startIteration();
81         solverByGrid(omp_get_thread_num());
82     }
83 }
84
85 void CSimuladorTemperatura::run_parallelismo_total() {
86     for (int g=0;g<NGRIDES;g++)
87         grid[g]->startIteration();
88
89     omp_set_num_threads(MAX_THREADS);
90     #pragma omp parallel
91     {
92         solverByThread(omp_get_thread_num());
93     }
94     for (int g = 0; g < NGRIDS; g++)
```

```
95         grid[g]->updateSolver();
96     }
97
98 void CSimuladorTemperatura::solverByGrid(int g) {
99     double erro = 1, _erro;
100    int iter = 0;
101    while (erro < MIN_ERRO || iter <= MAX_ITERATION) {
102        grid[g]->updateIteration(); // atualizo temp_nu para
103        calcular o erro da iteracao
104        for (int i = 0; i < grid[g]->getWidth(); i++)
105            for (int k = 0; k < grid[g]->getHeight(); k++)
106                calculatePointIteration(i, k, g);
107        _erro = grid[g]->maxErroIteration();
108        erro = erro < _erro ? _erro : erro;
109        iter++;
110    }
111    grid[g]->updateSolver();
112
113 void CSimuladorTemperatura::solverByThread(int thread_num) {
114     double erro = 0, _erro;
115     int iter = 0;
116     int x, y;
117     do {
118         for (int g = 0; g < NGRIDS; g++) {
119             for (int i = thread_num; i < grid[g]->getSize(); i+=
120                 MAX_THREADS) {
121                 x = i % grid[g]->getWidth();
122                 y = i / grid[g]->getWidth();
123
124                 (*grid[g])(x, y)->temp_nu = (*grid[g])(x, y)->
125                     temp_nup1;
126                 _erro = calculatePointIteration(x, y, g);
127                 erro = erro < _erro ? _erro : erro;
128             }
129             iter++;
130             if (erro < MIN_ERRO && iter > MIN_ITERATION)
131                 break;
132         } while (iter < MAX_ITERATION);
133         std::cout<<"iteracoes:"<< iter << "erro:"<< erro << std
134             ::endl;
```

```
133 }
134
135 double CSimuladorTemperatura::calculatePointIteration(int x, int y,
136     int g) {
137     if (!(*grid[g])(x, y)->active)
138         return 0.0;
139     if ((*grid[g])(x, y)->source)
140         return 0.0;
141     float n_x = 0;
142     float n_z = 0;
143     double inf = .0, sup = .0, esq = .0, dir = .0, cima = .0, baixo
144         = .0;
145     double thermalConstant;
146
147     if (y - 1 > 0) {
148         if ((*grid[g])(x, y - 1)->active) {
149             n_x++;
150             inf = (*grid[g])(x, y - 1)->temp_nup1*delta_z*delta_z;
151         }
152         if (y + 1 < grid[g]->getHeight()) {
153             if ((*grid[g])(x, y + 1)->active) {
154                 n_x++;
155                 sup = (*grid[g])(x, y + 1)->temp_nup1 * delta_z*delta_z
156                     ;
157             }
158         }
159         if (x - 1 > 0) {
160             if ((*grid[g])(x - 1, y)->active) {
161                 n_x++;
162                 esq = (*grid[g])(x - 1, y)->temp_nup1 * delta_z*
163                     delta_z;
164             }
165         }
166         if (x + 1 < grid[g]->getWidth()) {
167             if ((*grid[g])(x + 1, y)->active) {
168                 n_x++;
169                 dir = (*grid[g])(x + 1, y)->temp_nup1 * delta_z*
                     delta_z;
```

```

170         }
171     }
172
173     if ( g < NGRIDS -1) {
174         if (grid[g + 1]->operator()(x, y)->active) {
175             n_z++;
176             cima = (*grid[g + 1])(x, y)->temp_nup1*delta_x*delta_x;
177         }
178     }
179
180     if (g > 0) {
181         if (grid[g - 1]->operator()(x, y)->active) {
182             n_z++;
183             baixo = (*grid[g - 1])(x, y)->temp_nup1 * delta_x*
184                 delta_x;
185         }
186     }
187     thermalConstant = (*grid[g])(x, y)->material->getThermalConst
188     ((*grid[g])(x, y)->temp_nup1);
189
190     (*grid[g])(x, y)->temp_nup1 = (thermalConstant * (*grid[g])(x,
191     y)->temp*delta_x*delta_x*delta_z*delta_z/delta_t + inf + sup
192     + esq + dir + cima + baixo) / (n_x*delta_z*delta_z + n_z*
193     delta_x*delta_x + thermalConstant*delta_x*delta_x*delta_z*
194     delta_z/delta_t);
195
196     return (*grid[g])(x, y)->temp_nup1 - (*grid[g])(x, y)->temp_nu;
197 }
198
199 std::string CSimuladorTemperatura::saveGrid(std::string nameFile) {
200     std::ofstream file(nameFile);
201     int sizeGrid = grid[0]->getSize();
202     file << NGRIDS << "\n";
203     for (int g = 0; g < NGRIDS; g++) {
204         for (int i = 0; i < sizeGrid; i++) {
205             if ((*grid[g])[i]->active){
206                 file << i << " " << g << " ";
207                 file << (*grid[g])[i]->temp << " ";
208                 file << (*grid[g])[i]->active << " ";
209                 file << (*grid[g])[i]->source << " ";
210                 file << (*grid[g])[i]->material->getName() << "\n";
211             }
212         }
213     }
214 }
```

```
206         }
207     }
208     file.close();
209     return "Arquivo salvo!";
210 }
211
212 std::string CSimuladorTemperatura::openGrid(std::string nameFile) {
213
214     std::ifstream file(nameFile);
215
216     std::string _name;
217     int i, g;
218     double _temperature;
219     int _active, _source;
220     std::string _strGrids;
221     std::getline(file, _strGrids);
222
223     NGRIDS = std::stoi(_strGrids);
224     grid.resize(NGRIDS);
225     for(int gg = 0; gg<NGRIDS; gg++)
226         grid[gg] = new CGrid(width, height, 0.0);
227     while(file >> i >> g >> _temperature >> _active >> _source >>
228           _name){
229         grid[g]->draw(i, _temperature, _active, _source, _name)
230             ;
231         Tmax = Tmax < _temperature ? _temperature : Tmax;
232     }
233
234     file.close();
235     return "Arquivo carregado!";
236 }
237
238 void CSimuladorTemperatura::setActualTemperature(double
239   newTemperature) {
240     if (newTemperature > Tmax)
241         Tmax = newTemperature;
242     if (newTemperature < Tmin)
243         Tmin = newTemperature;
244     actualTemperature = newTemperature;
245 }
246
247 void CSimuladorTemperatura::changeMaterialPropertiesStatus() {
```

```

245     materialPropertiesStatus = materialPropertiesStatus ? false :
246         true;
247
248 double CSimuladorTemperatura::getProps(double temperature, std::
249     string material){
250     return materiais[material]->getThermalConst(temperature);
251 }
252
253 QColor CSimuladorTemperatura::getColor(std::string material){
254     return materiais[material]->getColor();
255 }
256
257 double CSimuladorTemperatura::maxTemp() {
258     double maxErro = 0;
259     double tempErro = 0;
260     for (int i = 0; i < NGRIDS; i++) {
261         tempErro = grid[i]->maxTemp();
262         maxErro = maxErro < tempErro ? tempErro : maxErro;
263     }
264     return maxErro;
265 }
266
267 double CSimuladorTemperatura::minTemp() {
268     double minErro = 0;
269     double tempErro = 0;
270     for (int i = 0; i < NGRIDS; i++) {
271         tempErro = grid[i]->minTemp();
272         minErro = minErro > tempErro ? tempErro : minErro;
273     }
274 }
```

Apresenta-se na listagem 7.8 o arquivo de cabeçalho da classe CGrid.

Listing 7.8: Arquivo de implementação da classe CGrid

```

1 #ifndef CGRID_HPP
2 #define CGRID_HPP
3
4 #include <vector>
5 #include <string>
6 #include "CCell.h"
7 #include <iostream>
```

```
8 #include "CMaterialCorrelacao.h"
9
10 class CGrid {
11 private:
12     int width, height;
13     std::vector<CCell> grid;
14 public:
15     CGrid() {
16         width = 0;
17         height = 0;
18     }
19
20     CGrid(int _width, int _height) : width{_width}, height{_height}
21     {
22         grid.resize(width * height);
23     }
24
25     CGrid(int _width, int _height, double temperature) {
26         resetSize(_width, _height, temperature);
27     }
28
29     void resetGrid(double temperature);
30
31     void resetSize(int _width, int _height, double temperature);
32
33     void draw_rec(int x, int y, double size, double temperature,
34                   bool isSourceActive, CMaterial* _material, bool eraser);
35     void draw_cir(int x, int y, double size, double temperature,
36                   bool isSourceActive, CMaterial* _material, bool eraser);
37     void draw(int x, double temperature, bool active, bool isSource
38               , std::string _material);
39
40     int getSize() { return width * height; }
41
42     void updateIteration();
43     void updateSolver();
44     void startIteration();
45     double maxErroIteration();
46
47     int getWidth() { return width; }
48     int getHeight() { return height; }
49     double getTemp(int position) { return grid[position].temp_nup1;
```

```

        }

46
47     double maxTemp();
48     double minTemp();

49
50     bool isActive(int x){ return grid[x].active;}
51     CCell* operator () (int x, int y) { return &grid[y * width + x
52     ]; }
53
54     CCell* operator [] (int x) { return &grid[x]; }
55
56 };
57
58 #endif

```

Apresenta-se na listagem 7.9 implementação da classe CGrid.

Listing 7.9: Arquivo de implementação da função main()

```

1 #include "CGrid.h"
2
3 void CGrid::resetSize(int _width, int _height, double temperature)
4 {
5     width = _width;
6     height = _height;
7     grid.resize(width * height);
8     for (int i = 0; i < width * height; i++)
9         grid[i].temp = temperature;
10
11 void CGrid::resetGrid(double temperature) {
12     for (int i = 0; i < width * height; i++) {
13         grid[i].active = false;
14         grid[i].active = false;
15         grid[i].source = false;
16         grid[i].temp = temperature;
17         grid[i].temp_nup1 = temperature;
18         grid[i].material = new CMaterial();
19     }
20 }
21
22 void CGrid::draw_rec(int x, int y, double size, double _temperature
23 , bool isSourceActive, CMaterial* _material, bool eraser) {
24     int start_x = (x - size / 2 >= 0) ? x - size / 2 : 0;
25     int start_y = (y - size / 2 >= 0) ? y - size / 2 : 0;

```

```

        size;
26     int max_y    = (y + size / 2 >= height) ? height : y - size/2 +
        size;
27     double temperatura = eraser?0:_temperature;
28
29     for (int i = start_x; i < max_x; i++){
30         for (int k = start_y; k < max_y; k++) {
31             grid[k * width + i].active = !eraser;
32             grid[k * width + i].temp = temperatura;
33             grid[k * width + i].source = isSourceActive;
34             grid[k * width + i].material = _material;
35         }
36     }
37 }
38
39 void CGrid::draw_cir(int x, int y, double radius, double
        _temperature, bool isSourceActive, CMaterial* _material, bool
        eraser) {
40     // vou montar um quadrado, e analisar se o cada ponto dessa
        regiao faz parte do circulo
41     int start_x = (x - (int)radius >= 0) ? ((int)x - (int)radius) :
        0;
42     int start_y = (y - (int)radius >= 0) ? ((int)y - (int)radius) :
        0;
43     int max_x    = (x + (int)radius >= width) ? width : ((int)x +
        (int)radius);
44     int max_y    = (y + (int)radius >= height) ? height : ((int)y +
        (int)radius);
45     double temperatura = eraser?0:_temperature;
46
47     for (int i = start_x; i < max_x; i++) {
48         for (int k = start_y; k < max_y; k++) {
49             if (((i*1.0 - x) * (i*1.0 - x) + (k*1.0 - y) * (k*1.0 -
                y)) < radius * radius) {
50                 grid[k * width + i].active = !eraser;
51                 grid[k * width + i].temp = temperatura;
52                 grid[k * width + i].source = isSourceActive;
53                 grid[k * width + i].material = _material;
54             }
55         }
56     }
57 }
```

```
58
59 void CGrid::draw(int x, double _temperature, bool active, bool
60     isSource, std::string _material) {
61     grid[x].temp = _temperature;
62     grid[x].active = active;
63     grid[x].source = isSource;
64     if (active)
65         grid[x].material = new CMaterialCorrelacao(_material);
66     else
67         grid[x].material = new CMaterial();
68 }
69
70 void CGrid::updateIteration() {
71     for (int i = 0; i < width * height; i++)
72         grid[i].temp_nu = grid[i].temp_nup1;
73 }
74
75 void CGrid::updateSolver() {
76     for (int i = 0; i < width * height; i++)
77         grid[i].temp = grid[i].temp_nup1;
78 }
79
80 double CGrid::maxErroIteration() {
81     double erro = 0.0;
82     double erro_posicao = 0.0;
83     for (int i = 0; i < width * height; i++) {
84         erro_posicao = grid[i].temp_nup1 - grid[i].temp_nu;
85         erro = abs(erro_posicao) > erro ? erro_posicao : erro;
86     }
87     return erro;
88 }
89
90 void CGrid::startIteration() {
91     for (int i = 0; i < width * height; i++)
92         grid[i].temp_nup1 = grid[i].temp;
93 }
94
95 double CGrid::maxTemp() {
96     double maxTemp = 0;
97     for (int i = 0; i < width * height; i++)
98         maxTemp = maxTemp < grid[i].temp ? grid[i].temp : maxTemp;
99     return maxTemp;
```

```

99 }
100
101 double CGrid::minTemp() {
102     double minTemp = 1000000;
103     for (int i = 0; i < width * height; i++)
104         minTemp = minTemp > grid[i].temp ? grid[i].temp : minTemp;
105     return minTemp;
106 }
```

Apresenta-se na listagem 7.10 o arquivo de cabeçalho da classe CCell.

Listing 7.10: Arquivo de implementação da classe CCell

```

1 ifndef CCELL_HPP
2 define CCELL_HPP
3
4 include <iostream>
5 include "CMaterial.h"
6
7 class CCell {
8 public:
9     bool active = false;
10    bool source = false;
11    double temp = 0;
12    double temp_nu = 0;
13    double temp_nup1 = 0;
14
15    CMaterial *material;
16    friend std::ostream& operator << (std::ostream& os, const CCell
17        & cell) { return os << cell.temp; }
18};
```

1 #endif

Apresenta-se na listagem 7.11 implementação da classe CCell.

Listing 7.11: Arquivo de implementação da função main()

```
#include "CCell.h"
```

Apresenta-se na listagem 7.12 o arquivo de cabeçalho da classe CMaterial.

Listing 7.12: Arquivo de implementação da classe CMaterial

```

1 ifndef CMATERIAL_HPP
2 define CMATERIAL_HPP
3
4 include <string>
```

```

5 #include <QColor>
6 #include <iostream>
7
8 class CMaterial {
9 public:
10     CMaterial(){}
11     CMaterial(std::string _name) {name = _name;}
12     virtual double getThermalConst(double T) {return 0.0*T;}
13
14     virtual QColor getColor()          { return QColor(0,0,0); }
15     virtual std::string getName()     { return name; }
16
17 protected:
18     std::string name;
19     QColor color;
20 };
21#endif

```

Apresenta-se na listagem 7.13 implementação da classe CMaterial.

Listing 7.13: Arquivo de implementação da função main()

```
#include "CMaterial.h"
```

Apresenta-se na listagem 7.14 o arquivo de cabeçalho da classe CMaterialCorrelacao.

Listing 7.14: Arquivo de implementação da classe CMaterialCorrelacao

```

1 ifndef CMATERIALCORRELACAO_H
2 define CMATERIALCORRELACAO_H
3
4 #include <QDir>
5 #include <string>
6 #include <QColor>
7 #include <fstream>
8 #include <iostream>
9
10 #include "CMaterial.h"
11
12 class CMaterialCorrelacao:public CMaterial {
13 public:
14     CMaterialCorrelacao(std::string fileName);
15     double getThermalConst(double T);
16
17     QColor getColor()          { return color; }
18     std::string getName()     { return name; }

```

```

19
20 protected:
21     std::string name;
22     QColor color;
23
24     double C0_rho, C1_rho;
25     double C0_cp, C1_cp, C2_cp;
26     double C0_k, C1_k, C2_k;
27 };
28 #endif

```

Apresenta-se na listagem 7.15 implementação da classe CMaterialCorrelacao.

Listing 7.15: Arquivo de implementação da função main()

```

1 #include "CMaterialCorrelacao.h"
2
3 CMaterialCorrelacao::CMaterialCorrelacao(std::string fileName){
4     std::string str_temp;
5     int r, g, b, alpha;
6     name = fileName;
7
8     QDir dir; std::string path = dir.absolutePath().toStdString();
9     std::ifstream file(path+"/materiais//"+fileName);
10    if (file.is_open()){
11        file >> str_temp; file >> r; file >> g; file >> b; file >>
12            alpha;
13        color = QColor(r, g, b, alpha);
14        file >> str_temp; file >> cp;
15        file >> str_temp; file >> rho;
16        file >> str_temp; file >> str_temp; // texto explicando a
17            conta
18        file >> str_temp; file >> C0_k;    file >> C1_k;    file >>
19            C2_k;
20    }
21    else {
22        std::cout<<"can't open file!" << std::endl;
23    }
24
25 double CMaterialCorrelacao::getThermalConst(double T) {
26     double k    = C0_k    + C1_k    * T + C2_k    * T * T;
27     return rho * cp/k;
28 }

```

```

27
28 double CMaterialCorrelacao::getK(double T) {
29     double k = C0_k + C1_k * T + C2_k * T * T;
30     return k<0 ? C0_k : k;
31 }

```

Apresenta-se na listagem 7.16 o arquivo de cabeçalho da classe CMaterialInterpolacao.

Listing 7.16: Arquivo de implementação da classe CMaterialInterpolacao

```

1 #ifndef CMATERIALINTERPOLACAO_H
2 #define CMATERIALINTERPOLACAO_H
3
4 #include <QDir>
5 #include <string>
6 #include <vector>
7 #include "CMaterial.h"
8 #include "CSegmentoReta.h"
9
10 class CMaterialInterpolacao : public CMaterial {
11 public:
12     CMaterialInterpolacao();
13     CMaterialInterpolacao(std::string _name);
14
15     double getThermalConst(double T);
16
17     QColor getColor() { return color; }
18     std::string getName() { return name; }
19
20     double getK(double T);
21 protected:
22     std::string name;
23     QColor color;
24
25 private:
26     std::vector<CSegmentoReta> retaInterpolacao;
27     double rho, cp;
28     double xmin, xmax, edx;
29
30 };
31
32#endif // CMATERIALINTERPOLACAO_H

```

Apresenta-se na listagem 7.17 implementação da classe CMaterialInterpolacao.

Listing 7.17: Arquivo de implementação da função main()

```

1 #include "CMaterialInterpolacao.h"
2
3 CMaterialInterpolacao::CMaterialInterpolacao(std::string fileName){
4     std::string str_temp;
5     int r, g, b, alpha;
6     name = fileName;
7
8     QDir dir; std::string path = dir.absolutePath().toStdString();
9     std::ifstream file(path+ "/materiais//"+fileName);
10    if (file.is_open()){
11
12        file >> str_temp; file >> r; file >> g; file >> b; file >>
13            alpha;
14        color = QColor(r, g, b, alpha);
15
16        file >> str_temp; file >> rho;
17        file >> str_temp; file >> cp;
18
19        file >> str_temp; /// texto
20
21        double x1, x2, y1, y2;
22        file >> x1 >> y1;
23        xmin = x1;
24        while(file >> x2 >> y2){
25            retaInterpolacao.push_back( CSegmentoReta(x1,y1,x2,y2)
26                );
27            x1 = x2;
28            y1 = y2;
29        }
30        xmax = x1;
31        edx = (xmax-xmin)/ (retaInterpolacao.size()-1);
32    }
33    else{
34        std::cout<<"can't open file!" << std::endl;
35    }
36 double CMaterialInterpolacao::getThermalConst(double T){
37     return rho*cp/getK(T);
38 }
39

```

```

40 double CMaterialInterpolacao::getK(double T){
41     if( T <= xmin )
42         return retaInterpolacao[0].Fx(T);
43     else if(T >= xmax)
44         return retaInterpolacao[retaInterpolacao.size()-1].Fx(T);
45     // chute inicial, et = Estimativa do Trecho de reta que atende
        valor de x.
46     int et = (T - xmin) / edx;
47     while(true){ // procura pelo trecho de reta que contempla x.
48         if( T < retaInterpolacao[et].Xmin() and et > 1 )
49             et--;
50         else if ( T > retaInterpolacao[et].Xmax() and et <
            retaInterpolacao.size()-1 )
51             et++;
52         else
53             break;
54     };
55     return retaInterpolacao[et].Fx( T ); // calculo de Fx(x).
56 }
```

Apresenta-se na listagem 7.18 o arquivo de cabeçalho da classe CSegmentoReta.

Listing 7.18: Arquivo de implementação da classe CSegmentoReta

```

1 #ifndef CSegmentoReta_h
2 #define CSegmentoReta_h
3
4 #include <iomanip>
5 #include <vector>
6
7 #include "CReta.h"
8
9 /// Class CSegmentoReta, representa uma reta com intervalo xmin->
    xmax .
10 class CSegmentoReta : public CReta
11 {
12 private:
13     double xmin = 0.0; ///< Inicio do segmento de reta.
14     double xmax = 0.0; ///< Fim do segmento de reta.
15     bool ok = false;   ///< Se verdadeiro, x usado esta dentro
                        intervalo válido (xmin->xmax)
16
17 public:
18     /// Construtor default.
```

```

19  CSegmentoReta ( ) { }
20
21  /// Construtor sobreescarregado, recebe pontos (x1,y1), (x2,y2).
22  CSegmentoReta (double x1, double y1, double x2, double y2)
23      :CRet(a(x1,y1,x2,y2),xmin{x1},xmax{x2} {})
24
25  /// Construtor copia.
26  CSegmentoReta (const CSegmentoReta& retaInterpolacao) {
27      xmin = retaInterpolacao.xmin;    xmax = retaInterpolacao.xmax;
28      ok = retaInterpolacao.ok;
29      x = retaInterpolacao.x;        y = retaInterpolacao.y;
30      a = retaInterpolacao.a;        b = retaInterpolacao.b;
31  }
32
33  // Metodos Get/Set
34  double Xmin( )           { return xmin;   }
35  void Xmin(double _xmin) { xmin = _xmin;   }
36  double Xmax( )           { return xmax;   }
37  void Xmax(double _xmax) { xmax = _xmax;   }
38
39  /// Se retorno for verdadeiro, valor de y esta dentro intervalo
40  /// xmin->xmax.
41  bool Ok()                { return ok;   }
42
43  /// Verifica se esta no intervalo de xmin->xmax.
44  bool TestarIntervalo (double _x) { return ok = ( _x >= xmin and
45                                              _x <= xmax)? 1:0;   }
46
47  /// Calcula valor de y = Fx(x);
48  virtual double Fx (double _x) {
49      TestarIntervalo(_x);
50      return CRet::Fx(_x);
51  }
52
53  /// Calcula valor de y = Fx(x);
54  double operator()(double _x) { return Fx(_x); }
55
56  /// Sobrecarga operador <<, permite uso cout << reta;
57  friend std::ostream& operator<<( std::ostream& os, const
58      CSegmentoReta& retaInterpolacao ) {
59      os.precision(10);
60      os<< retaInterpolacao.xmin << " ->" << retaInterpolacao.xmax

```

```

57     << "x:y=" << std::setw(15) << std::setprecision(10) <<
      retaInterpolacao.a << " + "
58     << std::setw(15) << std::setprecision(10) << retaInterpolacao
      .b << "*x";
59     return os;
60 }
61
62 /// Sobrecarga operador >>, permite uso cin >> reta;
63 friend std::istream& operator>>( std::istream& in, CSegmentoReta&
      retaInterpolacao ) {
64     in >> retaInterpolacao.xmin >> retaInterpolacao xmax
65     >> retaInterpolacao.a >> retaInterpolacao.b;
66     return in;
67 }
68
69 friend class CInterpolacaoLinear;
70 };
71#endif // CSegmentoReta_h

```

Apresenta-se na listagem 7.19 implementação da classe CSegmentoReta.

Listing 7.19: Arquivo de implementação da função main()

```
#include "CSegmentoReta.h"
```

Apresenta-se na listagem 7.20 o arquivo de cabeçalho da classe CReta.

Listing 7.20: Arquivo de implementação da classe CReta

```

1#ifndef CReta_H
2#define CReta_H
3
4#include <iostream>
5#include <iomanip>
6#include <fstream>
7
8/// Class CReta, representa uma reta y = a + b * x.
9class CReta
10{
11protected:
12    double x = 0.0; // Representa valor de x.
13    double y = 0.0; // Representa valor de y.
14    double b = 0.0; // Representa valor de b da equacao y = a + b*
                     x; normalmente e calculado.
15    double a = 0.0; // Representa valor de a da equacao y = a + b*
                     x; normalmente e calculado.

```

```
16
17 public:
18     /// Construtor default.
19     CReta( ){ }
20     /// Construtor sobre carregado, recebe a e b.
21     CReta( double _a, double _b): b{_b},a{_a}{ }
22
23     /// Construtor sobre carregado, recebe dados pontos (x1,y1) e (x2,
24     /// y2).
25     CReta( double x1, double y1, double x2, double y2) : b{(y2-y1)/(x2-x1)}, a{y1-b*x1} { }
26
27     /// Construtor de copia.
28     CReta( const CReta& reta): x{reta.x}, y{reta.y},a{reta.a}, b{reta.b} { }
29
30     // Metodos Get/Set
31     double X( ) { return x; }
32     void X( double _x ) { x = _x; }
33     double Y( ) { return y; }
34     void Y( double _y ) { y = _y; }
35     double A( ) { return a; }
36     void A( double _a ) { a = _a; }
37     double B( ) { return b; }
38     void B( double _b ) { b = _b; }
39
40     /// Calcula valor de y = Fx(x);
41     virtual double Fx( double _x ) { x = _x; return y = a + b
42                                         * x; }
43
44
45     /// Calcula valor de y = Fx(x);
46     double operator()(double x) { return Fx(x); }
47
48
49     /// Sobre carga operador <<, permite uso cout << reta;
50     friend std::ostream& operator<<( std::ostream& os, CReta& reta )
51     {
52         os << "y=" << std::setw(10) << reta.a << "+"
53             << std::setw(10) << reta.b << "*x";
54         return os;
55
56     /// Sobre carga operador >>, permite uso cin >> reta;
57     friend std::istream& operator>>( std::istream& in, CReta& reta )
```

```
52     {
53         in >> reta.a >> reta.b ;
54
55     /// Retorna string com a equacao y = a + b*x;
56     std::string Equacao() {     std::ostringstream os;      os << *
57         this;
58     }
59#endif // CReta_H
```

Apresenta-se na listagem 7.21 implementação da classe CReta.

Listing 7.21: Arquivo de implementação da função main()

```
1 #include "CReta.h"
```

Chapter 8

Teste

Neste capítulo serão apresentados os testes e resultados do simulador.

Inicialmente, o simulador será validado comparando uma solução analítica unidimensional conhecida com o resultado obtido pelo simulador. A seguir, serão apresentados resultados aplicados à indústria do petróleo, como:

- injeção térmica em reservatórios,
- simulação reduzida de *five-spot*,
- uma aplicação real na tecnologia, resfriamento de processadores.

8.1 Validação do simulador

Para validar os resultados do simulador, foi comparado os resultados do simulador, com a solução proposta por [Incropera 2008] (equação 5.57).

A solução para o caso unidimensional é:

$$\frac{T - T_s}{T_i - T_s} = \operatorname{erf} \left(\frac{x}{2\sqrt{\alpha t}} \right) \quad (8.1)$$

Onde erf é a *função erro de Gauss*, e α é a constante com as propriedades termofísicas:

$$\alpha = \frac{k}{\rho C_p} \quad (8.2)$$

As soluções horizontais e verticais do simulador são salvas em uma pasta em arquivos '.txt', com o respectivo tempo no nome do arquivo. A Figura 8.1 mostra a aplicação do problema no simulador.

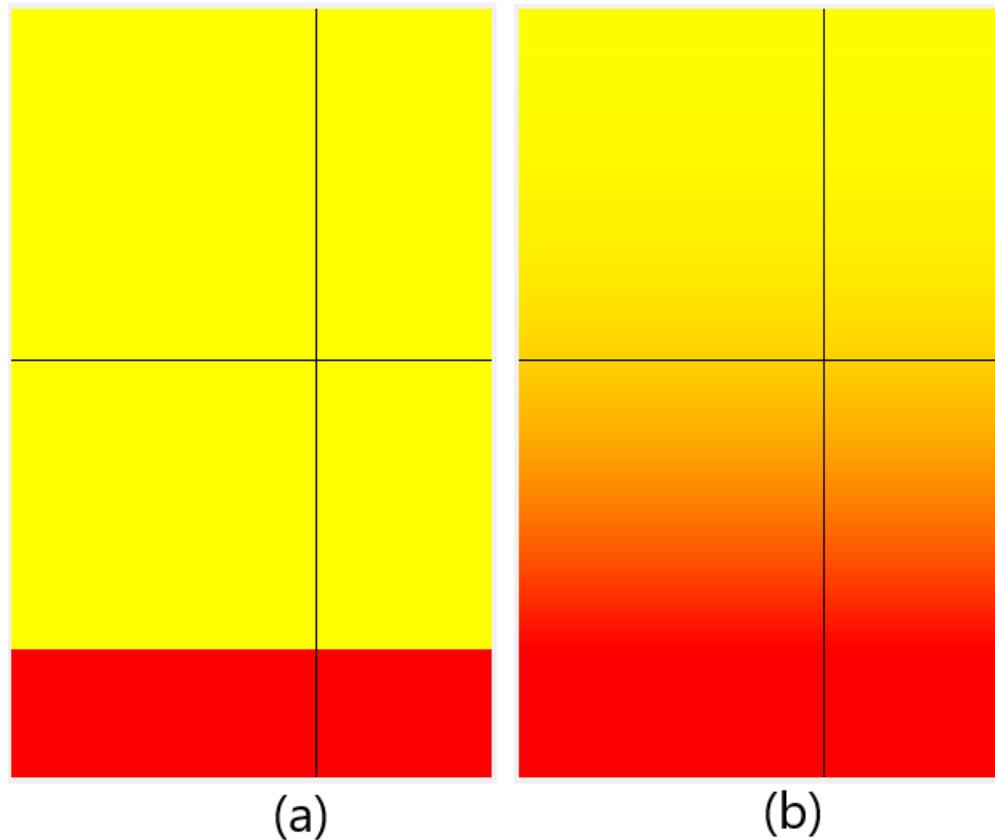


Figure 8.1: Aplicação do problema unidimensional no simulador. (a) é no tempo inicial e (b) depois de 100 segundos.

Para comparar os resultados do simulador com a solução analítica da Equação 8.1, foi programado um código em python apresentado na listatam 8.1.

Listing 8.1: Arquivo de implementação da validação em Python

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3 import math
4
5 def temperature(x,t, alfa):
6     Ti = 300
7     Tf = 1000
8     return Tf - (Tf - Ti)*math.erf(x/(2.0*math.sqrt(alfa*t)))
9
10 def maior_erro(x_sim, t_sim, t, alfa):
11     T_analitico = []
12     erro = []
13     erro_relativo = []
14
15     for x in x_sim:
16         T_analitico.append(temperature(x, t, alfa))
```

```
17
18     for i in range(len(t_sim)):
19         erro.append(abs(t_sim[i] - T_analitico[i]))
20         erro_relativo.append(erro[i]/t_sim[i]*100.0)
21     print('tempo:' + str(t))
22     print('erro:' + str(max(erro)))
23     print('erro_relativo:' + str(max(erro_relativo)))
24
25 x = np.linspace(0,0.10374,100)
26 t = [50.0, 100.0]
27
28 k = 40
29 rho = 1600
30 cp = 4000
31 alfa = k/(rho*cp)
32
33 for _t in t:
34     T = []
35     for i in x:
36         T.append(temperature(i, _t, alfa))
37     plt.plot(x, T, 'bo')
38
39#####
40 f = open('vertical100.000000.dat', 'r')
41 x_sim = []
42 t_sim = []
43 for i in f:
44     split = i.split(';')
45     x_sim.append(float(split[0]))
46     t_sim.append(float(split[1].replace('\n', '')))
47
48 t_sim.sort(reverse=True)
49 for i in range(len(t_sim)):
50     if t_sim[0] == 1000.0:
51         t_sim.pop(0)
52         x_sim.pop(-1)
53     else:
54         break
55 print('Tamanho:' + str(max(x_sim) - min(x_sim)))
56 plt.plot(x_sim, t_sim, 'r+')
57 maior_erro(x_sim, t_sim, 100.0, alfa)
```

```
58
59 f = open('vertical50.000000.dat', 'r')
60 x_sim = []
61 t_sim = []
62 for i in f:
63     split = i.split(';')
64     x_sim.append(float(split[0]))
65     t_sim.append(float(split[1].replace('\n', '')).replace(' ', '')))
66
67 t_sim.sort(reverse=True)
68 for i in range(len(t_sim)):
69     if t_sim[0] == 1000.0:
70         t_sim.pop(0)
71         x_sim.pop(-1)
72     else:
73         break
74 print('Tamanho:' + str(max(x_sim) - min(x_sim)))
75 plt.plot(x_sim, t_sim, 'r+')
76 maior_erro(x_sim, t_sim, 50.0, alfa)
77
78
79 plt.legend(['Analitico-100', 'Analitico-50', 'Simulador-100',
80             'Simulador-50'])
81 plt.show()
```

Como resultado é apresentado um gráfico com duas soluções, para os tempos de 50 e 100 segundos. Veja Figura 8.2. Observe que para os dois tempos a solução numérica, obtida com o simulador desenvolvido, esta de acordo com a solução teórica.

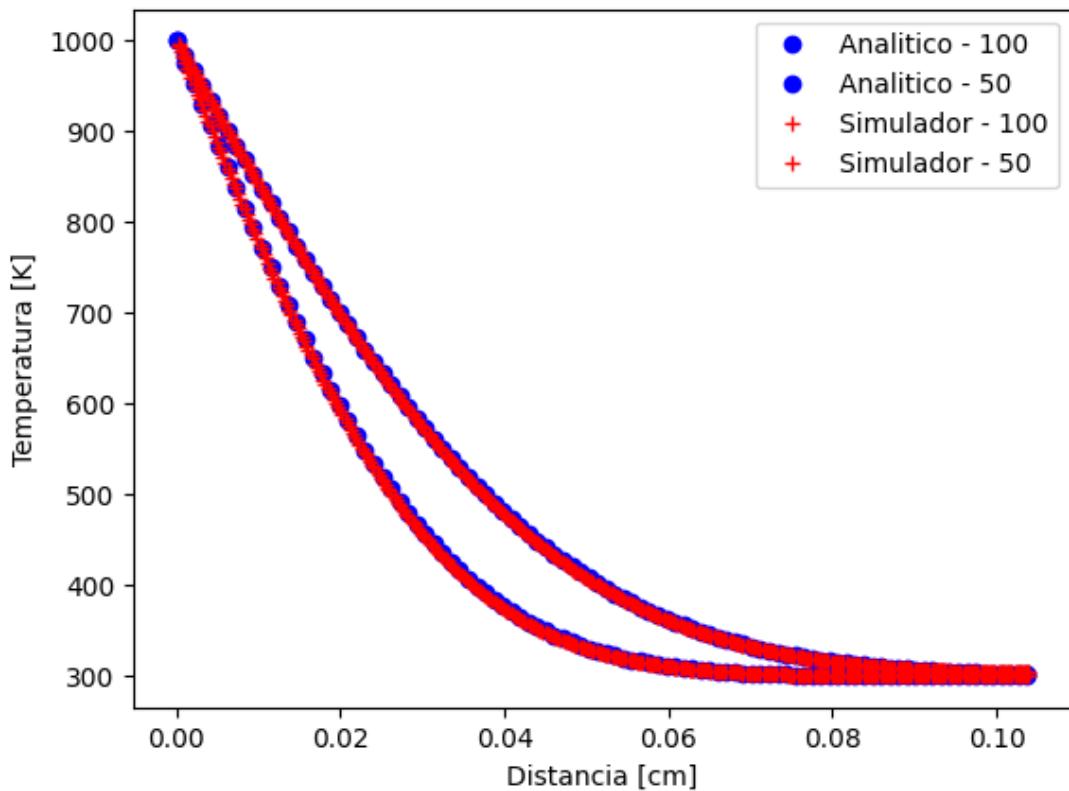


Figure 8.2: Comparação da solução da equação de calor com o resultado do simulador

Os dados da simulação foram obtidos utilizando um material com propriedades termofísicas constantes apresentadas na Tabela 8.1:

Table 8.1: Tabela com as propriedades termofísicas do modelo de validação

Propriedade	Valor
C_p	40.000
k	40
ρ	1.600

O erro do simulador foi de 0.69% para 50,0 segundos, e 0,88% para 100,0 segundos. O mínimo de iteração para cada variação de tempo foi de 800 iterações.

É importante mencionar que o número de iterações deve ser alta, pois o simulador resolve o método BTCS de forma iterativa, e só consegue 'avançar' a influência da temperatura em uma célula por iteração. O número mínimo de iterações para o simulador deve ser maior que o número de células na vertical (número de células na vertical é maior que na horizontal).

O modelo para simulação pode ser obtido no arquivo “Modelo_validacao.dat”.

8.2 Injeção de calor em reservatório - comparação com outro simulador

Como segundo teste do simulador, será comparado o resultado do simulador com o simulador desenvolvido no Trabalho de Conclusão de Curso do Guilherme [Lima 2020].

Para essa simulação, será considerado um reservatório de arenito com água, onde a porosidade é 20%. As fronteiras do reservatório estão com temperatura constante ao longo do tempo de 1000K (condição de contorno de Dirichlet), e um poço central com tamanho desprezível, também com temperatura constante de 1000K. O restante do reservatório está com temperatura de 300K.

As propriedades dos materiais estão na Tabela 8.2.

Table 8.2: Tabela com propriedades termofísicas [Dong, McCartney e Lu 2015]

Material	$k [W/m.K]$	$\rho [kg/m^3]$	$c_p [J/kg.K]$
Arenito	2,10	2270,0	710,00
Água	0,56	999,87	4.200,00

Como o simulador não consegue tratar mistura de materiais, será utilizado o desenvolvimento utilizado pelo trabalho do Guilherme, considerando porosidade de 0,20. Como não é apresentado uma equação para calcular densidade e a capacidade térmica da mistura isoladamente, será utilizado o mesmo modelo para a condutividade térmica (Eq. 8.3), onde ψ é a propriedade termofísica analisada.

$$\psi = \phi\psi_1 + (1 - \phi)\psi_2 \quad (8.3)$$

Com isso, é obtido a propriedade do arenito com água na Tabela 8.3.

Table 8.3: Tabela com propriedades termofísicas do arenito com água

Material	$k [W/m.K]$	$\rho [kg/m^3]$	$c_p [J/kg.K]$
Arenito com água	1,792	2015,974	1408,00

O resultado da simulação é mostrado na Figura 8.3. Em (a), é apresentado o modelo inicial, antes de iniciar a simulação. Em (b), é apresentado o modelo após 3600 segundos. Em (c), é mostrado o modelo pela renderização 3D.

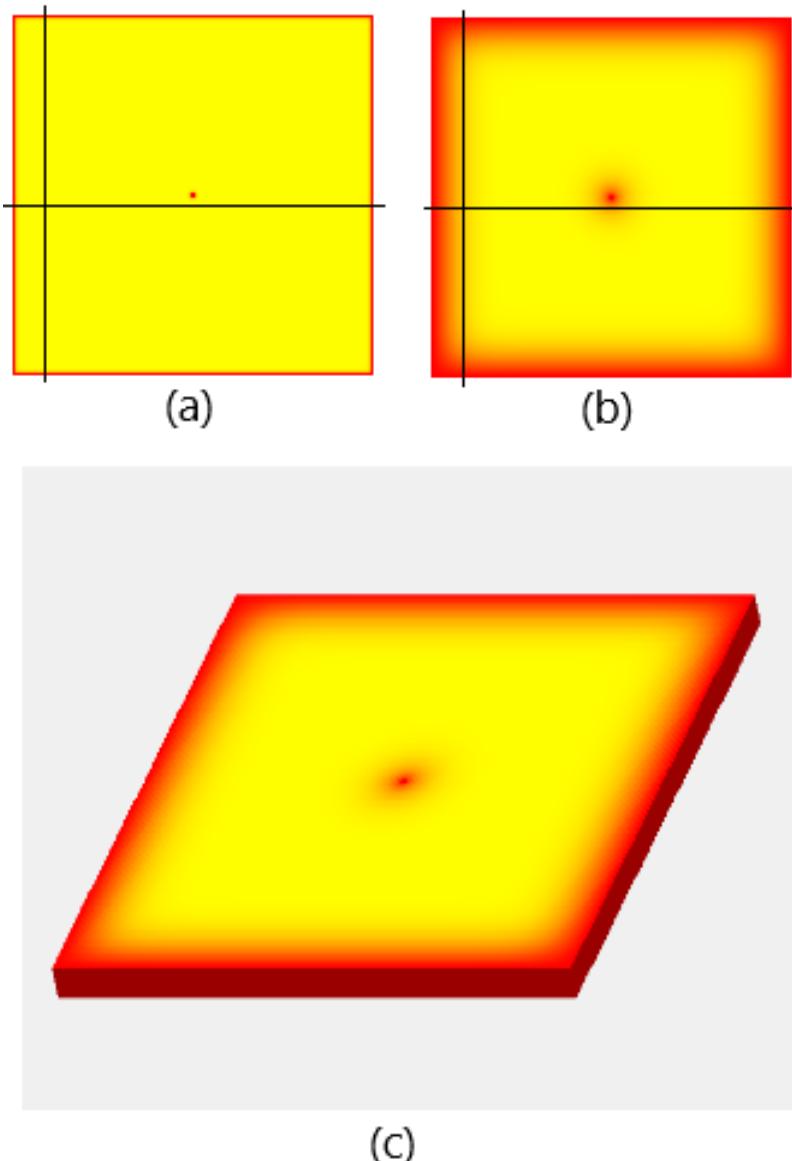


Figure 8.3: Resultados da simulação com renderização 3D

A Figura 8.4 mostra os gráficos da simulação. Em (a), é mostrada a temperatura ao longo do tempo, no ponto onde as duas retas de estudo se cruzam. Em (b), é mostrado a temperatura ao longo da reta horizontal de estudo. É possível observar os picos de temperatura elevada nas extremidades (região onde a temperatura é contante em 1000K), e um pico na região central, onde a reta se aproxima do ponto central de injeção térmica.

Em (c) é mostrado a temperatura ao longo da reta vertical de estudo, e é observado os dois picos das extremidades e uma região linear de temperatura.

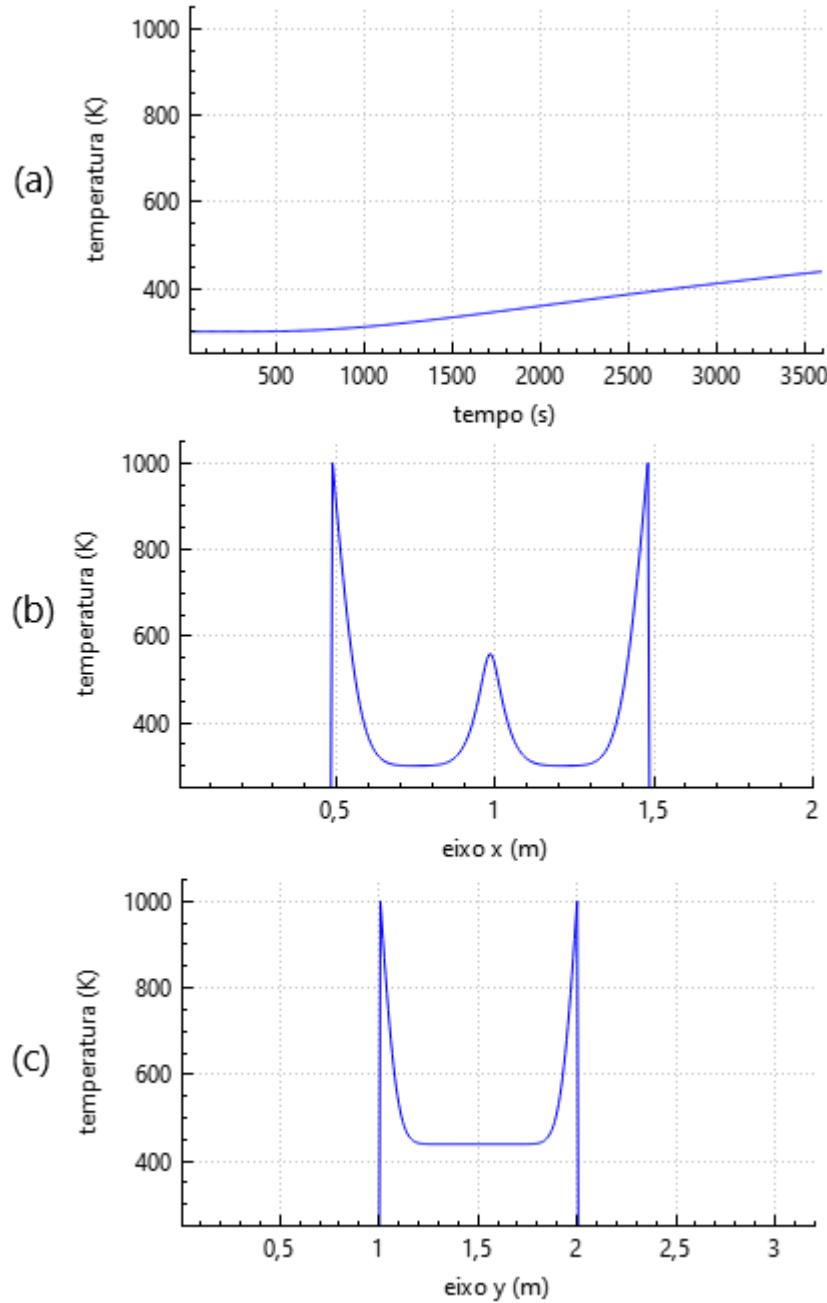


Figure 8.4: Gráficos da simulação para o tempo de 3.600 segundos

Para resolver esse problema, foi utilizado no simulador um $dx=0.00667$, posição do ponto de estudo em (87, 229).

O modelo para simulação pode ser obtido no arquivo “Modelo_guilherme.dat”.

8.3 Injeção de calor em reservatório - modelo five-spot

Dando sequência para os modelos de injeção de calor em reservatórios, um modelo bastante utilizado é o *five-spot*, caracterizado pela presença de 5 poços em um reservatório,

com 4 injetores, e 1 central produtor. Para a primeira simulação, os 5 poços serão injetores de calor.

As propriedades da rocha e do tamanho do reservatório são as mesmos da seção anterior (Tabela 8.3).

Na Figura 8.5 é apresentado o resultado do modelo *five-spot* com os 5 poços injetores.

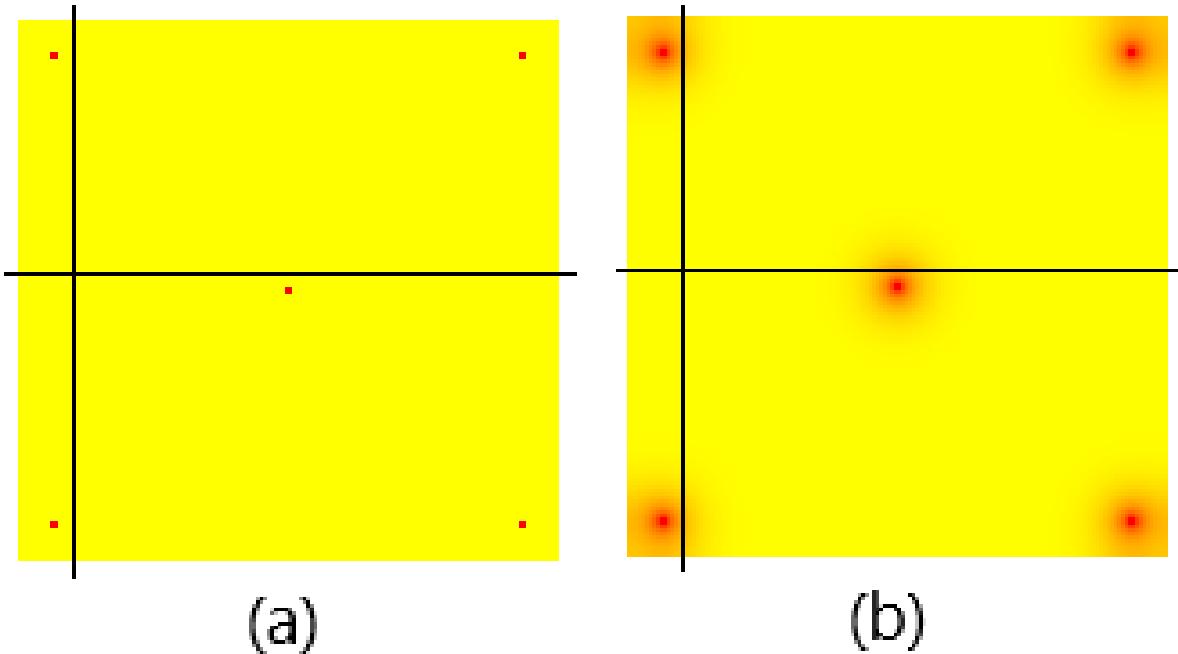


Figure 8.5: Resultados da simulação do primeiro modelo *five-spot* após 3.600 segundos

Analizando os resultados gráficos na Figura 8.6, em (a) é possível perceber um pico de temperatura no meio do eixo x, devido a proximidade com o poço central, e em (b), os picos estão nas extremidades, por causa dos poços superior-esquerdo e inferior esquerdo.

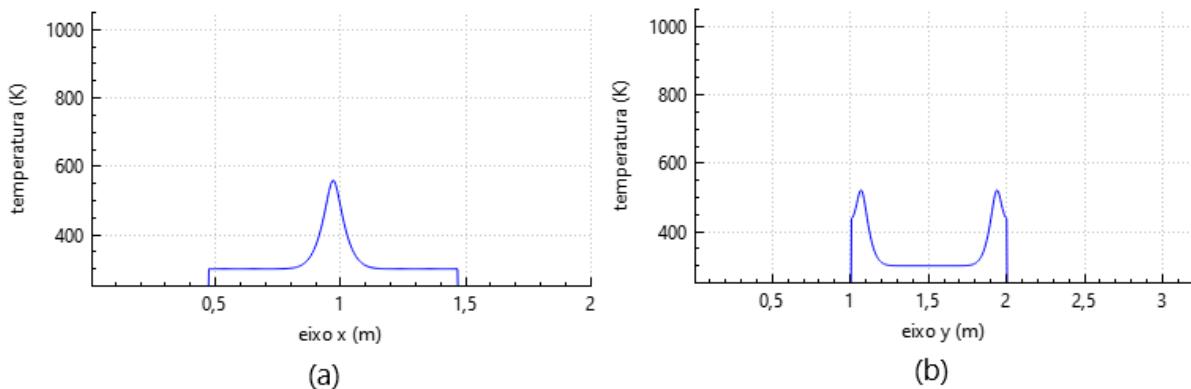


Figure 8.6: Gráficos da simulação do primeiro modelo *five-spot*

O modelo para simulação pode ser obtido no arquivo “Modelo_five_spot_1.dat”.

Agora será simulado um modelo *five-spot* com o poço central produtor. Para o simulador, será como um sumidouro de calor, onde a temperatura será constante em

300K (mesma temperatura do restante do reservatório).

Como é um sistema isolado, com fonte e sumidouro, após um longo período de tempo, é atingido regime permanente, onde a temperatura atingirá um equilíbrio, e não varia com o tempo. Na simulação, foi alcançado um regime próximo ao permanente, no tempo 460.800 segundos, ou 128 horas.

A Figura 8.7 mostra em (a) o cenário inicial, (b) o cenário final, próximo ao regime permanente e em (c) a renderização do reservatório em 3D.

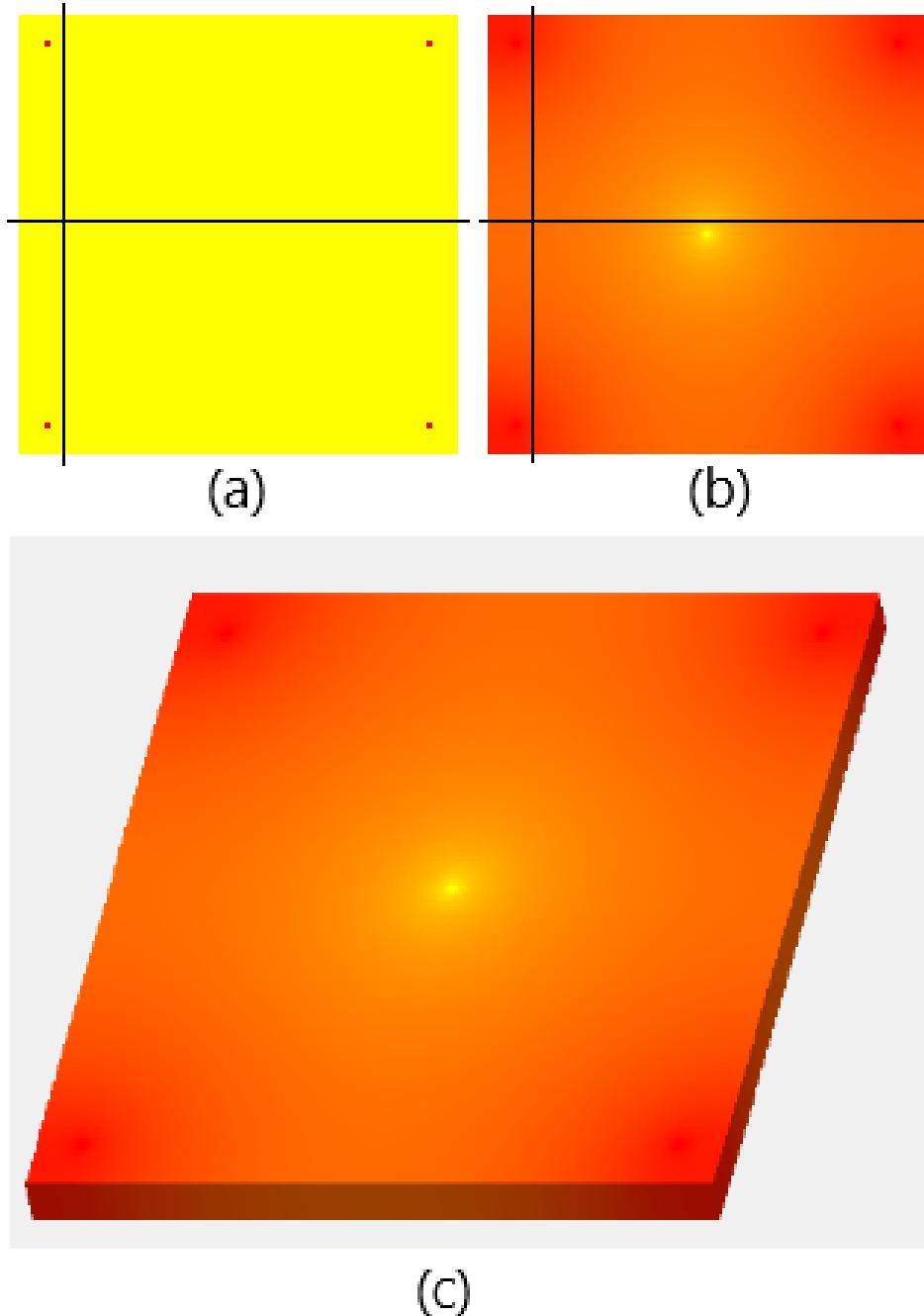


Figure 8.7: Resultados da simulação do segundo modelo *five-spot* após 460.800 segundos

Abaixo, seguem os gráficos na Figura 8.8. Em (a), é possível perceber uma temperatura

maior nas extremidades, diminuindo até as proximidades do poço. Em (b), é observado uma reta nas extremidades, e uma parábola na região central. Em (c), é apresentada a temperatura ao longo do tempo, no ponto central de estudo. Para atingir o regime permanente, a temperatura nesse gráfico deveria estabilizar, convergindo para uma reta constante.

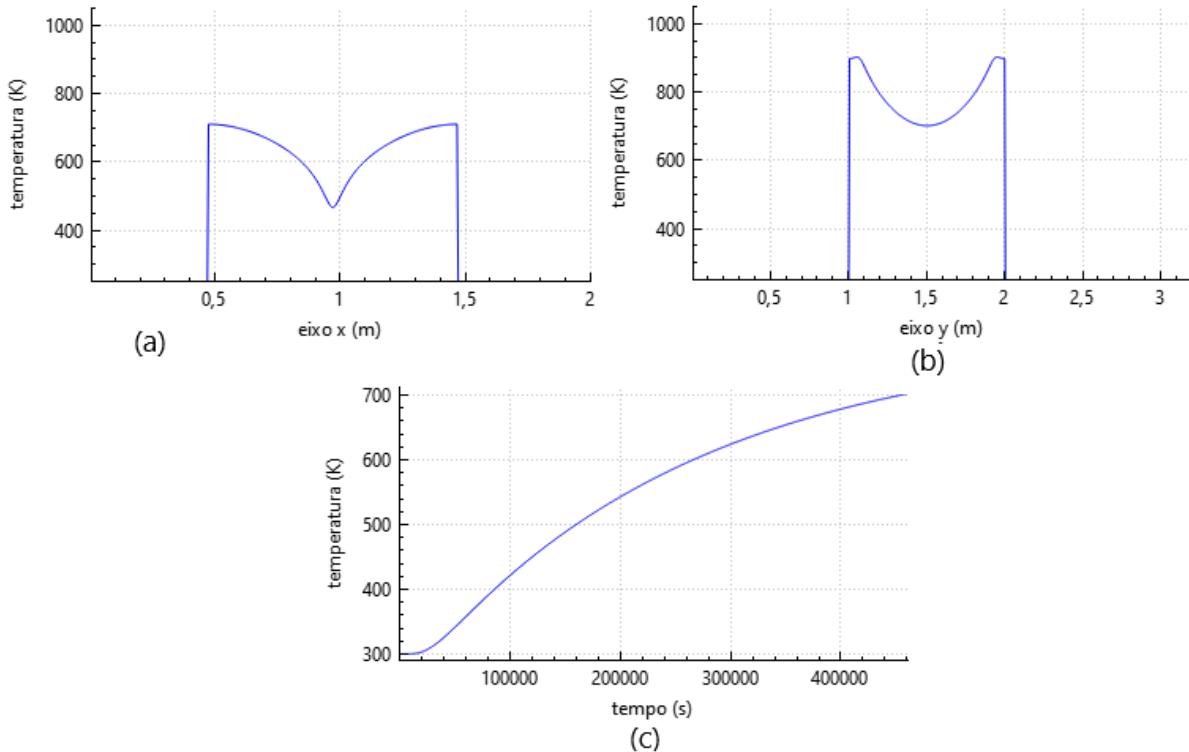


Figure 8.8: Gráficos da simulação do segundo modelo *five-spot*

O modelo para simulação pode ser obtido no arquivo “Modelo_five_spot_2.dat”.

8.4 Injeção de calor em reservatório - modelo 1

A seguir, é apresentado uma simulação para injeção térmica em um reservatório de petróleo, onde o poço está injetando calor com condutividade infinita e com penetração parcial.

As propriedades termofísicas utilizadas para a rocha são:

Table 8.4: Tabela com as propriedades termofísicas do modelo 1 - Arenito

Propriedade	Valor
C_p	920
k	1.6
ρ	2.600

As propriedades do poço são:

Table 8.5: Tabela com as propriedades termofísicas do modelo 1 - Ferro

Propriedade	Valor
C_p	593
k	10,33
ρ	8.020

Esse caso pode ser interpretado como uma fotografia da região próxima ao poço, sobre um poço com temperatura elevada que busca aquecer o reservatório, diminuindo assim a viscosidade do óleo e facilitando sua produção.

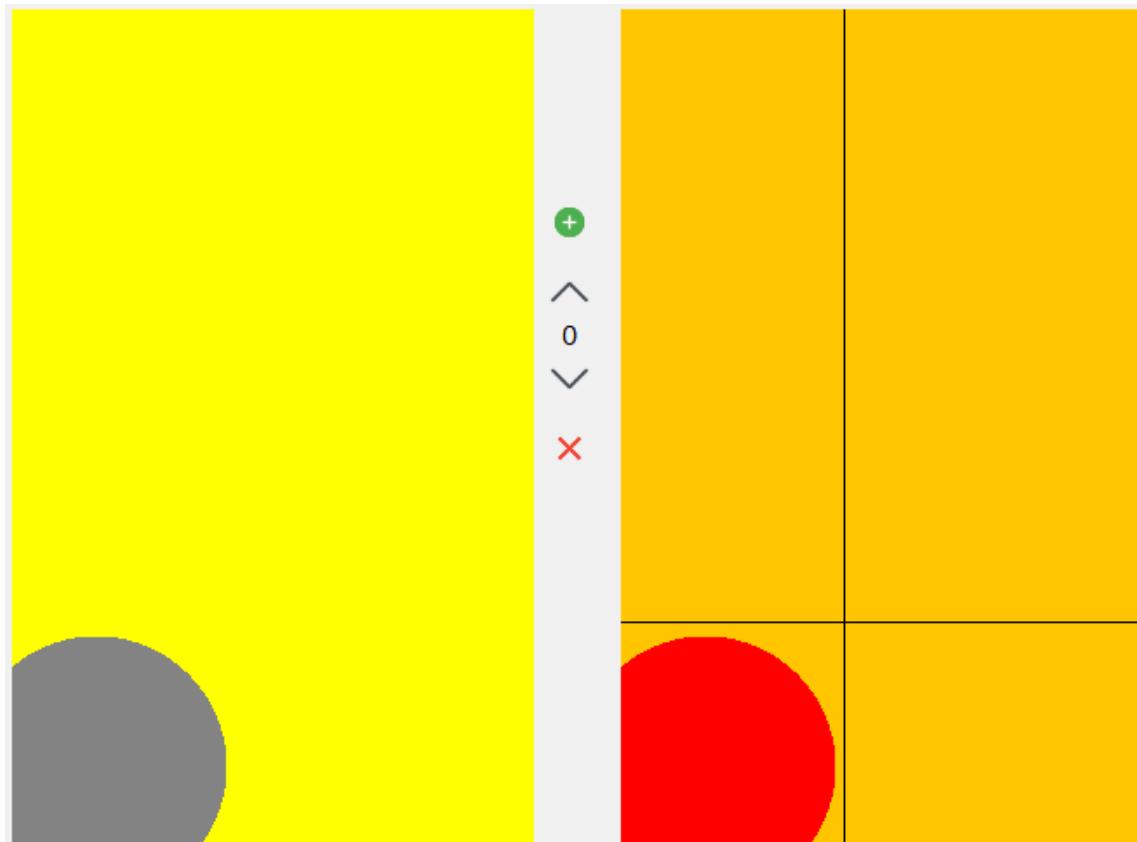


Figure 8.9: Modelo 1 de injeção térmica em reservatório

Com o modelo da Figura 8.9, é esperado que a variação de temperatura não atinja regiões distantes do reservatório devida à baixa condutividade térmica do arenito. E é exatamente isso que pode ser observado na Figura 8.10 com tempo de 4.000 segundos.

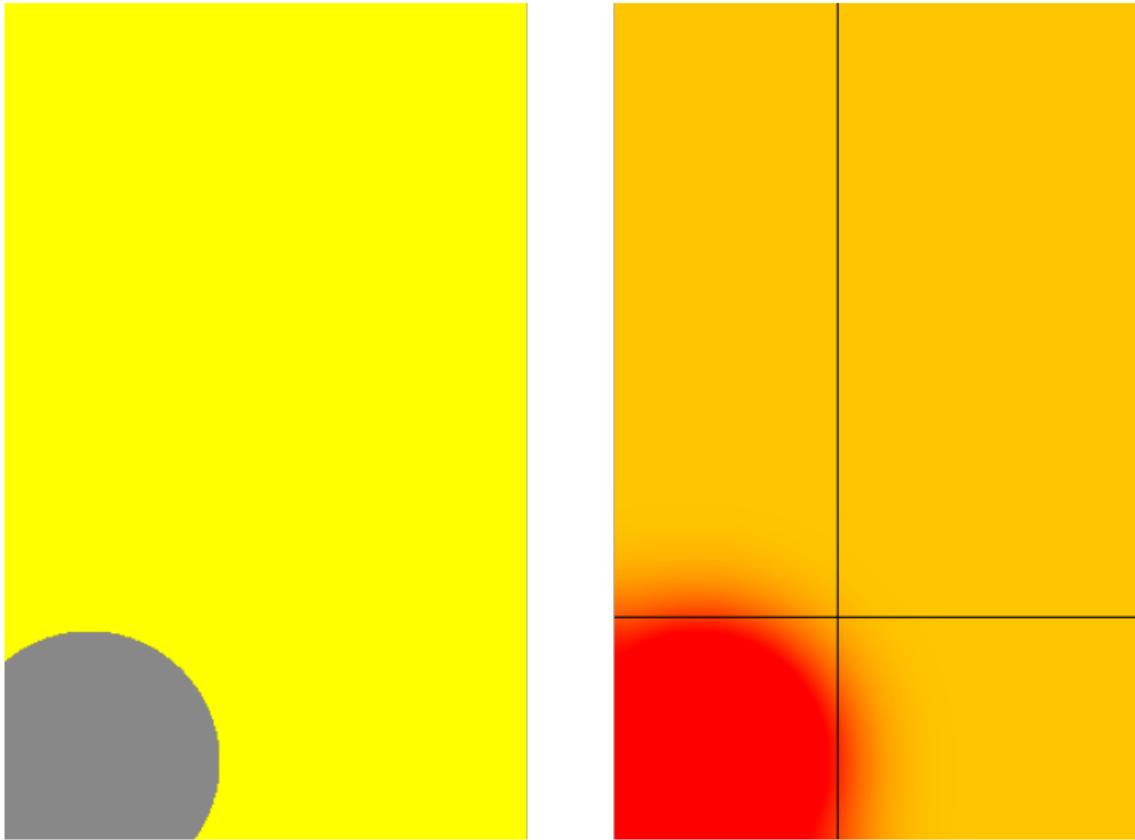


Figure 8.10: Modelo 1 de injeção térmica em reservatório após 4.000 segundos

Os gráficos são mostrados na Figura 8.11. Na esquerda, é mostrado a temperatura ao longo da reta horizontal preta, escolhido como ponto de estudo do modelo.

É interessante analisar a alta variação de temperatura para um tempo longo. Isso é esperado para materiais com baixíssimas condutividades térmicas. Caso o reservatório tivesse uma condutividade térmica alta, seria esperado que a temperatura fosse bem distribuída ao longo do reservatório.

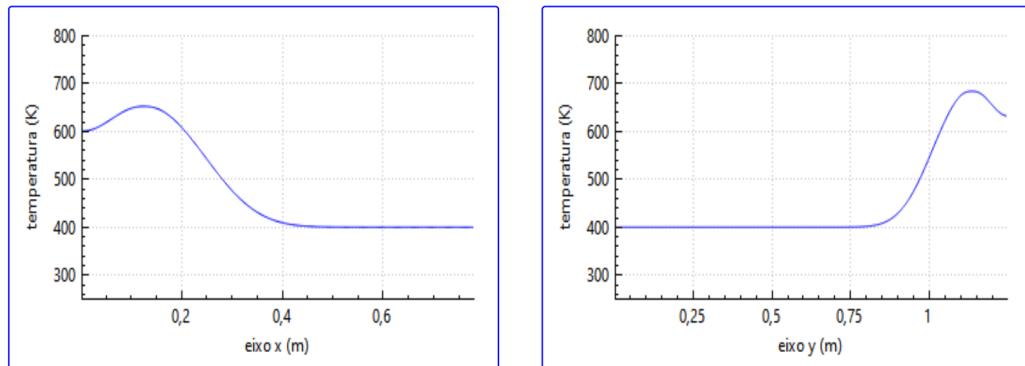


Figure 8.11: Graficos mostrando a variação de temperatura na região próxima ao poço

O modelo para simulação pode ser obtido no arquivo “Modelo_reservatorio_sem_agua.dat”.

8.5 Injeção de calor em reservatório - modelo 2

A seguir, será simulado o modelo 2 para o caso de injeção de calor em reservatório. A diferença fundamental para o caso 1, são os *fingers* de água quente adentrando no reservatório. Caso mais próximo da realidade.

No modelo 2, o poço continua com a condutividade térmica infinita, mas a água e o reservatório podem variar suas temperaturas, conforme mostrado na Figura 8.12

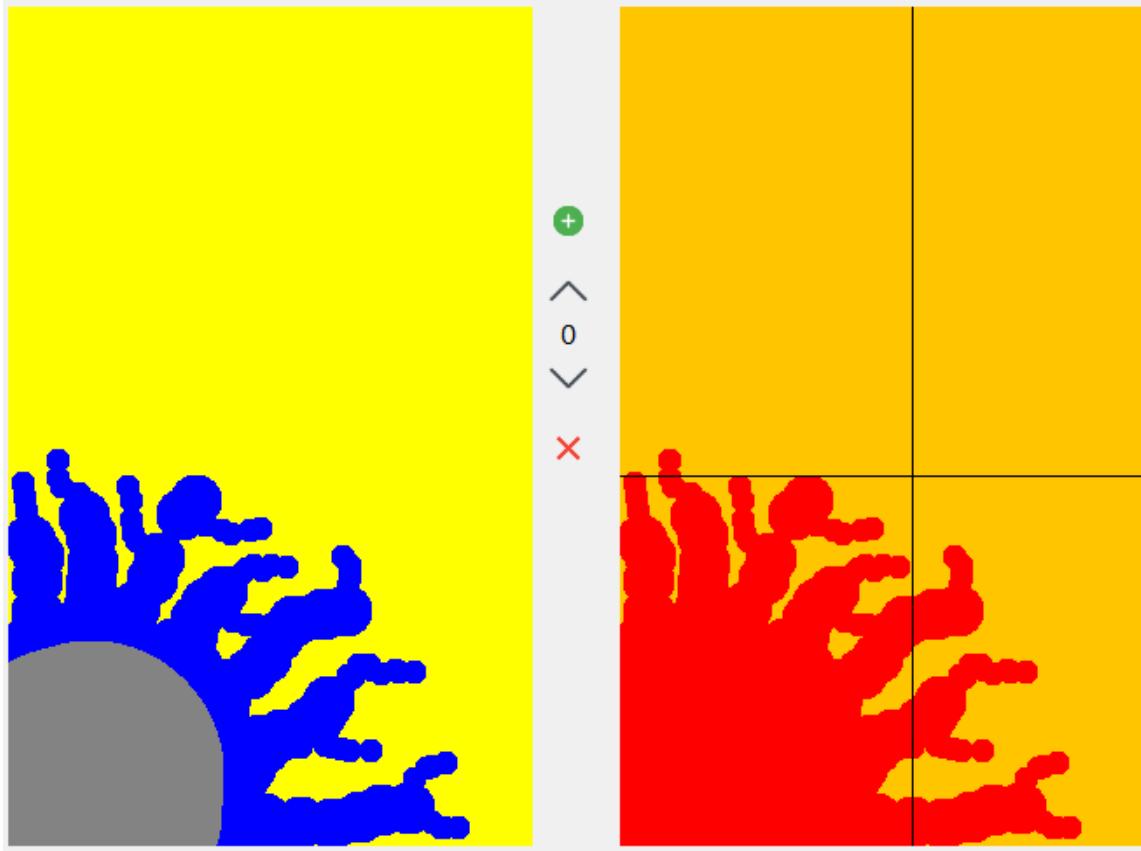


Figure 8.12: Tempo inicial da simulação. Na esquerda, o cinza representa o poço, azul a água e o amarelo, arenito. Na direita, é mostrado as temperaturas

Com a evolução do tempo, a região mais próxima dos *fingers* de água, é a mais alterada. A Figura 8.13 mostra esse cenário.

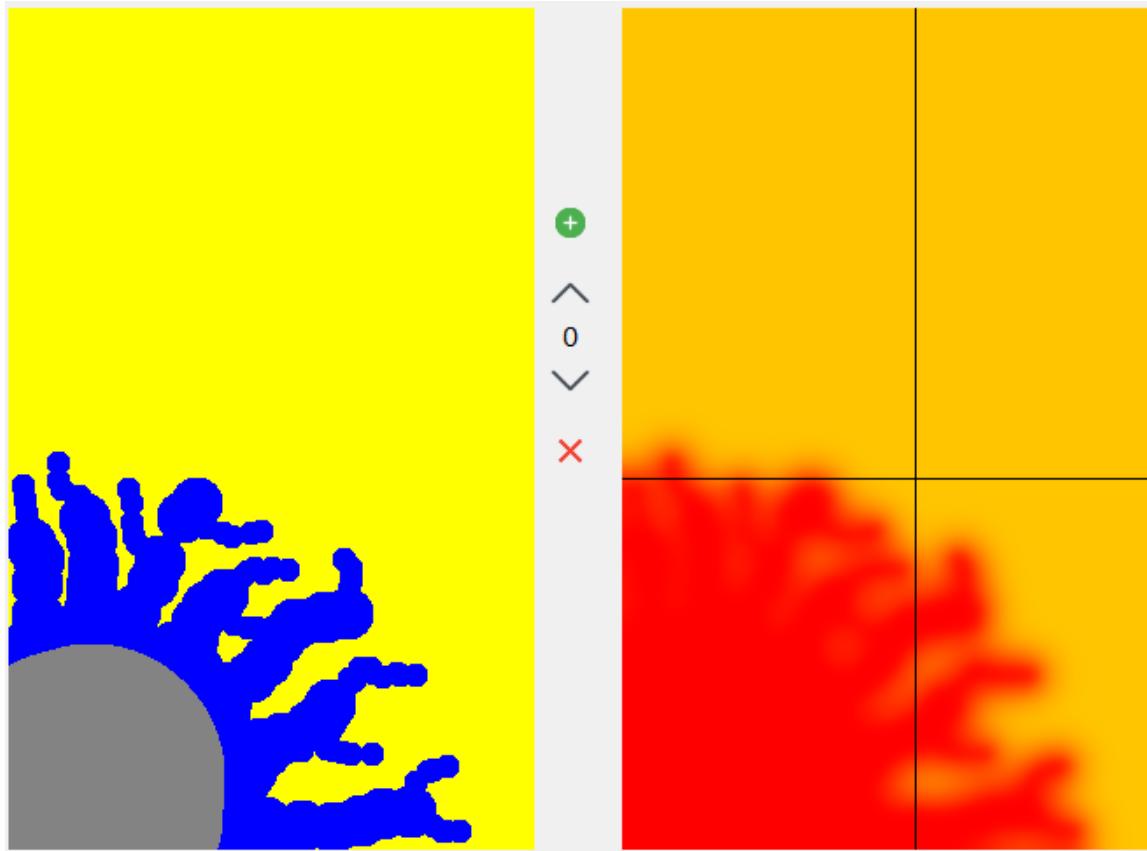


Figure 8.13: Evolução da simulação. Tempo de 610 segundos

Avançando mais no tempo, chegando a 7.180 segundos, é possível perceber que a região dos *fingers* de água está com temperatura bem distribuída (Figura 8.14), se assemelhando ao modelo 1, mas com poço muito mais largo.

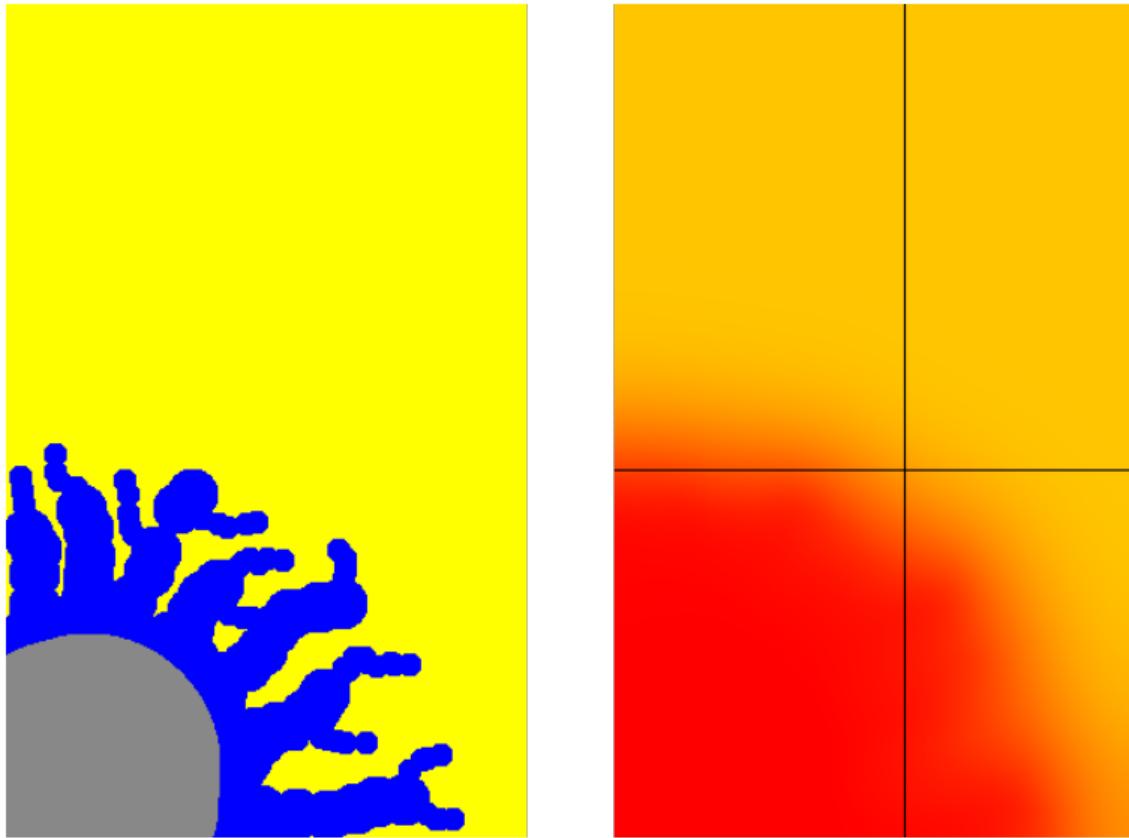


Figure 8.14: Tempo final de 7.180 segundos

A variação de temperatura ao longo das retas de estudo foram mais suaves em relação ao modelo 1 (Figura 8.15).

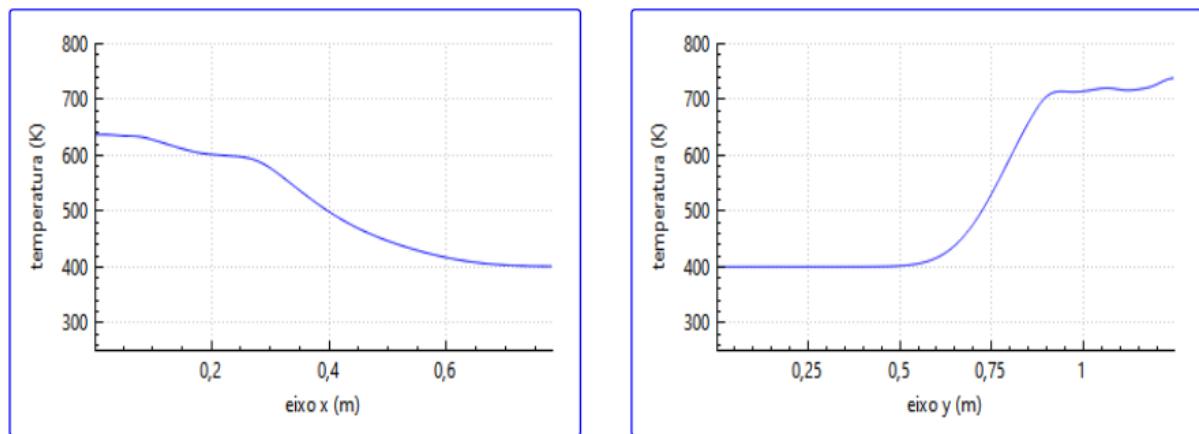


Figure 8.15: Gráficos do tempo final de 7.180 segundos

A comparação dos dois modelos, mostra quão efetivo é a injeção de água quente no reservatório em relação a um simples poço com temperatura elevada. A variação de temperatura atinge regiões mais distantes do reservatório, aquecendo um volume muito maior de óleo, que futuramente será produzido.

O modelo para simulação pode ser obtido no arquivo “Modelo_reservatorio_com_agua.dat”.

8.6 Resfriamento de processadores

Processadores são componentes elétricos de mais alta importância e complexidade do mundo moderno. São responsáveis por realizar numerosas operações matemáticas em curtíssimos espaços de tempo. Mas esse alto poder de processamento causa uma elevada geração de calor, a qual pode atrapalhar ou queimar o componente.

Então, para evitar danos no componente, foram criados diversos mecanismos de resfriamentos, como *air coolers* e *water coolers*. Mas esse problema fica complexo quando é analisado equipamentos com espaços reduzidos, como *smartphones* e *notebooks*.

Na figura 8.16, é mostrado o interior de um *notebook*. É possível perceber uma longa barra de cobre (*heatpipe*), cruzando pela GPU e CPU, os componentes com maior processamento e geração de calor.



Figure 8.16: Interior de um *notebook*, apresentando o *heatpipe*, que é a barra de cobre que cruza a GPU e CPU, e resfria na ventoinha

Utilizando o simulador, é possível simular o caso acima, utilizando cobre com propriedades constantes como material.

Table 8.6: Tabela com as propriedades termofísicas do modelo do *notebook* - Cobre

Propriedade	Valor
C_p	353
k	42
ρ	7.262

Na figura 8.17, é apresentado o modelo do resfriador. onde o grid 0, são referentes às fontes de calor (GPU, CPU), e acima é o sumidouro de calor (ventoinha). Já no grid 1, é mostrado o *heatpipe* interligando os componentes, e chegando à ventoinha.

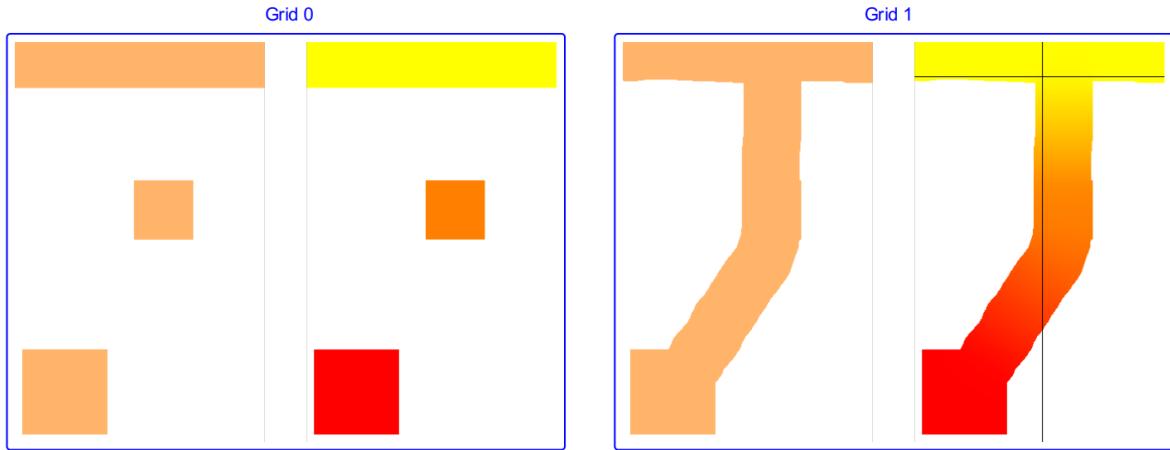


Figure 8.17: Simulação do sistema de resfriamento do notebook após chegar ao período permanente

Na figura 8.18, são apresentados os gráficos da temperatura ao longo da horizontal (esquerda) e vertical (direita).

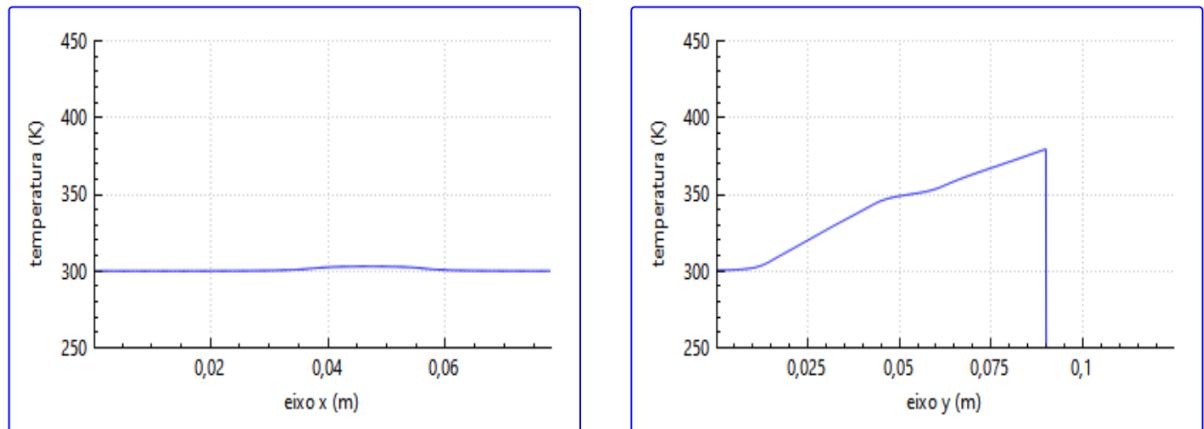


Figure 8.18: Interior de um *notebook*, apresentando o *heatpipe*, que é a barra de cobre que cruza a GPU e CPU, e resfria na ventoinha

A temperatura é rapidamente dispersada quando chega à ventoinha. O gráfico da direita da Figura 8.18, mostra o eixo y com duas quedas de temperatura, indicando que o componente do meio (CPU), deveria estar mais próximo do outro componente (GPU) para que a queda de temperatura seja linear, evitando um super-aquecimento de uma das partes.

Esse problema da localização é específico do modelo simulado, o qual foi desenhado sem ser totalmente fiel ao modelo da Figura 8.16. Tornando um caso mais interessante de se analisar devido ao erro milimétrico das posições do desenho.

O modelo para simulação pode ser obtido no arquivo “Modelo_notebook.dat”.

Chapter 9

Documentação

Neste capítulo é apresentado a documentação do software, mostrando como rodar o software, como utilizar e a documentação gerada pelo Doxygen. Por fim, são listadas as dependências externas.

9.1 Documentação do usuário

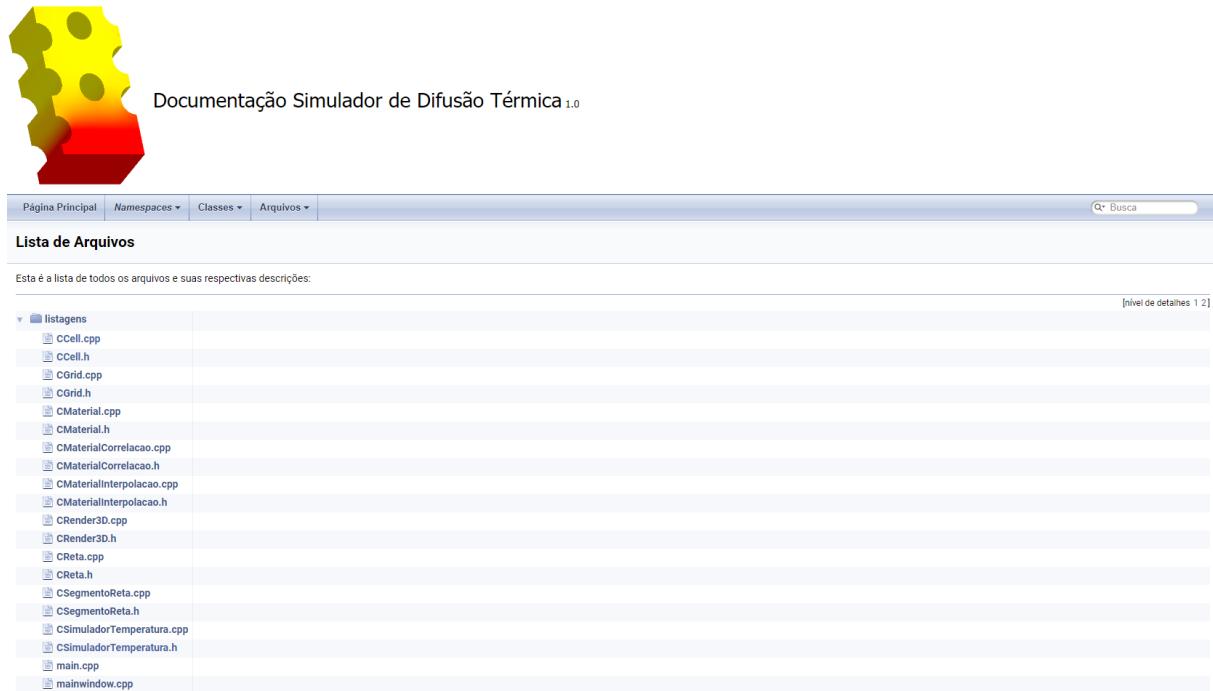
O Manual do Usuário é apresentado no Apêndice 10 - Manual do Usuário.

9.2 Documentação do desenvolvedor

Nesta seção são apresentadas informações para os desenvolvedores, como a documentação em html, e a listagem de algumas dependências específicas.

- Os códigos foram documentados utilizando o formato javadoc
 - <https://docs.oracle.com/javase/8/docs/technotes/tools/windows/javadoc.html>
 - <https://www.doxygen.nl/manual/docblocks.html>
- A documentação foi gerada utilizando o software doxygen
 - <https://www.doxygen.nl/>

Na Figura 9.1 mostra uma imagem da documentação gerada.

Figure 9.1: Logo e documentação do *software*

Ao clicar sobre qualquer item da listagem acima, será possível analisar o código daquele arquivo, como mostrado na Figura 9.2



Figure 9.2: Código fonte da classe CSimuladorTemperatura, no Doxygen

Chapter 10

Manual do Usuário

10.1 Instalação

O software foi disponibilizado no site <https://github.com/ldsc>.
Lá você encontra instruções atualizadas para baixar e instalar.

10.1.1 Dependências

Para compilar o software é necessário atender as seguintes dependências:

- Instalar o compilador g++ da GNU disponível em <http://gcc.gnu.org>.
 - Para instalar no GNU/Linux use o comando `dnf install gcc`.
- Biblioteca Qt disponível em <https://www.qt.io/download>;

10.2 Interface gráfica

A interface do programa é apresentada na Figura 10.1.

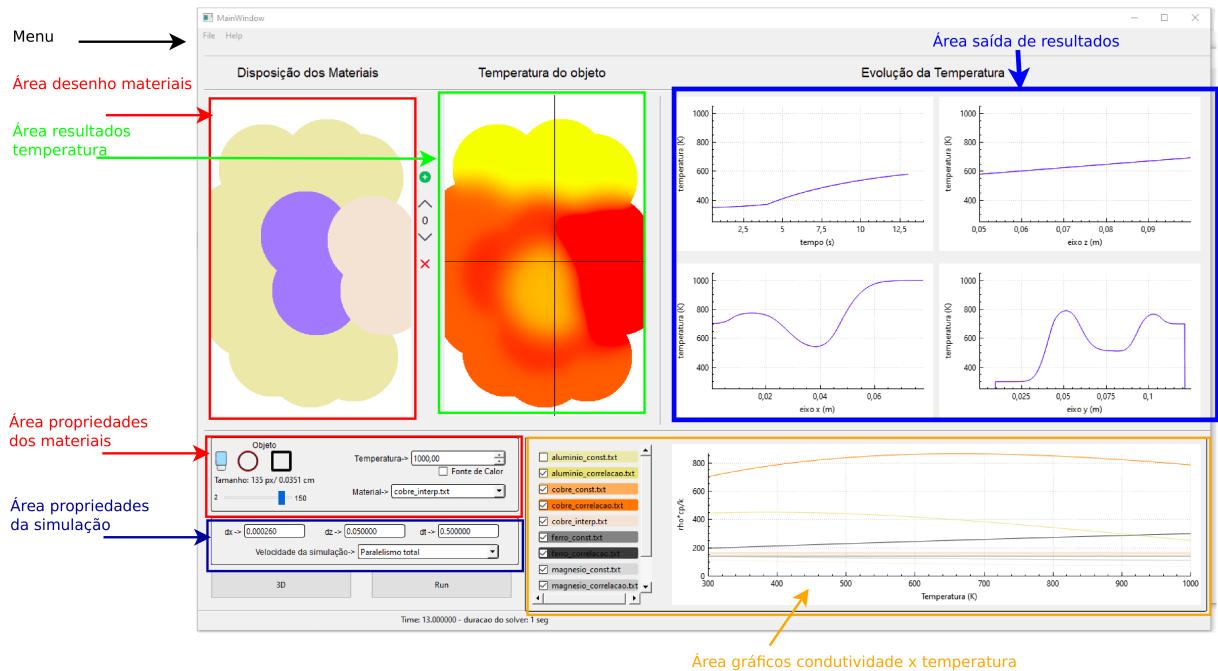


Figure 10.1: Imagem da Interface Gráfica

O Figura 10.1 mostra a janela principal do software e foram listadas 14 grupos de funcionalidades importantes ao usuário.

- Região onde o usuário desenha o objeto desejado. Para desenhar, deve ser clicado com o botão esquerdo do mouse.
- Retas e ponto de estudo, a partir dele, serão gerados os quatro gráficos da direita (5-8). Para escolher a posição, o usuário deve clicar com o botão direito do mouse.
- Região onde o usuário consegue diferenciar os materiais utilizados na simulação.
- Botões para o usuário navegar entre as camadas. Clicando no 'mais', ele pode criar uma camada. No 'X', pode excluir a camada atual, e as setas são para navegação.
- Gráfico da temperatura ao longo do tempo. Mostra a temperatura no ponto de estudo escolhido em 3, e é atualizado em toda evolução temporal da simulação.
- Gráfico da temperatura ao longo das camadas. Como só foi criado uma única camada, o gráfico ficará vazio.
- Gráfico da temperatura ao longo da horizontal. Mostra a temperatura ao longo da reta horizontal de estudo.
- Gráfico da temperatura ao longo da vertical. Mostra a temperatura ao longo da reta vertical de estudo.
- Gráfico com as propriedades termofísicas ao longo da temperatura. O usuário pode selecionar o material que quer analisar na região esquerda da área destacada.

- Propriedades do desenho. Aqui o usuário pode escolher se quer apagar o desenho ou não, o formato do pincel, tamanho do pincel, temperatura e material da área desenhada e se é fonte/sumidouro de calor ou não.
- Propriedades da simulação. Aqui o usuário pode configurar o tamanho da malha (dx), a distância entre as camadas (dz), o intervalo de tempo (dt), e os diversos tipos de velocidade da simulação: sem paralelismo, paralelismo por grid, paralelismo total (este último é o mais rápido, e é a escolha padrão)
- Botão para iniciar a janela com a renderização 3D. Para navegar nessa janela, são listados os seguintes botões:
 1. Espaço: muda a cor entre temperatura ou materiais.
 2. Pg Up: zoom in.
 3. Pg Down: zoom out.
 4. w/a/s/d: configura o ângulo do objeto.
 5. setas: move o objeto na janela.
 6. mouse: mesmas funcionalidades de (d).
- Botão para iniciar ou parar a simulação.
- Área com informações da posição/temperatura/material do desenho à esquerda, e informações da simulação (tempo atual e quanto tempo levou para resolver as iterações para chegar no novo tempo)

10.3 Como adicionar materiais

Apresenta-se neste apêndice instruções para adição de novos materiais.

Para adicionar qualquer material ao simulador, é necessário clicar em *Arquivo->Import material* e escolher o arquivo desejado.

É importante lembrar que os arquivos devem ter o formato que será ensinado a seguir.

A extensão do arquivo deve ser:

- '.constante',
- '.correlacao'
- '.interpolacao'

conforme o modelo escolhido.

A Figura 10.2 ilustra o local do menu que deve ser selecionado para adicionar o material.

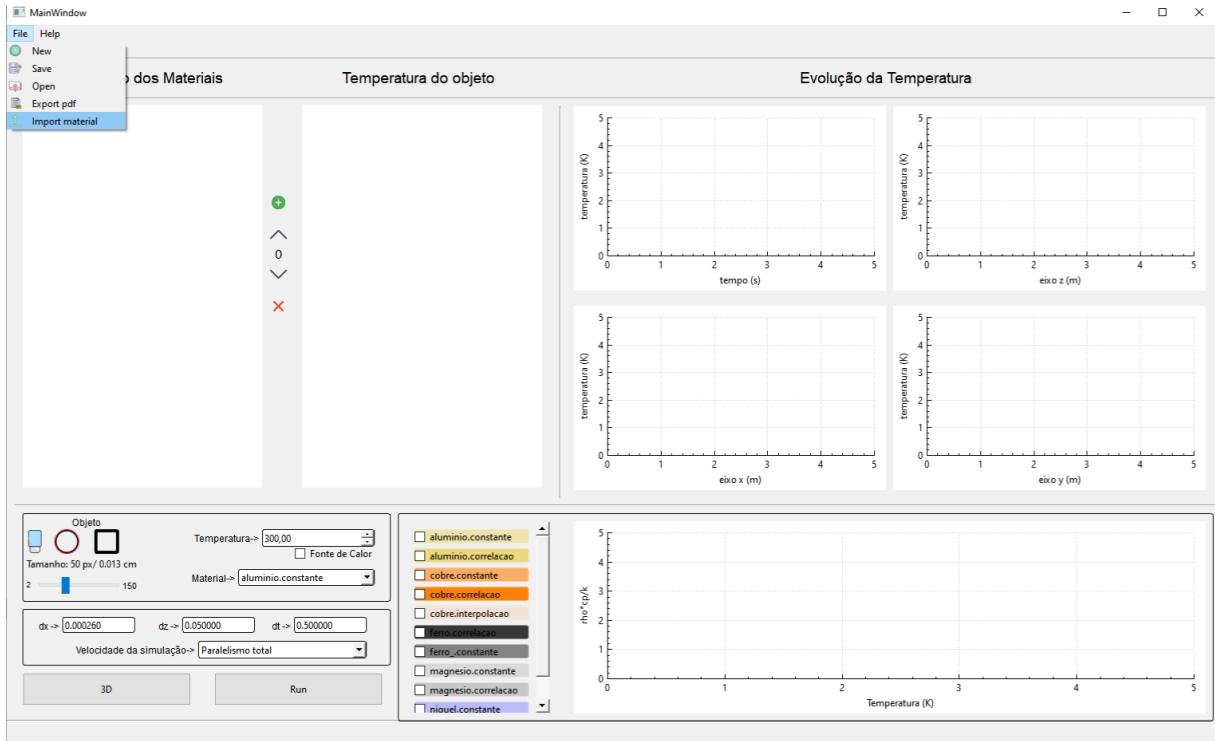


Figure 10.2: Como adicionar um material no simulador. Primeiro seleciona Arquivo, Import material. Uma janela será aberta, para o usuário escolher o material.

10.3.1 Método da correlação ou constante

Para adicionar um material que utilize métodos de correlação ou possuí propriedades termofísicas constantes, deverá ser criado um arquivo com extensão '.correlacao' ou '.constante', respectivamente.

O molde do arquivo é apresentado abaixo:

```
RGBA: 236 217 122 255
Cp: 2753
rho: 747.3
/// k=C0+C1*T-C2*T^2
k: 76.64 0.2633 0.0002
```

Onde a primeira linha contém o RGBA do material, a segunda linha contém o valor de Cp, seguindo por rho. Na quarta linha tem um comentário mostrando a equação da correlação utilizada, e na última linha, devem ser inseridos os valores de C0, C1 e C2, respectivamente.

10.3.2 Método de interpolação

Em muitos casos práticos as propriedades físicas de um material são obtidas em experimentos laboratoriais. É comum medir a condutividade do material para diferentes valores de temperatura, nestes casos, podemos usar métodos de interpolação para obter o valor de $k(T)$.

Para adicionar um material que utilize métodos de interpolação, deverá ser criado um arquivo com extensão '.interpolacao'.

O molde do arquivo é apresentado abaixo:

```
RGBA: 255 128 0 30
```

```
rho: 7.262
```

```
Cp: 7.925
```

```
-T---k:
```

```
100 0.2
```

```
200 0.4
```

```
300 0.5
```

```
400 0.55
```

```
500 0.6
```

```
600 0.65
```

```
700 0.7
```

```
800 0.75
```

```
900 0.8
```

Onde a primeira linha contém o *RGB*A do material, nas linhas abaixo contém *rho* e *Cp*. Abaixo da linha com *T* e *k*, são inseridos os valores da temperatura, e a respectiva condutividade térmica (*k*). O usuário pode adicionar quantas linhas desejar.

10.4 Como gerar relatório em PDF

Os resultados da simulação podem ser exportados em pdf, onde a primeira página apresenta informações da simulação, juntamente com os gráficos.

Nas páginas a seguir, são apresentados os grids, com um máximo de 6 grids por página.

O objeto 3D do relatório pode ser interpretado como chapas furadas de cobre, ferro e alumínio, com fontes de calor de níquel em vários pontos de cada chapa.

==> PROPRIEDADES DO GRID <==

Delta x: 0.00026 m

Delta z: 0.05 m

Delta t: 0.5 s

Largura total horizontal: 0.078 m

Largura total vertical: 0.1248 m

Largura total entre perfis (eixo z): 0.15 m

==> PROPRIEDADES DA SIMULAÇÃO <==

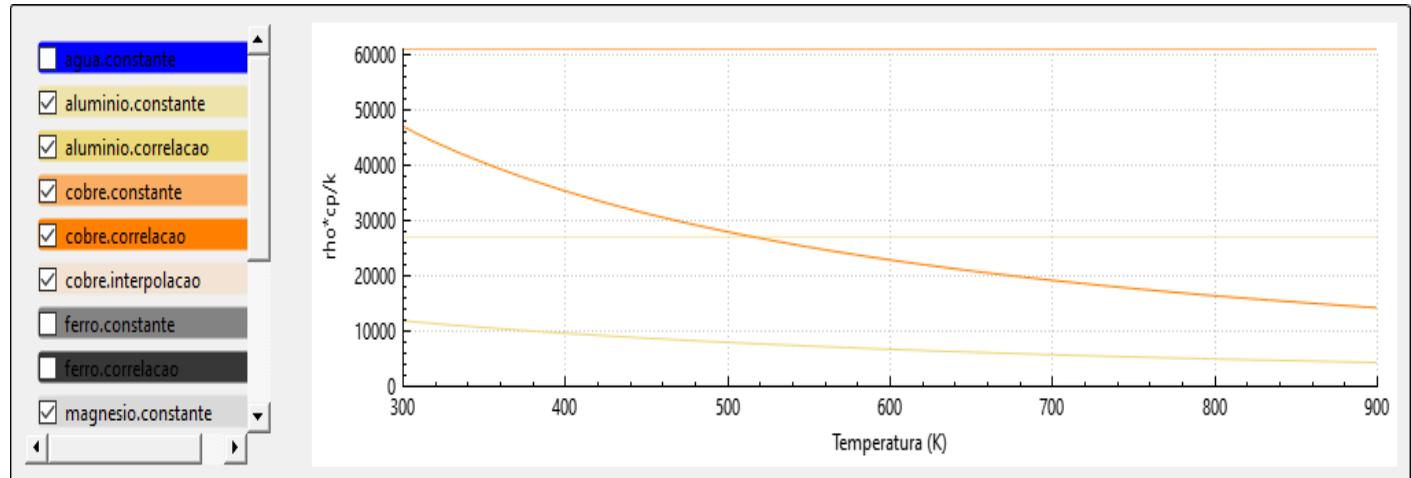
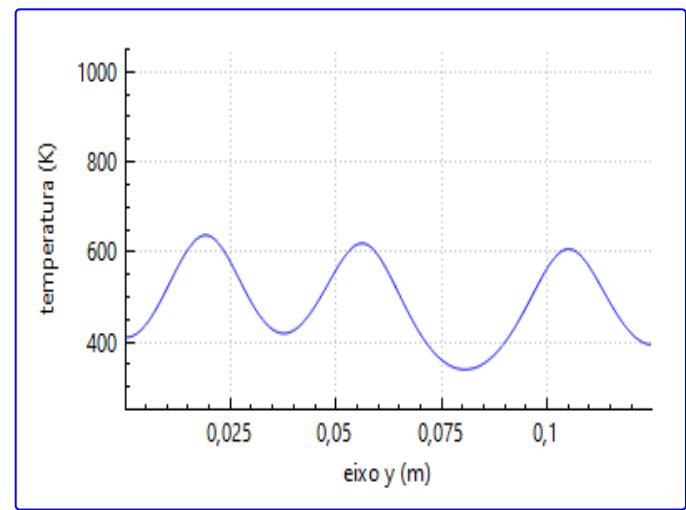
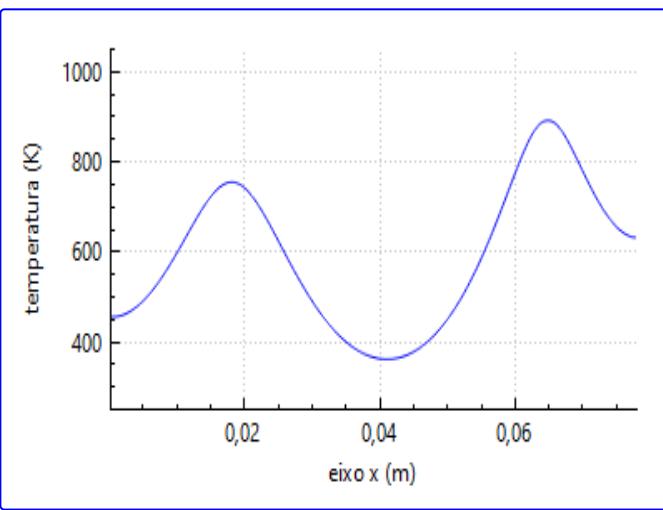
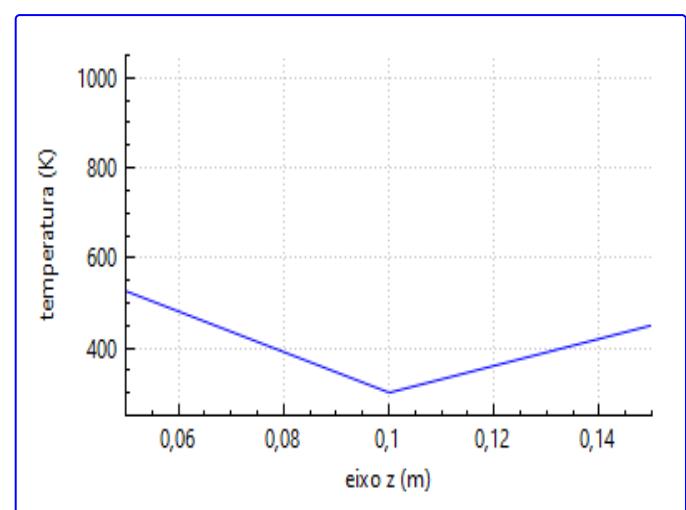
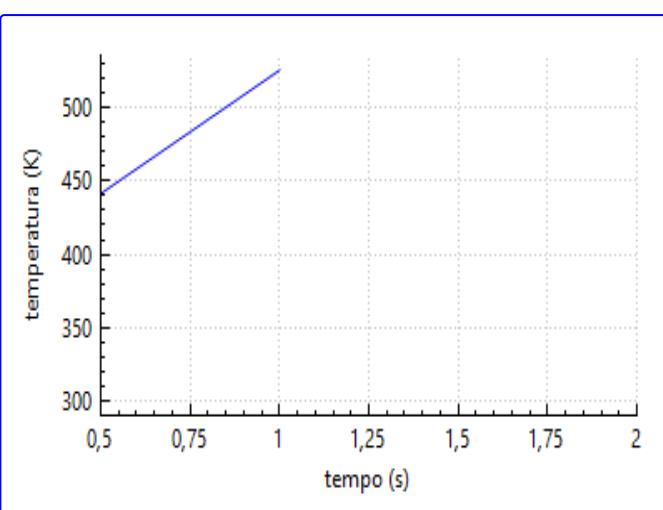
Temperatura máxima: 1000 K

Temperatura mínima: 300 K

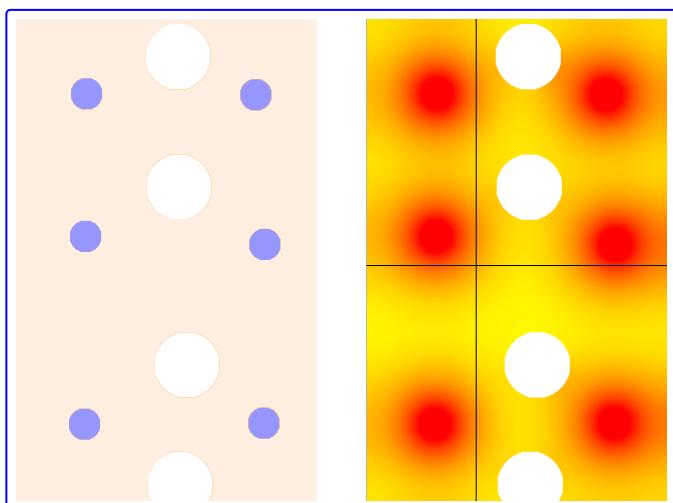
Tempo máximo: 1 s

Tipo de paralelismo: Paralelismo total

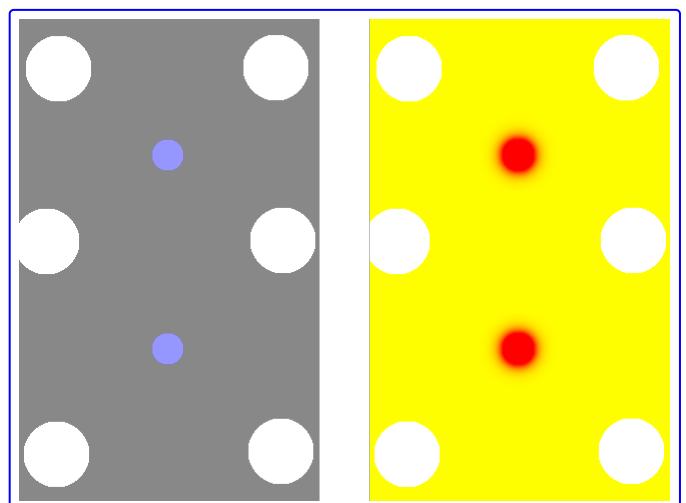
Coordenada do ponto de estudo (x,y,z): 0.02834,0.0637,0



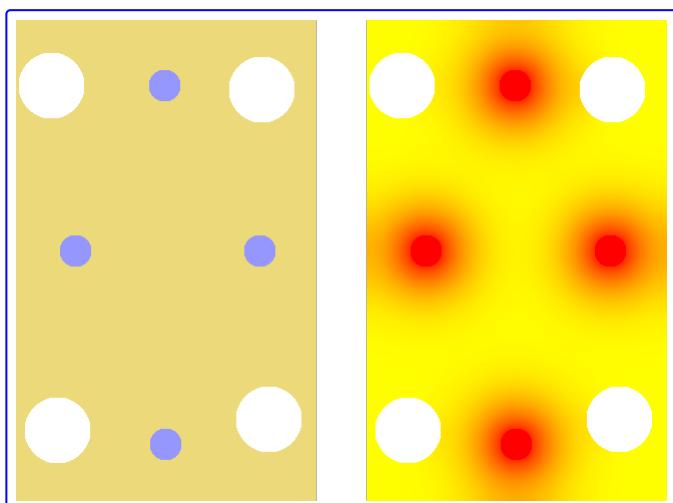
Grid 0



Grid 1



Grid 2



Bibliography

[BUENO 2003] BUENO, A. D. *Programação Orientada a Objeto com C++*. [S.l.]: Novatec, 2003. 12

[Dong, McCartney e Lu 2015] DONG, Y.; MCCARTNEY, J. S.; LU, N. Critical review of thermal conductivity models for unsaturated soils. Springer Science and Business Media LLC, v. 33, n. 2, p. 207–221, 2015. 7, 108

[FOURIER 1822] FOURIER, J. B. J. *Theorie Analytique de la Chaleur*. [s.n.], 1822. ISBN 978-1108001809. Disponível em: <https://www.ebook.de/de/product/8770220/jean_baptiste_joseph_fourier_theorie_analytique_de_la_chaleur.html>

[Herter e Lott] HERTER, T.; LOTT, K. Algorithms for decomposing 3-d orthogonal matrices into primitive rotations. Elsevier BV, v. 17, n. 5, p. 517–527. 26

[Incropera 2008] INCROPERA, F. *Fundamentos de transferência de calor e de massa*. [S.l.]: LTC, 2008. ISBN 8521615841. 14, 16, 21, 103

[Lima 2020] LIMA, G. R. *Simulador bidimensional de transferência de calor em meios porosos utilizando métodos numéricos de diferenças finitas*. 2020. 108

[NUSSENZVEIG 2014] NUSSENZVEIG, H. M. *Curso de física básica 2 : fluidos, oscilações e ondas, calor*. [S.l.] : Blucher, 2014. ISBN 978-85-212-0747-4. 12, 16

[RESNICK 2009] RESNICK, D. H. J. W. *Fundamentos de física, volume 2: gravitação, ondas e termodinâmica*. [S.l.: s.n.], 2009. ISBN 978-85-216-1606-1. 16

[Rosa, Carvalho e Xavier 2006] ROSA, A. J.; CARVALHO, R. D. S.; XAVIER, J. A. D. *Engenharia de reservatórios de petróleo*. [S.l.] : Intercincia, 2006. ISBN 85 – 7193 – 135 – 6. 2, 16

[THOMAS 2004] THOMAS, J. E. Fundamentos de engenharia de petróleo. [S.l. : s.n.], 2004. ISBN 85 – 7193 – 046 – 5. 2

[Valencia e Quested 2008] VALENCIA, J. J.; QUESTED, P. N. *Thermophysical Properties*. [S.l.], 2008. 22

Chapter 11

Disciplinas Relacionadas a Transferência de Calor

A Figura 11.1 ilustra o relacionamento das disciplinas do Curso de Engenharia de Petróleo da UENF que tem uma ligação mais direta com a transferência de calor.

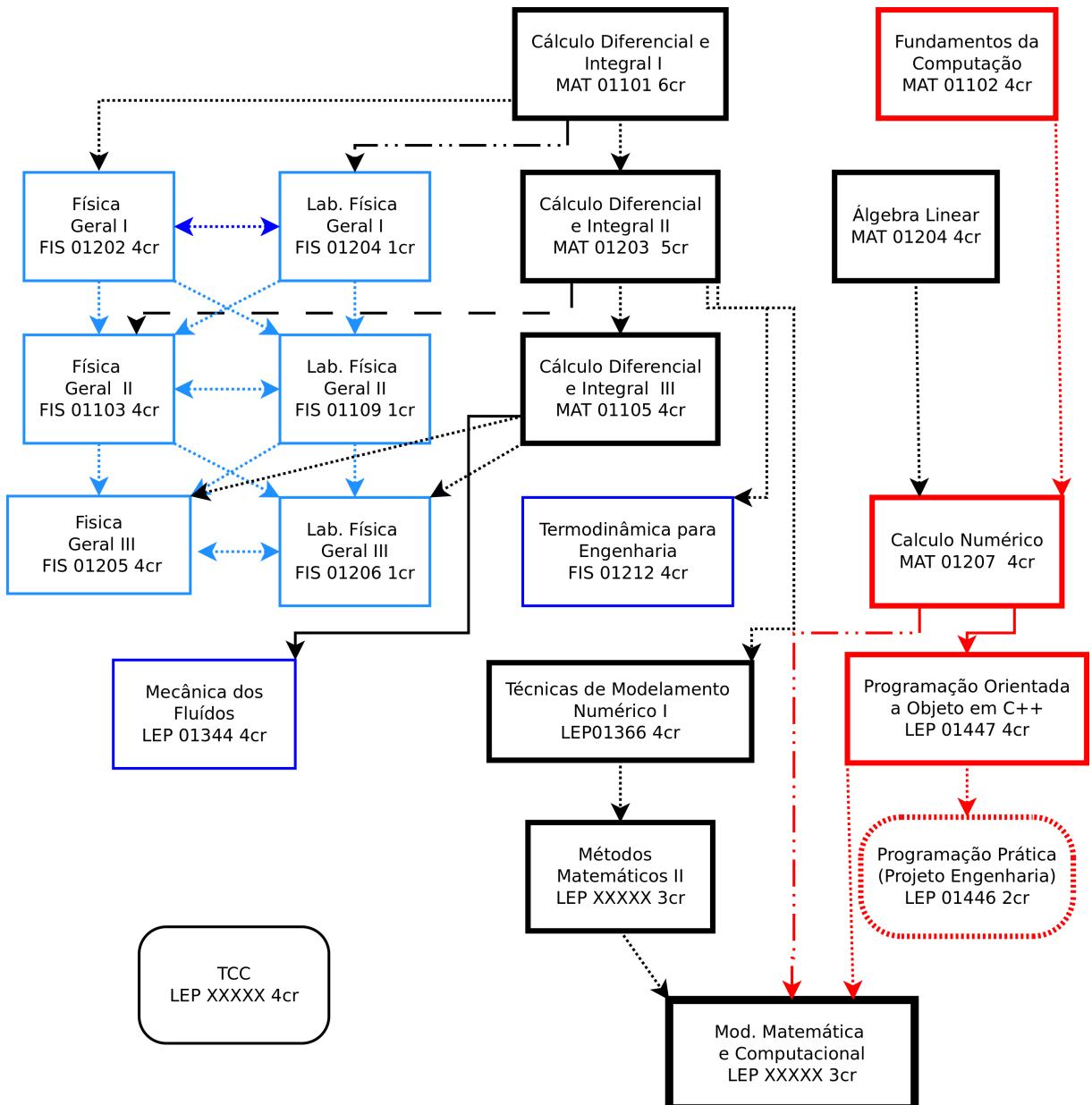


Figure 11.1: Principais disciplinas do curso relacionadas a Transferência de Calor

A título ilustrativo um resumo das ementas de algumas destas disciplinas é apresentado a seguir:

- Matemática:

- MAT01101 : Cálculo Diferencial e Integral I : 1- Funções. 2- Derivada. 3- Integral.
- MAT01203 : Cálculo Diferencial e Integral II : 1- Algumas superfícies especiais. 2- Funções vetoriais de uma variável real. 3- Funções reais de várias variáveis reais. 4- Derivadas parciais e diferenciabilidade. 5- Máximos e mínimos.
- MAT01109 : Cálculo Diferencial e Integral III : 1- Integrais dupla e tripla. 2- Tópicos em análise vetorial. 3- Integral de linha. 4- Integral de superfície. 5- Teoremas fundamentais.

–MAT01204 : Álgebra Linear: 1- Vetores em Rn. 2- Sistema linear de equações. 3- Espaços vetoriais. 4- Transformações lineares. 5- Autovalores e autovetores.

•Física:

–FIS01202 : Física Geral I: 1- Sistemas de medidas; 2- Movimento em uma dimensão; 3- Movimento em duas e três dimensões; 4- Leis de Newton; 5- Aplicações das Leis de Newton; 6- Trabalho e energia; 7- Conservação de energia; 8- Sistemas de partículas e conservação da quantidade de movimento linear; 9- Rotação; 10- Conservação da quantidade de movimento angular; 11- Equilíbrio estático e elasticidade; 12- Gravitação; 13- Fluídios.

–FIS01103 : Física Geral II: 1- Lei Zero da Termodinâmica; 2- Primeira e Segunda Lei da Termodinâmica; 3- Teoria Cinética dos Gases; 4- Propriedades Térmicas e Processos Térmicos; 5- Carga; 6- Lei de Coulomb; 7- Campo Elétrico; 8- Lei de Gauss; 9- Capacitores e Potencial Elétrico; 10- Campo Magnético; 11- Lei de Biot-Savart; 12- Lei de Ampère; 13- Indutores; 14- Leis de Maxwell.

–FIS01205 : Física Geral III: 1- Oscilações; 2- Ondas mecânicas; 3- Ondas Eletromagnéticas; 4- Propriedades da luz; 5- Interferência; 6- Difração; 7- Fóttons, Ondas de Matéria; 8- Tópicos Especiais: Ótica Geométrica, Relatividade.

–FIS01212 : Termodinâmica para Engenharia: 1- Conceitos. 2- Definições e Princípios Básico da Termodinâmica. 3- Propriedades de Substâncias Puras. 4- Leis da Termodinâmica. 5- Ciclos Motores e de Refrigeração. 6- Relações Termodinâmicas. 7- Sistemas Termodinâmicos. 8- Misturas e Soluções de Gases Perfeitos. 9- Equilíbrio Químico.

–LEP01344 : Mecânica dos Fluidos: 1- Estática dos fluidos. 2- Determinação experimental de propriedades dos fluidos. 3- Aplicações da Cinemática dos Fluidos. 4- Aplicações da Dinâmica dos fluidos. 5- Escoamento em condutos. 6- Análise dimensional, semelhança e modelos.

–LEP1852 : Transferência de Calor: 1- Introdução á Transferência de Calor. 2- Condução Unidimensional em Regime Permanente. 3- Fundamentos da convecção e da Radiação. 4- Trocadores de Calor. [optativa].

•Modelagem numérica:

–MAT01207 : Cálculo Numérico: I. Sistemas Numéricos e Erros. II. Zeros de Funções Reais. III. Matrizes e Resolução Numérica de Sistemas de Equações Lineares. IV. Interpolação. V. Integração Numérica. VI. Resolução Numérica de Equações Diferenciais Ordinárias.

- LEP01366 : Técnicas de Modelamento Numérico I: 1- Equações em diferenças.
2- Métodos de diferenças finitas para equações diferenciais ordinárias. 3- Métodos de diferenças finitas para equações diferenciais parciais.
- LEP01366 : Técnicas de Modelamento Numérico II.

•Modelagem computacional:

- MAT01102 : Fundamentos da Ciência da Computação : 1. Introdução à Computação. 2. Introdução à Programação. Algoritmos. 3. Programação. 4. Uso de Programas Aplicativos.
- LEP01447 : Programação Orientada a Objeto em C++ : 1- Filosofia, modelagem e conceitos. Diagramas usando a modelagem UML. 2- Etapas de desenvolvimento de um programa. 3- Sintaxe e conceitos de C++, tipos, classes, objetos, atributos, métodos. 4- Herança, polimorfismo, sobrecarga de função e de operadores. 5- Entrada/saída e as classes string e complex. 6- Introdução a STL, containers, iteradores, funções genéricas.
- LEP01579 : Programação Paralela e Concorrente I: 1- Introdução ao processamento paralelo e concorrente; 2 - Processamento paralelo com múltiplos processos; 3- Processamento paralelo usando Boost.Interprocess; 4- Introdução ao processamento paralelo com múltiplas threads de C; 5- Processamento paralelo com múltiplas threads de C++11/14/17/20; 7- Introdução ao processamento paralelo em um cluster de computadores; [optativa].

•Aplicação:

- LEP01446 : Programação Prática: Desenvolvimento de um programa de engenharia utilizando a sequência padrão: Especificação do sistema, elaboração, desenvolvimento da análise orientada a objeto (diagramas usando UML), desenvolvimento do projeto do sistema, desenvolvimento do projeto orientado a objeto, implementação do programa usando C++, testes do funcionamento do programa; manutenção e documentação do programa desenvolvido.

Index

- Análise de domínio, 12
Análise orientada a objeto, 31

Casos de uso, 9
Ciclo construção, 48
Ciclos de Planejamento/Detalhamento, 42
Como adicionar materiais, 125
Como gerar relatório em PDF, 127
Concepção, 5
Condição de fronteira, 18
Condutividade térmica variável, 21

Demonstrações matemáticas, 20
Dependências, 123
Diagrama de atividades, 37
Diagrama de colaboração, 35
Diagrama de componentes, 39
Diagrama de comunicação, 35
Diagrama de implantação-execução, 40
Diagrama de máquina de estado, 36
Diagrama de pacotes, 29
Diagrama de sequência, 33
Dicionário das classes, 31
Documentação, 121
Documentação do desenvolvedor, 121

Elaboração, 12
Escopo do problema, 1
especificação, 6
estado, 36

Formulação modelos computacionais, 22
Formulação modelos teóricos, 13
Formulação teórica, 14

Identificação de pacotes, 29

Implementação, 48
Injeção de calor em reservatório, 108
Instalação, 123
Interface gráfica, 123

Método da correlação ou constante, 126
Método de interpolação, 126
múltiplas-threads, 24
Manual do Usuário, 123
Metodologia utilizada, 4
modelo five-spot, 110

Objetivos, 3

processamento paralelo, 22
Projeto, 39
Projeto do sistema, 39

Renderização 3D, 25
Requisitos funcionais, 7
Requisitos não funcionais, 8
Resfriamento de processadores, 119

Termos e Unidades, 13
Teste, 103

Validação do simulador, 103
Versão 0.3 código fonte, 48
Versão 0.5 código fonte, 48