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

PROJETO ENGENHARIA SOFTWARE 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

Sumário

Capítulo 1

Introdução

1.1 Escopo do problema

O estudo do calor, é uma das principais áreas da física. Seu comportamento foi especulado desde os primeiros filósofos, com a grande ascensão no período de 1600 e 1800, posteriomente, dominado. Contou com a contribuição científica de diversos grandes nomes da física, como Newton e Fourier[FOURIER, 1822].

Na indústria do petróleo, esse problema é analisado cuidadosamente em todas as etapas, desde a exploração, quanto na produção e refino. Na exploração, é buscado um óleo maturado sob temperaturas entre 65^{o}C e 165^{o}C, acima de 180^{o}C, é propiciado a formação de gases leves, e acima de 210^{o}C, a formação de grafite [THOMAS, 2004].

Na produção, são utilizados trocadores de calor nas plataformas. Na engenharia de reservatórios, é utilizado vapor de água para aquecer o petróleo no reservatório como método de recuperação avançada [Rosa et al., 2006].

Desta forma, torna-se claro a importância do estudo da transferência de calor em regime transiente (quando a temperatura varia com o tempo), em objetos 3D com geometria complexa, e constituído por diversos materiais com propriedades físicas variáveis.

A solução analítica do problema do calor é possível para casos unidimensionais, com condições de contorno e iniciais bem definidas, e com propriedades dos materiais, constantes. Para resolver o problema do parágrafo anterior, é necessário a utilização de modelagem numérica computacional.

Conforme a complexidade do problema for evoluindo, a modelagem numérica exige cada vez mais do poder de processamento dos computadores, os quais, atualmente, possuem diversos núcleos lógicos capazes de resolverem cálculos numéricos independentemente, acelerando as simulações.

Para permitir as divisões das tarefas para os processadores, é utilizado uma linguagem de programação de baixo nível, com comandos mais próximos da linguagem da máquina.

Por fim, o ensino de engenharia pode ser aperfeiçoada 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 que resolve o problema de transferência de calor, utilizando métodos numéricos, programado orientado ao objeto com a linguagem C++, utilizando paralelismos e *multithreading,* e renderização 3D.

A finalidade deste projeto de engenharia é desenvolver um *software* capaz de resolver problemas de transferência de calor em quaisquer objeto 3D, constituído por qualquer material com propriedades dependente da temperatura, com interface de usuário amigável, e com renderização 3D, permitindo a visualização do problema.

Os principais tópicos atacados por este projeto são:

- Transferência de calor: entender as equações físicas, como o calor é propagado em materiais com diversos formatos e materiais, e com propriedades termofísicas variáveis, podendo estas serem obtidas em laboratório, adicionadas ao *software* e utilizados para simulação.

- Modelagem numérica: solução da equação diferencial da conservação de temperatura por meio de diferenças finitas, com o método implícito BTCS (*Backward Time, Centered Space*), e com condições de contorno que podem ser aplicadas em todo o sistema, permitindo geometrias 3D complexas.

- Programação em C++: por meio da orientação ao objeto em C++, o problema pode ser dividido em classes, paradigma este que melhora o controle e organização do problema, e facilita adaptações e incrementações ao simulador. Além disso, essa linguagem possui bibliotecas que auxiliam a utilização de paralelismos e *multithreading.*

- Interface ao usuário: com integração total ao core do simulador, a interface permite que o usuário tenha liberdade total para modificar as propriedades da simulação. E com a implementação de renderização 3D, é possível observar o objeto de maneira alternativa e integral.

Capítulo 2

Especificação

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

2.1 Características gerais

O *software* Simulador de difusão térmica 3D, é um software programado sob o Paradigma Orientado ao Objeto em C++, capaz de simular a transferência de calor em objetos tridimensionais, com formas e superfícies complexas e definidas pelo usuário, o qual também 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, e salvar os resultados em formato pdf.

Para resolver o problema da transferência do calor, é modelado a equação diferencial por diferenças finitas implícitas, especificamente pelo método BTCS (*Backward Time Centered Space*), que é incondicionalmente estável. As condições iniciais, são inseridas pelo usuário, e as condições de contorno externo são definitas por regiões que não trocam calor com o meio externo (condição de contorno de Neumann).

Tabela 2.1: Características básicas do programa

Nome	Simulador de difusão térmica 3D
Componentes principais	Método numérico implícito BTCS. Métodos de correlação e interpolação para propriedades termofísicas. Renderização 3D.
Missão	Simulador de transferência de calor em objetos 3D com superfícies complexas, formado por materiais com propriedades depententes da temperatura. E auxiliar no ensino das diversas disciplinas abrangidas por este trabalho.

2.2 Especificação

Deseja-se desenvolver um software com interface gráfica amigável ao usuário, onde seja possível desenhar o objeto 3D, por meio de perfis, com o usuário escolhendo a temperatura e o material. A simulação é governada pela Equação da Difusão Térmica, a qual é modelada por diferenças finitas, pelo método BTCS, com fronteiras seladas.

Na dinâmica de execução, o usuário deverá escolher o tamanho do objeto, a temperatura, em qual perfil está desenhando, o material e suas propriedades termofísicas, e um ponto de monitoramento e estudo. 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 do usuário e colocado o simulador para rodar, o simulador irá calcular a temperatura iterativamente em cada ponto, e só passará para o próximo tempo se o erro entre iterações for menor que um valor aceitável. Posteriormente, o desenho será atualizado e mostrará a nova distribuição de temperatura, e plotará os gráficos com os novos valores calculados.

O software será programado em C++, com paradigma orientado ao objeto, utilizando a biblioteca *Qt* para criar a interface do usuário, e *qcustomplot* para gerar os gráficos.

Para calcular as propriedades termofísicas dos materiais, são utilizados três modelos: propriedades constantes, por correlação e por interpolação.

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

- Dados relativos ao material:
  - c_p - capacidade térmica $[J/g \cdot K]$
  - 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=0.0026m $[m]$;
  - Δz - distância entre perfis, valor inicial: 0.05m $[m]$;
  - 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;
  - ν - número da iteração.

2.2.1 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, escolhendo também sua temperatura em cada ponto.

RF-03	O usuário tem a liberdade de escolher o material em cada ponto do objeto, juntamente com o método para obter as propriedades termofísicas.

RF-04	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-05	O usuário poderá escolher uma região de fonte ou sumidouro.

RF-06	O usuário poderá salvar e/ou carregar dados da simulação.

RF-07	O usuário poderá salvar os resultados da simulação em um arquivo pdf.

RF-08	O usuário pode adicionar materiais no simulador, e escolhar a forma de calcular suas propriedades termofísicas: constante, correlação ou interpolação.

RF-09	O usuário poderá comparar as propriedades termofísicas dos materiais.

RF-10	O usuário poderá acompanhar a evolução da temperatura em uma superfície 2D em todo intervalo de tempo.

RF-11	O usuário poderá visualizar o objeto 3D desenhado em uma janela separada.

2.2.2 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 *Windows*, *GNU/Linux* ou *Mac*.

RNF-03	A performance do programa pode ser alterada com a mudança do modelo de paralelismo.

RNF-04	A interface gráfica deve ser desenvolvida pela biblioteca Qt.

RNF-05	O usuário pode definir as propriedades físicas da simulação, como intervalo de tempo e espaço.

2.3 Casos de uso

2.3.1 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, rodando a simulação, analisando os resultados e salvando o objeto e resultados em pdf.

Tabela 2.2: Exemplo de caso de uso

Nome do caso de uso:	Cálculo da temperatura
Resumo/descrição:	Cálculo da distribuição de temperatura em determinadas condições.
Etapas:	1. Escolha da temperatura e do material 2. Desenhar o objeto desenhado 3. Escolher um ponto de estudo 4. Rodar a simulação e analisar resultados 5. Salvar objeto e 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.

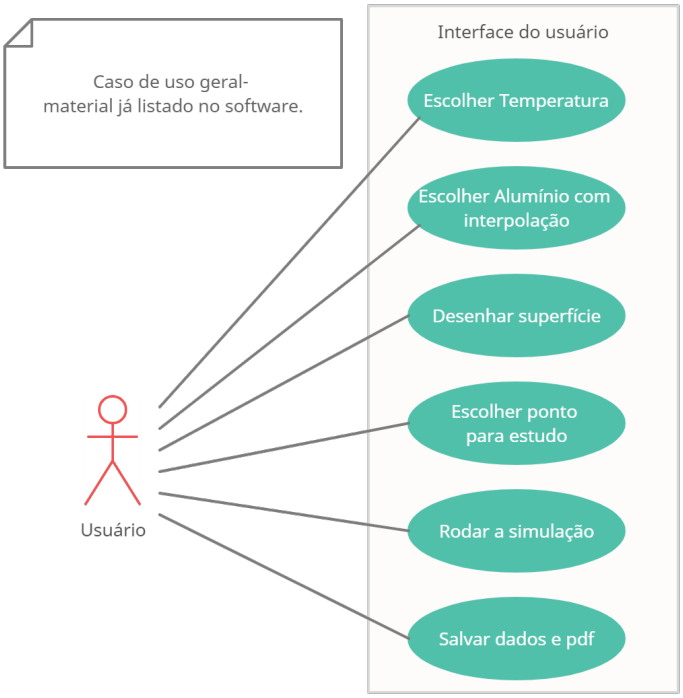


Figura 2.1: Diagrama de caso de uso – Caso de uso geral

2.3.2 Diagrama de caso de uso específico

O caso de uso específico na 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.

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

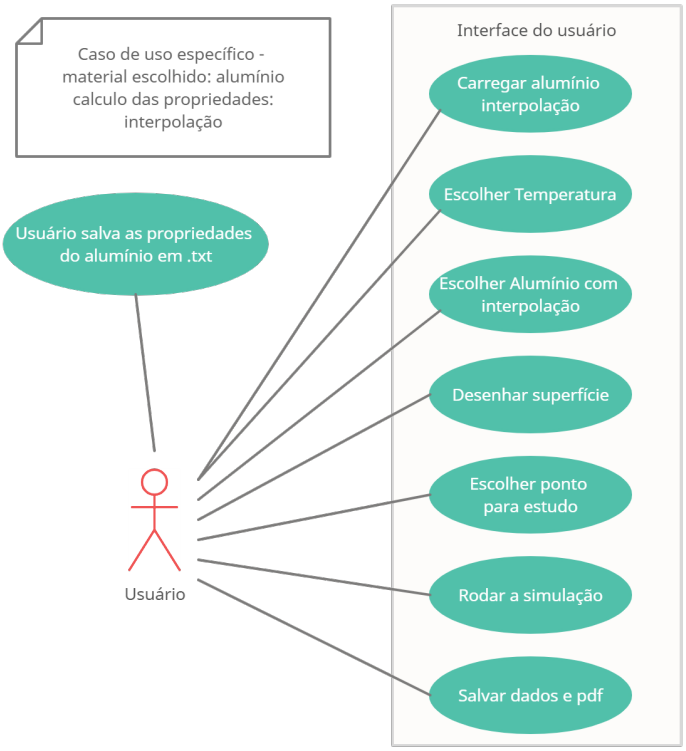


Figura 2.2: Diagrama de caso de uso específico

Capítulo 3

Elaboração

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

3.1 Análise de domínio

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

O presente trabalho pode ser dividido em quatro conceitos fundamentais:

1. Transferência de calor:

   Transferência de calor é uma subárea da termodinâmica, a qual é área 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 movi emento 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 orientada ao objeto com C++:

   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 e por permitir a orientação ao objeto.

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

3.2.1 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 (3.1). O desenvolvimento teórico para chegar nesta equação, pode ser acompanhado detalhadamente no [Incropera, 2008].

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

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

Onde ρ é a massa específica em $[kg/m^3]$, c_p é a capacidade térmica em $[J/(kg\cdot K)]$, k é a condutividade térmica em $[W/(m\cdot K)]$.

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

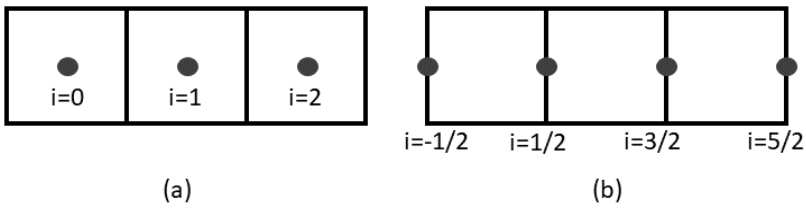


Figura 3.1: Tipos de malha, (a) bloco-centrado e (b) ponto-distribuído.

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:

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

Modelando as derivadas internas:

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

Com um pouco de álgebra:

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

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

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

Como as outras dimensões são simétricas:

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

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

A derivada temporal é atrasada no tempo:

$$\frac{\partial T}{\partial t} = \frac{T_{i,j}^{n+1} - T_{i,j}^{n}}{\Delta t} \tag{3.8}$$

Substituindo as diferenças finitas na equação geral:

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

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

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

Multiplicando pelo múltiplo comum:

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

Como a equação acima é complexa para ser reorganizada e resolvida por equações matriciais, será utilizado aproximações implícitas para resolver esse problema, para calcular a iteração $\nu+1$, será utilizado:

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

Onde C_1é a constante:

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

Agora, é necessário definir o cálculo das condutividades térmicas nas fronteiras. Para isso, será feito um análogo com a permeabilidade de rochas em série [Rosa et al., 2006], mas utilizando as equações de calor:

$$q_x=-kA\frac{dT}{dx}=-\frac{kA}{L}\Delta T \tag{3.14}$$

Isolando a diferença de temperatura:

$$\Delta T = -\frac{Lq_x}{kA} \tag{3.15}$$

A Figura 3.2 mostra um caso de condutividades térmicas em série. O calor (q) que entra no sistema, é igual ao que sai. E a diferença de temperatura entre a esquerda (0) e a direita (2), é soma das diferenças nesse meio, ou seja:

$$T_0 - T_2 = (T_0 - T_1) + (T_1 - T_2) \tag{3.16}$$

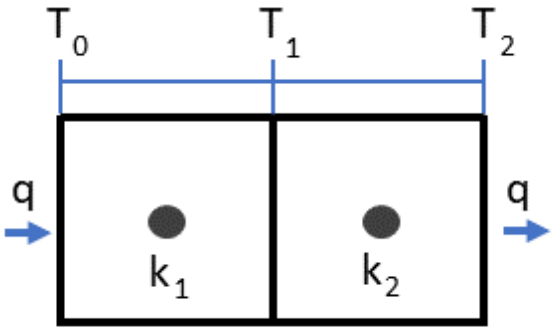


Figura 3.2: Representação de condutividade térmica em série.

Logo,

$$\Delta T_t = \Delta T_1 + \Delta T_2 \tag{3.17}$$

Onde as taxas de transferência de calor são:

$$-\frac{2Lq}{k_r A} = -\frac{Lq}{k_1 A} - \frac{Lq}{k_2 A} \tag{3.18}$$

Com alguns ajustes algébricos:

$$\frac{2}{k_r} = \frac{1}{k_1} + \frac{1}{k_2} \tag{3.19}$$

Ou, simplesmente:

$$k_r = \frac{2k_1 k_2}{k_1 + k_2} \tag{3.20}$$

É importante analisar a célula computacional, ou a região que é observada quando um ponto é calculado. 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 o ponto vermelho, é utilizado o mesmo ponto, mas no tempo anterior, e uma célula em cada sentido.

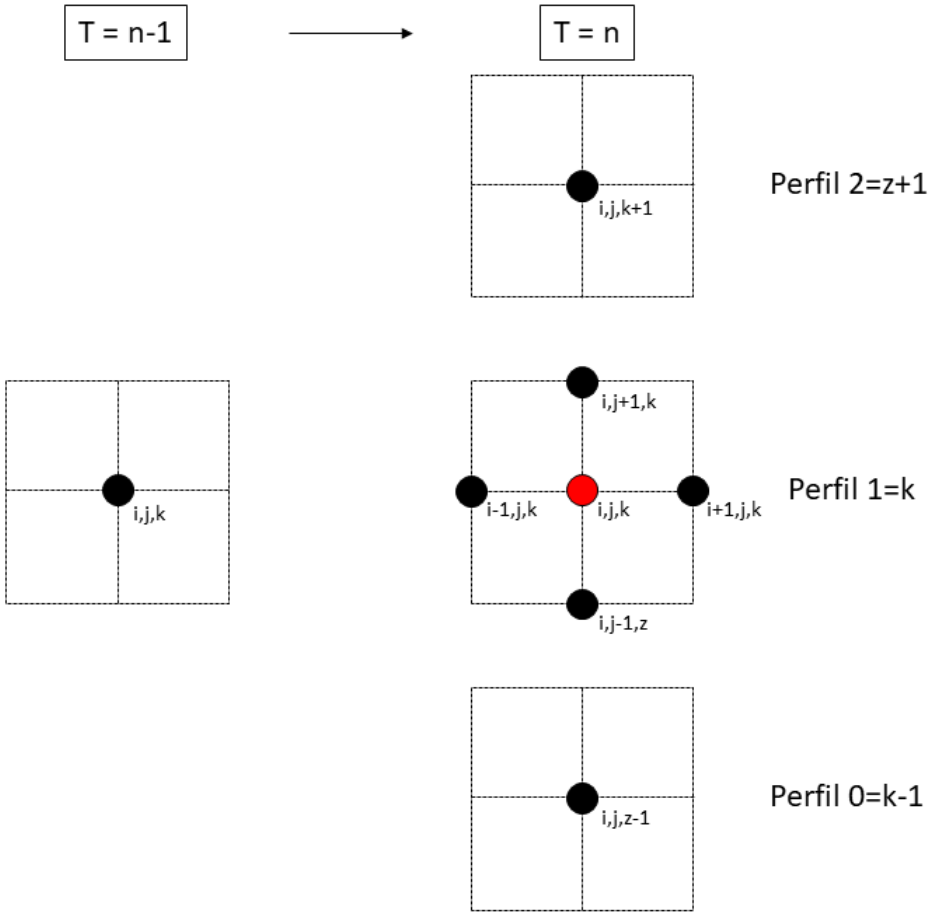


Figura 3.3: Malha utilizada para calcular um ponto de temperatura, cada ponto é o centro dos blocos.

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

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 que, nessa região, o objeto 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.

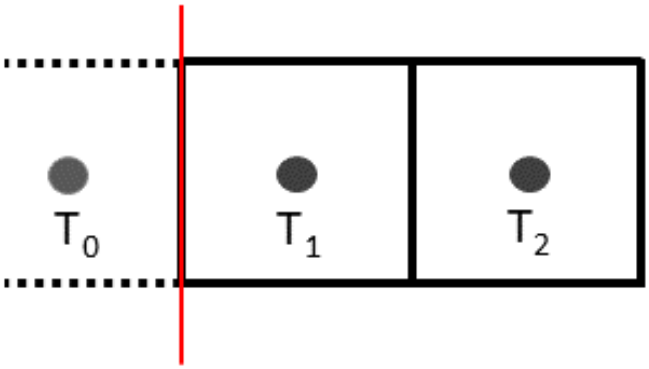


Figura 3.4: Análise da fronteira de Neumann.

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

$$k\frac{\partial T}{\partial x}_{i-\frac{1}{2},j,k} = 0\frac{T^{n+1}_{i,j,k} - T^{n+1}_{i-1,j,k}}{\Delta x} = 0 \tag{3.21}$$

A equação acima possui duas soluções:

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

Resolvendo a linha de baixo:

$$\frac{\partial T}{\partial x}_{i-\frac{1}{2},j,k} = \frac{T^{n+1}_{i,j,k} - T^{n+1}_{i-1,j,k}}{\Delta x} = 0 \tag{3.23}$$

$$T^{n+1}_{i-1,j,k} = T^{n+1}_{i,j,k} \tag{3.24}$$

Todas as seis fronteiras são simétricas, então:

$$\begin{aligned} T^{n+1}_{i-1,j,k} &= T^{n+1}_{i,j,k} \\ T^{n+1}_{i+1,j,k} &= T^{n+1}_{i,j,k} \\ T^{n+1}_{i,j-1,k} &= T^{n+1}_{i,j,k} \\ T^{n+1}_{i,j+1,k} &= T^{n+1}_{i,j,k} \\ T^{n+1}_{i,j,k-1} &= T^{n+1}_{i,j,k} \\ T^{n+1}_{i,j,k+1} &= T^{n+1}_{i,j,k} \end{aligned} \tag{3.25}$$

As equações encontradas na Eq. 3.25 dizem que, se existir uma fronteira, a temperatura inexistente deve ser substituída pela temperatura do próprio ponto. Ou, 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.

Demonstrações

Nesta parte, será analisado 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, 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 com sua temperatura interna ao longo do tempo. Então, partindo da equação geral:

$$\begin{gathered}
\overset{\nu+1}{T}{}^{n+1}_{i,j} = \\
C_1 \frac{\Delta z^2 \Delta x^2 c_p \rho}{\Delta t} \overset{\nu}{T}{}^{n}_{i,j} + \\
C_1 \Delta z^2 \left(\overset{\nu}{k}{}^{n+1}_{i-\frac{1}{2},j,k} \overset{\nu}{T}{}^{n+1}_{i-1,j,k} + \overset{\nu}{k}{}^{n+1}_{i+\frac{1}{2},j,k} \overset{\nu}{T}{}^{n+1}_{i+1,j,k} \right) + \\
C_1 \Delta z^2 \left(\overset{\nu}{k}{}^{n+1}_{i,j-\frac{1}{2},k} \overset{\nu}{T}{}^{n+1}_{i,j-1,k} + \overset{\nu}{k}{}^{n+1}_{i,j+\frac{1}{2},k} \overset{\nu}{T}{}^{n+1}_{i,j+1,k} \right) + \\
C_1 \Delta x^2 \left(\overset{\nu}{k}{}^{n+1}_{i,j,k-\frac{1}{2}} \overset{\nu}{T}{}^{n+1}_{i,j,k-1} + \overset{\nu}{k}{}^{n+1}_{i,j,k+\frac{1}{2}} \overset{\nu}{T}{}^{n+1}_{i,j,k+1} \right)
\end{gathered} \tag{3.26}$$

$$\frac{1}{C_1} = \begin{array}{l} \frac{\Delta z^2 \Delta x^2 c_p \rho}{\Delta t} + \\ \\ \end{array} \begin{array}{l} \Delta z^2 \left(\overset{\nu}{k}{}^{n+1}_{i-\frac{1}{2},j,k} + \overset{\nu}{k}{}^{n+1}_{i+\frac{1}{2},j,k} \right) + \\ \Delta z^2 \left(\overset{\nu}{k}{}^{n+1}_{i,j-\frac{1}{2},k} + \overset{\nu}{k}{}^{n+1}_{i,j+\frac{1}{2},k} \right) \quad + \Delta x^2 \left(\overset{\nu}{k}{}^{n+1}_{i,j,k-\frac{1}{2}} + \overset{\nu}{k}{}^{n+1}_{i,j,k+\frac{1}{2}} \right) \end{array} \tag{3.27}$$

Mas, como demonstrado na Eq. 3.22, quando houver fronteira, a condutividade térmica na fronteira é zero:

$$\begin{gathered}
\overset{\nu+1}{T}{}^{n+1}_{i,j} = \\
C_1 \frac{\Delta z^2 \Delta x^2 c_p \rho}{\Delta t} \overset{\nu}{T}{}^{n}_{i,j} + \\
C_1 \Delta z^2 \left(\cancelto{0}{\overset{\nu}{k}{}^{n+1}_{i-\frac{1}{2},j,k}} \overset{\nu}{T}{}^{n+1}_{i-1,j,k} + \cancelto{0}{\overset{\nu}{k}{}^{n+1}_{i+\frac{1}{2},j,k}} \overset{\nu}{T}{}^{n+1}_{i+1,j,k} \right) + \\
C_1 \Delta z^2 \left(\cancelto{0}{\overset{\nu}{k}{}^{n+1}_{i,j-\frac{1}{2},k}} \overset{\nu}{T}{}^{n+1}_{i,j-1,k} + \cancelto{0}{\overset{\nu}{k}{}^{n+1}_{i,j+\frac{1}{2},k}} \overset{\nu}{T}{}^{n+1}_{i,j+1,k} \right) + \\
C_1 \Delta x^2 \left(\cancelto{0}{\overset{\nu}{k}{}^{n+1}_{i,j,k-\frac{1}{2}}} \overset{\nu}{T}{}^{n+1}_{i,j,k-1} + \cancelto{0}{\overset{\nu}{k}{}^{n+1}_{i,j,k+\frac{1}{2}}} \overset{\nu}{T}{}^{n+1}_{i,j,k+1} \right)
\end{gathered} \tag{3.28}$$

$$\frac{1}{C_1} = \begin{array}{l} \frac{\Delta z^2 \Delta x^2 c_p \rho}{\Delta t} + \quad \Delta z^2 \left(\cancel{k^{n+1}_{i-\frac{1}{2},j,k}}^{\nu\,0} + \cancel{k^{n+1}_{i+\frac{1}{2},j,k}}^{\nu\,0} \right) + \\ \Delta z^2 \left(\cancel{k^{n+1}_{i,j-\frac{1}{2},k}}^{\nu\,0} + \cancel{k^{n+1}_{i,j+\frac{1}{2},k}}^{\nu\,0} \right) \quad + \Delta x^2 \left(\cancel{k^{n+1}_{i,j,k-\frac{1}{2}}}^{\nu\,0} + \cancel{k^{n+1}_{i,j,k+\frac{1}{2}}}^{\nu\,0} \right) \end{array} \tag{3.29}$$

Resultando em:

$$\overset{\nu+1}{T^{n+1}_{i,j}} = C_1 \frac{\Delta z^2 \Delta x^2 c_p \rho}{\Delta t} \overset{\nu}{T^n_{i,j}} \tag{3.30}$$

$$\frac{1}{C_1} = \frac{\Delta z^2 \Delta x^2 c_p \rho}{\Delta t} \tag{3.31}$$

Logo:

$$\overset{\nu+1}{T^{n+1}_{i,j}} = \frac{\Delta t}{\Delta z^2 \Delta x^2 c_p \rho} \frac{\Delta z^2 \Delta x^2 c_p \rho}{\Delta t} \overset{\nu}{T^n_{i,j}} \tag{3.32}$$

$$\overset{\nu+1}{T^{n+1}_{i,j}} = \overset{\nu}{T^n_{i,j}} \tag{3.33}$$

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. Partindo da equação geral:

$$\begin{array}{c} T^{n+1}_{i,j} = \\ C_1 \frac{\Delta z^2 \Delta x^2 c_p \rho}{\Delta t} T^n_{i,j} + \\ C_1 \Delta z^2 \left(k^{n+1}_{i-\frac{1}{2},j,k} T^{n+1}_{i-1,j,k} + k^{n+1}_{i+\frac{1}{2},j,k} T^{n+1}_{i+1,j,k} \right) + \\ C_1 \Delta z^2 \left(k^{n+1}_{i,j-\frac{1}{2},k} T^{n+1}_{i,j-1,k} + k^{n+1}_{i,j+\frac{1}{2},k} T^{n+1}_{i,j+1,k} \right) + \\ C_1 \Delta x^2 \left(k^{n+1}_{i,j,k-\frac{1}{2}} T^{n+1}_{i,j,k-1} + k^{n+1}_{i,j,k+\frac{1}{2}} T^{n+1}_{i,j,k+1} \right) \end{array} \tag{3.34}$$

$$\frac{1}{C_1} = \begin{array}{l} \frac{\Delta z^2 \Delta x^2 c_p \rho}{\Delta t} + \quad \Delta z^2 \left(k^{n+1}_{i-\frac{1}{2},j,k} + k^{n+1}_{i+\frac{1}{2},j,k} \right) + \\ \Delta z^2 \left(k^{n+1}_{i,j-\frac{1}{2},k} + k^{n+1}_{i,j+\frac{1}{2},k} \right) \quad + \Delta x^2 \left(k^{n+1}_{i,j,k-\frac{1}{2}} + k^{n+1}_{i,j,k+\frac{1}{2}} \right) \end{array} \tag{3.35}$$

Com a substituição de todas as condutividades térmicas nas interfaces por k, e simplificando para bidimensional:

$$\begin{array}{c} T^{n+1}_{i,j} = \\ C_1 \frac{\Delta z^2 \Delta x^2 c_p \rho}{\Delta t} T^n_{i,j} + \\ C_1 \Delta z^2 \left(k T^{n+1}_{i-1,j,k} + k T^{n+1}_{i+1,j,k} \right) + \\ C_1 \Delta z^2 \left(k T^{n+1}_{i,j-1,k} + k T^{n+1}_{i,j+1,k} \right) \end{array} \tag{3.36}$$

$$\frac{1}{C_1} = \frac{\Delta z^2 \Delta x^2 c_p \rho}{\Delta t} + \Delta z^2 (k + k) + \Delta z^2 (k + k) \tag{3.37}$$

Com alguns ajustes:

$$\begin{gathered}\frac{1}{C_1}T_{i,j}^{n+1} = \\ \frac{\Delta x^2 c_p \rho}{\Delta t}T_{i,j}^{n} + \\ k\left(T_{i-1,j,k}^{n+1} + T_{i+1,j,k}^{n+1}\right) + \\ k\left(T_{i,j-1,k}^{n+1} + T_{i,j+1,k}^{n+1}\right)\end{gathered} \tag{3.38}$$

$$\frac{1}{C_1} = \frac{\Delta x^2 c_p \rho}{\Delta t} + 2k + 2k \tag{3.39}$$

Logo:

$$\begin{gathered}\left(\frac{\Delta x^2 c_p \rho}{k\Delta t} + 2 + 2\right)T_{i,j}^{n+1} = \\ \frac{\Delta x^2 c_p \rho}{k\Delta t}T_{i,j}^{n} + \\ \left(T_{i-1,j,k}^{n+1} + T_{i+1,j,k}^{n+1}\right) + \\ \left(T_{i,j-1,k}^{n+1} + T_{i,j+1,k}^{n+1}\right)\end{gathered} \tag{3.40}$$

Chegando na equação:

$$\begin{gathered}\frac{c_p \rho}{k}\frac{T_{i,j}^{n+1} - T_{i,j}^{n}}{\Delta t} = \\ \frac{T_{i-1,j,k}^{n+1} - 2T_{i,j}^{n+1} + T_{i+1,j,k}^{n+1}}{\Delta x^2} + \\ \frac{T_{i,j-1,k}^{n+1} - 2T_{i,j}^{n+1} + T_{i,j+1,k}^{n+1}}{\Delta x^2}\end{gathered} \tag{3.41}$$

E essa é a igual implícita discretizada para um sistema homogêneo bidimensional, conforme [Incropera, 2008].

3.2.2 Condutividade térmica variável

A condutividade térmica (k), no projeto desenvolvido, pode variar com o espaço, pelo objeto ser constituído por mais de um material, com condutividade térmica distinta. Mas também pode variar com a temperatura e, consequentemente, com o tempo.

Será fornecido 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 and Quested, 2008]. O modelo proposto, é calculado, em geral, como:

$$k = C_0 + C_1 T - C_2 T^2 \tag{3.42}$$

Onde C_0, C_1e C_2 são constantes da correlação, específico 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.

Assim, é finalizada as demonstrações físico-numérica do problema da difusão térmica. A partir de agora, será elaborado o paralelismo/multithreading, e a renderização 3D.

3.2.3 Paralelismos/multi-thread

Os chips de processadores atuais, são constituídos por vários processadores menores, o que permite que um mesmo processador consiga realizar tarefas distintas. A ideia é separar tarefas distintas, para que um processador não fique travado em uma única tarefa.

Uma analogia para melhorar a explicação é a dos estudantes. Uma sala cheia de estudantes, recebe uma tarefa de resolver uma lista de exercícios. Se todos os exercícios forem resolvidas por um único aluno, levará muito tempo para terminar a tarefa (caso sem paralelismo). Se os alunos dividirem as tarefas entre si, ela será resolvida muito mais rapidamente.

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. Possuí 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 3.5 ilustra melhor esses três casos.

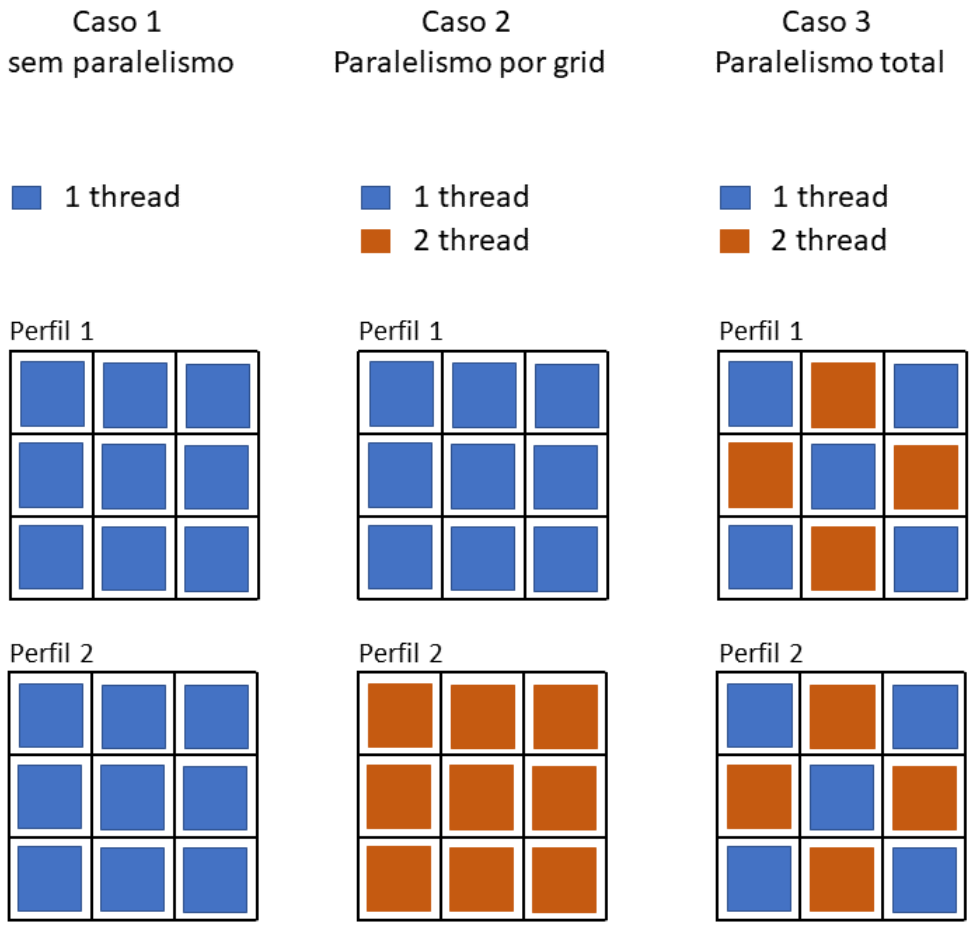


Figura 3.5: Figura ilustrando os três casos de paralelismo implementados para duas camadas com 9 células cada, e um processador com duas threads.

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 threads".

3.2.4 Renderização 3D

Após o usuário desenhar algum objeto no software, pode ser de interesse observar como seria em renderização 3D. Portanto, é implementado 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:

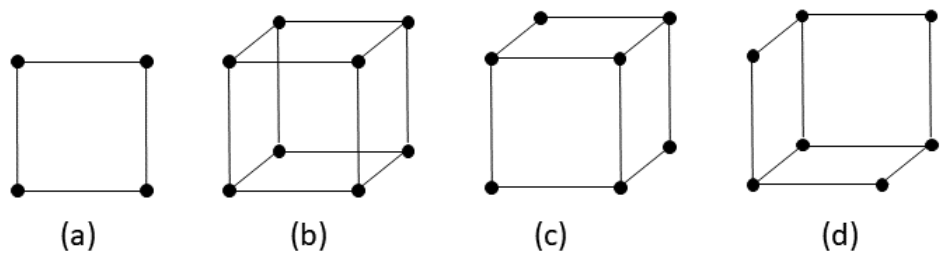


Figura 3.6: (a) Observador alinhado com uma das faces do cubo. (b) observador não está alinhado e não foram removidas arestas ocultas. O cérebro consegue interpretar que é um objeto 3D, mas fica confuso entre os casos (c) e (d).

Todos cantos do cubo da Figura 3.6 estão na mesma posição, o que mudou foi o ângulo do observador com o objeto.

Portanto, tendo em mãos os pontos das arestas, é multiplicado esses vetores com a matriz de rotação do autor [Herter and Lott,] mostrada na Eq. 3.43, a qual permite rotacionar qualquer ponto a partir dos três ângulos do observador.

$$R(\alpha,\beta,\gamma) = \begin{bmatrix} cos(\gamma)cos(\beta) & cos(\gamma)sin(\beta)sin(\alpha) - sin(\gamma)cos(\alpha) & cos(\gamma)sin(\beta)sin(\alpha) + sin(\gamma)cos(\alpha) \\ sin(\gamma)cos(\beta) & sin(\gamma)sin(\beta)sin(\alpha) + cos(\gamma)cos(\alpha) & sin(\gamma)sin(\beta)cos(\alpha) - cos(\gamma)sin(\alpha) \\ -sin(\beta) & cos(\beta) * sin(\alpha) & cos(\beta) * cos(\alpha) \end{bmatrix} \tag{3.43}$$

Ou seja, inicialmente, um cubo de aresta 3 cm, com uma margem de 1 cm, pode ser mostrado na tela (monitor) com os pontos do caso (a) da Figura 3.7, 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 x e y 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.

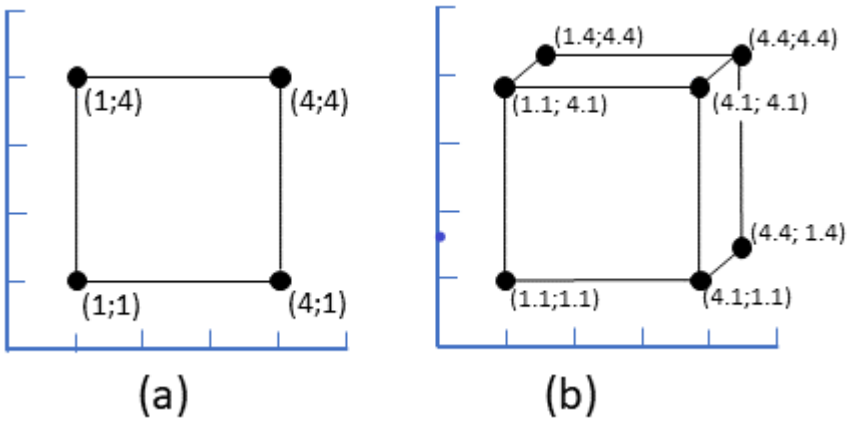


Figura 3.7: (a) o cubo está com ângulos nulos. (b) ângulo x e y estão com valor de 0.1 radianos.

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 numeroso 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.

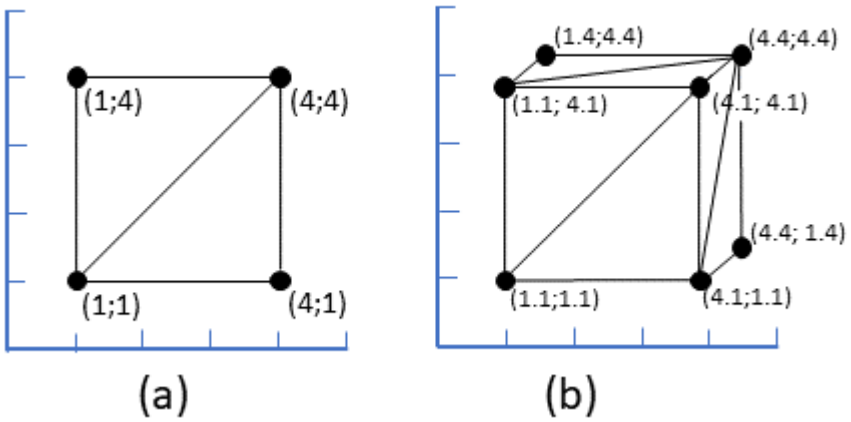


Figura 3.8: Mesmo desenho da figura anterior, mas agora renderizando a partir de triângulos.

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, é obtido dos vetores $\boldsymbol{a} = \{a_1, a_2, a_3\}$ e o vetor $\boldsymbol{b} = \{b_1, b_2, b_3\}$ permitindo a realização do produto vetorial:

$$\boldsymbol{a} \times \boldsymbol{b} = \begin{bmatrix} \boldsymbol{i} & \boldsymbol{j} & \boldsymbol{k} \\ a_1 & a_2 & a_3 \\ b_1 & b_2 & b_3 \end{bmatrix} \tag{3.44}$$

Ou simplesmente:

$$\boldsymbol{a} \times \boldsymbol{b} = (a_2 b_3 - a_3 b_2)\boldsymbol{i} - (a_1 b_3 - a_3 b_1)\boldsymbol{k} + (a_1 b_2 - a_2 b_1)\boldsymbol{j} \tag{3.45}$$

Utilizando a Regra da Mão Direita[1], é possível entender a utilidade da equação 3.45: o caso (a) da figura 3.9, 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]

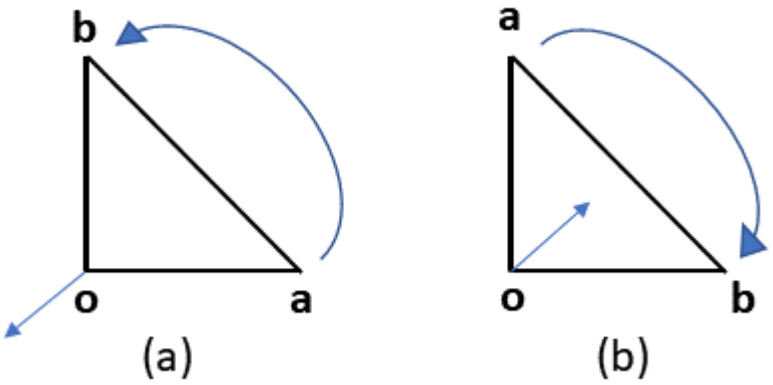


Figura 3.9: (a) mostra um caso onde a normal é na direção do leitor e (b) mostra um caso onde a normal é para dentro da folha.

Essa simples operação condicional do valor positivo/negativo de **j** da normal, reduz a renderização de objetos ocultos, e otimiza o software em duas vezes.

Uma outra condição implementada é a de avaliar se o objeto possui fronteira com outro objeto. Com isso, não é necessário renderizar 4 triângulos dessas duas superfícies em contato. Como estão em contato, não deve ser renderizada sob hipótese alguma.

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

[1]Para utilizar a Regra da Mão Direita, posicione o dedo polegar sobre o ponto **o**, e estique o indicador para o ponto **a**, agora, feche o indicador no sentido do ponto **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.

ocultas, evitando a criação de figuras confusas como no caso (b) da Figura 3.6. É uma técnica lenta, mas de fácil implementação.

3.3 Identificação de pacotes – assuntos

- Pacote de malhas: organiza o objeto desenhado em vetores, facilita o acesso do simulador às propriedades de cada célula.
- Pacote de simulação: nela 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.

3.4 Diagrama de pacotes – assuntos

Abaixo é apresentado o diagrama de pacotes (Figura 3.10).

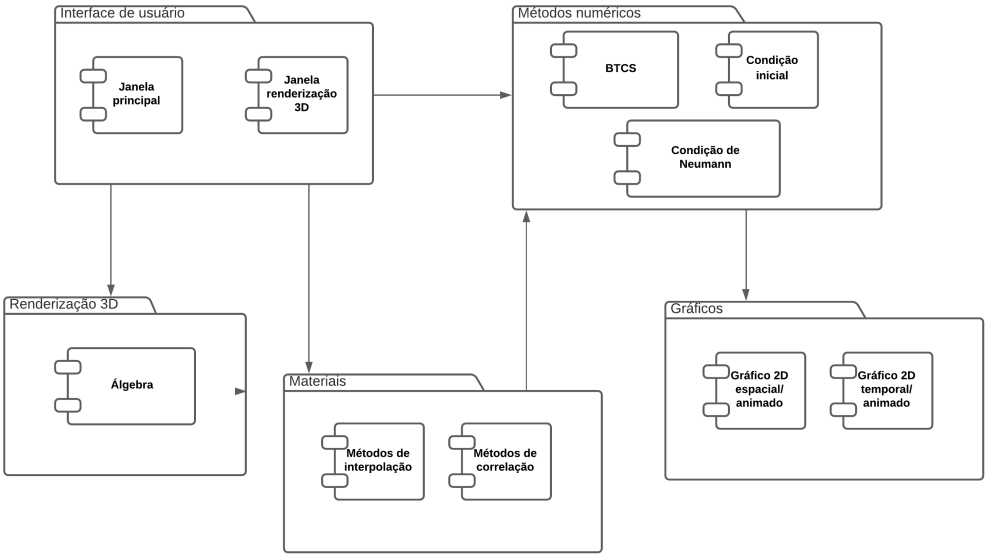


Figura 3.10: Diagrama de Pacotes

Capítulo 4

AOO – Análise Orientada a Objeto

Neste capítulo, será apresentado os objetos desenvolvidos no projeto, suas relações, atributos e métodos. Depois de uma breve explicação sobre cada objeto, será apresentado cinco diagramas de UML, para auxiliar no entendimento do software e suas relações. Serão apresentados os diagramas de classes, de sequência, de comunidação, 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 ensino de C++. Utilizar esse código pronto, mostra que o simulador está apto a adições de métodos para o cálculo das propriedades termofísicas.

1. **mainwindow.h:** Classe responsável pela janela principal. Consegue capturar os valores adicionados pelo usuário, e enviar para a classe do simulador. Permite o usuário desenhar o objeto, e apresenta os resultados por meio de gráficos e pela região de desenho.

2. **CRender3D.h:** Classe responsável por apresentar o objeto em renderização 3D. É criada a partir do mainwindow e recebe valores do simulador. Possui toda a álgebra necessária para a renderização.

3. **CSimuladorTemperatura.h:** Classe responsável por organizar as células do objeto, e por resolver o sistema numérico do problema da difusão térmica.

4. **CGrid.h:** Classe responsável por organizar as células do objeto em grids, importante para organizar as células, e facilitar a utilização pela classe CSimuladorTemperatura.

5. **CCell.h:** Classe responsável por armazenar informações da célula, como se ela está ativa ou não, se é fonte de calor ou não, qual o material e qual a temperatura.

6. **CMaterial.h:** Classe virtual responsável por prover os valores das propriedades termofísicas dos materiais, é chamada pelo CSimuladorTemperatura, e é sobreescrita por CMaterialCorrelacao.h ou CMaterialInterpolacao.h.

7. **CMaterialCorrelacao.h:** Classe responsável por calcular os valores das propriedades termofísicas com base na temperatura utilizando métodos de correlação.

8. **CMaterialInterpolacao.h:** Classe responsável por calcular os valores das propriedades termofísicas com base na temperatura, utilizando métodos de interporlação linear.

9. **CReta.h:** Classe responsável por calcular a interpolação linear.

10. **CSegmentoReta.h:** Classe responsável por armazenar segmento de reta para a classe CReta.h.

Uma outra classe externa foi utilizada no simulador:

1. **qcustomplot.h:** Classe responsável por gerar gráficos apresentados pelo mainwindow.h.

O diagrama de classes é apresentado na Figura 4.1.

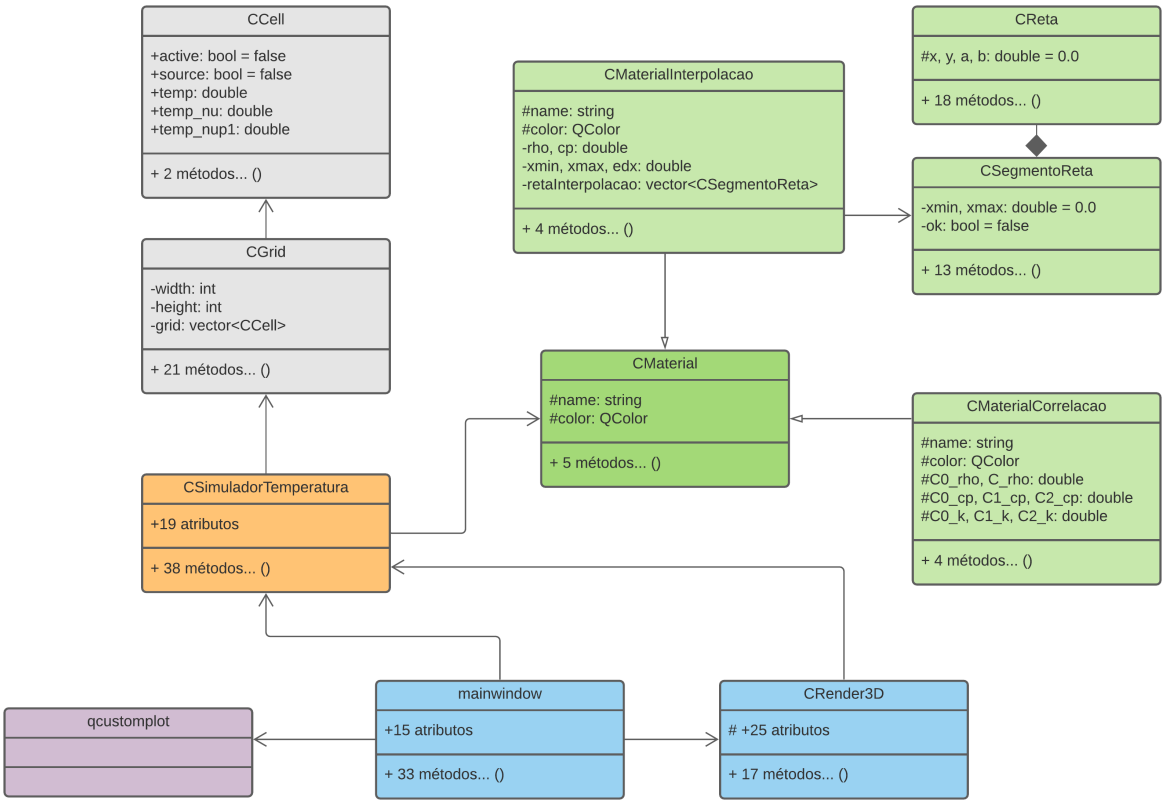


Figura 4.1: Diagrama de classes

4.2 Diagrama de seqüência – eventos e mensagens

O diagrama de seqüência enfatiza a troca de eventos e mensagens e sua ordem temporal. 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 geral

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

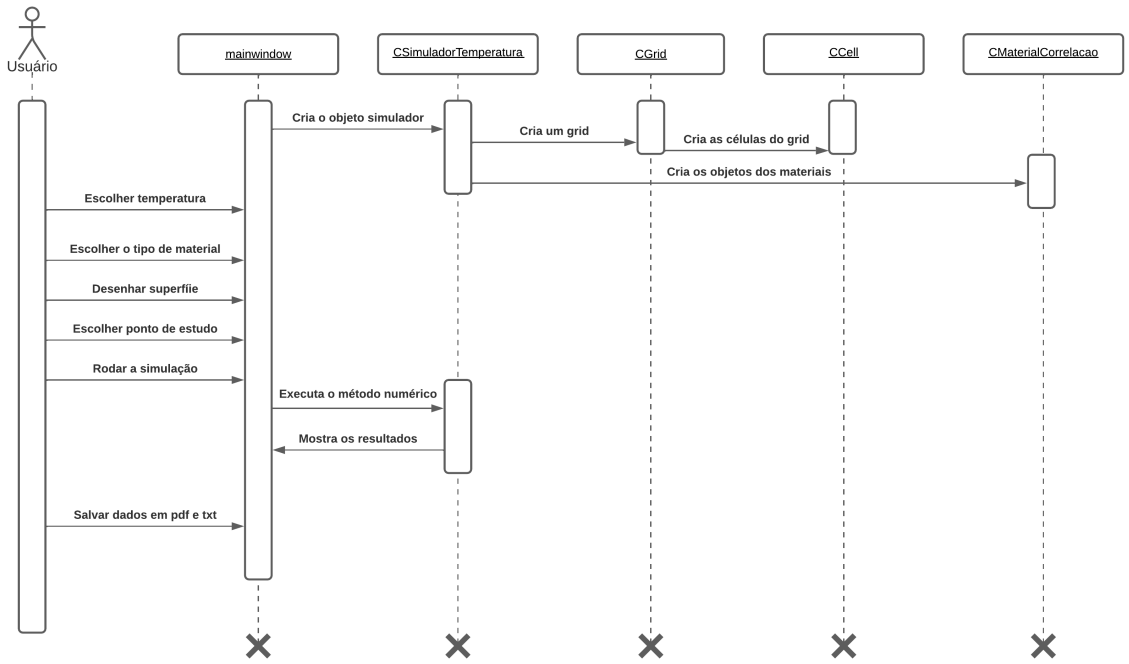


Figura 4.2: Diagrama de sequência

4.2.2 Diagrama de sequência específico

A seguir, é apresentado o diagrama de sequênia específico, conforme ilustrado na Figura 4.3.

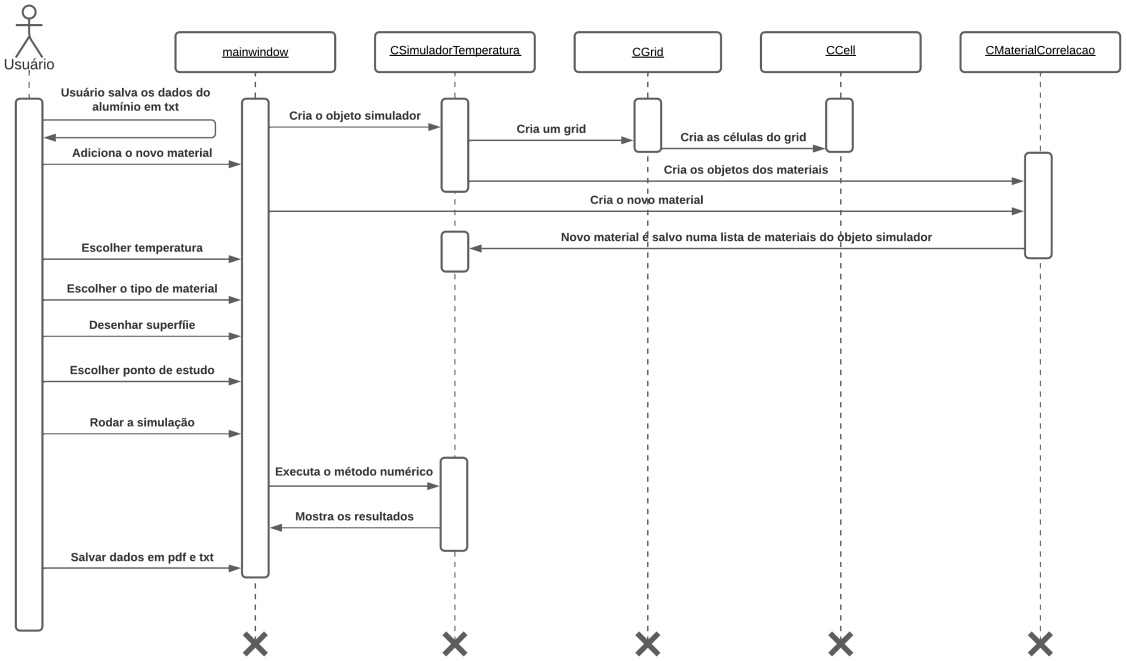


Figura 4.3: Diagrama de sequência

4.3 Diagrama de comunicação – colaboração

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

O diagrama de comuniação do caso de uso geral é apresentado na Figura 4.4.

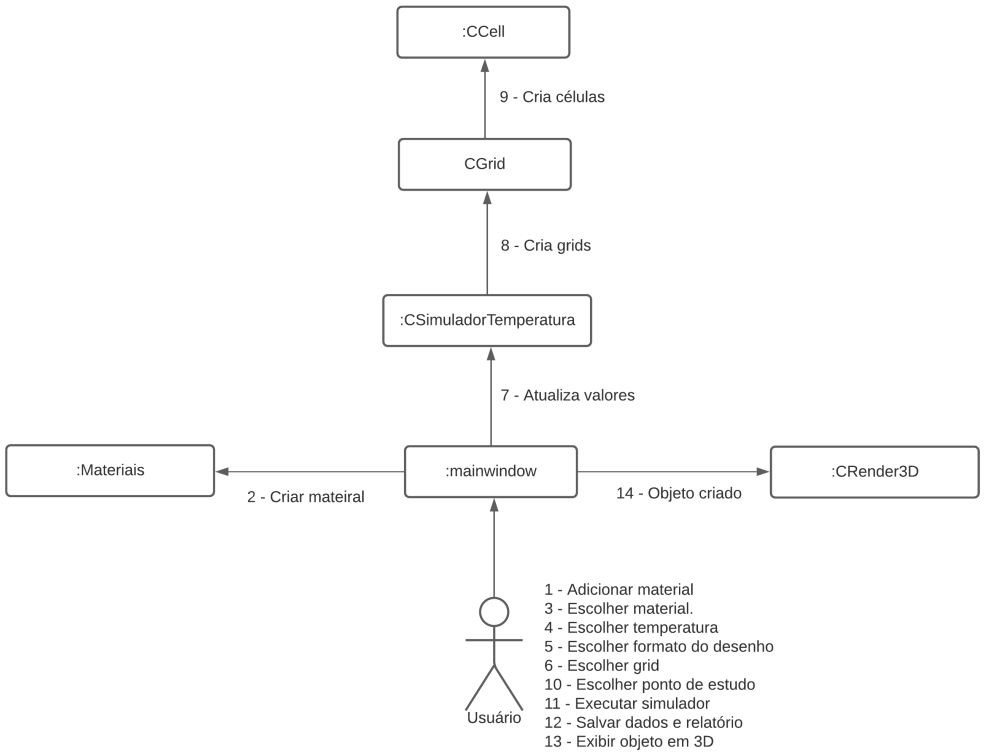


Figura 4.4: Diagrama de comunicação

4.4 Diagrama de máquina de estado

O diagrama de máquina de estado foca em analisar os estados de uma classe desde o momento de sua criação, até sua destruição. A Figura 4.5 mostra a máquina de estado para a classe CSimuladorTermico.

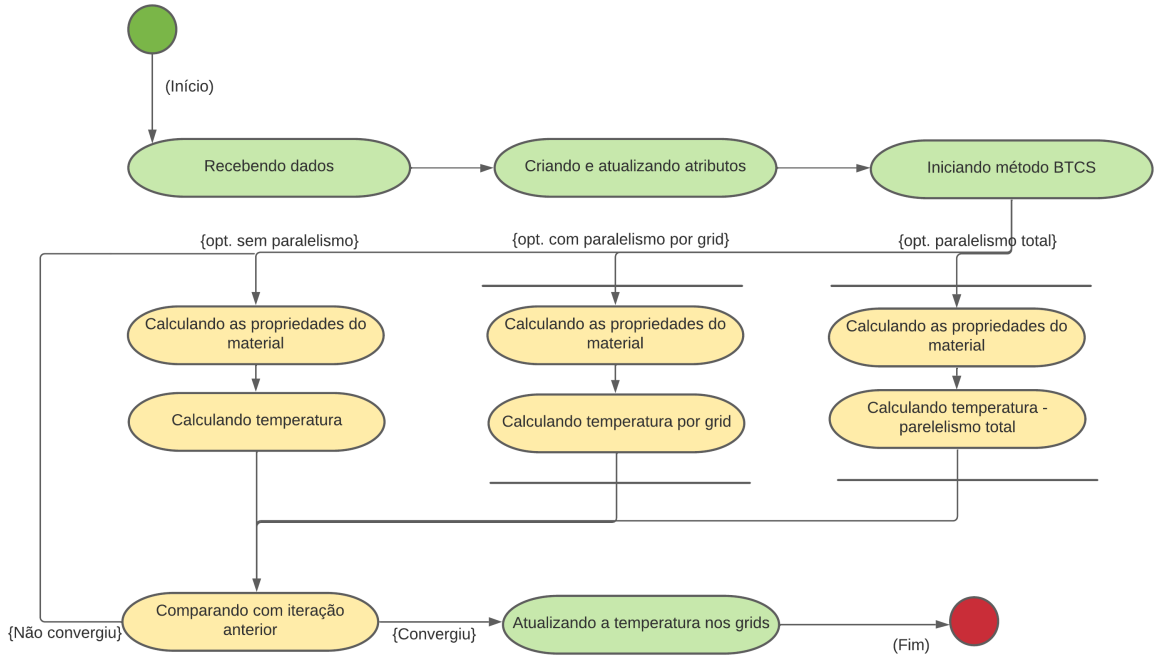


Figura 4.5: Diagrama de máquina de estado

4.5 Diagrama de atividades

No diagrama de atividades apresentado na Figura 4.6, é mostrado em detalhes uma atividade específica. Para o presente caso, será apresentado o diagrama de atividades da classe CRender3D.h, divido 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 observar. Ou seja, se existir uma superfície em contato com essa outra superfície, nenhuma das duas será criadas, pois estarão dentro do objeto.

A partir desse ponto, é obtido 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.

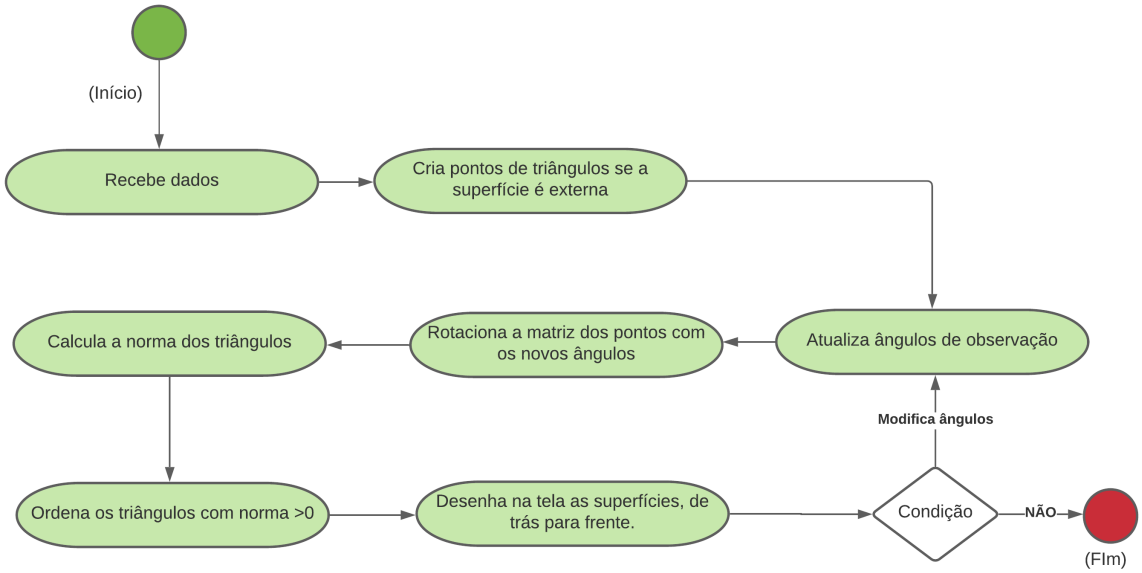


Figura 4.6: Diagrama de atividades

Capítulo 5

Projeto

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

5.1 Projeto do sistema

O software desenvolvido foi implementado com a linguagem C++, sob o paradigma de orientação ao objeto.

Inicialmente, foi utilizado a biblioteca *SFML* para a criação de janelas para o usuário, e utilizado o ambiente de desenvolvimento *Visual Studio*, tudo isso no sistema operacional *Windows 10.*

Inicialmente, foi desenvolvido um software simples, com uma mistura de janela-terminal mostrado na Figura 5.1. O usuário podia desenhar e simular, mas não tinha muita liberdade para escolher e adicionar materiais.

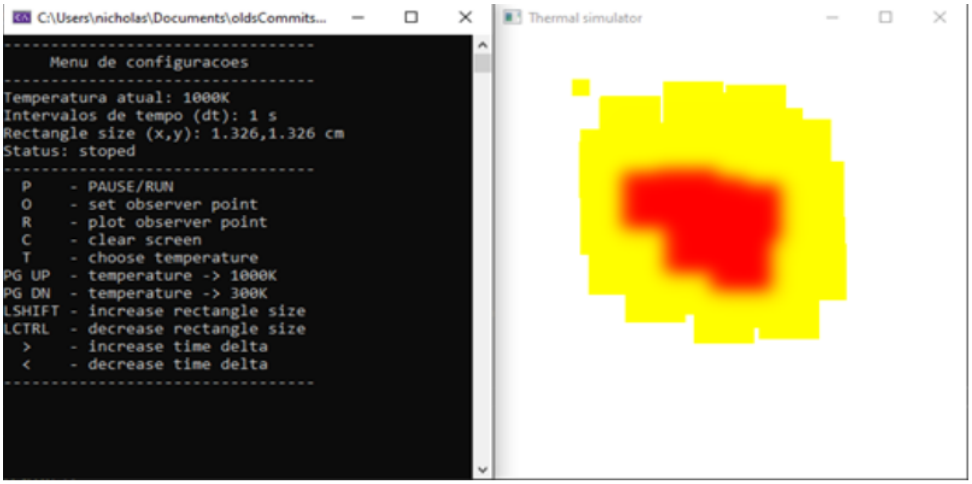


Figura 5.1: Versão 0.1, simples e utilizando a biblioteca *SFML*

Conforme a evolução pedia, foi criada uma segunda janela, a qual replica o desenho com as cores do material escolhido, como mostrado na Figura 5.2.

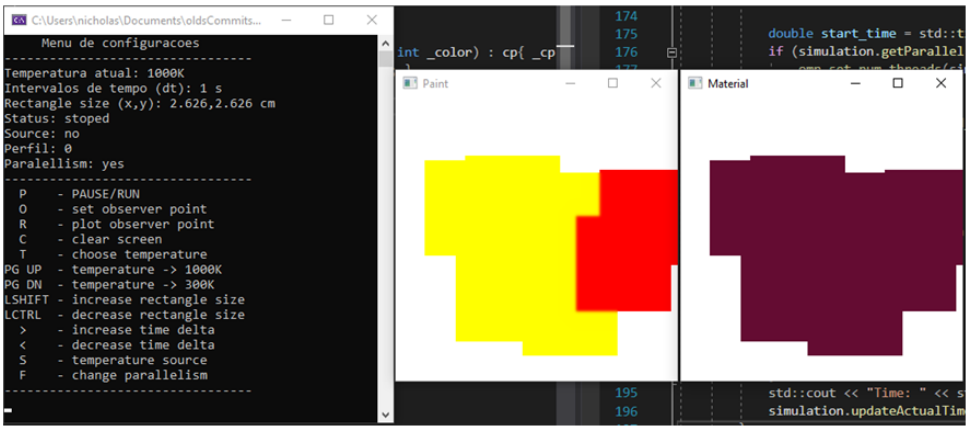


Figura 5.2: Versão 0.2, simples, mas preparando terreno para uma segunda janela dos materiais.

Por fim, foi montada a versão final utilizando essa biblioteca, mostrado na Figura 5.3. 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.

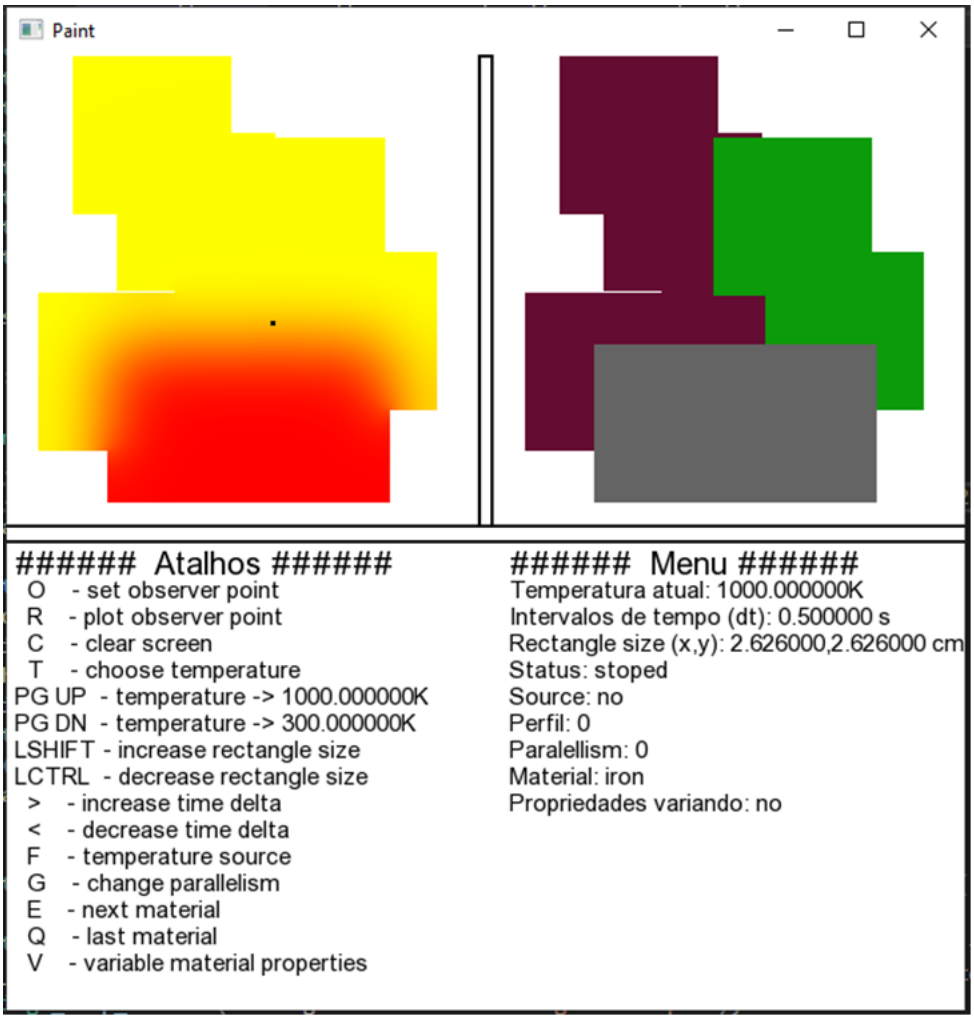


Figura 5.3: Versão 0.3, completa e complexa, mas muito lento.

Durante o desenvolvimento das versões anteriores, foi citado uma segunda biblioteca gráfica chamada *Qt*, mais rápida e completa que a anterior. Então surgiu essa necessidade de mudança.

Como o software foi programado com orientação ao objeto, foi rápido a migração, modificando, quase que somente, a classe da janela. Permitindo criar o software na versão 1.0, como na Figura 5.4.

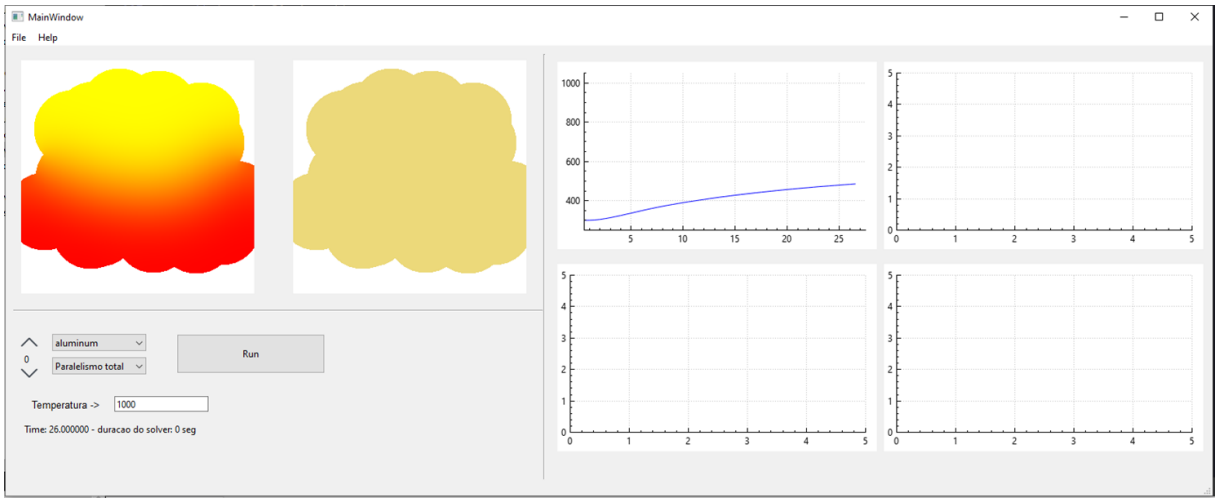


Figura 5.4: Versão 1.0, inicial e incompleta, mas utilizando a biblioteca *Qt*.

Para utilizar as ferramentas fornecidas por essa bibliteca, foi migrado do editor de texto *Visual Studio* para o *Qt Creator*, a Figura 5.5 apresenta o ambiente de trabalho.

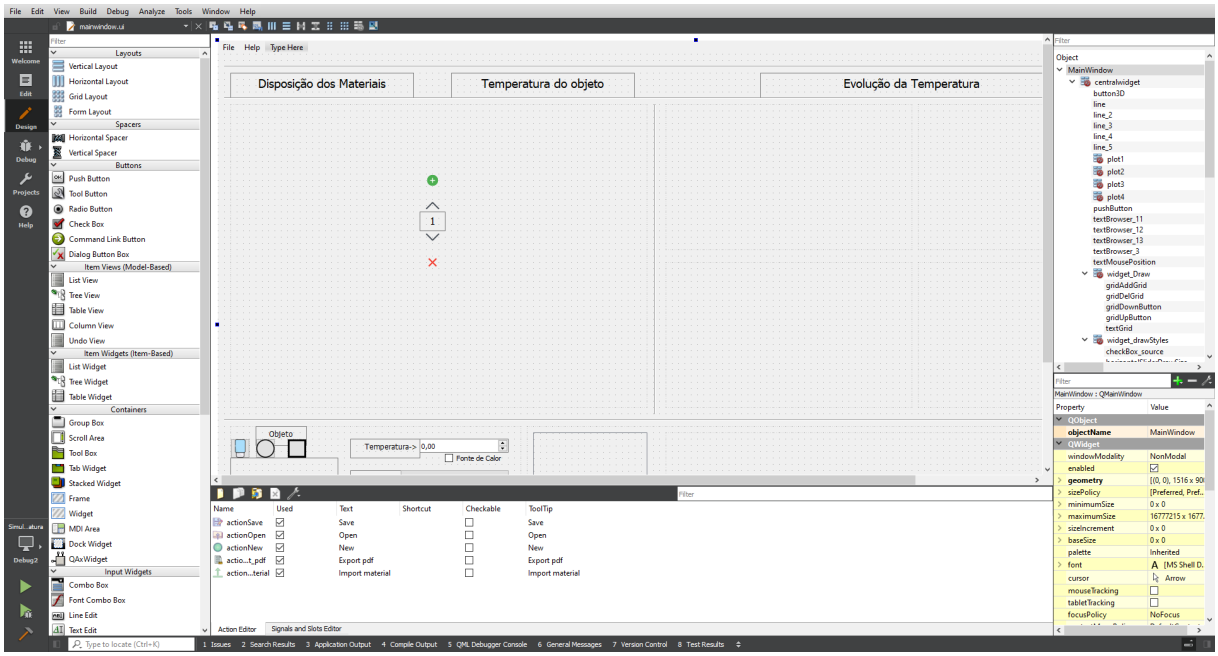


Figura 5.5: *Qt Creator*

A curva de evolução do software dentro do *Qt Creator* foi exponencial, permitindo a criação da versão final apresentada na figura 5.6, com duas áreas que apresentam os cortes desenhados, 4 gráficos com valores da temperatura ao longo do tempo ou espaço. Na região do canto inferior esquerdo, mostram opções para a simulação ou criação do objeto. Na direita, é mostrado as propriedades termofísicas de vários materiais ao longo da temperatura.

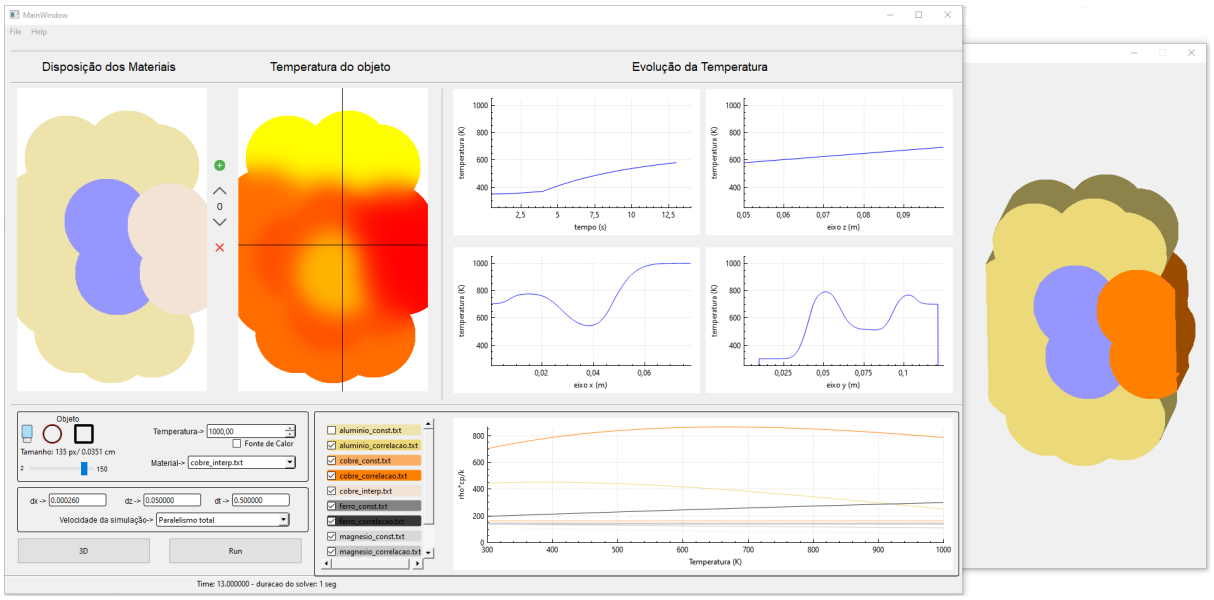


Figura 5.6: Versão 1.2, final. Na direita é apresentado a visualização 3D do objeto desenhado.

5.2 Diagrama de componentes

O diagrama de componentes mostra as relações entre todos os componentes do software, apresentado na Figura 5.7

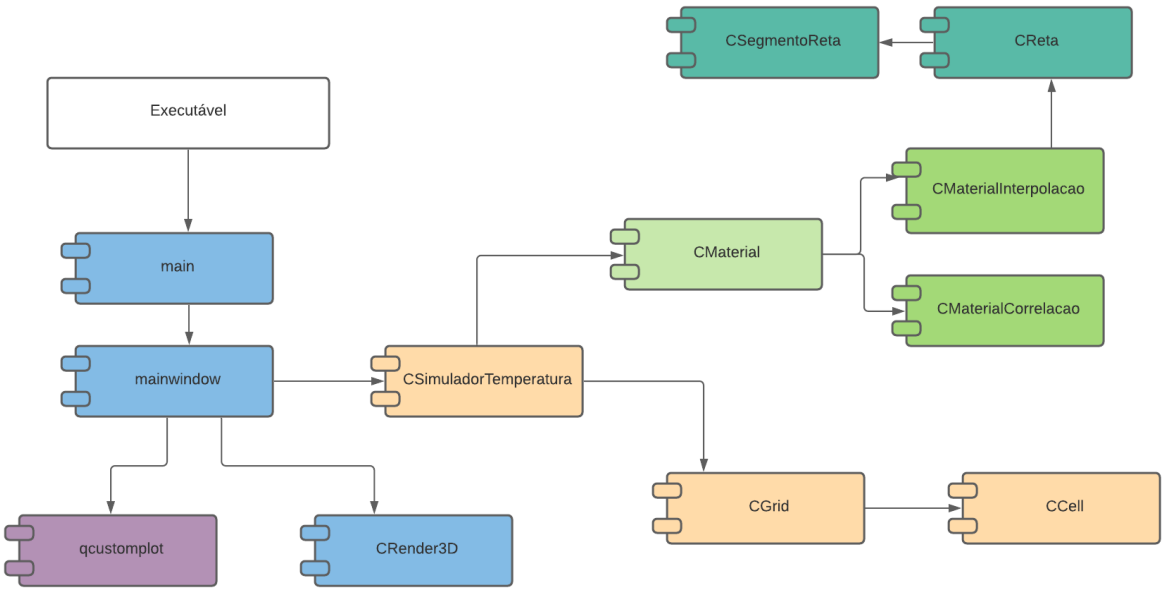


Figura 5.7: Diagrama de componentes

Começando pela esquerda na Figura 5.7, 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.

Os componentes em laranja claro são responsáveis pela simulação da difusão térmica.

5.3 Diagrama de implantação

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 5.8, é mostrado quais equipamentos o software Simulador de Difusão Térmica 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.

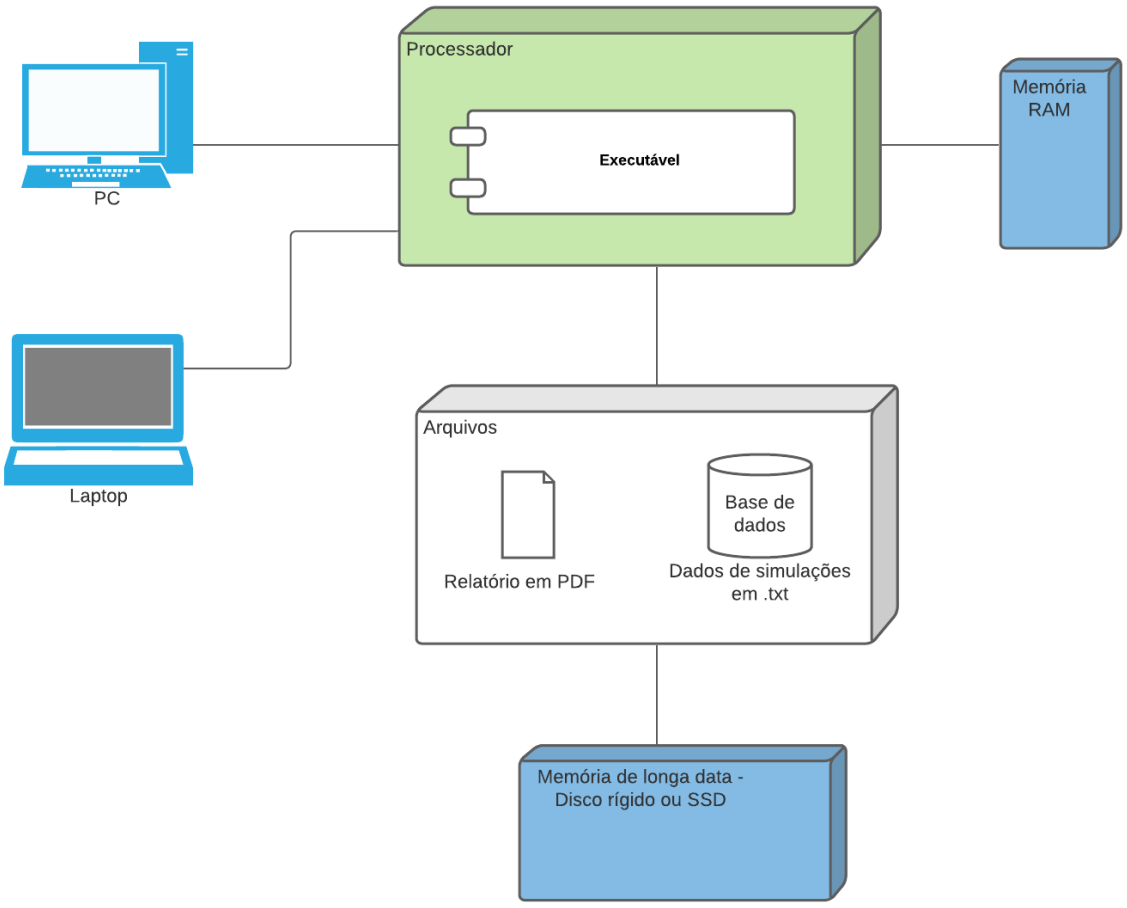


Figura 5.8: Diagrama de implementação

Capítulo 6

Implementação

Neste capítulo, são apresentados os códigos fonte implementados.

6.1 Código fonte

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

Apresenta-se na listagem 6.1 o arquivo com código da função main.

Listing 6.1: Arquivo de implementação da função main.

```
1#include "mainwindow.h"
2#include <QApplication>
3
4int 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 6.2 o arquivo de cabeçalho da classe mainwindow.

Listing 6.2: Arquivo de implementação da função mainwindow.

```
1#ifndef MAINWINDOW_H
2#define MAINWINDOW_H
3
4
5#include <QDir>                    /// Biblioteca que permite acessar
    diretórios.
6#include <QImage>                  /// desenhar pixels
7#include <QColor>                  /// escolher a cor dos pixels
```

```
8 #include <string>
9 #include <iostream>
10 #include <QPainter>                /// desenhar pixels
11 #include <QPrinter>                /// Biblioteca  que habilita a
     geração de pdf.
12 #include <QPainter>                /// Biblioteca que auxilia a
     geração do pdf.
13 #include <QPdfWriter>
14 #include <QMainWindow>
15 #include <QMouseEvent>             /// pegar acoes/posicao do mouse
16 #include <QFileDialog>
17 #include <QDirIterator>
18
19 #include "CRender3D.h"
20 #include "ui_mainwindow.h"
21 #include "CSimuladorTemperatura.h"
22
23
24 QT_BEGIN_NAMESPACE
25 namespace Ui { class MainWindow; }
26 QT_END_NAMESPACE
27
28 class MainWindow : public QMainWindow {
29     Q_OBJECT
30
31 public:
32     MainWindow(QWidget *parent = nullptr);
33     ~MainWindow();
34
35 private:
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 6.3 implementação da classe mainwindow.

Listing 6.3: Arquivo de implementação da função 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
          ::Format_ARGB32_Premultiplied);
11    timerId = startTimer(20);
12
13    ui->plot1->addGraph();
14    ui->plot2->addGraph();
15    ui->plot3->addGraph();
16    ui->plot4->addGraph();
17    ui->plot_MatProps->addGraph();
18    ui->plot1->xAxis->setLabel("tempo␣(s)");
19    ui->plot1->yAxis->setLabel("temperatura␣(K)");
20    ui->plot2->xAxis->setLabel("eixo␣z␣(m)");
21    ui->plot2->yAxis->setLabel("temperatura␣(K)");
22    ui->plot3->xAxis->setLabel("eixo␣x␣(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          ui->plot_MatProps->addGraph();
31      start_buttons();
32  }
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->
                currentText().toStdString();
43          double temperature = ui->spinBox_Temperature->value();
44          bool isSource = ui->checkBox_source->checkState();
45          int size = ui->horizontalSliderDrawSize->value();
46          simulador->setActualTemperature(temperature); ///
                importante para atualizar Tmin/Tmax
47
48          if (drawFormat =="circulo")
49              simulador->grid[currentGrid]->draw_cir(event->pos().x()
                    -left_margin-size_x-space_between_draws, event->pos
                    ().y()-up_margin, size/2, temperature, isSource,
                    simulador->getMaterial(actualMaterial),
                    eraserActivated);
50          else
51              simulador->grid[currentGrid]->draw_rec(event->pos().x()
                    -left_margin-size_x-space_between_draws, event->pos
                    ().y()-up_margin, size, temperature, isSource,
                    simulador->getMaterial(actualMaterial),
                    eraserActivated);
52      }
53      else if (event->buttons() == Qt::RightButton){
54          int x = event->pos().x()-left_margin-size_x-
                space_between_draws;
55          int y = event->pos().y()-up_margin;
56          if (x >= 0 && x < size_x && y >= 0 && y < size_y){
57              studyPoint = QPoint(x, y);
58              studyGrid = currentGrid;
59              time.clear();
```

```
60              temperature.clear();
61          }
62      }
63      update();
64  }
65
66  void MainWindow::printPosition(){
67      int x = QWidget::mapFromParent(QCursor::pos()).x() -
            left_margin-size_x-space_between_draws;
68      int y = QWidget::mapFromParent(QCursor::pos()).y() - up_margin;
69      QWidget::mapFromParent(QCursor::pos()).x();
70      std::string txt;
71      if ((x>0) && (x<size_x) && (y>0) && (y<size_y))
72          if (!simulador->grid[currentGrid]->operator()(x, y)->active
                )
73              txt = "(" + std::to_string(x) + ", " + std::to_string(y
                    ) + ")";
74          else
75              txt = "(" + std::to_string(x) + ", " + std::to_string(y
                    ) + ") - T: " +
76                    std::to_string(simulador->grid[currentGrid]->
                        operator()(x, y)->temp) + "K - "+ simulador
                        ->grid[currentGrid]->operator()(x, y)->
                        material->getName();
77      else
78          txt = "";
79
80      ui->textMousePosition->setText(QString::fromStdString(txt));
81  }
82
83  void MainWindow::printDrawSize(){
84      int size = ui->horizontalSliderDrawSize->value();
85      ui->textDrawSize->setText("Tamanho: "+QString::number(size) + "
            px/ "+QString::number(size*simulador->getDelta_x()) + " cm"
            );
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:
                                                        solid;"
98                                                  "border-color: rgb
                                                        (10,10,10)");
99
100     ui->widget_drawStyles->setStyleSheet(       "border-width: 1;"
101                                                 "border-radius: 3;"
102                                                 "border-style:
                                                        solid;"
103                                                 "border-color: rgb
                                                        (10,10,10)");
104
105     ui->widget_buttonCircle->setStyleSheet(     "border-width: 1;"
106                                                 "border-radius: 15;
                                                        "
107                                                 "border-style:
                                                        solid;"
108                                                 "border-color: rgb
                                                        (255,0,0)");
109
110     /// remover borda das caixas de texto
111     ui->textBrowser_3->setFrameStyle(QFrame::NoFrame);
112     ui->textBrowser_4->setFrameStyle(QFrame::NoFrame);
113     ui->textBrowser_5->setFrameStyle(QFrame::NoFrame);
114     ui->textBrowser_6->setFrameStyle(QFrame::NoFrame);
115     ui->textBrowser_7->setFrameStyle(QFrame::NoFrame);
116     ui->textBrowser_8->setFrameStyle(QFrame::NoFrame);
117     ui->textBrowser_9->setFrameStyle(QFrame::NoFrame);
118     ui->textBrowser_10->setFrameStyle(QFrame::NoFrame);
119     ui->textBrowser_11->setFrameStyle(QFrame::NoFrame);
120     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(
            currentGrid)));
135     QFont f = ui->textGrid->font();
136     f.setPixelSize(16);
137     ui->textGrid->setFont(f);
138     ui->textGrid->setAlignment(Qt::AlignCenter);
139 
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(
                materiais[i]));
144 
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(
            simulador->getDelta_t())));
155     ui->input_dx->setText(QString::fromStdString(std::to_string(
            simulador->getDelta_x())));
156     ui->input_dz->setText(QString::fromStdString(std::to_string(
            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(
              materiais[i]), checkboxes);
171         qss = QString("background-color: %1").arg(simulador->
              getColor(materiais[i]).name(QColor::HexArgb));
172         myCheckbox[i]->setStyleSheet(qss);
173         layout->addWidget(myCheckbox[i]);
174     }
175     ui->scrollArea->setWidget(checkboxes);
176     makePlotMatProps();
177 }
178
179 void MainWindow::paintEvent(QPaintEvent *e) {
180     QPainter painter(this);
181     *mImage = paint(currentGrid);
182     painter.drawImage(left_margin,up_margin, *mImage);
183     e->accept();
184 }
185
186 QImage MainWindow::paint(int grid) {
187     QImage img = QImage(size_x*2+space_between_draws, size_y,QImage
          ::Format_ARGB32_Premultiplied);
188
189     /// desenho da temperatura
190     for (int i = 0; i < size_x; i++){
191         for (int k = 0; k < size_y; k++){
192             if (!simulador->grid[grid]->operator()(i, k)->active)
193                 img.setPixelColor(i+size_x+space_between_draws,k,
                      QColor::fromRgb(255,255,255));
194             else
195                 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) && (
          studyPoint.y() > 0 || studyPoint.y() < size_y) && grid ==
```

```
        studyGrid){
200         for(int i = 0; i < size_x; i++)
201             img.setPixelColor(i+size_x+space_between_draws,
                    studyPoint.y(), QColor::fromRgb(0,0,0));
202         for(int i = 0; i < size_y; i++)
203             img.setPixelColor(studyPoint.x()+size_x+
                    space_between_draws, i, QColor::fromRgb(0,0,0));
204     }
205
206     /// desenho dos materiais
207     for (int i = 0; i < size_x; i++){
208         for (int k = 0; k < size_y; k++){
209             if (!simulador->grid[grid]->operator()(i, k)->active)
210                 img.setPixelColor(i,k, QColor::fromRgb(255,255,255)
                        );
211             else
212                 img.setPixelColor(i,k, simulador->grid[grid]->
                        operator()(i, k)->material->getColor());
213         }
214     }
215     return img;
216 }
217
218 QColor MainWindow::calcRGB(double temperatura){
219     double maxTemp = simulador->getTmax();
220     double minTemp = simulador->getTmin();
221     return QColor::fromRgb(255, (maxTemp - temperatura)*255/(
            maxTemp - minTemp), 0, 255);
222 }
223
224 void MainWindow::runSimulator(){
225     simulador->setDelta_t(std::stod(ui->input_dt->text().
            toStdString()));
226     simulador->setDelta_x(std::stod(ui->input_dx->text().
            toStdString()));
227     simulador->setDelta_z(std::stod(ui->input_dz->text().
            toStdString()));
228
229     time_t start_time = std::time(0);
230     std::string type = ui->parallel_comboBox->currentText().
            toStdString();
231     if(type == "Sem␣paralelismo")
```

```
232         simulador->run_sem_paralelismo();
233     if(type=="Paralelismo␣por␣grid")
234         simulador->run_paralelismo_por_grid();
235     if(type=="Paralelismo␣total")
236         simulador->run_paralelismo_total();
237     time.append((time.size()+1)*simulador->getDelta_t());
238 
239     std::string result = "Time:␣" + std::to_string(time[time.size()
        -1]) + "␣-␣duracao␣do␣solver:␣" + std::to_string(std::time
        (0) - start_time) + "␣seg";
240     ui->textBrowser_3->setText(QString::fromStdString(result));
241 
242     update();
243     makePlot1();
244     makePlot2();
245     makePlot3();
246     makePlot4();
247 }
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(
        currentGrid)));
270     ui->textGrid->setAlignment(Qt::AlignCenter);
```

```
271     update();
272 }
273
274 void MainWindow::on_gridUpButton_clicked()
275 {
276     currentGrid++;
277     if (currentGrid > simulador->getNGRIDS()-1)
278         currentGrid = simulador->getNGRIDS()-1;
279     /// texto do grid
280     ui->textGrid->setText(QString::fromStdString(std::to_string(
            currentGrid)));
281     ui->textGrid->setAlignment(Qt::AlignCenter);
282     update();
283 }
284
285 void MainWindow::makePlot1(){
286     temperature.append(simulador->grid[studyGrid]->operator()(
            studyPoint.x(), studyPoint.y())->temp);
287
288     ui->plot1->graph(0)->setData(time,temperature);
289     ui->plot1->xAxis->setRange(time[0], time[time.size()-1]+1);
290     ui->plot1->yAxis->setRange(simulador->getTmin()-50, simulador->
            getTmax()+50);
291     ui->plot1->replot();
292     ui->plot1->update();
293 }
294
295 void MainWindow::makePlot2(){
296     QVector<double> temperature_z(simulador->getNGRIDS());
297     QVector<double> labor_z(simulador->getNGRIDS());
298     for (int i = 0; i < simulador->getNGRIDS(); i++){
299         labor_z[i] = simulador->getDelta_z()*(i+1);
300         temperature_z[i] = simulador->grid[i]->operator()(
                studyPoint.x(), studyPoint.y())->temp;
301     }
302
303     ui->plot2->graph(0)->setData(labor_z,temperature_z);
304     ui->plot2->xAxis->setRange(labor_z[0], labor_z[labor_z.size()
            -1]);
305     ui->plot2->yAxis->setRange(simulador->getTmin()-50, simulador->
            getTmax()+50);
306     ui->plot2->replot();
```

```
307     ui->plot2->update();
308 }
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()+"\\
            save_results\\horizontal_"+std::to_string(time[time.size()
            -1]+1)+".dat");
314     for (int i = 0; i < size_x; i++){
315         labor_x[i] = simulador->getDelta_x()*(i+1);
316         temperature_x[i] = simulador->grid[studyGrid]->operator()(i
                , studyPoint.y())->temp;
317         file << labor_x[i] << ";␣" << temperature_x[i] << std::endl
                ;
318     }
319     file.close();
320     ui->plot3->graph(0)->setData(labor_x,temperature_x);
321     ui->plot3->xAxis->setRange(labor_x[0], labor_x[size_x-1]);
322     ui->plot3->yAxis->setRange(simulador->getTmin()-50, simulador->
            getTmax()+50);
323     ui->plot3->replot();
324     ui->plot3->update();
325 }
326
327 void MainWindow::makePlot4(){
328     QVector<double> temperature_y(size_y);
329     QVector<double> labor_y(size_y);
330     std::ofstream file(dir.absolutePath().toStdString()+"\\
            save_results\\vertical"+std::to_string(time[time.size()
            -1]+1)+".dat");
331     for (int i = 0; i < size_y; i++){
332         labor_y[i] = simulador->getDelta_x()*(i+1);
333         temperature_y[i] = simulador->grid[studyGrid]->operator()(
                studyPoint.x(), i)->temp;
334         file << labor_y[i] << ";␣" << temperature_y[i] << std::endl
                ;
335     }
336     file.close();
337     ui->plot4->graph(0)->setData(labor_y,temperature_y);
338     ui->plot4->xAxis->setRange(labor_y[0], labor_y[size_y-1]);
339     ui->plot4->yAxis->setRange(simulador->getTmin()-50, simulador->
```

```
        getTmax()+50);
340     ui->plot4->replot();
341     ui->plot4->update();
342 }
343
344 void MainWindow::makePlotMatProps(){
345     bool changeState = checkChangeMaterialsState();
346     if (!changeState)
347         return;
348     int nPoints = 100;
349     QVector<double> props(nPoints);
350     QVector<double> temperature_x(nPoints);
351     std::vector<std::string> materiais = simulador->getMateriais();
352     double max_props = 700;
353
354     double dT = (simulador->getTmax() - simulador->getTmin())/(
        nPoints-1);
355     for (unsigned int mat = 0; mat < materiais.size(); mat++){
356         if (selectedMateriails[mat]){
357             for (int i = 0; i < nPoints; i++){
358                 temperature_x[i] = dT*i + simulador->getTmin();
359                 props[i] = simulador->getProps(temperature_x[i],
                    materiais[mat]);
360             }
361         ui->plot_MatProps->graph(mat)->setPen(QPen(simulador->
            getColor(materiais[mat])));
362         ui->plot_MatProps->graph(mat)->setData(temperature_x,props)
            ;
363         for (int i = 0; i < nPoints; i++)
364             max_props = max_props < props[i]? props[i] : max_props;
                /// aqui ajusto o ylabel
365         }else{
366             ui->plot_MatProps->graph(mat)->data()->clear();
367         }
368     }
369     ui->plot_MatProps->xAxis->setRange(temperature_x[0],
        temperature_x[nPoints-1]);
370     ui->plot_MatProps->yAxis->setRange(0, max_props);
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 a
        file", path+"//save", tr("Dados (*.dat)"));
393     std::string txt = simulador->saveGrid(file_name.toStdString());
394     ui->textBrowser_3->setText(QString::fromStdString(txt));
395 }
396
397
398 void MainWindow::on_actionOpen_triggered()
399 {
400     QDir dir; QString path = dir.absolutePath();
401     QString file_name = QFileDialog::getOpenFileName(this, "Open a
        file", path+"//save", tr("Dados (*.dat)"));
402     std::string txt = simulador->openGrid(file_name.toStdString());
403     ui->textBrowser_3->setText(QString::fromStdString(txt));
404 }
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:␣1;"
433                                                 "border-radius:␣15;
                                                        "
434                                                 "border-style:␣
                                                        solid;"
435                                                 "border-color:␣rgb
                                                        (255,0,0)");
436     ui->widget_buttonSquare->setStyleSheet(     "border-width:␣0;"
437                                                 "border-radius:␣0;"
438                                                 "border-style:␣
                                                        solid;"
439                                                 "border-color:␣rgb
                                                        (255,0,0)");
440     drawFormat = "circulo";
441 }
442
443
444 void MainWindow::on_buttonSquare_clicked()
```

```
445 {
446     ui->widget_buttonSquare->setStyleSheet(      "border-width: 1;"
447                                                  "border-radius: 0;"
448                                                  "border-style:
                                                        solid;"
449                                                  "border-color: rgb
                                                        (255,0,0)");
450     ui->widget_buttonCircle->setStyleSheet(      "border-width: 0;"
451                                                  "border-radius: 15;
                                                        "
452                                                  "border-style:
                                                        solid;"
453                                                  "border-color: rgb
                                                        (255,0,0)");
454     drawFormat = "quadrado";
455
456 }
457
458
459 void MainWindow::on_buttonEraser_clicked()
460 {
461     if (eraserActivated){
462         ui->widget_eraser->setStyleSheet("border-width: 0;"
463                                          "border-radius: 0;"
464                                          "border-style: solid;"
465                                          "border-color: rgb
                                                (255,0,0)");
466     eraserActivated = false;
467     }
468     else{
469         ui->widget_eraser->setStyleSheet("border-width: 1;"
470                                          "border-radius: 5;"
471                                          "border-style: solid;"
472                                          "border-color: rgb
                                                (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
        arquivo PDF ele não executa o resto.
489         return "Erro ao abrir PDF";
490 
491 
492     painterPDF.setFont(QFont("Arial", 8));
493     painterPDF.drawText(40,140, "==> PROPRIEDADES DO GRID <==");
494     painterPDF.drawText(40,160, "Delta x: " + QString::number(
        simulador->getDelta_x())+" m");
495     painterPDF.drawText(40,180, "Delta z: " + QString::number(
        simulador->getDelta_z())+" m");
496     painterPDF.drawText(40,200, "Delta t: " + QString::number(
        simulador->getDelta_t())+" s");
497 
498     painterPDF.drawText(40,240, "Largura total horizontal: " +
        QString::number(simulador->getDelta_x()*size_x)+" m");
499     painterPDF.drawText(40,260, "Largura total vertical: " +
        QString::number(simulador->getDelta_x()*size_y)+" m");
500     painterPDF.drawText(40,280, "Largura total entre perfis (eixo z
        ): " + QString::number(simulador->getDelta_z()*simulador->
        getNGRIDS())+" m");
501 
502 
503 
504     painterPDF.drawText(400,140, "==> PROPRIEDADES DA SIMULAÇÃO <==
        ");
505     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␣"+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␣"+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(
            currentGrid)));
575     ui->textGrid->setAlignment(Qt::AlignCenter);
576     update();
577
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(
            currentGrid)));
589     ui->textGrid->setAlignment(Qt::AlignCenter);
590     update();
591 }
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 6.4 o arquivo de cabeçalho da classe CRender3D.

Listing 6.4: 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 =
          nullptr);
25     ~CRender3D();
26 protected:
27     void paintEvent(QPaintEvent *event) override;
28
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,
            QVector3D b);
45
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 6.5 implementação da classe CRender3D.

Listing 6.5: 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
21CRender3D::CRender3D(CSimuladorTemperatura *_simulador, QWidget *
      parent)
22    : QMainWindow(parent)
23{
24    simulador = _simulador;
25    this->setFixedSize(800,800);
26    this->adjustSize();
27    size_x = 500;
28    mImage = new QImage(size_x, size_y,QImage::
          Format_ARGB32_Premultiplied);
29    timerId = startTimer(0);
30    margin_x = 400;//simulador->getWidth();
31    margin_y = 400;//simulador->getHeight();
32    std::cout<<"criando␣cubos"<<std::endl;
33    dx = 1;//simulador->getDelta_x();
34    dy = dx;
35    dz = 1*simulador->getDelta_z()/simulador->getDelta_x();
36    double maxTemp = simulador->getTmax();
37    double minTemp = simulador->getTmin();
38    for(int g = 0; g<simulador->getNGRIDS(); g++){
39        for(int i = 0; i < simulador->grid[g]->getWidth(); i++){
40            for(int j = 0; j < simulador->grid[g]->getHeight(); j
                  ++){
41                if (simulador->grid[g]->operator()(i,j)->active){
42                    cube.push_back(createCube(QVector3D(i,j,(g+1)*
                          dz)));
43                    activeEdges.push_back(edges(i,j,g));
44                    colorsMaterial.push_back(simulador->grid[g]->
```

```
                        operator()(i,j)->material->getColor());
45                    colorsTemperature.push_back(getRGB(simulador->
                        grid[g]->operator()(i,j)->temp, minTemp,
                        maxTemp));
46                }
47            }
48        }
49    }
50
51    std::cout<<"cubos criados"<<std::endl;
52    createTriangles();
53    drawCube.resize(8);
54    update();
55}
56
57
58CRender3D::~CRender3D()
59{
60    //delete ui;
61}
62
63QVector<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
109 void 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 }
202
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],
                        drawCube[c]).z();
247                 if(prodVet > 0){
248                     pos_norm.push_back(std::pair(count, prodVet));
249                     count++;
250                     if(r == 0 || r == 1 || r == 8 || r == 9) ///
```

```
                        fronteiras de g
251                         coresDesenho.push_back(QColor(color[cb].red
                                (), color[cb].green(), color[cb].blue(),
                                 255));
252                     else
253                         coresDesenho.push_back(QColor(QColor(color[
                                cb].red()*0.6, color[cb].green()*0.6,
                                color[cb].blue()*0.6, 255)));
254
255                     QPolygon pol;
256                     pol << QPoint(drawCube[a].x(),drawCube[a].y())
257                         << QPoint(drawCube[b].x(),drawCube[b].y())
258                         << QPoint(drawCube[c].x(),drawCube[c].y());
259                     triangulosDesenho.push_back(pol);
260                 }
261             }
262         }
263     }
264
265     /// organizo conforme a profundidade
266     std::sort(pos_norm.begin(), pos_norm.end(), [](auto &left, auto
            &right) {
267         return left.second > right.second;
268     });
269
270     /// desenho na tela
271     int pos;
272     painter.setPen(QColor(0,0,0,0));
273     for(int i = 0; i<triangulosDesenho.size(); i++){
274         pos = pos_norm[i].first;
275         painter.setBrush(coresDesenho[pos]);
276         painter.drawPolygon(triangulosDesenho[pos]);
277     }
278
279     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,
      QVector3D b){
320     QVector3D ax = a - origem;
321     QVector3D bx = b - origem;
322     return QVector3D(ax.y()*bx.z()-ax.z()*bx.y(), -ax.x()*bx.z()+ax
```

```
        .z()*bx.x(), ax.x()*bx.y()-ax.y()*bx.x());
323 }
```

Apresenta-se na listagem 6.6 o arquivo de cabeçalho da classe CSimuladorTemperatura.

Listing 6.6: 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 getTmax() { return Tmax; }
82      double getTmin() { 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      std::vector<std::string> getMateriais() { return name_materiais
            ; }
92};
93#endif
```

Apresenta-se na listagem 6.7 implementação da classe CSimuladorTemperatura.

Listing 6.7: Arquivo de implementação da função `main()`.

```
1#include "CSimuladorTemperatura.h"
2
3void 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
11void CSimuladorTemperatura::resetGrid() {
12     for (int i = 0; i < NGRIDS; i++)
13         grid[i]->resetGrid(0.0);
14}
15
16void CSimuladorTemperatura::createListOfMaterials() {
17     ///*
18     std::string matName;
19     QDirIterator it(dir.absolutePath()+"//materiais", {"*.
           correlacao", "*.constante", "*.interpolacao"}, QDir::Files,
           QDirIterator::Subdirectories);
```

```
20      while (it.hasNext()) {
21          it.next();
22          matName = it.fileName().toStdString();
23          QFileInfo fi(it.fileName());
24          std::string type = fi.suffix().toStdString();
25          materiais[matName] = chooseMaterialType(matName, type);
26      }
27      for(auto const& imap: materiais)
28          name_materiais.push_back(imap.first);
29  }
30
31  CMaterial* CSimuladorTemperatura::chooseMaterialType(std::string
        name, std::string type){
32      std::ifstream file(dir.absolutePath().toStdString()+"/materiais
            //"+name);
33
34      if (type == "correlacao" || type == "constante")
35          return new CMaterialCorrelacao(name);
36      else if (type == "interpolacao")
37          return new CMaterialInterpolacao(name);
38  }
39
40  void CSimuladorTemperatura::addGrid(){
41      NGRIDS++;
42      grid.push_back(new CGrid(width, height, 0.0));
43  }
44
45  void CSimuladorTemperatura::delGrid(int _grid){
46      NGRIDS--;
47      grid.erase(grid.begin()+_grid);
48  }
49
50  std::string CSimuladorTemperatura::openMaterial(std::string
        nameFile){
51      std::ifstream file(nameFile);
52
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_paralelismo_por_grid() {
77      omp_set_num_threads(NGRIDS);
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_paralelismo_total() {
86      for (int g=0;g<NGRIDS;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
                calcular o erro da iteracao
103         for (int i = 0; i < grid[g]->getWidth(); i++)
104             for (int k = 0; k < grid[g]->getHeight(); k++)
105                 calculatePointIteration(i, k, g);
106         _erro = grid[g]->maxErroIteration();
107         erro = erro < _erro ? _erro : erro;
108         iter++;
109     }
110     grid[g]->updateSolver();
111 }
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+=
                    MAX_THREADS) {
120                 x = i % grid[g]->getWidth();
121                 y = i / grid[g]->getWidth();
122
123                 (*grid[g])(x, y)->temp_nu = (*grid[g])(x, y)->
                        temp_nup1;
124                 _erro = calculatePointIteration(x, y, g);
125                 erro = erro < _erro ? _erro : erro;
126             }
127         }
128         iter++;
129         if (erro < MIN_ERRO && iter > MIN_ITERATION)
130             break;
131     } while (iter < MAX_ITERATION);
132     std::cout<<"iteracoes: " << iter << " - erro: " << erro << std
        ::endl;
133 }
134
135 double CSimuladorTemperatura::calculatePointIteration(int x, int y,
      int g) {
136     if (!(*grid[g])(x,y)->active)
137         return 0.0;
```

```
138     if ((*grid[g])(x, y)->source)
139         return 0.0;
140     float n_x = 0;
141     float n_z = 0;
142     double inf = .0, sup = .0, esq = .0, dir = .0, cima = .0, baixo
            =.0;
143     double thermalConstant;
144 
145     if (y - 1 > 0) {
146         if ((*grid[g])(x, y - 1)->active) {
147             n_x++;
148             inf = (*grid[g])(x, y - 1)->temp_nup1*delta_z*delta_z;
149         }
150     }
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*
                    delta_z;
163         }
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*
                    delta_x;
184         }
185     }
186
187     thermalConstant = (*grid[g])(x, y)->material->getThermalConst
            ((*grid[g])(x, y)->temp_nup1);
188
189     (*grid[g])(x, y)->temp_nup1 = (thermalConstant * (*grid[g])(x,
            y)->temp*delta_x*delta_x*delta_z*delta_z/delta_t + inf + sup
             + esq + dir + cima + baixo) / (n_x*delta_z*delta_z + n_z*
            delta_x*delta_x + thermalConstant*delta_x*delta_x*delta_z*
            delta_z/delta_t);
190     return (*grid[g])(x, y)->temp_nup1 - (*grid[g])(x, y)->temp_nu;
191 }
192
193 std::string CSimuladorTemperatura::saveGrid(std::string nameFile) {
194     std::ofstream file(nameFile);
195     int sizeGrid = grid[0]->getSize();
196     file << NGRIDS <<"\n";
197     for (int g = 0; g < NGRIDS; g++) {
198         for (int i = 0; i < sizeGrid; i++) {
199             if((*grid[g])[i]->active){
200                 file << i << "␣" << g << "␣";
201                 file << (*grid[g])[i]->temp << "␣";
202                 file << (*grid[g])[i]->active << "␣";
203                 file << (*grid[g])[i]->source << "␣";
204                 file << (*grid[g])[i]->material->getName() << "\n";
205             }
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 >>
        _name){
228             grid[g]->draw(i, _temperature, _active, _source, _name)
                ;
229             Tmax = Tmax < _temperature ? _temperature : Tmax;
230     }
231
232     file.close();
233     return "Arquivo carregado!";
234 }
235
236 void CSimuladorTemperatura::setActualTemperature(double
     newTemperature) {
237     if (newTemperature > Tmax)
238         Tmax = newTemperature;
239     if (newTemperature < Tmin)
240         Tmin = newTemperature;
241     actualTemperature = newTemperature;
242 }
243
244 void CSimuladorTemperatura::changeMaterialPropertiesStatus() {
245     materialPropertiesStatus = materialPropertiesStatus ? false :
        true;
246 }
247
248 double CSimuladorTemperatura::getProps(double temperature, std::
     string material){
```

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

Apresenta-se na listagem 6.8 o arquivo de cabeçalho da classe CGrid.

Listing 6.8: 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         grid.resize(width * height);
22     }
23 
24     CGrid(int _width, int _height, double temperature) {
25         resetSize(_width, _height, temperature);
26     }
27 
28     void resetGrid(double temperature);
29 
30     void resetSize(int _width, int _height, double temperature);
31 
32     void draw_rec(int x, int y, double size, double temperature,
       bool isSourceActive, CMaterial* _material, bool eraser);
33     void draw_cir(int x, int y, double size, double temperature,
       bool isSourceActive, CMaterial* _material, bool eraser);
34     void draw(int x, double temperature, bool active, bool isSource
       , std::string _material);
35 
36     int getSize() { return width * height; }
37 
38     void updateIteration();
39     void updateSolver();
40     void startIteration();
41     double maxErroIteration();
42 
43     int getWidth() { return width;}
44     int getHeight() { return height; }
45     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     CCell* operator [] (int x) { return &grid[x]; }
53
54 };
55 #endif
```

Apresenta-se na listagem 6.9 implementação da classe CGrid.

Listing 6.9: Arquivo de implementação da função `main()`.

```
1 #include "CGrid.h"
2
3 void CGrid::resetSize(int _width, int _height, double temperature)
      {
4     width = _width;
5     height = _height;
6     grid.resize(width * height);
7     for (int i = 0; i < width * height; i++)
8         grid[i].temp = temperature;
9 }
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
      , bool isSourceActive, CMaterial* _material, bool eraser) {
23     int start_x = (x - size / 2 >= 0) ? x - size / 2 : 0;
24     int start_y = (y - size / 2 >= 0) ? y - size / 2 : 0;
25     int max_x   = (x + size / 2 >= width)  ? width  : x - size/2 +
           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
        isSource, std::string _material) {
60      grid[x].temp = _temperature;
61      grid[x].active = active;
62      grid[x].source = isSource;
```

```
63      if (active)
64          grid[x].material = new CMaterialCorrelacao(_material);
65      else
66          grid[x].material = new CMaterial();
67  }
68
69  void CGrid::updateIteration() {
70      for (int i = 0; i < width * height; i++)
71          grid[i].temp_nu = grid[i].temp_nup1;
72  }
73
74  void CGrid::updateSolver() {
75      for (int i = 0; i < width * height; i++)
76          grid[i].temp = grid[i].temp_nup1;
77  }
78
79  double CGrid::maxErroIteration() {
80      double erro = 0.0;
81      double erro_posicao = 0.0;
82      for (int i = 0; i < width * height; i++) {
83          erro_posicao = grid[i].temp_nup1 - grid[i].temp_nu;
84          erro = abs(erro_posicao) > erro ? erro_posicao : erro;
85      }
86      return erro;
87  }
88
89  void CGrid::startIteration() {
90      for (int i = 0; i < width * height; i++)
91          grid[i].temp_nup1 = grid[i].temp;
92  }
93
94  double CGrid::maxTemp() {
95      double maxTemp = 0;
96      for (int i = 0; i < width * height; i++)
97          maxTemp = maxTemp < grid[i].temp ? grid[i].temp : maxTemp;
98      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 6.10 o arquivo de cabeçalho da classe CCell.

Listing 6.10: 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
          & cell) { return os << cell.temp; }
17 };
18 #endif
```

Apresenta-se na listagem 6.11 implementação da classe CCell.

Listing 6.11: Arquivo de implementação da função `main()`.

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

Apresenta-se na listagem 6.12 o arquivo de cabeçalho da classe CMaterial.

Listing 6.12: 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 6.13 implementação da classe CMaterial.

Listing 6.13: Arquivo de implementação da função `main()`.

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

Apresenta-se na listagem 6.14 o arquivo de cabeçalho da classe CMaterialCorrelacao.

Listing 6.14: 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 6.15 implementação da classe CMaterialCorrelacao.

Listing 6.15: 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 >>
               alpha;
12         color = QColor(r, g, b, alpha);
13         file >> str_temp; file >> cp;
14         file >> str_temp; file >> rho;
15         file >> str_temp; file >> str_temp; /// texto explicando a
               conta
16         file >> str_temp; file >> C0_k;   file >> C1_k;   file >>
               C2_k;
17     }
18     else {
19         std::cout<<"can't open file!" << std::endl;
20     }
21 }
22
23 double CMaterialCorrelacao::getThermalConst(double T) {
24     double k   = C0_k   + C1_k   * T + C2_k  * T * T;
25     return rho * cp/k;
26 }
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 6.16 o arquivo de cabeçalho da classe CMaterialInterpolacao.

Listing 6.16: Arquivo de implementação da classe CMaterialInterpolacao.

```
#ifndef CMATERIALINTERPOLACAO_H
#define CMATERIALINTERPOLACAO_H

#include <QDir>
#include <string>
#include <vector>
#include "CMaterial.h"
#include "CSegmentoReta.h"

class CMaterialInterpolacao :public CMaterial {
public:
    CMaterialInterpolacao();
    CMaterialInterpolacao(std::string _name);

    double getThermalConst(double T);

    QColor getColor()       { return color; }
    std::string getName()   { return name; }

    double getK(double T);
protected:
    std::string name;
    QColor color;

private:
    std::vector<CSegmentoReta> retaInterpolacao;
    double rho, cp;
    double xmin, xmax, edx;

};

#endif // CMATERIALINTERPOLACAO_H
```

Apresenta-se na listagem 6.17 implementação da classe CMaterialInterpolacao.

Listing 6.17: Arquivo de implementação da função `main()`.

```
#include "CMaterialInterpolacao.h"

CMaterialInterpolacao::CMaterialInterpolacao(std::string fileName){
    std::string str_temp;
    int r, g, b, alpha;
    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 >>
                alpha;
13          color = QColor(r, g, b, alpha);
14
15          file >> str_temp; file >> rho;
16          file >> str_temp; file >> cp;
17
18          file >> str_temp; /// texto
19
20          double x1, x2, y1, y2;
21          file >> x1 >> y1;
22          xmin = x1;
23          while(file >> x2 >> y2){
24              retaInterpolacao.push_back( CSegmentoReta(x1,y1,x2,y2)
                    );
25              x1 = x2;
26              y1 = y2;
27          }
28          xmax = x1;
29          edx = (xmax-xmin)/ (retaInterpolacao.size()-1);
30      }
31      else{
32          std::cout<<"can't␣open␣file!" << std::endl;
33      }
34  }
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 6.18 o arquivo de cabeçalho da classe CSegmentoReta.

Listing 6.18: 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.
10class CSegmentoReta : public CReta
11{
12private:
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
17public:
18  /// Construtor default.
19  CSegmentoReta ( ) { }
20
21  /// Construtor sobrecarregado, recebe pontos (x1,y1), (x2,y2).
22  CSegmentoReta (double x1, double y1, double x2, double y2)
23    :CReta(x1,y1,x2,y2),xmin{x1},xmax{x2} {}
24
25  /// Construtor copia.
```

```
26  CSegmentoReta (const CSegmentoReta& retaInterpolacao )  {
27    xmin = retaInterpolacao.xmin;  xmax = retaInterpolacao.xmax;
         ok = retaInterpolacao.ok;
28    x = retaInterpolacao.x;      y = retaInterpolacao.y;
29    a = retaInterpolacao.a;      b = retaInterpolacao.b;
30  }
31
32  // Metodos Get/Set
33  double Xmin( )            { return xmin;  }
34  void Xmin(double _xmin ) { xmin = _xmin; }
35  double Xmax( )            { return xmax;  }
36  void Xmax(double _xmax ) { xmax = _xmax; }
37
38  /// Se retorno for verdadeiro, valor de y esta dentro intervalo
       xmin->xmax.
39  bool Ok()                 { return ok; }
40
41  /// Verifica se esta no intervalo de xmin->xmax.
42  bool TestarIntervalo (double _x)   { return ok = ( _x >= xmin and
       _x <= xmax)? 1:0;  }
43
44  /// Calcula valor de y = Fx(x);
45  virtual double Fx (double _x)  {
46    TestarIntervalo(_x);
47    return CReta::Fx(_x);
48  }
49
50  /// Calcula valor de y = Fx(x);
51  double operator()(double _x) { return Fx(_x); }
52
53  /// Sobrecarga operador <<, permite uso cout << reta;
54  friend std::ostream& operator<<( std::ostream& os, const
       CSegmentoReta& retaInterpolacao )  {
55    os.precision(10);
56    os<< retaInterpolacao.xmin << "␣->␣" << retaInterpolacao.xmax
57      << "␣:␣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 6.19 implementação da classe CSegmentoReta.

Listing 6.19: Arquivo de implementação da função `main()`.

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

Apresenta-se na listagem 6.20 o arquivo de cabeçalho da classe CReta.

Listing 6.20: Arquivo de implementação da classe CReta.

```
1 #ifndef CReta_H
2 #define CReta_H
3
4 #include <sstream>
5 #include <iomanip>
6 #include <fstream>
7
8 /// Class CReta,  representa uma reta y = a + b * x.
9 class CReta
10 {
11 protected:
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 sobrecarregado, recebe a e b.
21   CReta (double _a, double _b): b{_b},a{_a}{ }
22
```

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

```
57     return os.str();  }
58 };
59 #endif //CReta_H
```

Apresenta-se na listagem 6.21 implementação da classe CReta.

Listing 6.21: Arquivo de implementação da função `main()`.

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

Capítulo 7

Teste

Neste capítulo, será apresentado os testes e resultados do simulador. Inicialmente, o simulador será validado com a solução analítica unidimensional.

A seguir, serão apresentados resultados aplicados à indústria do petróleo, como injeção térmica em reservatórios, simulação reduzida de *five-spot* e, por fim, uma aplicação real na tecnologia.

7.1 Validação do simulador

Para validar os resultados do simulador, foi comparado os resultados do simulador, com a solução proposta no [Incropera, 2008] (equação 5.57).

A solução para o caso unidimensional é:

$$\frac{T - T_s}{T_i - T_s} = erf\left(\frac{x}{2\sqrt{\alpha t}}\right) \tag{7.1}$$

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

$$\alpha = \frac{k}{\rho C_p} \tag{7.2}$$

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 7.1 mostra a aplicação do problema no simulador.

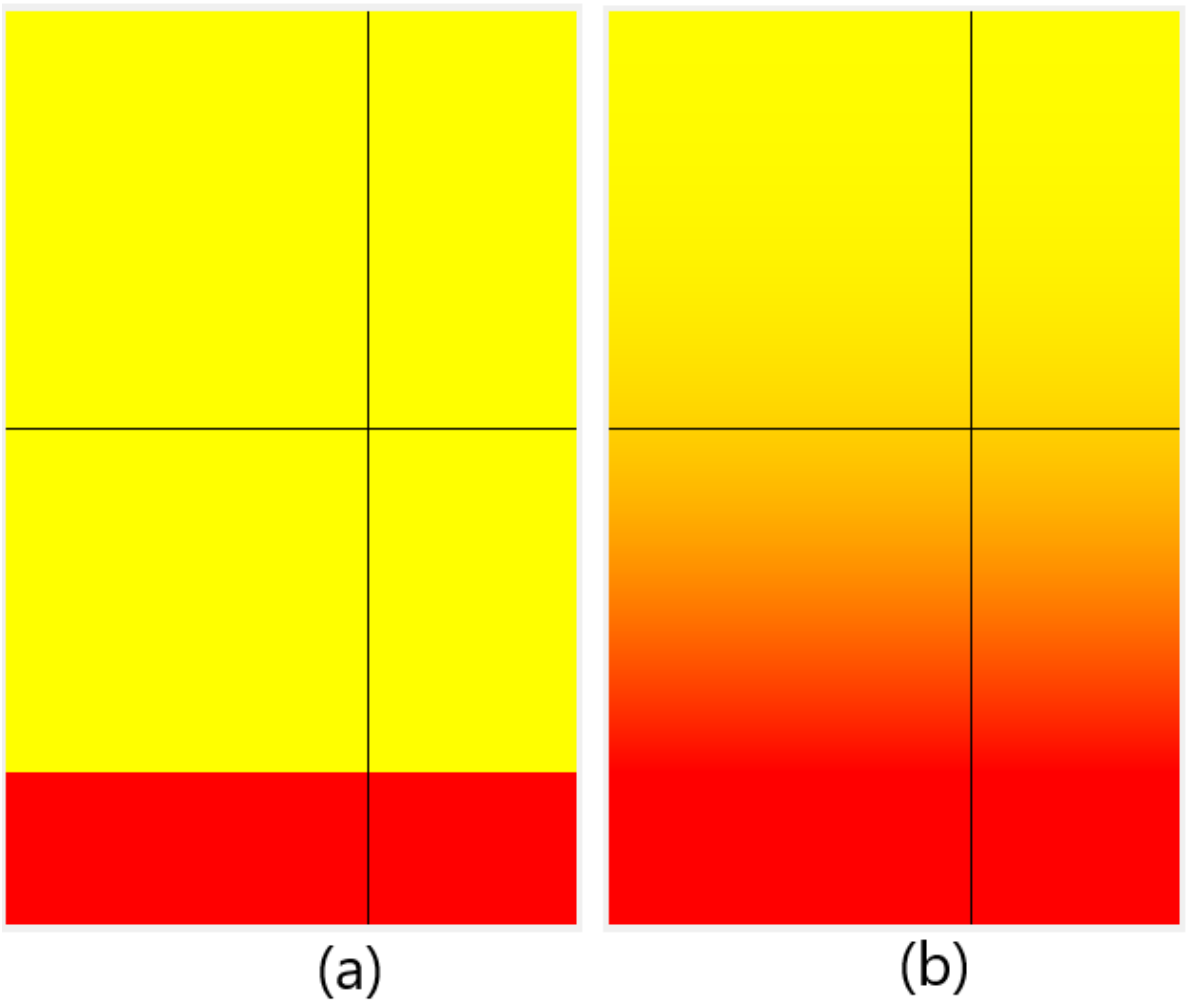


Figura 7.1: Aplicação do problema unidimensional no simulador. (a) é no tempo inicial e (b) depois de 100 segundos.

Para comparar os resultados do simulador com a solução analítica da Equação 7.1, foi programado um código em python apresentado abaixo:

Listing 7.1: Arquivo de implementação da validação.

```
import numpy as np
import matplotlib.pyplot as plt
import math

def temperature(x,t, alfa):
  Ti = 300
  Tf = 1000
  return Tf - (Tf - Ti)*math.erf(x/(2.0*math.sqrt(alfa*t)))

def maior_erro(x_sim, t_sim, t, alfa):
  T_analitico = []
  erro = []
  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
25x = np.linspace(0,0.10374,100)
26t = [50.0, 100.0]
27
28k = 40
29rho = 1600
30cp = 4000
31alfa = k/(rho*cp)
32
33for _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####################
40f = open('vertical100.000000.dat', 'r')
41x_sim = []
42t_sim = []
43for i in f:
44    split = i.split(';')
45    x_sim.append(float(split[0]))
46    t_sim.append(float(split[1].replace('\n', '').replace(' ', ''))
       )
47
48t_sim.sort(reverse=True)
49for 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',
     'Simulador - 50'])
80 plt.show()
```

Como resultado do código acima, é apresentado um gráfico comparando dois tempos: 50 e 100 segundos.

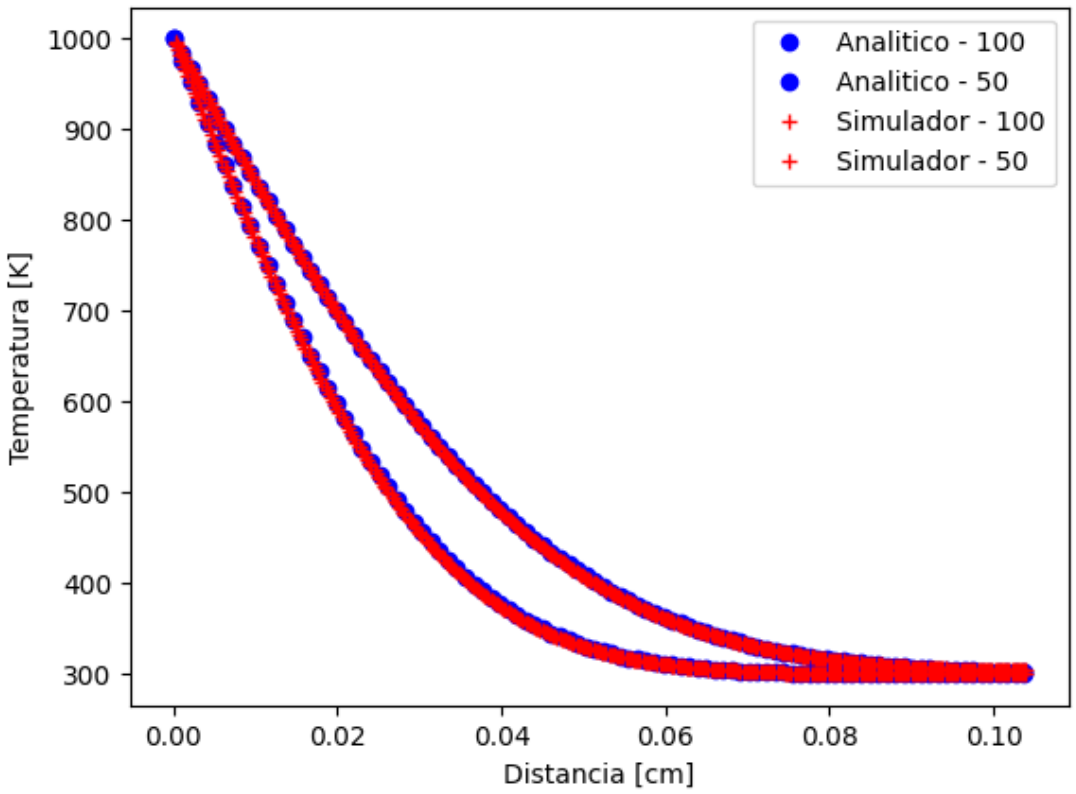


Figura 7.2: Comparação da solução da equação de calor com o resultado do simulador.

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

Tabela 7.1: 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 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 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".

7.2 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 7.2.

Tabela 7.2: Tabela com propriedades termofísicas [Dong et al., 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. 7.3), onde ψ é a propriedade termosfísica analisada.

$$\psi = \phi\psi_1 + (1-\phi)\psi_2 \tag{7.3}$$

Com isso, é obtido a propriedade do arenito com água na Tabela 7.3.

Tabela 7.3: 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 7.3. 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.

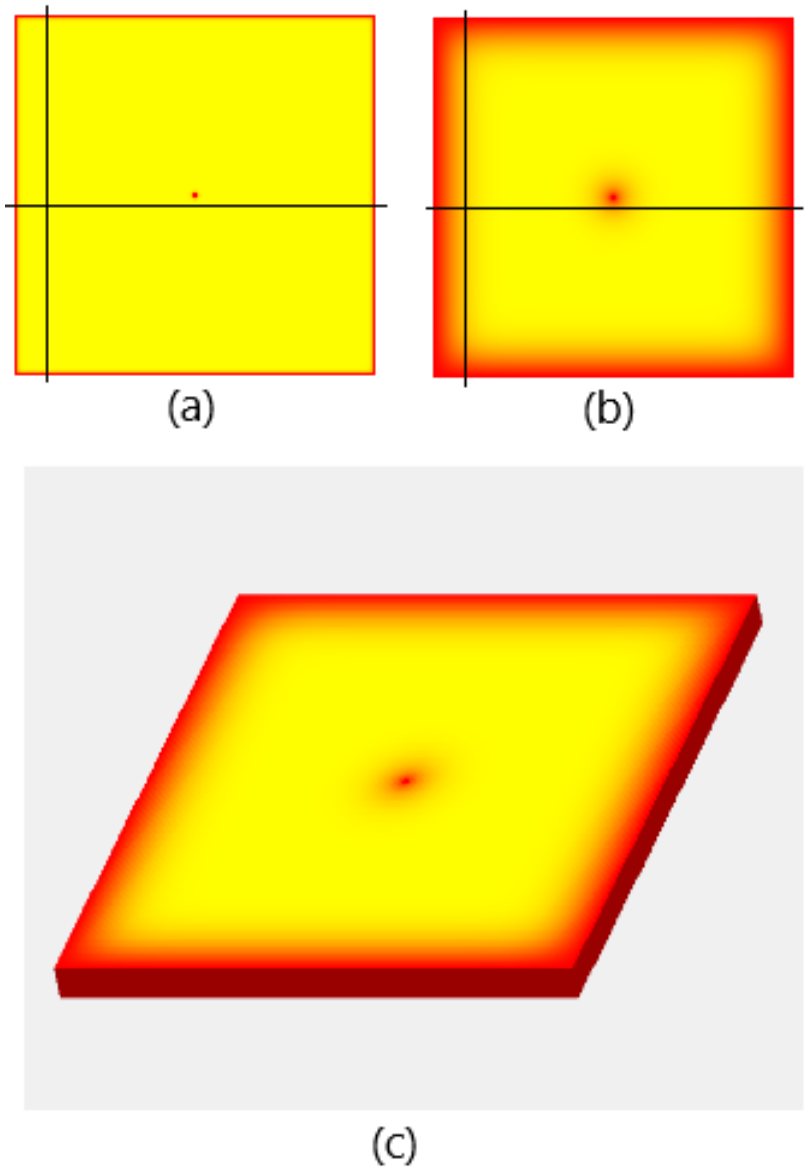


Figura 7.3: Resultados da simulação com renderização 3D.

A Figura 7.4 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.

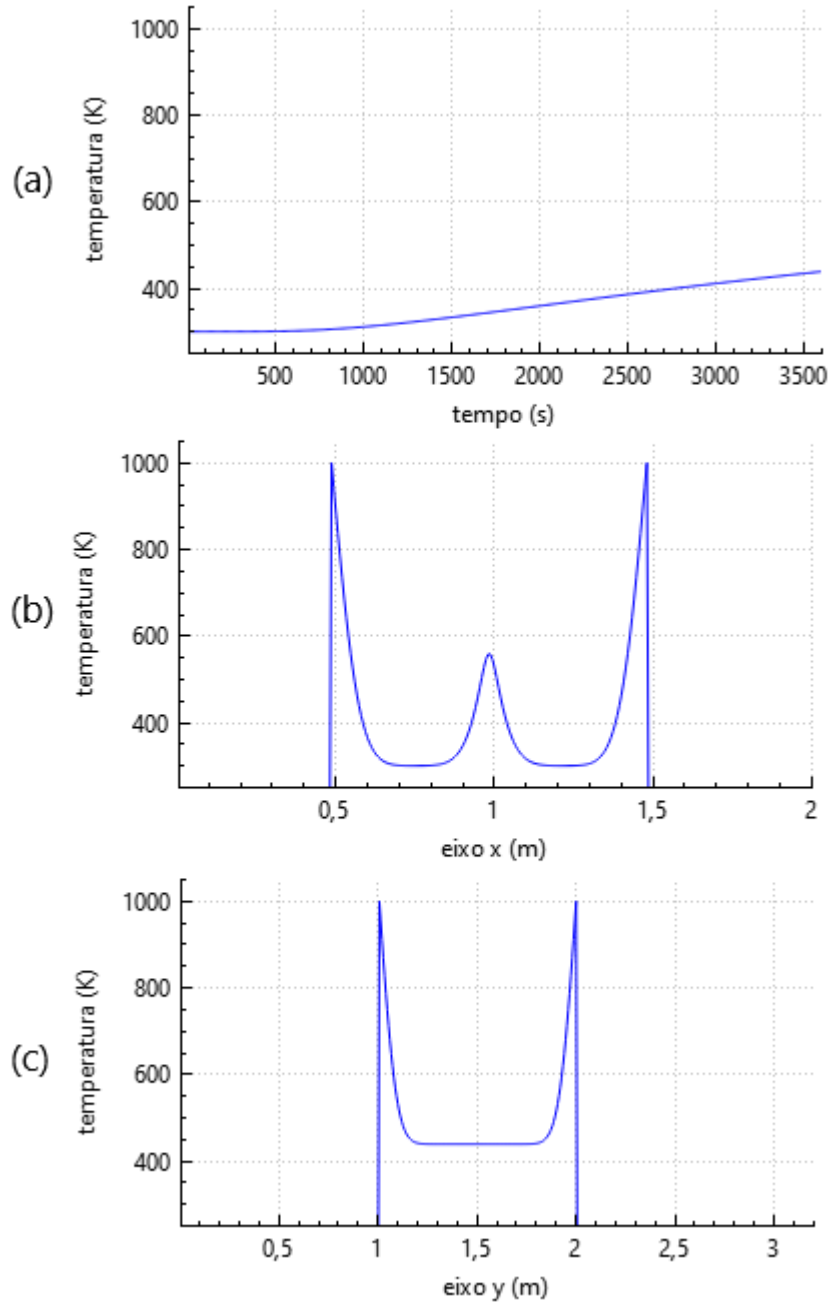


Figura 7.4: Gráficos da simulação para o tempo de 3.600 segundos.

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".

7.3 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 7.3).

Na Figura 7.5 é apresentado o resultado do modelo *five-spot* com os 5 poços injetores.

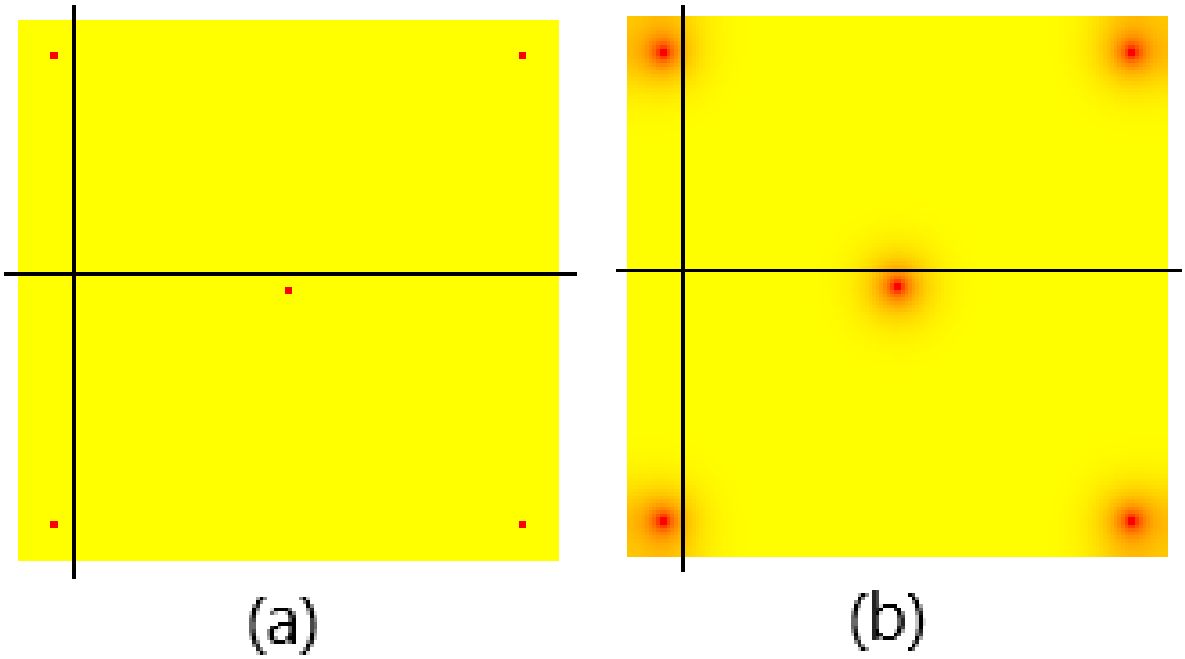


Figura 7.5: Resultados da simulação do primeiro modelo *five-spot* após 3.600 segundos.

Analisando os resultados gráficos na Figura 7.6, 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.

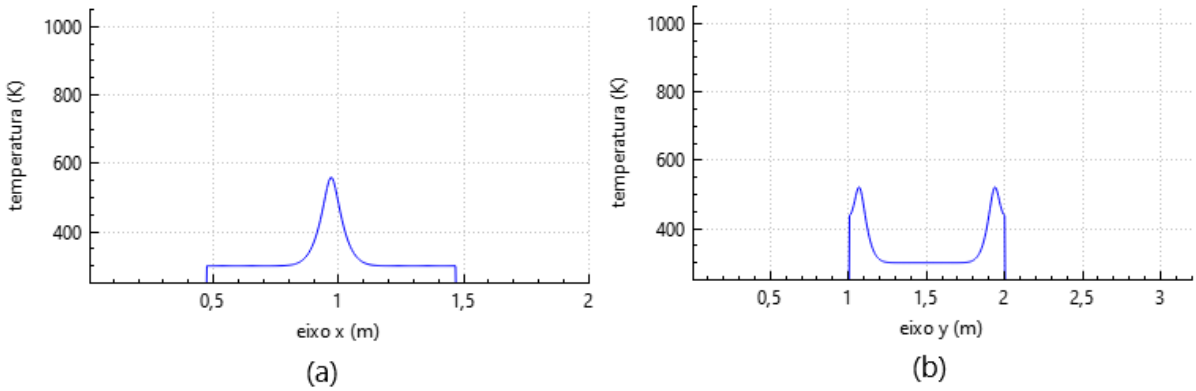


Figura 7.6: Gráficos da simulação do primeiro modelo *five-spot*.

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 temop 460.800 segundos, ou 128 horas.

A Figura 7.7 mostra em (a) o cenário inicial, (b) o cenário final, próximo ao regime permanente e em (c) a renderização do reservatório em 3D.

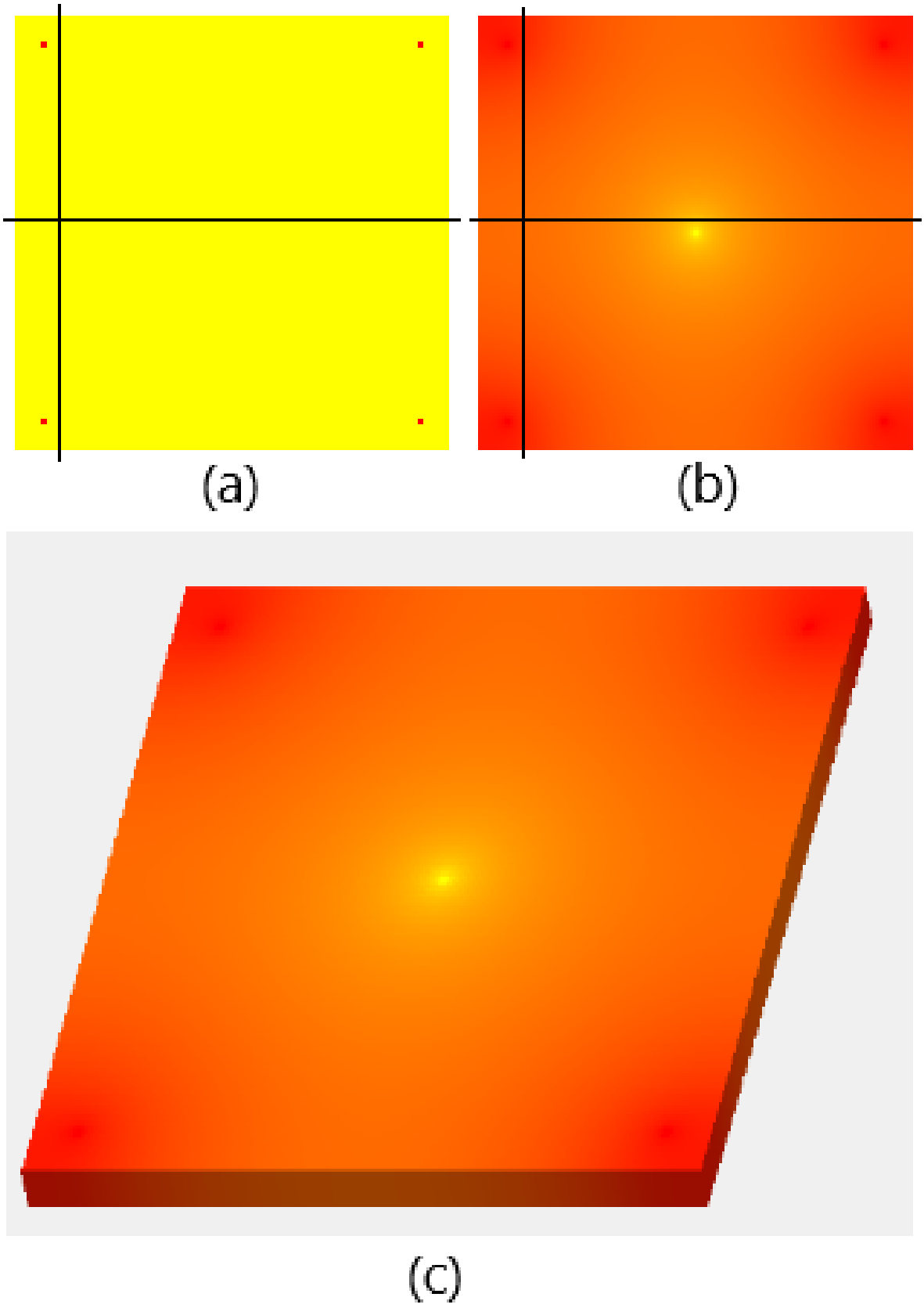


Figura 7.7: Resultados da simulação do segundo modelo *five-spot* após 460.800 segundos.

Abaixo, seguem os gráficos na Figura 7.8. Em (a), é possível perceber uma tem-

peratura 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.

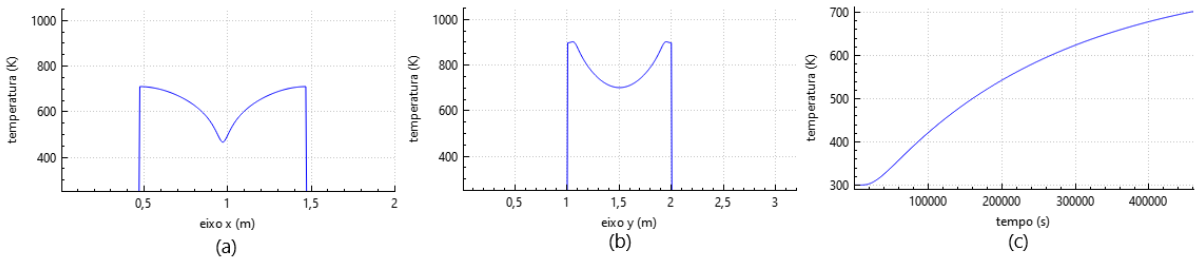


Figura 7.8: Gráficos da simulação do segundo modelo *five-spot*.

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

7.4 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:

Tabela 7.4: 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:

Tabela 7.5: 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.

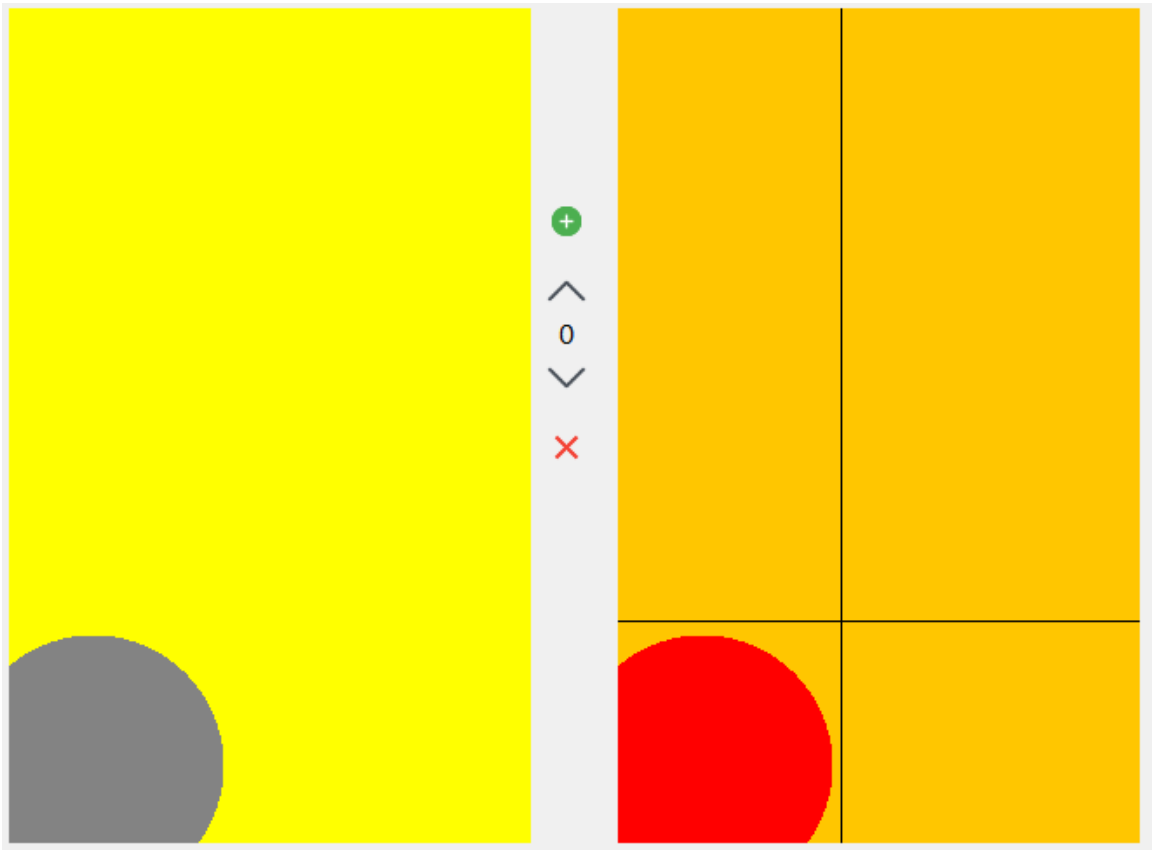


Figura 7.9: Modelo 1 de injeção térmica em reservatório.

Com o modelo da Figura 7.9, 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 7.10 com tempo de 4.000 segundos.

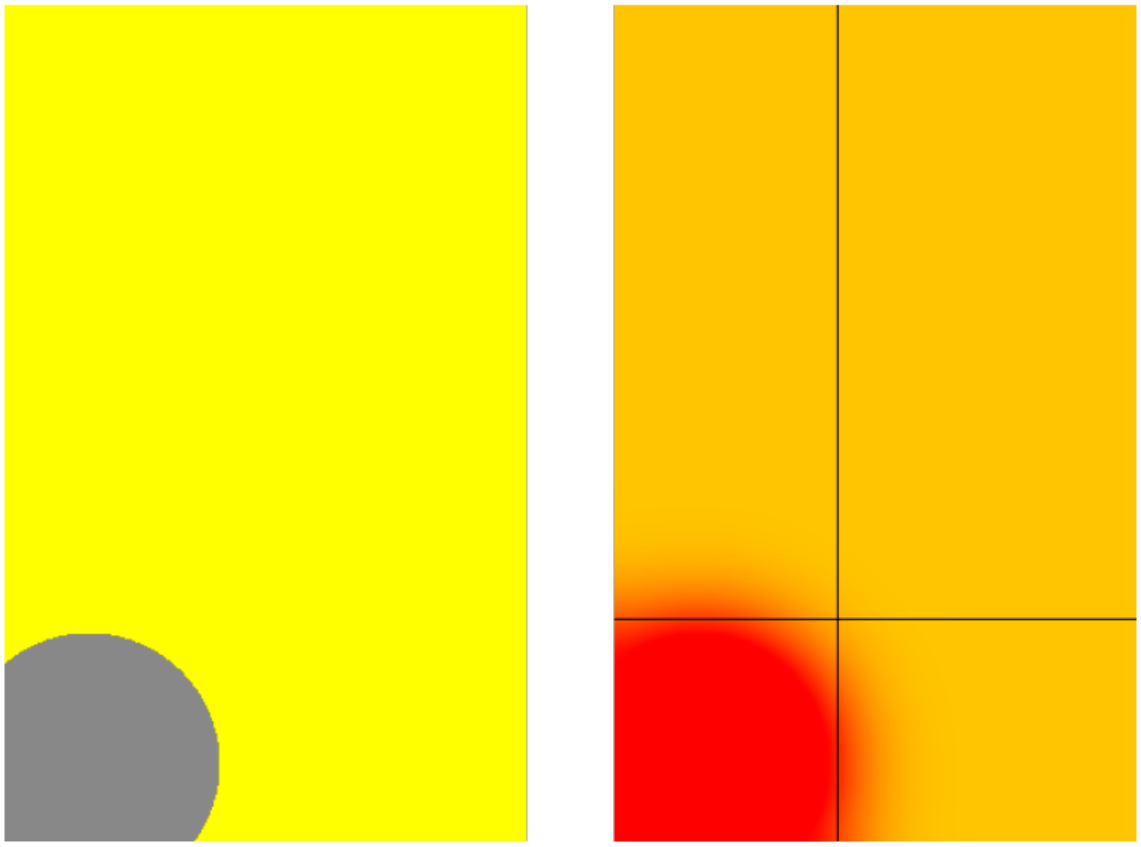


Figura 7.10: Modelo 1 de injeção térmica em reservatório após 4.000 segundos.

Os gráficos são mostrados na Figura 7.11. 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.

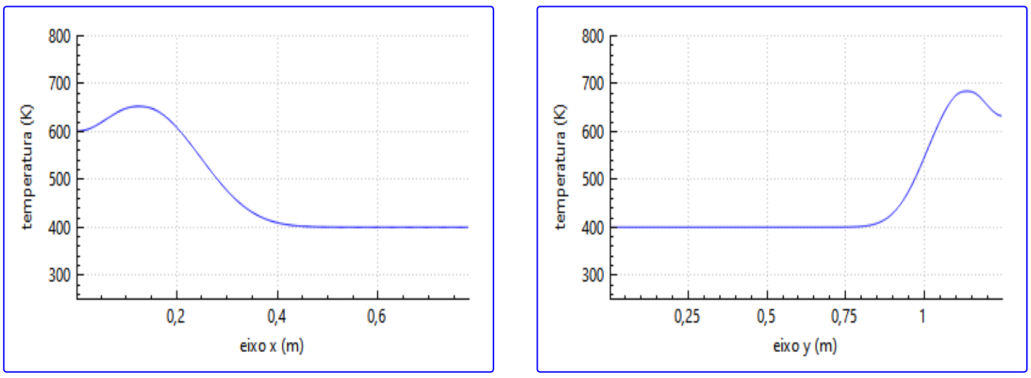


Figura 7.11: Graficos mostrando a variação de temperatura na região próxima ao poço.

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

7.5 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 7.12

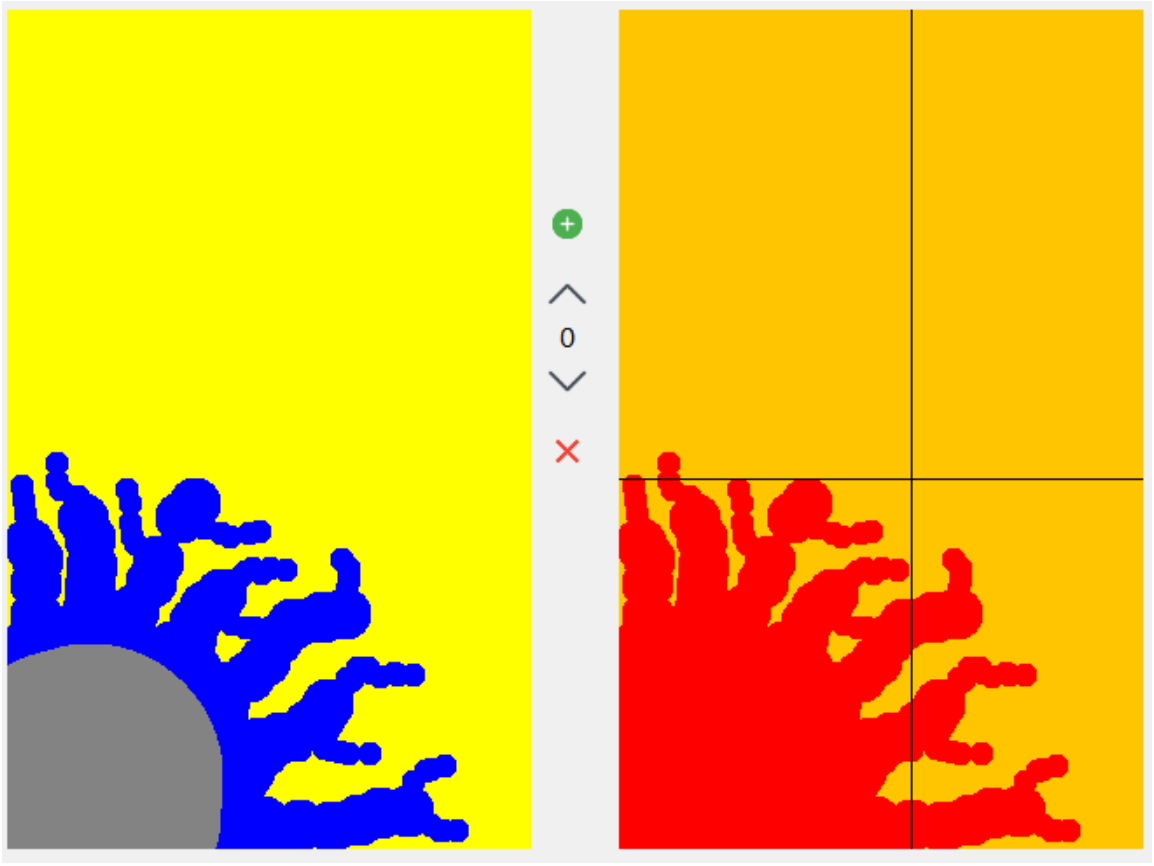


Figura 7.12: Tempo inicial da simulação. Na esquerda, o cinza representa o poço, azul a água e o amarelo, arenito. Na direita, é mostrado as temperaturas.

Com a evolução do tempo, a região mais próxima dos *fingers* de água, é a mais alterada. A Figura 7.13 mostra esse cenário.

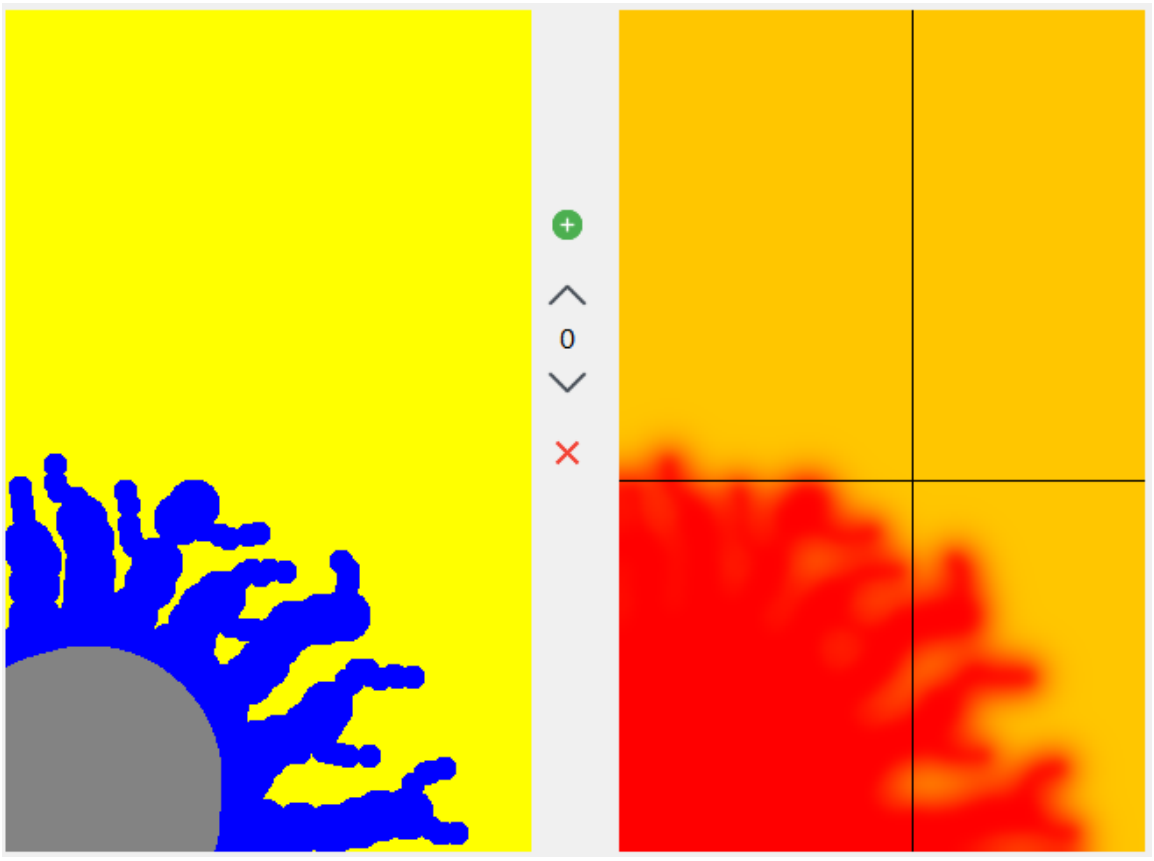


Figura 7.13: Evolução da simulação. Tempo de 610 segundos.

Avançando mais no tempo, chegando a 7.180 segundos, é possível perber que a região dos *fingers* de água está com temperatura bem distribuída (Figura 7.14), se assemelhando ao modelo 1, mas com poço muito mais largo.

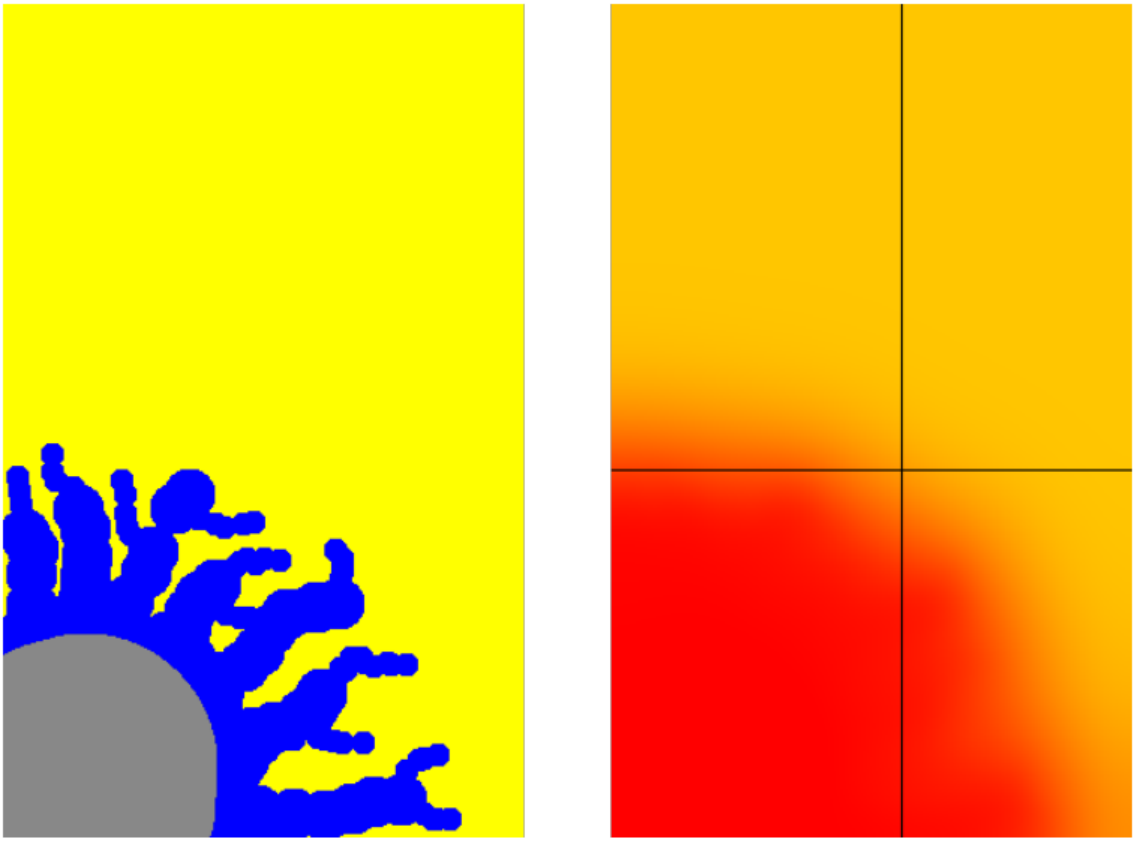


Figura 7.14: Tempo final de 7.180 segundos.

A variação de temperatura ao longo das retas de estudo foram mais suaves em relação ao modelo 1 (Figura 7.15).

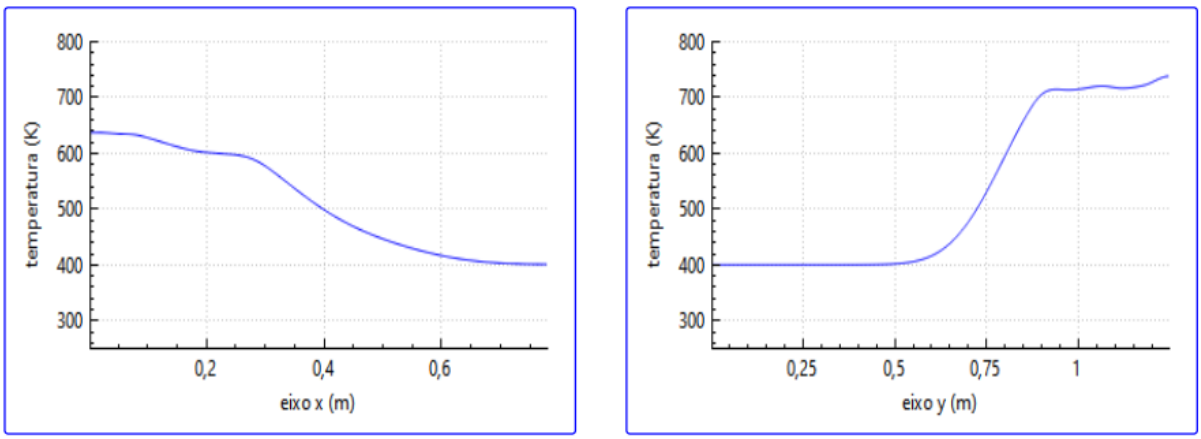


Figura 7.15: Gráficos do tempo final de 7.180 segundos.

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".

7.6 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 7.16, 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.

Figura 7.16: 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.

Tabela 7.6: Tabela com as propriedades termofísicas do modelo do *notebook* - Cobre

Propriedade	Valor
C_p	353
k	42
ρ	7.262

Na figura 7.17, 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.

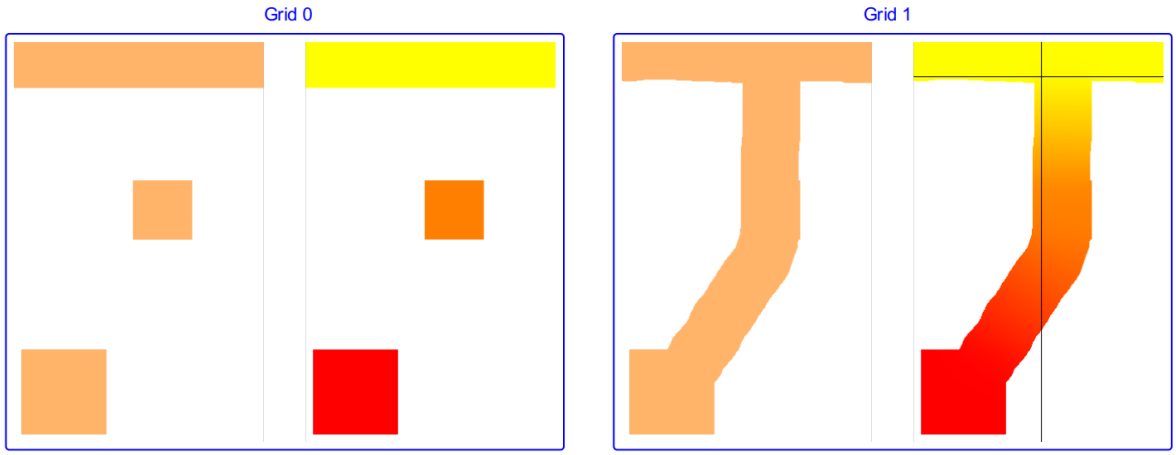


Figura 7.17: Simulação do sistema de resfriamento do notebook após chegar ao período permanente.

Na figura 7.18, são apresentados os gráficos da temperatura ao longo da horizontal (esquerda) e vertical (direita).

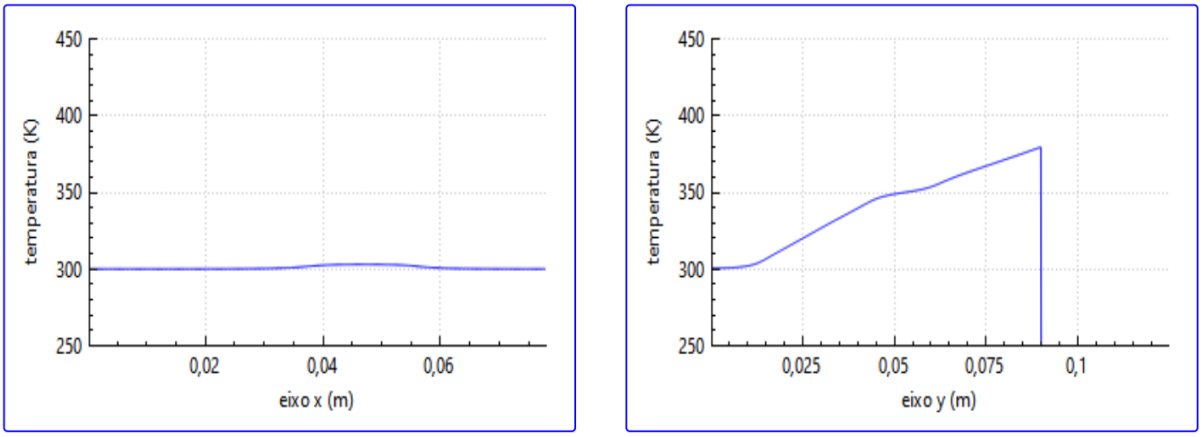


Figura 7.18: Interior de um *notebook*, apresentando o *heatpipe,* que é a barra de cobre que cruza a GPU e CPU, e resfria na ventoinha.

A temperatura é rapidamente dispersada quando chega à ventoinha. O gráfico da direita da Figura 7.18, 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 7.16. 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”.

Capítulo 8

Documentação

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

8.1 Documentação do usuário

Descreve-se aqui o manual do usuário, um guia que explica, passo a passo a forma de instalação e uso do software desenvolvido.

8.1.1 Como rodar o software

Para rodar o software será necessário a instação do Qt Creator. A partir desse software, será possível abrir o projeto desenvolvido, juntamente com todos os seus códigos.

8.1.2 Como utilizar o software

A seguir, será explicado todas funcionalidades da janela do software, o significado delas, e como utilizar.

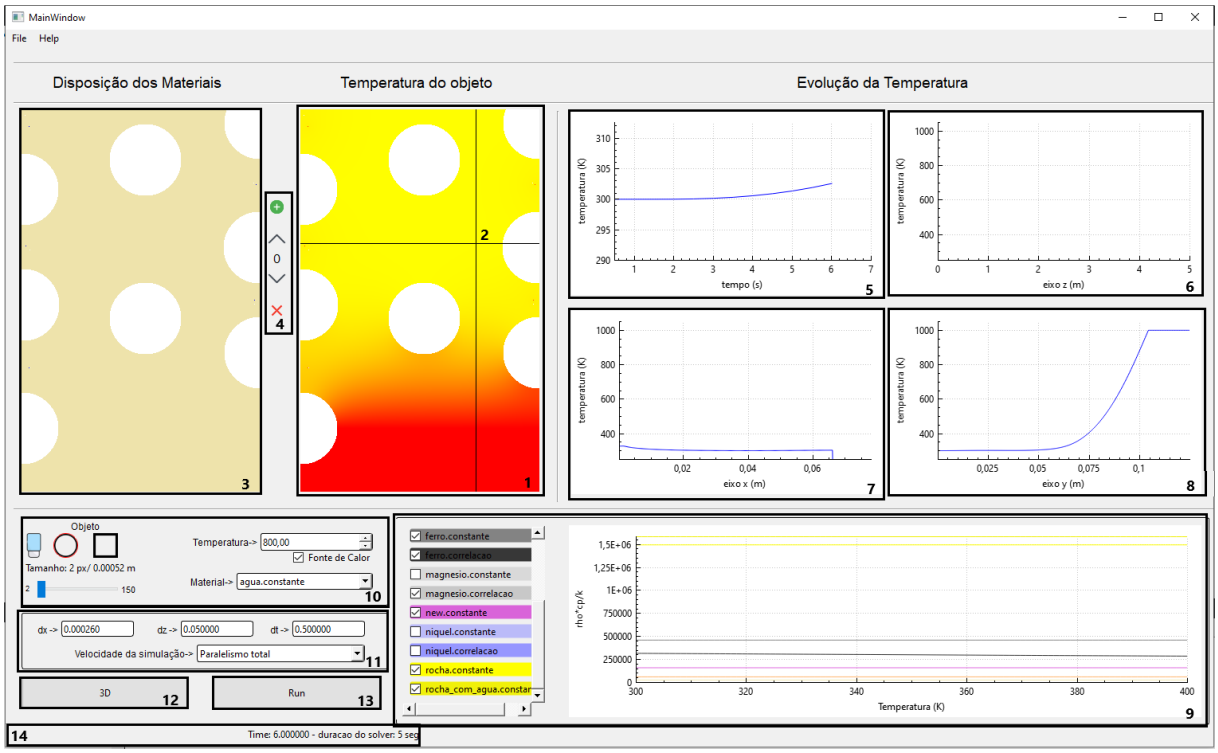


Figura 8.1: Guia para utilizar o software.

O Figura 8.1 mostra a janela principal do software e foram listadas 14 grupos de funcionalidades importantes ao usuário.

1. Região onde o usuário desenha o objeto desejado. Para desenhar, deve ser clicado com o botão esquerdo do mouse.

2. 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.

3. Região onde o usuário consegue diferenciar os materiais utilizados na simulação.

4. 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.

5. 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.

6. Gráfico da temperatura ao longo das camadas. Como só foi criado uma única camada, o gráfico ficará vazio.

7. Gráfico da temperatura ao longo da horizontal. Mostra a temperatura ao longo da reta horizontal de estudo.

8. Gráfico da temperatura ao longo da vertical. Mostra a temperatura ao longo da reta vertical de estudo.

9. Gráfico com as propriedades termofísicas ao longo da temperatura. O usuário pode selecionar o material que quer analisar na região esquerda da área destacada.

10. 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.

11. Propriedades da simulação. Aqui o usuáro 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)

12. Botão para iniciar a janela com a renderização 3D. Para navegar nessa janela, são listados os seguintes botões:

    (a) Espaço: muda a cor entre temperatura ou materiais.

    (b) Pg Up: zoom in.

    (c) Pg Down: zoom out.

    (d) w/a/s/d: configura o ângulo do objeto.

    (e) setas: move o objeto na janela.

    (f) mouse: mesmas funcionalidades de (d).

13. Botão para iniciar ou parar a simulação.

14. 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)

8.2 Documentação para desenvolvedor

Nesta seção, é apresentado informações para desenvolvedores, como a documentação em html, e a listagem de algumas dependências específicas.

A documentação em html foi gerada utilizando o *software* Doxygen, como mostrada na Figura 8.2, disponível em https://www.doxygen.nl/download.html;

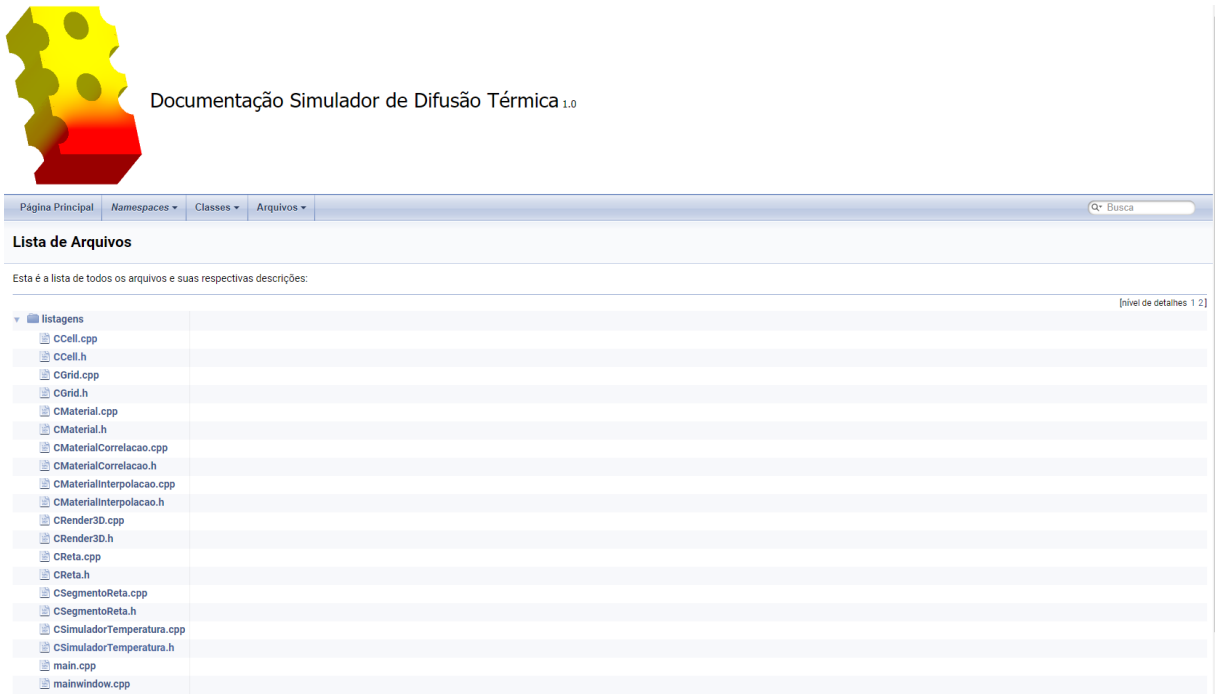


Figura 8.2: Logo e documentação do *software*

Ao clicar sobre qualquer item da listagem acima, será possível analisar o código daquele arquivo, como mostrado na Figura 8.3

Figura 8.3: Código fonte da classe CSimuladorTemperatura, no Doxygen

8.2.1 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 `yum install gcc`.
- Biblioteca Qt disponível em https://www.qt.io/download;

Referências Bibliográficas

[BUENO, 2003] BUENO, A. D. (2003). *ProgramaÃ§Ã£o Orientada a Objeto com C++*. Novatec. 9

[Dong et al., 2015] Dong, Y., McCartney, J. S., and Lu, N. (2015). Critical review of thermal conductivity models for unsaturated soils. 33(2):207–221. 99

[FOURIER, 1822] FOURIER, J. B. J. (1822). *Theorie Analytique de La Chaleur*. 1, 10

[Herter and Lott,] Herter, T. and Lott, K. Algorithms for decomposing 3-d orthogonal matrices into primitive rotations. 17(5):517–527. 21

[Incropera, 2008] Incropera, F. (2008). *Fundamentos de transfere?ncia de calor e de massa*. LTC. 10, 18, 94

[Lima, 2020] Lima, G. R. (2020). Simulador bidimensional de transferÃªncia de calor em meios porosos utilizando mÃ©todos numÃ©ricos de diferenÃ§as finitas. 99

[NUSSENZVEIG, 2014] NUSSENZVEIG, H. M. (2014). *Curso de fÃsica bÃ¡sica 2 : fluidos, oscilacÃµeseondas, calor. Blucher*. 9

[Rosa et al., 2006] Rosa, A. J., Carvalho, R. D. S., and Xavier, J. A. D. (2006). *Engenharia de reservatÃ³riosdepetro³eo. Intercincia*. 1, 12

[THOMAS, 2004] THOMAS, J. E. (2004). Fundamentos de engenharia de petrÃ³*leo*. 1

[Valencia and Quested, 2008] Valencia, J. J. and Quested, P. N. (2008). *Thermophysical Properties*. 18

Capítulo 9

Como adicionar 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 abaixo, e a extensão do arquivo deve ser '.constante', '.correlacao' ou '.interpolacao', conforme o modelo escolhido. a Figura 9.1 ilustra o local onde deve ser clicado para adicionar o material.

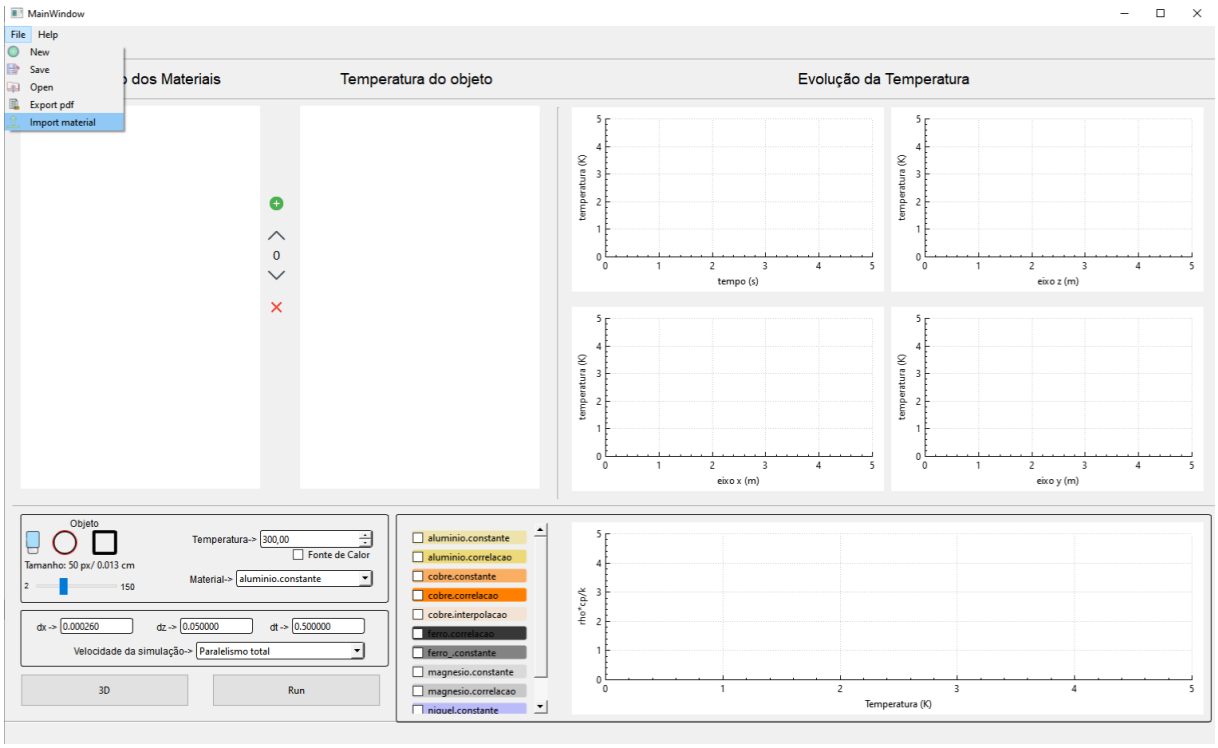


Figura 9.1: Como adicionar um material no simulador. Primeiro seleciona Arquivo, Import material. Uma janela será aberta, para o usuário escolher o material.

9.1 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.

9.2 Método de interpolação

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 RGBA 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.

Capítulo 10

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 <==

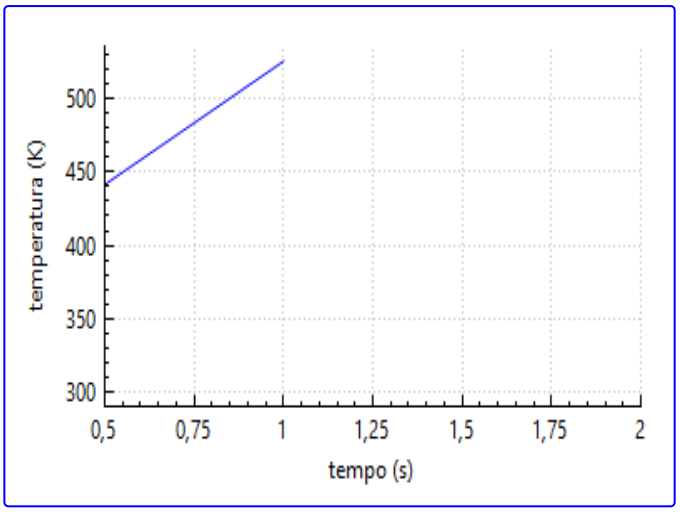
Temperatura máxima: 1000 K

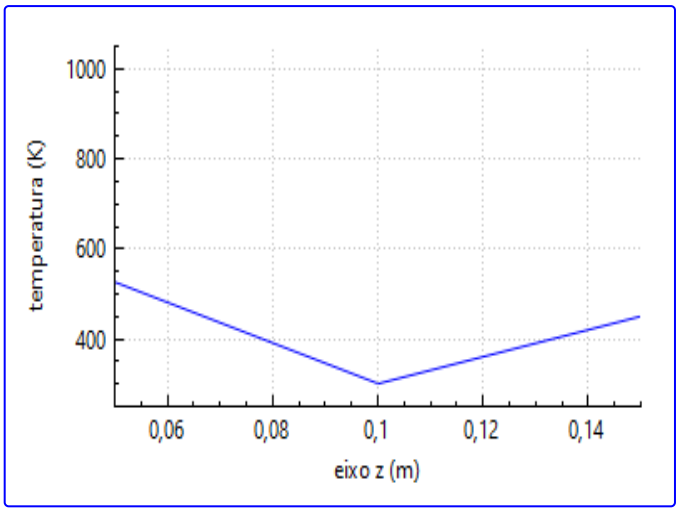
Temperatura mínima: 300 K

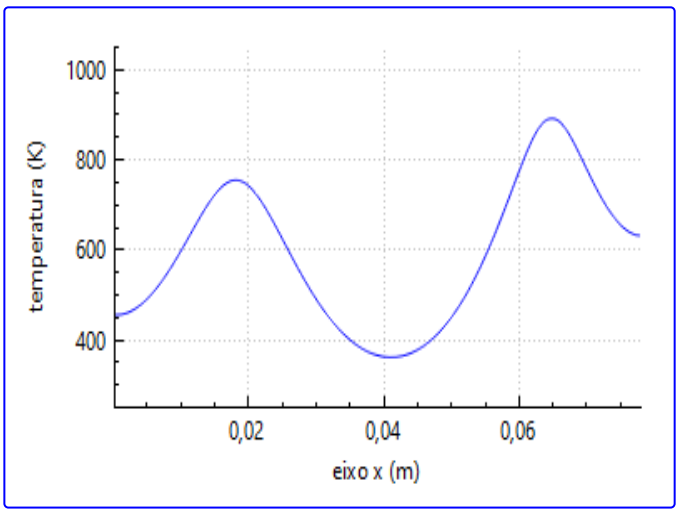
Tempo máximo: 1 s

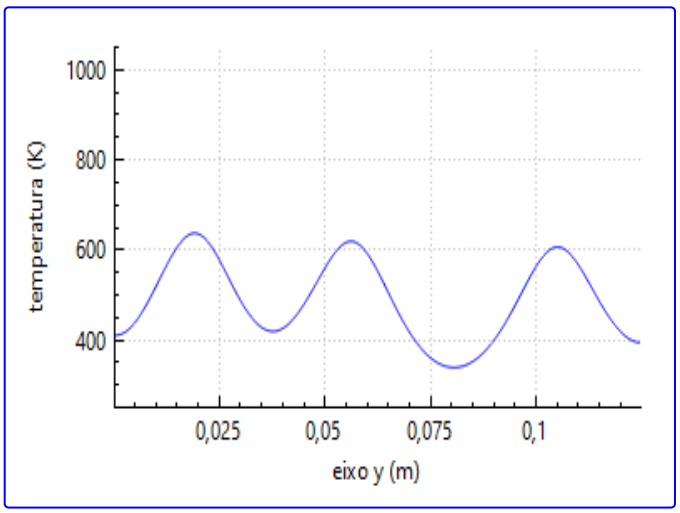
Tipo de paralelismo: Paralelismo total

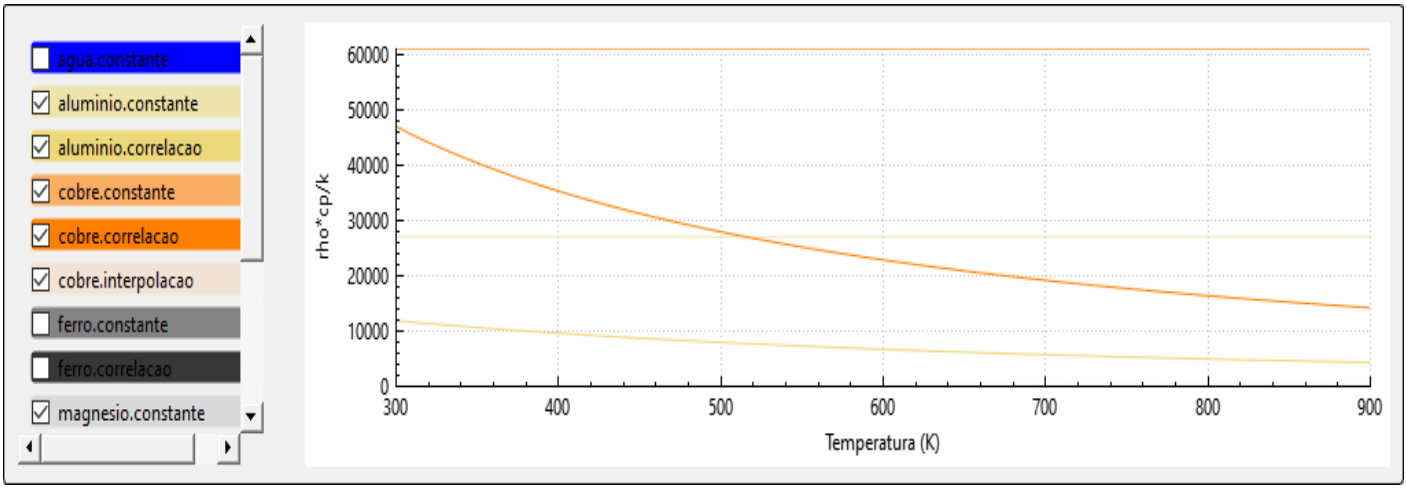
Coordenada do ponto de estudo (x,y,z): 0.02834,0.0637,0



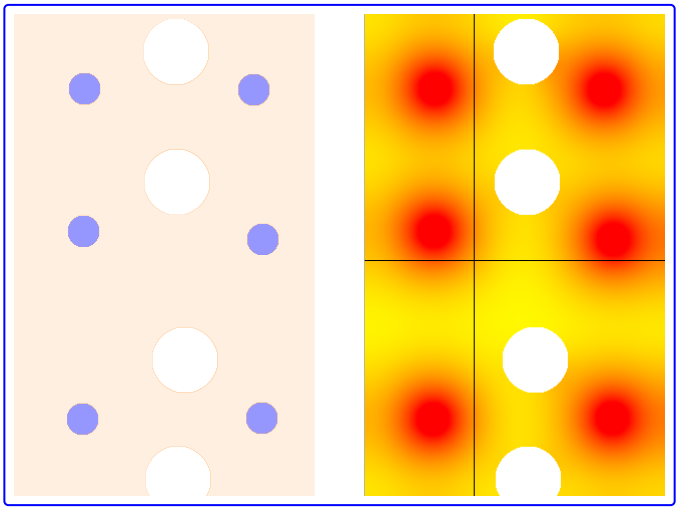




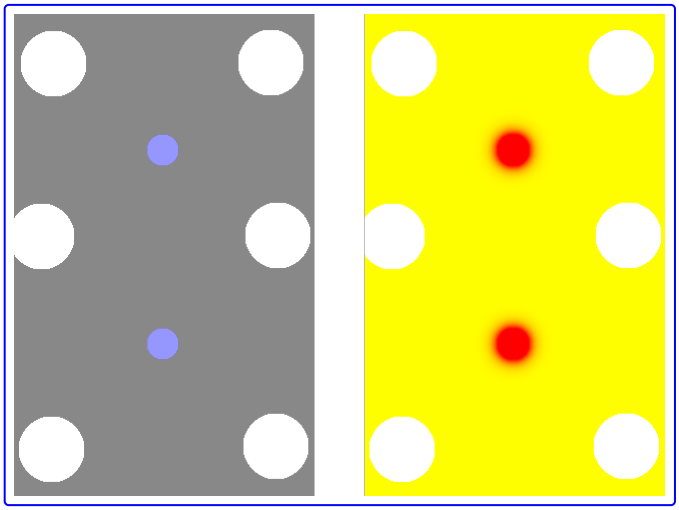




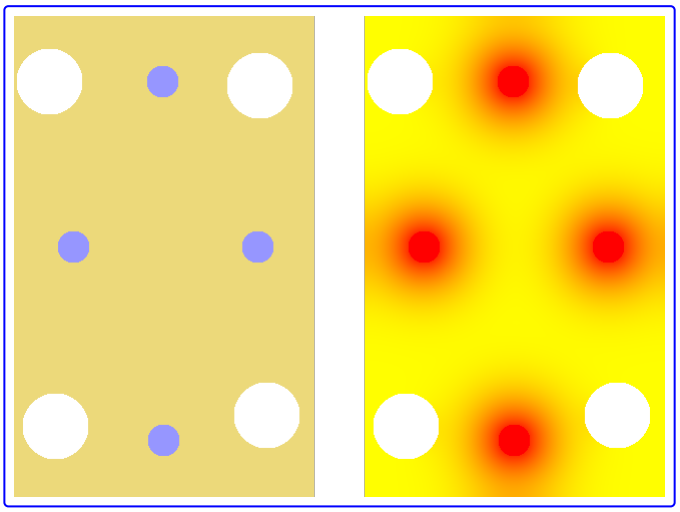
Grid 0



Grid 1



Grid 2



Índice Remissivo