Larissa Gabriela Barrozo da Silva Moura

Documentação de Implementação: Visualizador do fractal de Mandelbrot

1. Introdução

Este documento descreve a implementação do Mandelbrot , um projeto interativo de visualização de fractal. O objetivo central do projeto é demonstrar a interoperabilidade entre duas linguagens de programação com paradigmas e vocações distintas:

1. **Python: Utilizado para a camada de interface e gerenciamento de estado.**
2. **C: Utilizado para o motor de cálculo de alta performance.**

   A aplicação utiliza uma arquitetura onde o Python atua como se fosse o *frontend* gráfico e o C como o *backend* de cálculo, comunicando-se através de uma biblioteca compartilhada.

2. Visão Geral da Aplicação

A aplicação renderiza o conjunto de Mandelbrot definido pela função iterativa complexa:

$$z_{n+1} = z_n^2 + c$$

Onde:

- z é um número complexo iniciando em (0,0).
- c é a constante complexa correspondente ao ponto (pixel) no plano.

O número de iterações necessárias para a sequência divergir determina a cor do pixel. A aplicação suporta:

- Zoom Interativo: Aproximação e distanciamento via cliques do mouse.
- Navegação (Panning): Centralização do fractal no ponto clicado.
- Ajuste de Iterações: Controle em tempo real da precisão do cálculo através de um slider.
- Estatísticas: Exibição do tempo de processamento e dados do plano complexo.

3. Responsabilidades das Linguagens

3.1 O Python é responsável por:

- Criar a janela gráfica e elementos de controle (via Tkinter).
- Manipular entradas do usuário (cliques de mouse e ajuste de parâmetros).
- Gerenciar o estado do programa (coordenadas do plano, nível de zoom).
- Orquestrar a execução via threads para manter a interface responsiva durante o cálculo.
- Converter os dados brutos de pixels em imagens visíveis (via Pillow).

3.2 O C é responsável por:

- Implementar o algoritmo de escape de Mandelbrot de forma otimizada.
- Gerenciar buffers de memória para armazenamento de pixels e dados de iteração.
- Implementar o algoritmo de coloração por histograma, que garante uma transição suave de cores e melhor percepção visual do fractal.
- Exportar as funções através de uma biblioteca dinâmica (.so ou .dll).

4. Interface entre Python e C

A comunicação entre as linguagens é implementada através da biblioteca ctypes, permitindo chamadas de funções nativas e compartilhamento de memória.

4.1 Carregamento da Biblioteca Compartilhada

O objeto compartilhado compilado é carregado dinamicamente:

Python: **lib = ctypes.CDLL("./mandelbrot.so") # ou .dll no Windows**

4.2 Declaração da Assinatura da Função

Os tipos de argumentos e o tipo de retorno são declarados para garantir a conversão correta de tipos entre as linguagens:

lib.calcular_mandelbrot.restype = ctypes.POINTER(DadosMandelbrot)

lib.calcular_mandelbrot.argtypes = [

ctypes.c_int, ctypes.c_int, # largura, altura

ctypes.c_double, ctypes.c_double, # x_min, x_max

ctypes.c_double, ctypes.c_double, # y_min, y_max

ctypes.c_int # max_iteracoes]

4.3 Estratégia de Compartilhamento de Memória - Interoperabilidade

Para garantir que ambas as linguagens "entendam" o mesmo bloco de memória, definiu uma struct idêntica em ambos os lados:

4.3.1. No C (main.c e mandelbrot.h)

Python.h no main.c e no

O arquivo main.c que usa #include <Python.h> é uma **Extensão Nativa**.

- **Tradução de Tipos:** PyArg_ParseTuple é uma conversa onde o C "pergunta" ao interpretador Python: *"O que tem dentro desse objeto Python?"* e o converte para double ou int.

- **Criação de Objetos:** Com Py_BuildValue, o C está ativamente criando um objeto que o Python entende.

4.3.2 No Python (main.py):

ctypes no main.py

No main.py, usa a biblioteca ctypes, permitindo o **compartilhamento de memória binária**.

- **Espelhamento de Estruturas:** se define class DadosMandelbrot(ctypes.Structure), do qual se diz ao Python para reservar um bloco de memória que segue exatamente o layout da struct do C.

C:
```
typedef struct {
    int largura;
    int altura;
    int max_iteracoes;
    double x_min, x_max, y_min, y_max;
    uint8_t* pixels;
} DadosMandelbrot;
```

PYTHON:
```
class DadosMandelbrot(ctypes.Structure):
    _fields_ = [
        ("largura", ctypes.c_int),
        ("altura", ctypes.c_int),
        ("max_iteracoes", ctypes.c_int),
        ("x_min", ctypes.c_double),
        ("x_max", ctypes.c_double),
        ("y_min", ctypes.c_double),
        ("y_max", ctypes.c_double),
        ("pixels",
ctypes.POINTER(ctypes.c_uint8))
    ]
```

- **Conversa via Ponteiros:** O comando ponteiro.contents , em def _processar_calculo(self), no main.py, acessa a memória RAM que o C alocou. O Python não "copia" os dados nesse momento, ele olha para o mesmo endereço de memória que o C acabou de escrever.
- **Leitura de Bytes Brutos:** O uso de ctypes.string_at(dados.pixels, tamanho_buffer) é o ponto alto da conversa. O Python entra "dentro" do ponteiro de pixels gerado pelo C e lê os dados brutos para montar a imagem :

```
def _processar_calculo(self):
….. codigo
 dados = ponteiro.contents
      tamanho_buffer = dados.largura * dados.altura * 3
 pixels_brutos = ctypes.string_at(dados.pixels, tamanho_buffer) ….codigo
```

A comunicação utiliza uma estrutura de dados (struct) definida em C e espelhada em Python. O C aloca um buffer de memória para os pixels:

dados->pixels = (uint8_t*)malloc(largura * altura * 3);

O Python acessa essa memória diretamente por um ponteiro, evitando cópias desnecessárias e garantindo alta performance na transferência de dados da imagem e para garantir a segurança de memória , o Python chama explicitamente a função de liberação do C após processar a imagem:

1. O Python chama a função em C, que utiliza malloc para alocar um buffer de memória para os pixels no *Heap*.
2. **Processamento:** C preenche esse buffer com os cálculos do fractal e retorna o ponteiro da estrutura para o Python.
3. **Acesso à Memória:**O Python acessa essa memória diretamente através do ponteiro (ponteiro.contents), minimizando o overhead de comunicação.
4. **Extração de Dados:** Através de ctypes.string_at(dados.pixels, tamanho_buffer), o Python realiza uma leitura rápida dos bytes brutos diretamente do endereço de memória alocado pelo C
5. **Reconstrução da Imagem:** O objeto Image.frombytes do Pillow reconstrói a matriz visual a partir desses bytes.
6. **Gerenciamento de Ciclo de Vida:** Para garantir a **Memory Safety**, o Python ordena explicitamente a liberação da memória no C através da função liberar_dados_mandelbrot, evitando *memory leaks*.

Trecho correspondente no código:

pixels_brutos = ctypes.string_at(dados.pixels, tamanho_buffer)

imagem = Image.frombytes("RGB", (dados.largura, dados.altura), pixels_brutos)

lib.liberar_dados_mandelbrot(ponteiro)

5. Método de Interface (FFI via ctypes)

A integração entre as duas linguagens ocorre através uma **Foreign Function Interface (FFI)** utilizando o módulo ctypes do Python. O método de comunicação baseia-se em **Compartilhamento de Memória via Estruturas**:

1. **Mapeamento de Tipos:** No C, define-se uma struct contendo as dimensões da imagem e o ponteiro para o array de pixels (uint8_t*). No Python, uma classe equivalente é criada herdando de ctypes.Structure.

   No main.py: class DadosMandelbrot(ctypes.Structure):

   ```
   _fields_ = [
      ("largura", ctypes.c_int),
      # ... todos os campos double ...
      ("pixels", ctypes.POINTER(ctypes.c_uint8))
   ]
   ```

2. **Carregamento Dinâmico:** O Python carrega a biblioteca compilada (.dll ou .so) em tempo de execução, na função carregar_biblioteca():

   biblioteca = ctypes.CDLL(caminho_lib) # Aqui o Python "puxa" a DLL/SO para dentro de si

3. **Ponteiros de Memória:** O Python chama a função em C passando parâmetros de configuração. O C aloca a memória necessária, preenche o buffer com os cálculos e retorna o ponteiro da estrutura para o Python. O Python então lê diretamente esse endereço de memória para gerar a imagem.

   biblioteca.calcular_mandelbrot.restype = ctypes.POINTER(DadosMandelbrot) # O C devolve um ponteiro

6. Análise de Dependência Mútua

6.1. Indispensabilidade do C:

Aritmética de Ponto Flutuante: O Python não possui a lógica de cálculo do fractal internamente. Sem a biblioteca compilada em C, a aplicação é incapaz de gerar a matriz de pixels. Estão representadas na chamada **lib.calcular_mandelbrot(...)**

Otimização de Memória: O motor em C realiza o pós-processamento de cores via **Histogram Coloring** diretamente ao nível de bytes (uint8_t), uma tarefa que em Python puro resultaria em latência inaceitável para uma aplicação em tempo real.

Binário Específico: O arquivo mandelbrot.c é compilado como uma biblioteca dinâmica (.so ou .dll). Ele é um "motor sem chassi": possui a força bruta, mas não possui interface, dependendo do Python para receber os parâmetros de entrada.

6.2 Indispensabilidade do Python:

O código C foi desenvolvido estritamente como uma biblioteca dinâmica. Ele é incapaz de ser executado como um programa autônomo, dependendo do Python para o fornecimento de parâmetros (coordenadas, dimensões, iterações), que retorna um ponteiro para os pixels, e o controle de fluxo e exibição gráfica.

É o Python que executa a **Matemática da Câmera**, convertendo cliques no Canvas em coordenadas do plano complexo e aplicando **Zoom Logarítmico** antes de despachar os dados para o C. Está no método ajustar_zoom(self, evento, fator). É o Python quem calcula as novas coordenadas x_min, x_max, etc., baseadas no clique do mouse.

Garantia de Ciclo de Vida (Memory Safety): Como o C não possui *Garbage Collector*, o Python atua como o **Gestor de Memória**. Via blocos try/finally, o Python garante que a memória alocada no *Heap* do C seja obrigatoriamente liberada, em: **lib.liberar_dados_mandelbrot(ponteiro),** prevenindo *Memory Leaks* que travariam o sistema operacional.

Ponte de Dados (FFI): O Python utiliza a biblioteca ctypes para realizar a tradução de tipos de dados. Sem essa camada, o resultado binário do C seria apenas um bloco de memória bruto e ilegível para o usuário final.

Além de que a dependência do Python também vem de se não encontrar mandelbrot.dll/.so, o programa dá erro e não funciona. Ou seja, O Python **não funciona** sem a biblioteca C (dá erro se a DLL não existir), via ctypes, isso caracteriza um **Acoplamento Forte** , verificada no **main.py** através da configuração rigorosa de tipos **(argtypes e restype)**, em tempo de execução, o que é um conceito clássico de CLP (Conceitos de Linguagens de Programação).

- *O main.py gerencia o ciclo de vida, a interface gráfica (Tkinter) e a manipulação de imagens (Pillow). Sem o C, ele não possui o algoritmo de cálculo e não consegue renderizar nada.*
- *O main.py garante Robustez. Se o C falhar, o Python captura a exceção, se o Python for removido, o código C permanece inerte por ser uma biblioteca estritamente passiva.*

Categoria	**Responsabilidade / Método**	**Descrição Técnica (Implementação)**
Papel do Python	**Frontend e Controle**	Gerenciamento da Interface Gráfica (Tkinter), controle de Zoom e Panning, orquestração de Threads e conversão de bytes para imagem (Pillow).
Papel do C	**Backend e Performance**	Motor de cálculo intensivo (Mandelbrot), processamento de cores por Histograma e alocação dinâmica de memória bruta (Heap).
Método de Interface	**FFI via ctypes**	Carregamento dinâmico da biblioteca (.dll ou .so) em tempo de execução, permitindo que o Python chame funções nativas do C.
Intercâmbio de Dados	**Compartilhamento de Structs**	Mapeamento binário onde a struct no C é espelhada exatamente pela class Structure no Python, garantindo que ambas leiam os mesmos offsets de memória.
Performance	**Ponteiros de Memória**	O Python acessa o endereço de memória (pointer) alocado pelo C diretamente, evitando a cópia de milhões de pixels e garantindo alta velocidade.

Segurança	**Gestão de Ciclo de Vida**	O Python atua como gestor de memória, garantindo que o comando de liberação (free) seja enviado ao C para evitar vazamentos (*memory leaks*).

7. Conclusão

O projeto demonstra com sucesso a integração entre C e Python, aplicando cada linguagem onde ela é mais eficiente. O uso de **FFI (Foreign Function Interface)** via ctypes permitiu o compartilhamento direto de memória, resultando em uma aplicação gráfica fluida e de alta performance. Essa arquitetura demonstra o domínio de Interoperabilidade entre Linguagens, requisito central da disciplina. O trabalho destaca conceitos fundamentais de linguagens de programação, como gerenciamento de memória, tipos de dados e interoperabilidade.

8. Referências Bibliográficas

1. **CHERITAT, Arnaud.** *Mandelbrot set*. Wiki-draw, Institut de Mathématiques de Toulouse. Disponívelhttps://www.math.univ-toulouse.fr/~cheritat/wiki-draw/index.php/Mandelbrot_set.*(Esta fonte fundamenta a lógica geométrica e a representação visual do plano complexo utilizada no motor de cálculo em C.)*
2. ***PYTHON SOFTWARE FOUNDATION.*** *O Tutorial de Python. Documentação Oficial (pt-br). Disponível em: https://docs.python.org/pt-br/3/tutorial/index.html. (Referência para a arquitetura da linguagem, gerenciamento de tipos dinâmicos e a interface de interoperabilidade ctypes.)*
3. ***MANDELBROT.SITE.*** *The Mandelbrot Set: Mathematics and Algorithm. Disponível em: https://mandelbrot.site/..(Utilizada para a implementação do algoritmo de escape e as técnicas de otimização de renderização de fractais.)*
4. ***GARCIA, J. S.*** *Fractais de Mandelbrot não são apenas bonitos: eles ensinaram os matemáticos a modelar o mundo real. The Conversation. Disponível em: https://theconversation.com/fractais-de-mandelbrot-nao-sao-apenas-bonitos-eles-ensinaram-os-matematicos-a-modelar-o-mundo-real-244906. (Fonte utilizada para contextualizar a aplicação dos fractais em modelagem de fenômenos complexos e sistemas dinâmicos).*
5. ***MATHIGON.*** *Conjunto de Mandelbrot. Fractals Course. Disponível em: https://pt.mathigon.org/course/fractals/mandelbrot. (Utilizada para a didática visual da órbita dos números complexos e critério de divergência do algoritmo).*