Trabalho Prático I - Análise de um Sistema de Renderização 2D com Oclusão

Guilherme Asafe de Siqueira Meireles
Matrícula: 2023075135
g.asafe.ufmg@gmail.com

Outubro de 2025

Sumário

1 Introdução

Este trabalho tem como objetivo a aplicação dos conceitos da disciplina de Estrutura de Dados. O problema trata de objetos que podem ser definidos dentro de um certo intervalo, onde podemos imaginar a interseção entre eles. Cada objeto apresenta um subintervalo fixo e podemos movê-lo, modificando assim as suas coordenadas x e y.

O objetivo do trabalho é avaliar a capacidade do aluno de aplicar os conceitos ensinados em sala de aula, como a ordenação de uma estrutura de dados, operações como inserção ou remoção, e a análise de complexidade do algoritmo. Espera-se do discente uma boa implementação para a resolução do problema, buscando um algoritmo estável e eficiente.

A implementação tem como base armazenar os objetos da entrada numa `lista encadeada` e, para resolver a oclusão, ordenamos os objetos desta lista com o algoritmo `Mergesort`.

Este documento está organizado da seguinte forma: a Secção 2 detalha a implementação do sistema; a Secção 3 apresenta as instruções de compilação e execução; a Secção 4 contém a análise de complexidade formal; a Secção 5 discute as estratégias de robustez; a Secção 6 mostra a análise experimental; e, por fim, a Secção 7 conclui o trabalho.

2 Implementação

O código é organizado em ficheiros que são responsáveis por armazenar os objetos de entrada, mover estes objetos e processar a cena para determinar a visibilidade.

2.1 Estruturas Implementadas

O código tem como estrutura de dados principal uma `lista encadeada`, que permite inserir, buscar e, futuramente, deletar objetos de forma dinâmica.

2.2 Funcionamento do programa e principais funções

- **Função `main`:** Atua como o orquestrador central. Lê os comandos da entrada, analisa-os e delega as tarefas para as classes e funções apropriadas.

- **Função `listas::insert`:** Responsável por adicionar um novo objeto à cena. A inserção no final da lista encadeada é uma operação de tempo constante, $\mathcal{O}(1)$.

- **Função `movimento::movimentar`:** Atualiza a posição de um objeto. Invoca a função `listas::busca` (custo $O(n)$) para encontrar o objeto e depois altera as suas coordenadas.

- **Função `Cena::processaCena`:** Esta é a função central do algoritmo de visibilidade. Após a ordenação, esta função percorre a lista de objetos e, para cada um, determina quais dos seus segmentos horizontais estão visíveis. Para tal, mantém uma lista auxiliar de "intervalos ocupados"(‘IntervaloOcupado‘), que representa o espaço no eixo X já coberto por objetos mais próximos.

- **Função `Cena::gravaCena`:** Após o processamento da visibilidade, esta função ordena a lista de segmentos visíveis gerados com base no ID do objeto, utilizando novamente o Merge Sort, para garantir que a saída final esteja agrupada corretamente.

- **Função `SaidaFinal::imprime`:** É chamada uma única vez no final da execução para percorrer a lista de todos os segmentos acumulados e imprimi-los na saída padrão com a formatação exata.

2.3 Ambiente de Desenvolvimento

O programa foi implementado numa distribuição Linux (Ubuntu), e a linguagem de programação utilizada foi C++.

3 Instruções de compilação e execução

Nesta seção, são apresentados os passos necessários para compilar e executar o programa em um ambiente Linux.

- **Acesse o diretório raiz do projeto:** Abra um terminal e navegue até a pasta principal do trabalho prático, onde se encontra o ficheiro `makefile`. Supondo que o projeto está em `~/Documents/Data-Struct/`, o comando seria:

```
cd ~/Documents/Data-Struct/TP/
```

- **Compile o programa:** Com o terminal no diretório raiz do projeto, execute o comando `make`. Este comando irá invocar o compilador g++ com as flags especificadas, gerando o ficheiro executável.

  ```
  make all
  ```

  Ao final do processo, um ficheiro executável chamado `tp1.out` será criado dentro do diretório `bin/`.

- **Execute o programa:** Para executar o programa, utilize o seguinte comando, que usa um ficheiro de entrada e redireciona a saída para um ficheiro de resultados:

  ```
  ./bin/tp1.out < bin/inputs1.txt > output.txt
  ```

- **Verificação:** Com o comando acima, o programa irá ler os dados de `inputs1.txt` e gravar o resultado no ficheiro `output.txt`. Pode então comparar este ficheiro com a saída esperada para verificar a correção.

4 Análise de complexidade

A escolha do TAD facilita a análise, pois conhecemos o custo das operações e do algoritmo de ordenação utilizado.

4.1 Operações da Lista Encadeada

A inserção tem complexidade $\mathcal{O}(1)$. A operação `search`, usada para a classe `movimento`, tem complexidade $\Theta(n)$. A operação `delete` (se fosse implementada para remoção genérica) teria complexidade $\Theta(n)$.

4.2 Método Mergesort

Utilizamos o algoritmo Mergesort em listas encadeadas para ordenar os objetos e identificar a oclusão. Este algoritmo tem complexidade $\mathcal{O}(n \cdot \log n)$ para o melhor caso, pior caso e caso médio, logo podemos classificá-lo como $\Theta(n \cdot \log n)$. Isto garante que o tempo de ordenação é consistente, independentemente da ordem inicial dos dados.

4.3 Análise de Complexidade de Tempo da Cena

- **Afirmação:** A função `Cena::processaCena` tem uma complexidade de tempo quadrática, $\Theta(n^2)$.
- **Prova Formal:** A complexidade é dominada pelo laço principal que itera sobre os n objetos. Dentro deste laço, o custo das operações depende do tamanho da lista `canvas_ocupado`. No pior cenário, ao processar o i-ésimo objeto, esta lista terá $i - 1$ nós. O custo para processar o objeto i é, portanto, $O(i)$. Somando o custo para todos os n objetos:

$$C(n) = \sum_{i=1}^{n} O(i-1) = O\left(\frac{(n-1)n}{2}\right) = \Theta(n^2)$$

4.4 Conclusão da Análise de Tempo

A complexidade de tempo total de uma operação de cena ('C') é a soma das complexidades de cada etapa (aqui não ignoramos a complexidade das operações na lista):

$$\text{Ordenação } (\Theta(n \cdot \log n)) + \text{Processamento da Cena } (\Theta(n^2)) = \Theta(n^2)$$

Logo, o nosso algoritmo tem complexidade de tempo $\Theta(n^2)$ por operação de cena.

4.5 Análise de Complexidade de Espaço

- **Afirmação:** A complexidade de espaço do programa é $O(n + s)$, onde s é o número de segmentos de saída, podendo atingir $O(n^2)$ no pior caso.

- **Prova Formal:** O espaço é a soma do armazenamento dos n objetos ($\Theta(n)$), da memória auxiliar usada nos cálculos ($O(n)$) e do armazenamento dos s segmentos de saída ($O(s)$). Como o número de segmentos, s, pode crescer quadraticamente com n em cenários de alta fragmentação, este termo domina.

Complexidade Total de Espaço: $O(n^2)$

5 Estratégias de Robustez

5.1 Validação de Entradas

O programa valida os dados de entrada no ciclo principal em `main.cpp`. Se uma linha estiver mal formatada ou com dados em falta, a operação é ignorada, evitando que o programa prossiga com dados corrompidos.

5.2 Tratamento de Erros com Exceções

O programa utiliza exceções para lidar com erros lógicos. Por exemplo, se um comando 'M' tenta mover um objeto que não existe, a função `listas::busca` lança uma `std::runtime_error`, que é capturada pela função `movimento::movimentar`, informando o erro de forma controlada.

5.3 Programação Defensiva

A implementação inclui verificações internas para garantir a consistência. Por exemplo, a função `Cena::insereSegmento` verifica se um segmento é válido (`inicio < fim`) antes de alocar memória. Adicionalmente, a chamada a `minha_cena.resetSegmentos()` previne erros de dupla libertação de memória.

5.4 Verificação de Fugas de Memória (Memory Leaks)

Foi utilizada a ferramenta **Valgrind** para analisar a gestão de memória. Os testes confirmaram que, no final da execução, não existem fugas de memória, assegurando que todos os recursos alocados são corretamente devolvidos ao sistema.

6 Análise Experimental

Foi realizada uma análise experimental para avaliar o impacto da frequência de ordenação no desempenho do programa. Para isso, o código foi modificado para reordenar a lista de objetos apenas quando o número de modificações atingisse um **Limiar (L)** pré-definido.

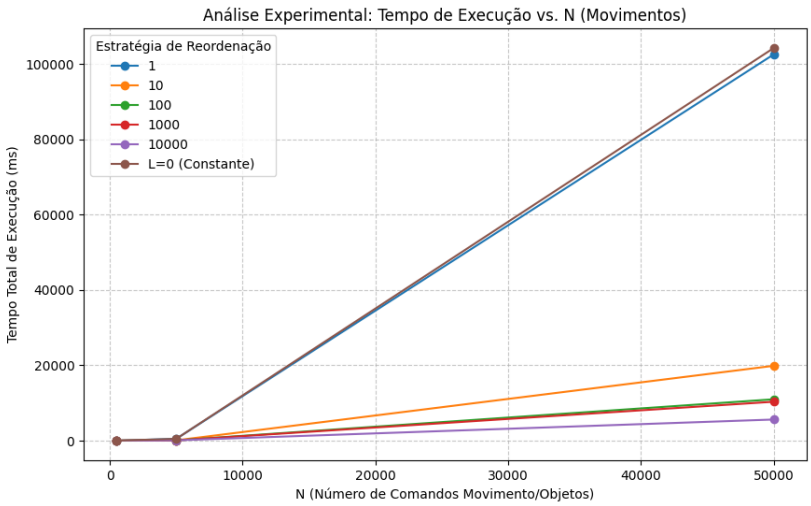

Figura 1: Tempo de Execução vs. Número de Movimentos para diferentes Limiares de Reordenação (L).

6.1 Conclusão Experimental

Os resultados, apresentados na Figura 1, demonstram que:

- **Reordenar constantemente (L=0 e L=1) é ineficiente**, sendo a causa do maior tempo de execução.
- **Adotar uma estratégia "preguiçosa"(L $\geq$ 10) melhora drasticamente o desempenho**, reduzindo o tempo de execução em até 80% nos testes com 50.000 movimentos.
- Os ganhos de desempenho tornam-se marginais para valores de L muito altos.

Conclui-se, portanto, que uma estratégia de **reordenação sob demanda é vastamente superior**.

7 Conclusão

Este trabalho lidou com o problema da visibilidade 2D com oclusão, utilizando um sistema em C++ com listas encadeadas e o algoritmo Merge Sort.

A principal contribuição foi a análise experimental, que validou a superioridade de uma estratégia de reordenação sob demanda, melhorando o desempenho em até 80% em comparação com uma abordagem de reordenação constante.

Por meio da resolução deste trabalho, foi possível praticar os conceitos de estruturas de dados, algoritmos de ordenação e gestão de memória em C++. Os principais desafios foram a correta implementação da complexa lógica de oclusão e a prevenção de erros de gestão de memória dinâmica.

8 Referências

Marcio Costa Santos. (2025). Slides virtuais da disciplina de estruturas de dados. Departamento de Ciência da Computação, UFMG.

9 Referências Bibliográficas

Referências

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, 3rd edition, 2009.