Trabalho Prático 1 - Identificação de Objetos Oclusos

Téo Mendes Araújo - Matrícula: 2024015454

10 de Outubro de 2025

Contents

1 Introdução **3**

2 Método **3**
2.1 Objeto . . . 3
2.2 Cena . . . 3
2.3 Leitura e Armazenamento . . . 3
2.4 Geração da Cena (`gerarCena`) . . . 4
2.5 vector<T> . . . 5

3 Análise de Complexidade **5**
3.1 gerarCena() . . . 5
3.2 vector . . . 5

4 Estratégias de Robustez **6**
4.1 Validação de Entradas . . . 6
4.2 Tratamento de Erros . . . 6
4.3 Programação Defensiva . . . 6
4.4 Gerenciamento de Memória . . . 6

5 Análise Experimental de Desempenho **7**
5.1 Resultados . . . 7

6 Conclusão **7**

7 Bibliografia **8**

1 Introdução

O problema consiste em objetos espalhados no espaço paralelamente, para então serem projetados em relação a um plano unidimensional. O desafio central é determinar quais partes desses objetos permanecem visíveis após a oclusão causada por objetos que estão à frente. Embora não haja uma oclusão em perspectiva tridimensional, o problema se resume a um algoritmo que processa corretamente a profundidade para renderizar apenas os segmentos visíveis.

Portanto, para a resolução desse problema, o algoritmo essencialmente cria a cena de acordo com um instante de tempo específico, ordena os objetos pela sua profundidade (coordenada Y), processa a oclusão para remover ou redimensionar os objetos encobertos e, finalmente, reorganiza os objetos visíveis de acordo com seu identificador para a saída final.

2 Método

2.1 Objeto

A classe `objeto` encapsula os atributos de uma instância de um objeto em um determinado momento no tempo. Seus principais atributos são:

- `id`: Índice do objeto.
- `tempo`: Marca de tempo associada àquela posição.
- `x`, `y`: Coordenadas do centro do objeto.
- `largura`: A largura do segmento de reta.

A classe provê um construtor para inicializar os atributos e métodos de acesso *getters* (`getId()`, `getX()`, `getY()`, etc.) e *setters* (`setX()`, `setY()`, `setLargura()`) que respeitam o Paradigma de Orientação a Objetos (POO).

2.2 Cena

A classe `Cena` é o TAD responsável por gerenciar a "lista" de todos os objetos e seus movimentos, além de conter a lógica para a geração das cenas visíveis. Seus papéis são:

- **Gerenciar Dados:** Armazena todos os objetos e seus respectivos históricos de movimento através do atributo `objetos` (`vector<vector<objeto»`).
- **Processar Entradas:** Oferece métodos que interagem com a criação de objetos (`addObject()`) e a geração de movimentos (`addMovement()`).
- **Gerar Cenas:** Implementa a lógica central de geração de cenas, desde a reorganização dos vetores até a verificação de oclusão.
- **Armazenar Resultado:** Armazena o resultado da última cena gerada em um vetor `cena`.

2.3 Leitura e Armazenamento

O programa principal (`main.cpp`) realiza a leitura sequencial da entrada padrão (`std::cin`). Com base no caractere inicial de cada linha ('O', 'M' ou 'C'), a ação correspondente é executada. Caso seja 'O', um objeto é adicionado; se for 'M', um movimento é registrado; e se for 'C', uma cena é gerada e impressa na saída padrão.

2.4 Geração da Cena (`gerarCena`)

Esse é o principal método, sendo responsável pela lógica da geração de uma cena. É um aglomerado de outros métodos, onde os utiliza a fim de gerar a cena, desde evocar métodos que vão verificar a oclusão e reorganização do vetor afim de obter a cena, por isso é composto por uma sequência de etapas pré-estabelecidas:

1. **Filtragem por Tempo (`cenaSortTime`):** Inicialmente, o método constrói o (`cena`) contendo o estado mais recente de cada objeto, considerando apenas os registros de tempo anteriores ou iguais ao instante solicitado. A lógica itera sobre o histórico de cada objeto para encontrar o movimento válido mais recente.

2. **Ordenação por Profundidade (`mergeSort`):** o vetor cena é então ordenado com base na coordenada `y` dos objetos. Essa ordenação por profundidade posiciona os objetos com menor valor de `y` (mais próximos do observador) nas primeiras posições. A escolha do algoritmo **Merge Sort** justifica-se por sua complexidade de O(n log n).

3. **Cálculo de Oclusão (`sortOverlap`):** Esta etapa, a mais complexa do processo, determina as porções visíveis de cada objeto. O algoritmo percorre o vetor ordenado e, para cada objeto, compara-o com todos os subsequentes (que estão "atrás"). A lógica de sobreposição no eixo X define o tratamento da oclusão:

    - Se um objeto traseiro for totalmente encoberto, ele é descartado da cena.
    - Se a oclusão for parcial, os atributos `x` e `largura` do objeto traseiro são recalculados para representar apenas o segmento visível.
    - Caso um objeto frontal divida um traseiro em duas partes, um novo `objeto` é instanciado e inserido na cena para representar o segundo segmento visível, preservando o `id` original.

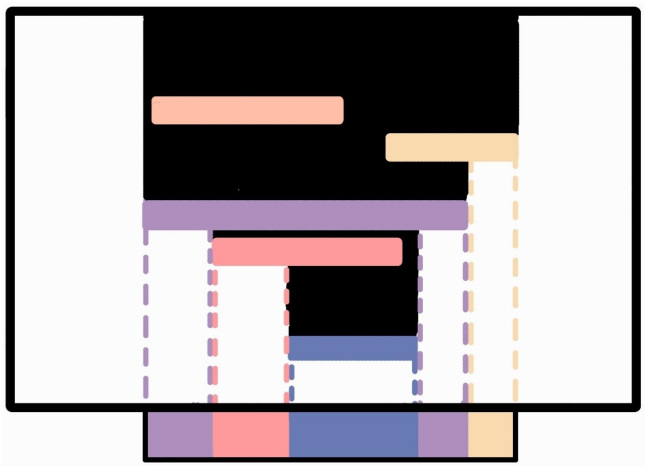

Figure 1: Representação do Algoritmo de oclusão

4. **Ordenação Final (`mergeSortById`):** Concluído o tratamento de oclusão, a cena contém um conjunto de segmentos de reta, possivelmente com múltiplos segmentos para um mesmo `id` original. Para cumprir o requisito de formato da saída, o vetor é finalmente reordenado em ordem crescente pelo atributo `id`.

2.5 vector<T>

Para gerir os dados, foi implementado um TAD que emula o comportamento do contêiner `std::vector`. Esta estrutura funciona como um arranjo unidimensional, porém com capacidade de redimensionamento dinâmico.

Para expandir dinamicamente a capacidade do vetor, um novo bloco de memória de tamanho (`tamanho_atual + 1`) é alocado sempre que um elemento é inserido. Todos os elementos do bloco original são copiados para o novo espaço, e a memória antiga é liberada.

Para replicar as características desse contêiner, foram implementados os seus métodos principais: `insert`, `remove`, `push_back` e `pop_back`. Adicionalmente, foram desenvolvidas as funções `push_front` e `pop_front` para oferecer maior flexibilidade.

3 Análise de Complexidade

A eficiência de um algoritmo é medida pela sua complexidade de tempo e espaço, que descrevem como os recursos computacionais (tempo de processamento e memória) escalam com o aumento do tamanho da entrada. Nesta seção, é apresentada uma análise detalhada do pior caso para os principais componentes do sistema: o método orquestrador `gerarCena()` e as operações fundamentais do TAD `vector` customizado.

3.1 gerarCena()

A complexidade de tempo e espaço do método principal, `gerarCena()`, é determinada pela soma das complexidades de suas etapas constituintes. Cada sub-rotina — filtragem, ordenação e cálculo de oclusão — contribui para o custo computacional total. A tabela a seguir detalha a análise para cada componente no pior caso, permitindo a identificação dos principais gargalos de desempenho do algoritmo.

Como os resultados demonstram, a análise revela que a etapa de `sortOverlap` é o fator dominante para o tempo de execução, com sua complexidade quadrática $O(n^2)$, enquanto a filtragem por tempo introduz uma dependência da faixa de tempo M. Em contrapartida, o requisito de espaço é ditado pela necessidade de um buffer auxiliar no `mergeSort`, resultando em uma complexidade espacial linear de $O(n)$.

Table 1: Análise de Complexidade de `gerarCena()`

Componente	**Análise de Tempo (Pior Caso)**	**Análise de Espaço (Pior Caso)**
`sortOverlap()`	$2n^2 + 2n \implies O(n^2)$	$6 \implies O(1)$
`cenaSortTime()`	$O(n \cdot M)$	$3 \implies O(1)$
$2 \times$ `mergeSort()`	$O(n \log n)$	$n1 + n2 \implies O(n)$
Gasto Total	$\mathbf{O(n^2 + nM)}$	$\mathbf{O(n)}$

3.2 vector

O desempenho das operações de alto nível, como o cálculo de oclusão que modifica a cena, depende fundamentalmente da eficiência da estrutura de dados subjacente. Uma vez que um TAD `vector` foi implementado para este trabalho, é crucial analisar a complexidade de suas operações de modificação mais custosas: `insert()` e `remove()`.

A análise, apresentada na tabela abaixo, mostra que ambos os métodos exibem uma complexidade de tempo linear, $O(n)$, pois, no pior caso, exigem o deslocamento de todos os elementos subsequentes à posição da modificação. A complexidade de espaço, também linear, reflete a estratégia de realocação de memória, na qual um novo bloco de memória é alocado e todos os elementos são copiados, consumindo temporariamente até o dobro do espaço original.

Table 2: Análise de Complexidade (Métodos de `vector<T>`)

Método	**Análise de Tempo (Pior Caso)**	**Análise de Espaço (Pior Caso)**
`insert()`	$n+1 \implies O(n)$	$2n+1 \implies O(n)$
`remove()`	$n-1 \implies O(n)$	$2n-1 \implies O(n)$

4 Estratégias de Robustez

Para garantir a confiabilidade e a resiliência do programa, diversas estratégias foram adotadas.

4.1 Validação de Entradas

O loop principal em `main.cpp` utiliza uma estrutura `switch` para processar apenas os caracteres de comando esperados ('O', 'M', 'C'). Quaisquer outros caracteres na entrada são ignorados, impedindo que o programa entre em um estado indefinido ou termine inesperadamente devido a uma entrada malformada.

4.2 Tratamento de Erros

A classe `vector` customizada implementa verificação de limites (*bounds checking*). O operador de acesso `[]` e os métodos de manipulação como `insert()` e `remove()` lançam uma exceção do tipo `std::out_of_range` se um índice ou posição inválida for fornecida. Isso previne erros de segmentação e corrupção de memória que poderiam ocorrer com acessos fora dos limites do array.

4.3 Programação Defensiva

Antes da execução de pontos criticos do algoritmo, há verificação se a entrada é válida, além disso há uso de macros para melhorar legibilidade. em todas as funções em que não havera a alterção efetiva há a atribuição de 'const', além de não copiar os inputs, usando sua referência.

4.4 Gerenciamento de Memória

A classe `vector` gerencia a memória automaticamente. O destrutor (`~vector()`) e o método `clear()` são responsáveis por liberar a memória alocada com `delete[]`, prevenindo vazamentos de memória (*memory leaks*) quando um vetor sai de escopo ou é limpo.

5 Análise Experimental de Desempenho

Para avaliar o desempenho do algoritmo, foi conduzida uma análise experimental focada no tempo de execução da função `gerarCena` sob diferentes cargas de trabalho. Os testes variaram o número de objetos, a quantidade de movimentos e a complexidade da cena.

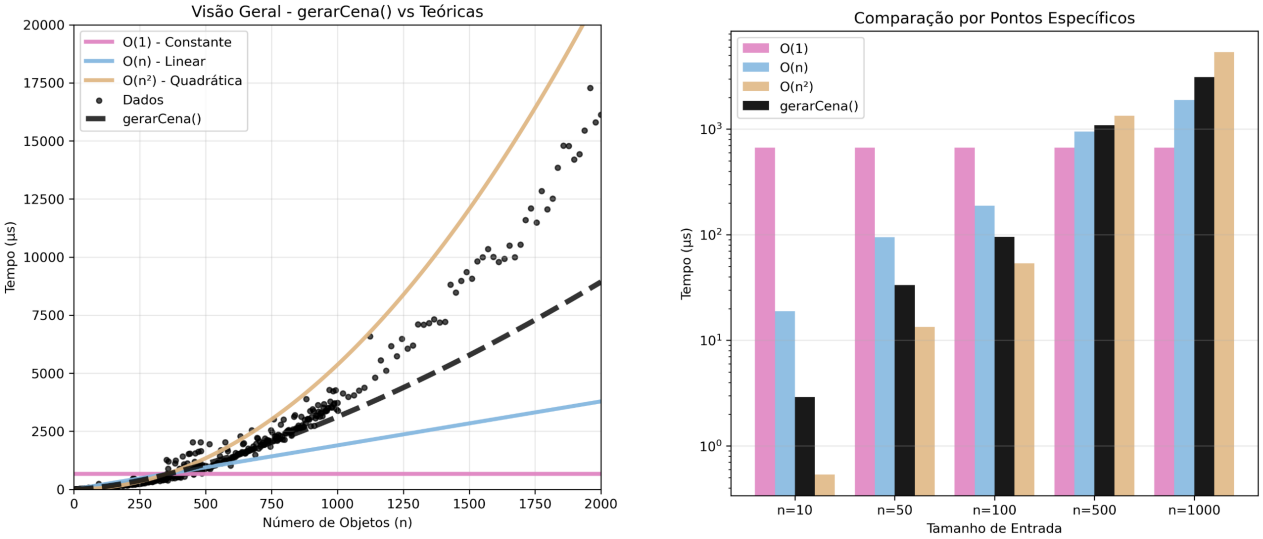

Figure 2: Gráficos de análise de desempenho da geração de cena.

5.1 Resultados

Os resultados experimentais, visualizados confirmam a análise teórica de complexidade. O gráfico da esquerda demonstra que o tempo de execução total cresce de forma quadrática com o aumento do número de objetos na cena, o que é consistente com a complexidade $O(n^2)$ do algoritmo `sortOverlap`.

O gráfico de colunas à direita (Figura **??**) detalha a distribuição do tempo de execução discretamente entre as principais etapas do método `gerarCena`. Fica evidente que a etapa de cálculo de oclusão (`sortOverlap`) é o principal gargalo de desempenho, consumindo a maior parte do tempo de processamento. As etapas de ordenação (`mergeSort`) e filtragem por tempo (`cenaSortTime`) têm um impacto consideravelmente menor, alinhando-se às suas complexidades de $O(n \log n)$ e $O(N \cdot M)$, respectivamente.

6 Conclusão

Este trabalho prático abordou o problema de identificação e renderização de objetos oclusos em um ambiente unidimensional. A solução foi desenvolvida utilizando TAD's que modularizaram a representação dos elementos e a lógica de processamento da cena. Conforme as restrições do projeto, foi necessário aprender à implementar um vetor dinâmico para evitar o uso de contêineres da biblioteca padrão do C++.

A análise de complexidade e os testes experimentais demonstraram que o algoritmo de cálculo de oclusão, com sua complexidade quadrática, é o fator dominante no desempenho do sistema. Este projeto proporcionou prática na implementação de estruturas de dados customizadas, no desenvolvimento de algoritmos para problemas de geometria computacional e na análise de desempenho, reforçando a importância fundamental das escolhas algorítmicas para a escalabilidade de uma solução.

7 Bibliografia

References

[1] cplusplus.com. *std::vector*. Disponível em: https://cplusplus.com/reference/vector/vector/. Acesso em: 10 de out. de 2025.

[2] Anisio, L., Wagner, W., & Washington, W. (2025). *DCC205/DCC221 Estruturas de Dados - TP1 v1.0*. UFMG.