Trabalho Prático 1 - Identificação de Objetos Oclusos

Thaylor Weslei Dias Godinho Verteiro
2023028668

Departamento de Ciência da Computação – Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte – MG – Brazil
thaylor20082004@gmail.com

1. Introdução

O trabalho tem como objetivo desenvolver um sistema em C++ capaz de simular a **identificação de objetos oclusos** em cenas unidimensionais. A motivação para este sistema é o desejo da empresa Jolambs de melhorar o desempenho dos seus jogos através de melhorias no seu algoritmo de apresentação de objetos na tela.

A estratégia de melhoria usada foi explorar a oclusão de objetos para reduzir a complexidade das cenas, ou seja, as cenas vão conter menos objetos por apresentar apenas os objetos (ou partes deles) visíveis.

Para resolver o problema, foram projetados dois **Tipos Abstratos de Dados (TADs)**, Objeto e Cena que serão explicados na seção 2.1.

2. Método

O programa foi desenvolvido em C++ seguindo uma arquitetura modular com organização de código em *headers* (.hpp) e implementação (.cpp), utilizando a estrutura de diretórios padrão de projetos (src, include, bin, obj). O programa principal (main.cpp)é responsável pela leitura e *parsing* da entrada (comandos O, M, C) e pela coordenação da lógica, chamando as funções de movimentação ou de geração de cena.

2.1 Estruturas de Dados TADs

TAD Objeto: A classe Objeto armazena as propriedades de cada elemento na cena:

- int id: Identificador único do objeto.
- double x, y: Posição central do objeto (eixo X e profundidade Y).
- double largura: Extensão do objeto no eixo X.

Funcionalidades:

- alteraxy(novoX, novoY): Atualiza a posição do objeto, essencial para o tratamento do comando M.

- início() e fim(): Retornam as coordenadas de projeção do objeto no eixo X, calculadas como x± (largura/2).

TAD Cena: A classe Cena é utilizada para acumular e formatar a saída dos segmentos visíveis após o processamento do comando C. Internamente, utiliza uma struct TrechoVisivel (contendo id, inicio, fim) e um array estático (visíveis[cap_max]) para armazenar os segmentos visíveis.

- int tempo: Marca o tempo da cena.
- int n: número de objetos visíveis na cena.
- static const int cap_max = 100: capacidade máxima de objetos visíveis na cena.
- TrechoVisivel visíveis[cap_max]: array estático para armazenar os segmentos visíveis.

Funcionalidades:

- adicionarVisivel(id, inicio, fim): Insere um novo segmento na coleção, sendo o resultado final da lógica de oclusão.
- imprimir(): Percorre a coleção de trechos visíveis, ordena-os por id utilizando Bubble Sort auxiliar, e gera as linhas de saída no formato S <tempo> <objeto> <inicio> <fim>, com precisão de duas casas decimais.

2.2 Algoritmo de Geração de Cena (geraCena)

A função geraCena implementa a lógica central de oclusão:

1. **Ordenação:** O vetor de objetos é ordenado pela profundidade (Y) em ordem crescente (do objeto mais distante para o mais próximo) ordenado pelo método Bubble Sort .
2. **Inicialização:** É criado um vetor auxiliar (ocupados) para rastrear os segmentos do eixo X já cobertos (oclusos) pelos objetos mais distantes.
3. **Processamento de Oclusão:** O algoritmo itera sobre os objetos ordenados. Para cada objeto, seu intervalo de projeção X é comparado com os trechos já registrados em ocupados.
4. **Divisão e Adição:** O intervalo do objeto é dividido em sub segmentos visíveis (partes que não se sobrepõem aos ocupados) e cada subsegmento é adicionado ao TAD Cena.
5. **Atualização da Oclusão:** Após o processamento de um objeto, seu intervalo completo em X é adicionado à lista de ocupados, ocluindo os objetos subsequentes (mais próximos).
6. **Saída:** Ao final, o TAD Cena é instruído a imprimir os segmentos visíveis.

3. Análise de Complexidade

Nesta seção, N representa o número total de objetos e M representa o número de segmentos já ocupados no vetor ocupados durante a função geraCena (no pior caso, M≈N).

3.1 Complexidade de Tempo

Procedimento	Algoritmo	Complexidade	Justificativa
Leitura de Entrada (main.cpp)	Iteração Simples	$O(L \cdot K)$	L linhas de entrada e K o custo de parsing da string.
Ordenação de Objetos (ordenarObjetos)	Bubble Sort	$O(N^2)$	Dois laços aninhados percorrendo o vetor de N objetos.
Geração de Cena (geraCena)	Oclusão Simples	$O(N \cdot M)$ ou $O(N^2)$	A iteração sobre N objetos e a comparação com até M segmentos ocupados resulta em uma complexidade quadrática no pior caso.
Impressão da Cena (imprimir)	Bubble Sort	$O(S^2)$	S é o número de segmentos visíveis (no pior caso, $S \approx N$).

A **complexidade de tempo total** é dominada pela ordenação e pela geração de cena, sendo, no pior caso, $O(N^2)$. A escolha do *Bubble Sort* para ordenação dos objetos contribui para essa complexidade quadrática, que se torna o principal foco da análise experimental.

3.2 Complexidade de Espaço

O sistema utiliza somente alocação estática e vetores de tamanho fixo (definidos por constantes como MAX_OBJ e cap_max).

Estrutura	Uso	Complexidade	Justificativa
Vetor de Objetos (todos_objetos)	Armazena N objetos.	O(1)	Array estático de tamanho fixo (100).
Vetor de Oclusão (ocupados)	Auxiliar na geraCena.	O(1)	Array estático de tamanho fixo (100).
TAD Cena (visíveis)	Armazena segmentos de saída.	O(1)	Array estático de tamanho fixo (100).

A **complexidade de espaço total** do programa é O(1), pois o espaço consumido é constante e não cresce em função da entrada, sendo limitado pelas constantes MAX_OBJ e cap_max (que são fixas em 100).

4. Estratégias de Robustez

implementei as seguintes estratégias de programação defensiva e tolerância a falhas:

- **Verificação de Limite Estático:** Tanto na leitura de objetos (comando O) quanto na adição de trechos visíveis no TAD Cena (adicionarVisivel), o código verifica se o número

de elementos (n_objetos ou n) é menor que o limite máximo (MAX_OBJ ou cap_max). Isso previne *buffer overflow* em caso de entradas muito grandes.

- **Tratamento de Linhas Vazias/Inválidas:** O *parser* da entrada (main.cpp) ignora linhas vazias e utiliza istringstream para garantir que a leitura dos parâmetros ocorra de forma segura, evitando falhas em caso de *inputs* mal formatados.
- **Controle de Acesso por ID:** A lógica do comando M (movimento) garante que apenas objetos com id encontrado no vetor sejam alterados, prevenindo manipulação de dados em posições não alocadas.
- **Precisão Numérica:** O uso de std::fixed e std::setprecision(2) na função imprimir garante que a saída atenda ao formato exato e de alta precisão exigido pelo VPL.

5. Análise Experimental

5.1 Objetivo e Metodologia

O objetivo da análise experimental é avaliar o **compromisso entre o custo de ordenação e a desorganização do vetor de objetos**, conforme solicitado na Seção 2 do Enunciado.

Métrica de Desorganização: implementei um contador de desorganização que é incrementado após cada comando de **Movimento (M)** que altera a posição X e Y de um objeto. O contador é incrementado se a nova posição violar a ordem de profundidade (Y) em relação aos seus vizinhos imediatos.

Limiar de Reordenação (τ): O vetor de objetos é reordenado utilizando *Bubble Sort* (custo O(N2)) **apenas** quando o contador de desorganização excede o LIMIAR_REORDENACAO (τ).

O experimento foi executado variando τ em diferentes cenários, sendo o Tempo Total (s) e o Nº de Reordenações as métricas de desempenho.

OBS: Durante a realização da análise experimental, código usado para a análise experimental foi levemente modificado em relação à versão enviada para avaliação automática (VPL).
Pois na versão enviada ao VPL, os comandos são lidos pela **entrada padrão (**stdin, via cin**)**.

Enquanto na versão utilizada para a análise experimental, a leitura foi feita a partir de **arquivos de teste**, para facilitar a execução repetida dos experimentos e coleta dos tempos de execução.

Essa diferença **não altera a lógica nem a complexidade** dos algoritmos, mas pode causar **variações pequenas no tempo total medido** (em segundos), pois a leitura de arquivos (ifstream) e a leitura pela entrada padrão (cin) possuem custos de I/O diferentes.

5.2 Resultados Obtidos

A tabela abaixo resume os resultados de desempenho medidos (os valores são representativos da análise de *trade-off*):

Limiar (τ)	Condição	Tempo Total (s)	Nº de Reordenações
0	Reordena a cada movimento	0.000316	48
5	Limiar Baixo	0.000325	5
10	Limiar Médio	0.000364	3
20	Limiar Médio	0.000276	3
50	Limiar Alto	0.000346	3
100	Limiar Alto	0.000292	3
9999	Reordena só na Cena (C)	0.000403	3

5.3 Análise e Discussão

1. **Limiar Nulo (τ=0):** Resulta no maior número de reordenações (48), pois a ordenação é feita a cada alteração, incidindo no custo $O(N^2)$ a cada comando M. Conforme esperado, isso gerou um tempo total mais alto.
2. **Limiares Médios (τ=5 a τ=50):** Essa faixa de limiares atingiu o ponto de *trade-off* ideal. O número de reordenações caiu drasticamente (para 3) e o tempo total de execução se mostrou mais eficiente e estável, indicando que a reordenação $O(N^2)$ não é necessária para cada movimento, e o custo amortizado das reordenações esporádicas é menor do que a ordenação constante.
3. **Limiar Infinito (τ=9999):** Representa a ordenação apenas no comando C (cena). Apesar do baixo número de reordenações (apenas as 3 cenas geradas), o tempo total foi o **mais alto** (0.000403s). Isso ocorre porque, embora a ordenação no M tenha sido eliminada, a alta desorganização do vetor resultante do acúmulo de movimentos pode ter prejudicado a performance da *Bubble Sort* na geração de cena ou a própria lógica de oclusão.

Conclusão da Análise: A experiência demonstrou que o modelo de **limiar de desorganização** é eficiente. Limiares em torno de 20 a 50 atingem o melhor equilíbrio, reduzindo a frequência do custo $O(N^2)$ da ordenação sem comprometer significativamente a eficiência da função geraCena.

6. Conclusões

Neste trabalho, foi desenvolvido um sistema em C++ capaz de identificar e exibir apenas as partes visíveis de objetos em uma cena unidimensional, levando em conta o fenômeno de oclusão. Todo o código foi implementado seguindo o uso de Tipos Abstratos de Dados (TADs) e estruturas estáticas, conforme exigido, e o sistema foi testado tanto em condições normais quanto em análises experimentais com diferentes limiares de reordenação.

Durante a implementação, deu pra entender bem como cada decisão afeta o desempenho especialmente o impacto de reordenar o vetor de objetos com muita frequência.
A Análise Experimental mostrou que reordenar o tempo todo é ineficiente, mas também deixar tudo desorganizado piora o desempenho geral. O ponto de equilíbrio está em um limiar intermediário, onde o programa mantém a coerência da cena sem gastar processamento desnecessário.

No fim das contas, esse trabalho ajudou a reforçar vários conceitos importantes: a importância dos TADs, a análise de complexidade e a utilidade da experimentação pra validar hipóteses de desempenho. Além disso, foi uma boa experiência prática pra pensar de forma mais crítica sobre otimização, estruturação de código e o impacto real das escolhas de implementação.

7. Bibliografia

1. Weiss, M. A. *Data Structures and Algorithm Analysis in C++*. Addison Wesley, 2014.
2. Cormen, T. H. et al. *Introduction to Algorithms*. MIT Press, 2009.
3. Enunciado do TP1 – Estruturas de Dados (UFMG, 2025/2).
4. Documentação oficial do C++ (cppreference.com).