Trabalho Prático 1: Identificação de Objetos Oclusos

Marney Santos de Melo
Matrícula: 2024106034

Outubro de 2025

Sumário

1 Introdução

Na área da computação, em especial na computação gráfica para desenvolvimento de jogos, um problema recorrente é a complexidade de renderizar cenas de maneira eficiente. Uma estratégia muito utilizada para otimizar o desempenho é o descarte de oclusão (occlusion culling), uma estratégia onde a ideia principal consiste em processar e renderizar apenas objetos, ou parte de objetos, que estejam visíveis a partir de um 'view point', ou seja, renderizar apenas os objetos que estejam no campo de visão do observador e não estejam integralmente/parcialmente ocultos por outros objetos.

O trabalho proposto aborda uma versão simplificada deste problema, focando em um cenário unidimensional onde os objetos são representados por retas paralelas a um eixo. O objetivo é desenvolver um programa em C ou C++ que seja capaz de receber uma série de entradas que criam ou movimentam objetos e geram cenas em um determinado momento de tempo. A cada cena gerada, o algoritmo deve indentificar e imprimir quais partes dos objetos (retas) estão efetivamente visíveis, considerando a oclusão por outros objetos.

A solução pensada envolve a implementação de Tipos Abstratos de Dados (TADs) para representar os objetos e a cena final. Como tópico principal, o trabalho investiga o **compromisso** de desempenho entre duas estratégias de ordenação dos objetos, que é um passo fundamental do algoritmo para ***Identificação de Objetos Oclusos***: manter a coleção de objetos sempre ordenada a cada movimentação ou ordená-la apenas quando a geração da cena for solicitada.

2 Método

O sistema foi desenvolvido na linguagem C++ e compilado com a flag para o padrão C++11 através do g++, utilizando o Sistema Operacional Linux Ubuntu 22.04 LTS, seguindo as restrições presentes no enunciado. A solução foi estruturada em torno de Tipos Abstratos de Dados (TADs) para encapsular as principais entidades do problema e o algoritmo de geração de cenas.

2.1 Tipos Abstratos de Dados (TADs)

Foram implementadas classes para representar os objetos, a cena final e um gerenciador para a coleção de objetos.

- **Classe `Objeto`:** Armazena os atributos de um objeto individual, como ID, coordenadas do centro (x, y), largura e as posições de início e fim no eixo X, calculadas a partir do centro e da largura. Possui um método `'AtualizaPosicao'` para modificar suas coordenadas e recalcular seus limites, além de possuir getters e um construtor.

- **Classe `GerenciadorDeObjetos`:** Gerencia um array estático de instâncias da classe `Objeto`. É responsável por processar os comandos de criação ('O') e movimento ('M'), adicionando novos objetos ou atualizando a posição de objetos existentes no array.

- **Classe `Cena`:** Gerencia um array estático de `SegmentoVisivel`, uma struct que armazena o ID, o início e o fim de uma parte visível de um objeto. Possui métodos

para adicionar novos segmentos visíveis e para imprimir o conteúdo final da cena no formato especificado no enunciado: `S <tempo> <objeto> <inicio> <fim>`.

2.2 Algoritmo de Geração de Cena

O algoritmo implementado na função `geraCena` segue a estrutura apresentada na *Figura 4: Algoritmo para construção de cenas*, presente no enunciado, e é executado sempre que um comando 'C' é recebido. A execução do algoritmo é dividido em duas fases principais.

2.2.1 Fase 1: Ordenação por Profundidade

A primeira ação da função é ordenar o array de todos os objetos com base em sua coordenada Y (profundidade), do menor para o maior. Isso organiza os objetos da frente para o fundo, algo indispensável para a proxima fase - Cálculo de Oclusão -, pois apenas objetos mais próximos (menor Y) podem ocultar objetos mais distantes. Para esta etapa, foi implementado um algoritmo *MergeSort*.

Nota: A escolha por algoritmo de ordenação estável foi feita visando a robustez do programa em casos de objetos com a mesma profundidade (coordenada Y). Esta decisão será discutida em mais detalhes na Seção 4: Estratégias de Robustez.

2.2.2 Fase 2: Cálculo de Oclusão

Para gerenciar as áreas já ocupadas na cena, o algoritmo utiliza uma estrutura auxiliar chamada *Intervalo*, que armazena um par de coordenadas de início e fim no eixo X.

Esses intervalos são armazenados em um array denominado *'horizonte'* (conhecido tambem como scanline em algortimos de varredura), que representa os trechos já preenchidos por objetos analisados. Em suma, o *horizonte* contém a união de todos os intervalos no eixo X que já foram ocupados.

Dessa forma, a regra de oclusão é direta: ao analisar um novo objeto, qualquer parte de seu intervalo que coincida com um dos intervalos presentes no horizonte é considerada oclusa (invisível) e, portanto, não é adicionada à cena final.

Após o cálculo das partes visíveis de um objeto, seu intervalo completo é adicionado ao horizonte, atualizando-o para que possa "esconder"os objetos subsequentes.

Resumidamente, o cálculo de oclusão é feito por meio da comparação entre os elementos do array de objetos ordenados e o array de objetos já analisados (horizonte). Se houver coincidência, a parte coincidente do novo objeto é automaticamente considerada oclusa, uma vez que os objetos já analisados têm sempre uma coordenada Y menor ou igual à do novo objeto a ser analisado.

Essa lógica pressupõe que, em casos de profundidades iguais, a ordem de aparecimento dos objetos define qual será visível e qual será ocluso, o que reforça a importância do uso de um algoritmo de ordenação estável, conforme mencionado anteriormente.

3 Análise de Complexidade

A análise de complexidade foca na função `geraCena`, que é o núcleo computacional do programa. Seja n o número total de objetos na cena.

3.1 Complexidade de Tempo

A complexidade de tempo total é a soma de suas duas fases de execução citadas ateriormente, a *Ordenação* e o *Cálculo de Oclusão*.

- **Ordenação:** A primeira etapa do algoritmo consiste em organizar todos os n objetos por profundidade. Para essa tarefa, foram implementados três algoritmos com características distintas para a análise experimental: `Merge Sort`, `Quick Sort` e `Insertion Sort`.

  - Os algoritmos **Merge Sort** e **Quick Sort** foram escolhidos por sua excelente complexidade de tempo média de $O(n \log n)$. É importante ressaltar que, enquanto o Merge Sort garante esse desempenho em todos os cenários, o Quick Sort, em seu pior caso (embora raro na prática), pode apresentar uma complexidade quadrática de $O(n^2)$, caso os pivôs sejam mal escolhidos (p. ex. o menor ou maior elemento do sub-vetor).
  - O **Insertion Sort** foi escolhido por uma característica interessante: apesar de ter complexidade quadrática ($O(n^2)$) no caso médio e pior caso, ele alcança um desempenho linear ($O(n)$) quando o vetor já está quase ordenado. Essa vantagem no melhor caso o torna ideal para a estratégia experimental de reordenar o vetor a cada movimento, já que apenas um elemento do vetor é movido a cada movimento.

- **Loop de Oclusão:** Após a ordenação, é iniciada a fase de cálculo da visibilidade, a fase mais custosa do programa. O algoritmo percorre cada um dos n objetos em um loop principal. A complexidade surge porque, para cada objeto, é necessário compará-lo com o "horizonte", o array que contém todos os objetos ja analisados. No pior cenário, onde poucos objetos se ocultam, o horizonte cresce a cada passo. Isso significa que o trabalho aumenta progressivamente: para o primeiro objeto o custo é mínimo, para o segundo é maior, para o terceiro maior ainda, e assim por diante. Esse trabalho acumulado se assemelha à soma de uma progressão aritmética:

$$(1 + 2 + \cdots + (n-1)) = \sum_{i=0}^{n-1} i = \frac{n(n-1)}{2} = \frac{n^2 - n}{2} \tag{1}$$

  resultando assim em uma complexidade $O(n^2)$

A complexidade total é a do termo dominante, ou seja, o que mais cresce, portanto, a complexidade de tempo da função `geraCena` é $\mathbf{O(n^2)}$.

3.2 Complexidade de Espaço

A complexidade de espaço refere-se a memória adicional utilizada pelo algoritmo.

- **Estruturas de Dados:** Os arrays para o `horizonte` e para a mesclagem (`temp_horizonte`) podem, no pior caso, armazenar n intervalos disjuntos, resultando em um uso de espaço de $O(n)$.

- **Ordenação:** O Merge Sort requer um array auxiliar de tamanho n, consumindo $O(n)$ de espaço. O Quick Sort é um algoritmo in-place, ou seja, não utiliza memória adicional significativa,tendo em médio o espaço extra com complexidade $O(\log n)$, mas no pior caso pode chegar a $O(n)$.

O fator dominante é linear, portanto, a complexidade de espaço é $\mathbf{O(n)}$.

4 Estratégias de Robustez

Para garantir a estabilidade do programa, algumas estratégias de programação defensiva foram consideradas.

- **Algoritmo de Ordenação Estável:** A decisão de utilizar um algoritmo de ordenação estável, como o MergeSort foi tomada para garantir a consistência em casos onde os objetos possuem a mesma profundidade e ocorre a interseção entre eles no eixo X, garantindo que o objeto criado primeiro tenha prioridade de visibilidade, fazendo com que o outro objeto seja automaticamente ocluso.

- **Limites de Capacidade:** Como o sistema utiliza arrays estáticos com tamanho pré-definido (`MAX_OBJETOS`), foram implementadas verificações nas funções de adição de objetos e segmentos. Caso o número de elementos exceda a capacidade máxima do array, uma mensagem de erro é exibida no `std::cerr` e a adição é não é concluída, prevenindo um overflow.

- **Entrada de Dados:** A implementação assume que o formato da entrada é sempre válido, conforme o especificado. Uma versão mais robusta incluiria validações para cada comando lido, tratando casos de comandos não esperados, falta de parâmetros ou tipos de dados incorretos, garantindo que o programa não apresente comportamento inesperado com entradas não consistentes.

5 Análise Experimental

O objetivo desta análise foi avaliar experimentalmente o **compromisso** de desempenho entre diferentes estratégias de ordenação, um passo fundamental para o algoritmo de oclusão. Foram comparadas três abordagens principais e investigado o impacto de limiares de reordenação intermediários.

5.1 Estratégias Avaliadas

- **Estratégia 1 (Ordenar a cada Movimento):** O array de objetos é reordenado a cada comando 'M' utilizando o algoritmo **Insertion Sort**, ideal para arrays quase ordenados.

- **Estratégia 2 (Ordenar sob Demanda - Merge Sort):** O array só é ordenado quando um comando 'C' é recebido, utilizando o **Merge Sort**, que garante complexidade de pior caso $O(n \log n)$.

- **Estratégia 3 (Ordenar sob Demanda - Quick Sort):** Idêntica a Estratégia 2, mas utilizando o **Quick Sort**.

5.2 Metodologia

Os testes foram executados na mesma máquina e a métrica de desempenho foi o tempo gasto exclusivamente nas operações de ordenação, medido em nanossegundos com a biblioteca `<chrono>` do C++11. Foram gerados casos de teste variando dois parâmetros principais:

- **Número de Objetos (n):** Testes com $n \in \{100, 500, 1000, 5000\}$.
- **Frequência de Movimentos (f):** Testes com alta frequência (100 comandos 'M' para cada 'C') e baixa frequência (10 'M' para cada 'C').
- **Número de Cenas (C):** Os testes de frequência (com N=1000) foram executados com $C = 500$, enquanto os testes de escalabilidade (com N variável) utilizaram $C = 100$.

5.3 Resultados e Discussão

Os dados coletados nos experimentos foram analisados e serão apresentados a seguir em tabelas e graficos. A análise foca no tempo de ordenação, que foi isolado para uma comparação justa entre as estratégias.

Estratégia	**Baixa Frequência (ms)**	**Alta Frequência (ms)**
InsertionSort por Movimento	3015.6	570.9
QuickSort por Cena	1049.0	286.8
MergeSort por Cena	**57.3**	**62.2**

Tabela 1: Comparação do Tempo de Ordenação (em ms) para N=1000 nos cenários de baixa e alta frequência de movimentos.

Limiar	**QuickSort (ms)**	**MergeSort (ms)**
10	8928.4	7664.7
50	1260.4	682.4
100	781.6	356.1
250	508.0	182.2
500	392.3	131.6
1000	354.1	**105.3**

Tabela 2: Análise do Tempo de Ordenação (em ms) vs. Limiar para o cenário de alta frequência (N=1000).

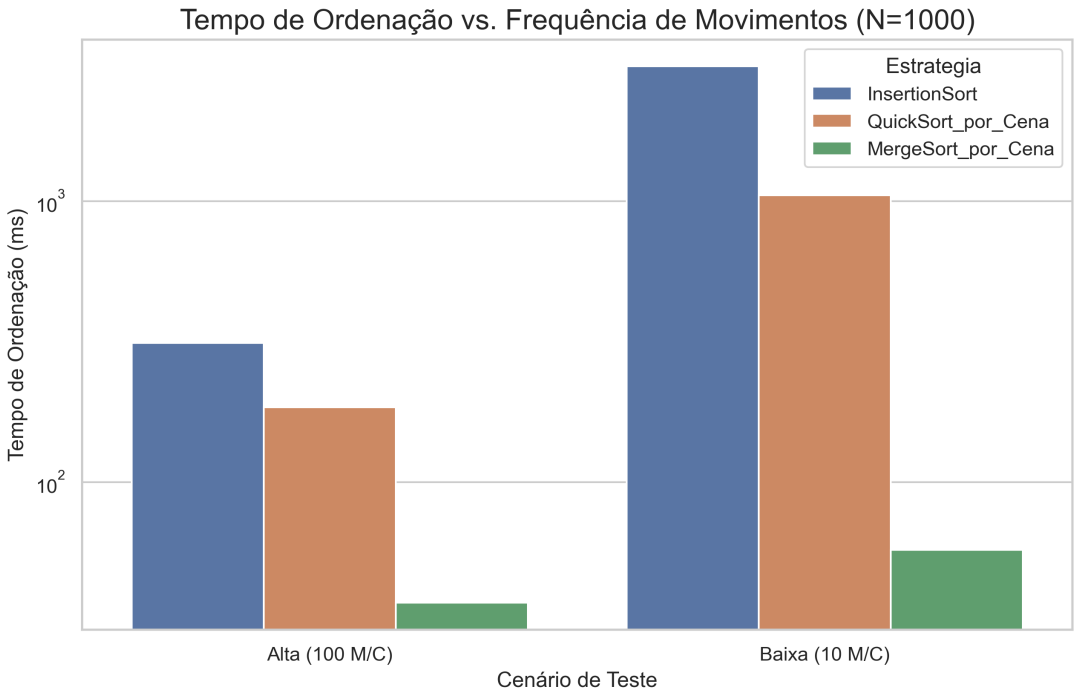

Figura 1: Comparativo de desempenho entre as três estratégias principais.

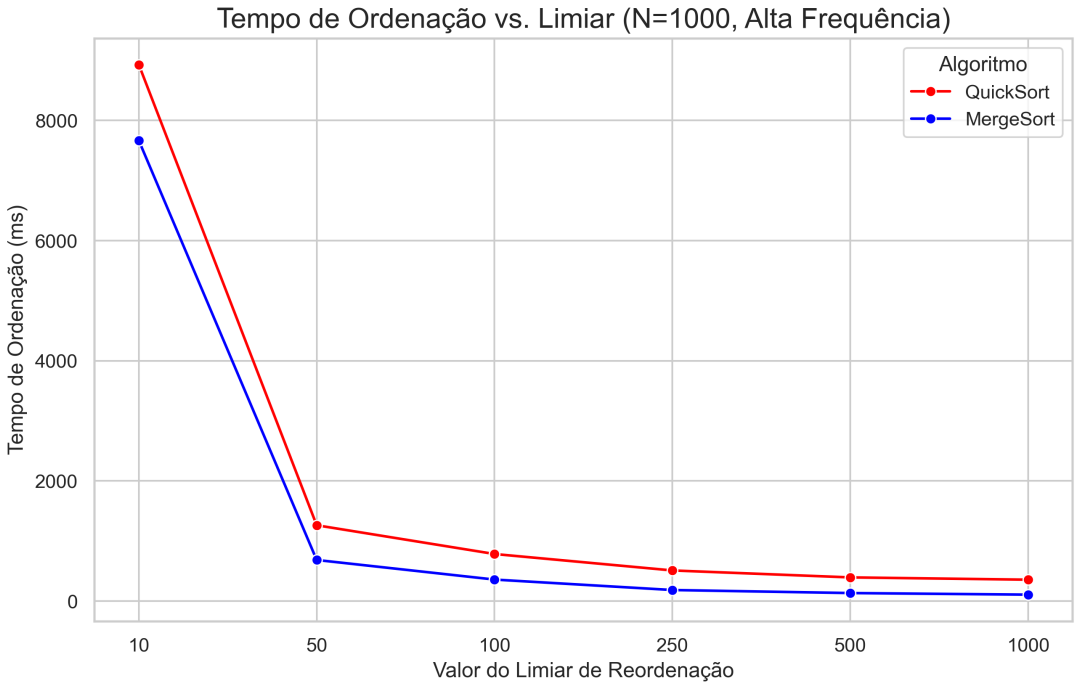

Figura 2: Análise do impacto do limiar de reordenação no tempo total de ordenação. A tendência para ambos os algoritmos é a melhora do desempenho a medida que limiar aumenta.

A análise dos dados permite extrair as seguintes conclusões:

- **Superioridade da Estratégia "Ordenar sob Demanda":** Conforme a Tabela 1 e a Figura 1, as estratégias que ordenam apenas no comando 'C' ('QuickSort'

e ‘MergeSort‘) foram muito mais eficientes do que ordenação a cada movimento (‘InsertionSort‘), principalmente no cenário de baixa frequência.

- **Desempenho Inesperado do Quick Sort:** Contrariando a expectativa de que o ‘QuickSort‘ seria tão rápido quanto o ‘MergeSort‘, os dados mostraram que o quickSort apresentou tempos significativamente mais altos, sugerindo que o método encontrou casos de particionamento desfavoráveis com os dados gerados aleatoriamente, se aproximando de sua complexidade de pior caso.

- **Análise do Limiar:** A Tabela 2 e a Figura 2 demonstram que o custo de ordenação diminui drasticamente à medida que o limiar aumenta. Não foi encontrado um ”ponto ótimo”intermediário, o melhor desempenho ocorreu com o maior limiar testado. Isso reforça a conclusão de que a melhor abordagem é esperar o máximo possível para reordenar, validando a superioridade da estratégia de ”Ordenar sob Demanda”.

5.4 Observação sobre a Largura dos Objetos

Uma observação teórica adicional, que não foi verificada experimentalmente neste trabalho, diz respeito ao impacto da largura dos objetos no desempenho. É razoável concluir que cenários com objetos de grande largura teriam um tempo de execução total menor na etapa de oclusão. Isso ocorre porque objetos largos aumentam a probabilidade de sobreposição, fazendo com que os primeiros objetos processados cubram rapidamente uma grande porção do ”horizonte”. Consequentemente, a maioria dos objetos mais ao fundo (maior y) seria descartada mais rapidamente, no início de suas respectivas verificações. Este cenário se opõe diretamente ao pior caso de complexidade do algoritmo ($O(n^2)$), que ocorre justamente quando os objetos tem uma largura pequena e não se sobrepõem, fazendo com que o programa ao pior caso, fazendo todas as verificações possíveis para cada um dos objetos (exatamente a PA analisada em 3.1).

6 Conclusões

Neste trabalho, foi implementado em C++ um algoritmo para a resolução do problema de ***Identificação de Objetos Oclusos*** em um cenário unidimensional. O sistema é capaz de processar comandos de criação e movimentação de objetos e gerar uma representação da cena contendo apenas os objetos (ou parte de objetos) visíveis.

O desenvolvimento permitiu a aplicação prática de conceitos fundamentais de estruturas de dados, como a implementação e análise de complexidade de algoritmos. A análise de complexidade revelou que o algoritmo de geração de cena possui uma complexidade de tempo de $O(n^2)$, sendo o a fase mais custosa o cálculo de oclusão (ou visibilidade) em relação ao horizonte. A análise experimental reforçou a importância da escolha do algoritmo correto para o contexto do problema, enfatizando o compromisso de desempenho entre realizar atualizações frequentes e baratas versus atualizações raras e custosas.

7 Bibliografia

Referências

[1] Anisio, Marcio, Wagner, Washington. (2025). *DCC205/DCC221 Estruturas de Dados: Trabalho Prático 1 - Identificação de Objetos Oclusos*. DCC/ICEx/UFMG.

[2] Anisio, Marcio, Wagner, Washington, (2025). *Slides da Disciplina Estrutura de Dados* [Material Didático disponível via moodle]. DCC/ICEx/UFMG.

[3] Chamowicz, L. (2025). *Slides Disciplina Programação e Desenvolvimento de Software II* [Material Didático disponível via moodle]. DCC/ICEx/UFMG.

[4] L. Downs, T. Möller, and C. H. Séquin, *Occlusion Horizons for Driving through Urban Scenery*, In: *Proceedings of the ACM Symposium on Interactive 3D Graphics*, pp. 199–200, 2001. DOI: 10.1145/364338.364378. Disponível também em: https://www.researchgate.net/publication/220792024_Occlusion_horizons_for_driving_through_urban_scenery.

[5] Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2022). *Introduction to algorithms* (4th ed.). MIT Press. O Capítulo 17, em especial a Seção 17.3 ("Interval Trees").

[6] Foley, J. D., van Dam, A., Feiner, S. K., & Hughes, J. F. (1996). *Computer Graphics: Principles and Practice* (2nd ed. in C). Addison-Wesley. O Capítulo 15 ("Visible-Surface Determination").

[7] GeeksforGeeks. *Insertion Sort*. GeeksforGeeks. Acessado em 24 de setembro de 2025. Disponível em: https://www.geeksforgeeks.org/insertion-sort/

[8] GeeksforGeeks. *Quick Sort*. GeeksforGeeks. Acessado em 24 de setembro de 2025. Disponível em: https://www.geeksforgeeks.org/quick-sort/

[9] GeeksforGeeks. *Merge Sort*. GeeksforGeeks. Acessado em 24 de setembro de 2025. Disponível em: https://www.geeksforgeeks.org/merge-sort/