TRABALHO PRÁTICO 1 - IDENTIFICAÇÃO DE OBJETOS OCLUSOS

Murillo Kelvin de Andrade Santos
2024106530
Departamento de Ciência da Computação -
Universidade Federal de Minas Gerais
Belo Horizonte – MG - Brasil
DCC205-TF

October 10, 2025

1 Introdução

No campo da computação gráfica, a otimização da renderização de cenas é um fator crucial para o desempenho de aplicações interativas, como jogos e simulações. Uma técnica primordial para alcançar essa otimização é a remoção de superfícies ocultas, um processo que evita o processamento de objetos que não são visíveis para o observador. Ao desenhar apenas os elementos visíveis, o sistema reduz a carga sobre a unidade de processamento, resultando numa experiência mais fluida e eficiente.

Este trabalho aborda o problema da identificação de oclusão em um cenário unidimensional, conforme proposto no âmbito da disciplina de Estruturas de Dados. O objetivo é implementar um sistema em C++ capaz de interpretar um conjunto de comandos que definem objetos, seus movimentos e a geração de cenas. Para cada comando de cena, o programa deve determinar quais segmentos de reta, ou partes deles, são visíveis, considerando que objetos com maior profundidade (menor coordenada Y) podem ocultar total ou parcialmente aqueles que estão mais distantes.

A solução desenvolvida utiliza Tipos Abstratos de Dados (TADs) para modularizar a representação dos objetos e a lógica de renderização da cena.

O armazenamento dos dados principais é realizado através de arrays estáticos com tamanho pré-definido. O núcleo do algoritmo de oclusão itera sobre cada objeto e o compara com todos os outros para determinar os segmentos visíveis, resultando em uma complexidade de tempo quadrática. Adicionalmente, foi conduzida uma análise experimental para avaliar o compromisso entre diferentes estratégias de ordenação e o desempenho geral do sistema.

2 Método (Implementação)

A solução foi desenvolvida integralmente em C++, respeitando a estrutura de projeto e as restrições impostas pelo enunciado, notadamente a não utilização das bibliotecas de estruturas de dados padrão da linguagem. O foco foi a criação de uma arquitetura modular e eficiente.

2.1 Estrutura de Dados Principal

A coleção de objetos e movimentos da cena é armazenada em arrays estáticos de `Objeto` e `Movimento`, respectivamente. A gestão de memória é estática, utilizando buffers com capacidade máxima pré-definida no arquivo `include/Config.hpp` (`MAX_OBJETOS` e `MAX_MOVIMENTOS`). Esta abordagem elimina a necessidade de realocação dinâmica de memória para as estruturas de dados principais, simplificando o código e evitando a sobrecarga associada à realocação.

2.2 Tipos Abstratos de Dados (TADs)

A implementação foi dividida em TADs para garantir a separação de responsabilidades:

- **TAD Objeto (`include/Objeto.hpp`):** Implementado como uma `struct`, representa um segmento de reta unidimensional. Armazena um `id` inteiro, as coordenadas do seu centro (`centro_x`), sua profundidade (coordenada `y`) e sua `largura`.
- **TAD Cena (`src/Cena.cpp`):** Encapsula a lógica de processamento e geração da cena. Sua função principal, `gerarCena`, executa o algoritmo de identificação de oclusão.
- **TAD Movimento (`include/Tipos.hpp`):** Representa a alteração de posição de um objeto em um determinado instante de tempo.

2.3 Algoritmo de Geração de Cena

O algoritmo implementado em `gerarCena` é o núcleo do trabalho. Para cada cena a ser gerada em um tempo `t`:

1. **Atualização de Posições:** O estado dos objetos é atualizado, aplicando todos os movimentos cujo tempo seja menor ou igual a `t`. Os movimentos são previamente ordenados para garantir a ordem correta das atualizações.

2. **Cálculo de Oclusão por Força Bruta:** O algoritmo itera sobre cada objeto ("objeto alvo") e o compara com todos os outros ("oclusores") em um laço aninhado. Um objeto é um oclusor se sua coordenada `y` for menor (ou igual, com `id` maior).

3. **Subtração de Intervalos:** O intervalo do oclusor é subtraído do intervalo visível do objeto alvo.

4. **Armazenamento e Ordenação:** Os segmentos visíveis resultantes são armazenados em um array dinâmico temporário. Ao final, este array é ordenado pelo `id_objeto` para garantir uma saída determinística.

5. **Impressão Formatada:** Os resultados são impressos no formato especificado.

3 Análise de Complexidade

A seguir, a análise de complexidade para as principais operações do sistema.

- **Adicionar Objeto/Movimento:**
  - **Tempo:** $O(1)$, pois a inserção ocorre em um array estático.
  - **Espaço:** $O(1)$, pois o espaço é pré-definido por constantes.
- **Processar Arquivo (`processarArquivo`):**
  - **Leitura:** $O(L)$, onde L é o número de linhas.
  - **Ordenação de Movimentos:** $O(M \log M)$ com Merge Sort, onde M é o número de movimentos.
- **Gerar Cena (`gerarCena`):**

- **Tempo:** Dominado por duas etapas principais. A atualização de posições, que ocorre antes, tem custo de $O(M \times N)$. O cálculo de oclusão tem custo de $O(N^2)$.
- **Espaço:** Aloca espaço auxiliar $O(N)$ para estruturas temporárias.

4 Estratégias de Robustez

Para garantir a estabilidade do programa, foram implementadas as seguintes estratégias:

- **Validação de Argumentos:** O programa verifica se um arquivo de entrada foi fornecido.
- **Verificação de Abertura de Arquivo:** O programa valida se o arquivo pôde ser aberto, exibindo um erro caso contrário.
- **Controle de Limite dos Buffers:** O sistema verifica se o número de objetos ou movimentos excede a capacidade máxima dos buffers estáticos, ignorando entradas excedentes e emitindo um aviso.
- **Gestão de Memória Temporária:** A alocação dinâmica com `realloc` para estruturas temporárias dentro de `gerarCena` é gerenciada cuidadosamente.

5 Análise Experimental

5.1 Análise de Estratégias de Ordenação

Para avaliar o compromisso entre a frequência de ordenação e o custo de processamento, foi realizada uma análise experimental. O experimento avaliou o tempo de execução de diferentes estratégias de ordenação em função do "Nível de Desordem" do vetor de objetos. A desordem varia de 0.0 (totalmente ordenado) a 1.0 (totalmente aleatório). O teste foi executado em três cenários, com 10, 100 e 1000 objetos.

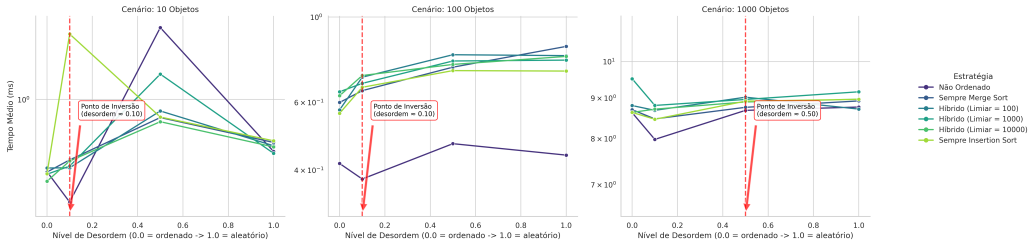

Figure 1: Análise de performance comparando estratégias de ordenação sob diferentes níveis de desordem e escalas de problema.

A análise dos resultados, apresentada na Figura 1, revela dinâmicas distintas para cada cenário:

- **Cenário com 10 Objetos:** O tempo de execução é extremamente baixo e as medições são voláteis, indicando que o ruído do sistema operacional supera o custo real do algoritmo. A otimização da ordenação em problemas de escala tão pequena se mostra impraticável e desnecessária.

- **Cenário com 100 Objetos:** Este cenário ilustra o benefício de uma abordagem adaptativa. O `Sempre Insertion Sort` é mais rápido para dados quase ordenados, mas seu desempenho degrada rapidamente. O `Sempre Merge Sort` é mais lento inicialmente, mas seu custo permanece estável. O "Ponto de Inversão" em um nível de desordem de aproximadamente 0.10 demonstra o momento em que o Merge Sort se torna mais vantajoso.

- **Cenário com 1000 Objetos:** As curvas de tempo se aproximam. Embora a ordenação ainda traga benefícios, a diferença de performance entre as estratégias é menos pronunciada, pois o custo da etapa de atualização de movimentos ($O(M \times N)$) torna-se o principal gargalo do sistema, diminuindo o impacto relativo da otimização do algoritmo de ordenação.

5.2 Análise de Parâmetros da Simulação

Foram realizados experimentos adicionais para avaliar como diferentes características dos objetos na cena impactam o desempenho geral.

5.2.1 Impacto da Densidade Espacial

Este experimento avaliou como a densidade dos objetos afeta o tempo de execução. A densidade foi controlada alterando-se o "Range Espacial" no qual os centros dos objetos são gerados; um range menor implica em uma maior densidade. Conforme a hipótese, uma maior densidade deveria levar a mais oclusões, simplificando a cena e reduzindo o tempo de processamento.

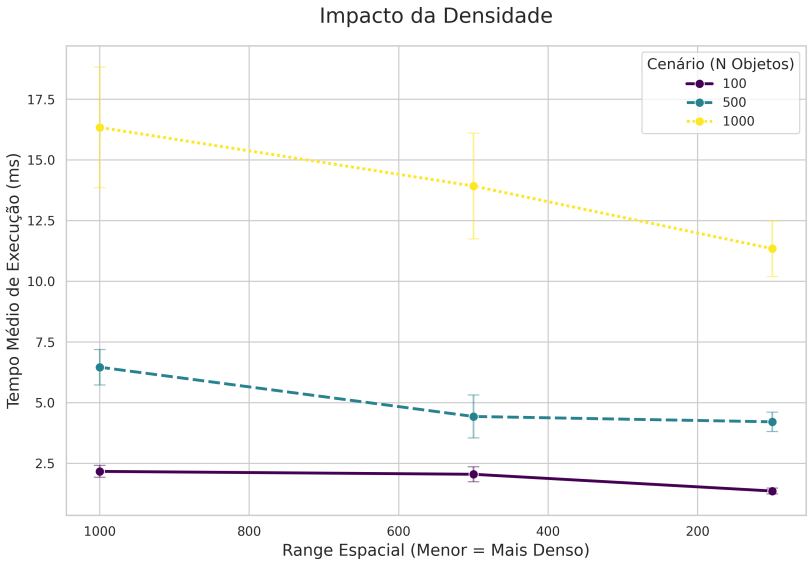

Figure 2: Impacto da densidade (controlada pelo range espacial) no tempo de execução.

Os resultados na Figura 2 confirmam a hipótese: para todos os cenários, à medida que o range espacial diminui (densidade aumenta), o tempo médio de execução tende a cair. Isso ocorre porque objetos mais próximos têm maior probabilidade de se sobreporem, resultando em menos segmentos visíveis a serem processados pelo algoritmo de oclusão.

5.2.2 Impacto da Largura dos Objetos

A largura dos objetos também influencia diretamente a quantidade de oclusões. Espera-se que objetos mais largos causem mais oclusões, simplificando a cena.

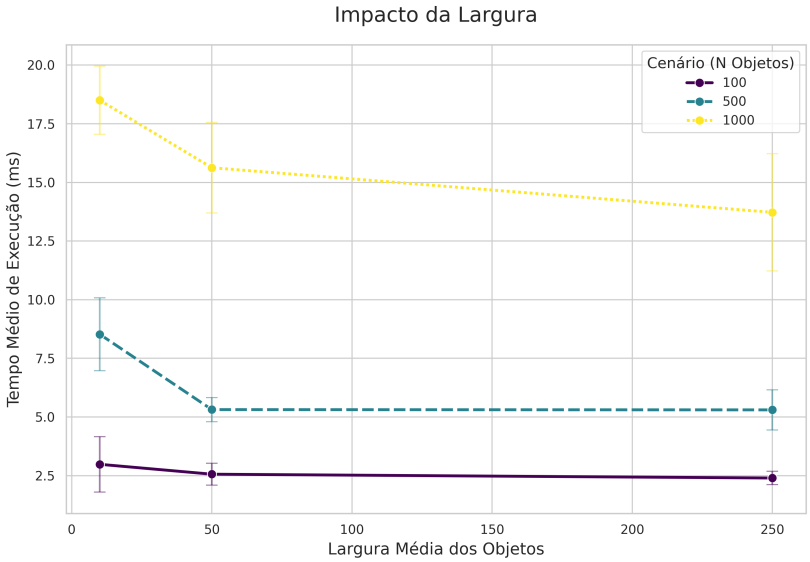

Figure 3: Impacto da largura média dos objetos no tempo de execução.

Como visto na Figura 3, o aumento da largura média dos objetos leva a uma redução no tempo de execução. O efeito é mais acentuado em cenários com mais objetos, onde a probabilidade de oclusão é naturalmente maior. Isso valida que a complexidade da cena final, em termos de segmentos visíveis, é um fator determinante para o desempenho.

5.2.3 Impacto da Velocidade (Nível de Desordem)

A velocidade com que os objetos se movem está diretamente ligada à desorganização do vetor de objetos entre os passos de renderização, exigindo mais esforço do algoritmo de ordenação.

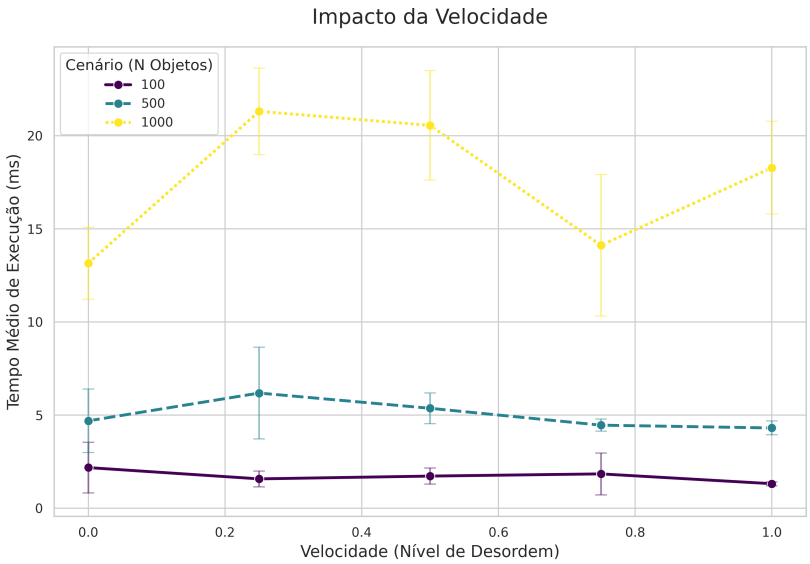

Figure 4: Impacto da velocidade (nível de desordem) no tempo de execução.

A Figura 4 mostra que, para um número médio de objetos (N=100), um maior nível de desordem (maior velocidade) tende a aumentar o tempo de execução, pois o algoritmo de ordenação precisa realizar mais trocas. No entanto, para N=1000, o impacto da desordem no tempo total é menos significativo. Isso reforça a conclusão anterior de que, em cenários de alta carga, o gargalo $O(M \times N)$ da atualização de estados mascara os custos relativos da etapa de ordenação.

6 Proposta de Otimização: Algoritmo de Linha de Varredura

O gargalo de desempenho do sistema atual é o algoritmo de cálculo de oclusão com complexidade $O(N^2)$. Uma otimização significativa pode ser alcançada substituindo-o por um algoritmo baseado em **linha de varredura** (*sweep-line*), que possui complexidade $O(N \log N)$.

- **Estratégia:** Criar "pontos de evento" para o início e fim de cada objeto, ordená-los por sua coordenada X e percorrer o eixo mantendo uma estrutura de dados ativa (ex: árvore balanceada) ordenada por profundidade para determinar a visibilidade em cada intervalo.
- **Justificativa:** A ordenação dos pontos custa $O(N \log N)$ e cada um

dos 2N eventos é processado em tempo $O(\log N)$. A complexidade total seria dominada pela ordenação, resultando em $O(N \log N)$.

7 Conclusão

Neste trabalho, foi implementada uma solução funcional para o problema de identificação de objetos oclusos. A análise experimental demonstrou que a escolha de um algoritmo de ordenação ideal é dependente do contexto, incluindo a escala do problema, a desordem inicial dos dados e a existência de outros gargalos computacionais no sistema. Foi provado que estratégias adaptativas (híbridas) são mais robustas, com seu benefício sendo mais pronunciado em cenários de médio porte. A análise de complexidade teórica, validada pelos experimentos, reforça a importância de otimizar a etapa correta do algoritmo para obter ganhos de performance significativos.

8 Bibliografia

References

[1] Cormen, T., Leiserson, C., Rivest R., Stein, C. *Introduction to Algorithms.*

[2] Slides da disciplina DCC205 - *Estruturas de Dados, apresentados em sala de aula. DCC/ICEx/UFMG.*

[3] Paulo Feofillof (2019) CS IME USP, *Algorithms Merge Sort.*

[4] Learn Microsoft (2024), *Ponteiros (C++).*

[5] Idbrii StackOverflow (2024), *How do I use arrays in C++, Pointer arithmetic.*

[6] Bentley, J. L., & Ottmann, T. A. (1979). *Algorithms for reporting and counting geometric intersections.* IEEE Transactions on Computers, C-28(9), 643–647.

[7] Stroustrup, B. (2014). *A Linguagem de Programação C++ (4ª ed.).* Bookman.

[8] Celes, W., Cerqueira, R., & Rangel, J. L. (2004). *Estruturas de Dados com C++.* Editora Campus.