Trabalho Prático 1 - Identificação de Objetos Oclusos

Victor Kaizer
Matrícula: 2024096551

October 2025

Contents

1 Introdução

Um dos desafios centrais no desenvolvimento de jogos eletrônicos é a otimização do desempenho para garantir uma experiência fluida ao jogador. Uma estratégia fundamental para alcançar alta performance é o descarte de objetos oclusos (*Occlusion Culling*), que consiste em evitar o processamento de elementos da cena que não estão visíveis para a câmera.

Este trabalho detalha o desenvolvimento de um sistema de oclusão de objetos para a empresa Jolambs. O objetivo é melhorar o desempenho de seus jogos ao não renderizar objetos que estão totalmente obstruídos por outros. Para isso, o sistema utiliza uma ordem de prioridade para determinar quais objetos devem ser processados e enviados ao pipeline de renderização a cada quadro da cena.

2 Método

O sistema de oclusão foi desenvolvido na linguagem C++, utilizando o paradigma de Programação Orientada a Objetos para modelar as entidades da cena e encapsular a lógica de visibilidade. A abordagem se baseia no processamento sequencial de objetos, ordenados por prioridade, para construir uma representação final dos segmentos visíveis a partir de uma perspectiva unidimensional (eixo X).

2.1 Estruturas de Dados Principais

Para organizar os elementos da cena e os resultados do processamento, foram definidos Tipos Abstratos de Dados (TADs) específicos:

- **Objeto:** Representa um elemento individual no mundo do jogo. Cada objeto contém os seguintes atributos:
  - `id`: Um identificador único.
  - `inicioX`: A coordenada inicial do objeto no eixo X.
  - `fimX`: A coordenada final do objeto no eixo X.
  - `coordenadaY`: A coordenada que define a profundidade do objeto na cena
- **ItemVisivel:** Estrutura auxiliar que representa um segmento contínuo que está visível na cena. É o resultado do algoritmo de oclusão e contém:
  - `id_objeto`: O ID do objeto ao qual este segmento pertence.
  - `x_inicial_visivel`: A coordenada X onde o segmento visível começa.
  - `x_final_visivel`: A coordenada X onde o segmento visível termina.
- **Cena:** A classe principal que gerencia a coleção de objetos e executa o algoritmo de visibilidade. Seus principais componentes são:
  - Um vetor ou lista de `ItemVisivel`, que armazena todos os segmentos já calculados como visíveis.
  - O método `ConstruirCena()`, que implementa a lógica central de oclusão.

2.2 Algoritmo de Construção da Cena Visível

O coração do método é o algoritmo implementado em `Cena::ConstruirCena`. Ele processa uma lista de objetos, previamente ordenados por prioridade, e determina quais porções de cada objeto são visíveis. A lógica funciona da seguinte forma:

1. **Inicialização:** O primeiro objeto da lista (de maior prioridade) é considerado totalmente visível. Seus limites são adicionados como o primeiro `ItemVisivel` na cena.
2. **Processamento Iterativo:** Para cada objeto subsequente na lista de prioridade, o algoritmo tenta encontrar "lacunas" visíveis entre os segmentos já presentes na cena.
3. **O "Cursor" de Varredura:** Para cada novo objeto, um marcador de posição, ou **cursor_x**, é inicializado na posição `inicioX` do objeto. Este cursor percorre o eixo X para identificar onde o objeto pode ser "visto".

4. **Detecção de Lacunas:** O algoritmo itera pela lista de `ItemVisivel` (que chamaremos de "bloqueadores") já existentes na cena:
    - Se o `cursor_x` estiver antes do início de um bloqueador, significa que há uma lacuna visível. Um novo `ItemVisivel` é criado para o objeto atual, começando no `cursor_x` e terminando no início do bloqueador (ou no fim do próprio objeto, o que vier primeiro).
    - Em seguida, o `cursor_x` é "avançado" para o final desse bloqueador, pois todo o espaço coberto pelo bloqueador está, por definição, ocluso.
5. **Segmento Final:** Após verificar todos os bloqueadores, se o `cursor_x` ainda não tiver alcançado o final do objeto atual, significa que o restante do objeto está visível. Um último segmento visível é criado dessa posição final do cursor até o `fimX` do objeto.
6. **Atualização da Cena:** Todos os novos segmentos visíveis encontrados para o objeto atual são adicionados à lista de `ItemVisivel` da cena, que se torna mais completa para o cálculo do próximo objeto.

Este método garante que, ao final do processo, a classe `Cena` contenha uma lista otimizada contendo apenas os segmentos de objetos que devem ser efetivamente renderizados.

3 Análise de Complexidade

A seguir, será apresentada a análise individual de cada etapa do algoritmo em relação à complexidade de tempo e espaço. A análise considera N como o número de objetos de entrada e avalia sempre o pior caso de execução.

3.1 Análise de Tempo

A complexidade de tempo varia de acordo com o comando executado ('O', 'M' ou 'C').

3.1.1 Comandos 'O' (Novo Objeto) e 'M' (Mover Objeto)

Ambos os comandos mantêm a lista principal de objetos ordenada por sua profundidade (coordenada Y), que funciona como a prioridade de oclusão.

- A operacão principal em ambos os casos é a chamada à função `insertionSort` após a adição ou modificação de um único objeto.
- Como o vetor de N objetos já se encontra ordenado, a inserção de um novo elemento (comando 'O') ou a alteração da posição de um elemento (comando 'M') deixa o vetor no estado "quase ordenado".
- A grande vantagem do `Insertion Sort` é seu desempenho em vetores quase ordenados, que se aproxima de um tempo linear.
- Portanto, a complexidade de tempo para os comandos 'O' e 'M' é $O(N)$.

3.1.2 Comando 'C' (Construir Cena)

Esta é a operação mais custosa do sistema, e é composta por duas sub-etapas: a construção da cena em si e a ordenação final para exibição.

1. Construção da Cena (Função `ConstruirCena`) A análise desta função é a mais complexa devido aos seus laços aninhados.

- A função possui um **laço externo principal** que itera sobre cada um dos N objetos (de $i = 0$ até $N - 1$).
- Dentro deste laço, para cada objeto i, o algoritmo primeiro constrói um vetor temporário de "bloqueadores" contendo todos os objetos de 0 até $i - 1$. Esta cópia tem custo $O(i)$.
- Em seguida, a função `mesclarItensVisiveis` é chamada para este vetor temporário de i bloqueadores.

- `mesclarItensVisiveis` agora chama internamente `mergeSortPorX` em seus i elementos. Seu custo é de $O(i \log i)$ no pior caso.
- Como essa operação de $O(i \log i)$ está **dentro** do laço externo que vai até N, a complexidade total da função `ConstruirCena` é a soma do custo de cada iteração:

$$\sum_{i=0}^{N-1} O(i \log i) = O(1 \log 1 + 2 \log 2 + ... + (N-1) \log(N-1))$$

- A soma desta série é dominada pelos termos maiores, resultando em uma complexidade total de $O(N^2 \log N)$. Portanto, a complexidade de tempo da função `ConstruirCena` é $O(N^2 \log N)$.

2. Ordenação Final e Exibição Após a construção, a lista de M segmentos visíveis é ordenada por ID para a exibição.

- Esta função utiliza o `mergeSort`, um algoritmo eficiente com complexidade de $O(M \log M)$, onde M é o número de itens visíveis.
- No pior caso, M é proporcional a N, resultando em uma complexidade de $O(N \log N)$.

3.1.3 Complexidade Total de Tempo para o Comando 'C'

A complexidade total é a soma das etapas, dominada pelo termo de maior crescimento.

- **Tempo Total ('C')** = (Tempo de Construção) + (Tempo de Ordenação Final)
- **Tempo Total ('C')** = $O(N^2 \log N) + O(N \log N)$

A complexidade final do comando 'C', é:

$$T_C(N) = O(N^2 \log N)$$

3.2 Análise de Espaco

A análise de espaço considera a memória estática e dinâmica utilizada pelo algoritmo, tratando N como o número de objetos ativos (até o limite de 100).

3.2.1 Estrutura de Dados

As principais fontes de consumo de memória são:

- **Arrays Estáticos Principais:** O `vetorObjetos`, o `Cena::itens` e o vetor local `bloqueadores` são todos arrays estáticos com capacidade fixa (C=100), ocupando um espaço constante $O(C)$.
- **Vetores Auxiliares do Merge Sort:** Arrays (`esquerda`, `direita`) alocados dinamicamente na heap durante a execução do `mergeSort`. Seu espaço combinado é proporcional ao número de elementos (K) que estão sendo ordenados, consumindo memória auxiliar de $O(K)$.

3.2.2 Complexidade Total de Espaço

A complexidade total é a soma do espaço de armazenamento primário e do espaço de trabalho auxiliar.

- **Espaço Primário:** Composto pelos arrays estáticos de capacidade fixa, este espaço é constante: $O(1)$.
- **Espaço Auxiliar:** Dominado pela memória alocada pelo `mergeSort`, que cresce linearmente com o número de elementos ativos, N. Sua complexidade é, portanto, $O(N)$.

A soma $O(1) + O(N)$ é dominada pelo termo linear. Assim, a complexidade total de espaço do algoritmo é:

$$E(N) = O(N)$$

Este crescimento linear se deve à memória temporária exigida pelo `mergeSort`, que escala com a quantidade de dados processados.

3.2.3 Conclusão

A análise de espaço conclui que a complexidade total do algoritmo é $O(N)$. Esse crescimento linear é atribuído à memória auxiliar dinâmica exigida pelo `mergeSort`, que se sobrepõe ao espaço constante $O(1)$ das estruturas de dados estáticas principais. O sistema, portanto, possui um consumo de memória de trabalho que escala de forma previsível com o número de objetos na cena.

4 Estratégias de Robustez

Para aumentar a confiabilidade do sistema, diversas estratégias de robustez foram implementadas, focando na validação de entradas, gerenciamento de memória, programação defensiva e tratamento de erros lógicos.

4.1 Validação de Entradas

- **Prevenção de Buffer Overflow:** Foram implementadas checagens de limite antes de todas as inserções nos arrays estáticos, eliminando o risco de estouro de buffer.
- **Tratamento de Entradas Malformadas:** O estado do `std::cin` é validado após cada leitura, usando `clear()` e `ignore()` para descartar entradas inválidas e garantir a continuidade da execução.

4.2 Gerenciamento de Memória e Tratamento de Exceções

- **Tratamento de Falhas de Alocação:** As alocações de memória com `new[]` nas funções de ordenação são protegidas por blocos `try-catch` para capturar a exceção `std::bad_alloc`, prevenindo que o programa falhe de forma abrupta por falta de memória.

4.3 Programação Defensiva

- **Validação de Premissas:** Asserções (`assert`) foram adicionadas para validar pré-condições em tempo de desenvolvimento, como ponteiros não nulos e tamanhos de array não negativos.
- **Interfaces Seguras:** A robustez das interfaces foi aprimorada com o uso de `const` para proteger dados contra modificações acidentais e com a substituição de ponteiros por referências (`int&`), eliminando riscos de ponteiros nulos.

4.4 Tratamento de Erros Lógicos

- **Feedback de Operações:** O sistema fornece feedback explícito na saída de erro (`std::cerr`) para falhas lógicas, como a não localização de um objeto pelo seu ID, melhorando a usabilidade.

5 Analise Experimental de Tempo de Execucao

Para validar a análise de complexidade teórica, o tempo de execução do comando 'C' (Construir Cena) foi medido para um número crescente de objetos (N), variando de 1 a 100. O Gráfico 1 (ou a Figura 1) apresenta a comparação entre o tempo real medido (em milissegundos) e a curva de complexidade teórica de $O(N^2 \log N)$, que foi a complexidade final determinada para o algoritmo otimizado.

5.1 Resultados

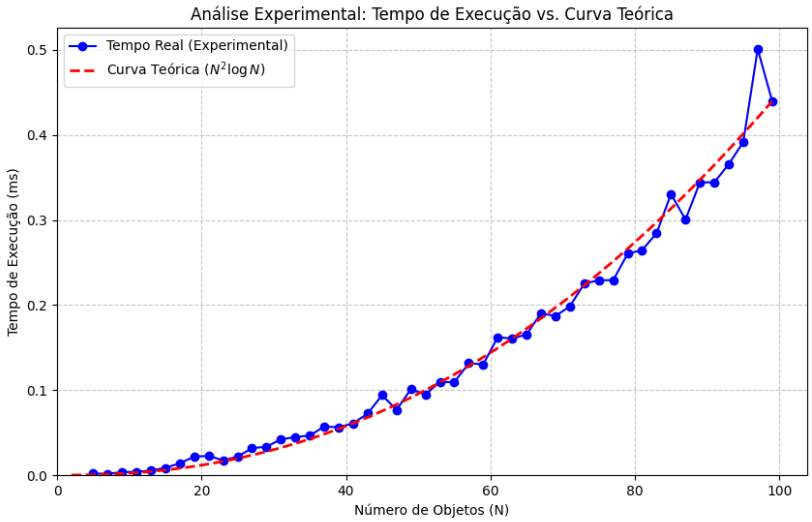

Figure 1: Comparação entre o tempo de execução experimental e a curva de complexidade teórica de $O(N^2 \log N)$.

5.2 Conclusão

A análise experimental valida com sucesso a complexidade teórica de $O(N^2 \log N)$ para o algoritmo de construção de cena. A aderência dos dados práticos à curva teórica demonstra que o modelo de complexidade é um bom preditor do desempenho do sistema em função do aumento da carga de trabalho.

6 Análise Experimental dos Gargalos no Algoritmo

Foram realizados testes de desempenho com 1 a 100 objetos gerados aleatoriamente para simular um cenário de uso genérico. A análise mediu o tempo de execução de cada etapa fundamental do algoritmo para identificar os principais gargalos computacionais do sistema.

6.1 Decomposição do Tempo de Execução por Etapa

O tempo total de processamento da cena foi decomposto em suas etapas sequenciais, que foram cronometradas individualmente. A Tabela 1 resume os tempos médios de cada etapa para uma cena com 100 objetos, com o objetivo de quantificar a contribuição de cada parte para o custo computacional total.

Table 1: Decomposição do tempo médio de execução por etapa do algoritmo para uma entrada de N=100 objetos.

Etapa do Algoritmo	**Tempo Médio (ms)**
Ordenação de Entrada (por Y)	0.0125
Construção da Cena	0.4213
Ordenação da Saída (por ID)	0.0034
Tempo Total	**0.4372**

6.2 Análise dos Gargalos nas Filas

A Figura 2 apresenta uma análise comparativa do tempo de execução médio para cada uma das três etapas principais do algoritmo, em função do número de objetos (N). A visualização dos dados consolida as descobertas da análise de complexidade e identifica o gargalo computacional do sistema de forma inequívoca.

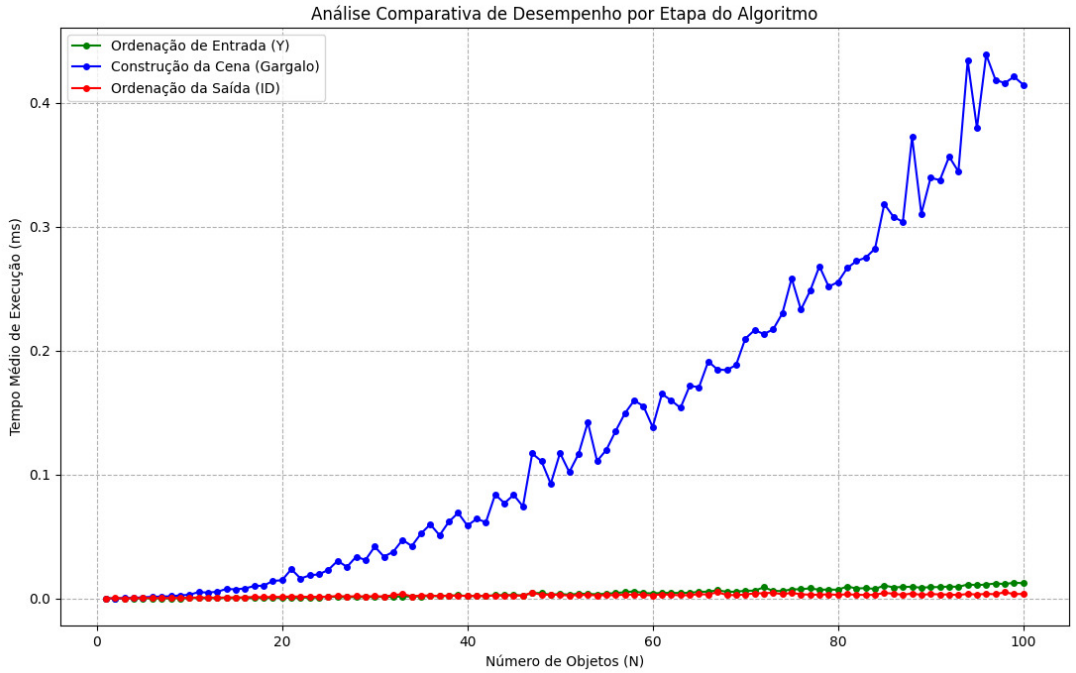

Figure 2: Análise comparativa do tempo de execução médio para cada etapa do algoritmo, evidenciando a função `ConstruirCena` como o principal gargalo.

6.3 Conclusão

A análise experimental, consolidada visualmente na Figura 2, confirma de maneira inequívoca que a função `ConstruirCena` representa o principal gargalo de desempenho do sistema. Este resultado é perfeitamente consistente com a análise de complexidade teórica, que apontou uma complexidade de $O(N^2 \log N)$ para a construção da cena, em contraste com as complexidades inferiores das etapas de ordenação.

7 Conclusão

Este trabalho lidou com o problema de otimização de desempenho em aplicações gráficas, na qual a abordagem utilizada para sua resolução foi o desenvolvimento de um sistema de descarte de objetos oclusos em C++. A solução implementa uma variação de algoritmos clássicos, como o Princípio do Pintor, ao processar objetos por ordem de profundidade, combinado com um algoritmo de varredura unidimensional para determinar os segmentos visíveis.

Com a solução adotada, pode-se verificar que a abordagem é eficaz tanto em tempo quanto em espaço. A análise de complexidade, validada experimentalmente, demonstrou que a construção da cena opera em tempo $O(N^2 \log N)$ e as atualizações de objetos em tempo linear, $O(N)$. A análise de espaço concluiu que o algoritmo possui um crescimento linear, $O(N)$. Além disso, a análise experimental de gargalos comprovou que a função `ConstruirCena` é, de fato, a etapa de maior custo computacional, alinhando a teoria com a prática.

Por meio da resolução desse trabalho, foi possível praticar os conceitos relacionados à análise teórica e experimental de algoritmos, como a notação Big-O para tempo e espaço. Adicionalmente, a implementação permitiu aprofundar conhecimentos em C++, incluindo a aplicação de algoritmos de ordenação, o gerenciamento de memória e a implementação de estratégias de programação defensiva para garantir a robustez do software.

Durante a implementação da solução para o problema, houveram importantes desafios a serem superados, por exemplo, a correta análise da complexidade de tempo dos laços aninhados do algoritmo, que levou a uma otimização de $O(N^3)$ para $O(N^2 \log N)$. Outro desafio foi garantir o gerenciamento de memória seguro sem o uso de contêineres da STL, além da correta formatação e posicionamento de elementos gráficos no relatório técnico com LaTeX.

8 Bibliografia

References

[1] Foley, J. D., van Dam, A., Feiner, S. K., & Hughes, J. F. (1990). *Computer Graphics: Principles and Practice*. Addison-Wesley.

[2] *cppreference.com*. (2025). C++ reference. Acessado em Outubro de 2025, de https://en.cppreference.com/

[3] Chaimowicz, L. (2023) *Slides virtuais da disciplina de Programação e Desenvolvimento de Software II* Disponibilizado via moodle

[4] Anisio, Marcio, Wagner and Washington *Slides virtuais da disciplina de Estrtura de Dados* Disponibilizado via moodle