# Trabalho Prático 1 - Identificação de Objetos Oclusos

**Igor Henrique Martins de Almeida**

Departamento de Ciência da Computação - Universidade Federal de Minas Gerais (UFMG)

Belo Horizonte - MG - Brasil

igorhma@ufmg.br

## 1. Introdução

Esta documentação lida com o problema da identificação de objetos oclusos em um cenário unidimensional, uma estratégia utilizada para otimizar o desempenho de jogos ao reduzir a complexidade das cenas renderizadas, exibindo apenas os segmentos de objetos que são visíveis ao observador. O objetivo desta documentação é apresentar a solução desenvolvida para o problema dado, detalhando a implementação de um programa em linguagem C capaz de processar a criação, movimentação e visualização de objetos. Além disso, o documento aborda a análise de complexidade dos algoritmos empregados, as estratégias de robustez para a estabilidade do programa e a análise experimental sobre diferentes abordagens de ordenação de objetos.

Para resolver o problema, foi seguida uma abordagem modular com o código organizado em diretórios específicos para cabeçalhos (*include*), fontes (*src*) e executáveis (*bin*). A solução utiliza Tipos Abstratos de Dados (TADs) para representar os objetos e as cenas, implementados como estruturas e armazenados em vetores estáticos para que os dados ficassem contíguos em memória.

A documentação está organizada da seguinte forma: a Seção 2 descreve em detalhes a implementação, as estruturas de dados e os algoritmos utilizados. A Seção 3 fornece as instruções para a compilação e execução do programa. A Seção 4 apresenta a análise formal da complexidade de tempo e espaço. A Seção 5 detalha as estratégias de programação defensiva adotadas para se manter a robustez do código. A Seção 6 expõe a análise experimental realizada. Por fim, a Seção 7 apresenta as conclusões do trabalho e a Seção 8, a bibliografia consultada.

## 2. Implementação

Para a implementação da solução, o código foi organizado de forma modular, seguindo a estrutura de diretórios exigida. A pasta *include* armazena os arquivos de cabeçalho (.h), que definem as interfaces dos Tipos Abstratos de Dados (TADs) e os protótipos das funções. A pasta *src* contém os arquivos de código-fonte (.c), onde as

funcionalidades são de fato implementadas. A compilação, gerenciada pelo *Makefile*, gera os arquivos objetos no diretório *obj* e o executável final, *tp1.out*, no diretório *bin*.

As principais estruturas de dados utilizadas foram *Objeto_t*, *Cena_t* e *Intervalo_t*, todas definidas como *structs* em C. A *Objeto_t* armazena as informações de cada objeto no cenário, incluindo seu *id*, as coordenadas de seu centro (*x*, *y*), sua *largura* e os pontos *inicioX* e *fimX* que delimitam sua extensão no eixo horizontal. A *Cena_t* representa um segmento visível de um objeto em um determinado tempo, contendo o *id_obj*, e as coordenadas de *inicio* e *fim* desse segmento. Por fim, a *Intervalo_t* é uma estrutura auxiliar, composta por *inicio* e *fim*, utilizada para gerenciar os intervalos de oclusão. Vetores estáticos foram empregados para armazenar os objetos da cena (*vetorObjs*), os segmentos visíveis (*vetorCena*) e os intervalos oclusos (*intervalo_ocluso*), evitando a alocação dinâmica de memória e garantindo a contiguidade dos dados.

O programa principal opera em um laço que processa a entrada padrão linha por linha. Ele identifica o tipo de operação (“O” para objeto, “M” para movimentação, “C” para cena) e age de acordo. Ao ler um “O”, cria um objeto e o adiciona ao vetor de objetos. Para uma linha “M”, a função *AttPos* é chamada para atualizar as coordenadas *x* e *y* do objeto especificado. A lógica de ordenação é controlada por diretivas de compilação, podendo ser realizada sempre que ocorre um movimento, somente quando um contador de desorganização atinge um limiar pré-definido ou apenas quando é lido uma entrada de geração de cena (“C”). Em qualquer das ocasiões a ordenação é feita via *QuickSort*.

Os testes do programa foram realizados em um ambiente com o sistema operacional Ubuntu 24.04.3, utilizando a linguagem C. A compilação do código foi feita com o compilador GCC em sua versão 13.3.0. O hardware utilizado para a execução dos testes foi um processador Intel Core i5-12400F e 32GB de memória RAM DDR4.

## 3. Instruções de compilação e execução

Para compilar e executar o programa, siga os passos descritos abaixo:

- Acesse o diretório raiz do projeto através do seu terminal;
- Execute o *Makefile* para compilar o projeto utilizando o seguinte comando: “*make all*”;

- Com esse comando, os arquivos de código-fonte serão compilados, os arquivos objeto (.o) serão gerados na pasta *obj* e o arquivo executável, chamado *tp1.out*, será criado na pasta *bin*;
- Para executar o programa, utilize o seguinte comando, que redireciona um arquivo de entrada para o programa: "*./bin/tp1.out < caminho/para/seu/arquivo_de_entrada.txt*";
- A saída do programa, contendo as cenas geradas, é mostrada diretamente no terminal. Caso queira salvar a saída em um arquivo, proceda da seguinte maneira: "*./bin/tp1.out < caminho/para/seu/arquivo_de_entrada.txt > arquivo_de_saida.txt*".

## 4. Análise de complexidade

Nesta seção, é apresentada a análise de complexidade de tempo e espaço para as principais funções implementadas e para o programa como um todo. Para essa análise, considere *N* como o número de objetos presentes no cenário.

| Função\Complexidade | Tempo | Espaço |
|---|---|---|
| ***AttPos*** | Essa função realiza um número constante de operações de atribuição e cálculos aritméticos para atualizar a posição de um objeto, independente do número total de objetos. Portanto, sua complexidade de tempo é Θ(1) | A função utiliza apenas o espaço do parâmetro recebido e não aloca memória adicional. Portanto, sua complexidade de espaço também é Θ(1) |
| ***AdicionarIntervaloOclusao*** | A principal operação nesta função é a reordenação do vetor de intervalos oclusos, que contém no máximo N elementos. A chamada à função *QuickSortIntervalo* domina a execução, resultando em uma complexidade de tempo de *O*(N logN). O passo de mesclagem dos intervalos, que vem em seguida, é executado em tempo linear, ou seja, *O*(N). | A função opera sobre o vetor estático *intervalo_ocluso* e não aloca memória adicional, tendo assim uma complexidade de Θ(1). |

| | | |
|---|---|---|
| ***GerarCena*** | Essa função é a mais custosa do programa. Ela itera sobre os N objetos do cenário. Dentro deste laço, para cada objeto, a função *AdicionarIntervaloOclusao* é chamada, com um custo de $O(N \log N)$. Além disso, a função *AddObjeto* percorre os intervalos de oclusão já existentes. Essa combinação de operações dentro do laço principal eleva a complexidade de tempo da função para aproximadamente $O(N^2 \log N)$. | A função utiliza os vetores estáticos *vetor_cena* e *intervalo_ocluso*, que possuem tamanho máximo pré-definido. Portanto, o espaço utilizado não cresce com a entrada, e sua complexidade é $\Theta(1)$. |

Em relação ao programa como um todo, e levando em consideração a tabela acima com o custo das principais funções, podemos afirmar que, independente da estratégia de ordenação adotada, o programa terá um custo $O(N^2 \log N)$ em relação a sua complexidade de tempo. Isso se deve ao fato de que a função *GerarCena* é a função mais custosa do programa e sempre é chamada ao menos uma vez durante a execução. Para a complexidade de espaço, temos que o custo será $\Theta(1)$, uma vez que, como os principais vetores utilizados ao longo do programa (*vetorObjs*, *vetorCena* e *intervalo_ocluso*) foram implementados estaticamente com tamanho máximo fixo, o programa não precisa alocar mais memória ao decorrer de sua execução.

## 5. Estratégias de Robustez

Para garantir a estabilidade e o funcionamento correto do programa, diversas estratégias de programação defensiva e tolerância a falhas foram implementadas, com foco em prevenir erros de tempo de execução e lidar com entradas inesperadas ou estados inválidos. As principais estratégias adotadas estão descritas abaixo:

**Verificação de ponteiros nulos:** antes de realizar operações de troca ou particionamento nos vetores de objetos e cenas, funções como *Troca*, *Particao*, *TrocaCena* e *ParticaoCena* verificam se os ponteiros recebidos são nulos. Caso um ponteiro nulo seja detectado, uma mensagem de erro é impressa e a função retorna prematuramente, evitando falhas de segmentação. A função *AttPos* também realiza uma checagem similar para evitar a desreferência de um ponteiro nulo.

**Validação do formato da entrada:** no laço principal do programa, o valor de retorno da função *sscanf* é verificado após a leitura de cada linha. Este valor indica o número de variáveis que foram atribuídas com sucesso. Se o número de itens lidos não corresponder ao esperado para o tipo de operação, o programa imprime um aviso, indicando uma linha malformada, e a ignora, prosseguindo para a próxima. Essa medida impede que o programa opere com dados corrompidos ou não inicializados, que poderiam surgir de um arquivo de entrada com erros de formatação.

**Validação de índices:** as funções de particionamento validam os índices validam os índices *inicio* e *fim* para garantir que estejam dentro dos limites válidos do vetor. Se os índices forem inválidos, um erro é reportado e a função retorna.

**Prevenção de overflow no buffer:** na função *AdicionarIntervaloOcluso*, antes de adicionar um novo intervalo, há uma verificação para garantir que o número de intervalos não exceda o tamanho máximo do vetor. Uma proteção similar existe na função *AddObjeto* para o vetor de cena. Essa abordagem impede a escrita de dados fora dos limites do array.

**Tratamento de estados inválidos:** o laço principal do programa foi projetado para ser tolerante a falhas. Ele ignora linhas em branco que possam existir na entrada. Adicionalmente, caso uma operação “M” ou “C” seja lida antes de qualquer objeto ter sido criado, o programa simplesmente continua para a próxima linha, evitando operações em um vetor vazio. Se um comando de movimento especificar um ID de objeto que não exista, um aviso é emitido e a operação é ignorada.

## 6. Análise Experimental

Esta seção apresenta a análise de desempenho de três estratégias distintas para a ordenação dos objetos na cena, com o objetivo de determinar a abordagem mais eficiente sob diferentes condições. A métrica de avaliação utilizada foi o tempo de execução total do programa, medido em segundos.

As três estratégias avaliadas foram:

1. **Ordenar na cena:** a ordenação dos objetos é realizada apenas no momento em que uma cena é solicitada. É uma abordagem que adia o custo da ordenação para o momento da renderização.
2. **Ordenar com limiar:** a ordenação é executada a cada inserção ou movimento de objeto mas somente se um determinado limiar de “desordem” for atingido. Esta abordagem busca um equilíbrio, evitando ordenações desnecessárias.

3. **Ordena sempre:** a ordenação é executada a cada operação de inserção ou movimento de um objeto. Esta é uma abordagem que mantém a estrutura de dados constantemente ordenada.

Foram conduzidos quatro experimentos principais, variando sistematicamente os parâmetros de entrada para avaliar o impacto na performance de cada estratégia. Para cada experimento foram gerados arquivos baseados nos arquivos de teste disponibilizados pelo moodle.

**Experimento 1: variação da largura dos objetos**

Neste experimento, o número de objetos (25) e a velocidade (10) foram mantidos constantes, enquanto a faixa de largura dos objetos foi variada. O objetivo era observar como a complexidade da sobreposição de objetos impactava o desempenho. Os resultados indicam que a variação da largura teve um impacto marginal no tempo de execução total para todas as estratégias. As abordagens "Ordena na Cena" e "Ordena com Limiar" apresentaram desempenho muito similar e superior à "Ordena Sempre", que consistentemente demonstrou ser a mais custosa devido às suas repetidas operações de ordenação.

**Experimento 2: variação de velocidade**

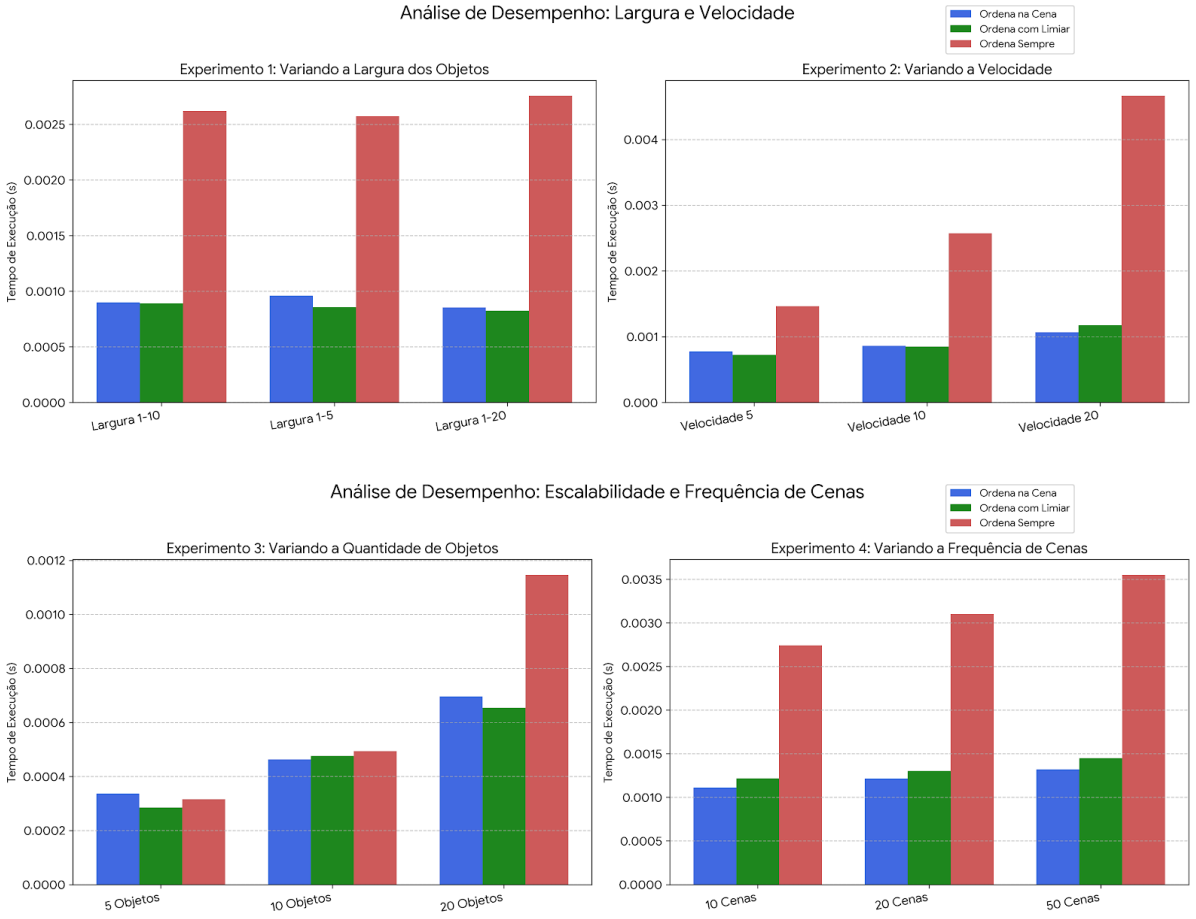
Mantendo o número de objetos (25) e a largura (1 a 10) fixos, a velocidade (quantidade de objetos mórbidos por unidade de tempo) foi alterada. Com o aumento da velocidade, a quantidade de operações de movimentos cresce, introduzindo mais desordem na estrutura de dados. Os resultados mostram que a estratégia "Ordena Sempre" é a mais penalizada pelo aumento da velocidade, pois o custo de reordenar a cada movimento se torna proibitivo. As estratégias "Ordena na Cena" e "Ordena com Limiar" se mostraram mais robustas a essa variação com uma vantagem de desempenho que se acentua conforme a velocidade aumenta.

**Experimento 3: variação da quantidade de objetos**

Este experimento avaliou a escalabilidade das soluções, variando o número de objetos (e ajustando a velocidade proporcionalmente). Como esperado, o tempo de execução aumentou para todas as estratégias como o crescimento do número de objetos. Novamente, "Ordena Sempre" e "Ordena com Limiar" mantiveram um desempenho próximo e significativamente mais eficiente, indicando que adiar ou otimizar a frequência da ordenação é crucial para lidar com um grande número de elementos.

**Experimento 4: variação a quantidade de cenas geradas**

Aqui, o número de objetos (25), a largura e a velocidade foram fixados, enquanto a frequência de geração de cenas foi aumentada. Este cenário teste o trade-off entre as abordagens. Quando poucas cenas são geradas, o custo da estratégia "Ordena na Cena" é minimizado. Conforme a frequência de geração de cenas aumenta, o custo dessa estratégia também cresce, pois a ordenação é executada mais vezes. No entanto, mesmo com um aumento no número de cenas, os dados mostram que "Ordena na Cena" e "Ordena com Limiar" permaneceram como as opções mais performáticas, superando a sobrecarga constante da estratégia "Ordena Sempre".




Os resultados demonstram de forma consistente que a estratégia "Ordena Sempre" é a menos eficiente em todos os cenários testados. O custo de manter o vetor de objetos constantemente ordenado supera em muito os benefícios.

As estratégias “Ordena na Cena” e “Ordena com Limiar” apresentaram desempenho muito similar e superior. A escolha entre as duas pode depender do caso de uso específico:

- **Ordena na Cena:** é ideal para cenários com baixa frequência de renderização e alta dinamicidade (muitos movimentos), pois minimiza o número de ordenações.
- **Ordena com Limiar:** representa um excelente equilíbrio, evitando ordenações em pequenas modificações e garantindo que a estrutura não fique excessivamente desordenada, o que poderia, em teoria, impactar o tempo de geração de uma cena individual.

Para o problema geral proposto, ambas as estratégias se mostraram robustas e eficientes.

## 7. Conclusão

Este trabalho lidou com o problema da identificação de objetos unidimensionais oclusos, no qual a abordagem utilizada para sua resolução foi a implementação de um sistema em linguagem C para processar, movimentar e renderizar cenas de objetos, com foco na análise de diferentes estratégias de ordenação para otimizar o desempenho. Com a solução adotada, foi possível verificar que adiar a operação de ordenação para o momento da geração de cena ou executá-la apenas quando um limiar de desordem é atingido são abordagens significativamente mais eficientes do que manter o vetor de objetos constantemente ordenado, especialmente em cenários com alta frequência de movimentação.

Por meio da resolução do trabalho, foi possível praticar conceitos fundamentais da disciplina de Estrutura de Dados. A implementação exigiu o aprofundamento no projeto e uso de Tipos Abstratos de Dados (TADs) para encapsular as informações de objetos e cenas, a aplicação de algoritmos de ordenação como o QuickSort e a análise formal de complexidade de tempo e espaço das soluções desenvolvidas. Além disso, a condução da análise experimental reforçou a importância de validar hipóteses teóricas com medições práticas de desempenho.

Durante a implementação da solução para o problema, houveram importantes desafios a serem superados. A lógica para identificar e mesclar os segmentos visíveis dos objetos, evitando a sobreposição incorreta, exigiu um planejamento cuidadoso do algoritmo de oclusão. Outro desafio foi o desenho de um experimento robusto, com a

criação de casos de teste que variam os parâmetros de forma controlada para permitir uma comparação justa entre as estratégias e a obtenção de conclusões válidas sobre suas vantagens e desvantagens.

## 8. Bibliografia

Cunha, W., Lacerda, A., Meira Jr., W., Santos, M. (2025). Slides virtuais da disciplina de estrutura de dados.

Disponibilizados via moodle. Departamento de Ciência da Computação.

Universidade Federal de Minas Gerais. Belo Horizonte.