Trabalho Prático 1
Identificação de Objetos Oclusos

Luísa Barros Ribeiro Andrade - 202303412
Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte - MG - Brasil
luisabra@ufmg.br

1 Introdução

O trabalho tem como objetivo melhorar o desempenho dos jogos da empresa Jolambs, aprimorando a apresentação dos objetos na tela, através da oclusão de objetos, para reduzir a complexidade das cenas, apresentando somente os segmentos visíveis dos objetos. Os objetos são segmentos de reta, a profundidade é definida pela coordenada Y e a oclusão é resolvida na dimensão horizontal, no eixo X, (Oclusão 1D).

Portanto, o problema principal é gerenciar a ordenação. O sistema processa comandos de Movimento (M), que alteram as posições dos objetos, desorganizando a ordem de profundidade, a geração da cena exige uma lista ordenada para o processo de oclusão correto.

Para solucionar o problema foi feita a Ordenação Condicional, que ordena somente quando necessário. A ordenação é realizada pelo Bubble Sort ($O(n^2)$). O processamento dos objetos em ordem de profundidade é feito pelo algoritmo de geração da cena, do mais próximo para o mais distante (menor Y para o maior Y), assim, ele calcula o quanto cada objeto é coberto pelos objetos já processados.

2 Método

O programa foi desenvolvido na linguagem C++, compilado pelo GCC da GNU Compiler Collection. A máquina utilizada possui as seguinte especificações:

- Sistema operacional: Ubuntu 24.04.2 LTS
- Compilador: gcc (Ubuntu 13.3.0-6ubuntu2 24.04) 13.3.0
- Processador: 11th Gen Intel(R) Core(TM) i5-1135G7 @ 2.40GHz
- RAM: 8Gb ((utilizável: 7,73 GB)

2.1 Estruturas de Dados

O projeto é modularizado em torno de dois **Tipos Abstratos de Dados** (TADs) centrais, implementados em `objeto.cpp` e `cena.cpp`, com a lógica de controle em `main.cpp`. Foram utilizados **Arrays Dinâmicos** para armazenamento das coleções e **Structs** para definir as entidades.

O TAD Objeto é um struct que define e armazena os atributos posição (x, y), a largura, o início e o fim de cada objeto, para oclusão 1D. A Coleção de Objetos é armazenada em `main.cpp` como um array dinâmico `Objeto*`. A inserção de novos objetos (pelo comando 'O') é feita por meio de redimensionamento por duplicação, garantindo que a alocação de memória seja eficiente com custo O(1), independente do volume de entrada.

O TAD Cena encapsula a lógica de geração e armazenamento de Segmentos, que é um *struct* que armazena o trecho visível. Ele é composto por um contador de segmentos na cena atual `(numSegmentos)`, a capacidade máxima da cena atual `(capacidade)` e por um *array* dinâmico `(Segmento*)` que armazena os resultados finais, nele é utilizado redimensionamento por duplicações assim como em `Objeto*`, garantindo a tolerância a falhas por capacidade, que se refere ao risco de *buffer overlow*, que seria o risco de erro por estouro de *buffer* por utilizar um *array* de tamanho fixo para armazenar uma quantidade variável de dados. O Segmento é a *struct* que define a saída no formato 'S', contendo *id*, tempo, início e fim do trecho visível.

2.2 Funções Implementadas

No TAD Objeto, as funções de manipulação são `criaObjeto`, que inicializa a estrutura, e `atualizaObjeto`, que modifica a posição do objeto em resposta do Movimento (comando 'M') e recalcula o inicio e o fim (bordas). A função essencial para ordenação é a função `ordenaPorY` em `objeto.cpp`, que implementa *Bubble Sort,* ordena o *array* de objetos por profundidade de Y, de maneira crescente, ou seja, do mais próximo (menor Y) para o mais distante (maior Y). é feita uma otimização importante dentro da função `ordenaPorY`, que é a *flag* `trocou`, que permite saída antecipada, o que reduz o custo para O(n) quando a lista está ordenada.

No TAD Cena é implementado o Algoritmo de Oclusão 1D na função `geraCena` em `cena.cpp`, que é a principal função implementada nesse TAD. O algoritmo processa os objetos em ordem crescente, para cada objeto ele mantém uma lista dos segmentos cobertos pelos objetos que estão à frente, usando *buffers* estáticos auxiliares, como `cobIni` e `cobFim`, ou seja, ele calcula as partes visíveis de cada objeto a partir das áreas cobertas por objetos anteriores. O ponto crítico da oclusão é União de Intervalos, onde é feita uma ordenação interna dos segmentos cobertos, por meio de *Bublle Sort*, e uma mesclagem linear para consolidar a área oclusa, então, com base na área coberta definida, os segmentos visíveis do objeto atual são calculados e adicionados à Cena.

Além disso, é implementada também a função `adicionaSegmento`, no TAD Cena, que é essencial para a robustez, é responsável por verificar se o número de segmentos excede a capacidade atual, e se sim, aplica o redimensionamento por duplicação, evitando falhas por estouro de capacidade. Após a impressão da cena, o contador `numSegmentos` é zerado, o que permite a reutilização eficiente da memória, que já é alocada para próxima cena. Após a geração de segmentos visíveis, é implementado a função `ordenaSegmentosPorId` em que é utilizado *Selection Sort* para organizar a saída final.

2.3 Lógica de Controle

No arquivo `main.cpp` é feito o gerenciamento do fluxo de entrada e a heurística de desempenho. A cada movimentação (comando 'M'), ele adiciona `atualizaObjeto` e incrementa o contador de desorganização e ativa a *flag* booleana `ordenacao_necessaria` para sinalizar que, desde o último estado ordenado, ocorreu pelo menos uma mudança que exige reavaliação da ordem.

O uso da Reordenação Condicional se dá quando o módulo de controle avalia as condições de ordenação, após o de Cena ('C') é lido, o *Bubble Sort* (`ordenaPorY`) só é executado se o contador de desorganização foi maior ou igual ao limiar de ordenação (`desorganizacao_couter >= LIMIAR_DE_ORDENACAO`). Ou seja, a ordenação é feita somente quando necessário. A validação experimental dessa heurística prova que essa é um abordagem mais eficiente, pois evita o custo O(n^2) para cada movimento e garante que a função `geraCena` opere sobre dados corretos ou em estado de desorganização mínima.

2.4 Ambiente de compilação

O projeto utiliza um arquivo utilitário `Makefile` para gerenciar a compilação. O arquivo define as regras de construção para o executável principal (`tp1.out`), garantindo que o código fonte seja compilado de forma modular e eficiente. O processo compila os arquivos de código-fonte separadamente, gera arquivos objeto, por fim, os conecta no executável final. Por fim, o arquivo `makefile` fornece o comando `clean` para remover os arquivos gerados, mantendo o ambiente de desenvolvimento limpo.

3 Análise de Complexidade

A análise de complexidade avalia o desempenho do sistema em termos do número de objetos (n), número total de comandos na entrada (m) e o número de segmentos visíveis (s, onde s $\leq$ n).

3.1 Complexidade de Tempo

- **Complexidade das Funções de Ordenação e Oclusão:**

  A função `ordenaPorY` responsável por ordenar os objetos, possui um custo O(n^2) no pior caso, visto que, é feito a partir de *Bubble Sort*. Essa é a operação que Reordenação Condicional busca compensar o custo.

  A função `geraCena` responsável pela geração da Cena, possui custo O(n^2). O algoritmo itera O(n) vezes sobre os objetos. A interação interna mais cara é a União de Intervalos, que realiza uma ordenação interna sobre C (número constante de) segmentos cobertos, então como C pode ser até n, o custo é de O(n^2). Assim, temos $O(n) \times O(n^2) \rightarrow O(n^3)$, mas como a maior parte do projeto

engloba operações de complexidade $O(n^2)$, muitas vezes, a complexidade $O(n^3)$ não é atingida. Por isso, podemos afirmar que a complexidade é limitada no caso médio de $O(n^2)$.

A função `ordenaSegmentosPorId` possui custo $O(s^2)$ (ou $O(n^2)$). O *Selection Sort* é executado sobre os s segmentos gerados para formatar a saída.

- **Análise do Custo Médio por Operação:**

  A função `adicionaSegmento` possui custo O(1) (constante). A adição d um segmento, na maioria das vezes é O(1). O custo O(s) ocorre quando o *array* precisa ser duplicados, mas a técnica de redimensionamento por Duplicação garante que o custo médio por operação de adição seja contante.

  A função `criaObjeto` também possui o custo O(1), amortizado pela mesmo razão, aplicado ao *array* de objetos.

- **Complexidade total:**

  A complexidade total do programa é dada pela soma dos custos de todos os m comandos lidos:

$$T_{total} = O(m \cdot n^2)$$

  O projeto é bem-sucedido na otimização ao reduzir a frequência das operações $O(n^2)$, na ordenação de objetos, minimizando o coeficiente associado ao termo $(m \cdot n^2)$ ao longo da execução.

3.2 Complexidade de Espaço

A complexidade de espaço é avaliada pela memória alocada dinamicamente em relação ao número de objetos n. Ela é limitada pela necessidade de armazenar objetos e os segmentos. Temos que a complexidade da Coleção de Objetos é O(n), da Lista de Segmentos é O(s) (ou O(n)) e as variáveis auxiliares/*buffers* é O(1), devido aos *arrays* de tamanho fixo. Portanto, temos que a complexidade de espaço do programa é **O(n).**

4 Estratégias de Robustez

O projeto foi construído com mecanismos robustos para garantir que ele funcione de maneira segura e eficiente mesmo diante de entradas inesperados, ou com grandes volumes de dados, garantindo a integridade do sistema e a segurança da memória.

- **Gerenciamento da memória dinâmica:** para prevenir *memory leaks* (vazamento de memória), e para garantir a correta liberação dos recursos alocados na *heap* ao final da execução. Esse gerenciamento é implementado em `main.cpp`, com o uso correto de alocação dinâmica, ou seja, com a alocação de array `(new[])` é sempre pareado com a desalocação correta de arrays `(delete[])`.

- **Tolerância de falhas por capacidade:** é uma estratégia de evitar falhas por estouro de *buffer (buffer overlow)* em tempo de execução, permitindo que o sistema lide com qualquer volume de entrada e saída (*inputs* e *outputs*). Isso é a possível a partir da técnica de Redimensionamento por Duplicação, que é usada sempre que o programa tenta adicionar um novo elemento a um *array* dinâmico que já atingiu sua capacidade máxima. Isso se dá da seguinte forma, quando a capacidade máxima de um array é atingida, um array novo é alocado, os elementos são copiados do array antigo para o novo, a memória do antigo é libera, o ponteiro é atualizado para o novo bloco. Isso garante que a inserção tenha um custo constante O(1).

- **Garantia da pré-condição na ordenação:** assegura que a função de oclusão `(geraCena)` sempre seja chamada somente quando a lista de objetos está ordenada por profundidade, ou seja, seguindo essa pré-condição é garantido que o resultado seja correto, já que o algoritmo de oclusão 1D só funciona se a lista tiver ordenada.

- **Otimização do Bubble Sort:** reduz o custo $O(n^2)$ da ordenação para O(n) quando o array já está quase ordenado, utilizando programação defensiva para o desempenho. A função `ordenaPorY` usa a instrução `if(!trocou) break;` para encerrar o algoritmo prematuramente, indicando que a lista já está ordenada, diminuindo a complexidade do programa.

- **Verificação de input vazio e limpeza de estado:** previne *runtime errors* e garante a reutilização eficiente de memória. Isso previve a execução do algoritmo `geraCena` em um array vazio, e o reset do estado permite que o TAD Cena reutilize a memória já alocada.

5 Análise Experimental

O principal objetivo na análise experimental é avaliar a eficácia do método de Reordenação Condicional. O experimento busca determinar um limiar de ordenação que minimiza o tempo de execução total, equilibrando o custo da ordenação O(n^2) com a frequência de movimentos.

A metodologia do experimento consiste na instrumentação de métricas dentro do arquivo main.cpp para obter dados brutos, que foram analisados posteriormente em arquivo de *notebook Jupyter*. Os parâmetros de coleta são:

- **Limiar:** a variável de controle, lida via argumento de linha de comando (`argv[1]`)
- **num_obj:** tamanho de entrada (número de objetos(n))
- **ordenações:** contagem bruta de chamadas ás função `ordenaPorY`.
- **cenas:** contagem de chamadas da função `geraCena`.
- **tempo:** tempo de execução total do programa
- **tempo_sort:** tempo gasto somente dentro das funções de ordenção.
- **memoria:** custo de memória do projeto.

Assim, o processamento de dados é feito em um arquivo *notebook Jupyter*, usando linguagem *Python*, que é responsável por transformar os dados brutos em dados estatisticamente válidos. O processo detalhados consiste nos seguintes passos:

- **Carregamento e limpeza**: o arquivo `resultado.csv` é carregado para gravar os dados coletados a partir da execução do programa com diferentes valores de entrada, que foram disponibilizados para teste no enunciado do TP (`inputs1.txt`, ..., `inputs5.txt`), variando o número de objetos por entrada e o limiar de ordenação.
- **Agregação e tratamento estatístico**: os dados foram agrupados pelas chaves `limiar` e `num_obj`. Para cada grupo é calculado a média e o desvio padrão de todas as métricas de tempo e contagem. A média minimiza as distorções causadas pelo sistema operacional nas medições de tempo, e o desvio padrão quantifica a dispersão dos dados, evidenciada pelas barras de erro.
- **Visualização**: O *notebook* plotou gráficos, exibindo as médias do tempo em função do limiar de ordenação para cada tamanho de entrada (n), permitindo a identificação visual robusta do ponto ótimo de desempenho.

5.1 Análise do Trade-off e ponto ótimo de desempenho

Nesse caso o trade-off consiste na escolha entre frequência de ordenação e custo total de execução.

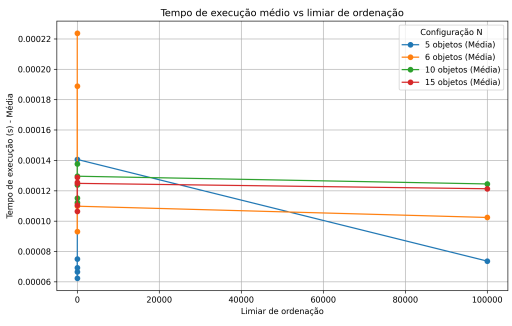

Figura 1: Tempo médio gasto na execução por limiar

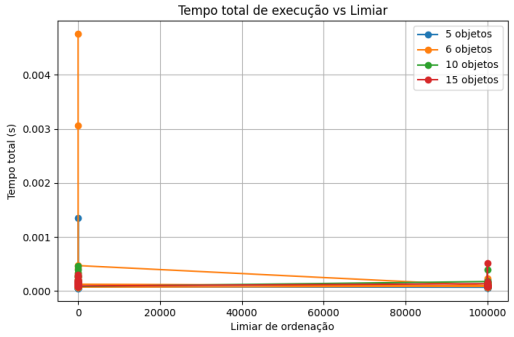

Figura 2: Tempo exato gasto na execução por limiar

O ponto de referência (Limiar = 0) demonstra ter o pior tempo de execução, o que confirma que a execução recorrente da operação O(n^2) é o que gera o custo alto do sistema, o que justifica a necessidade da reordenação condicional, já que diminui o número de operações.

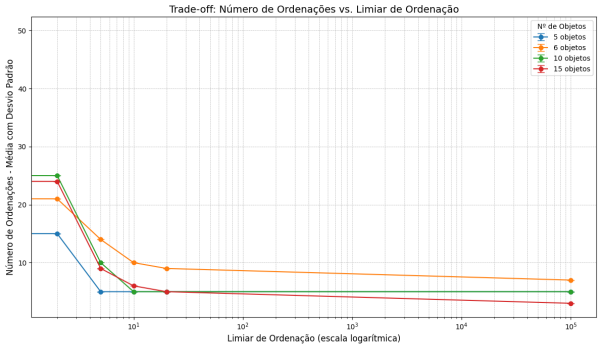

Figura 3: Trade-off entre número de ordenações e limiar (escala logarítmica)

Esses gráficos mostram que o número de ordenações (eixo Y) e o tempo de execução variam junto com o limiar. A fase de declínio, entre o limiar de 0 até 2, mostra que, o tempo de execução diminui drasticamente, o que ocorre porque o sistema passa da abordagem ineficiente para reordenação condicional. A fase de aumento, com o limiar > 5, ou após o ponto ótimo, o tempo de execução começa a aumentar novamente, o sistema continua evitando a ordenação, mas o tempo gasto nas outras operações quadráticas (`geraCena` e `ordenaSegmentosPorId`) torna-se fator limitante.

O Ponto de Eficiência Ótimo, quando apresenta o menor tempo, ocorre para no intervalo Limiar ≈ 2 a 5, ou seja nesse intervalo é quando a economia obtida ao evitar o Bubble Sort $O(n^2)$ é maximizada, sem que o prejuízo causado pela desorganização se torne dominante.

5.2 Análise da estrutura de custo de memória

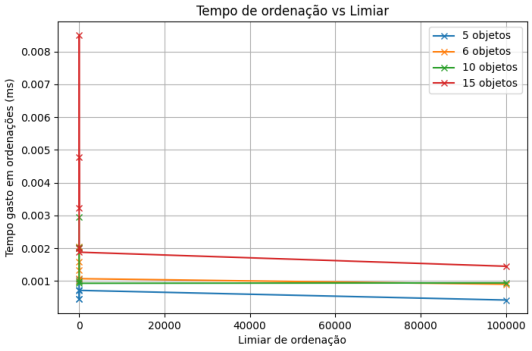

Figura 4: Tempo gasto somente na ordenação

A partir do gráfico 4 é possível ver que o tempo gasto somente na ordenação (`Tsort`) é uma fração muito pequena do tempo total de execução, principalmente para limiares altos. Isso mostra que os custo das funções de ordenação é um fator limitante, que determina o custo de execução.

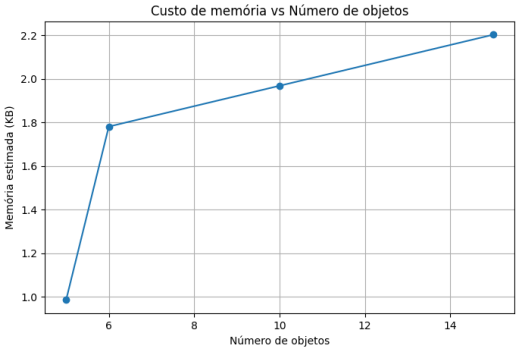

Figura 5: Custo de memória pelo custo de objetos

O gráfico 5 mostra o custo de memória utilizadas no projeto e prova a robustez da implementação. O custo de memória é constante, limitado a O(n) e não varia em função do limiar de ordenação, o que valida a técnica de redimensionamento por duplicação.

5.3 Conclusão dos experimentos

A análise prova a hipótese inicial de que a estratégia de Reordenação Condicional é vital para a eficiência do sistema. O sucesso reside na identificação do ponto ótimo (limiar $\approx$ 2 a 5) onde o custo é O(n^2) da ordenação é evitado em comparação com o ponto de referência (limiar = 0).

6 Conclusão

O projeto implementou um sistema de gerenciamento de cenas e oclusão de objetos, atendendo os requisitos de otimização das imagens dos jogos da empresa Jolambs.

Foi feita a validação experimental da heurística de reordenação condicional, que provou ser uma estratégia crucial para evitar o aumento do custo desnecessário, melhorando o custo de ordenação em O(n^2) em ambiente dinâmicos. Além disso, foi possível identificar o ponto de ótimo de desempenho dentro do intervalo limiar $\approx$ 2 a 5, onde o custo de ordenação é amortizado sem que a eficiência da oclusão seja comprometida. Isso mostra que a otimização de desempenho não se resume a escolher o melhor algoritmo de ordenação, mas controlar a frequência de execução do programa, visto que, mesmo com um algoritmo não tão eficiente (Bubble Sort) foi possível ter um bom desempenho.

Em termos de arquitetura o projeto possui estratégias concretas para robustez, implementando os TAD Objeto e Cena com garantia de correção lógica e segurança da memória com técnica de redimensionamento por duplicação garantindo que o custo de memória fosse eficiente, mantendo em O(1) (amortizado). Além disso, a análise de complexidade confirmou que o custo total do projeto é limitado por O($m \cdot n^2$).

Por fim, é possível concluir que o trabalho alcançou o objetivo de performance, fornecendo um sistema que não apenas calcula a oclusão de maneira correta e eficiente, como também resulta em uma base sólida para futura renderização dos jogos da empresa Jolambs.

Referências

1 Anisio Lacerda, Marcio Santos, Wagner Meira Jr., Washintgton Cunha **Slides virtuais da disciplina de estruturas de dados**. 2025. Material didático (Slides de aula). Disponibilizado via Moodle. Departamento de Ciência da Computação, Universidade Federal de Minas Gerais, Belo Horizonte. Acesso restrito.

2 ZIVIANI, N. **Projeto de Algoritmos com Implementações em Pascal e C**. 3ª Edição. Cengage Learning, 2011.

3 CORMEN, T.; LEISERSON, C.; RIVEST, R.; STEIN, C. **Introduction to Algorithms**. Third Edition. MIT Press, 2009. Versão Traduzida: *Algoritmos – Teoria e Prática*. 3a. Edição. Elsevier, 2012.