Trabalho Prático 1 – Identificação de Objetos Oclusos

Bernardo Zschaber Morato Nogueira

[1]Departamento de Ciência da Computação – Universidade Federal de Minas Gerais (UFMG) Belo Horizonte – MG – Brazil

bernardozschaber@ufmg.br

1 Capa

1 Capa 1
2 Introdução 2
3 Método 2
3.1 Organização do código 2
3.2 Estruturas de dados 2
3.3 Tipos abstratos de dados 2
3.4 Funções principais 3
3.5 Configuração de teste 3
4 Análise de complexidade 3
4.1 Análise de tempo 3
4.1.1 Etapas do processo 3
4.1.2 Função checkIfDisorganized 4
4.1.3 Função sortObjectsByY 4
4.1.4 Função generateScene 4
4.1.5 Função prepareScene 4
4.1.6 Complexidade total de tempo do sistema 4
4.2 Análise de espaço 5
4.2.1 Arrays principais 5
4.2.2 Estruturas internas do TAD Scene 5
4.2.3 Variáveis auxiliares 5
4.2.4 Métricas experimentais 5
4.2.5 Complexidade total de espaço 5
4.3 Conclusão 5
5 Estratégias de robustez 6
5.1 Validação de entradas 6
5.2 Tratamento de erros com exceções 6
5.3 Programação defensiva 6
5.4 Gerenciamento de memória 6
5.5 Consistência de dados 6
6 Análise experimental 7
6.1 Experimento 1: Crescimento do tempo de geração de cenas 7
6.1.1 Objetivo 7
6.1.2 Configuração 7
6.1.3 Resultados 7
6.1.4 Conclusão 8
6.2 Experimento 2: Impacto de movimentos e objetos no tempo de ordenação 8
6.2.1 Objetivo 8
6.2.2 Configuração 8
6.2.3 Resultados 8
6.2.4 Conclusão 9
6.3 Experimento 3: Crescimento dramático da complexidade das cenas 9
6.3.1 Objetivo 9
6.3.2 Configuração 9
6.3.3 Resultados 9
6.3.4 Conclusão 10
7 Conclusões 10
8 Bibliografia 10

2 Introdução

A renderização eficiente de cenas é um desafio fundamental no desenvolvimento de jogos, especialmente quando há muitos objetos na tela. A empresa Jolambs busca melhorar o desempenho de seus jogos através da otimização do algoritmo de apresentação de objetos, explorando a oclusão para reduzir a complexidade das cenas. O problema consiste em identificar quais objetos (ou partes deles) são visíveis em determinado momento, descartando aqueles que estão completamente ocultos por outros objetos mais próximos ao observador.

O sistema deve processar objetos unidimensionais representados por segmentos paralelos ao eixo X, cada um com posição, largura e profundidade (coordenada Y). Os objetos podem se mover ao longo do tempo, e o sistema deve gerar cenas que mostrem apenas os segmentos visíveis, considerando as oclusões causadas pela sobreposição em profundidade. Este é um problema clássico de geometria computacional aplicada a jogos, onde a eficiência do algoritmo impacta diretamente na performance do sistema.

Como solução, foi desenvolvido um programa em C++ que implementa dois Tipos Abstratos de Dados (TADs): Object, para gerenciar os objetos e suas propriedades, e Scene, para construir as cenas identificando segmentos visíveis. O programa utiliza alocação estática de memória e uma estratégia de ordenação sob demanda, onde o vetor de objetos é ordenado por profundidade apenas no momento da geração de cada cena. Esta abordagem foi escolhida após análise do compromisso entre frequência de ordenação e custo computacional, sendo validada através de experimentos que variam parâmetros como número de objetos, largura e densidade espacial.

3 Método

3.1 Organização do código

O projeto está organizado em três módulos principais: Object, Scene e main. Cada módulo possui arquivos de interface (.hpp) e implementação (.cpp), seguindo princípios de encapsulamento. Os arquivos principais são object.hpp/cpp, scene.hpp/cpp e main.cpp. Reitera-se a organização da estrutura do projeto conforme presente na pasta raiz do projeto: pasta src armazena os arquivos de código scene.cpp e object.cpp, include contém os headers do projeto object.hpp e scene.hpp, enquanto que bin abriga o executável tp1.out e no diretório obj estão os arquivos *.o gerados pelo Makefile.

3.2 Estruturas de dados

O sistema utiliza alocação estática de memória com arrays de tamanho fixo definido pela constante MAX_TAM = 20000. Esta abordagem garante contiguidade dos dados em memória e melhor localidade de referência, aspectos fundamentais para a análise experimental realizada.

Três estruturas auxiliares suportam o sistema: Movement (armazena tempo, identificador e nova posição de movimentações), SceneSegment (representa segmentos visíveis com identificador do objeto, início e fim) e ExperimentalMetrics (coleta métricas de desempenho como contadores de movimentos, ordenações e tempos de execução).

3.3 Tipos abstratos de dados

A solução implementa dois TADs que encapsulam as entidades principais do problema e suas operações:

TAD Object: Encapsula as propriedades de um objeto unidimensional (identificador, posição x e y, largura). Fornece métodos de consulta (getters) e atualização de posição. A coordenada Y representa a profundidade e é utilizada para determinar oclusões.

TAD Scene: Responsável pela geração de cenas identificando segmentos visíveis. Mantém internamente um array estático de segmentos e o tempo da cena. Suas principais funcionalidades incluem geração de cenas (generateScene), aplicação de movimentos (applyMovements), ordenação de objetos por profundidade (sortObjectsByY) e salvamento da cena em formato texto (saveScene).

3.4 Funções principais

As funções abaixo implementam a lógica central do sistema, desde a verificação de ordenação até a geração completa das cenas:

checkIfDisorganized: verifica se o vetor de objetos mantém a ordenação por coordenada Y, percorrendo o array e identificando pares adjacentes fora de ordem.

sortObjectsByY: ordena o vetor de objetos pela coordenada Y utilizando o algoritmo Bubble Sort, do mais próximo (menor Y) ao mais distante (maior Y).

generateScene: implementa o algoritmo central do sistema. Primeiro ordena os objetos por profundidade, depois calcula para cada objeto seus intervalos visíveis através de uma abordagem de subtração de intervalos: inicia-se com o intervalo completo do objeto e remove iterativamente as partes oclusas por segmentos já processados (objetos mais próximos), garantindo que apenas porções visíveis sejam adicionadas à cena.

O programa principal coordena o fluxo de execução: lê a entrada completa (objetos, movimentos e comandos de cena), armazena os dados em arrays estáticos, processa as cenas através do método prepareScene e gera a saída formatada.

3.5 Configuração de teste

O programa foi desenvolvido e testado na seguinte configuração:

- Sistema Operacional: Linux Debian 13
- Linguagem de Programação: C++
- Compilador: g++ com flags -std=c++11 -Wall -O2
- Processador: Intel Core i5-13450HX (2.40 GHz, 10 cores / 16 threads)
- Memória RAM: 16,0 GB DDR5 4800 MHz

4 Análise de Complexidade

A análise de complexidade considera as principais funções implementadas no sistema, avaliando tanto o custo temporal quanto espacial das operações. A seguir, detalhamos essas complexidades, separadamente.

4.1 Análise de Tempo

4.1.1 Etapas do Processo

O processamento de cenas no sistema envolve as seguintes etapas principais:

1. Verificação de desorganização: A função checkIfDisorganized percorre o vetor linearmente, verificando a ordenação em $O(n)$.
2. Ordenação por profundidade: A função sortObjectsByY utiliza Bubble Sort para ordenar os objetos, com custo $\Theta(n^2)$.
3. Aplicação de movimentos: Para cada movimento, busca-se o objeto correspondente no vetor, resultando em $O(m \cdot n)$ no pior caso.
4. Cálculo de intervalos visíveis: Para cada objeto, são calculados os intervalos não oclusos através da subtração iterativa com segmentos já processados, custando $O(n \cdot s \cdot i)$.
5. Salvamento da cena: Os segmentos são ordenados por ID utilizando Bubble Sort, com custo $O(s^2)$.

4.1.2 Função checkIfDisorganized

Esta função percorre o vetor de objetos uma única vez, comparando pares adjacentes para verificar a ordenação por coordenada Y. A operação de comparação é executada em tempo constante O(1), e o laço percorre até n-1 posições no pior caso, onde n é o número de objetos. Portanto, a complexidade assintótica de tempo é **$\Theta(n)$.**

4.1.3 Função sortObjectsByY

A função utiliza o algoritmo Bubble Sort para ordenar o vetor de objetos por profundidade. O algoritmo possui dois laços aninhados: o externo executa n-1 iterações e o interno executa até n-i-1 comparações e trocas. No pior caso (vetor totalmente invertido), são realizadas n(n-1)/2 comparações e trocas, resultando em complexidade assintótica de tempo **$\Theta(n^2)$**.

4.1.4 Função generateScene

A função responsável pela geração de cenas é a mais complexa do sistema. A execução dela pode ser fragmentada em três etapas principais:

1. Ordenação: Chama sortObjectsByY, com custo $\Theta(n^2)$.

2. Cálculo de intervalos visíveis: Para cada um dos n objetos, percorre os segmentos já processados (até s segmentos) e, para cada segmento, processa os intervalos atuais (até i intervalos). No pior caso, cada objeto gera múltiplos segmentos, mas o número total de segmentos na cena é limitado. A complexidade desta etapa é **$O(n \cdot s \cdot i)$**, onde s e i são limitados pelo número de objetos e pela fragmentação dos intervalos.

3. Adição de segmentos: Operação de inserção em array é O(1) por segmento.

   Portanto, a complexidade assintótica total da função é dominada pela ordenação e pelo processamento de intervalos, resultando em **$\Theta(n^2 + n \cdot s \cdot i)$**. Em cenários típicos onde a fragmentação é limitada, a complexidade pode ser aproximada por: **$\Theta(n^2)$**.

4.1.5 Função prepareScene

Tal coordena o processamento de múltiplas cenas. Para cada uma das c cenas solicitadas, executa:

1. Aplicação de movimentos: Percorre até m movimentos e, para cada movimento aplicável, busca o objeto correspondente em tempo O(n). Esta etapa tem custo **$O(m \cdot n)$** por cena.

2. Geração da cena: Chama generateScene com custo $\Theta(n^2)$.

3. Salvamento: A ordenação dos segmentos por ID utiliza Bubble Sort com custo $O(s^2)$, onde s é o número de segmentos.

Considerando c cenas, a complexidade assintótica total é **$\Theta(c \cdot (m \cdot n + n^2 + s^2))$**.

4.1.6 Complexidade Total de Tempo do Sistema

O programa principal realiza a leitura da entrada em tempo linear O(n + m + c) e em seguida chama prepareScene. A complexidade assintótica de tempo do sistema completo é dominada pelo processamento das cenas:

$$\Theta(c \cdot (m \cdot n + n^2))$$

Em cenários onde o número de movimentos m é muito maior que n, o custo pode ser aproximado por:

$$\Theta(c \cdot m \cdot n).$$

4.2 Análise de Espaço

4.2.1 Arrays Principais

A solução utiliza três arrays estáticos principais:

1. Array de objetos: armazena até MAX_TAM objetos, cada um contendo 4 campos (id, x, y, width). Espaço: O(MAX_TAM).
2. Array de movimentos: armazena até MAX_TAM movimentos, cada um com 4 campos (time, obj_id, x, y). Espaço: O(MAX_TAM).
3. Array de cenas: armazena até MAX_TAM instâncias de Scene. Espaço: O(MAX_TAM).

4.2.2 Estruturas Internas do TAD Scene

Cada instância de Scene mantém:

- Array de segmentos: array estático com capacidade para MAX_TAM segmentos, cada um contendo 3 campos (obj_id, start, end). Espaço: O(MAX_TAM) por cena.
- Variáveis auxiliares: contador de segmentos e tempo da cena ocupam espaço constante O(1).

4.2.3 Variáveis Auxiliares

Durante a execução de generateScene, são utilizados:

- Array de intervalos: dois arrays bidimensionais locais de tamanho fixo 100×2 para armazenar intervalos temporários durante o cálculo de oclusões. Espaço: **O(1)** (tamanho fixo).
- Variáveis de controle: contadores e índices ocupam espaço constante O(1).

4.2.4 Métricas Experimentais

A estrutura ExperimentalMetrics é estática e compartilhada, ocupando espaço constante independente do número de objetos ou cenas: **O(1)**.

4.2.5 Complexidade Total de Espaço

Para calcular a complexidade total de espaço, soma-se as principais estruturas, onde c é o número de cenas processadas:

$$O(MAX\_TAM) + O(MAX\_TAM) + O(c \cdot MAX\_TAM)$$

Considerando que MAX_TAM = 20000 é uma constante fixa do sistema, e assumindo que c < MAX_TAM, a complexidade espacial total é:

$$O(MAX\_TAM) = O(1) \text{ em relação aos parâmetros de entrada } (n, m, c).$$

Entretanto, se considerarmos MAX_TAM como parâmetro variável, a saber, em um caso em que é permitida a alocação dinâmica de memória, a complexidade espacial é dada por: $\Theta(MAX\_TAM \cdot c)$.

4.3 Conclusão

A análise confirma que o algoritmo possui complexidade de tempo $\Theta(c \cdot (m \cdot n + n^2))$, dominada pela ordenação quadrática e pela aplicação de movimentos em múltiplas cenas. A complexidade espacial é $O(MAX\_TAM \cdot c)$, com alocação estática garantindo previsibilidade no uso de memória. A escolha por alocação estática favorece a localidade de referência, embora imponha limites fixos de capacidade. O uso do Bubble Sort, apesar da complexidade quadrática, é adequado para os tamanhos de entrada testados e simplifica a implementação.

5 Estratégias de Robustez

Para garantir a confiabilidade e estabilidade do sistema de geração de cenas, diversas estratégias de programação defensiva foram implementadas:

5.1 Validação de Entradas

O programa verifica continuamente o formato dos dados de entrada durante a leitura do arquivo. Cada linha é processada através de um stringstream que valida a presença dos campos esperados. Linhas vazias ou malformadas são automaticamente ignoradas através da verificação if(!(ss >> type)) continue,o que possibilita o programa prosseguir sem interrupções.

A validação também verifica a semântica dos dados: tempos de cena devem ser não-negativos, e tipos de comando devem ser exclusivamente 'O', 'M' ou 'C'. Entradas que violam essas restrições geram exceções específicas que são tratadas adequadamente.

5.2 Tratamento de Erros com Exceções

Blocos try-catch foram implementados no módulo de leitura para capturar e tratar variados erros:

- overflow_error: lançada quando qualquer dos vetores estáticos atinge o limite MAX_TAM. Ao capturar esta exceção, o programa exibe uma mensagem clara ao usuário e interrompe a leitura, prevenindo acessos fora dos limites dos arrays.
- invalid_argument: gerada quando valores inválidos são detectados, como tempos negativos em comandos de movimento ou cena. O tratamento permite que o programa continue processando as próximas linhas.
- runtime_error: utilizada para tipos de comando não reconhecidos. A captura desta exceção registra o erro e prossegue com a leitura, tornando o sistema tolerante a falhas pontuais no formato de entrada.

5.3 Programação Defensiva

A implementação incorpora verificações de limites antes de todas as operações críticas:

i. Verificação de capacidade: antes de inserir novos elementos nos vetores, o sistema verifica se há espaço disponível comparando os índices atuais com MAX_TAM. Esta verificação previne buffer overflow e garante que o programa opere dentro dos limites estabelecidos.

ii. Validação de índices: funções como addSegment em scene.cpp verificam se numSegments < MAX_TAM antes de adicionar novos segmentos, emitindo uma mensagem de erro (“Erro: Limite de segmentos atingido”) caso o limite seja excedido.

iii. Tratamento de casos extremos: o código considera cenários como vetores vazios, objetos únicos e ausência de movimentos, garantindo comportamento correto mesmo em entradas mínimas.

5.4 Gerenciamento de Memória

Embora o sistema utilize alocação estática para os arrays principais, o gerenciamento cuidadoso de memória foi implementado nos TADs:

- Construtores e destrutores: os TADs Object e Scene implementam construtores de cópia, operadores de atribuição e destrutores apropriados, seguindo a regra dos três em C++.
- Inicialização adequada: todos os contadores e ponteiros são inicializados nos construtores, evitando uso de memória não inicializada.
- Ausência de vazamentos: como a alocação é totalmente estática, não há chamadas para new ou malloc nos arrays principais, eliminando o risco de memory leaks relacionados à alocação dinâmica.

5.5 Consistência de Dados

O sistema mantém invariantes importantes durante toda a execução:

- Sincronização de índices: Os índices objIdx, movIdx e sceneIdx são incrementados atomicamente após cada inserção bem-sucedida, garantindo consistência entre o número de elementos e seus índices.
- Ordenação preservada: Embora o vetor de objetos possa ficar temporariamente desorganizado após movimentos, a função checkIfDisorganized registra essas ocorrências, e a ordenação é restaurada antes de cada geração de cena.
- Validação de segmentos: Apenas intervalos válidos (onde end > start) são adicionados às cenas, prevenindo segmentos degenerados.

As estratégias supracitadas tornam o programa robusto e resiliente, permitindo a detecção e tratamento apropriado de falhas, além de garantir a consistência dos dados e o uso seguro dos recursos alocados estaticamente.

6 Análise Experimental

Para avaliar o desempenho do sistema de geração de cenas, foram realizados três experimentos controlados que isolam diferentes parâmetros do algoritmo. Todos os experimentos utilizaram alocação estática de memória e a estratégia de limiar infinito (ordenação apenas durante a geração de cenas).

6.1 Experimento 1: Crescimento do Tempo de Geração de Cenas

6.1.1 Objetivo

Avaliar como o tempo de geração de cenas (sceneGenerationTime) cresce em função do número de objetos, mantendo outros parâmetros constantes para isolar o impacto da quantidade de objetos no desempenho do algoritmo.

6.1.2 Configuração

Os testes foram realizados com 100 casos, de acordo com os seguintes parâmetros:

- Objetos: 1 até 5000 (progressão linear)
- Movimentos: 10000 (fixo)
- Cenas: 100 (fixo, uma a cada 100 movimentos)
- Largura: 80-100 (fixo, ordem de grandeza 100)
- Espaço: 100×100 (fixo)

6.1.3 Resultados

O gráfico da Figura 1 apresenta o tempo experimental de geração de cenas (em azul) com uma curva de tendência (em roxo/laranja). Os resultados demonstram um crescimento claramente não-linear, confirmando o comportamento esperado pela análise teórica de complexidade $\Theta(n^2)$ do algoritmo Bubble Sort utilizado na ordenação.

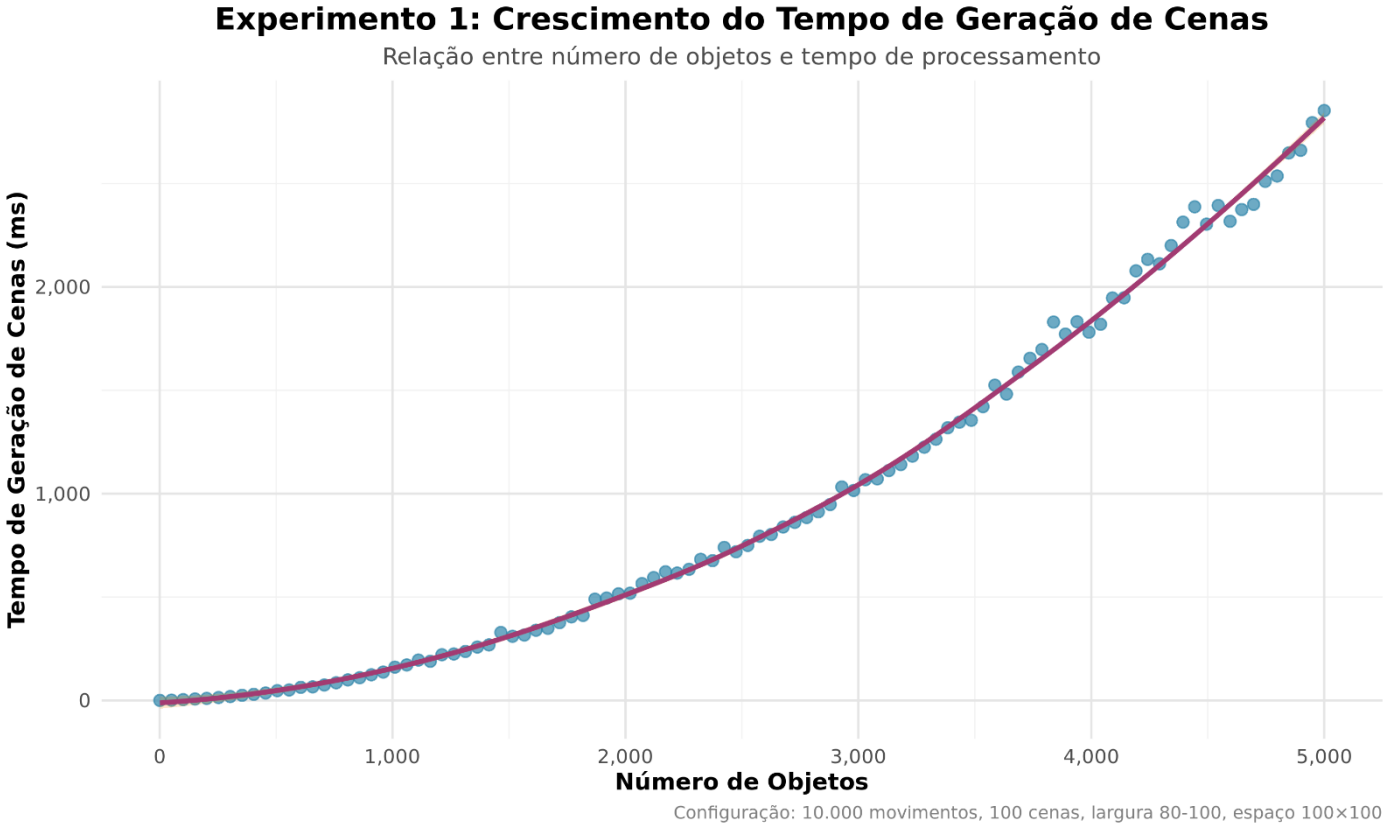

Figura 1: Tempo de geração de cenas (ms) em função do número de objetos

Observações principais:

- Para 1 objeto: tempo próximo de zero (~0.008 ms)
- Para 5000 objetos: tempo de aproximadamente 2853 ms (~2.85 segundos)
- Crescimento aproximadamente quadrático, conforme esperado.

O tempo de geração de cenas é dominado pela ordenação do vetor de objetos por profundidade (coordenada Y), que ocorre uma vez por cena gerada. Com 100 cenas e crescimento quadrático do tempo de ordenação, observa-se que o custo total segue a relação $sceneGenerationTime \propto n^2 \times numCenas$.

6.1.4 Conclusão

Os resultados confirmam que o algoritmo segue a complexidade teórica esperada de $\Theta(n^2)$ devido ao uso do Bubble Sort. O crescimento quadrático torna-se perceptível a partir de aproximadamente 2000 objetos, onde o tempo ultrapassa 500ms.

6.2 Experimento 2: Impacto de Movimentos e Objetos no Tempo de Ordenação

6.2.1 Objetivo

Analisar a relação entre o número de movimentos, número de objetos e o tempo acumulado de ordenação (sortTime), validando o compromisso da estratégia de limiar infinito adotada.

6.2.2 Configuração

Os testes foram realizados com 100 casos, de acordo com os seguintes indicadores:

- Objetos: 1 até 5000 (progressão linear)
- Cenas: 1 até 1000 (progressão linear)
- Movimentos: 10000 (fixo)
- Largura: 80-100 (fixo)
- Espaço: 100×100 (fixo)

Essa configuração permite observar o efeito combinado do aumento de objetos (custo por ordenação) e aumento de cenas (frequência de ordenações).

6.2.3 Resultados

O gráfico da Figura 2 apresenta o tempo de ordenação em função do número total de movimentos, com pontos coloridos pelo número de objetos (azul = poucos objetos, vermelho = muitos objetos). A linha de tendência representa o crescimento teórico esperado do Bubble Sort $O(n^2)$.

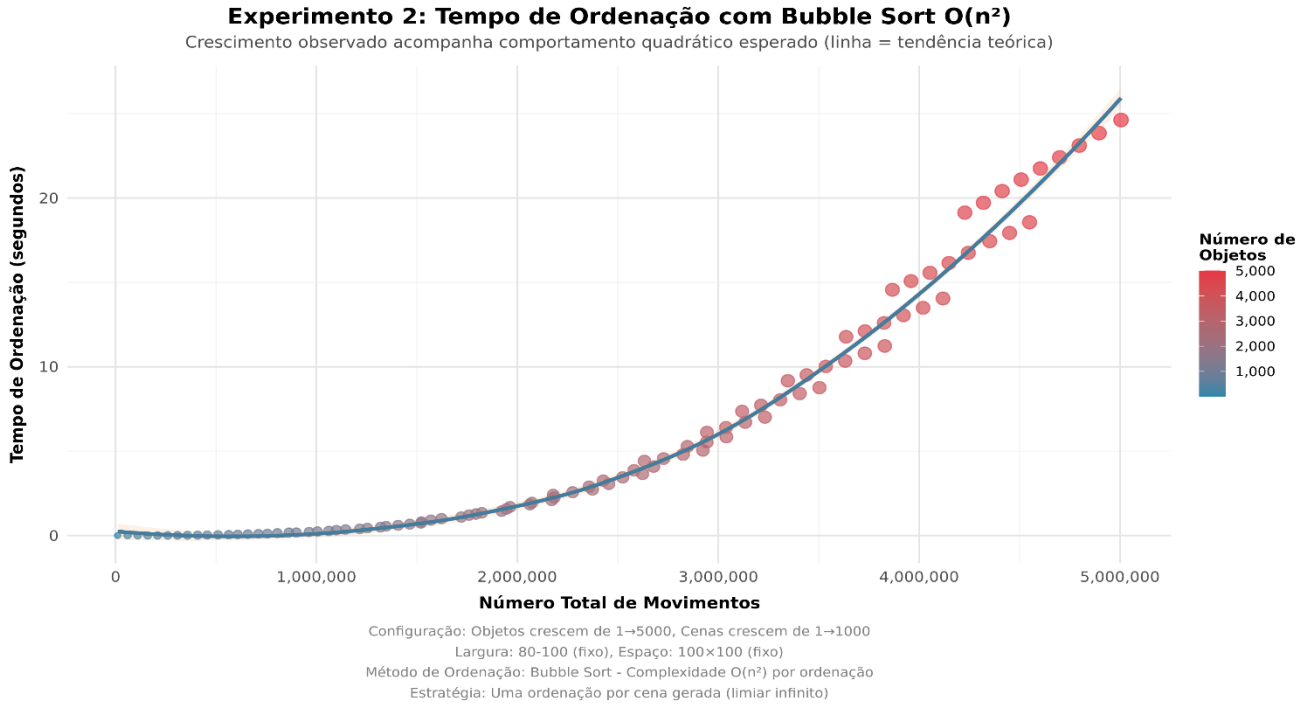

Figura 2: Tempo de ordenação com Bubble Sort $O(n^2)$ - crescimento experimental acompanha comportamento quadrático esperado.

Métricas observadas:

- Total de desorganizações: superior a 500 milhões ao longo de todos os testes
- Total de ordenações: 59.500 (uma por cena gerada)
- Correlação Movimentos × SortTime: muito forte (≥ 0.9)
- Correlação Objetos × SortTime: muito forte (≥ 0.9)
- Correlação Cenas × SortTime: muito forte (≥ 0.9)

Tratando-se da análise de compromisso, a estratégia de limiar infinito adotada resultou em aproximadamente 500 milhões de desorganizações, acompanhadas de apenas 59.500 ordenações. Isso representa uma razão de 8.400 desorganizações por ordenação, em média, validando a eficiência da abordagem: tolerar desorganizações temporárias reduz drasticamente o número de operações custosas ($O(n^2)$) realizadas.

6.2.4 Conclusão

Os resultados demonstram que a estratégia de limiar infinito é altamente eficaz: embora o vetor fique desorganizado frequentemente, o custo total é controlado ao concentrar as ordenações apenas nos momentos necessários (geração de cenas). As três correlações muito fortes (> 0.9) confirmam que o tempo de ordenação é previsível e segue o modelo teórico sortTime $\propto n^2 \times$ cenas. O efeito combinado de objetos e cenas domina o tempo total de execução.

6.3 Experimento 3: Crescimento Dramático da Complexidade das Cenas

6.3.1 Objetivo

Avaliar como a variação simultânea de todos os parâmetros (objetos, largura, densidade espacial) afeta a complexidade das cenas, medida através da métrica avgSegmentsPerScene.

6.3.2 Configuração

Os testes foram realizados com 100 casos, variando, desta vez, todos os parâmetros simultaneamente:

- Objetos: 1 até 5000 (crescente)
- Movimentos: 1 até 10000 (crescente)
- Cenas: 1 até 1000 (crescente)
- Largura: 800-1000 até 0.8-1.0 (decrescente, ordem de grandeza 1000→1)
- Espaço: 100×100 até 10000×10000 (crescente)

Essa configuração foi projetada para maximizar o efeito observável: testes iniciais têm objetos muito grandes em espaços pequenos (alta oclusão), enquanto testes finais têm objetos minúsculos em espaços gigantescos (baixa oclusão).

6.3.3 Resultados

O gráfico da Figura 3 apresenta o crescimento dramático de avgSegmentsPerScene ao longo da progressão dos testes. A curva demonstra aumento exponencial na complexidade das cenas conforme objetos diminuem e espaço aumenta.

- Teste 1: (largura ~1000, espaço 100×100): ~1-5 segmentos por cena (objetos enormes causam oclusão massiva)
- Teste 50: (largura ~500, espaço ~5050×5050): ~100-500 segmentos por cena
- Teste 100: (largura ~1, espaço 10000×10000): ~4860 segmentos por cena (objetos minúsculos raramente se ocluem)

Em se tratando da análise do crescimento, o esse fator foi superior a 1000×, demonstrando que a densidade espacial (objetos por unidade de área) é o fator dominante na complexidade das cenas. A área do espaço cresce quadraticamente (10.000 → 100.000.000), enquanto a largura dos objetos diminui linearmente (1000 → 1), resultando em densidade que cai dramaticamente de 5.0 para 0.00002 objetos/unidade².

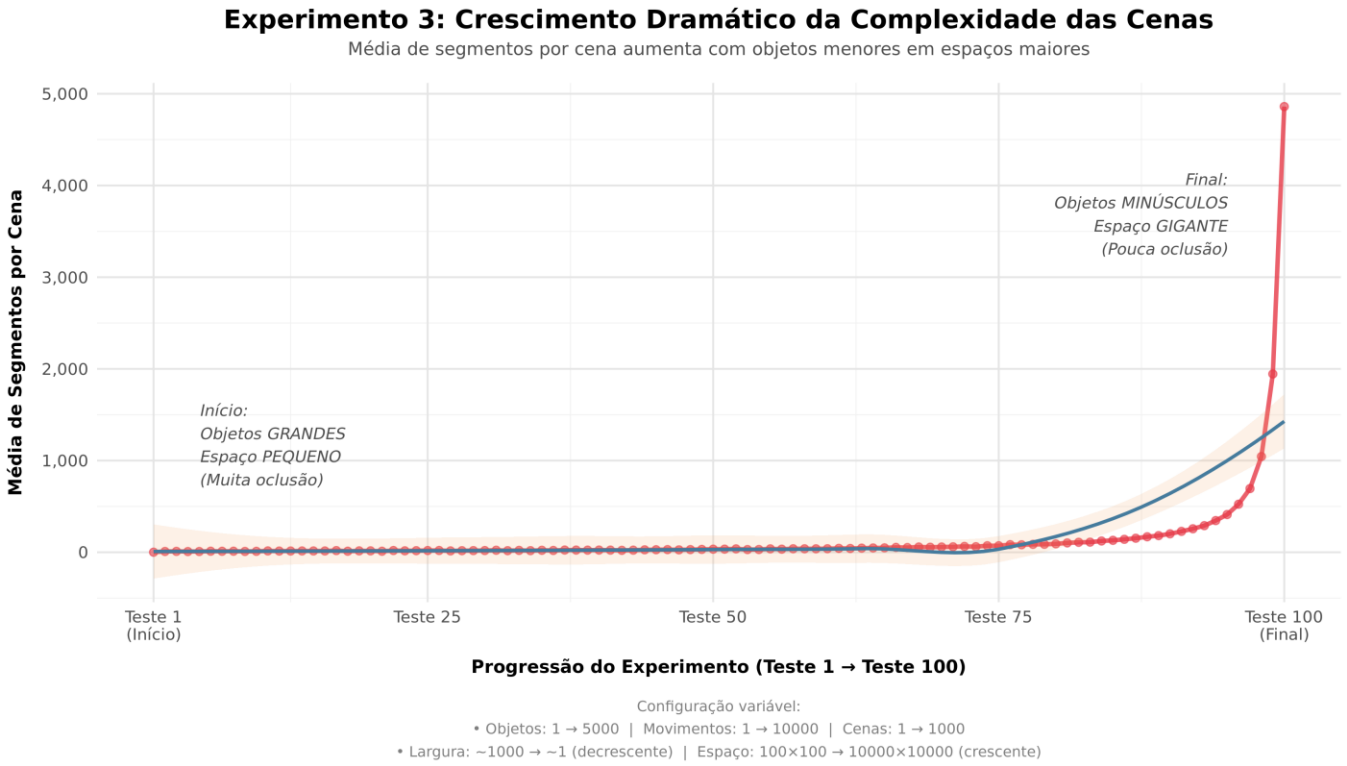

Figura 3: Crescimento dramático da complexidade das cenas - média de segmentos por cena aumenta com objetos menores em espaços maiores.

6.3.4 Conclusão

Os resultados confirmam a hipótese do enunciado: quanto maior a largura dos objetos e maior a densidade espacial, menos complexa é a cena devido ao aumento de oclusões. Inversamente, objetos pequenos em espaços grandes resultam em cenas extremamente complexas com milhares de segmentos visíveis. A correlação forte (> 0.8) entre a progressão dos testes e a métrica “avgSegmentsPerScene” calculada nos testes valida o modelo experimental. Este experimento demonstra que o algoritmo de cálculo de intervalos visíveis escala adequadamente mesmo para cenários de alta complexidade.

7 Conclusões

O trabalho realizado implementou um sistema de geração de cenas com detecção de oclusões para objetos unidimensionais, permitindo avaliar o compromisso entre frequência de ordenação e custo computacional através da estratégia de limiar infinito. A solução utiliza alocação estática de memória e dois tipos abstratos de dados criados para encapsular objetos e cenas, gerando apenas os segmentos visíveis após processamento de oclusões.

Os principais aprendizados incluem a aplicação prática de análise de complexidade algorítmica, compreensão do impacto de decisões de projeto, como a escolha do momento de ordenação no desempenho total do sistema, e a importância da análise experimental para validar hipóteses teóricas. A implementação evidenciou que tolerar desorganização temporária do vetor (limiar infinito) reduz drasticamente o número de ordenações necessárias sem comprometer a corretude, resultando em ganho de eficiência comparado a abordagens mais conservadoras. Além disso, os experimentos demonstraram que parâmetros como densidade espacial e largura dos objetos impactam exponencialmente a complexidade das cenas geradas, validando as relações teóricas entre oclusão e visibilidade de objetos.

8 Bibliografia

[1] ZIVIANI, N. Projeto de Algoritmos com Implementações em Java e C++. São Paulo: Cengage Learning, 2010. ISBN 9788522108213.

[2] CALAIS, P. Combinando Python e R. Apresentação no evento GDG DevFest Nova Lima 2024.

[3] LACERDA, A.; SANTOS, M.; MEIRA JR., W.; CUNHA, W. Estruturas de Dados. Notas de aula da disciplina DCC205 - Estruturas de Dados, Departamento de Ciência da Computação, ICEx/UFMG.