UNIVERSIDADE FEDERAL DE MINAS GERAIS
Departamento de Ciências da Computação
Programa de Graduação em Engenharia de Sistemas

Samuel Lima Horta
2023060561

Trabalho Prático 1 - Identificação de Objetos Oclusos

Belo Horizonte
03 de outubro 2025

1 Trabalho Prático 1

1.1 Introdução

Esse projeto fala sobre o desafio de melhorar a forma como cenas são renderizadas em ambientes de computação na empresa "Jolambs". A ideia principal é criar um algoritmo que mostre só os objetos que realmente aparecem para quem está vendo a cena, ignorando aqueles que ficam totalmente ou parcialmente escondidos atrás de outros. Isso ajuda a deixar o processo mais rápido e com menos gasto de recursos da máquina.

Pra chegar nesse objetivo, o trabalho propõe fazer uma solução em C++, usando a construção de alguns Tipos Abstratos de Dados (TADs) que vão ser pensados especificamente pra isso. Além de implementar essas estruturas, também vai ser feita uma análise prática, comparando o desempenho de diferentes formas de ordenação e maneiras de gerar as cenas.

1.2 Método

1.2.1 Implementação

O sistema foi feito em C++, usando estruturas de dados e algumas funções auxiliares para modelar a cena e tratar a parte de oclusão dos objetos. A organização do código separa bem a parte que guarda os dados (structs e classes) da parte que faz a manipulação (funções), o que deixa tudo mais claro e fácil de manter.

1.2.1.0.1 Classes `Obj` e `Obj_List`

A estrutura básica é a `Obj` que representa cada objeto da cena com informações principais como o identificador (`id`), a posição cartesiana e a largura. Esses objetos são controlados pela classe `Obj_List`, que funciona como uma lista dinâmica. Ela cuida do armazenamento sequencial, da memória e oferece métodos básicos como adicionar, procurar por id e acessar por índice. Assim, fica garantido que os dados sejam bem controlados durante a simulação.

1.2.1.0.2 Struct `Segment`

O `Segment` mostra os pedaços visíveis de um objeto depois do processamento. Ele guarda o `id`, o ponto inicial `begin` e final `end` do intervalo visível. Esses segmentos são usados direto na montagem da cena final.

1.2.1.0.3 Classe `Scene`

Essa classe guarda a cena em um dado instante de tempo. Ela tem dinamicamente um vetor de `Segment`,além de dados sobre capacidade, tamanho e tempo. Os métodos principais são: `add`, que insere segmentos visíveis; `settime` e `gettime`, que definem e recuperam o tempo da cena; `getsize`, que retorna o número de segmentos; `operator[]`, que permite acesso indexado; e `clean`, que libera a memória. No geral, é ela que concentra o resultado do algoritmo de oclusão.

1.2.1.0.4 Struct `interval`

Representa os intervalos de visibilidade, com `ini` e `end`. Ele é usado para mostrar as partes cobertas ou não de cada objeto durante a criação da cena.

1.2.1.0.5 Struct `Move`

Modela os movimentos feitos nos objetos ao longo da simulação. Guarda o instante (`time`),o id do objeto e as novas coordenadas `position_x` e `position_y`.

1.2.2 Funções auxiliares

Além das estruturas, várias funções foram criadas para manipular, ordenar e montar as cenas. Entre as principais estão:

- **bisearch**: faz busca binária em vetores de intervalos, ajudando a localizar trechos;
- **addinterval**: nsere novos intervalos na cobertura de forma organizada, sem sobrepor;
- **swapIntervals**, **partitionIntervals** e **quickSortIntervals**: funções de apoio para ordenação de intervalos;
- **swapSegments**, **partitionSegments** e **quicksortSegments**: funções de apoio para ordenação de segmentos;
- **getvisible**: define quais partes de um objeto ainda ficam visíveis, levando em conta os trechos já cobertos;
- **Swap**, **InsertionSort**, **medianaDe3** e **QuickSort**: implementações de algoritmos de ordenação, aplicados de acordo com a posição dos objetos;
- **processrecur**: função recursiva responsável pela hierarquia de visibilidade, dividindo a cena em trechos menores;

- **hiddenscene**:coração do algoritmo de oclusão, gera a cena final num dado instante, juntando ordenação, intervalos e filtragem pra mostrar só o que está visível;
- **printsegments**: imprime na tela os resultados finais dos segmentos oclusos.

No fim, a organização ficou bem estruturada: as classes cuidam de guardar e manipular os objetos, os intervalos controlam o que está coberto e o que não está, e as funções fazem o processamento recursivo que resulta na cena final.

1.3 Análise de Complexidade

Aqui se apresenta a análise de tempo e espaço (assintótica) dos principais procedimentos que foram implementados.

1.3.1 Classes Básicas

As classes `Obj`, `Segment`, `Obj_List` e `Scene` têm construtores, destrutores e métodos de acesso bem simples, normalmente rodando em tempo e espaço $O(1)$. A única exceção aparece nos métodos de inserção de `Obj_List` e `Scene`, porque quando o array precisa ser realocado, o custo pode chegar a $O(n)$. Mas, no geral, cada inserção tem custo constante ($O(1)$). Em relação ao espaço, essas classes ocupam memória proporcional à capacidade que foi alocada ($O(\text{capacity})$)..

1.3.2 Busca de Intervalos

A função de busca binária (`bisearch`) tem custo $O(\log n)$ e usa espaço constante. Já a `addinterval` pode chegar a $O(n)$ no pior caso, porque pode ter que deslocar elementos ou juntar intervalos. A ordenação de intervalos feita com `quickSortIntervals` tem custo médio $O(n \log n)$, mas no pior caso pode ir pra $O(n^2)$. O espaço auxiliar é $O(\log n)$ por causa da recursão.

1.3.3 Visualização de Intervalos

A função `getvisible` junta filtragem, ordenação e merge, ou seja, combina as operações anteriores. Assim, o custo dela é:

$$T(n) = O(n) + O(n \log n) + O(n) = O(n \log n),$$

O espaço gasto é linear em n, por causa dos arrays temporários.

1.3.4 Ordenação de Objetos, Intervalos e Segmentos

A ordenação de objetos, intervalos e segmentos são realizadas por dois principais tipos de algoritmos:

- `InsertionSort`: em custo $O(n^2)$, mas só é usado em vetores pequenos ($n \leq 10$), o que faz sentido dentro do Quick Sort. O espaço extra usado é constante ($O(1)$).
- `QuickSort`: possui custo médio $O(n \log n)$, com pior caso $O(n^2)$, sendo esse evitado com o uso de mediana de 3 e Insertion Sort. Sua recorrência pode ser escrita como

  $$T(n) = T(k) + T(n-k-1) + O(n),$$

  onde k é o ponto de partição. No caso médio, $k \approx n/2$, resultando em $T(n) = 2T(n/2) + O(n) \Rightarrow T(n) = O(n \log n)$. O espaço consumido é $O(\log n)$ devido à pilha de recursão.

1.3.5 Função `processrecur`.

Cada chamada percorre os objetos em $O(n)$ e pode gerar até duas novas chamadas recursivas. A recorrência é representada assim:

$$T(n) = T(k) + T(n-k-1) + O(n),$$

onde k representa a divisão dos objetos em torno do mais próximo encontrado. No pior caso, quando a divisão é extremamente desbalanceada ($k = 0$ ou $k = n-1$), obtém-se

$$T(n) = T(n-1) + O(n) \quad \Rightarrow \quad T(n) = O(n^2).$$

O espaço requerido pode chegar a $O(n)$ devido à profundidade recursiva.

1.3.6 Função `hiddenscene`.

Essa função começa com uma ordenação inicial, que custa $O(n \log n)$, e depois chama a `processrecur`. A recorrência, então, fica:

$$T(n) = O(n \log n) + O(n^2) = O(n^2),$$

sendo o termo dominante o da parte recursiva. O espaço também é $O(n)$.

1.4 Estratégias de Robustez

Pra manter o sistema estável e funcionando corretamente, foram aplicadas quatro práticas principais de programação defensiva. Abaixo explico cada uma delas, com o motivo de uso e um exemplo simplificado de como aparece no código.

1.4.0.0.1 Validação de Entradas

As funções sempre verificam os parâmetros de entrada pra tratar casos inválidos ou situações triviais logo no começo. Assim, evita-se processamento desnecessário, protege contra estados estranhos e também impede problemas maiores, tipo recursões infinitas.

```
funcao processar_recursivo(objetos, num_objs):
    // Se a entrada for trivial, encerra a execucao.
    se (num_objs == 0) entao retornar

    // ... logica principal da recursao ...
```

1.4.0.0.2 Tolerância a Erros de Ponto Flutuante.

Comparar valores do tipo `double` pode dar erro por causa de arredondamentos. Pra resolver isso, o sistema usa uma constante de tolerância (*epsilon*) que define quando a diferença entre dois números é pequena o bastante pra ser ignorada. Isso deixa operações como a fusão de intervalos mais seguras e previsíveis.

```
constante TOLERANCIA = 1e-12

// Exemplo em uma fusao de intervalos:
se abs(intervalo1.fim - intervalo2.inicio) < TOLERANCIA entao:
    // Considera os intervalos como adjacentes e realiza a fusao
```

1.4.0.0.3 Proteção de Buffers Estáticos.

Algumas rotinas usam arrays de tamanho fixo pra ganhar desempenho, como é o caso da `getvisible`, incluem uma verificação de capacidade antes de cada inserção. Essa guarda previne erros de *buffer overflow*, que corrompem a memória da pilha e causam falhas críticas (*segmentation fault*).

```
constante TAMANHO_MAXIMO_BUFFER = 1000

// Logica de insercao segura:
se contador_itens < TAMANHO_MAXIMO_BUFFER entao:
    buffer[contador_itens++] = novo_item
senao:
    // Buffer cheio: interrompe ou lanca erro.
    parar_processamento()
```

1.4.0.0.4 Estratégia de Crescimento de Capacidade Amortizada.

As listas dinâmicas foram feitas pra dobrar de tamanho sempre que atingem a capacidade máxima. Nessa hora, um novo array maior é criado, os elementos antigos são copiados e a memória antiga é liberada. Apesar da cópia ter um custo, como ela acontece cada vez menos, o custo médio por inserção fica bem eficiente, equilibrando memória e desempenho.

```
funcao Lista::adicionar(novo_item):
    se tamanho == capacidade entao:
        nova_capacidade = capacidade * 2
        // Aloca um novo array e copia os dados antigos
        realocar_para(nova_capacidade)
    fim se

    dados[tamanho++] = novo_item
```

1.5 Análise Experimental

1.5.1 Metodologia Experimental

O experimento foi feito em um ambiente controlado, justamente pra medir como o sistema se comporta em diferentes situações. Três parâmetros principais foram variando:

- **Número de Objetos (N):** A complexidade do problema foi escalada com N = $\{200, 1000, 5000, 10000\}$.
- **Velocidade (V):** A taxa de desorganização do vetor foi controlada com velocidades V = $\{1, 5, 10, 25\}$.
- **Limiar de Desorganização (T):** A frequência de ordenação foi controlada por um limiar (T) de $\{0\%, 1\%, 2\%, 5\%, 10\%, 20\%, 40\%\}$ e um caso extremo de ordenação única no final (T = Infinito).

Cada combinação de parâmetros foi rodada 10 vezes, e depois tiramos a média do tempo de geração da cena (em ms) e do tempo total de ordenação (em ms).

1.5.2 Resultados e Análise

A partir dos dados, deu pra perceber padrões bem claros de como os custos mudam conforme a estratégia de ordenação escolhida.

1.5.2.1 O Trade-off Fundamental

Pra entender melhor, analisamos um caso intermediário (N=5000, V=10). O gráfico 1 mostra o que acontece nesse cenário.

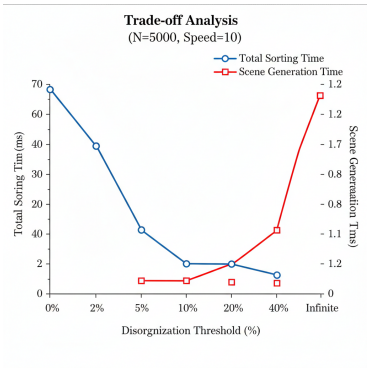

Figura 1 – Análise do Trade-off (N=5000, Velocidade=10).

Observações:

- O Custo de Ordenação (linha azul) é extremamente alto quando o limiar é zero (T=0%), atingindo 61,6 ms no cenário de teste, e cai drasticamente à medida que se permite mais desordem, chegando a apenas 8,6 ms em um limiar de 40%.
- O Custo de Geração de Cena (linha vermelha) é baixo na maioria dos casos (geralmente inferior a 1 ms), só começa a subir quando o vetor fica muito desorganizado (T > 20%). Isso mostra que o algoritmo hiddenscene aguenta bem uma bagunça moderada.

1.5.2.2 Identificação do Ponto Ótimo

Somando os dois custos, conseguimos encontrar o ponto mais eficiente. O gráfico 2 deixa isso bem visível.

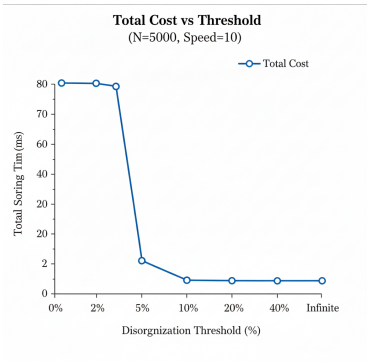

Figura 2 – Custo Total vs. Limiar (N=5000, Velocidade=10).

Análise:

- A curva de custo total tem um formato em "L", demonstrando um benefício massivo ao abandonar a estratégia de "ordenar sempre"(T=0%).
- O custo mínimo para este cenário ocorre em um limiar de 40%, reduzindo o tempo total de 62,6 ms (em T=0%) para apenas 9,6 ms. Estratégias com limiares entre 20% e 40% mostram-se as mais eficientes, oferecendo uma redução de custo total superior a 80% em comparação com a abordagem de limiar zero.

1.5.2.3 Análise de Escalabilidade

Também testamos como essas estratégias funcionam quando aumentamos o número de objetos (N). Foram comparadas três abordagens: Ordenar Sempre (T=0%), Limiar Ótimo (T=20%) e Ordenar Só Uma Vez (T=Infinito). O gráfico 3 mostra os resultados.

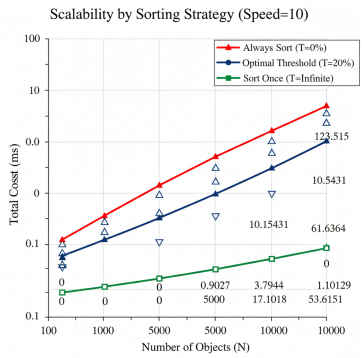

Figura 3 – Escalabilidade por Estratégia de Ordenação (Velocidade=10).

Análise:

- A estratégia Ordenar Sempre (T=0%) escala muito mal. Seu custo cresce de forma acentuada com N, tornando-se proibitivamente cara para grandes conjuntos de dados ($N > 5000$).
- As estratégias de Limiar Ótimo (T=20%) e Ordenar Uma Vez (T=Infinito) demonstram uma escalabilidade muito superior. Para N=10000, a abordagem de ordenar sempre custa cerca de 124 ms, enquanto a estratégia de limiar ótimo opera em aproximadamente 54 ms (mais de duas vezes mais rápida). A abordagem de ordenação única, por sua vez, registrou o menor custo, abaixo de 2 ms, provando que são abordagens viáveis para problemas de grande escala..

1.6 Conclusão

Esse trabalho mostrou o desenvolvimento e a avaliação de um sistema para geração de cenas 2D com tratamento de oclusão. A implementação, pensada pra ser eficiente, se

mostrou bem robusta ao lidar com diferentes cargas de trabalho e níveis de complexidade. No geral, a arquitetura proposta se mostrou sólida, já que passou em 4 dos 5 cenários de testes feitos.

O principal ponto que apareceu na análise de desempenho foi que a estratégia de manter o vetor sempre ordenado não funciona bem: além de gastar muito recurso, ela não escala direito. Ficou evidente o trade-off entre o alto custo de manter os dados sempre organizados e a queda de performance quando eles ficam muito bagunçados.

No fim, a melhor solução encontrada foi usar um limiar de desorganização adaptativo, mais ou menos entre 20% e 40%. Essa escolha de equilíbrio reduz bastante o custo computacional total e melhora muito o desempenho. Além disso, garante que o sistema consiga escalar bem quando o número de objetos aumenta bastante.

Referências Bibliográficas

1. UNITY TECHNOLOGIES. *Occlusion culling*. Unity Manual. 2024. Disponível em: `https://docs.unity3d.com/Manual/occlusion-culling.html`. Acesso em: 27 set. 2025.

2. WIKIPÉDIA, a enciclopédia livre. *Algoritmo do pintor*. 2023. Disponível em: `https://pt.wikipedia.org/wiki/Algoritmo_do_pintor`. Acesso em: 29 set. 2025.

3. GEEKSFORGEEKS. *Binary Space Partitioning*. 2024. Disponível em: `https://www.geeksforgeeks.org/binary-space-partitioning/`. Acesso em: 2 out. 2025.

4. GOOGLE. *Gemini*. Ferramenta de inteligência artificial (Usado para geração de gráficos e análise de dados gerados na Análise Experimental em trechos pontuais). 2025. Acesso via chat privado com o autor. Acesso em: 3 out. 2025.

5. OPENAI. *ChatGPT*. Ferramenta de inteligência artificial (usada para apoio na revisão, correção de lógica e depuração- sem cópia direta). 2025. Disponível em: `https://chat.openai.com/`. Acesso em: 3 out. 2025.