Relatório Trabalho 3 Geometria Computacional

Matheus T. Batista[1], **Davi G. Lazzarin**[1]

[1]Universidade Federal do Paraná,

{mtb21, dgl20}@inf.ufpr.br

1. Introdução

O trabalho consiste em construir uma BSP no $\mathbb{R}^3$ para ordenar triângulos e calcular interseção de semi-retas com os triângulos.

1.1. Entrada

A entrada consiste em três partes:

1. Uma lista de n pontos, cada um descrito por suas coordenadas (x, y, z). Esses pontos são lidos em ordem e recebem índices de 1 a n conforme a ordem de leitura.
2. Uma lista de T triângulos. Cada triângulo é descrito por três índices i_1, i_2, i_3, que referenciam os pontos lidos anteriormente.
3. Uma lista de L segmentos de reta. Cada segmento é descrito por seis números inteiros

   $$(p_{ix},\ p_{iy},\ p_{iz},\ p_{fx},\ p_{fy},\ p_{fz}),$$

   onde (p_{ix}, p_{iy}, p_{iz}) são as coordenadas do ponto inicial e (p_{fx}, p_{fy}, p_{fz}) as coordenadas do ponto final.

1.2. Saída

A saída é uma lista de L linhas onde cada linha contém um valor k sendo $0 \leq k \leq n$ que representa a quantidade de triângulos intersectados pelo segmento de reta, seguido por $k$valores de 1 a n em ordem crescente que representa o índice do triângulo que aquele segmento intersecta.

2. Desenvolvimento

2.1. Visão geral

2.1.1. Estrutura do código

O programa foi escrito em C++ e foram utilizados 3 arquivos de código-fonte, sendo eles:

- **`main.cpp`** - contém a lógica principal do programa, leitura dos dados, processamento e escrita da saída.
- **`poligono.hpp`** - contém a declaração de algumas estruturas para representação de entidades geométricas, como pontos e segmentos de reta, além de algumas funções para manipulação dessas.
- **`bsp.hpp`** - contém estruturas e funções para criação e manipulação da bsp no $\mathbb{R}^3$.

2.1.2. Lógica do programa

Após a leitura dos dados da entrada, os valores são armazenados em vetores de ”ponto”, ”triângulo”e ”segmento”, sendo esses tipos de dados definidos em ”**poligono.hpp**”. Em seguida, a BSP recebe o vetor de triângulos e constrói a estrutura, retornando a raiz da BSP. Após isso, percorremos o vetor de segmentos e para cada segmento, fazemos uma busca na BSP e isso nos retorna quantos triângulos são intersectados pelo segmento. Antes de consultar pelo próximo segmento, a impressão da saída é realizada.

2.2. BSP

Para a construção da BSP, foram necessárias algumas estruturas auxiliares para representação dos hiperplanos dos triângulos.

Sobre as estruturas de dados:

Foi criada uma estrutura **hiperplano** que possui:

- **d**: os coeficientes produzidos pelo triângulo que o gera.
- **normal**: o vetor normal que indica o deslocamento do hiperplano, necessário para verificar em qual meio espaço um objeto está.

Para explicar a necessidade desses atributos, vamos começar com a definição do hiperplano em $\mathbb{R}^3$ que é:

$$a_1c_1 + a_2c_2 + \cdots + a_nc_n + c_{n+1} = 0$$

Temos que **normal** são os $c_1, c_2, \ldots, c_n$: o vetor normal do plano. E que **d** é o c_{n+1}: o termo independente do plano.

Funções/métodos da estrutura **hiperplano**:

- **meioEspaco(ponto p)**: dá o valor que indica de qual lado do hiperplano o ponto está.
- **ladoSegmento(segmento s)**: retorna o lado no qual o segmento se encontra.
- **ladoTriangulo(triangulo t)**: retorna o lado no qual o triângulo se encontra.

É criada uma `struct BSPNode`, que representa um nó da árvore, ele é definido do seguinte modo:

- Atributos:
  - **planoDivisor**: um hiperplano, que separa o conjunto de triângulos em dois grupos (esquerda e direita).
  - **triangulosSobreSegmento**: os triângulos que intersectam o plano.
  - **esquerda**: subárvore que fica atrás do plano.
  - **direita**: subárvore que fica à frente do plano.
- Construtores:
  - Há duas maneiras de se criar um **BSPNode**:
    1. Um nó interno, com hiperplano.
    2. Uma folha, sem hiperplano.

    Os dois não precisam de criações diretas para seus filhos (**esquerda** e **direita**), já que a estrutura é criada de baixo para cima (os filhos precisam estar processados para criar o nó pai).

- Funções/métodos:
  - **`ehFolha()`**: Uma função que verifica se é folha.

O algoritmo tem como base o apresentado em de Berg et al. (2010). No caso mostrado no livro, a função que gera a BSP assume que não há interseções entre os triângulos, enquanto o trabalho atual trata explicitamente o caso em que o plano de divisão corta um triângulo. Nessa situação, o triângulo é dividido em um triângulo e um quadrilátero, sendo que o quadrilátero é posteriormente decomposto em dois triângulos adicionais.

A função principal que engloba a criação da BSP é a função `criaBSP`, abaixo está a definição do mesmo. Seja:

Algorithm 1 criaBSP(S)

Require: Conjunto de triângulos S
if $|S| \leq 1$ **then**
  Cria e retorna uma folha contendo os triângulos de S
else
  Cria o hiperplano h a partir do triângulo t_1 de S
  Inicializa os conjuntos h^+, h^- e P como vazios
  for all triângulo t em $S - \{t_1\}$ **do**
    if t está à direita de h **then**
      Adiciona t a h^+
    else if t está à esquerda de h **then**
      Adiciona t a h^-
    else
      Adiciona t a $\mathcal{P}$
    end if
  end for
  $esquerda \leftarrow criaBSP(h^-)$
  $direita \leftarrow criaBSP(h^+)$
  Cria o nó p
  $p.planoDivisor \leftarrow h$
  $p.triangulosSobreSegmento \leftarrow P$
  $p.esquerda \leftarrow esquerda$
  $p.direita \leftarrow direita$
  return p
end if

2.3. Busca dos segmentos

A busca é implementada através de uma função recursiva que percorre a árvore, que pode ser definida conceitualmente como **`Busca(Nó, Segmento)`**. A função recebe o nó atual da árvore a ser visitado e o segmento de reta a ser testado.

O algoritmo em cada nó segue dois passos principais:

1. **Testar Interseções Locais:** Primeiramente, o algoritmo testa se o segmento de consulta s intersecta qualquer um dos triângulos armazenados na lista de polígonos coplanares do nó atual. Para cada triângulo nesta lista, um teste de

interseção geométrico direto é realizado. Se uma interseção é detectada, o ID do triângulo original é adicionado à lista de resultados.

2. **Classificar e Recorrer:** Em seguida, o algoritmo classifica a posição do segmento de reta s em relação ao plano divisor $\mathcal{P}$ do nó atual. O resultado desta classificação determina como a busca prosseguirá recursivamente.

O caso base da recursão ocorre quando o nó a ser visitado é nulo (`nullptr`), o que representa uma região vazia do espaço, e a busca naquela ramificação termina.

3. Análise dos Casos de Travessia

A classificação do segmento em relação ao plano divisor do nó define o caminho da busca. Existem quatro casos possíveis:

3.1. Caso 1: Segmento Contido em um dos Semi-espaços

Se o segmento de reta s está inteiramente contido no semi-espaço positivo ou no semi-espaço negativo do plano $\mathcal{P}$, ele não pode intersectar nenhum triângulo que esteja no semi-espaço oposto.

- **Ação:** A busca recursiva continua *apenas* na sub-árvore correspondente àquele semi-espaço (a sub-árvore direita para o lado positivo, e a esquerda para o lado negativo). A outra sub-árvore inteira é podada, resultando em um ganho significativo de eficiência.

3.2. Caso 2: Segmento Coplanar ao Plano Divisor

Se o segmento s está inteiramente contido no plano divisor $\mathcal{P}$, todas as suas possíveis interseções com os triângulos da árvore já ocorreram com os triângulos coplanares armazenados neste nó (testados no Passo 1).

- **Ação:** A busca nesta ramificação termina, pois não há necessidade de explorar as sub-árvores filhas.

3.3. Caso 3: Segmento Intersecta o Plano Divisor

Este é o caso em que o segmento de reta s cruza o plano $\mathcal{P}$, tendo uma parte em cada semi-espaço.

- **Ação:** Como o segmento existe em ambos os semi-espaços, ele tem o potencial de intersectar triângulos em ambas as sub-árvores. Portanto, a busca recursiva deve ser chamada para as *duas* sub-árvores filhas, a esquerda e a direita.

3.4. Custo da Consulta de Interseção

A grande vantagem da árvore BSP se manifesta na eficiência das consultas. Seja n o número de triângulos na árvore e k o número de triângulos intersectados reportados.

- **Caso Esperado:** Em uma árvore balanceada, uma consulta com um segmento de reta percorre um caminho da raiz até uma ou mais folhas. O custo esperado para encontrar as regiões do espaço que o segmento atravessa é proporcional à altura da árvore, ou seja, $\mathcal{O}(\log n)$. O custo total da consulta é então $\mathcal{O}(k \cdot \log n)$, onde k é o número de interseções reportadas.
- **Pior Caso:** No pior cenário, o segmento pode cruzar muitos hiperplanos, forçando a visita a uma grande porção dos nós da árvore. Nesse caso, o custo da consulta pode se aproximar de $\mathcal{O}(n)$, perdendo a vantagem logarítmica da estrutura.

4. Pós-processamento e Resultado Final

Durante a construção da árvore BSP, um triângulo original pode ser dividido em vários pedaços. Como resultado, o segmento de consulta pode intersectar múltiplos fragmentos que pertencem ao mesmo triângulo original, fazendo com que o ID daquele triângulo seja adicionado à lista de resultados várias vezes.

Para garantir que a saída final esteja de acordo com os requisitos do trabalho, um passo de pós-processamento é necessário após a conclusão da busca para cada segmento:

1. **Ordenação:** A lista de IDs de triângulos intersectados é ordenada numericamente.
2. **Remoção de Duplicatas:** Utiliza-se um algoritmo para remover os IDs duplicados da lista ordenada, garantindo que cada triângulo original seja reportado apenas uma vez.

O resultado final impresso para cada segmento é a contagem de triângulos únicos intersectados, seguida pela lista ordenada de seus IDs.

4.1. Bugs

Foi criado uma entrada no qual não foi possível obter todas as interseções entre os triângulos e os segmentos. Não foi identificado a causa da inconsistência. Segue abaixo a entrada, a saída e a sua representação gráfica pela figura 1.

4.1.1. Entrada

```
8 8 4
10 10 10
10 10 90
10 90 10
10 90 90
90 10 10
90 10 90
90 90 10
90 90 90
1 2 3
2 4 3
5 6 7
6 8 7
1 2 5
2 6 5
3 4 7
4 8 7
0 0 0 100 100 100
10 10 10 90 90 90
10 90 10 90 10 90
50 0 50 50 100 50
```

4.1.2. Saída

```
4 1 4 5 8
2 4 8
3 3 4 6
4 5 6 7 8
```

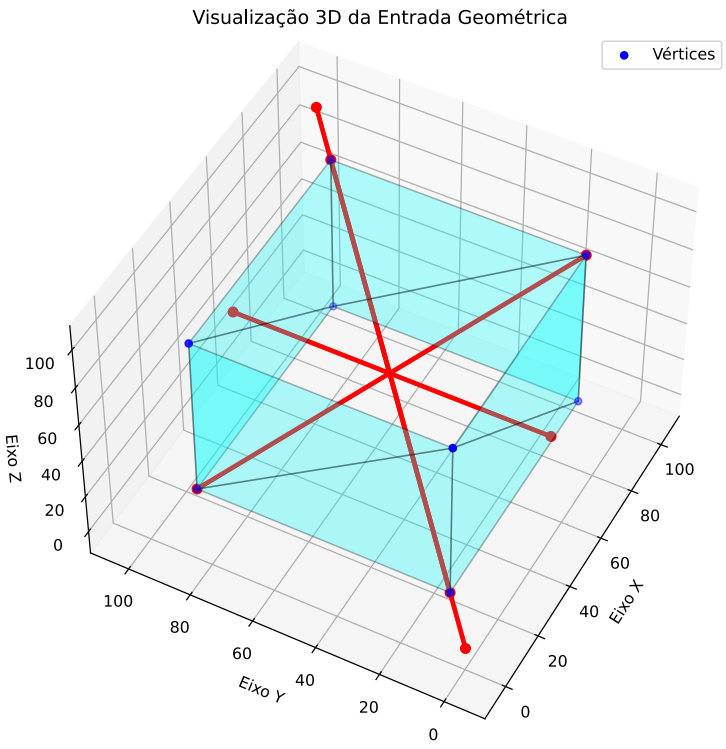

Figura 1. Representação gráfica da entrada.

Referências

M. de Berg, O. Cheong, M. van Kreveld, and M. Overmars. *Computational geometry*. Springer, Berlin, Germany, 3 edition, Oct. 2010.