Implementação de BSP Tree para Detecção de Interseções 3D

João Pedro Vicente Ramalho (GRR20224169)
Gabriela Fanaia De Almeida Dias Dorst (GRR20220070)
Universidade Federal do Paraná

21 de junho de 2025

1 Introdução

Este trabalho apresenta a implementação de uma Árvore BSP (Binary Space Partitioning) para detecção de interseções entre segmentos de reta e triângulos em ambientes 3D. A BSP Tree é uma estrutura hierárquica que particiona recursivamente o espaço usando planos arbitrários, organizando os objetos geométricos de forma a otimizar operações de consulta. O algoritmo desenvolvido é capaz de:

- Processar malhas triangulares 3D;
- Construir uma estrutura hierárquica de particionamento espacial;
- Responder consultas sobre interseções segmento-triângulo.

2 Fluxo Principal

O algoritmo segue a seguinte sequência de operações:

1. **Leitura de entrada**: Processa os dados de entrada contendo:
   - Número de vértices, triângulos e segmentos
   - Coordenadas dos vértices
   - Definição dos triângulos (índices de vértices)
   - Definição dos segmentos (coordenadas dos pontos extremos)
2. **Construção da BSP Tree**:
   - Particionamento recursivo do espaço usando planos definidos pelos triângulos
   - Classificação e divisão de triângulos em relação aos planos
   - Armazenamento hierárquico em nós da árvore
   - Complexidade: $O(n^2)$ no pior caso
3. **Consulta de interseções**:

- Para cada segmento, percorre a árvore hierarquicamente
- Coleta índices dos triângulos intersectados
- Complexidade: $O(k \log n)$ (melhor caso/médio) / $O(n^2)$ (pior caso)

3 Implementação Detalhada

3.1 Estruturas de Dados Principais

Algorithm 1 Estrutura do Nó da BSP Tree

```
1: classe Node:
2:     __slots__ = ('plane', 'triangles', 'positive_child', 'negative_child')
3:     def __init__(self):
4:         self.plane
5:         self.triangles
6:         self.positive_child
7:         self.negative_child
```

A classe `Node` representa um nó da árvore BSP (Binary Space Partitioning). A estrutura utiliza `__slots__` para definir um conjunto fixo de atributos, otimizando o uso de memória. O atributo `plane` armazena os coeficientes do plano (a, b, c, d) utilizado para a divisão do espaço. O atributo `triangles` contém os triângulos que estão coplanares ao plano do nó. Já `positive_child` e `negative_child` são referências às subárvores do lado positivo e negativo do plano, respectivamente.

3.2 Operações Geométricas

Criação de Planos

A definição de planos é realizada pela função `make_plane`, que recebe três pontos não colineares (p_0, p_1, p_2) no espaço 3D. Primeiramente, calculam-se dois vetores contidos no plano: $\overrightarrow{v_1} = p_1 - p_0$ e $\overrightarrow{v_2} = p_2 - p_0$. O vetor normal $\vec{n}$ é obtido através do produto vetorial $\vec{n} = \overrightarrow{v_1} \times \overrightarrow{v_2}$, cujos componentes (a, b, c) definem a inclinação do plano. A constante d da equação geral $ax + by + cz + d = 0$ é derivada de $d = -\vec{n} \cdot p_0$. Caso os pontos sejam colineares (detectado quando $\|\vec{n}\| < 10^{-10}$), a função retorna um valor inválido, pois não definem um plano único.

Classificação de Pontos

A função `classify_point` determina a posição relativa de um ponto p em relação a um plano. Através da avaliação da expressão $f(p) = a \cdot p_x + b \cdot p_y + c \cdot p_z + d$, classifica o ponto em três categorias:

- **Coplanares**: Quando $|f(p)| < \epsilon$, indicando que o ponto pertence ao plano dentro de uma tolerância numérica ($\epsilon = 10^{-10}$).
- **Lado Positivo**: Se $f(p) > \epsilon$, situando-se no semi-espaço para o qual o vetor normal $\vec{n}$ aponta.

- **Lado Negativo**: Se $f(p) < -\epsilon$, posicionando-se no semi-espaço oposto ao vetor normal.

Esta classificação é importante para o particionamento hierárquico do espaço durante a construção da árvore e para orientar a travessia durante consultas de interseção.

3.3 Divisão de Triângulos

Durante a construção da BSP Tree, é comum encontrar triângulos que atravessam o plano de partição. Nestes casos, o triângulo precisa ser dividido em sub-triângulos que se ajustem aos semi-espaços definidos pelo plano. O algoritmo `split_triangle` realiza essa operação.

Algorithm 2 Divisão de Triângulo

```
1: function SPLIT_TRIANGLE(triângulo, plano)
2:     vértices ← lista de pontos de triângulo
3:     positivos ← lista vazia
4:     negativos ← lista vazia
5:     coplanares ← lista vazia
6:     for cada ponto em vértices do
7:         lado ← classify_point(plano, ponto)
8:         if lado = POSITIVE then
9:             positivos.adicionar(ponto)
10:        else if lado = NEGATIVE then
11:            negativos.adicionar(ponto)
12:        else
13:            coplanares.adicionar(ponto)
14:    if |positivos|=1 e |negativos|=2 then
15:        P ← o único elemento de positivos
16:        N1, N2 ← elementos de negativos
17:        I1 ← intersect_edge_plane(P, N1, plano)
18:        I2 ← intersect_edge_plane(P, N2, plano)
19:        return [P, I1, I2], [N1, I1, I2], [N1, I2, N2]
20:    else if |positivos|=2 e |negativos|=1 then
21:        N ← o único elemento de negativos
22:        P1, P2 ← elementos de positivos
23:        I1 ← intersect_edge_plane(N, P1, plano)
24:        I2 ← intersect_edge_plane(N, P2, plano)
25:        return [N, I1, I2], [I1, P1, I2], [I1, I2, P2]
26:    else if |positivos|=1 e |negativos|=1 e |coplanares|=1 then
27:        P ← o único elemento de positivos
28:        N ← o único elemento de negativos
29:        C ← o único elemento de coplanares
30:        I ← intersect_edge_plane(P, N, plano)
31:        return [P, C, I], [N, C, I]
32:    else
33:        return lista contendo o triângulo original
```

O método split_triangle trata quatro casos principais:

1. **Um vértice positivo e dois negativos**: Primeiro calculamos os pontos de interseção I_1 e I_2 entre o vértice positivo e cada um dos vértices negativos, usando `intersect_edge_plane`. Em seguida, retornamos três sub-triângulos que cobrem todo o triângulo original sem sobreposição:

$$[P,\, I_1,\, I_2], \quad [N_1,\, I_1,\, I_2], \quad [N_1,\, I_2,\, N_2].$$

2. **Dois vértices positivos e um negativo**: O processo é similar ao caso anterior, mas invertido. Calculamos as interseções entre o único vértice negativo e cada um dos dois positivos, gerando dois pontos I_1 e I_2, e então formamos três sub-triângulos:

$$[N,\, I_1,\, I_2], \quad [I_1,\, P_1,\, I_2], \quad [I_1,\, I_2,\, P_2].$$

3. **Um vértice positivo, um negativo e um coplanar**: Calcula-se apenas uma interseção I entre o vértice positivo e o negativo. O vértice coplanar permanece intacto em ambos os sub-triângulos. Assim, retornamos dois sub-triângulos:

$$[P,\, C,\, I], \quad [N,\, C,\, I].$$

4. **Nenhum cruzamento ou todos coplanares**: Se o triângulo não cruza o plano (todos os vértices ficam de um mesmo lado) ou todos são coplanares, não há divisão e simplesmente retornamos o triângulo original.

Dessa forma, `split_triangle` garante que cada parte resultante esteja totalmente contida em um dos semi-espaços determinados pelo plano de corte, mantendo a consistência geométrica necessária para a construção da BSP Tree.

3.4 Interseção Segmento-Triângulo

A função `intersect_segment_triangle` é usada para as consultas na BSP Tree, ela é responsável por determinar se um segmento de reta 3D intersecta um triângulo específico. Em casos especiais, triângulos degenerados (colineares) são imediatamente detectados quando `make_plane` retorna um plano inválido, resultando em False pois não definem uma superfície intersectável.

Para segmentos paralelos ao plano do triângulo, a função adota outra estratégia: primeiro verifica se os extremos do segmento estão dentro do triângulo (via `point_in_triangle`). Em seguida, testa se algum vértice está sobre o segmento (via `point_on_segment`). Se ambas as verificações falharem, recorre a uma projeção 2D: o triângulo e o segmento são projetados no plano cartesiano que ignora a coordenada dominante da normal (ex: se a normal tem maior componente em z, projeta no plano xy), onde interseções com arestas são testadas em 2D.

No caso geral (não paralelo), calcula-se o ponto exato de interseção entre o segmento e o plano do triângulo. Se este ponto estiver dentro dos limites do segmento, aplica-se um teste de inclusão no triângulo via produtos vetoriais.

3.5 Construção da BSP Tree

A construção da *binary space partitioning tree* (BSP Tree) é um processo recursivo que organiza hierarquicamente os triângulos em um espaço 3D, facilitando operações de interseção. Este algoritmo segue uma estratégia de *divisão e conquista*, particionando o espaço geometricamente.

Algorithm 3 Construção da BSP Tree

```
1: function BUILD_BSP(triângulos)
2:     if triângulos está vazio then
3:         return None
4:     node ← novo Node()
5:     primeiro ← triângulos[0]
6:     node.plane ← make_plane(primeiro.v0, primeiro.v1, primeiro.v2)
7:     node.triangles ← [primeiro]
8:     pos_tris ← lista vazia
9:     neg_tris ← lista vazia
10:    for cada triângulo em triângulos[1:] do
11:        classif ← classify_triangle(node.plane, triângulo)
12:        if classif = COPLANAR then
13:            node.triangles.adicionar(triângulo)
14:        else if classif = POSITIVE then
15:            pos_tris.adicionar(triângulo)
16:        else if classif = NEGATIVE then
17:            neg_tris.adicionar(triângulo)
18:        else
19:            parts ← split_triangle(triângulo, node.plane)
20:            for cada parte em parts do
21:                adiciona parte em node.triangles, pos_tris ou neg_tris conforme classificação
22:    node.positive_child ← BUILD_BSP(pos_tris)
23:    node.negative_child ← BUILD_BSP(neg_tris)
24:    return node
```

A construção da BSP Tree baseia-se em três operações:

1. **Seleção do Plano de Partição**: cada nó da árvore é associado a um plano de corte, tipicamente definido por um dos triângulos da cena.
2. **Classificação Geométrica**: todos os triângulos são classificados em relação ao plano atual.
3. **Divisão Recursiva**: o processo se repete para os conjuntos de triângulos em cada semi-espaço até que todos os triângulos sejam classificados.

O algoritmo inicia selecionando o primeiro triângulo da lista para definir o plano de partição inicial, calculado pela função `make_plane`. Em seguida, cada triângulo restante é classificado em COPLANAR, POSITIVE, NEGATIVE ou CRUZANTE. Triângulos coplanares são mantidos em `node.triangles`; triângulos positivos e negativos são enviados para as listas `pos_tris` e `neg_tris`, respectivamente; já triângulos que cruzam o plano são subdivididos em até três sub-triângulos (ou dois, no caso de um vértice coplanar) pela função `split_triangle`, que calcula pontos de interseção nas arestas. Cada sub-triângulo resultante é então reclassificado e direcionado ao semi-espaço correspondente, continuando o processo recursivo. Por fim, `build_bsp` é chamada recursivamente para `pos_tris` e `neg_tris`, instanciando os nós `positive_child` e `negative_child`, e retorna o nó completo.

3.6 Consulta de Interseções

A operação de consulta na BSP Tree percorre recursivamente a estrutura para coletar os índices de todos os triângulos que intersectam um segmento dado. Em cada nó, testamos primeiro as interseções locais com os triângulos armazenados, depois usamos a classificação dos extremos do segmento em relação ao plano do nó para decidir quais subárvores visitar.

Algorithm 4 Traversal da BSP Tree

```
1: function TRAVERSE_BSP(segment, node, result_set)
2:     if node = None then
3:         return
4:     for cada (tri_idx, tri) em node.triangles do
5:         if intersect_segment_triangle(segment, tri) then
6:             result_set.add(tri_idx)
7:     p0 ← primeiro ponto de segmento
8:     p1 ← segundo ponto de segmento
9:     side0 ← classify_point(n.plane, p0)
10:    side1 ← classify_point(n.plane, p1)
11:    if side0 ∈ {POSITIVE, COPLANAR} e side1 ∈ {POSITIVE, COPLANAR}
       then
12:        traverse_bsp(segment, node.positive_child, result_set)
13:    else if side0 ∈ {NEGATIVE, COPLANAR} e side1 ∈
       {NEGATIVE, COPLANAR} then
14:        traverse_bsp(segment, node.negative_child, result_set)
15:    else
16:        traverse_bsp(segment, node.positive_child, result_set)
17:        traverse_bsp(segment, node.negative_child, result_set)
```

A função `traverse_bsp` inicia verificando se o nó atual é nulo, encerrando a recursão se for o caso. Em seguida, itera sobre todos os triângulos armazenados no nó, testando a interseção com o segmento e adicionando ao conjunto de resultados aqueles que intersectam. Depois, classifica cada extremidade do segmento em relação ao plano de partição do nó. Se ambos os pontos estiverem no mesmo lado (positivo ou coplanar, ou negativo ou coplanar), a função desce apenas na subárvore correspondente. Caso contrário, o segmento cruza o plano e é necessário visitar ambas as subárvores para garantir que nenhuma interseção seja perdida.

4 Validação Experimental

O algoritmo foi validado com 30 casos de teste que incluem:

- **Triângulos em várias orientações**: Planos paralelos, perpendiculares e aleatórios
- **Segmentos especiais**: Paralelos a planos, coplanares a triângulos
- **Casos degenerados**: Triângulos colineares, segmentos de comprimento zero