**Principle documentation**

**Sunsiy10**

**2014.12.25**

This work realizes the Boolean operation of two-dimensional general polygons, and its principle is based on the merge operation of the spatial binary partition tree (BspTree) in two-dimensional space, referring to the literature [1] and [2]. The existing algorithm is used to generate the edges of the resulting general polygon [3]. Here's a closer look at how it works and how this exercise leverages these principles and algorithms.

1. **Introduction to BspTree**

The principle of BspTree is simple, its essence is a binary tree. Each intermediate node of the tree represents a two-dimensional region, and contains a partition of a two-dimensional region; The two subregions are represented by the left and right children of the middle node, respectively. Each leaf node of the tree represents an area of a two-dimensional plane.

p

h+

h-

p

Region r

h+ h-

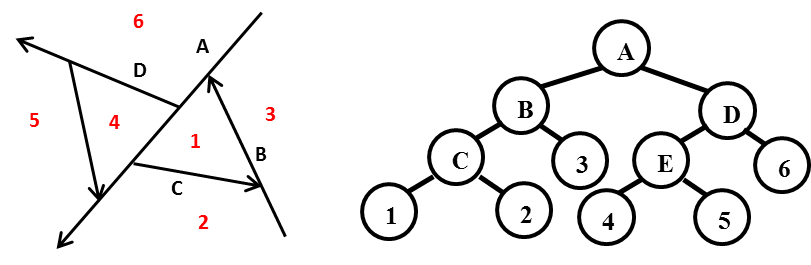
a. A division in the regions r and r, b. the corresponding BspTree in a

Figure 1 Introduction to Bsptree

We use a small example diagram to more clearly illustrate how BspTree works, as shown in Figure 1. The rectangle in figure A represents a region R, and there is a division P in R that divides R into two subregions H+ sum h-, the corresponding BspTree is shown in figure b. Each division has a direction, here I specify that the left side of the division is the + area and the right side is the - area. Correspondingly, for any intermediate node n in the BspTree, the left child indicates that n divides the left subregion, and the right child indicates that n divides the right subregion.

1. **Use BspTree to represent the area of** a simple polygon

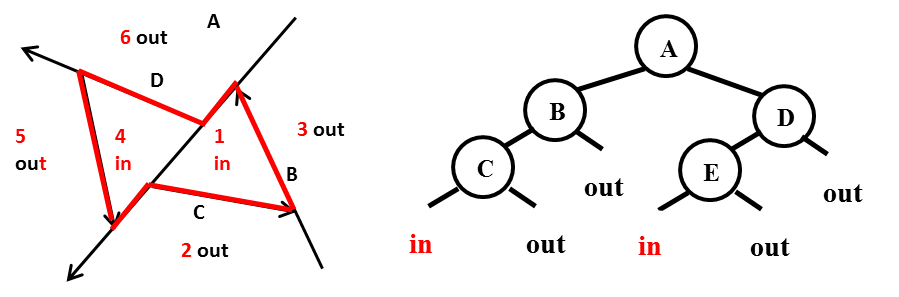
Based on the nature of BspTree and my previous rules, BspTree can be used to represent areas in space, as shown in Figure 2. In Figure 2, the areas are represented by numbers and the divisions are represented by letters.



a. Spatial segmentation of the region b Use BspTree to represent each area

Figure 2 BspTree is used to represent the spatial area

For simple polygons consisting of regions 1 and 4, if BspTree is used to represent the area of this polygon, an attribute should be added to the leaf node to indicate that the area represented by the current leaf node belongs to this polygon, as shown in Figure 3. We add the in/out position attribute to the leaf node, in means this area is inside the target polygon, and out means this region is outside the target polygon. The BspTree in Figure b in Figure 3 represents a simple polygon region consisting of regions 1 and 4.



BspTree representation of the target simple polygon area in b.a

Figure 3 A simple polygon region is represented by BspTree

1. **Bsptree's Merge operation**

After obtaining the BspTree representation of two simple polygons, the BspTree merge operation can be used to merge the two BspTrees to achieve Boolean operations between the polygons.

**3.1 Introduction to the merge algorithm**

When Merge two BspTrees (set to A and B), select one as the split BspTree and one as the split BspTree。 You might as well let A be the split BspTree and B be the split BspTree. The final result: BspTree is set to C.

BspTree's Merge algorithm flows as follows:

**MergeBspTree:(A, B:BspTree)->BspTree**

**::=**

**Types**

**PartitonedBspTree:(negTree, posTree:BspTree)**

**Imports**

**Merge\_Tree\_With\_Cell:(A,B:BspTree) ->BspTree**

**Partition\_BspTree:(BspTree,Partition)->PartitionedBspTree**

**Definition**

**IF A.is\_a\_cell OR B.is\_a\_cell**

**THEN**

**VAL := Merge\_Tree\_With\_Cell(A,B)**

**ELSE**

**Partition\_BspTree(B, A.partition)->B\_partitioned**

**VAL.neg\_subtree := Merge\_BspTree(A.neg\_subtree, B. negTree)**

**VAL.pos\_subtree := Merge\_BspTree(A.pos\_subtree, B. posTree)**

**VAL.attribute = A. attribute**

**END IF**

**RETURN VAL**

**END MergeBspTree**

The algorithm uses a recursive idea to first determine whether A and B have a leaf node. If A and B have a leaf node, call the Cell merge method, which includes processing for different types of Boolean operations. If neither A nor B is a leaf node, then divide B into two sub-BspTrees with , denoted B.posTree and B.negTree, respectively. This is followed by a recursive merger of the left child and B. posTree, and the right child and B. negTree.

This is followed by a careful description of the Partition\_BspTree and Merge\_Tree\_With\_Cell methods.

**3.2 Divide the BspTree with a division**

Here Partition\_BspTree method is described in detail.

When splitting a BspTree(T) with a partition (P), P is split from the root node of T, and the recursive idea is used to gradually obtain two split results neg Tree, posTree。

**3.2.1 Split intermediate nodes**

When splitting to one of T's intermediate nodes, target P and node.partition (T.p). Different position relationships have different segmentation strategies.

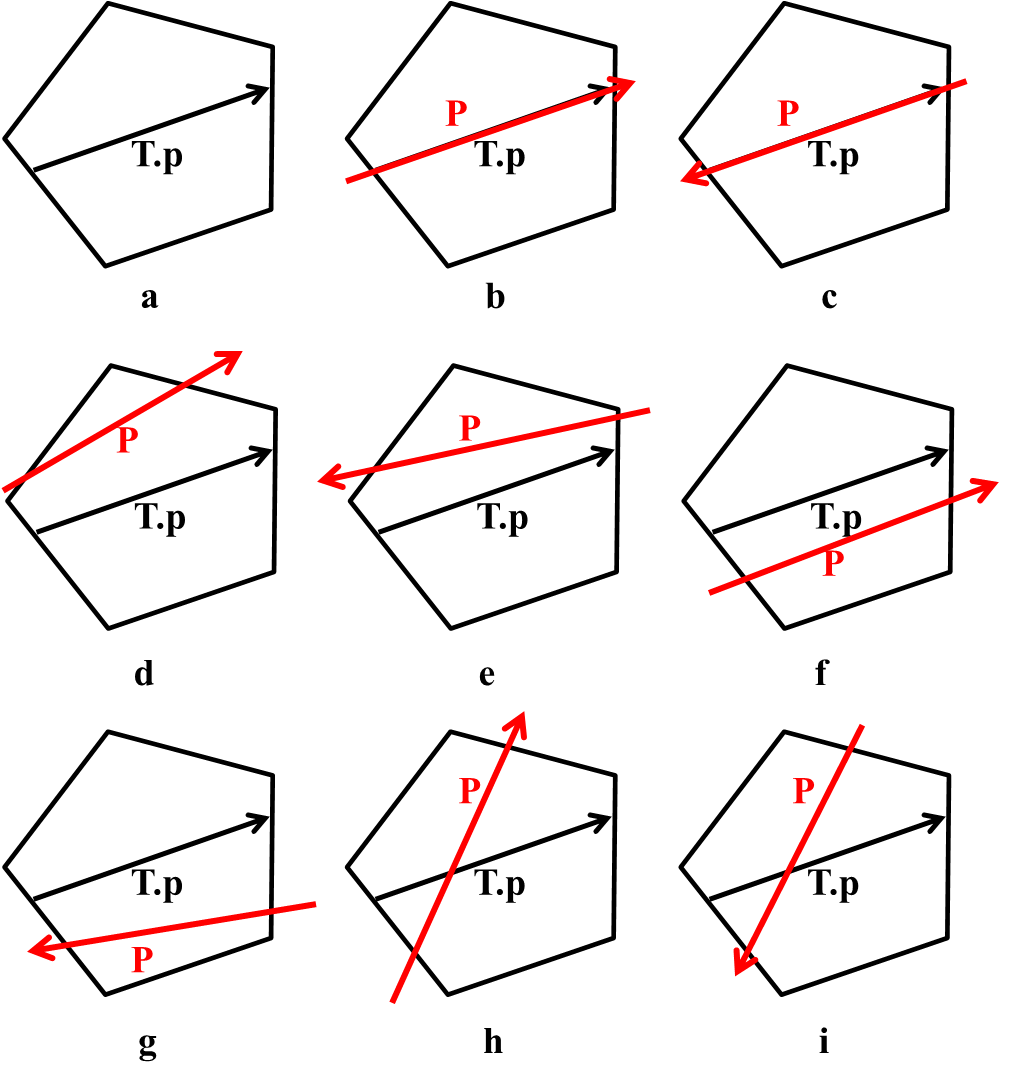


Figure 4 Position relationship between P and T.p

There are a total of eight position relationships between P and T.p, as shown in the b to i plot in Figure 4. Figure A in Figure 4 shows the area and the included division corresponding to the intermediate node to be split. The following describes the segmentation strategy used for each case in Partition\_BspTree method.

**Case 1: P coincides with T.p and is in the same direction (Figure 4b)**

posTree = T.pos\_subtree

negTree = T.neg\_subtree

**Case 2: P coincides with T.p and reverses (Figure 4c)**

posTree = T.neg\_subtree

negTree = T.pos\_subtree

**Case 3: P to the left of T.p and T.p to the right of P (Figure 4d)**

pos Tree = T.pos\_subtree.in\_P\_pos //in\_P\_pos represents the part to the left of P, here

It means that the T.pos\_subtree is left after being split by P

BspTree on the edge

negTree. neg\_subtree = T. pos\_subtree. in\_P\_neg

in\_P\_neg is represented in the part to the right of P, here

It means that the T.pos\_subtree is split by P to the right of P

BspTree on the edge

negTree. neg\_subtree = T.neg\_subtree

negTree.partition = T.partition

**Case 4: P to the left of T.p and T.p to the left of P (Figure 4e)**

postTree.pos\_subtree = T.pos\_subtree.in\_P\_pos

posTree.neg\_subtree = T.neg\_subtree

posTree.partition = T.partition

negTree = T. pos\_subtree. in\_P\_neg

**Case 5: P is to the right of T.p and T.p is to the left of P (Figure 4f)**

posTree.pos\_subtree = T.pos\_subtree

posTree.neg\_subtree = T.neg\_subtree. in\_P\_pos

posTree.partition = T.partition

negTree = T. neg\_subtree. in\_P\_neg

**Case 6: P is to the right of T.p and T.p is to the right of P (Figure 4 g)**

posTree = T.neg\_subtree.in\_P\_pos

negTree.pos\_subtree = T. pos\_subtree

negTree.neg\_subtree = T.neg\_subtree.in\_P\_neg

negTree.partition = T.partition

**Case 7: P intersects T.p and P goes from T.p right to T.p to the left (Figure 4h)**

posTree.pos\_subtree = T.pos\_subtree. in\_P\_pos

posTree.neg\_subtree = T.neg\_subtree. in\_P\_pos

posTree.partition = T.partition

negTree.pos\_subtree = T. pos\_subtree.in\_P\_neg

negTree.neg\_subtree = T.neg\_subtree.in\_P\_neg

negTree.partition = T.partition

**Case 8: P intersects T.p and P goes from T.p to the right of T.p (Figure 4i)**

posTree.pos\_subtree = T.pos\_subtree. in\_P\_pos

posTree.neg\_subtree = T.neg\_subtree. in\_P\_pos

posTree.partition = T.partition

negTree.pos\_subtree = T. pos\_subtree.in\_P\_neg

negTree.neg\_subtree = T.neg\_subtree.in\_P\_neg

negTree.partition = T.partition

During partitioning, the implementation of in\_P\_pos and in\_P\_neg implies that Partition\_BspTree method is called, so Partition\_BspTree is a recursive method.

**3.2.2 Split leaf nodes**

When splitting to one of the leaf nodes of T, two identical leaf nodes are returned. The position attribute that returns the leaf node is the same as the position attribute of the split leaf node.

**3.3 Leaf merging algorithm**

The leaf merging method is intuitive, and operations related to Boolean operations are included in the leaf merging algorithm. The original algorithm of the reference paper [1] had errors, which were corrected here. The algorithm is as follows:

**Merge\_Tree\_With\_Cell:(A,B:BspTree) -> BspTree**

**::=**

**VAL :=**

**IF A.is\_an\_InCell**

**THEN**

**CASE operation**

**Union -> A**

**Intersection -> B**

**Difference -> Complement\_BspTree(B)**

**END**

**ELSE IF A.is\_an\_OutCell**

**THEN**

**CASE operation**

**Union -> B**

**Intersection -> A**

**Difference -> A**

**END**

**ELSE IF B.is\_an\_InCell**

**THEN**

**CASE operation**

**Union -> B**

**Intersection -> A**

**Difference -> OutCell**

**END**

**ELSE**

**THEN**

**CASE operation**

**Union -> A**

**Intersection -> B**

**Difference -> A**

**END**

**END Merge\_Tree\_With\_Cell**

1. **Gets the result of a polygon Boolean operation**

After the merge operation of BspTree, the result BspTree needs to be further processed, and the division in BspTree is eliminated and clipped accordingly, so as to obtain the edge of the polygon as the final Boolean operation.

When generating the edges of the resulting polygon, this paper uses the method of reference [3]. The idea of implementing the method is as follows:

Based on this principle, iterate through all the leaf nodes of Bsptree (each node represents an area), record the boundaries of the area represented by these leaf nodes, and judge each edge that makes up these boundaries. If an edge is both an edge of an inner region and an edge of an outer region, then that edge belongs to the boundary of the resulting polygon.

For specific implementation, it is necessary to add 4 side lists to the node data structure of BspTree, called leftIn, leftOut, and rightIn , rightOut。 The leftIn record divides the edge of the boundary that contributes to the inner region to its left, and the leftOut record divides the edge of the boundary that contributes to the outer region to its left, rightIn, rightOut Similar to this.

When processing the edges of the resulting polygon on the resulting BspTree, there are two steps, that is, two functions are called one after the other:

gb\_generateCellPolygons

gb\_generateBSPTreeFaces

Among them gb\_generateCellPolygons recursively traverse the leaf nodes of each BspTree and extract the boundaries of the corresponding areas of the leaf nodes. The boundary is composed of some parts of the division of the intermediate nodes of BspTree, and the divided parts of these constituent boundaries are saved in the 4 lists of the corresponding intermediate nodes Middle.

The gb\_generateBSPTreeFaces then recursively traverses the intermediate nodes of each BspTree, either in leftIn and rightOut or together Take out the edges of leftOut and rightIn to get the boundary of the final polygon.

1. **Implement process and** **complexity analysis**

**5.1 Implementation Process**

**5.1.1 Boolean function implementation**

The implementation process of the function is mainly divided into three steps, namely:

Step 1: Convert general polygons to BspTree;

Step 2: Merge BspTree with bool operation;

Step 3: Get the edges of the resulting polygon.

The first step is to convert a general polygon to BspTree, and the process is as follows:

For any region in the general polygon, convert all the rings in the middle to BspTree, and then use the BspTree of the outer ring to do Boolean differences with the BspTree of each inner ring in turn BspTree. Then the BspTree of all regions of the general polygon is combined with a Boolean operation to obtain the BspTree of the general polygon.

When converting a ring to BspTree, first take one edge s in the ring and divide the rest of the edge into two parts, L and R, with the L part at edge s To the left of the line, the R part is to the right of the line where the edge s is located. (If the remaining edges intersect the line where the edge s is located, the side is divided into two segments and then placed in the corresponding part) BspTree root node division. L's and R's are then treated in the same way to get the left and right children of BspTree, and continue to recurse until L and R If both are empty, add two children (leaf nodes) to the current node, the left child is marked as in, and the right child is marked as out.

The second and third steps are implemented according to the algorithms introduced in the previous sections 3 and 4, respectively.

**5.1.2 Legality Check**

The legality check is mainly divided into three parts: the legality check of the ring, the legality check of the region and the legality check of the general polygon.

1. Ring legality check

The validity check of the ring mainly considers two questions, namely whether the direction is correct and whether the nonadjacent edges in the ring have intersections (including partial coincidents). Among them, when judging the correctness of the direction, according to the order of the points recorded in the ring data structure, find the point with the smallest coordinate in the first X direction, and judge the direction of the ring according to this point.

1. Legality check of the zone

There are four points to consider when checking the legitimacy of a zone: first consider the legality of each ring; Consider whether the edges between rings have intersections, except for the case where vertices coincide; Consider whether there are overlapping regions with an area greater than 0 between any two inner rings; Finally, consider whether the inner ring is inside the outer ring. The last two points can be judged by using the Boolean operations "cross" and "difference" respectively, as detailed in section 2.2 of the development technical document.

1. Legality check for general polygons

There are three points to consider when checking the legitimacy of general polygons: first consider the legality of each region; Then consider whether there is an intersection between the regions except for the case where the vertices coincide; Finally, consider whether there are coincident regions with an area greater than 0 between any two inner rings. The last point can be judged by the Boolean operation "intersection", which can be found in section 2.2 of the development technical document.

**5.2 Complexity analysis**

Complexity is analyzed here. After counting the two general polygons A, B is represented by BspTree, the number of sides of the tree is sum, and the number of sides is N=.

* + 1. **Complexity analysis of BspTree mergers**

In the case of merging, let A be the split BspTree and B be the split BspTree. The process can be seen in Section 3.1.

Considering the upper bound of complexity, when merging the general polygons A and B represented by BspTree, it is assumed that B is split with the partition of the node in A The partition of each node in A is compared to the partition of all nodes in B at most once, with a complexity of O(| | ), i.e. O(). Each comparison needs to consider all the edges of the region represented by the node in B, and the variables are at most on the order of O(N). So the total complexity is O().

In order to solve the problem of high complexity, the arrangements of hyperplanes method in literature [4] is used to reduce the complexity to Θ(). 。 The literature [4] is Bruce Fountain Naylor's doctoral dissertation, and there are no publicly available resources to query, so this technique is not used in the implementation of the algorithm

But by optimizing the merge process, I reduced the complexity of BspTree merges to O(). The key to optimization is to reduce the complexity of comparing the partition of node A with the partition of node B from the original O(N). O(1)。 The optimization process is described below:

The complexity of comparing the partition of a node in A with the partition of a node in B mainly lies in the need to determine the area represented by the node in B The position relationship in R, the position relationship is shown in Figure 4. It is mainly reflected in two categories:

1. When two partitions have an intersection point X, it is necessary to determine whether the intersection point X is inside the region R. Before optimization, it is necessary to judge the boundary of the sub-intersection point and the region R one by one, and the magnitude of the number of boundary edges is O(N).
2. When both partitions are located in the firing line, the partially directed line segment is inside the region R and the intersection point X is in the region When R is outside, it is necessary to determine the position relationship of the two partitions, as shown in Figure 4(d) indicating the same direction, Figure 4(e). Indicates the opposite direction. In this case, it is necessary to rely on the part of the region where the rays of the two partitions are located (directed line segments) and the intersection point X to determine the direction of the two partitions, as shown in Figure 5. When the directed line segments in the region are all close to X from the start point to the end point or both end points are close to the start point from X, the two partitions are in the same direction; Otherwise, the two partitions are reversed. In this case, it is necessary to use each edge of the boundary of region R to cut the rays where the two partitions are located, and only the directed line segments that remain inside the region R are retained. The complexity of this process is O(N).

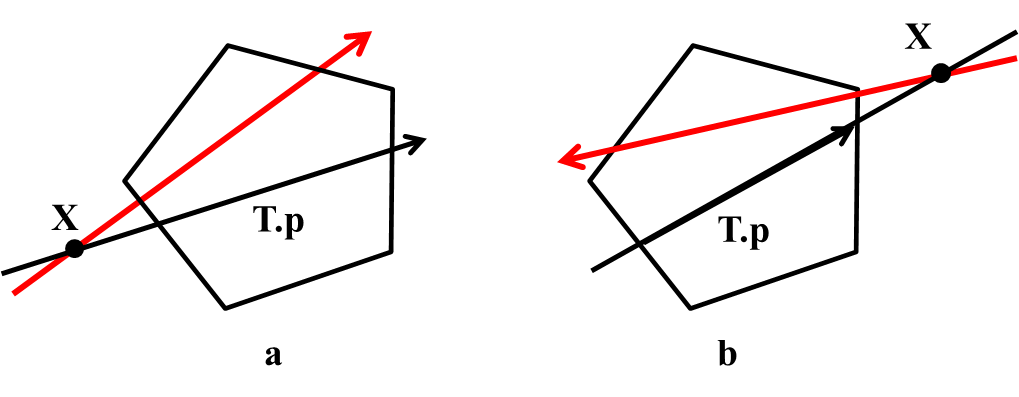


Figure 5 The direction of the two partitions is judged according to the directed line segment and the intersection point X in the region

The following is an optimization scheme for these two cases:

For (1), simply record an additional directed line segment on the ray where the A node partition is located, which is called partitionLine , partitionLine initialized to the whole ray, the direction is the same as the A node partition. When the partition of node A splits a node of B, the partition of the node in B will partitionLine is divided into left and right parts and each part corresponds to the partitioning process for B's child. In this way, each time a node in B is split, the partitionLine is the ray in which the partition in A is located The part inside the region R that the node in B represents (i.e., a directed line segment). So just compare whether the intersection point X is inside the partitionLine to know if the intersection point X is inside the region R.

For (2), the above optimization scheme can already obtain the directed line segment of the partition of node A in the region R, so only need to consider the node in B partition。 The scheme here is similar to the one that handles (1), except that initialization occurs when the BspTree is built, i.e. the ring is converted to a BspTree. Because when converting a ring to a BspTree, because each node that generates a BspTree needs to divide the remaining edges in the ring into left and right categories The complexity of each generated BspTree node is O(), ( is the number of sides of the ring ) plus the initialization process ( With the complexity of O()), the complexity remains O(), so the complexity of converting the ring to BspTree does not change.

* + 1. **The edge of the resulting polygon is obtained**

The complexity of obtaining the polygon Boolean operation result is mainly the complexity of the gb\_generateCellPolygons, which is mainly related to the number of leaf nodes and the distance from the leaf node to the root node of the merged BspTree. The number of leaf nodes n of the merged BspTree is between O(N) and O(), considering the binary tree structure, when n is closer O(N), the more unbalanced the tree, the distance from the leaf node to the root node is close to O(N); When n is closer to O(), the tree is more balanced, and the distance from leaf node to root node is closer to O(). Since the total complexity is the sum of the product of each leaf node and its distance from the root node, the total complexity is between and .

* + 1. **Converts general polygons to BspTree**

The complexity of converting each ring to BspTree is O().

The complexity of converting a region to a BspTree is the complexity of each ring transformation in the region plus the complexity of the Boolean "difference" between the rings.

The complexity of converting general polygons to BspTree is the complexity of each region conversion plus the complexity of the Boolean operation "intersection" between regions.

**Bibliography:**

[1] Naylor B, Amanatides J, Thibault W. Merging BSP trees yields polyhedral set operations[C]//ACM SIGGRAPH Computer Graphics. ACM, 1990, 24(4): 115-124.

[2] Thibault W C, Naylor B F. Set operations on polyhedra using binary space partitioning trees[C]//ACM SIGGRAPH Computer Graphics. ACM, 1987, 21(4): 153-162.

[3] Thibault W C. Application of binary space partitioning trees to geometric modeling and ray-tracing[J]. Ph.D. Dissertation, Georgia Institute of Technology, Atlanta, Georgia, 1987.

[4] Naylor B F. A priori based techniques for determining visibility priority for 3-d scenes[J]. Ph.D. Thesis, University of Texas at Dallas 1981.