**原理文档**

**Sunsiy10**

**2014.12.25**

本次作业实现了二维一般多边形的布尔运算，其原理基于二维空间的空间二叉分割树（BspTree）的合并（Merge）操作，参考了文献[1]和[2]。在生成结果一般多边形的边时采用了现有的算法[3]。下面详细介绍工作原理以及本次作业是如何利这些原理和算法实现的。

1. **BspTree介绍**

BspTree的原理很简单，其本质是一棵二叉树。树的每个中间节点表示一个二维区域，同时包含一个二维区域的划分（partition）；划分得到的两个子区域分别用中间节点的左右孩子来表示。树的每个叶子节点表示一个二维平面的区域。

p

h+

h-

p

Region r

h+ h-

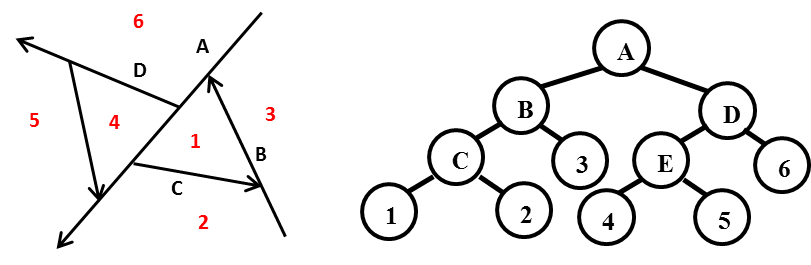
a.区域r和r中的一个划分p b. a中对应的BspTree

图1 Bsptree介绍

我们用一个小的示例图可以更清楚地表示BspTree的工作原理，如图1所示。a图中的矩形表示一个区域r，r中有一个划分p将r分成了两个子区域h+和h-，对应的BspTree即为b图中所示。每个划分都有一个方向，这里我规定划分的左边为+区域，右边为-区域。与此相对应，对于BspTree中的任意一个中间节点n，其左孩子表示n划分左边的子区域，右孩子表示n划分右边的子区域。

1. **用BspTree表示简单多边形的区域**

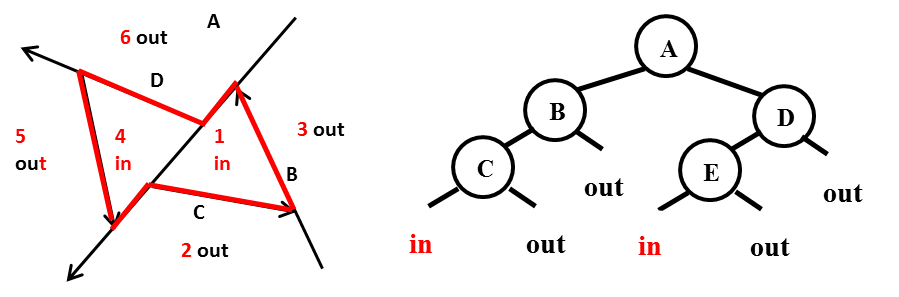
根据BspTree的性质和我之前的规定，便可以用BspTree来表示空间中的区域，如图2所示。图2中用数字表示区域，用字母表示划分。



a.区域的空间分割 b.用BspTree表示各个区域

图2 用BspTree表示空间区域

对于区域1和区域4组成的简单多边形，若用BspTree表示此多边形的区域，则应为叶子节点增加一个属性来表示当前叶子节点所表示的区域属于此多边形，如图3所示。我们为叶子节点加上in/out位置属性，in表示此区域在目标多边形内部，out表示此区域在目标多边形外部。图3中b图中的BspTree表示的就是区域1和区域4组成的简单多边形区域。



a.目标简单多边形(红色边界) b.a中目标简单多边形区域的BspTree表示

图3 用BspTree表示简单多边形区域

1. **Bsptree的Merge操作**

在得到两个简单多边形的BspTree表示形式后，便可用BspTree的Merge操作来合并两个BspTree，从而实现多边形之间的布尔运算。

**3.1 Merge算法介绍**

在对两个BspTree（分别设为A和B）进行Merge操作时，选择一个作为分割BspTree，一个作为被分割BspTree。不妨设A为分割BspTree，B为被分割BspTree。最终的结果BspTree设为C。

BspTree的Merge算法流程如下：

**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**

算法采用递归思想，首先判断A和B是否有一个叶子节点。如果A和B有一个叶子节点，则调用叶子(Cell)合并方法，叶子合并方法中包含了对不同类型布尔运算的处理。如果A和B都不是叶子节点，则用的划分将B分割成两个子BspTree，分别记为B.posTree和B.negTree。随后采用递归的方式合并的左孩子和B.posTree，的右孩子和B.negTree。

随后对Partition\_BspTree和Merge\_Tree\_With\_Cell这两个方法进行仔细的说明。

**3.2 用一个划分分割BspTree**

这里详细介绍Partition\_BspTree方法。

当用一个划分(P)分割一个BspTree(T)时，P从T的根节点开始分割，利用递归思想逐步得到两个分割结果negTree, posTree。

**3.2.1 分割中间节点**

在分割到T的某个中间节点node时，针对P和node.partition(T.p)不同的位置关系有不同的分割策略。

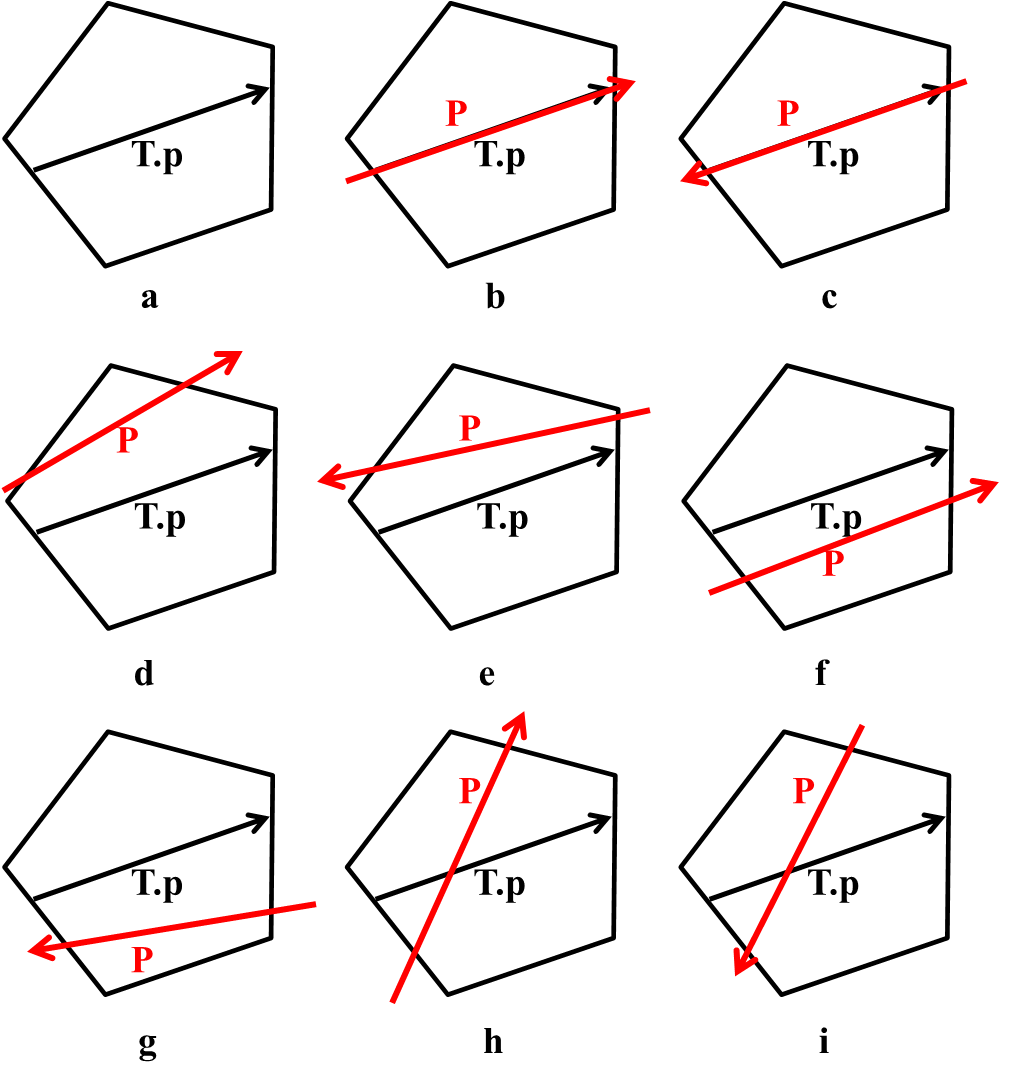


图4 P和T.p的位置关系

P和T.p的位置关系一共有8种，如图4中b到i图所示。图4中a图表示待分割的中间节点对应的区域和包含的划分。下面介绍Partition\_BspTree方法中对每种情况采取的分割策略。

**情况1：P与T.p重合且同向（图4 b）**

posTree = T.pos\_subtree

negTree = T.neg\_subtree

**情况2：P与T.p重合且反向（图4 c）**

posTree = T.neg\_subtree

negTree = T.pos\_subtree

**情况3：P在T.p左边，T.p在P右边（图4 d）**

posTree = T.pos\_subtree.in\_P\_pos //in\_P\_pos表示在P左边的部分，这里

//是指T.pos\_subtree被P分割后在P左

//边的BspTree

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

//in\_P\_neg表示在P右边的部分，这里

//是指T.pos\_subtree被P分割后在P右

//边的BspTree

negTree. neg\_subtree = T.neg\_subtree

negTree.partition = T.partition

**情况4：P在T.p左边，T.p在P左边（图4 e）**

posTree.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

**情况5：P在T.p右边，T.p在P左边（图4 f）**

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

**情况6：P在T.p右边，T.p在P右边（图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

**情况7：P与T.p相交且P从T.p右边进入T.p左边（图4 h）**

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

**情况8：P与T.p相交且P从T.p左边进入T.p右边（图4 i）**

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

在分割过程中，in\_P\_pos和in\_P\_neg的实现意味着调用了Partition\_BspTree方法，所以Partition\_BspTree是一个递归的方法。

**3.2.2 分割叶子节点**

在分割到T的某个叶子节点时，返回两个相同的叶子节点。返回叶子节点的位置属性与被分割叶子节点的位置属性相同。

**3.3 叶子合并算法**

叶子合并方法比较直观，与布尔运算有关的操作都包含在了叶子合并算法中。参考论文[1]原算法有错误，这里进行了改正。算法如下：

**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. **获取多边形布尔运算结果**

经过BspTree的Merge操作后，得到的结果BspTree还需要进一步处理，对BspTree中的划分进行相应的消除和裁剪，从而得到最终的布尔运算结果多边形的边。

在生成结果多边形的边时，本文采用了参考文献[3]的方法。实现方法的思路如下：

结果多边形的任意一条边一定是多边形内部和外部的分割线，基于这个原理，遍历Bsptree的所有叶子节点（每个节点代表一个区域），将这些叶子节点代表的区域的边界记录下来，判断组成这些边界的每条边。如果一条边既为一个内部区域的边又为一个外部区域的边，那这条边就属于结果多边形的边界。

具体实现时，需要在BspTree的节点数据结构中加入4个边的list，分别叫做leftIn，leftOut，rightIn, rightOut。leftIn记录划分对其左边的内部区域贡献的边界的边，leftOut记录划分对其左边的外部区域贡献的边界的边，rightIn, rightOut与此类似。

在对结果BspTree处理生成结果多边形的边时，分为两个步骤，即相继调用两个函数：

gb\_generateCellPolygons

gb\_generateBSPTreeFaces

其中gb\_generateCellPolygons采用递归的方式遍历每个BspTree的叶子节点，并将叶子节点对应区域的边界提取出。边界是由BspTree的中间节点的划分的某些部分组成的，将这些组成边界的划分的部分保存在对应的中间节点的4个list中。

随后gb\_generateBSPTreeFaces采用递归的方式遍历每个BspTree的中间节点，将同在leftIn和rightOut或者同在leftOut和rightIn的边取出，即可得到最终的多边形的边界。

1. **实现过程及复杂度分析**

**5.1 实现过程**

**5.1.1 布尔运算功能实现**

功能的实现过程主要分为三步，分别为：

第一步：将一般多边形转换为BspTree；

第二步：用bool运算合并BspTree；

第三步：得到结果多边形的边。

其中第一步将一般多边形转换为BspTree的过程如下：

对于一般多边形中的任意一个区域，将中所有的环转换为BspTree，然后用外环的BspTree依次与每个内环的BspTree做布尔运算差，即可得到的BspTree。随后将一般多边形中的所有区域的BspTree做布尔运算并，即可得到一般多边形的BspTree。

将环转换为BspTree时，首先取环中的一条边s，将其余的边划分成两部分L和R，L部分在边s所在直线的左边，R部分在边s所在直线的右边。（其余的边若出现与边s所在直线相交的情况，则将此边分成两段，再分别放入对应的部分）边s即为BspTree根节点的划分。随后用相同的方式处理L和R得到BspTree的左孩子和右孩子，并不断递归下去，直到L和R都为空，则为当前节点添加两个孩子（叶子节点），左孩子标为in，右孩子标为out。

第二步与第三步分别按照之前第三大节和第四大节介绍的算法实现。

**5.1.2 合法性检查**

合法性检查主要分为三部分：环的合法性检查，区域的合法性检查和一般多边形的合法性检查。

1. 环的合法性检查

环的合法性检查主要考虑两个问题，分别为方向是否正确以及环中不相邻的边与边是否有交点（包含部分重合的情况）。其中，在判断方向正确性时，依据环数据结构记录点的顺序，找到第一个X方向坐标最小的点，依据此点判断环的方向。

1. 区域的合法性检查

检查区域的合法性时有四点需要考虑：首先考虑每个环的合法性；再考虑环之间除顶点重合的情况外，环之间的边是否有交点；再考虑任意两个内环之间是否有面积大于0的重合区域；最后考虑内环是否在外环内部。其中最后两点可分别利用布尔运算“交”和“差”来判断，具体情况可见开发技术文档2.2小节。

1. 一般多边形的合法性检查

检查一般多边形的合法性时有三点需要考虑：首先考虑每个区域的合法性；再考虑区域之间除顶点重合的情况外，边是否有交点；最后考虑任意两个内环之间是否有面积大于0的重合区域。其中最后一点可用布尔运算“交”来判断，具体情况可见开发技术文档2.2小节。

**5.2 复杂度分析**

这里分析复杂度。计两个一般多边形A，B用BspTree表示后，树的边数分别为和，计N=。

* + 1. **BspTree合并的复杂度分析**

在合并时，设A为分割BspTree，B为被分割BspTree。过程可见3.1小节。

考虑复杂度上界，在合并BspTree表示的一般多边形A和B时，不妨假设用A中节点的partition分割B，则A中的每个节点的partition最多与B中的所有节点的partition比较一次，复杂度为O(||)，即O()。而每比较一次都需要考虑B中节点所表示区域的所有边，变数最多是O(N)数量级的。所以总的复杂度为O()。

文献[1]为了解决复杂度高的问题，利用了文献[4]中的arrangements of hyperplanes方法将复杂度降为ϴ()。文献[4]为Bruce Fountain Naylor的博士论文，没有公开的资源可查询，故在实现算法时没有用到这一技术

但是通过对合并过程的优化，我将BspTree合并的复杂度降到了O()。优化的关键在于把A中节点的partition与B中节点的partition比较一次的复杂度由原来的O(N)降为O(1)。下面讲述优化过程：

A中节点的partition与B中节点的partition比较的复杂度主要在于需要判断这两个partition在B中节点所表示区域R中的位置关系，位置关系如图4所示。主要体现在两大类情况：

1. 当两个partition存在交点X时，需要判断这个交点X是否在区域R内部。优化前，需要将次交点与区域R的边界逐个做判断，而边界边数的量级为O(N)。
2. 当两个partition所在射线都存在部分有向线段在区域R内部且交点X在区域R外部时，需要判断两个partition的位置关系，如图4(d)表示方向相同，图4(e)表示方向相反。这时需要依据两个partition所在射线在区域内部的部分(有向线段)以及交点X来判断两个partition的方向，如图5所示。当区域内部的有向线段都是起点相比终点距离X近或者都是终点相比起点距离X近时，两个partition同向；否则两个partition反向。这时需要用区域R的边界的每条边来切割两个partition所在射线，只保留在区域R内部的有向线段。这个过程的复杂度为O(N)。

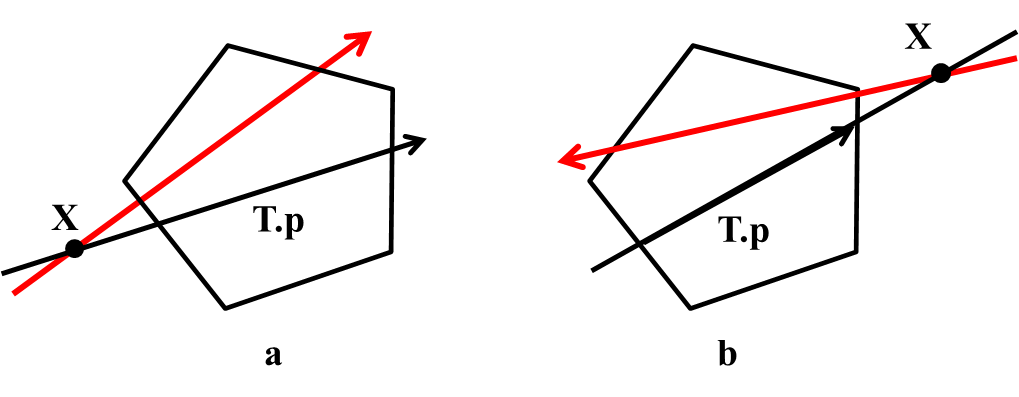


图5 依据区域内部有向线段以及交点X来判断两个partition的方向

下面针对这两种情况，依次给出优化方案：

对于(1)，只需额外记录一个在A节点partition所在射线上的有向线段，记其为partitionLine，partitionLine初始化为整条射线，方向与A节点partition一致。在A节点的partition分割B的某个节点时，依据B中节点的partition将partitionLine分成左右两部分并将每部分对应用于B的子孩子的分割过程。这样下来在每次分割B中节点时，partitionLine就是A中partition所在射线在B中节点所表示区域R内部的部分（即一条有向线段）。所以只需比较交点X是否在partitionLine内部就可知道交点X是否在区域R内部。

对于(2)，上面的优化方案已经可以得到A中节点的partition所在摄像在区域R中的有向线段，所以只需要考虑B中节点的partition。这里的方案与处理(1)的方案类似，不同之处在于初始化发生在BspTree建立时，即将环转换成BspTree的过程。由于在将环转换成BspTree时，由于每生成一个BspTree的节点时都需要将环中剩余的边分成左右两类，所以每生成一个BspTree节点的复杂度为O()，(为环的边数)在加上初始化过程(复杂度为O())后，复杂度仍为O()，所以将环转换成BspTree的复杂度没有变化。

* + 1. **得到结果多边形的边**

获取多边形布尔运算结果的复杂度主要为gb\_generateCellPolygons的复杂度，主要与合并后的BspTree的叶子节点数和叶子节点到根节点的距离有关。而合并后的BspTree叶子节点数n介于O(N)和O()之间，考虑二叉树结构，当n越接近O(N)时，树越不平衡，叶子节点到根节点的距离接近O(N)；当n越接近O()时，树越平衡，叶子节点到根节点的距离接近O()。由于总复杂度为每个叶子节点与其到根节点距离的乘积之和，故总复杂度介于和之间。

* + 1. **将一般多边形转换为BspTree**

将每个环转换成BspTree的复杂度均为O()。

将区域转换成BspTree的复杂度为区域内每个环转换的复杂度加上环之间布尔运算“差”的复杂度。

将一般多边形转换成BspTree的复杂度为每个区域转换的复杂度加上区域之间布尔运算“交”的复杂度。

**参考文献：**

[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.