# Problem.

We have learned the plane sweep algorithm, which can find all the intersections of line segments in , where is the number of the line segments and is the number of the intersection points. This algorithm is useful if we deal with only the static set of line segments, but many geometric applications such as motion planning have dynamic updates in the line segments set. Every time when a new line segment is added or the geometry of an existing line segment changes, this approach requires time of computation to find the intersections, which is undesirable. In this final project, I propose kd-tree data structure for line segments to efficiently detect the line segments intersection for a given line segment against the other line segments in time in the practical cases. Our cost function finds the optimal separating plane for kd-tree construction, which improves the query performance.

# Building kd-tree for line segments

Kd-tree is a space partitioning data structure for points, but it can also be used for lines with some modification. Given line segments over the two dimensional space, a kd-tree is built by the recursive scheme as shown in Fig. 1.

**function** TreeNode::insert(LineSegment *L*)

**if** *L* is located completely on the left side of this node’s plane *p* **then**

insertLeft(*L*)

**if** *L* is located completely on the right side of this node’s plane *p* **then**

insertRight(*L*)

**if** *L* crosses this node’s plane *p* **then**

(, ) = split(*L*, *p*)

insertLeft()

insertRight()

**function** TreeNode::insertLeft(LineSegment *L*)

**if** exists **then**

->insert(*L*)

**else**

*addLeafNode*(*L*)

**function** TreeNode::insertRight(LineSegment L)

**if**  exists **then**

->insert(*L*)

**else**

*addLeafNode*(*L*)

Figure 1. The pseudo code to build a kd-tree for line segments

The major difference of the kd-tree for line segments compared to the one for points is that it has to deal with a case when a line segment crosses a separating plane. To deal with this case, we have to split a line segment, and traverse the both child nodes. In the worst case, the line segment crosses most of the separating planes, and we have to traverse almost all the tree nodes. The naïve solution would be to use a spatial median splitting, in which the dimension is chosen I round robin fashion, and the plane is positioned at the spatial median of the space. This approach is simple, but it does not avoid the line splitting at all. To address this issue, my approach uses a cost function which estimates the traversing cost and the intersection computation cost inspired by the SAH based approach [1]. The cost of adding a separating plane is defined as follows:

where is a cost for a node traversal, is the probability to traverse to the left child node, and is the probability to traverse to the right child node. For each subdivision of a space, the local optimal plane which minimizes the above cost function is chosen.

# Tree traversal for the intersection detection

Given a line segment , our kd-tree data structure can efficiently find the intersections. It starts with the root node of the kd-tree by checking whether intersects the line segment of the root node. If it intersects, then we are done. Otherwise, we check if lies entirely on one side of the separating plane, or crosses the plane. For the former case, we go to either left or right child node which completely contains , and recursively check the same things. For the latter case, we split by the separating plane, and go to both left and right child node for the recursive checks. The pseud code of the traversal algorithm is as shown in Fig. 2.

**function** TreeNode::intersects(LineSegment *L*)

**if** this node’s line segment intersects *L* **then**

**return** **true**

**if** *L* is located completely on the left side of this node’s plane *p* **then**

**if**  exists **then**

**return** ->intersects(*L*)

**else**

**return** **false**

**if** *L* is located completely on the right side of this node’s plane *p* **then**

**if**  exists **then**

**return** ->intersects(*L*)

**else**

**return** **false**

**if** *L* crosses this node’s plane *p* **then**

(, ) = split(*L*, *p*)

**if**  exists **then**

**if** ->intersects() **then**

**return true**

**if**  exists **then**

**if** ->intersects() **then**

**return true**

**return false**

Figure 2. The pseudo code to detect the intersection for a given line segment

# Analysis

During the tree construction, each insertion of a line segment traverses the tree from the root to a leaf node. In the ideal case in which no separating plane splits a line segment, it is obvious that the tree constructions takes , where is the number of the line segments and is the height of the tree. Since our tree is balanced, the height of the tree is . Thus, the computation time is in the ideal case.

However, it is unlikely that there is no split in the practical cases. Let be the probability that a line segment intersects a separating plane. Then, the number of nodes that will be traversed is

where is the height of the tree. Thus,

Suppose we have a1000 units 1000 units of two-dimensional space and each line segment has at most 10 units long. Then, the probability is at most . If we have a tree with =30, then will become around 30, which is almost same as the height of the tree. This implies that for the expected computation time, we can achieve in the practical cases.

The intersection detection algorithm traverses the tree from the root to one of the leaf nodes or multiple leaf nodes if the line segment gets split. In the ideal case in which there is no split, the computation time is . Since the tree is balanced, the height of the tree is . Thus, the computation time is in the ideal case. For the average computation time, we can use the same analysis that we did above. Thus, we can achieve for the intersection detection in the practical cases as well.

# Results

First, I compared the computation time of the kd-tree based and the brute force approach of the intersection detection. I used a1000 units 1000 units of two-dimensional space. Each line segment has at most 10 units long, and is distributed randomly over the space. I ran the intersection detection 1000 times to get the average computation time. The result is as shown in Fig. 3.

Figure . The comparison of the query time between the kd-tree and N^2 algorithm.

Second, I compared the computation time of the kd-tree between the cost function based and the spatial median approach.

Figure .

# Conclusion

# Refereces:

[1] WALD, I. AND HARVAN, V. 2006. On building fast kd-Trees for Ray Tracing, and on doing that in O(N log N) In *Proceedings of the 2006 IEEE symposium on interactive ray tracing.* p. 61-69.