The split data structures used to deal with simple geometry problems

盛泽辰 2023K8009915018

1 Background

Computer graphics is the science and art of communicating through computer displays and interactive devices,and the fundamental of graphics is the calculation and storage methods for graphical data. To compute and display vision graaphics efficiently,data structures and technologies such as line segment trees, kd trees, BSP trees, and rectangular window queries play a crucial role. These data structures have been widely utilized in practice and play important roles in various practical application fields such as image processing, memory management, and machine learning. This article aims to synthesize relevant academic papers and explore the basic concepts, characteristics, and applications of these data structures in various fields,and provide practical methods to implement.

Line segment tree is an efficient interval query data structure, developed by J L. Bentley(Bentley, J. L., 1975) first proposed , which was widely used to solve problems such as the maximum, minimum, or sum of any index element. The basic idea is to use the structure of a tree to store the information of each node, including start number, end number, parent node pointer, left child pointer, right child pointer, and sum value. When updating, it is only necessary to trace back from the current node to the root node, with a time complexity of $O(logn)$. When querying, the search is based on the start and end numbers of nodes, with a time complexity of $O(logn)$. Although the construction of line segment trees is relatively complex and has high memory overhead, their efficient interval query capability makes them indispensable in many algorithms. For example, in memory management algorithms based on line segment trees, by constructing a memory management line segment tree, efficient and flexible memory allocation and recycling management can be carried out, significantly reducing the generation of memory fragments. KD tree(K-Dimension Tree) is a data structure designed for searching multidimensional spatial data,which have a similar idea and also play an important role in fields such as image processing and computer graphics. BSP tree, also known as binary space partition tree (Fuchs et al., 1980), is one of the important means used in the field of graphics to accelerate large-scale scene computation, object puncture search, and other problems. During the rendering process, BSP trees can quickly determine which triangles are visible and which are obscured, thereby significantly improving rendering efficiency. Nowadays, it has been widely applied in fields such as virtual reality, 3D modeling, and scientific visualization.

Rectangular window query is a technique for conducting range queries in two-dimensional space,it is one of the operations frequently performed on geometric data structures.The basic idea is to use data structures such as line segment trees, tree arrays, etc. to maintain information in two-dimensional space, in order to quickly answer queries within a given rectangular area. Rectangular window query has a wide range of applications in geographic information systems, image processing, database management, and other fields. For example, in geographic information systems, rectangular window queries can quickly obtain geographic information within a specified area, providing important basis for decision support.

In summary, data structures and techniques such as line segment trees, kd trees, BSP trees, and rectangular window queries play an important role in computer science and computational geometry. They not only enrich the research content of data structures, but also provide powerful tools and methods for multiple fields such as image processing, memory management, and machine learning. With the continuous development of computer technology and the increasing demand for

applications, these data structures and technologies will continue to play a greater role in promoting the development and innovation of related fields. This article will take relevant topics as the starting point to introduce their practical applications and methods in computer and graphics, and explore more efficient practical implementations from this perspective.On the basis of pre-study, we will pay more attention to the efficiency of the algorithms search and structure construction,and show our practical research result by principle explanationand real problems solving method.

2 Literature Review

There has already been significant progress in the field of computational geometry, particularly in the development and optimization of various data structures and algorithms. Previous researchers have achieved numerous excellent results in this field.

One of the most notable advancements is the use of KD trees for efficient multidimensional data retrieval. According to (Bentley, 1975), KD trees have been widely adopted in applications such as nearest neighbor search, collision detection, and ray tracing due to the KD-trees' ability to partition space effectively.(Li et al., 2017)

The line segment tree,a more simple and more compatibile data structure ,also has been proven to be highly effective for range queries in two-dimensional space. As discussed by (Bentley, 1975), line segment trees allow for efficient querying and updating of segments, making them suitable for applications in geographic information systems and computer graphics.

Rectangular window queries, which utilize data structures like tree arrays and line segment trees, have also gained attention for their ability to quickly retrieve information within a specified area. This technique has been applied in various fields, including image processing and database management, as highlighted by (Samet, 1989).

BSP (Binary Space Partitioning) trees as the most popular used geometry algorithms,have been extensively studied for their applications in rendering and visibility determination in computer graphics. IN the paper (Fuchs et al., 1980) demonstrates the effectiveness of BSP trees in managing complex scenes and improving rendering performance.

In this article, we will use the commonality of the above research - data segmentation - as a clue and combine it with practical applications to provide more practical solutions. Overall, the continuous development of these data structures and algorithms has significantly contributed to the advancement of computer graphics. The integration of KD trees, line segment trees, and BSP trees into various applications has not only enhanced computational efficiency but also opened new studying directions for research and innovation in the field.

3 Methodology

In this research, we verified the practical feasibility and performance of the discussed data structures in this article (line segment tree, kd-tree, and slice search) primarily through coding and testing. Some simple code examples are provided in the appendix to demonstrate the basic principles and applications of these data structures. Also, we will conduct a comparative analysis of the query time consumption between the kd-tree decomposition method and the overall reorganization method to verify the effectiveness of our method in reducing query time cost.

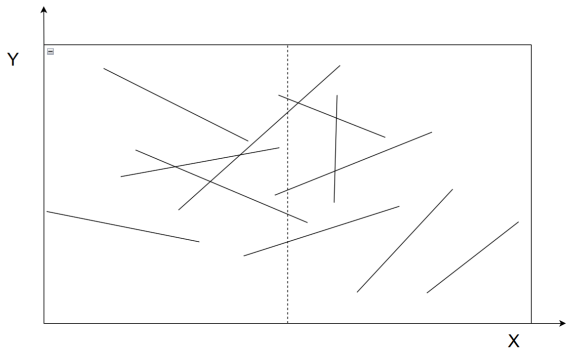

Figure 1: Example of Line Segment Problem

4 Lead-in:2D puncture query

Firstly, introduce the 2D puncture query as a simple introduction to the core idea of this algorithm.

Assuming that during a website's daily operation, we have already recorded all users' online time slots. Each recorded time period can be regarded as a line segment from the starting point to the ending point. Now consider obtaining a real-time traffic map of users based on the data collected throughout the day, displaying with an accuracy of 5 seconds how many users are online at a certain time of the day. In a whole day, there are 86400 seconds, and with an accuracy of 5 seconds, 17280 point values need to be calculated. So, how to efficiently calculate the number of online users at a certain moment? This idea can be abstracted as the following question, and Figure 1 showed this problem:

Input:A set of S line segments on a plane

Target:Vertical line segment l

Output:All s in S intersects with l

For such a set of line segments S, given the endpoints p, q of each line segment, they are represented in the form of $left(x_1, y_1\ right)$ and $left(x_2, y_2\ right)$. Namely:

$S = \{l_1, l_2, \cdots l_n\} = \{(p_1, q_1), (p_2, q_2), \cdots (p_n, q_n)\}$

This clearly marks the two endpoints of the line segment and the dependency relationship between the line segment and its corresponding endpoints. The question that needs to be answered now is: Given a straight line y parallel to the y axis, find the intersection segment between it and the set of line segments S. Obviously, the most intuitive method is to have the line $x = a$ intersect with all the line segments in the set S in sequence, making it easy to obtain all the intersecting line segments. The query time of this method is $O(n)$, which is proportional to the number of line segments in S. This method can be further optimized through line segment trees to reduce it to $O(logn)$ similar to binary search.

Firstly, consider storing the set of line segments S in a tree like data structure.

For this set, we know the endpoints of all line segments. Therefore, using endpoints as spatial features, a one-dimensional space $[p_2min, q_2max]$ is divided.

Based on the form of a binary tree, it is natural to consider using the median as the segmentation point for recursive construction. Sort the endpoints of $2n$ line segments in S. Take the median point and store it in this root node. If it is less than this value, enter the left subtree,

and if it is greater than this value, enter the right subtree. Naturally, there are no size related changes during the construction process, so only one sorting is needed, When the value is less than or greater than the median and there are no endpoints, terminate the recursion and record it as a leaf node.

To answer the puncture query of vertical line segments, it is also necessary to store the relevant information of the line segments in the nodes. Record each node as u . The segmentation information of nodes is stored in $u.split$. Each node has now saved the segmentation information of its left and right subtrees, the spatial range contained in this node can be determined by the $u.split$ value of the predecessor node, denoted as $[u_{min}, u_{max})$. This is a left closed and right open interval. If the node is the predecessor's left node, then this node can only obtain a right boundary of $u.parent.split$, so it still needs to continue to seek the left boundary. In fact, this process can be solved during the construction of the tree, as long as a containing interval is maintained during the construction process. Because all line segments have endpoint values of p , q in $(-\infty, +\infty)$, as mentioned earlier, extreme endpoint values can be selected. Define the initial range naturally. Afterwards, in the recursive process of construction, the range interval is updated and passed to the next construction process based on the $u.split$ of each node. Each node can maintain a dynamic array, denoted as $u.lines$, to store the line segments related to that node. To ensure that there are no duplicate line segments l on the search path from the root node to the leaf node, the correlation here can be defined as: the space contained by the node is defined by line segments l . The left and right endpoints are completely included, that is

$[u_{min}, u_{max}) \subseteq [p_i, q_i]$

The line segments that meet this relationship will be stored in this node. This maintains a good property: when a line $x = a$ is given, the sum of these dynamic arrays $u.lines$ traversed during the search process starting from the root node is the result obtained: $s \in S$ intersecting l .

The programming language description of this construction method can be found in Listing 1.

For the already constructed line segment tree, simply start from the root node, compare $u.spilt$ with the target value along the way, traverse the entire tree, and add the $u.lines$ elements from the traversed nodes to the result set R . When the recursive traversal is completed, R contains all the results. There are many ways to implement search methods, but due to article limitations, they will not be detailed explained here.

Note that we only focus on the projection of line segment s on the x axis here. This is because in this problem, we are only concerned about the perpendicular line $x = a$, and the set of line segments S can actually be considered in a one-dimensional case. It can be extended to higher dimensions through simple combinations, and its good performance can be maintained by paying attention to maintaining its binary properties. (Edelsbrunner, Maurer, 1981)

The following text will continue to discuss similar window query problems in two-dimensional situations.

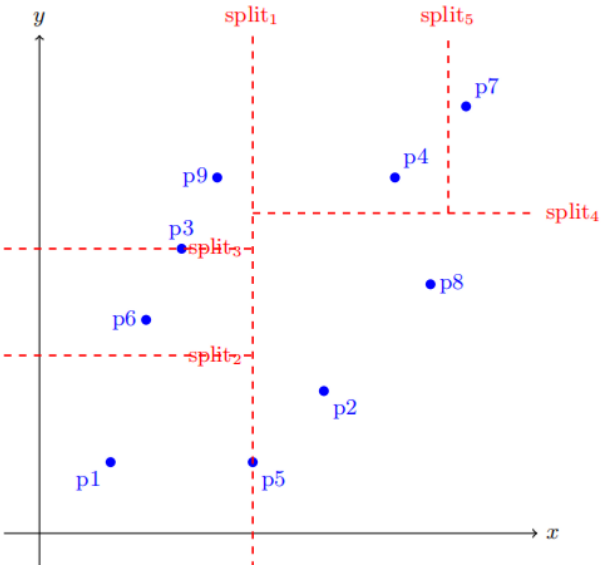

Figure 2: Two-Dimensional kd-Tree Split

5 KD tree:often widely used geometry data structure on 2D plane

Given a set of n discrete points (x_i, y_i) on a two-dimensional plane, and a rectangle with four sides parallel to the coordinate axis plane, query all points within this window. Due to the expansion of the scope to a two-dimensional plane, it is difficult to define the complete order relationship of point coordinates, so the binary search method cannot be used for further processing.

Similar to the query in the one-dimensional scenario mentioned earlier, the two-dimensional kd tree is a natural induction of the one-dimensional scenario, and can efficiently answer such rectangular search problems In a one-dimensional line segment tree, the data is binarized by considering the median of the sorted point set as the partition value for further processing In the two-dimensional case, consider a set of points P on a plane, where elements are denoted as p , We still consider using the divide and conquer method to solve the problem, but we will no longer divide it based on a single x or y axis. By cross using horizontal and vertical lines for partitioning, a well-defined partitioning relationship can be established at each node of the KD tree. The data that each node should store: pointers to the left and right subtrees, the split line *split* (usually determined by the median of the x or y values of the set of points to be processed during construction), and the point (x_i, y_i) stored by this node. During the construction process, we first segment based on the x value of the point set, and then enter the left (right) subtree. At this point, we use the y value for segmentation until there is only one point left in the processing area, store it, and return it. Therefore, all points will be stored in the leaf nodes of the tree. The time consumed by this construction is approximately $O(nlogn)$.

When performing a rectangular window cut query, algorithms can continue searching downwards from the root node based on the already constructed kd-tree. Due to the natural storage of split lines *split* in each node, it is possible to compare *split* with. Then find the relative position relationship of the target rectangle to determine whether to enter the left subtree or right subtree. If the dividing line *split* has already crossed the rectangle, enter the left and right subtrees to

continue searching. Until reaching the leaf node, add the storage point values in the leaf node to the lookup set. According to this search method, all points existing within the rectangular window can be efficiently answered with a time complexity of $O(logn)$.

The code implementation of KD tree and a simple rectangle search demonstration are shown in Appendix 2.

Obviously, KD trees can answer a wider range of geometric search problems(Langetepe , Zachmann, 2006), and here we only use rectangles as a simple demonstration. In fact, efficient response can be achieved for searching any geometric region. In addition, KD trees also play an important role in ray tracing parallel acceleration computation (Li et al., 2014) and large-scale shard processing (Hubo et al., 2006).

Otherwise, in dynamic scenarios, multiple node insertions may cause the KD tree to become significantly unbalanced, and reorganizing the entire tree is a highly costly operation. However, by decomposing the KD tree into KD substructures, the cost of reconstruction can be distributed among smaller substructures. The decomposition method can be determined based on the number of points present in the space: suppose there are N points to be processed in the current space. Consider the binary representation of N: $N = b_m b_{m-1}...b_1 b_0$, where $b_i \in \{0, 1\}$, $N = 2^m b_m + 2^{m-1} b_{m-1} + ... + 2^1 b_1 + 2^0 b_0$. The decomposition of the tree is then determined based on this binary representation. Let the entire structure be W, decomposed into $\sum w_i$, where if $b_i = 0$, then w_i is empty, and if $b_i = 1$, then w_i is a KD tree substructure containing 2^i points. This approach is intuitive and successfully balances search time and reconstruction time.

Under this decomposition, the algorithm for querying all points that meet the criteria remains unchanged, only requiring traversal of all substructures. It is not difficult to prove that the average query time cost of the forest structure after decomposition is no greater than that of $O((\log N)^2)$. In the original entire structure W, N leaf nodes and $N - 1$ intermediate nodes need to be stored, while the forest after binary decomposition requires $N + (\frac{N}{2} - 1) + (\frac{N}{4} - 1) + \ldots + (1 - 1) = N - \log_2 N + \frac{\frac{N}{2}(1-(\frac{1}{2})^{\log_2 N})}{1-\frac{1}{2}} = 2N - 1 - \log_2 N$ nodes, with unchanged space complexity. It exhibits good insertion properties: in the decomposed forest, inserting a new point, the number of points becomes $N + 1$, and adding 1 to the binary decomposition of N results in a carry, meaning that the nodes of the corresponding KD tree will be placed into a higher-level KD tree structure. For example, if there are already 9(1001) nodes, after inserting one node, there are 10(1010) nodes. The carry from b_0 means that w_0 needs to be emptied, and the nodes in w_0 together with the newly added node form w_1. It can be verified that compared to adding several nodes to the entire KD tree and then reorganizing the entire KD tree to achieve balance, this decomposition method achieves an absolutely balanced binary tree forest while maintaining a much lower balancing cost. The query operation needs to be performed separately on all relevant subtrees, and then the results need to be merged. Therefore, the total query time complexity is the sum of all subtree query times,which means:

$$O(1) + O(2) + O(3) + \ldots + O(\log N) = O\left(\sum_{i=1}^{\log N} i\right) = O\left(\frac{(\log N)(\log N + 1)}{2}\right) = O((\log N)^2)$$

In theory, query time increases, but in practical applications, decomposition may bring other optimizations, such as better cache locality, parallel processing, etc. These optimizations may

compensate for the increase in query time in certain scenarios.

The comparison of query times between the KD tree decomposition method and the overall reorganization method, calculated using a Python program, is shown in the figure below: When

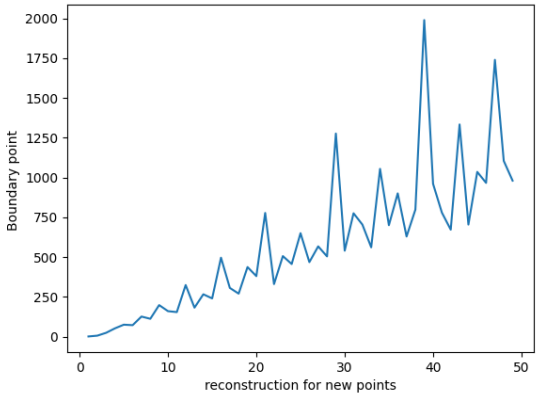

Figure 3: Comparison of Query Time between KD Tree Decomposition Method and Whole Recombination Method

the number of points to be processed is large, the query time of the kd tree decomposition method is significantly shorter than that of the overall recombination method. That is to say, the kd tree decomposition method can effectively reduce the query time cost in the case of a large number of points to be processed, and the time complexity of the query operation is still $O(logn)$.

The insertion of a kd tree only requires searching along the root node of the tree, performing a splitting operation at the leaf node that should be inserted, and constructing left and right child nodes from the original leaf node. The original storage points and the points to be inserted can be stored in two additional child nodes. The deletion operation can be seen as the reverse operation of insertion, which can be easily completed within $O(logn)$ time.

6 Slice search: an example of solving intersection with light rays

The idea of slice search is based on the intersection idea of the first two algorithms. Consider a set of arbitrary geometric polygons on a two-dimensional plane, distributed randomly throughout space, with uncertain area and shape. Given a ray l with its starting point P pointing in any direction, it can be imagined as a beam of light. The question we are interested in now is: Given these sets of graphs G . And how to find all objects intersecting with the light l in a reasonable amount of time? After obtaining the answer to this question, we will be able to easily determine G for l . In terms of visibility (by determining the object that the ray first touches, this object will determine what the ray sees), in addition, the exact position of the intersection between the geometric object and the ray can also be easily obtained.

It is not difficult to think of storing these graphic objects through spatial segmentation, such as the k-d tree mentioned earlier or the more complex BSP tree (a more reasonable partitioning method than k-d tree) The data structure. However, considering the particularity of the problem,

we need to point out a prominent drawback of the k-d tree mentioned earlier: since the vast majority of objects are ultimately stored in leaf nodes At most $n/2$nodes in the final constructed tree have no stored objects. As the number of layers increases, we will see more and more space wasted due to this characteristic. And, if there are too many shapes on the entire plane, which can also cause another problem: as the number of layers increases, the performance of each ray intersection query will deteriorate. In addition, due to the representation of object shapes, there are various types of objects, and processing individual objects can also incur significant overhead.

Firstly, for the problem of object handling, we can imagine a simple but effective way: completely wrap it with a circle as a fuzzy representation for searching this object at coarse granularity. The circle can be simply defined as follows: for a polygon consisting of a set of vertices $P_1(x_1, y_1), P_2(x_2, y_2), \dots P_n(x_n, y_n)$, it can be taken The center of gravity $barP(\ barx,\ bary)$ can serve as the center of this approximate circle, and the maximum distance from each vertex to $barP(\ barx,\ bary)$ is the approximate circle The radius. Although this representation method is rough, it fits two purposes very well: 1.The entire geometry is surrounded by the circle 2.This circle only requires two parameters, the center coordinates and radius It can be represented.

Ray l is parameterized based on any two points $(p_x, p_y), (q_x, q_y) : \forall M \in\ l, M = (1-t)P + tQ, (t \geq 0)$. P is defined as the starting point of the ray, and Q is any point on the ray. This means that after specifying this specific ray form, we only need to store the coordinates of two points to represent its all information.

The key to this algorithm lies in the reasonable segmentation of space to achieve a balance between time and space overhead. In the previous partitioning of the k-d tree, there were splitting lines $split_x, split_y$ parallel to the x axis and y axis. However, in this problem scenario, the $split_y$ parallel to the y axis did not play a significant role in the search. Imagine if the ray has already been determined to be aligned with the vertical dividing line $split_y$ during the search process, if they intersect, both the left and right nodes should be searched. But if only the $split - x$ parallel to the x axis is retained, the problem that can also arise is that the algorithm does not know where to start searching from within the structure, and during the recursive process down the tree, it may generate a large number of completely impossible connections with it.

Therefore, it is considered to divide the space in a more flexible way, taking only the dividing line parallel to the y axis, and dividing the entire plane into multiple rectangular slices accordingly. The dividing line $split_y$ can be uniformly selected on the plane, but a more efficient approach is to consider the distribution of objects on the plane before making a decision. For example, by dividing every ten approximate circles into slices, the effectiveness of the slices is ensured, that is, the memory space occupied will not be wasted due to the uneven distribution of geometric objects. A coarse-grained approximate circle representation, so only the center (x, y) and radius r are needed to determine whether the circle is located in a sheet-like region. Each slice is bounded by $split_{y1}$ and $split_{y2}$. If $y_1 \leq x - r < x + r \leq y_2$, then the entire approximate circle is completely within the slice. Otherwise, if $y_1 \leq x + r$ Or if $x - r \leq y_2$ holds, it is also considered that the approximate circle is within the slice. If none of the above conditions are met, the approximate circle is completely non intersecting with the slice.

Based on the parameterized representation of the provided rays, it is evident that the intersection relationship between rays and slices can be obtained based on the relative positions of two

points, P and Q If P is on the left side of Q , then only P needs to be considered. The slice, as well as all slices to the right of this slice, and left, only consider this slice and the slices to the left. In each slice, we consider: based on an idea like line segment tree mentioned, making multiple parallel ones divide the slice into two at the dividing line $spilt_x$ on the x axis. Afterwards, using the same approach for binary search can reduce the time complexity of searching in each slice to logarithmic level.

Finally, we obtained a coarse-grained set containing all possible shapes that intersect with light rays, and then performed fine-grained analysis on these objects using analytic geometry methods. By using the coordinates of polygon vertices, calculate whether the object truly intersects with the light ray, as well as the specific coordinates of the intersection point. By calculating the distance from the intersection point coordinates, select the nearest point as the actual visible point of the light (completely opaque object), or for the intersection point,to sort and render from back to front.

Appendix 3 provides a simplified version of slice search example.

Objectively speaking, people's interest often lies in graphics that are more valuable in three-dimensional space, and these algorithms can be extended to three-dimensional space in various ways without losing its superiority. For example, extending the segmentation slice in two-dimensional space into an infinitely long block area divided by four vertical lines in three-dimensional space, and then taking an approximate sphere to perform segmentation and search within the block area. There are many similar ideas, which will not be further explored here.

7 Conclusion

In this article, the segmentation method is used as the research thread to introduce various geometric data structures, including line segment trees, kd trees, slice search, and their practical applications. Through code implementation and testing, the feasibility and performance of these data structures in practical applications have been verified. By comparing and analyzing the query time consumption of the KD tree decomposition method and the overall recombination method, we have verified the effectiveness of our method in reducing query time costs. By discussing intersection point queries, rectangular window queries, and slice searches, the practical applications and methods of these data structures in computer graphics are demonstrated, providing more efficient practical implementation methods for these data structures. Of course, our research didn't explore further about detailed optimization methods in various fields, such as ray tracing and non-photorealistic rendering, which needs further discussion and research.

References

Bentley, J. L. (1975). Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9), 509-517.

Edelsbrunner, H., Maurer, H. (1981). On the intersection of Orthogonal objects. *Information Processing Letters*, 13(4–5), 177–181.

Fuchs, H., Kedem, Z. M., Naylor, B. F. (1980). On visible surface generation by a priori tree structures. *ACM SIGGRAPH Computer Graphics*, 14(3), 124-133.

Hubo, L., Zhang, Y. (2006). The quantized KD-Tree: efficient ray tracing of compressed point clouds. *ACM SIGGRAPH 2006 Sketches*, 105–113.

Langetepe, E., Zachmann, G. (2006). Geometric data structures for computer graphics.

Li, Z., Wang, T., Deng, Y. (2014). Proceedings of the 18th meeting of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games: Fully parallel KD-Tree construction for real-time ray tracing (p. 159).

Li, Z., Deng, Y., Gu, M. (2017). Path compression KD-trees with multi-layer parallel construction A case study on ray tracing. *I3D ' 17: Proceedings of the 21st ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, 1–8.

Samet, H. (2006). *Foundations of Multidimensional and Metric Data Structures.* Morgan Kaufmann.

Appendix

Listing 1: **Building line segment tree**

```
#define LINE_SEGEMENT_TREE_H
#include <algorithm>
#include <cfloat>
#include <memory>
#include <ranges>
#include <vector>
#endif //LINE_SEGEMENT_TREE_H

class point {
public:
    float x,y;
    point()=default;
    point(const float _x, const float _y) :x(_x),y(_y){}
};

class line {
    public:
    point p,q;
    //notice:p is the left point
    line(const point& _p,const point& _q){
        if(_p.x>_q.x) {p=_q;q=_p;}else{p=_p;q=_q;}
    }
};
class node {
public:
    float split,min,max;
    std::vector<line> lines;
    std::shared_ptr<node> left;
    std::shared_ptr<node> right;
    std::shared_ptr<node> pred;
    node():split(0),min(FLT_MIN),max(FLT_MAX),left(nullptr),right(nullptr),pred(nullptr){}
    node(float _min,float _max):split(0),min(_min),max(_max),left(nullptr),right(nullptr),pred(
        nullptr){}
};
class segment_tree {
    public:
    std::shared_ptr<node> root;
    void build_tree(const std::vector<line>& lines,const std::vector<point>& points,float min,
        float max);
    static auto give_min(const float tar,float& min){min=min<=tar?min:tar;}
    static auto give_max(const float tar,float& max){max=max>=tar?max:tar;}
    explicit segment_tree(const std::vector<line>& lines) {
        //give lines to make this tree
        std::vector<point> points;
        float min=FLT_MAX,max=FLT_MIN;
        for(const auto& line : lines) {
```

```
45              give_min(line.p.x,min);
46              give_max(line.p.x,max);
47              give_min(line.q.x,min);
48              give_max(line.q.x,max);
49              points.push_back(line.p);
50              points.push_back(line.q);
51          }
52          root = std::make_shared<node>(min,max);
53          std::ranges::sort(points,[](const point& a,const point& b)->bool{return a.x<b.x;});
54          build_tree(root,lines,points,min,max);
55      }
56      static bool range_search(const std::shared_ptr<node>& p_node,const line& _line) {
57          if(p_node==nullptr){return true;}
58          return _line.p.x<=p_node->min && _line.q.x>=p_node->max;
59      }
60      static void build_tree(const std::shared_ptr<node>& p_node,const std::vector<line>& lines,
            const std::vector<point>& points,
61                             const float min,const float max) {
62          //[min,max) is the range,p_node is the ptr points to the obj.
63          //vector lines and points is which to be built into the tree.
64          if(points.empty()) return;
65          std::vector<line> newlines_l,newlines_r;
66          std::vector<point> newpoints_l,newpoints_r;
67          p_node->split=points[points.size()/2].x;
68          for(auto& point : points) {
69              if(point.x<p_node->split) newpoints_l.push_back(point);
70              if(point.x>p_node->split) newpoints_r.push_back(point);
71              //this make the point vector for left and right children
72          }
73          for(auto& line : lines) {
74              if(range_search(p_node,line) && !range_search(p_node->pred,line)) {
75                  p_node->lines.push_back(line);
76                  continue;
77              }
78              if(line.p.x<p_node->split) newlines_l.push_back(line);
79                  else newlines_r.push_back(line);
80          }
81          p_node->left=std::make_shared<node>(min,p_node->split);
82          p_node->left->pred=p_node;
83          build_tree(p_node->left,newlines_l,newpoints_l,min,p_node->split);
84          p_node->right=std::make_shared<node>(p_node->split,max);
85          p_node->right->pred=p_node;
86          build_tree(p_node->right,newlines_r,newpoints_r,p_node->split,max);
87      }
88      void get_answer(const float t,std::vector<line>& answer) {
89          answer.clear();
90          if(t<root->min || t>root->max) return;
91          if(root==nullptr) return;
92          answer.insert(answer.end(),root->lines.begin(),root->lines.end());
```

```
93          if(t<=root->split) {get_answer(t,answer,root->left);}
94          else{get_answer(t,answer,root->right);}
95      }
96      void get_answer(const float t,std::vector<line>& answer,const std::shared_ptr<node>& p_node
            ) {
97          if(p_node==nullptr) return;
98          answer.insert(answer.end(),p_node->lines.begin(),p_node->lines.end());
99          if(t<p_node->split) {get_answer(t,answer,p_node->left);}
100         else{get_answer(t,answer,p_node->right);}
101     }
102 };
```

Listing 2: **Building KD tree**

```
1   #ifndef KD_TREE_H
2   #define KD_TREE_H
3   #include <bits/stl_pair.h>
4
5   #endif //KD_TREE_H
6   class KD_Tree {
7   public:
8       struct Node ;
9       Node *ptr=nullptr;
10      struct Node {
11          enum spilt_dim{X=0,Y=1};//split by line x=k or t=k
12          int dim;
13          int split;
14          std::pair<int,int> point;// save only one point(x,y) or(-1,-1)
15          Node* left;
16          Node* right;
17          Node(std::vector<std::pair<int,int>> points_vector, const int pre_dim) {
18              dim = (pre_dim+1)/2;
19              point={-1,-1};
20              if(points_vector.empty()) {
21                  //null situation
22                  left = right = nullptr;
23                  split=0;
24                  point={-1,-1};
25                  return;
26              }
27              if(points_vector.size()==1) {
28                  point = points_vector[0];//save this point
29                  left = right = nullptr;
30                  split=0;
31                  return;
32              }
33              //have more than 2 points
34
35              if(dim==0) {
36                  //split by line x=k
```

```
37                 std::ranges::sort(points_vector,[](const std::pair<int,int>& a,const std::pair<
                       int,int>& b){return a.first<b.first;});
38                 //sort this vector,then get mid
39                 std::vector<std::pair<int,int>> split_points_l;//new vector for next
                       construction
40                 std::vector<std::pair<int,int>> split_points_r;
41                 split=points_vector[points_vector.size()/2].first;
42                 for(int i=0;i<points_vector.size()/2;i++) {
43                     split_points_l.push_back(points_vector[i]);
44                 }
45                 for(int i=(int)points_vector.size()/2;i<points_vector.size();i++) {
46                     split_points_r.push_back(points_vector[i]);
47                 }
48                 point={0,0};
49                 left=new Node(split_points_l,dim);
50                 right=new Node(split_points_r,dim);
51             }else {
52                 //split by line x=k
53                 std::ranges::sort(points_vector,[](const std::pair<int,int>& a,const std::pair<
                       int,int>& b){return a.second<b.second;});
54                 //sort this vector,then get mid
55                 std::vector<std::pair<int,int>> split_points_l;//new vector for next
                       construction
56                 std::vector<std::pair<int,int>> split_points_r;
57                 split=points_vector[points_vector.size()/2].second;
58                 for(int i=0;i<points_vector.size()/2;i++) {
59                     split_points_l.push_back(points_vector[i]);
60                 }
61                 for(int i=static_cast<int>(points_vector.size())/2;i<points_vector.size();i++)
                       {
62                     split_points_r.push_back(points_vector[i]);
63                 }
64                 point={-1,-1};
65                 left=new Node(split_points_l,dim);
66                 right=new Node(split_points_r,dim);
67             }
68         }
69
70     };
71     void search(int,int,int,int,std::vector<std::pair<int,int>>& result,KD_Tree::Node*);
72     explicit KD_Tree(const std::vector<std::pair<int,int>>& points_vector);
73
74     void insert_point(std::pair<int,int> point);
75
76     void search_point(int,int,int,int,std::vector<std::pair<int,int>>& result);
77     static int test() {
78         std::vector<std::pair<int,int>> v={std::make_pair(12,12),std::make_pair(5,6),std::
                make_pair(7,8)},result;
79         KD_Tree mytree(v);
```

```
80          mytree.search_point(2,6,5,8,result);
81          std::cout<<"result: ";
82          for(int i=0;i<result.size();i++) {
83              std::cout<<result[i].first<<" "<<result[i].second<<std::endl;
84          }
85          return 0;
86      }
87  };
88  inline KD_Tree::KD_Tree(const std::vector<std::pair<int, int>> &points_vector) {
89      ptr = new Node(points_vector,1);//the first node is split by x=k
90  }
91
92  inline void KD_Tree::insert_point(std::pair<int,int> point) {
93      //point.emplace_back(point);
94
95  }
96  inline void KD_Tree::search_point(int xa,int xb,int ya,int yb, std::vector<std::pair<int, int>
        > &result) {
97  if(ptr==nullptr) {return ;}
98      KD_Tree::search(xa,xb,ya,yb,result,ptr);
99  }
100
101 inline void KD_Tree::search(int xa,int xb,int ya,int yb, std::vector<std::pair<int, int> > &
        result,KD_Tree::Node* ptr) {
102     if(ptr==nullptr) {return ;}
103     if(ptr->dim==0) {
104         //spilt by line x=k
105         if(ptr->point!=std::make_pair(-1,-1)) {
106             if(ptr->point.first>=xa && ptr->point.first<=xb && ptr->point.second>=ya && ptr->
                    point.second<=yb) {
107                 result.push_back(ptr->point);
108             }
109             return;//finish this search and return
110         }
111         if(xb<=ptr->split) {
112             //search area at left side
113             //so we just continue search left side
114             search(xa,xb,ya,yb,result,ptr->left);
115             return;
116         }
117         if(xa>=ptr->split) {
118             //search right side
119             search(xa,xb,ya,yb,result,ptr->right);
120             return;
121         }
122         search(xa,xb,ya,yb,result,ptr->left);
123         search(xa,xb,ya,yb,result,ptr->right);
124         return;
125     }else {
```

```
126         //spilt by line y=k
127         if(ptr->point!=std::make_pair(-1,-1)) {
128             if(ptr->point.first>=xa && ptr->point.first<=xb && ptr->point.second>=ya && ptr->
                    point.second<=yb) {
129                 result.push_back(ptr->point);
130             }
131             return;
132         }
133         if(yb<=ptr->split) {
134             search(xa,xb,ya,yb,result,ptr->left);
135             return;
136         }
137         if(ya>=ptr->split) {
138             search(xa,xb,ya,yb,result,ptr->right);
139             return;
140         }
141         search(xa,xb,ya,yb,result,ptr->left);
142         search(xa,xb,ya,yb,result,ptr->right);
143         return;
144     }
145 }
```

Listing 3: **Slice search**

```
1  //
2  // Created by aoyeningluosi on 24-11-17.
3  //
4
5  #ifndef GRAPH_H
6  #define GRAPH_H
7  #include <cfloat>
8  #include <numeric>
9  #include <set>
10 #include <valarray>
11 #include <vector>
12 #include <bits/fs_fwd.h>
13
14 #include "line_segment_tree.h"
15
16 #endif //GRAPHLH
17
18 namespace Graph {
19     class point {
20     public:
21         float x,y;
22         point()=default;
23         point(const float _x, const float _y) :x(_x),y(_y){}
24         bool operator==(const point &other) const {return x==other.x && y==other.y;}
25     };
26     struct line_vec {
```

```
27          point p{0,0},q{0,0};
28          float x=0,y=0;
29          //notice:p is the left point,p to q
30          line_vec()=default;
31          line_vec(const point& p, const point& q) :p(p),q(q),x(q.x-p.x),y(q.y-p.y){}
32          line_vec(float p_x,float p_y,float q_x,float q_y) :p(p_x,p_y),q(q_x,q_y),x(q.x-p.x),y(q
                .y-p.y){}
33          line_vec(const line_vec &other)= default;
34          static auto cmp(const line_vec& l1,const line_vec& l2) {
35              //a1b2 - a2b1
36              return l1.x*l2.y-l1.y*l2.x;
37          }
38          static auto cross(const line_vec& a, const line_vec& b) {
39              const line_vec al(a.p,b.q),ar(a.p,b.p),bl(b.p,a.p),br(b.p,a.q);
40              return ((cmp(a,al)>=0 && cmp(a,ar)<=0) || (cmp(a,al)<=0 && cmp(a,ar)>=0))&& ((cmp(b
                    ,bl)>=0 && cmp(b,br)<=0) || (cmp(b,bl)<=0 && cmp(b,br)>=0));
41          }
42          static auto cross_point(const line_vec& a, const line_vec& b) {
43              if(!cross(a,b))
44                  return point(FLT_MIN,FLT_MIN);
45              const float m1=(a.q.y-a.p.y)/(a.q.x-a.p.x),m2=(b.q.y-b.p.y)/(b.q.x-b.p.x);
46              const float b1=a.p.y-m1*a.p.x,b2=b.p.y-m2*b.p.x;
47              return point((b2-b1)/(m1-m2),(m1*b2-m2*b1)/(m1-m2));
48          }
49      };
50      struct light {
51          point init_point,advance_point;
52          light(const point& p,const point& q):init_point(p),advance_point(q){}
53      };
54      class graph {
55          enum class state{non,fin};//when this graph is ok,state==fin
56          point center_point{FLT_MIN,FLT_MIN};
57          float circle_r=0;
58          std::vector<line_vec> lines;
59      public:
60          [[nodiscard]] auto get_center()const{return std::make_pair(center_point,circle_r);}
61          state graph_state= state::non;
62          graph()=default;
63          explicit graph(const std::vector<line_vec>& _lines) : lines(_lines){}
64
65          void assign_center() {
66              center_point=point{0,0};
67              std::accumulate(lines.begin(),lines.end(),center_point,[](const point& center,const
                     line_vec & add)
68                  ->point{return point{center.x+add.p.x,center.y+add.p.y};});
69              center_point.x/=static_cast<float>(lines.size());center_point.y/=static_cast<float
                    >(lines.size());
70              for(auto &l:lines)
71                  circle_r=sqrt(std::pow(l.p.x-center_point.x,2)
```

```
72                      +std::pow(l.p.y-center_point.y,2)) > circle_r?static_cast<float>(sqrt(std::
                            pow(l.p.x-center_point.x,2)
73                          +std::pow(l.p.y-center_point.y,2))):circle_r;
74          }
75
76          void check_finish() {
77              if(lines.size()<2){return;}
78              if((lines.end()-1)->q==lines.begin()->p){graph_state=state::fin;assign_center();}
79          }
80
81          void add_line(const point& p, const point& q) {
82              if(graph_state==state::fin)
83                  return;
84              if(lines.empty())
85                  lines.emplace_back(p,q);
86              else if(const point& last_point=(lines.end()-1)->q; last_point.x==p.x && last_point
                    .y==p.y) {
87                  lines.emplace_back(p,q);
88                  check_finish();
89              }
90          }
91          bool light_if_cross(const light& l) {
92              for(const auto &line:lines) {
93                  auto a1=l.init_point.y-l.advance_point.y
94                  ,b1=l.advance_point.x-l.init_point.x
95                  ,d1=l.init_point.x*(l.init_point.y-l.advance_point.y)+l.init_point.y*(l.
                        advance_point.x-l.init_point.x);
96                  auto a2=line.p.y-line.q.y
97                  ,b2=line.q.x-line.p.x
98                  ,d2=line.p.x*(line.p.y-line.q.y)+line.p.y*(line.q.x-line.p.x);
99                  auto x=(b2*d1-b1*d2)/(a1*b2-a2*b1),y=(a1*d2-a2*d1)/(a1*b2-a2*b1);
100                 auto light_alpha=(x-l.init_point.x)/(l.advance_point.x-l.init_point.x),
                        vec_line_alpha=(x-line.p.x)/(line.q.x-line.p.x);
101                 if(light_alpha>=0 && vec_line_alpha>=0 && vec_line_alpha<=1)
102                     return true;
103             }
104             return false;
105         }
106
107         std::vector<point> light_cross_point(const light& l) {
108             std::vector<point> res;
109             for(const auto &line:lines) {
110                 auto a1=l.init_point.y-l.advance_point.y
111                 ,b1=l.advance_point.x-l.init_point.x
112                 ,d1=l.init_point.x*(l.init_point.y-l.advance_point.y)+l.init_point.y*(l.
                        advance_point.x-l.init_point.x);
113                 auto a2=line.p.y-line.q.y
114                 ,b2=line.q.x-line.p.x
115                 ,d2=line.p.x*(line.p.y-line.q.y)+line.p.y*(line.q.x-line.p.x);
```

```
116                 auto x=(b2*d1-b1*d2)/(a1*b2-a2*b1),y=(a1*d2-a2*d1)/(a1*b2-a2*b1);
117                 auto light_alpha=(x-l.init_point.x)/(l.advance_point.x-l.init_point.x),
                        vec_line_alpha=(x-line.p.x)/(line.q.x-line.p.x);
118                 if(light_alpha>=0 && vec_line_alpha>=0 && vec_line_alpha<=1)
119                     res.emplace_back(x,y);
120             }
121             return res;
122         }
123     };
124 class block_split_tree_node {
125 public:
126     float split{0};
127     std::shared_ptr<block_split_tree_node> parent;
128     std::shared_ptr<block_split_tree_node> left,right;
129 
130     std::vector<graph> graph_on_split;
131     block_split_tree_node()=default;
132     block_split_tree_node(const std::shared_ptr<block_split_tree_node>& _parent,
133         std::vector<graph> graphs) :parent(_parent){
134         if(graphs.empty()){left=right=nullptr;return;}
135         if(graphs.size()==1){
136             graph_on_split.push_back(graphs[0]);
137             left=right=nullptr;split=graphs[0].get_center().first.y;
138             return;
139         }
140         std::ranges::sort(graphs,[](const graph& a,const graph& b)
141             {return a.get_center().first.y<b.get_center().first.y;});
142         split=graphs[graphs.size()/2].get_center().first.y;
143         graphs.erase(graphs.begin()+graphs.size()/2);
144         std::vector<graph> left_pass,right_pass;
145         for(auto& p:graphs) {
146             if(p.get_center().first.y-p.get_center().second<=split && p.get_center().first.y+p.
                    get_center().second>=split) {
147                 graph_on_split.push_back(p);
148                 continue;
149             }
150             if(p.get_center().first.y+p.get_center().second<=split) {
151                 left_pass.push_back(p);
152                 continue;
153             }
154             right_pass.push_back(p);
155         }
156         left=std::make_shared<block_split_tree_node>(std::shared_ptr<block_split_tree_node>(
                this),left_pass);
157         right=std::make_shared<block_split_tree_node>(std::shared_ptr<block_split_tree_node>(
                this),right_pass);
158     }
159 };
160 class block_split_tree {
```

```
161     public:
162     std::shared_ptr<block_split_tree_node> root;
163     explicit block_split_tree(const std::vector<graph> &graphs) {
164         root=std::make_shared<block_split_tree_node>(nullptr,graphs);
165     }
166 };
167     class ground_block {
168         float left=0,right=0;
169         std::vector<graph> graphs;
170         std::shared_ptr<block_split_tree> ptr;
171     public:
172         std::shared_ptr<block_split_tree> get_ptr(){return ptr;}
173         ground_block(const float& _left,const float& _right):left(_left),right(_right){}
174         [[nodiscard]] bool contain(const std::pair<point,float>& p) const {
175             return !(p.first.x+p.second<left || p.first.x-p.second>right);//contain is true
176         }
177         void attend(const graph& g){graphs.push_back(g);}
178         void cut() {
179             ptr=std::make_shared<block_split_tree>(graphs);
180         }
181         std::vector<graph> light_find(light _light,const std::shared_ptr<block_split_tree>&
                pointer) {
182             //first light_find
183             std::vector<graph> result;
184             if(pointer==nullptr) return std::vector<graph>{0};
185             auto& target_vec=pointer->root->graph_on_split;
186             for(auto& g:target_vec) {
187                 //in this target vector ,find those graph who can cross with light
188                 if(g.light_if_cross(_light))
189                     result.push_back(g);
190             }
191             light_find(_light,pointer->root->left,result);
192             light_find(_light,pointer->root->right,result);
193             return result;
194         }
195         void light_find(light _light,const std::shared_ptr<block_split_tree_node>& pointer,std
                ::vector<graph>& result) {
196             if(pointer==nullptr) return;
197             for(auto& target_vec=pointer->graph_on_split; auto& g:target_vec) {
198                 //in this target vector ,find those graph who can cross with light
199                 if(g.light_if_cross(_light))
200                     result.push_back(g);
201             }
202             light_find(_light,pointer->left,result);
203             light_find(_light,pointer->right,result);
204         }
205     };
206     class ground {
207         int graph_number=0,block_number=0;
```

```
208         float length=0;
209         point left_lower_corner{FLT_MAX,FLT_MAX},right_upper_corner{FLT_MIN,FLT_MIN};
210         std::vector<float> split;
211         std::vector<ground_block> blocks;
212     public:
213         ground()=default;
214         std::vector<std::pair<point,float>> point_pairs;
215         void init_ground(const std::vector<graph>& graph_set) {
216             std::ranges::for_each(graph_set,[this](const graph& g){point_pairs.push_back(g.
                    get_center());++graph_number;});
217             const auto x_minmax=std::minmax(point_pairs.begin(),point_pairs.end(),
218                 [](const auto& a,const auto&b){return a.first.x<b.first.x;});
219             const auto y_minmax=std::minmax(point_pairs.begin(),point_pairs.end(),
220                 [](const auto& a,const auto&b){return a.first.y<b.first.y;});
221             left_lower_corner=point{x_minmax.first->first.x,y_minmax.first->first.y};
222             right_upper_corner=point{x_minmax.second->first.x,y_minmax.second->first.y};
223             length=right_upper_corner.x-left_lower_corner.x;
224             block_number=graph_number/5;
225             const float step=length/static_cast<float>(block_number);
226             for(int i=0;i<=block_number;i++)
227                 split.push_back(left_lower_corner.x+step*static_cast<float>(i));
228             for(int i=0;i<block_number;i++)blocks.emplace_back(split[i],split[i+1]);
229             for(const auto& g:graph_set) {
230                 const auto target=std::ranges::find_if(blocks,
231                     [g](const auto& blk){return blk.contain(g.get_center());});
232                 target->attend(g);
233             }
234             std::ranges::for_each(blocks,[](const auto&block){block.cut();});
235         }
236         void light_find(const light& target_light,std::vector<graph>& result) {
237             auto target=std::find_if(blocks.begin(),blocks.end(),
238                 [target_light](const ground_block& _b){return _b.contain(std::make_pair(
                        target_light.init_point,0));});
239             std::for_each(target,blocks.end(),[&target_light,&result](ground_block& b)
240                 {auto arr=b.light_find(target_light,b.get_ptr());std::ranges::copy(arr,std::
                        back_inserter(result));});
241         }
242     };
243 }
```

AI Tools Statement

In the writing of this research report, GPT-4O was used to assist in the collection and organization of literature information. In addition, GitHub Copilot was used in code testing to complete the code and generate error improvement suggestions.