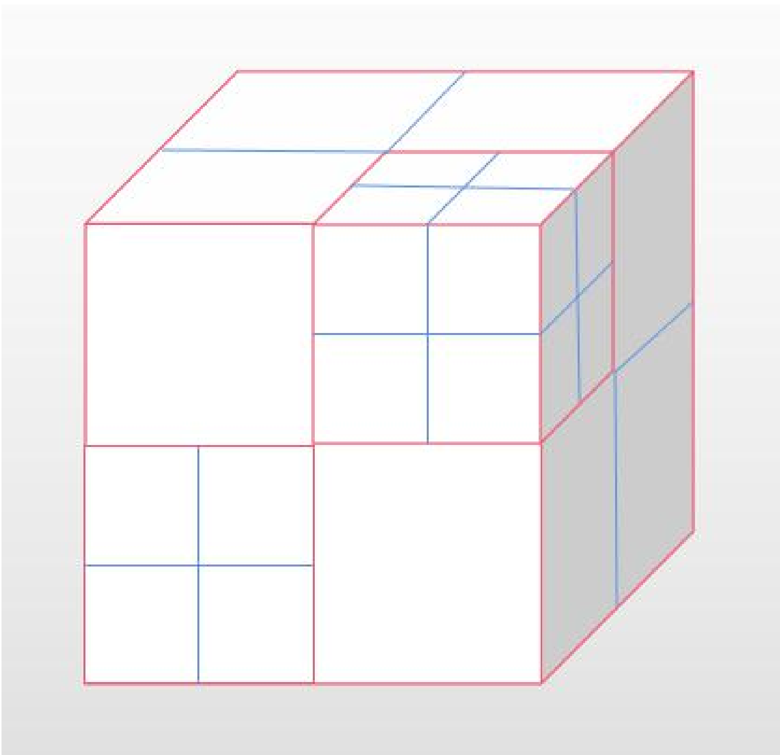
Replication of Octree structure and related algorithm in PCL final report

王志越

1. Octree Structure

Octree is a structure for spatial division in 3 dimensions and each node represents a cubic space. Like Quadtree, Octree is to be recursively subdivided into eight octants with same size. For each node, it is a cubic box that encapsulates many the points in the point cloud file and for root node, it encapsulate all the points. There are two cases of leaf node. One is that if there is no point in a node cubic box, then it is meaningless to divide the node and it will be the leaf node. The other case is that if the node depth reaches an assigned tree depth, it will not be divided and it will be a leaf node.



2. K Nearest Neighbor Search Algorithm

For K nearest neighbor search algorithm, it describes that find K points in point cloud that are nearest to a given point. In PCL, it maintains a K size array. For each point in nodes, it calculates the distance between it and the given points, and then resize the array into K size. In order to accelerate the calculation speed, the calculation sequence will consider the distance between the node and the given points and after the calculation of points in one node, it will resize the array and record the longest distance in this K size array. Then when check the other node, it will check whether the distance between the center of the node and the given point is less than the sum of the longest distance in K size array and the half of the diagonal length for this node. If it is true, then it will calculate all the points in this node and resize to K size. Otherwise, it will not calculate.

Pseudo-code:

Procedure KNNSearch(point *p*, int *K*, node *n*, int *depth*, int *smallest_square_distance*, Array *candidates*):

    square_diagonal_length ← getDiagonalLength(*depth*)^2
    Array *search_nodes*
    For each *node* **in** *n.child_nodes*:
        node_point_distance ← distance(*node.center*, *p*)^2
        Add *node* into *search_nodes*
    Sort *search_nodes* according to *node_point_distance*
    For each *node* **in** *search_nodes*:
        If *node_point_distance* < *smallest_square_distance* + *square_diagonal_distance*/4
        +square_root(*smallest_square_distance* * *square_diagonal_distance*) **then**:
            If *node* is branch_node **then**:
                smallest_square_distance ←
                KNNSearch(p, K, node, depth+1, smallest_square_distance, candidates)

Else:
For each *point* **in** *node*:
square_point_distance ← distance(*point*, *p*)^2
If *point_distance* < *smallest_square_distance* **then**:
Add *point* into *candidates*
Sort *candidates* according to *square_point_distance*
Resize candidates to *K*
smallest_square_distance ← largest *square_point_distance* in *candidates*
Else:
break
Return *smallest_square_distance*

3. Radius Search Algorithm

The algorithm describes to find all the points that are in the ball with given radius and given point as center. This algorithm is similar to KNN search algorithm but with constant radius as distance threshold and candidates size is unlimited.

Pseudo-code:

Procedure RadiusSearch(point *p*, int *radiusSquared*, node *n*, int *depth*, Array *candidates*)
square_diagonal_length ← getDiagonalLength(*depth*)^2
Array *search_nodes*
For each *node* **in** *n.child_nodes*:
node_point_distance ← distance(*node.center*, *p*)^2
Add *node* into *search_nodes*
Sort *search_nodes* according to *node_point_distance*
For each *node* **in** *search_nodes*:
If *node_point_distance* < *radiusSquared* + *square_diagonal_distance*/4
+square_root(*radiusSquared* * *square_diagonal_distance*) **then**:
If *node* is branch_node **then**:
RadiusSearch(*p*, *radiusSquared*, *node*, *depth*+1, *candidates*)
Else:
For each point **in** node:
square_point_distance ← distance(*point*, *p*)^2
If *point_distance* < *smallest_square_distance* **then**:
Add *point* into *candidates*
Else:
break

4. Point Cloud Compression

For limited channel transferring, enormous data set for point cloud will cause the big influence on transferring efficacy. Point cloud compression is able to compress the structure of the octree to reduce the file size for transferring the point cloud information and then improve the efficiency.

For point cloud compression, there are two parts information. One is for tree structure. For each node, it corresponds to one 8 bits number. And each bit represents the category of its child nodes. 1 means the child node exists and 0 means the child node does not exist. Recursively traverse the tree and store each 8 bits number into array. Then the array contains the structure of the octree. The other is for point information. During traversing the octree, when reach the leaf node, an array will store the point color and position in the node. Then the array contains the structure of the octree. For decompression, invert the procedure will get the octree.

Pseudo-code:

Procedure Compression(node *n*, Array *binary_tree*, Array *leaf_container*):
    node_bit ← 0
    For each *node* **in** *n.child_nodes*:
        If *node* exists **then**:
            node_bit ← *node_bit* shift left 1 unit + 1
        Else:
            node_bit ← *node_bit* shift left 1 unit
    Add *node_bit* into *binary_tree*
    For each *node* **in** *n.child_nodes*:
        If *node* exists **then**:
            If *node* is branch node **then**:
                Compression(*node*, *binary_tree*, *leaf_container*)
            Else:
                Add position and color information in *node.points* into *leaf_container*

Procedure Decompression(node *n*, int *remain_depth*, Array *binary_tree*, Array *leaf_container*):

node_bit ← pop first element in *binary_tree*

For each *i* from 1 to 8:

If *i*th bit of *node_bit* is 1 **then**:

If *remain_depth* > 1 **then**:

branch_node ← *n*.createBreanchNode(*i*)

Decompression(*branch_node*, *remain_depth*-1, *binary_tree*, *leaf_container*)

Else:

leaf_node ← *n*.createLeafNode(*i*)

points ← pop first element in *leaf_container*

Add *points* into *leaf_node*