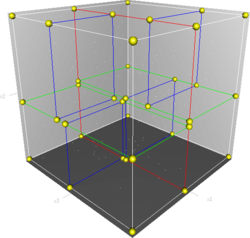
KD-Tree background (<https://en.wikipedia.org/wiki/K-d_tree>)

The *k*-d tree is a [binary tree](https://en.wikipedia.org/wiki/Binary_tree) in which every node is a k-dimensional point. Every non-leaf node can be thought of as implicitly generating a splitting [hyperplane](https://en.wikipedia.org/wiki/Hyperplane) that divides the space into two parts, known as [half-spaces](https://en.wikipedia.org/wiki/Half-space_(geometry)). Points to the left of this hyperplane are represented by the left subtree of that node and points right of the hyperplane are represented by the right subtree. The hyperplane direction is chosen in the following way: every node in the tree is associated with one of the k-dimensions, with the hyperplane perpendicular to that dimension's axis. So, for example, if for a particular split the "x" axis is chosen, all points in the subtree with a smaller "x" value than the node will appear in the left subtree and all points with larger "x" value will be in the right subtree. In such a case, the hyperplane would be set by the x-value of the point, and its [normal](https://en.wikipedia.org/wiki/Surface_normal) would be the unit x-axis.

[](https://en.wikipedia.org/wiki/File:3dtree.png)

A 3-dimensional k-d tree. The first split (red) cuts the root cell (white) into two subcells, each of which is then split (green) into two subcells. Finally, each of those four is split (blue) into two subcells. Since there is no more splitting, the final eight are called leaf cells.

**ClosestPointQuery**

Class description:

template<

class TYPE, ; data type of the each of the axis.

int DIM ; dimension of the space.

>class ClosestPointQuery

In class data types:

1, Point: Point type is defined by std::array, which can be used to describe n dimensional points;

2, MyIterator / MyConstIterator: iterators used to traverse the points array;

3, Mesh: an array of points in the image;

Class attributes:

1, kdhead: root point to the kdtree;

2, nodepool: continuous node arrays for the kd tree. The kd tree once knows the mesh size, it will allocate all the memory needed one time. This can avoid the need to allocate each kdnodes one by one, which will be very slow;

3, currentFree: an index to indicate the current free kdnode in the nodepool;

4, invalidPoint: this is only used to return from a query when there is no closest point found within a given max distance range.

**Operations of KD tree (implemented 2 creations, one search, no add/remove is supported at the moment.):**

1. Construction 1:

ClosestPointQuery::BuildKD(mesh, depth)

This algorithm is the one described in the Wikipedia page (<https://en.wikipedia.org/wiki/K-d_tree>). By using a stack, the recursive call is eliminated in case the call stack is too small. The algorithm   
takes O (nlog2n) time, and O (logn) space (the stack) to avoid recursive call.

1. Select axis depending on current tree depth. Select the depth % DIM axis;
2. Sort the points in current sub tree by their coordinates of the current axis;
3. Choose the middle point as the sub tree root, create a kdnode;
4. Increase depth by 1, repeat from first step for left points and right points.
5. Construction 2:

ClosestPointQuery::BuildKDFast(mesh, depth)

This algorithm is from “*Building a Balanced k-d Tree in O(kn log n) Time*” of Journal of Computer Graphics Techniques Vol. 4, No. 1, 2015. A detailed example is in the above mentioned.

Although it is supposed to be faster than the previous implementation, the actual test proves that due to the large chunks of memory copy (maybe there is a way to avoid memory copy), this algorithm is slightly slower than the previous implementation.

1. In this solution, an auxiliary array of vectors is used to record current sequence of the points, based on different super keys. For an 3d kd-tree, the array contains 3 vectors:  
   references [0], sorted by comparing values of super keys composed in xyz order;  
   references [1], sorted by comparing values of super keys composed in yzx order;  
   references [2], sorted by comparing values of super keys composed in zxy order;
2. The division is firstly done in x axis (let’s name it super axis), then y axis, then z axis, and then start over from x again; simply select the element in the middle of the current sub tree, this is an index number from the super axis which could be used to position the point with a middle x value in the space; A kdnode is allocated for this point.
3. Then reorder the y axis by traversing references [1] (index 0 is for x, 1 for y and 2 for z), pick each index number, compare the point with the middle point selected in the previous step. Compare the two, smaller point index will be moved to the lower half of the array, larger point to the upper half. In this situation the super key is still with xyz order; Same step applies for z axis, reorder references[2] array; note that the x axis array is still ordered so no need to reorder it at this stage;
4. Now the three arrays are divided into two parts, with half of the length of the original. Repeat from step 2 with two half trees and an increased depth, which means this time the division will be based on y axis (super axis change to y now), hence the comparison is based on super key yzx now.

The following example is from the above mentioned thesis:

1, the three arrays are ordered separately by keys composed of xyz, yzx, zxy combinations; the first axis in the combination is called a super axis; it is also the one used to divide the space.

2, the first middle division is from x axis (first super axis), this takes 5 out of the array and get the point (7,2,6), just the middle of x axis;

3, merge sort y axis array, start from the first element, namely 13; compare point in index 13, (2,1,3) to the middle point (7,2,6), key order xyz, it is smaller than the middle, so index 13 is in the lower half of the array (starts from 0);

4, compare the point of second index 4 to the middle, now it’s greater than the middle, so 4 is in the upper half of the array (starts from n/2 + 1, in this case it is 8);

5, if the point picked is equal to the middle, simply skip it.

6, continue this step until y axis is divided in to two, than do the same with z axis. Keep in mind that the lower and upper halves are filled one by one. This is why we need a copy of the original array.

7, change super axis from x to y, go to 2; until the tree is no long dividable.



1. Nearest neighbor search

ClosestPointQuery::operator()(point, maxDist)

Binary search tree, with the complexity of O(logn) time, and O(logn) space.

1. Binary search from root, until one of the leaves is reached.
2. Store all nodes along with the search path in an priority queue, comparing the distances of each of the point to the point being searched;
3. Return the closest distance point from the priority.
4. If no point is found within maxDist range, return an invalid point;
5. Add/Remove elements  
   Not implemented;
6. Balancing  
   Not implemented;