A database approach to very large LiDAR data management

LIU Hua *[a], HUANG Zhengdong [a], ZHAN Qingming[a] , LIN Peng[a]

[a]School of Urban Design / Research Center for Digital City, Wuhan University
Donghu Nanlu 8, Wuhan, China 430072

KEYWORDS: LiDAR data, Octree, Quadtree, Local KD tree, Display precision

ABSTRACT:

This paper presents an approach to realizing LiDAR data management in DBMS. We use octree to partition data space, and build a local KD tree at each octree's node. In data organization, we take precise control on the size of the KD tree's nodes. To effectively visualize point cloud data, the display precision is defined. The basic concept is to judge whether a node is displayed or not, by computing the size of a node's data range after projection. We also discusses the method of screen-buffer and makes KD traversing from front to back, which can reduce the number of points for displaying and accelerate display speed. This method is particularly suitable for very dense data or data far away from the viewpoint.

1. INTRODUCTION

Light Detection and Ranging (LiDAR) is a data acquisition technique based on laser technology, which has been used widely in recent years. Advantages of using LiDAR include the following: LiDAR allows rapid generation of a large-scale DTM (digital terrain model); LiDAR is daylight independent, is relatively weather independent, and is extremely precise. In addition, because LiDAR operates at much shorter wavelengths, it has higher accuracy and resolution than microwave radar [1]. Depending on specific conditions, the level precision of LiDAR system ranges less than 1m, and height precision ranges between 15cm and 20cm [2]. After being processed, LiDAR data may generate highly accurate digital Elevation model (DEM), contour map and orthophoto map.

Because of the huge amount of data and also the limitation of computer hardware, there has been no effective approach to organize and manage the LiDAR data. The visualization of the data has also not been satisfactorily resolved. There are some algorithms for the display of large volumes of data, which usually are based on the LOD of triangular net[3]. Common method to solve the problem is writing the data into a compact file, by means of which to speed up frustum clipping and processing.

However, for discrete LiDAR points, traditional methods of LOD and hidden surface removal [4] are no longer applicable. Furthermore, it only considers view frustum clipping, but pays no attention to the relationship between the data points, so LOD and hidden surface removal are not appropriate. As a result, it surely can't gain a satisfactory result. In this paper, we use database to organize data and establish the relationship between them, and build local KD tree for data index. The algorithms of display precision control are presented.

2. KD TREE AND IMPROVEMENT

In order to process and manage large volumes of LiDAR data, an efficient data structure is very important. The KD tree may provide suitable solution.

KD tree is a binary tree in a K-dimensional space. In the traditional binary search tree, the data classification standard is a key word which is usually a number which have a certain attribute, such as the coordinate on the X axis. For the K-dimensional data, only one key word is not enough to effectively partition the multidimensional data. KD tree makes the key word alterable, which defines key word based on each node and the coordinate on each axis will play the role of key word in turn[5]. A usually mode is to make the (N%K+1)th-dimensional coordinate value as key word if the node is in the N-level, that is:

$$node.split = node.lefel\%K + 1$$

There are two ways to build a KD tree: one is direct insertion, but as KD tree hasn't the ability to balance, the form of tree totally depends on the order of input; the other way is to calculate the form of KD tree in good balance and get the order to establish the balance tree. In this way, the tree would get balanced at the cost of pre-calculation.

Compared with the quadtree and octree, the KD tree has several advantages, such as balance, constructing based on data partition, and no empty nodes. The disadvantage is that, for some uniform distribution of data, the depth of the tree will be deeper. That's because each node of KD tree only has two sub-trees, while quadtree and octree have four and eight respectively.
In traditional KD tree, each node has only one data point, which means node must be split into two sub-nodes when it has more than one point. This method is very wasteful both for query and representation. So, a better standard which determines whether the node should be split or not is necessary.

There are two basic methods to judge whether the node will split or not: the method based on the number of points, and based on the space.

The method based on the number of points is given a fixed number N, which is defined as the largest number of points in each node. If the number of point in a particular node is less than or equal to N, the node needn't to be split, otherwise

continue to be split down. So, we should add a variable for each leaf node to denote the number of points in this node, while inserting a new data, examine the variable whether it's greater than N, if it is, split the node.

The method based on the space is given a fixed spatial scales R, such as the volume or length of smallest bounding box enclosing all data, or the radius of smallest sphere containing all data in the node. When the node's spatial scales are smaller than R, the node needn't to be split. This approach requires adding some parameters of smallest bounding box or sphere. And after each inserting, KD tree need to modify all parameters from the root to the node where the inserted data stayed.

The N and R defined as a standard above should be estimated from original data in advance. The above two methods show some limitations if data are not evenly distributed. With the number-based standard, the node involves a very large region in sparse areas, and splits too much in dense areas; while in the spaced-based standard, the node contains only one point in sparse area, and involves large number of points in dense areas. It takes advantage of finding nearest point while based on the number of points, and takes advantage of simple operation in display while based on the space. For our data management purpose, we choose the space-based standard, but some rules are added:

1). Based on the needs of displaying and processing, set a normal space scale rMin for splitting. If the node's spatial scale is less than rMin, the node needn't be split;

2). Considering the operational environment and complexity, set the maximum number of points nMax. If the node's number of points is larger than nMax, the node must be split into two nodes until the number is less then nMax;

3). While the node's spatial scale is no less than rMin and the number of points is no larger than nMax, we could split the node as less as possible by taking in some error. Assume the node is P. If it is split into two sub-nodes----A and B, calculate the 'percentage of precision loss' f(P). Give a variable u, if f(P)<u, point P needn't to be split into two.

PL is the plane which minimizes the squared distances to the points of P. g(P) means the squared minimum distances, V1(P) and V2 represent the volume of smallest bounding box of P and the cube of rMin respectively. k is a specified coefficient defined by the programmer. The formula can be defined as f(P)= g(P)/[g(A)+g(B)], and u=k*V1(P)/V2 .

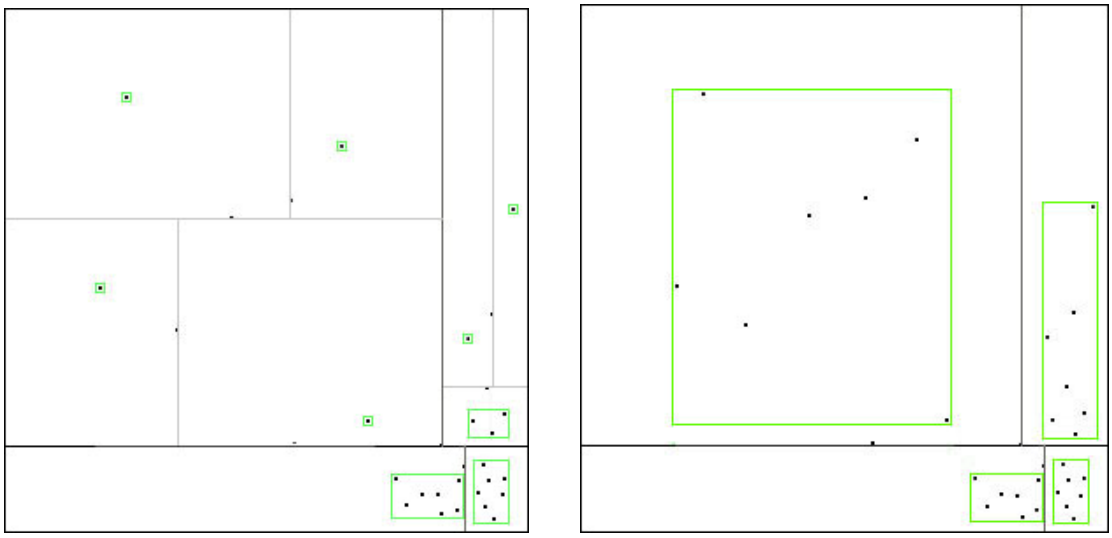


Figure 1. The left figure shows a KD partition based on the space, where the node contains only one point is spares areas, and can involves a large number of points in dense areas. The other shows a KD partition based on the number of 7, this method can make the region very large or very small depending on the dense.

The structure of KD tree's node and the tree is:

```
template <typename Xtype>
class KDNode
{
public:
        int axis ;
        Xtype x[SD];
        Xtype (*xList)[SD];                     // Definition of data in nodes
        int dataNum;                                    // Definition of the number of data in each node
        void insertData(Xtype x[][3],int n);            //While the node needn't to split, put all the data into node
        Xtype bound_cube_min[3];                // The bound of smallest bounding box
        Xtype bound_cube_max[3];
        KDNode(Xtype* x0, int axis0);
        KDNODE*         Insert(Xtype* x);
```

```
        KDNODE*         FindParent(Xtype* x0);      // Search node, return the parent node
        KDNODE* Parent ;
        KDNODE* Left ;
        KDNODE* Right ;
};
template <typename Xtype>
class KDTree
{
public:
        KDNODE*   Root ;
        KDTree();
        bool                    add(Xtype* x);
        void                    insertToKD(Xtype* data);        // The function of add the non-leaf node
        KDNODE*                 find_nearest(Xtype* x0);        // Find the nearest node
        Xtype                   x_min[SD], x_max[SD];           // Set bounding box
};
```

3. LOCAL KD TREE

Building a KD tree for a whole set of LiDAR data in traditional way is less possible. One reason is the tremendous volume of data. Another reason is that it's required to sort the whole data based on the coordinates in the three axial direction before building KD tree, whose complexity is O(N*ln(N)). As large data set consumes too much time, data partition is necessary. In this way, we divide the data into small pieces and build local KD tree for each piece of data, to help the organization and management of data.

3.1 Using octree to confirm the bound of local KD tree

In order to avoid excessive search, we adopt the octree to partition space, which makes point's location known in advance. It saves time of comparing with node's key word during searching, and facilitates judging whether it's in the view frustum.

The standard for splitting octree's node is defined as: if the node's number of points is less than a pre-defined number, it needn't to be split. The pre-defined number is determined by the capacity of KD tree. That can make the scale of KD tree uniform, which is convenient for post-processing. In order to get the balance between the octree's depth and KD tree's saturation, the number should be defined as 2^n.

The definition of a local KD tree includes two steps. Firstly, the bound and density of the data should be determined to establish octree. Then, each node is split into eight sub-nodes through the center position of the bounding box until the node's number of point is less than 2^n. The data of each leaf node in octree make up of a local KD tree. In the octree, the nodes only store their information about bounding box. Each leaf node is given an index value for the convenience of research.

3.2 View frustum clipping

According to the position of viewpoint and the frustum, we compute all octree's nodes in view frustum. Read the data in each node and build local KD trees.

The program that computes which nodes are in the view frustum can be like this: seek from the root, if the node's bounding box and the view frustum have an intersection, seek for node's two children, otherwise return.

We note that in each vision, we need to identify the called KD trees, and then read data from database to build KD trees. The process is very cumbersome. In viewing frustum clipping, the general view transform only changes the edge of the scene. For local KD trees, a view transform is likely to retain most of KD trees, only a small part of the trees will be changed into or out of the view frustum. As the Figure 2 shows, when the view frustum moved right, only several green regions enter the frustum and several red regions quit. So, while the scene is redrawn, we only need to compute these special trees, destruct the red trees and construct the green trees. That will avoid destructing and reconstructing many trees.

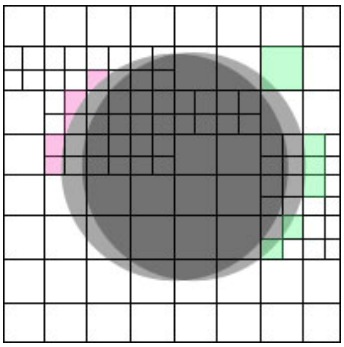


Figure 2. A view transform is likely to retain most of KD trees. In this picture, only the green regions enter the frustum and the red regions quit.

3.3 Traverse the octree front to back

For binary tree, a node has only two sub-nodes, the front to back traversing order can be directly derived by comparing the view position with the node's key word. But for octree which has eight children, the order from front to back is not so simple. Consider the cube shown in the picture, it represents a node of octree and it has eight children. If the viewpoint is in node 1, it's certain that we will traverse node 1 first and node 7 at the end, because node 1 is in front of any other nodes and node 7 is behind any others.

Notice that, three nodes, node 2, 4 and 5, have a public plane with node 1 and only node 1 is in front of them. Moreover, they are separated by lines AB, CD and EF, any one of which isn't in front of any other. There are three other nodes, i.e., node 3, 6 and 8, that have a public line with node 1, they are in the back

of node 1, 2, 4 and 5, and are separated by the three lines too. Thus, the traversing sequence should be:

[1]——[2, 4, 5]——[3, 6, 8]——[7]

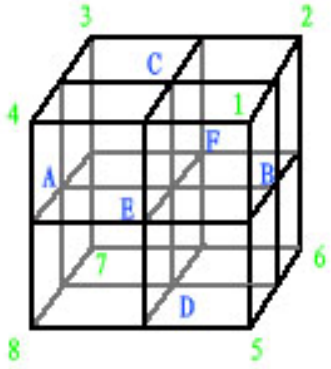


Figure 3. A cube which has a root and eight sub-nodes.

Add a variable i for marking the traversing sequence and the initial value is 0. Test the three-dimensional coordinates respectively, if a coordinate of viewpoint is not in the node's bound, the node's i will add one. When traversing the eight sub-nodes, variable i gives the sequence.

The pseudo-code is below:

```
int node.calculateFrontOrBack()
{     int i=0;
for(int j=0;j<3;j++)
      if(!(eye[i]∈node.bound))
                  i++;
return i;
}
tree.walkNode()
{…
      for(int j=0;j<4;j++)
                  for(int k=0;k<8;k++)
                              if(subNode[i].i==j)

      tree.walkNode(subNode[i]);
…}
```

For the issue of managing large volume of LiDAR data using local KD trees, we choose octree to determine the bound of local KD trees. Data are put into RDBMS in specific order in pre-process, the KD trees are computed during run time, and local KD trees can be quickly built by reading data from the database. We also make some improvements on other details. These methods are suitable for large amount of data, and can achieve a satisfactory processing speed.

4. DISPLAY PRECISION

In displaying LiDAR data using local KD trees, there may be some trees or nodes which are far from the viewpoint. Drawing these trees or their nodes has less effect to the observer. Consider a node far from the viewpoint that the region of data's projection on screen is only one pixel, despite how many points the node contains. So only one point can be drawn instead of drawing hundreds, even thousands of points from the node. Based on this principle, we use the concept of screen precision to control display based on local KD tree. That is, if a small area on the screen contains many points, we only draw a representational point instead of drawing all of them. The small area usually to be defined as a pixel and the representational point can be the point that splits the node into two.

Based on this idea, it's needed to get two-dimension coordinates on screen from any point after projection. The region's size consisting of the points' two-dimension coordinates on screen will determine whether its sub-nodes will been shown or not.

Calculating screen coordinates is very simple in orthographic projection. In perspective projection, the formula for calculating is shown below.

Let eye position to be eye[3], view[3] means the direction of view, and up[3] indicates which direction is up. Calculating the vector multiplication exterior, make a coordinate system, eye[3] as the origin, exterior as X axis, up as Y axis, view as Z axis. For any data P, its projection is indicated in Figure 4. So the point P' will be the mapped point of P on screen. The lengths of E'P' and A'P' will be the X and Y coordinates of P'.

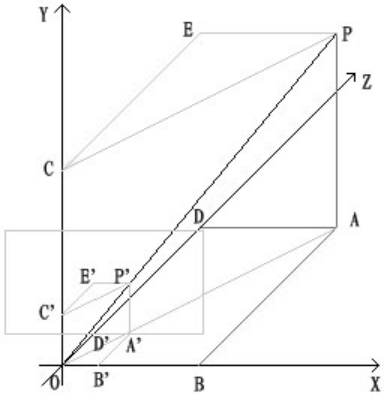


Figure 4. The eye position is point O, view is Z-axis, up is Y-axis, and exterior is X-axis.

OD' is the distance from P' to the XY plane, its length can be count by the frustum and the size of scene. Because of

$$\frac{P'E'}{OD'}=\frac{A'D'}{OD'}=\frac{AD}{OD}=\frac{PE}{AB}$$
$$\frac{P'A'}{PA}=\frac{OA'}{OA}=\frac{OD'}{OD}=\frac{OD'}{AB}$$

, we can get:

$$P'E'=\frac{PE}{AB}*OD' \quad , \quad P'E'=\frac{PE}{AB}*OD' \quad .$$

PA, PE and AB respectively are the distance from P to plane XZ, YZ and XY. So we could get the length of E'P' and A'P'.

In order to examine a node's precision, we need to calculate all points in it, he process is time-consuming and very expensive.

A simple approach is using the vertexes of convex hull to replace the whole data. Assume the data set in a node is S, and build a convex hull CH(S), in which the set of vertexes is P(S). For any point in P(S), compute its screen coordinates. Let the set of two-dimension point is P'(s'), compute its convex hull CH(P'(S)). If CH(P'(S)) is small enough, we will not show the node's sub-node.

But in fact, the complexity of computing convex hull is O(n*ln(n)), it means that the computing is also time consuming. Moreover, the convex hull would also have errors. As shown in Figure5, if the green points are vertexes of the convex hull, then the CH(P'(S)) is the region encircled by the green lines. But we can only guarantee the region that contains green points is actually the area we need. The gray area may have no any point to be projected to, which indicates the inaccurate area.
To simplify calculation, we use smallest bounding box along the axis to replace convex hull, which makes the computation more imprecise (Figure 5). The area using bounding box is the region encircled by the red lines. Compared with convex hull, it generates four more squares (indicated as pink color). But this method may discard the cumbersome computing for convex hull, and effectively speed up the processing speed.

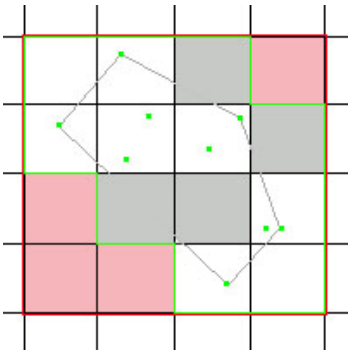


Figure 5. The smallest bounding box along the axis would take the place of convex hull.

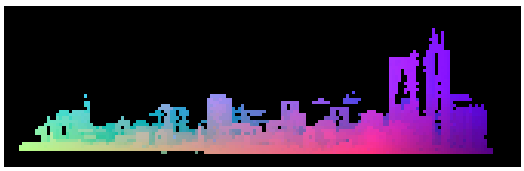
The above approach can reduce the points on screen, and accelerate displaying speed. At the same time, some more improvement will make the effect more satisfactory.

1) The KD tree is divided into three dimensions, after projecting to two dimensional planes, there will be many overlapping points projected to the same pixel. As a result, the larger the data set is, the more there are overlapped points. To enhance the efficiency further, we propose a BOOL screen buffer, which records whether the pixel on the screen has been correctly drawn. When we want to draw a point, examine the corresponding pixel on screen, if it has been drawn, we will not draw again.
2) Based on the above improvements, it's possible that we draw a point that is far from us first, and then when we want to draw a closer point, we find the pixel has been drawn, so we give up drawing. That is, the point isn't drawn in a 'correct' way. In order to consider the depth of point, we can use the depth buffer, but it spends too much computing resources. A simple approach is to traverse the KD tree in a "front to back" way. The order to traverse a node is: examine which sub-node contains the displayed point, traverse it, then draw the node's split data, and then traverse other nodes.
3) Classical KD tree defines the split axis of X, Y and Z in turn, which ensures the fairness and makes it convenient to deal with. But in this way, the different density on three dimensions may make the node's region anomalous. The best definition for split axis is the dimension of largest spatial extent [6], which can make node's region nearly the same length in three dimensions and ensure a stable efficiency.
4). After discussion on screen buffer, the standard of node's split can be added as: if pixels in the region after node's projection have all been drawn, the process ends. This standard will simplify point display in tremendous volumes of data.

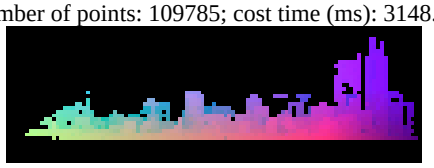
5. EXPERIMENTAL RESULTS

In a computer with Pentium4 2.4GHz CPU, 1G memory, NVIDIA GeForce FX 5200 graphics card, we run the program in three ways. These respectively include the original method, the method using 1.5 pixels display precision, the method of using 1.5 pixels display precision and screen buffer. We utilize a data set of 3,200,079 points, with a disk size of 87.5M. Four pictures for experiment are shown below:

Number of points: 237953; cost time (ms):6152.31



Number of points: 109785; cost time (ms): 3148.24



Number of points: 49468; cost time (ms): 1588.09

Number of points: 20202; cost time (ms): 757.467

Figure 6. Experiments results

As shown above, with the decrease of model's percentage on screen, the number of points becomes smaller and the display time becomes shorter. The number of points and display time maintain a linear relationship. This is consistent with our original intention of using the points that will be shown and discard the points that need not be shown.

With the increase of the length from viewpoint to model, the number and time become smaller and smaller. Compared with the way without screen buffer, the screen buffer reduces the number in screen but increases the display time, that's because in this way it adds a judge of whether the pixel has been shown or not for each 'big' node. This drawback can be improved by the fourth improvement.

Noted that the number of points and the display time maintain a linear relationship, we can get the result that, the display time in

this method depends on the number of points shown in the screen, not the volume of data, that is, regardless of the amount of LiDAR data, we could draw the scene within an acceptable time.

with display precision		with display precision and screen buffer	
Sum of Points	Render time(ms)	Sum of Points	Render time(ms)
109785	3183.41	38415	3293.98
49468	1617.26	14467	1646.21
20202	759.041	4259	769.161
7743	316.376	1207	320.897
2700	118.496	321	120.296
992	47.2825	85	46.4422
259	11.6568	21	12.4862
123	5.55098	11	5.99378
17	0.800381	4	0.850667
1	0.0798984	1	0.0804572

Table. 1. The parameter in the ways of taking 1.5 pixel precision, and 1.5 pixel precision with screen buffer:

6. CONCLUSION

This paper discusses several aspects in managing and displaying LiDAR data, i.e., data partition using octree, building local KD trees, improving the KD node, and accelerating display of very large LiDAR data. Experiments show that the approach is particularly efficient for very dense points or points far away from the view point. The display time for the whole scene no longer relies on the volume of data, but on the amount of points shown on screen. Nevertheless, KD tree hasn't the ability to insert or delete data dynamically. More efforts are needed for our methods to process LiDAR data more dynamically.

Acknowledgement: The research is funded by the Chinese National High-tech R&D Program (863 Program: 2006AA12Z151).

REFERENCES

David, L., R. Martin, D.C. Jonathan, V. Amitabh, W. Benjamin & H. Robert, 2003. *Level of Detail for 3D Graphics*, Morgan Kaufmann

Jelalian, A., 1992. Laser Radar Systems, Boston: Artech House

Markus, G. & P. Hanspeter, 2007. *Point-Based Graphics (The Morgan Kaufmann Series in Computer Graphics)*, Morgan Kaufmann Publishers, pp.152-154

Moore, A.W., 1991. Efficient memory-based learning for robot control [Ph.D dissertation]. University of Cambridge

Samuel, R.E., 2001. *Binary Space Partioning Trees and Polygon Removal in Real Time 3D Rendering*, Uppsala University, pp.16-34

Yong H., 2003. Automated Extraction of Digital Terrain Models, Roads and Buildings Using Airborne Lidar Data: [Ph.D Dissertation for], Univ. of Calgary