FILTERING OF POINT CLOUDS FROM PHOTOGRAMMETRIC SURFACE RECONSTRUCTION

K. Wenzel, M. Rothermel, D. Fritsch, N. Haala

Institute for Photogrammetry, University of Stuttgart
Geschwister-Scholl-Str. 24D, 70174 Stuttgart, Germany

[konrad.wenzel, mathias.rothermel, dieter.fritsch, norbert.haala]@ifp.uni-stuttgart.de

KEY WORDS: Point Clouds, Photogrammetry, Laser Scanning, Surface Reconstruction, Filtering, Big Data

ABSTRACT:

The density and data volumes for recorded 3D surfaces increase steadily. In particular during photogrammetric surface reconstruction and laser scanning applications these volumes often exceed the limits of the available hardware and software. The large point clouds and meshes acquired during the projects contain billions of vertices and require scalable data handling frameworks for further processing. Beside the scalability to big data, these methods also should adapt to non-uniform data density and precision resulting from varying acquisition distances, as required for data from Photogrammetry and Laser Scanning. For this purpose, we present a framework called Pine Tree, which is based on an out-of-core octree. It enables fast local data queries, such as nearest neighbor queries for filtering, while dynamically storing and loading data from the hard disk. This way, large amounts of data can be processed on limited main memory. Within this paper, we describe the Pine Tree approach as well as its underlying methods. Furthermore, examples for a filtering task are shown, where overlapping point clouds are thinned out by preserving the locally densest point cloud only. By adding an optional redundancy constraint, point validation and outlier rejection can be applied.

1. INTRODUCTION

1.1 Motivation

Large data handling is a constant challenge for the field of 3D surface processing. On the one hand, computer hardware constantly improves and enables the processing of larger datasets. On the other hand, the sensors and methods for data retrieval improve even faster.

A typical example are laser scanners – nowadays scanning with at least one million samples per second. Besides, dense image matching algorithms improve constantly, which enables the retrieval of surfaces using any kind of imagery. Due to the evolving camera technology, sensors with high quality with respect to geometric and radiometric resolution are available. Together with the right software solution, high density datasets can be acquired efficiently.

One example for such a method is our software solution SURE [Rothermel et al., 2012], which retrieves up to one 3D points per pixel. Thus, each image acquired leads to point clouds with at least a few million 3D points depending on the sensor resolution. Since highly overlapping imagery is beneficial for the reconstructed surface in terms of reliability, density, precision, edge sharpness and occlusions, many images are taken. This leads to datasets containing thousands of images and leading typically to billions of points.

In order to be able to handle and process such data sizes efficiently, flexible and intelligent data storage and access methods are required. Out-of-core approaches provide a solution by dynamically storing and loading parts of the data from the hard disk instead of keeping the whole data in the main memory. In order to access data efficiently, tree structures can be used, which reduce the complexity of queries like nearest neighborhood searches. Within this paper, we present an implementation combining these techniques – particularly designed for data from Photogrammetry and Laser Scanning.

1.2 Specification

Big data. The framework should be able to handle point clouds or other spatial data of huge size.

Inhomogeneous data. The distribution, density and precision of the acquired data are varying.

Data queries. Efficient searching on the data is essential for filtering, visualization or queries on bounding volumes.

Data updating. Data adding and removal are essential for manipulation processes such as filtering.

Precision preservation. In order to maintain data quality, simplification, resampling or lossy compression are avoided.

1.3 Previous work

1.3.1 Tree data structures

By sorting in data into tree structures, efficient data queries can be performed. Instead of traversing all elements, decision are taken sequentially on each level of nodes in the tree. In the case of spatial data, the data can be partitioned according to their coordinates.

For example, the quad-tree [Finkel & Bentley, 1974] partitions the space into four equally sized squares. Each square is one node of the tree, so that during a query the complexity can be reduced by a quarter by evaluating the data in the corresponding square only, instead of each item. By splitting again each square into sub-squares until a maximum of elements is contained in each node, a multi-level tree can be created. In 3D space, the same method is applied by partitioning each cube into eight sub cubes, which leads to an octree, as introduced by [Meagher et al., 1980].
Another partitioning method is applied for data-driven R-Tree [Guttmann, 1984], which groups 2D spatial data elements according to their minimum bounding rectangle. Within the subsequent level, the data is again partitioned into a defined number of bounding rectangles. By enforcing a minimum number of rectangles, a balanced data representation is enforced. This means, that the distribution of data in the tree is rather equal and thus, enables equally efficient data queries. The R-Tree is particularly suitable for objects with extension, such as volumetric shapes.

Another popular data driven tree structure is the kd-tree, which was introduced by [Bentley et al., 1975]. At each subdivision, the space partitioned into two parts along one dimension. In order to maintain a balanced tree, the data can be split into two equal parts, as introduced as the adaptive kd-tree. This enables equally efficient data queries in particular for point based data.

When choosing the optimal tree structure for our purposes, we require good properties in particular for point based objects, which makes the kd-tree a good candidate due to his high performance of data queries. However, a key problem of data driven tree structures is the capability of data update. Each point insertion or deletion requires the modification of large parts of the actual tree structure.

For our application, we require high flexibility for adding and removing data. Thus, we prefer the space driven portioning of the octree, even though queries can be slower due to less good balancing of the tree. In order to compensate the drawbacks of the lower balance, smarter access by involving a cached access history can be used.

Another beneficial feature of the octree is the linear implementation due to the splitting into eight equally sized subnodes at each level. This supports fast traversal and prediction. Instead of visualizing the full resolution for the whole scene, only the visible objects are loaded at a minimum Level-of-Detail.

### 1.3.2 Out-of-core

Out-of-core refers to algorithms, which store the data not only in the main memory, but also stream from an external memory source. This is particularly useful when the data to be accessed can be significantly larger to the available main memory.

Out-of-core tree structures are widely used – in research often for visualization, such as [Ueng et al., 1997], [Corrêa et al., 2002] or [Lindstrom, 2003]. Beside this application, also processing on the data is performed – for example Poisson surface reconstruction like in [Boltho et al., 2007] or mesh simplification [Cignoni et al., 2003].

In general, only few open implementations are available. The Point Cloud Library (PCL) at pointclouds.org offers an out-of-core module for point clouds. However, at the current version 1.7, it is rather focused on visualization and uniformly distributed point clouds only, since the level is fixed.

The out-of-core data structure introduced by [Elseberg et al., 2011] and published in the 3D Toolkit, overcomes this limitation and works with dynamically splitting the tree until a minimum number of points are exceeded or predefined maximum depth is reached. Thus, it adapts much better to non-uniformly distributed data, since no subdivision is performed at unoccupied parts of the dataset.

For our purposes of point cloud processing, we would like to implement a flexible data structure focussed on non-uniformly distributed point clouds supporting unlimited custom data fields. For the purpose of automatic adaption to data of varying density, a maximum depth shall be avoided. It shall support frequent adding and removal of data globally and locally, where the tree should adjust automatically.

## 2. PINE TREE

### 2.1 Approach

The Pine Tree is a framework implementing a tree structure designed for the processing of large spatial data. It is exposing methods for data updating and querying, while taking decisions for data storage and retrieval automatically based on these techniques automatically. This enables the development of processing algorithms for large data with low efforts.

The Pine Tree framework is an octree, and thus allows fast data adding and update. Its regular spatial driven portioning can be implemented efficiently for operations in-core, but also enables switching to out-of-core storage for parts of the tree if required. In order to meet the remaining requirements discussed in the specification of section 1.2, several additional features have been implemented.

**Dynamic depth.** Instead of limiting the tree structure to a pre-defined depth, the depth is dynamically adapted locally. By defining thresholds for the maximum data storage, the depth can be controlled dynamically to the needs of the particular tasks of updating and querying. Thus, the requirement of local adaption to non-uniformly distributed datasets is met.

**Dynamic loading and writing.** Since an octree is used, each node is split into eight subsequent nodes. These nodes can be stored in a folder structure on the hard disk, where each folder has eight subfolders. At the final node (leaf node), the data can be stored as a file.
Dynamic memory management. In order to perform efficient processing on the tree, as much data should be in the memory (in-core) as possible, while not exceeding the limits of the main memory. For this purpose, a maximum count of nodes in-core can be defined. The framework detects, writes and de-allocates unused parts of the tree based on a usage history.

Dynamic folding and unfolding. Operations on the trees like nearest-neighbour queries require as few data per node as possible for the matter of efficiency. In contrast to that, the writing of small blocks to the hard disk should be avoided, since the access latency on the disk is low. Also, data is read and written in minimum block sizes so that many small data blocks lead to high reading and writing times.

For this purpose, we implemented a dynamic folding on data writing, where the data of multiple nodes is merged before writing to maintain large data blocks out-of-core. When reading data, the node can be unfolded again, to maintain a small number of samples per node for faster access in-core.

2.2 Implementation

2.2.1 Layers

The Pine Tree is divided into data containers (Nodes), data operators (Node Walkers) – which enable saving memory as well as node tracking in a global data management (Node Manager). Each of these layers can be specified as follows.

Nodes. A node defines the basic element of the tree. It is described by its bounding box, which is defined by three coordinates and a width. Besides its bounding box, a node contains a pointer to its parent node and to its eight child nodes if available. Furthermore, it can contain data.

Node Walkers. The Node Walkers point to a Node and to the Node Manager, enabling various operations on a node. Such operations are data writing and reading as well as node traversal and node merging. Most of the functions are recursively calling themselves in order to operate on multiple levels in the tree.

Node Manager. The node manager is handling the general tree structure. Every Node Walker is registering operations of node creation and removal at the Node Manager. Therefore, the amount of nodes in memory is known to the Node Manager – enabling the dynamic decision for data loading and writing.

2.2.2 Tasks

The key tasks of the tree are implemented for Node Walkers, which point to a specific Node of the tree as well as the Node Manager.

Finding nodes. In order to find the node corresponding to an X, Y, Z position, the tree can be traversed from top to down. At each node, the containing sub-node is found. By repeating this process until the node in the final level is reached, the destination leaf node can be determined.

In order to improve speed for the frequent operation of node finding, we do avoid a check on the bounding box for all eight sub-nodes, but only compare to each coordinate of the node center once. This leads to a binary triple for all three coordinates indicating whether it is greater than the center and then find the corresponding sub-node based on a look-up table.

Data adding. In order to add data to the tree, the destination node is found for each point. Subsequently, the data is added to the node data vector. In order to avoid time consuming data allocation, data copying and data de-allocation as happening for typical push back approaches, we determine the destination nodes firstly and then add the data to the nodes as blocks.

Node splitting and merging. While most in-core processes require a small count of data items per node for efficient operations, the out-of-core part requires large data blocks. Therefore, node splitting and merging is frequently required. During splitting, the node data is distributed to its eight sub-nodes and during merging, the data of the sub-nodes is added to their parent node.

Data reading and writing. In order to write a node to disk, the subsequent tree structure is created as folder structure on the hard drive, where each folder indicates the sub-node number (e.g Tree/0/3/4/2 for a branch node at level 4). Nodes are only written to the hard disk, if they contain data. By merging nodes, the resulting larger files can be accessed more efficiently.

2.2.3 Memory management

The memory management in the Node Manager is based on a Node history. This history is represented by a cycle stack vector of pointers to Nodes with a constant predefined size n. By storing the index i of the last accessed element, the elements can be written in a cycling manner with overwriting always the oldest element.

Node tracking. In order to keep track of the nodes in memory, the history is updated each time a new node is loaded to the memory. This occurs in particular during node splitting and node reading. Instead of keeping track of all leaf nodes, we only track their directly parental branch nodes to reduce the processing overhead.

Node de-allocation. As soon as a new node is registered, we write it to the history vector at the position i, which represents the oldest node. The previous node at this position is written to disk and de-allocated.

Node sharing. The key challenge is the avoidance of de-allocation, if a node is still in use. Besides the fact that multiple processes can add the same node to the history, there might also be a dependency from another process to one of the child nodes. De-allocation would then lead to highly frequent reading and writing operations, which would slow down the process.

The solution to the problem of de-allocating shared nodes, is to keep track whether each branch is been in use by other branches. Instead of explicitly validating this for each element of the history at each time of node registration, we encode this access implicitly in our tree structure.

Each time a node is registered, we store the access number defined by the history in the index i in the node. De-allocation and writing for a node is only performed, if the access number is equal to the current access index i. This way, we can avoid the de-allocation of nodes, which are still required by other nodes in memory.

In order to minimize the overhead of the node tracking, we only register a node pointer if it is not the equal to the previously registered node and if it is not equal to the previously registered...
parent node. This way we can avoid overhead during frequent occurring subsequent splitting steps. Thus, few registrations are required during tree unfolding – e.g. during data adding, in particular if large blocks of data are processed at once or dynamic splitting during processing for faster queries.

Folding and unfolding. A key bottleneck of an out-of-core tree is the access to the hard disk. While for the processing in-core many nodes with small data portions are beneficial, the writing of many small files onto the hard disk requires a lot of time. This is due to the general access latency for each file, but also the overhead of updating the file system tree structure. For this purpose, we merge nodes until they contain a minimum data count before writing and split them after reading.

2.2.4 Usage

The Pine Tree is implemented in a C++ environment. Currently, the tree can contain point clouds with various fields and can easily be extended to other data types like meshes or volumetric representations. Operations on the tree can be implemented using only basic functions like getData or getSubnodes. The whole memory management is performed automatically in background.

3. POINT CLOUD FILTERING

Point cloud filtering is a typical application for spatial trees. In particular, queries for the nearest neighbours to a point are required – for example in order to merge multiple points to one for the matter of resampling.

Typical implementations of such search tree structures require the whole point cloud in-core and then create a tree containing the indexes of the point clouds for queries. Thus, the available physical memory (RAM) is often exceeded, since the dataset itself and the tree structure itself requires too much memory.

A solution for this is often random-subsampling, where only each n-th point is kept while the others are rejected – regardless of their spatial context. However, the loss of data required for further processing leads subsequently to a reduction of quality. Consequently, filtering techniques without a loss in data quality or density are desired.

3.1 Filtering of overlapping point clouds

Within the following section, we would like to present a simple filter exploiting and reducing the redundancy occurring for overlapping point clouds. Within the local neighbourhood only the densest point cloud is preserved, while points from clouds with locally less density get rejected.

This is particularly useful for applications, where point clouds were retrieved from multiple stations – as occurring for point clouds from dense image matching and laser scanning. The overlaying clouds can on the one hand be used to validate each other in order to reject outliers by enforcing a minimum redundancy (minimum fold). On the other hand, only the locally densest cloud is preserved, which rejects noisier data from stations far away.

3.1.1 Algorithm

The core concept of the algorithm is to split the tree, until each node contains from each point cloud one point at maximum. Thus, no node can have two points from the same point cloud.

By preserving the information which cloud was the densest locally during the recursive splitting, we can then preserve only the point from this cloud while rejecting all points from clouds with locally less resolution. Consequently, only the cloud with the highest density remains.

The splitting of the tree is performed by splitting the leaf to become a branch node, recursively calling the filtering function again on the sub-nodes and merging the resulting nodes. Consequently, the tree is unfolded locally for the filtering and subsequently folded again by merging the resulting data.

Additionally to the local density filtering, we can constrain a certain redundancy to perform point validation. This is particularly useful when outliers remain in the point cloud – for example if only stereo image pairs were used for the point retrieval. Thus, a consistency validation can be performed in object space for point clouds.

The redundancy constraint can be applied on the final leaf nodes after the recursive splitting process described above. Since each leaf node can have only one point cloud source, we can reject the leaf as soon as the point count is less than the minimum defined fold (min_fold). Table 1 contains the pseudo code for the whole algorithm including a redundancy constraint.

<table>
<thead>
<tr>
<th>Function MultiCloudFilter ( Node * node, Integer min_fold )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set split_max to min_fold*10</td>
</tr>
<tr>
<td>If node is Branch</td>
</tr>
<tr>
<td>Apply MultiCloudFilter on each Sub-Node</td>
</tr>
<tr>
<td>Else // is leaf</td>
</tr>
<tr>
<td>Set data_count to Node Data Count</td>
</tr>
<tr>
<td>If data_count is less than split_max</td>
</tr>
<tr>
<td>Split this node</td>
</tr>
<tr>
<td>Apply MultiCloudFilter on this node</td>
</tr>
<tr>
<td>Merge this node</td>
</tr>
<tr>
<td>Else If data_count is less than min_fold</td>
</tr>
<tr>
<td>Reject data</td>
</tr>
<tr>
<td>Else // data_count &gt; split max</td>
</tr>
<tr>
<td>Get point cloud source classes</td>
</tr>
<tr>
<td>Get class ID with maximum occurrence</td>
</tr>
<tr>
<td>Set as max_occ_class</td>
</tr>
<tr>
<td>If classes count is less than min_fold</td>
</tr>
<tr>
<td>Reject data</td>
</tr>
<tr>
<td>Else If ( maximum_occurrence &gt; 1 )</td>
</tr>
<tr>
<td>Set densest_cloud to max_occ_class</td>
</tr>
<tr>
<td>Split this node</td>
</tr>
<tr>
<td>Apply MultiCloudFilter on this node</td>
</tr>
<tr>
<td>Merge this node</td>
</tr>
<tr>
<td>Else // Valid leaf reached</td>
</tr>
<tr>
<td>Keep point from class densest_cloud</td>
</tr>
<tr>
<td>Reject all other data</td>
</tr>
<tr>
<td>End If</td>
</tr>
<tr>
<td>End If</td>
</tr>
<tr>
<td>End Function</td>
</tr>
</tbody>
</table>

Table 1: Pseudo-Code for simple Multi-Cloud-Filter
3.1.2 Results

In order to evaluate the performance of the tree as well as the filter, several data sets have been processed with the software SURE [Rothermel et al., 2012]. The resulting highly overlapping point clouds for each image have been added to the tree without previous sorting. Subsequently, the data was added to the PineTree and the filtering approach described in section 3 was applied.

Fig. 1: Troll. 9 images with 4 MP each. Fold: 2. Pointcount: 15.2 Mio. before, 2.3 Mio. after filtering. Time: 40s

Fig. 2: St. Andrews. 43 images with 5MP each. Fold 2. Pointcount: 57.4 Mio. before, 8.3 Mio. after filtering. Time: 105s

Fig. 3: Lady chapel. 73 images with 12MP each. Left: Fold 2, Right: Fold 3. Pointcount: 311 Mio. before, 47.6 Mio. after filtering with fold 2, 15.8 Mio. after filtering with fold 3. Time: ~20 Minutes (no SSD). Dataset: [Moussa et al., 2013]
CONCLUSIONS

Within this paper, we presented an out-of-core data structure in order to be able to process large amounts of spatial data. It is based on a regular octree and is particularly designed for non-uniformly distributed datasets due to its dynamic level structure. A memory management layer based on an access history is used to dynamically read and write data from and to the hard-disk, in order to limit the amount of required memory in-core. Automatic folding and unfolding of the tree enables a flexible adjustment of node density during processing.

The resulting framework is currently supporting point clouds and can thus be used for any kind of data querying due to the spatially indexed data structure. This is particularly useful for efficient nearest neighbor searches like required in filtering tasks.

We presented a simple filter application for the removal and usage of redundancy in overlapping data sets. It preserves the locally densest cloud, which is in particular useful if point clouds with varying resolution are combined. Additionally, a redundancy constraint can be used to validate point clouds in object space.

Within future work such filtering approaches can easily be extended to exploit additional information such as precision information or to utilize the available redundancy to reduce the local noise. Currently, the Pine Tree framework is focused on point clouds. Within future work, it can be extended to support meshes and scalar fields. Beside data processing, the tree structure can also be utilized for further processing and visualization tasks.

ACKNOWLEDGEMENTS

We would like to acknowledge the work of Minwei Tang, who evaluated existing out-of-core algorithms in his study thesis “Out-of-core algorithms for large point clouds in photogrammetric applications.”

REFERENCES


