AN OUT-OF-CORE OCTREE FOR MASSIVE POINT CLOUD PROCESSING

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, Big Data, Photogrammetry, Laser Scanning, Filtering, Surface Reconstruction, Dense Matching

ABSTRACT:

Image based surface acquisition using dense image matching methods enables the retrieval of 3D information for each pixel. While the sensor resolutions increase, also the desire of acquiring and handling datasets with more images increases. Consequently, massive point clouds and other surface representations are produced, which need to be accessed and processed efficiently. For this purpose, we present a framework called Pine Tree, which is based on an out-of-core octree. It is currently focused on point clouds, but can be extended to other data types such as meshes or volumetric representations. It enables spatially indexed data storage and quick data queries. Moreover, only parts of the data are loaded from the hard disk to the memory, in order to be able to process big data on common hardware. Within this paper, we present the Pine Tree framework as well as an example filtering operation, which uses the redundancy of overlapping point clouds in order to perform outlier rejection and data reduction while preserving accuracy.

1. INTRODUCTION

Point cloud datasets in common projects can contain billions of points nowadays – in particular for dense image matching applications, where on the one hand the amount of images and on the other hand the resolution for each image increases. Quick data access is essential for general bounding box retrieval, but also for processing tasks like filtering, surface reconstruction or segmentation.

For this purpose, we would like to implement a flexible data structure focused on non-uniformly distributed point clouds supporting unlimited custom data fields. In order to adapt automatically to data of varying density, a maximum depth shall be avoided. Furthermore, frequent update and removal of data should be supported. Consequently, the following requirements can be defined.

Big data. The framework should be able to handle massive point clouds or other big spatial data.

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.

1.1 Out-of-core octree

In order to meet the specified requirements concerning data querying, we are using an octree [Meagher, 1980] as access structure. Octrees are based on a space-driven partitioning approach which can have the disadvantage of imbalance if the data is not well distributed. In such cases, data-driven approaches like KD-Trees as proposed by [Bentley et al., 1975] would require less memory for the tree structure and enable faster data access.

However, in our case we want to support quick data update, which is better supported by the regular octree, since the tree partitioning is not relying on the current data, and thus, doesn’t require an update of the structure when adding or removing data. Furthermore, the octree is well suited for out-of-core implementations, since the partitioning at each level is identical and thus, does not require reading additional information.

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 than 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 [Bolitho 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 tree is created completely for an a priori defined depth.

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.
2. PINE TREE

2.1 Approach

The Pine Tree framework provides methods for data updating and querying, while taking decisions for data storage and retrieval automatically. This enables the development of processing algorithms for large data with low efforts.

In order to meet the remaining requirements discussed in the specification of section 1, several features have been implemented:

**Dynamic depth.** Instead of limiting the tree structure to a predefined 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.

2.2 Tasks

**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.

**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.

2.3 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. Beside 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.

2.4 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 getNodes or getSubnodes. The whole memory management is performed automatically in background.
3. POINT CLOUD FILTERING

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. On the other hand, only the locally densest cloud is preserved, which rejects noisier data from stations far away.

3.1 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 consistency 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.

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). Results are shown in the appendix.

4. CONCLUSIONS

Within this paper, we presented an out-of-core octree structure for processing massive point clouds. By indexing data spatially bounding box or nearest neighbor queries can be performed efficiently for tasks like filtering or the retrieval of regions of interest. The out-of-core memory management only loads desired parts of the dataset from the hard disk and thus enables big data handling on common hardware.

Within an example application, overlapping clouds from multiple sources are filtered by preserving only the locally densest point cloud. Additionally, redundancy constraints can be used to validate data and to reject outliers.

Within future work, the tree structure can be extended to support other data types such as meshes or scalar fields to achieve a volumetric data representation. Furthermore, the tree can be applied for visualization purposes or tasks like simplification, segmentation and modeling.

5. 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.”

6. REFERENCES


Appendix: 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. The point density is adapting to the local maximum resolution available, while rejecting redundancy and outliers.

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

Fig. 3: Perth - airborne image dataset. Left: result overview, right: detail. Pointcount: 394 Mio. points before filtering, 73 Mio. points after filtering with fold 2. Imagery kindly provided by Aerodata International Surveys.

Fig. 4: Munich – airborne image dataset. Result for filtering with fold 2. Pointcount: 2.8 Bio. points before filtering, 437 Mio. points after filtering with fold 2, 218 Mio. points after filtering with fold 3.