Analysis of Interactive Editing Operations for Out-of-Core Point-Cloud Hierarchies

Claus Scheiblauer
Vienna University of Technology, Austria
scheiblauer@cg.tuwien.ac.at

Michael Wimmer
Vienna University of Technology, Austria
wimmer@cg.tuwien.ac.at

ABSTRACT

In this paper we compare the time and space complexity of editing operations on two data structures which are suitable for visualizing huge point clouds. The first data structure was introduced by Scheiblauer and Wimmer [SW11] and uses only the original points from a source data set for building a level-of-detail hierarchy that can be used for rendering points clouds. The second data structure introduced by Wand et al. [WBB+07] requires additional points for the level-of-detail hierarchy and therefore needs more memory when stored on disk. Both data structures are based on an octree hierarchy and allow for deleting and inserting points. Besides analyzing and comparing these two data structures we also introduce an improvement to the points deleting algorithm for the data structure of Wand et al. [WBB+07], which thus allows for a more efficient node loading strategy during rendering.

Keywords

viewing algorithms, point clouds, complexity analysis, data structures

1 INTRODUCTION

Within the last 10 years the generation of huge point clouds with more than 10^9 points has become quite common, due to the fact that distance measurement devices like laser or triangulation scanners have increased the scanning speed and density of their sampling measurements. The latest generation of laser scanners is capable of generating point data sets consisting of 10^9 points or more with a single scan [Rie13, FAR13]. Rendering or processing point clouds of such huge sizes is still a challenge, as they usually do not fit into the main memory of normal computer workstations [WBB+07, SW11]. Even though nowadays 16GB of main memory are not uncommon, it is still not enough to handle those point clouds, as the data for the point positions alone consumes 12GB, assuming that every point position consists of three 32bit floating point numbers.

Although it is possible to split point clouds into smaller parts and work only on these, the general view for the complete data gets lost when only working with a part of the complete point cloud. Another possibility would be to downsample the original data, but then the editing operations can only be performed on a coarser resolution of the point cloud than would be available. Therefore data structures that allow to render, process, and search within huge point clouds are necessary if an overview of the whole data set is advantageous, or if tasks have to be accomplished on the original data set.

The contributions of this paper are:

- A complexity analysis and comparison of the Modifiable Nested Octree (MNO) introduced by Scheiblauer and Wimmer [SW11] and the octree-based data structure introduced by Wand et al. [WBB+07].
- An improvement to the points deleting algorithm for the data structure of Wand et al. [WBB+07], which thus allows for a more efficient node loading strategy during rendering.

We show that editing and processing operations on the points stored in these data structures are comparable in complexity to editing and processing operations on specialized data structures. These operations can be performed on out-of-core managed point clouds efficiently, if the data is managed properly with a least-recently-used (LRU) cache [Den68, Sam06].

2 PREVIOUS WORK

There exists a plethora of data structures that can be used to store and process point data sets. A structured overview of the most well known variants is given by Samet [Sam06]. Deciding on which data structure to use is dependent on the specific use case and task that

shall be accomplished. Each data structure is designed for a set of use cases, and one data structure is usually not the best choice for all tasks. Another distinction between the variants of the data structures are the implementation complexities of each.

A simple data structure for ordering and fast searching of points in 2D is the point quadtree. A quadtree is a hierarchical data structure where at each node the underlying space is subdivided into 4 subnodes (children) along axis-parallel lines that go through a given data point. An octree is analogous to a quadtree, but in 3D. Exactly one point is stored at each node of the hierarchy. Searching or inserting a point triggers a $O(log(N))$ complexity search from the root node to the required node. N is the total number of points inserted. Deleting a point is quite involved, as the hierarchy has to be maintained also when points from interior nodes have been deleted, and the simplest method to do this is to rebuild the data structure from scratch.

While a point quadtree subdivides the data space of the points, other variants like the point region quadtree (PR quadtree) subdivide the space the data exists in. Nodes of the PR quadtree are subdivided at arbitrary axis-aligned lines, e.g., the ones through the center of the node, not at lines that go through given data points as in the ordinary quadtree. Another difference is that data points are only stored at leaf nodes, while interior nodes are used for searching the requested point in the proper node. Complexities for inserting and searching are dependent on the height of the PR quadtree, which in turn is dependent on the minimum distance between 2 neighboring points. To alleviate this dependency, a bucket PR quadtree can be used. In this variant of the PR quadtree up to m points are allowed in a leaf node.

If a data set is too large to fit into main memory, n-way trees can be used to access such a data set. A well known data structure for this use case is the B-tree [BM72]. Like the bucket PR quadtree it stores the data in its leaf nodes and the keys for accessing the data in its interior nodes. There are two main differences to a bucket PR tree though, first the number of children per node is not limited to 4, and second, the leaf nodes are all at the same tree level. A B-tree has only 3 to 5 levels typically [Bay08], and therefore it is a very efficient data structure for accessing points on slow secondary storage devices, because the number of accesses to the storage device is kept low. B-trees and its variants like the B+ tree or the B* tree are also used by file systems, e.g., by NTFS, the Windows file system [Mic03], to manage file accessess.

Other requirements for a data structure occure when huge point sets have to be rendered to screen. In this case the points have to be organized in a way that is easy to handle for a graphics card, but also efficient to load from disk. Ideally the points are available in chunks that can be directly used for rendering. Another requirement is that the data structure has to support a level-of-detail (LOD) mechanism. Data structures have been proposed [GM04, PSL05] which comply to these requirements. They hold the tree access structure in memory all the time, and only load the visible points from disk, which is efficient for rendering. Editing operations on these point data structures are inefficient for different reasons. In [GM04] a uniform sampling density is assumed, and the original point set is subsampled to get the points for each hierarchy node. Adding new points, which invalidate the uniform sampling assumption, is not supported. In [PSL05] a global indexing scheme is applied to the point set which has to be updated if some points were to be removed or added.

Data structures that allow for rendering but also for efficient editing operations on out-of-core managed point clouds have been proposed [WBB+07, SW11]. We compare the editing operations of these two data structures, where Wand et al. [WBB+07] use additionally created points and Scheiblauer and Wimmer [SW11] use original points for a LOD hierarchy.

Processing data sets out-of-core can be done efficiently if the editing algorithms load chunks of data into memory before processing them, and only work on these chunks [Vit08]. Samet [Sam06] mentions that holding recently accessed chunks in a cache store could further reduce the number of disk accesses when processing data. Such cache stores are implemented in all before mentioned point rendering and editing algorithms.

3 MICHAEL WAND OCTREE RECAP

Wand et al. [WBB+07] introduced a multiresolution hierarchical data structure for rendering and manipulating huge point clouds. It is based on a region octree [Sam06], which is a data structure where the children of a node are congruent boxes regularly subdividing the space of the parent node. In the Michael Wand Octree (MWO) each interior node has inscribed a grid, and each cell of the grid can store one point. These points are used for a LOD representation of the point cloud. They are additionally created during build up of the data structure. The original points are stored in the leaf nodes. An example of a 2D MWO with 3 levels can be seen in Figure 1.

For our tests we do not use the original implementation of the point hierarchy in the XGRT system [WBB+09], but re-implemented it according to [WBB+07] in our point cloud rendering framework. Selecting points is done with a Selection Octree [SW11], which is a separate data structure describing a space inside which all points are selected.

3.1 Inserting points

Listing 1 shows how points are inserted into an MWO. Leaf nodes can only hold up to a predefined threshold

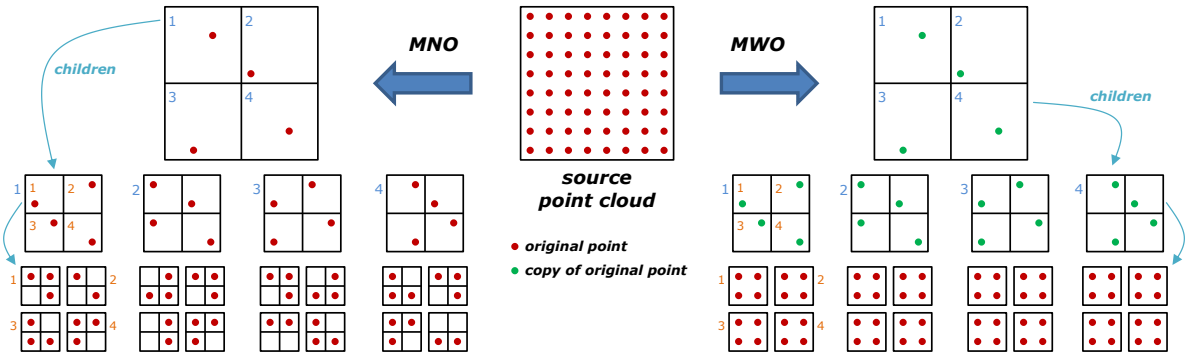

Figure 1: A source point cloud represented as MNO (left) and MWO (right). Both point hierarchies use a 2x2 grid at the interior nodes. The MNO uses the grid also at the leaf nodes, while the MWO uses an array.

m_{leaf}, e.g., 100.000 points. Each interior node holds a grid, and at each grid cell a point is stored. Furthermore a 64bit integer is maintained to count the points falling into a grid cell. For the grid we use a resolution of 2^7 cells on each of the 3 space axis, so in total a maximum of $(2^7)^3 = 128^3$ points can be stored at one interior node. Usually the number of points in an interior node is much lower for the data sets we tested.

```
foreach point P in source point data do  // Main loop
  root node R->insert(P)
end

node:: insert (P):                       // Method
  if node is leaf
    insert P into array of points at node
    iterate ()
  else
    insert P into grid
    if grid cell GC containing P is empty
      store a copy of P in GC
    increment counter of GC by 1
    appropriate child C->insert(P)
end

node:: iterate ():                       // Method
  if num points in array > m_leaf   // m_leaf=max num points
    create grid from points
    foreach point P in array do
      appropriate child C->insert(P)
    convert this node to inner node
end
```

Listing 1: Inserting points into an MWO.

The points of the nodes are stored in one file per node. Leaf nodes store only point data, while the interior nodes store the point data and the counter for each point in the file.

3.2 Deleting points

Listing 2 shows how points are deleted from an MWO. The points which are going to be deleted have to be selected beforehand, which is done using a Selection Octree [SW11]. All leaf node points inside the Selection Octree are then removed from the MWO, and the hierarchy is checked for consistency.

```
foreach leaf node N inside or intersecting SelOctree do
  define array of deleted points D
  if N is inside SelOctree
    put all points into D
  else
    put only points inside SelOctree into D
  if all points of N are in D
    delete N from disk
  foreach point P in D do
    foreach parent node PN up to root node R do
      decrease counter of grid cell containing P by 1
root node R->validate()
end

node:: validate ():                      // Method
  foreach child C that is an inner node do
    C->validate()
  sum up points N in direct and indirect leaf node children
  if N <= m_leaf                         // m_leaf=max num points
    pull up all points from leaf nodes to this node
    delete empty nodes from disk
    convert this node to leaf node
end
```

Listing 2: Deleting points from an MWO.

Note that there is a subtlety involved when deleting the points. When inserting new points into the hierarchy, the representative points in the grid cells are chosen from the points inserted into a node, e.g., a copy of the first point that falls into a grid cell. When deleting points it might happen that the representative point is the copy of a point that has been deleted, therefore the representative point is not covering the space the remaining points are actually in. This can lead to problems during rendering when using certain types of hierarchy traversals. See Section 4 how these can be solved.

3.3 Rendering

During rendering a depth-first traversal chooses the nodes that will be rendered in the current frame. For this the octree is traversed until the screen-projected size of a node is below a user defined threshold. This node and all of its siblings (inside the view frustum) are then chosen for rendering. Note that the screen-projected sizes of the siblings are not considered, as all

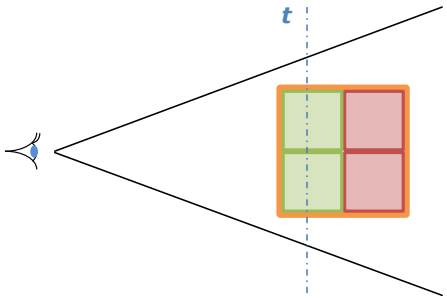

Figure 2: Viewfrustum with an interior node of an MWO and its children. The front of the node has passed the threshold distance t, and the 2 green children have to be rendered. The 2 red children need not be rendered.

siblings in the view frustum have to be chosen to completely replace the parent node (see also Section 4). The loading of the nodes happens asynchronously in a separate thread, so the parent node is still rendered as a coarser LOD level until the children are available.

4 MICHAEL WAND OCTREE OPTIMIZATION

The MWO uses representative points in the interior nodes to enable a LOD mechanism for point clouds. During rendering a depth-first traversal is used for choosing the nodes to render (see Section 3.3). This traversal tends to render more nodes than necessary for the current viewpoint. An example can be seen in Figure 2. The viewpoint is on the left, the orange square represents an interior node of the hierarchy, and the green and red squares represent the interior node's children. The threshold distance t is the distance at which the projected size of the orange node becomes larger than the allowed threshold (for the threshold we use the projected side length of the node's bounding box). A child is thus rendered as soon as its projected double size is larger than the threshold (which is the size of the parent node at the child's position), and its projected size is still smaller than the threshold. In Figure 2 only the front half of the orange node has passed t, therefore only the 2 frontmost children of the node fulfill the conditions to be rendered. The back half of the parent node is still beyond t, so the 2 backmost children need not be rendered, but in a depth-first traversal all children have to be rendered as soon as one child is in front of t.

Instead of a depth-first traversal an importance driven traversal can be used. The importance can be based on different parameters, e.g., the distance to the view point, or how centered the node is on the screen. The importance driven traversal stops when no more nodes can be found which fulfill the conditions to be rendered. In such a traversal no nodes will be rendered whose screen-projected size is too small. In Figure 2, only the 2 frontmost children will be rendered together with the parent node, as its LOD representation still suffices to appropriately represent the point model in the back area. This often reduces the number of nodes that have to be loaded during rendering, as can be seen in the results (Section 8).

The importance driven traversal requires to render nodes and their children at the same time. Therefore it is important that the points in the interior nodes of the octree actually represent the geometry of the point model, otherwise artifacts as shown in Figure 3 might occur. The center image is rendered with an importance driven traversal, and the points in the interior nodes do not represent the actual geometry of the point model. This can happen if the positions of the points in the interior nodes are not updated after deleting points from the leaf nodes. The image on the right shows no artifacts, and this can be achieved in two different ways, either by using a depth-first traversal, or by using an importance driven traversal and updating the position of the points in the interior nodes during deleting. The pixel overdraw on screen due to the additionally rendered parent nodes for the importance driven traversal is neglectable, as usually less nodes have to be rendered, and nodes that have all children available are not rendered.

This means, depending on the effort spent when deleting points, different rendering traversals can (or have to) be used. Since an important driven traversal is often favorable, after deleting points we are replacing the points in the interor nodes with points still available in the leaf nodes. For every grid cell of an interior node whose representative point is inside the Selection Octree (and the point's copy in the leaf node has therefore been deleted), we search for a leaf node in the hierarchy that intersects this grid cell, and take a random point (that is also inside the grid cell's bounding box) of this leaf cell as new representative point. This step can be done in a separate traversal, after deleting points from the leaf nodes. The overhead due to this traversal is very low (see results in Section 8).

5 MODIFIABLE NESTED OCTREE RECAP

The MNO is a data structure designed for fast rendering, inserting, and deleting of points in an out-of-core setting [SW11]. This is achieved by a hierarchy of grids which are actually nodes in a region octree [Sam06]. Figure 1 depicts 3 levels of a 2D version of an MNO. Each level further down the hierarchy refines the point cloud representation of the upper levels.

5.1 Inserting points

Listing 3 shows how points are inserted into an MNO. We use a resolution of 2^7 cells on each of the 3 space axis for the grids in the nodes, so up to 128^3 points can be stored at one node.

Figure 3: From the original point cloud (left image) a spherical volume is deleted. The center image shows the artifacts that remain when the interior nodes are not updated after deleting points. In the right image the points in the interior nodes have been updated according to the deleted volume.

```
foreach point P in source point data do  // Main loop
  root node R->insert(P)
end

node::insert(P):                          // Method
  insert P into grid
  if grid cell GC containing P is empty
    store P in GC
  else
    store P in array of points for child C
    if num points in array for child C >= m_min
      if num points in node without points for C >= m_min
        foreach point PC in array for child C do
          C->insert(PC)
        delete array for child C
end
```

Listing 3: Inserting points into an MNO.

5.2 Deleting points

```
root node R->delete()
root node R->validate()
end

node::delete():                           // Method
  foreach child C inside or intersecting SelOctree do
    C->delete()
  if node is inside SelOctree
    delete node from disk
  else
    if node is leaf
      delete points inside SelOctree
    else
      foreach point P inside SelOctree do
        intersect grid cell GC containing P with direct and
            indirect child nodes
        delete P
        if at any child node exists a point PC inside GC
          insert PC into GC
end

node::validate():                         // Method
  foreach child C that is an inner node do
    C->validate()
  foreach child C do
    if num points in C < m_min
      insert all points from C into this node
      delete C from disk
end
```

Listing 4: Deleting points from an MNO.

Listing 4 shows how points are deleted from an MNO. The points to be deleted are first selected with a Selection Octree [SW11] (see also Section 3). Points are deleted from leaf and interior nodes. Afterwards the hierarchy is checked for consistency.

5.3 Rendering

Rendering of an MNO is done with an importance driven traversal of the hierarchy nodes. Nodes that are in the center of the viewport and close to the viewpoint are most important. Nodes are chosen for rendering until a given maximum number of rendering points is reached, or until further nodes cannot be considered because their screen-projected size is below a pre-defined threshold. Nodes required for rendering but not available in memory are then requested to be loaded from disk. This is done asynchronously, so rendering does not stop while waiting for the points to be loaded.

6 COMPLEXITY ANALYSIS FOR IN-CORE PROCESSING

The MNO [SW11] as well as the MWO [WBB+07] are designed for rendering and accessing large amounts of point data in short time. Search operations on the other hand are not as efficient compared to search operations in specialized data structures. The largest bottleneck is loading points from disk to memory, which can be alleviated to some degree by using an LRU cache. We will first analyze the behavior of the data structures assuming infinite memory to exclude the effects of disk access.

6.1 Building Up Modifiable Nested Octree

In 2D space, building up a Modifiable Nested Quadtree is similar to building up a point quadtree [FB74], as both data structures store points in interior nodes as well as leaf nodes. While the point quadtree subdivides space at given data points, the Modifiable Nested Quadtree creates a hierarchy of congruent squares and each hierarchy level subdivides the space at the center of the previous hierarchy level. A point quadtree stores at each node exactly one point. The average build up

	Build up		Delete	
Hierarchy	Time	Space	Leaf Node	Inner Node
MNO	$O(N)$ to $O(Nlog(N))$	$O(N)$	$O(log(N))$	$O(log(N))$
MWO	$O(N)$ to $O(Nlog(N))$	$O(NC)$	$O(M+log(N))$	$O(M+log(N))$

Table 1: The time and space complexities for building MNO and MWO hierarchies, and for deleting points from leaf and inner nodes.

	Search		
Hierarchy	Single Point	Sphere	Cuboid
MNO	$O(log(N))$	$O(F+2^nM)$	$O(F+4^nM)$
MWO	$O(M+log(N))$	$O(F+2^nM)$	$O(F+4^nM)$

Table 2: The time complexities for searching points in MNO and MWO hierarchies.

	Build up Time		Delete	
Hierarchy	Worst Case	Average	Single Point	Average
MNO	$O(CN+CNlog(N))$	$O(kC+Nlog(N))$	$O(C+Clog(N))$	$O(kC+Nlog(N))$
MWO	$O(CN+CNlog(N))$	$O(kC+Nlog(N))$	$O(C+M+Clog(N))$	$O(kC+NM+Nlog(N))$

Table 3: The time complexities for the out-of-core build up and deleting points from MNO and MWO hierarchies.

	Search		
Hierarchy	Single Point	Sphere	Sphere Avg.
MNO	$O(Clog(N))$	$O(CF+2^n(M+C))$	$O(kC+F+2^nM)$
MWO	$O(M+Clog(N))$	$O(CF+2^n(M+C))$	$O(kC+F+2^nM)$

Table 4: The time complexities for searching points in MNO and MWO hierarchies.

time for a point quadtree is proportional to $Nlog_4(N)$ where N is the total number of points used for building the quadtree[FB74].

In 3D it becomes a point octree. For a complete point octree the average build up time becomes roughly $Nlog_8(N)$. This estimation can be derived by calculating the total path length (TPL) [WN73] for a completely filled point octree. The TPL is defined to be the summed up path length when searching all nodes in a tree, and starting the search always at the root node. The path length of a node and one of its direct children is 1.

Like a point octree an MNO stores points at interior nodes as well as leaf nodes. In an MNO though up to m (e.g., 128^3) points can be stored at each node. When using a hash table for storing the points, inserting a point into a node takes constant time on the average [CLRS09]. The possible reduction is therefore dependent on the ratio m/N. If this ratio is ≥ 1, then complexity becomes $O(N)$. For ratios approaching 0, the complexity again approaches $O(Nlog_8(N))$, which is equivalent to $O(Nlog(N))$ (see Table 1).

For point clouds from laser scanners the number of points that are stored at each node is much smaller than m, since the points are distributed highly nonuniform in space. Therefore $m \gg N/k$, where k is the number of nodes in an MNO. Huge point clouds where $N \gg N/k$ show a complexity of roughly $O(Nlog(N))$, while smaller point clouds tend to show a $O(N)$ complexity. The space requirements for storing an MNO on disk are $O(N)$, since no additional points have to be saved.

6.2 Building Up Michael Wand Octree

In 2D space a Michael Wand Quadtree can be compared to a bucket point region quadtree (bucket PR quadtree) [Sam06]. This is a quadtree where the children of a node subdivide the node's space into four congruent squares. The data points are only stored in leaf nodes, and the interor nodes are used to direct the traversal of the quadtree. Each leaf node stores up to m_{leaf} points. The Michael Wand Quadree stores for the level-of-detail mechanism (see Section 3) additionally up to $m_{interior}$ points in the interior nodes.

In an octree where the points are only stored at the leaf nodes, the time complexity for build up is $O(Nlog_8(N)) \Rightarrow O(Nlog(N))$, according to the total path length (see also Section 6.1). Using buckets at interior nodes and leaf nodes the complexity for building up an MWO changes dependeing on the ratio m_{leaf}/N. For ratios ≥ 1 the time complexity becomes $O(N)$, and for ratios approaching 0 the time complexity approaches $O(Nlog(N))$ (see Table 1).

The distribution of the samples in the point cloud determines the fan out of the hierarchy nodes as well as the fill rate of the grids at the interior nodes. If the point cloud represents a line, i.e., an object with dimensionality $d = 1$, the fan out of the interior nodes will be similar to a binary tree, i.e., each node will have about 2 child nodes. Similar for $d = 2$ or $d = 3$ the interior nodes will have about 4 or 8 children. Assuming a grid with G cells per axis, then the grid has approximately G^d cells filled. Note that since the MWO uses point buckets at the nodes, their fan out will rather represent the global characteristic of the data set. The granularity

of the nodes is too big to capture the true dimensionality. For a better estimation of the dimensionality, the points within the nodes have to be used.

We are not making any assumptions about the point data, therefore an initial estimate for d, without having previously built MWOs of similar data sets, is very difficult. Instead we use the average number of points in the interior nodes, X, of a previously built MWO as parameter for the approximation of the dimensionality. With X and G given, then $d = log_G(X)$. From this an estimation for the fan out F can be derived as $F = 2^d$.

When given the total number of points N in a data set and having an estimation for F and the related dimensionality d, the space requirements for the interior nodes of a new MWO of a similar data set can be evaluated using the following 3 equations.

The total number of leaf nodes L can be estimated by using F as an approximation to the number of new leaf nodes that are created when a leaf node has to be split during build up. The maximum number of leaf nodes is then bound by

$$L = \frac{N * F}{m_{leaf}} \tag{1}$$

and on the average each of the new leaf nodes will hold at least m_{leaf}/F points. Having an estimation for L, the total number of interior nodes I_{nodes} can be estimated with

$$I_{nodes} = \frac{L-1}{F-1} \tag{2}$$

which is the number of interior nodes for any tree with an average fan out F. By using the dimensionality d together with G, the number of grid cells on one axis, the total number of points in the interior nodes I_{points} can be evaluted by

$$I_{points} = I_{nodes} * G^d \tag{3}$$

The space complexity can be written as $O(NC)$, where C is a constant depending on the estimated parameters d and F as well as on the given parameters G and m_{leaf}. For $L \gg 1$ the term $L-1$ can be replaced by L and Equation (2) can thus be approximated by $L/(F-1)$. With this simplification C can be written as

$$C = 1 + \frac{F * G^d}{(F-1) * m_{leaf}} \tag{4}$$

For example a 2D object with $d = 2$ and $F = 4$ has a value of $C = 1 + (4/3) * (G^2/m_{leaf})$.

When using the parameters suggested by [WBB+07], $G = 128$ and $m_{leaf} = 100.000$, then $C = 1.218$ for a two-dimensional object and $C = 24.97$ for a complete 3D volume. This means that for point clouds representing a single 2D area the number of additional points I_{points} is estimated to be 21% of the number of original points, while for a point cloud representing a complete 3D volume it is 24 times the number of original points. For the point clouds we tested with these parameters the number of additional points was 40% of the original points, suggesting the dimensionality of the data sets is almost totally 2D.

6.3 Searching

Searching for a single point in one of the two hierarchies means first searching for the node that holds the point and then searching for the point within this node. The complexities are listed in Table 2. The search for a point within an MNO node is of constant complexity $O(1)$, since the points in the nodes are stored inside grids (which are built once with an $O(N)$ complexity). For an MWO the search for a point inside the leaf node is of $O(M)$, if no search data structure (like a kd-tree [Ben75]) for the points in the leaf nodes has been created.

6.3.1 Modifiable Nested Octree

A range search in an MNO is similar to the range search in a PR quadtree, only in 3 dimensions. A PR quadtree subdivides the space hierarchically into congruent squares. All points are stored in its leaf nodes, one point per leaf node. A range search in a PR quadtree, where the range is a rectangular area limited by axis aligned lines, has a complexity of $O(F + 2^n)$, where F is the number of reported points and n is the maximum depth of the PR quadtree [Sam06]. The 2^n term is derived from the maximum number of nodes that one line of the rectangle can intersect with. A PR quadtree in the 3D case becomes a PR octree. Searching all points in a PR octree that are inside an axis aligned rectangular cuboid is proportional to $O(F+4^n)$, where 4^n accounts for the lateral surfaces of the cuboid intersecting the octree nodes.

A range search in an MNO has to take interior nodes as well as leaf nodes into account. This means all levels of the MNO contribute to the search algorithm complexity. This results in a performance proportional to $O(F + \sum_{i=0}^{n} 4^i) \Rightarrow O(F + 4^n)$. The complexity does therefore not increase for an MNO.

When using a sphere as the search range, it has to be discretized for counting the nodes intersecting the sphere. The number of nodes intersecting a discretized sphere is the solution to the diophantine inequation $(R-1/2)^2 \leq x^2+y^2+z^2 < (R+1/2)^2$ [And94], where $R \in \mathbb{N}$ is the radius of the discrete sphere, and $x, y, z \in \mathbb{Z}$. For increasing R it converges to $4\pi R^2$. The worst case complexity of the range search is found when using

$R = 2^n/2$. This is the radius of the inscribing sphere for the axis aligned bounding box of the root node, expressed in number of nodes at depth n. The complexity of this range search is thus proportional to $O(F + \sum_{i=0}^{n} 4\pi 2^{2i}/2^2) \Rightarrow O(F + \sum_{i=0}^{n} 2^{2i}\pi) \Rightarrow O(F + 2^n)$.

After identifying the nodes intersecting with the search range, all points within the intersecting nodes have to be tested if they are inside or outside the search range. This is of linear complexity $O(M)$ per node, where M is the maximum number of points in a node. In total the range search complexity for a cuboid inside an MNO is $O(F + 4^n M)$ and for a sphere it is $O(F + 2^n M)$.

6.3.2 Michael Wand Octree

During a range search in an MWO points only have to be searched for in the leaf nodes, contrary to an MNO, but since the upper levels of the MNO do not contribute to the search complexity, both hierarchies have the same complexities (see Table 2).

6.4 Deleting

Deleting a point from a leaf node means basically searching for it, and on the average this has a time complexity of $O(log(N))$ (see Table 2). In an MWO points are always deleted from the leaf nodes and the interor nodes are updated later. In an MNO points are directly deleted from leaf nodes and interior nodes.

6.4.1 Modifiable Nested Octree

In a node the search for the grid cell a point falls into can be done in $O(1)$. When a point is deleted from a leaf node, the LOD structure remains intact. When a point is deleted from an interior node, two more steps are required, i.e., finding a replacement point and pulling it up to the node where the other point has been deleted from.

A replacement point is searched for in a leaf node that encompasses (or is inside of) the bounds of the node's grid cell the other point was deleted from ($O(log(N))$ complexity). This replacement point is then deleted from the leaf node and inserted into the node the other point was deleted from ($O(1)$ complexity). Therefore the total complexity of deleting a point from an interior node is $O(2\,log(N)) \Rightarrow O(log(N))$.

6.4.2 Michael Wand Octree

The search for a point in a leaf node is done in linear time $O(M)$, where $M = m_{leaf}$. After deleting the point from the leaf node, the interior nodes on the path from the leaf node to the root node have to update the counters of the grid cells into which the point falls ($O(log(N))$ complexity). The total complexity for deleting a point is $O(M + 2\,log(N))$.

When using the optimization of Section 4 the points of the interior nodes inside the Selection Octree have to be updated. They have to look for a replacement point in an existing leaf node. With this additional step the total complexity for deleting a point is $O(M + 3\,log(N)) \Rightarrow O(M + log(N))$.

7 COMPLEXITY ANALYSIS FOR OUT-OF-CORE PROCESSING

Having only a limited amount of main memory, the point data can become larger than the available main memory and an out-of-core management for the data becomes necessary. This way the points of the hierarchy nodes are streamed from disk during processing. Accessing the disk has a large overhead compared to memory access, therefore reducing the number of disk accesses is important for the efficiency of the out-of-core data management. In the worst case, e.g., if the locality of an operation on the point cloud is bad, each access to a hierarchy node is preceded by a disk access.

One strategy to reduce the number of disk accesses is to use an LRU cache, which manages nodes according to the time they were last used. If points for a new node have to be loaded from disk, the points of the node which has been accessed the longest time ago are swapped out of memory. Generally, using a memory caching method can increase performance a lot. The parameters influencing the efficiency of a LRU cache are the size of the cache and the access pattern of the operations to the nodes of the point-cloud hierarchy.

7.1 Building up and Inserting

The complexities for building up a point-cloud hierarchy are given in Table 3. C is a huge constant representing disk access. The term NC accounts for writing every point to disk, and $CN\,log(N)$ for loading the necessary nodes. In the average case, when using an LRU cache, nodes can often be accessed in the cache. Parameter k is the total number of nodes in the hierarchy. To reduce disk access time a solid state drive (SSD) can be used.

7.2 Searching

The complexities for searching are given in Table 4. We give only the complexities for the spherical search, as the cuboid search can be derived analogously. In the average case the parameter kC accounts for loading and reporting the found points.

7.3 Deleting

The complexities for deleting are given in Table 3. C accounts for writing the changed nodes to disk. In the average case parameter kC accounts for loading and saving the nodes. The dominating terms are dependent on the distribution of the deleted points. If only few points per node are deleted, kC will be the dominating term, otherwise $N\,log(N)$ will be dominating.

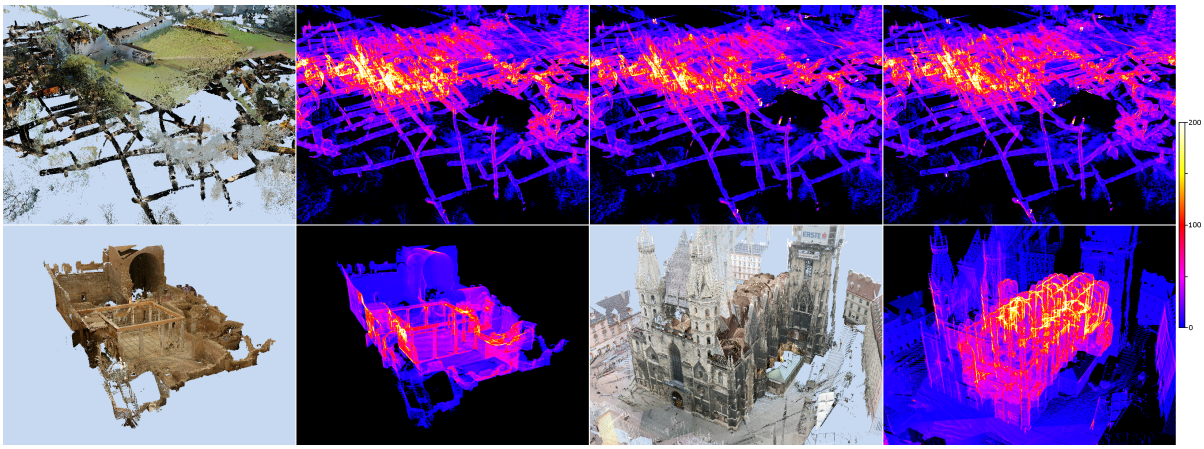

Figure 4: In the top row the Domitilla model together with some overdraw heat-map visualizations is shown, from 2nd image left to right this are the MNO, the MWO with 30k leaf nodes and priority-based traversal, and the MWO with 30k leaf nodes and depth first traversal. In the bottom row the Hanghaus and Stephansdom models are shown, and the MNO is used for heat-map visualization. All traversals use a maximum projected grid cell size of 1 pixel and are rendered with 1 pixel per point.

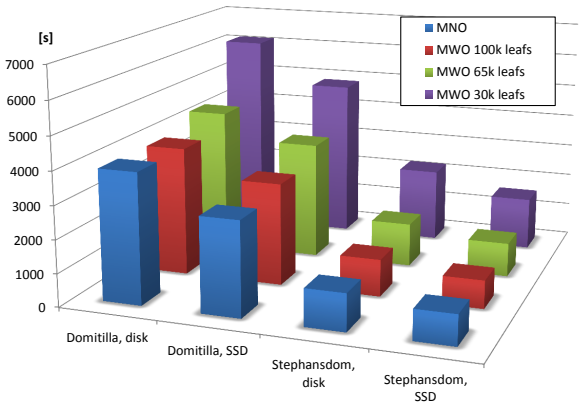

Figure 5: Buildup timings for the Domitilla and Stephansdom models on disk and on SSD.

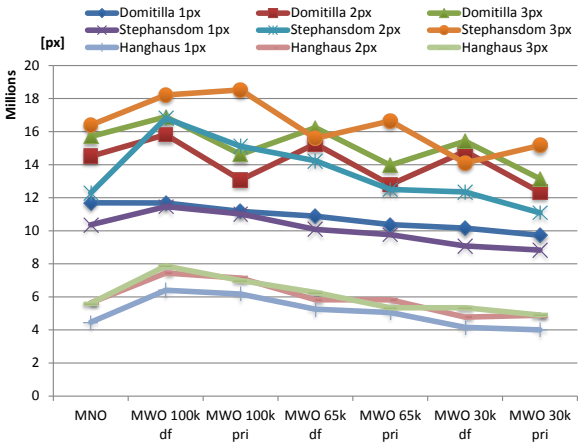

Figure 6: Total number of pixels drawn at different settings for the projected grid cell size.

8 RESULTS

We implemented the MNO and the MWO data structures in our point rendering and editing framework, to be able to compare both data structures directly. Our test computer has an Intel Core i7-2600 quad core CPU, 16 GB RAM, two 10.000 rpm hard disks in a RAID 0, and 4 solid state drives (SSDs) in a RAID 0. All operations with the data structures use an LRU cache of 100 million points. Rendering is done in OpenGL.

For the tests we use 3 point data sets, the Domitilla catacomb with $1.92 * 10^9$ points (courtesy of the FWF START project "The Domitilla-Catacomb in Rome."), the Stephansdom with $460 * 10^6$ points, and the Hanghaus with $14.7 * 10^6$ points. In Figure 4 the data sets (together with some overdraw heat maps) are shown. From each data set we build 4 models, one MNO with minimum 1000 points per node, and 3 MWOs with 30k, 65k, and 100k maximum points per leaf node. The space requirements for an MWO increase with smaller leaf node size, e.g., for the Domitilla model the MWO with 30k leaf nodes is about 2.2 times larger than the original data. This leads to higher build up times as can be seen in Figure 5. Build up times for all Hanghaus models are 30 seconds or less. Furthermore we measured the total number of fragments written to the 800x600 pixels viewport in 21 different settings per model. For each model we tested a maximum projected grid cell size of 1, 2, and 3 pixels, and the models were then rendered with screen aligned OpenGL points of the according size. For each grid cell size we rendered the MNO model and the 3 MWO variants. Each MWO was rendered once with a priority-based traversal, and once with a depth first traversal. All renderings were done from one viewpoint per model. The results are given in Figure 6, and except for the Stephansdom

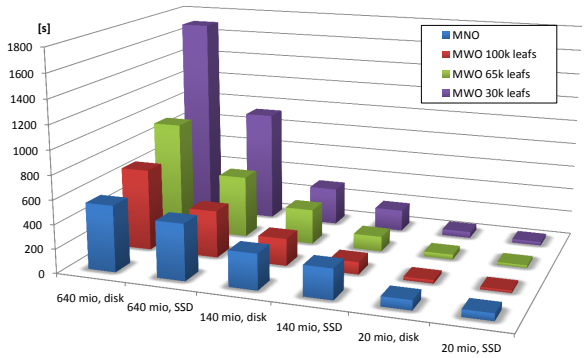

Figure 7: Timings for deleting differently sized selections from the Domitilla model.

MWOs rendered with 3 pixels and the Hanghaus 30k MWO rendered with 2 pixels projected grid size, the priority-based traversal causes less overdraw. Editing operations were tested by deleting different sized selections from the largest point cloud, the Domitilla model. The results in Figure 7 show that the usage of an SSD and also a smaller selection size do not speed up the operation for the MNO as for the MWO models. This is caused by a more expensive pull up operation in the MNO, which uses nodes in the LRU cache. The time overhead for updating the interior nodes after deleting is about 1% for the largest, 4% for the medium, and about 10% for the smallest selection.

9 CONCLUSION

We have presented a complexity analysis for the MNO and the MWO, two data structures which can be used for editing and rendering huge point clouds. The data structures show similar complexities for build up and editing operations. The memory requirements and thus processing times for the MWO are dependent on the leaf node size, while the memory requirements for the MNO are the same as for the original data set. The MNO is better suited for large editing operations, as less nodes have to be processed. The MWO (especially with 30k leaf nodes) is better suited for rendering, as often less points are required to render a point cloud from a certain viewpoint, resulting in higher frame rates. We also introduced a lightweight extension to the points deleting algorithm for the MWO, which allows to use efficient node selection strategies during rendering, leading to less pixel overdraw.

10 REFERENCES

[And94] Andres, E. Discrete circles, rings, and spheres. Computers and Graphics, 18(5):695-706, 1994.

[Bay08] Bayer, R. B-tree and UB-tree. Scholarpedia, 3(11):7742, 2008.

[Ben75] Bentley, J.L. Multidimensional Binary Search Trees Used for Associative Searching. Communications of the ACM, 18(9):509-517, September 1975.

[BM72] Bayer, R., and McCreight, E.M. Organization and Maintenance of Large Ordered Indexes. Acta Informatica, 1:173-189, 1972.

[CLRS09] Cormen, T.H., Leiserson, C.E., Rivest, R.L., and Stein, C. Introduction to Algorithms, Third Edition. The MIT Press, 3rd edition, 2009.

[Den68] Denning, P.J. The Working Set Model for Program Behavior. Communications of the ACM (CACM), 11(5):323-333, May 1968.

[FAR13] FARO Technologies, Inc. http://www.faro.com/, 2013.

[FB74] Finkel, R.A., and Bentley, J.L. Quad Trees A Data Structure for Retrieval on Composite Keys. Acta Informatica, 4:1-9, 1974.

[GM04] Gobbetti, E., and Marton, F. Layered point clouds. Computers & Graphics, 28(6):815-826, 2004.

[Mic03] Microsoft. How NTFS Works. Microsoft TechNet Webpage, 2003.

[PSL05] Pajarola, R., Sainz, M., and Lario, R. Xsplat: External memory multiresolution point visualization. In Proceedings IASTED Invernational Conference on Visualization, Imaging and Image Processing, pages 628-633, 2005.

[Rie13] Riegl Laser Measurement Systems. http://www.riegl.com/, 2013.

[Sam06] Samet, H. Foundations of Multidimensional and Metric Data Structures. Elsevier, 2006.

[SW11] Scheiblauer, C., and Wimmer, M. Out-of-Core Selection and Editing of Huge Point Clouds. Computers & Graphics, 35(2):342-351, April 2011.

[Vit08] Vitter, J.S. Algorithms and Data Structures for External Memory. Foundations and Trends in Theoretical Computer Science, 2(4):305-474, 2008.

[WBB+07] Wand, M., Berner, A., Bokeloh, M., Fleck, A., Hoffmann, M., Jenke, P., Maier, B., Staneker, D., and Schilling, A. Interactive Editing of Large Point Clouds. In Symposium on Point Based Graphics, pages 37-45, Prague, Czech Republic, 2007. Eurographics Association.

[WBB+09] Wand, M., Berner, A., Bokeloh, M., Fleck, A., Hoffmann, M., Jenke, P., Maier, B., Staneker, D., and Parys, R. Xgrt extensible graphics toolkit, 2009.

[WN73] Wong, C.K., and Nievergelt, J. Upper Bounds for the Total Path Length of Binary Trees. J. ACM, 20(1):1-6, January 1973.