Out-of-Core Selection and Editing of Huge Point Clouds

Claus Scheiblauer*, Michael Wimmer

Institute of Computer Graphics and Algorithms, Favoritenstrasse 9-11 / E186, A-1040 Wien, Austria

Abstract

In this paper we present an out-of-core editing system for point clouds, which allows selecting and modifying arbitrary parts of a huge point cloud interactively. We can use the selections to segment the point cloud, to delete points, or to render a preview of the model without the points in the selections. Furthermore we allow for inserting points into an already existing point cloud. All operations are conducted on a rendering optimized data structure that uses the raw point cloud from a laser scanner, and no additionally created points are needed for an efficient level-of-detail (LOD) representation using this data structure. We also propose an algorithm to alleviate the artifacts when rendering a point cloud with large discrepancies in density in different areas by estimating point sizes heuristically. These estimated point sizes can be used to mimic a closed surface on the raw point cloud, also when the point cloud is composed of several raw laser scans.

Keywords: Point-based rendering, Viewing algorithms, Graphics data structures and data types

1. Introduction

In recent years the amount of data produced by laser scanners has increased tremendously. Some examples for the application of laser scanning are in the field of cultural heritage to document places and buildings of historical interest, in the field of geodesy to measure the earth's topography, or in the production industry to document the status of a large industrial plant in order to support change management. In all these areas the physical size of the scanned objects is ever increasing. Another reason for the tremendous amount of data produced is that the possible number of samples per scan is surpassing 1 Billion (10^9) for the latest generation of laser scanners [1, 2].

The point clouds resulting from laser scans are usually used to inspect the scanned objects. But it is also important to be able to *interact* with the point clouds to extract more information from the model. For example in the field of building archeology, the construction phases of a building can be recovered by interactively segmenting the model. Some areas of the model that are to be investigated are possibly occluded by other areas of the model, so the uninteresting parts should be hidden for the time of the investigation by selecting them and making them invisible.

We propose a point-based editing system that can be used to segment a point-based model into arbitrary regions, even if the model does not fit completely into main memory. We provide a tool that makes working with point clouds feasible and intuitive. One important aspect is that the data structure we use for the point-based models does not require additionally created points compared to the original dataset. We also propose a point size heuristic that can be quickly evaluated and is therefore well suited for a point cloud that is often changed by editing operations. The benefit for users is that they can interact with the point cloud directly and do not have to await a costly and error-prone conversion to a mesh-based structure, and that they can work on the whole point cloud at once, always having all points that possibly belong to a region available for manipuation. In particular, the contributions of this paper are:

- A method for interactive selection of arbitrary parts of a point cloud using a so-called Selection Octree.
- Inserting points into and deleting points from a rendering optimized data structure.
- A point size heuristic for estimating the splat size radii in a point-based model to mimic a closed surface in a sufficiently densely sampled area.

The Selection Octree is a tool to handle different visualizations of the same model without actually having to permanently modify the large model. For example, an archaeologist might create several selections including different parts of a model representing different creation times. He can then easily switch between visualizations of different time periods.

The insertion and deletion operations for the rendering optimized data structure are required for editing operations on the point model, which is a compound of all available laser scans. Instead of building one point cloud, we could use the individual scans separately, but for the exploration and interaction with data from laser scans it is more efficient to work on a single point model, because this allows recognizing the sampling densities of individual areas of the model and rendering with a sampling density adapted to the current view. This would not be possible when using the scans separately. The information which point belongs to which scan position, which might be

*Tel: +43 (1) 58801-18646, Fax: +43 (1) 58801-18698
Email address: scheiblauer@cg.tuwien.ac.at (Claus Scheiblauer)

useful for some processing tasks, is not lost in our system, as we can store arbitrary attributes per point, e.g., the scan position index.

2. Previous Work

A general overview of point-based rendering and processing is given in [3], where in-core and out-of-core rendering algorithms are presented, but no out-of-core editing algorithm.

For point clouds that can be held in main memory, Weyrich et. al. [4] presented a plugin for the Pointshop 3D point manipulation system [5] that allows manipulating the raw data from a range scanner. In the Digital Michelangelo project [6], range images (e.g., images with depth) were sub-sampled to build range image pyramids to deal with the tremendous amount of data during rendering and editing the point clouds. Scanalyze [7] is a system for aligning large triangle meshes out-of-core. It uses level-of-detail (LOD) representations of the meshes to be able to merge and align different meshes. The meshes are generated from point clouds from scanned objects. In the Active Points [8] system a color editing tool is used, because they claim that the appearance of a model is more important than the actual geometry. Boubekeur and Schlick [9] point sample a large polygon-based model, so that the geometry of the large model is approximated in-core and can be interactively textured. None of these systems is suitable for editing large unstructured point clouds.

In [10], the potential of out-of-core point rendering systems is described in the context of cultural heritage. The paper shows the interaction possibilities with huge point clouds by describing a number of use cases where different tools are used to support the archaeological work. However, that paper focuses on the user side and does not describe algorithmic implementation strategies. These are the focus of the current paper, which shows the algorithmic contributions that lead to the tools and use cases described in [10].

Wand et. al [11] also describe an editing system for large point clouds. They build up an octree and store a quantization grid at the inner nodes of the octree, where each grid cell stores an arbitrary point that is inserted into the data structure to represent all points at lower levels in the hierarchy. This is similar to the idea of Inner Octrees in [12] where each octree node stores an octree itself, the so called Inner Octree. The points inserted in one octree node are stored at the lowest Inner Octree level. For editing, Wand et al. use a weight at each point in an inner node grid cell, which is then updated during editing operations. A disadvantage of their implemented system is that they reserve disk space for about 3 times of the size of the actual point data.

Other data structures that are well suited for fast rendering are difficult to adapt for editing. Dachsbacher et al. [13] developed a hardware-accelerated LOD point rendering algorithm for models that can be stored in-core. Pajarola et al. [14] extended the work of [13] for an out-of-core setting. They had to introduce a global indexing scheme on the point data. Gobetti and Marton [15] showed a point rendering system for very large models that uses only the points of the original point cloud and is therefore memory efficient. The point-based models have to be preprocessed though, because they assume a homogeneous point density for rendering. Kalaigh and Varshney [16] analyze point clouds statistically using a clustering-based hierarchical principal component analysis (PCA) of the point geometry, which results in a compact statistical representation from which the point cloud can be rendered directly. All three aforementioned approaches for rendering huge point clouds are not well suited to delete points or insert new points, because then the LOD hierarchies would have to be repaired in a costly operation.

In [17], a number of external memory sorting algorithms are described, as well as algorithms for external memory problems like range searches and directory lookups. Dynamic data structures for inserting and deleting elements are created by combining efficient smaller static data structures and updating them.

Spatial databases provide a way to organize multi-dimensional data in a way that database queries can be processed efficiently [18]. Typical queries are, e.g., point queries, region queries, or nearest neighbors. The data structures used for spatial databases are related to the problem of organizing a point cloud for efficient rendering, but they do not consider the problem of rendering LODs for a reduced version of the whole point cloud. The Quadtree data structure (or Octree in 3D) is also used for spatial databases, and it is the base data structure for the Nested Octree [12].

Volumeshop [19] uses a selection volume that stores real values in the range [0,1], where 0 means “not selected” and 1 means “fully selected”. The selection volume is evaluated at every frame to modify the transfer function in the selected areas. This way selected areas can be emphasized by a different color or opacity value. Selection volumes are extended by Bürger et. al [20] to support sub-voxel editing operations by using a high resolution volume that is an upsampled representation of the original volume. Editing operations in this volume are performed directly on the GPU.

3. Modifiable Nested Octree

We choose the Nested Octree [12] as base data structure for manipulations, because we want to visualize huge point models in real time. Due to this requirement, the chosen data structure has to fulfill certain criteria. It should support out-of-core rendering and real-time rendering, so it has to support LODs, because when rendering a huge point cloud not all points can be shown in every frame. We would also like to have a compact representation of the point cloud, i.e., a representation that does not need additionally created points for rendering or editing. Furthermore the build up process should be deterministic, so we can easily update the LOD structure when inserting or deleting points.

There are also drawbacks when using this rendering optimized data structure. Due to the nested representation of the LODs, a neighbor query has to search nodes at all hierarchy levels, because the potential neighbors of a point can be at any hierarchy level. The most efficient way to do a neighbor query is to first determine a maximum radius in which neighbors shall be searched, and then collect all points from all hierarchy levels

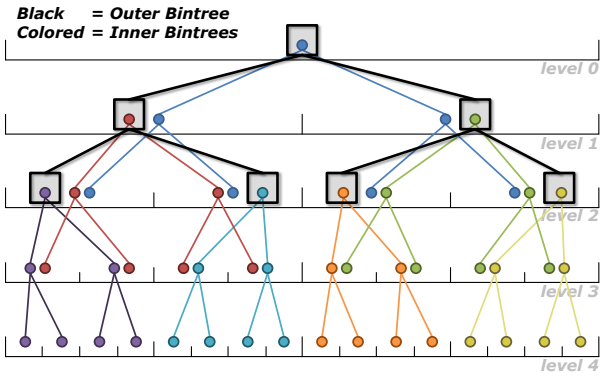

Figure 1: The hierarchy of a completely filled Nested Bintree of depth 5, where Inner Bintrees and the Outer Bintree have a depth of 3. The root nodes of the Inner Bintrees are held in the nodes of the Outer Bintree. The points reside at all levels of the Inner Bintrees.

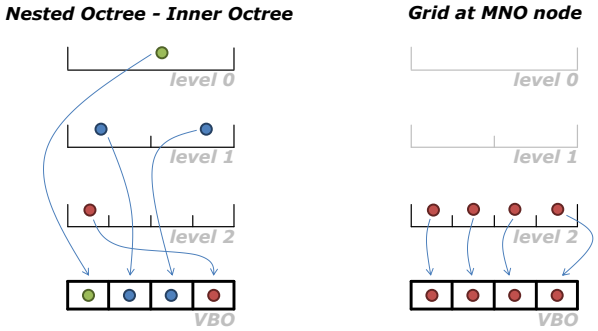

Figure 2: The difference between a Nested Octree and a Modifiable Nested Octree is in the storage of the points. The Nested Octree stores the points in its nodes in Inner Octrees (left), while the MNO stores the points in grids (right). Inserting and deleting points are costly operations for an Inner Octree, because the octree has to be rebuilt. An Inner Octree also has to maintain a certain order when mapping the points to a VBO [12], while a grid can map the points sequentially to a VBO.

that are within this radius. This is not necessary in an octree where the points are only stored at the leaf nodes.

Figure 1 shows an example of the original Nested Octree data structure (a Nested Bintree for representation). In this example the Inner Bintrees are fully occupied, i.e., all levels of the Inner Bintrees hold points. For rendering, the tree structure of an Inner Bintree is linearized [12], and the points are stored in Vertex Buffer Objects (VBOs) and sent to the graphics card. When rendering a frame, the bounding boxes of the nodes are projected to screen space, and all nodes whose projected size is larger than a predefined threshold are chosen for rendering, and all points of the chosen nodes are rendered.

We made some changes to the original Nested Octree data structure so we can use it for efficiently inserting and deleting points and call it Modifiable Nested Octree (MNO). We basically abandon the idea of maintaining a hierarchy in the Inner Octrees, and replace them by regular grids. This corresponds to removing the upper levels of the Inner Octrees and only retaining the leaf nodes (Figure 2). An additional advantage is that the overdraw generated by points in the upper levels of fully occupied Inner Octrees is removed.

With these changes the LOD hierarchy now works similar to the LOD hierarchy of the Layered Point Clouds [15]. They subsample the original point cloud at each node (using a kd-tree) to get the points for this node. The advantage of using a regular grid at each node of the hierarchy, and inserting the points there, is that the point model becomes editable.

3.1. Inserting Points

Inserting points into a new MNO is now very simple. All points start to search at the root node of the MNO for an empty cell in the root node's regular grid. The regular grid's cell into which a point falls is determined by calculating the grid index for the point by subdividing the root node's bounding box to the resolution of the regular grid, and inserting the point into the grid. If a point falls into an empty grid cell it is stored there. If it falls into a grid cell that has already been occupied, it is calculated into which child node of the root node the point falls, by calculating the child node index from the point coordinates and the center of the root node's bounding box. Inserting the point into the child node is now recursive, and the point is checked against nodes further down the hierarchy until it finds an empty grid cell.

An optimization to this algorithm is to hold back a certain number of points before they are inserted into a child node that does not yet exist. This way, a new child node will always hold at least this number of points, as nodes that hold too few points are inefficient when loading them from hard disk during rendering.

In our implementation the grid is represented as a hash table, and we calculate the keys for the hash table to check if a position in the grid is still free. For best performance we use a speed-optimized hash map implementation from Google [21].

3.2. Build-Up Process

The build-up process starts by reading point data files and converting them into a binary data stream to get a uniform representation of all points, independent of the data format they are originally stored in. When reading the point data files, the bounding box of the points contained in the files is calculated. The binary data stream stores the points intertwined, i.e., a point with all its attributes is stored as a continuous array of bytes. Examples for attributes are position, color, normal, or scan position index.

Note that many out-of-core algorithms rely on a first pass to create locally coherent work items that can be treated in-core (e.g., distribution sort [22]). However, in the case of point clouds it is not easily possible to find borders for these buckets, and therefore a subdivision so that each bucket can be treated in-core, because the density of the points can be very inhomogeneous.

After creating the binary data stream, the points are then read in batches of several 1000 points from the data stream and inserted according to the insertion algorithm (Section 3.1). The basic insertion algorithm only works for models that completely fit into main memory. To make it work for larger models, we manage the nodes of the already built up MNO in a least-recently-used (LRU) cache [11]. The points of the nodes

are swapped in and out of memory according to the requirements of the insertion algorithm. The LRU swapping algorithm uses the main memory as cache for the nodes of the MNO, so if a point model fits completely in main memory the build up algorithm is as fast as the basic insertion algorithm.

The points of one node are stored in one file on disk, and all files are stored in one directory. The access to the files is managed by the file system, e.g., on our test machine we use the NTFS file system which is used by Windows. NTFS indexes the files in a directory with a B-tree [23], which minimizes the number of disk accesses to find a file.

This out-of-core algorithm works especially well if there is spatial coherence among points, which is often the case in scanned models. The original build-up algorithm [12] swaps points to disk every three levels they are filtered down the octree hierarchy. This is reasonable when the amount of main memory is no more than 512MB, but today the main memory on high-end PCs is already 4GB or more, which can be used as cache during build up. The number of points that can be held in-core depends on the attributes per point and on the size of the LRU cache. Assuming that one point has the attributes position and color represented in 16 bytes and that the LRU cache has a size of 10GB, then a point cloud with some 625M points can be built in-core.

3.3. Adding Points

When adding new points to an existing model, and all new points fit into the bounding box of the model, then the new points are simply inserted into the existing model, i.e., the new points are converted to a binary stream and then inserted. If this is not the case, and new points are outside the current bounding box, the bounding box of the model has to be enlarged to enclose all the new points before inserting the new points. Furthermore the LOD hierarchy of the MNO has to be updated to keep it consistent.

To find the new bounding box of the model we add superior octree levels to the bounding box of the root node of the existing MNO. This is done iteratively. From the 8 possible superior bounding boxes for a new octree level we choose the one bounding box that minimizes the distance (center to center) between the bounding box of the new superior octree level and the bounding box of the new points. Superior octree levels are added this way until the root node of the octree encloses the new points completely.

After this, the new points are inserted into the MNO, which now has one or more empty nodes above the old root node. Here we use again the LRU cache to manage swapping the points of those nodes in and out of memory. When all new points have been inserted, it can still be the case that there is no connection between the old root node and the new root node in the LOD hierarchy as illustrated in Figure 3. To find points that fill the LOD hierarchy accordingly, we insert the points of the old root node (actually a copy of them) into all newly created octree nodes, to find the grid cells in the new octree nodes that need to be filled. We are only interested in the grid cells, so we delete the copies of the points of the old root node from these grid cells again. If a point falls into an already occupied grid cell, nothing has to be done. If a point falls in an empty grid cell, the bounding box of the grid cell is then used to find a leaf node of the MNO from which points can be used to fill this grid cell. When an appropriate leaf node is found, one point is inserted into the new grid cell and deleted from the leaf node. If a leaf node becomes empty it is deleted from the MNO.

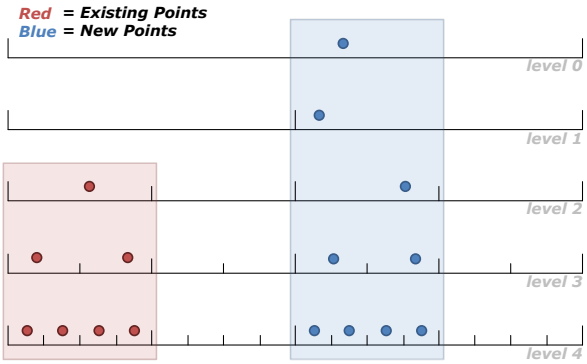

Figure 3: Schematic illustration of the LOD hierarchy of an MNO after inserting new points. The red points are part of the old MNO (red bounding box), and the blue points are newly inserted. The bounding box of the blue points does not intersect with the old MNO, so in level 1 the LOD hierarchy from the new root node to the old root node is broken (there should be a point in the left part of level 1).

Note that an octree node can only be determined unambiguously as long as the bounding box of the new grid cell is smaller than or equal to the bounding box of the octree node. If the bounding box is larger than the bounding box of an octree node, then it does not matter which child to follow to find a leaf node. In this case, when there are more than one child at an octree node, we randomly select one child node and continue searching for a leaf node from there.

3.4. Deleting Points

For deleting points, we also use the LRU cache to manage the points in the nodes of the MNO. When deleting points from the MNO, holes in the LOD hierarchy can appear. The holes in the LOD hierarchy are filled again by pulling up points from a leaf node that has a bounding box that is inside the bounding box of the cell that holds the point to be deleted. If no point can be found to replace the deleted point, then this cell remains empty. The deleting algorithm traverses the MNO in postorder, so leaf nodes are processed first, and a leaf node is simply removed from the octree hierarchy when all points in this node have been deleted.

3.5. Rendering

When rendering an MNO, the rendering algorithm traverses the MNO from the root node to find the nodes that are large enough when projected to screen space and therefore shall be rendered. For this purpose the bounding box of the frontmost grid cell of an MNO node, with respect to the current view point, is projected to screen space, and if its size is above a predefined threshold (e.g., 1 pixel) the node is inserted into a priority queue that is ordered by the importance of the nodes.

The most important nodes are the ones (partly) within the view frustum, close to the viewpoint, and in the center of the screen. Rendering starts with the most important node. When rendering a node, we render all points of the node, and we render the points of that node at the same size. If not using a fixed point size for rendering, we use the point size heuristic described below (Section 5).

As geometric primitive we use screen-aligned quadratic splats, as they are the fastest to render. Nodes that are chosen for rendering but are currently not in memory are swapped to memory in a separate thread. This is done by sending a request to the reading thread to read the file containing the points of the node. The reading thread then passes back a memory buffer with the loaded points, which can then be moved directly to the GPU for rendering. Points that are not used for rendering any more are dropped from the GPU and stored in a LRU cache in main memory. Only when points fall out of the main memory cache they have to be loaded again from disk if needed.

We extended the rendering capabilities of the original Nested Octree implementation to allow more than one point cloud to be rendered out-of-core at the same time. When several point clouds are loaded into the scene, they use the same node budget, i.e., the same priority queue, and the importance function designating which node to render is evaluated globally on all available and visible nodes from all point clouds.

4. Selection Octrees

Selecting points in an out-of-core setting is not trivial, since not all points are available in memory at any time. If selection happens on individual points of a surface, moving closer to the surface could make a more detailed LOD level appear, whose points would not be selected although they belong to the selected part of the surface. To overcome this problem we have chosen to define the selection not on individual points, but on the space that the selection encompasses. For this, we introduce the so-called Selection Octree as a selection primitive. The Selection Octree is a separate data structure with the same root node as the MNO of the point model. All points whose position is within the Selection Octree are marked as selected. Points that are loaded after defining the selection just have to check whether they are inside the Selection Octree or not, and are then marked accordingly. The points on disk are not changed during selection. The selection remains valid when the user changes the viewpoint.

Contrary to selection volumes used in volume rendering [19, 20], we do not use a regular grid as base data structure for the selection volume, as the points of raw laser scans are irregularly distributed. Due to this we use an octree, as it is built hierarchically and therefore uses less memory in areas with low point density.

We store the Selection Octrees in-core, and the number of Selection Octrees that can be handled at the same time is per se not limited, as we simply store the Selection Octrees in a list. The only limit that might occur is the amount of available main memory. A typical Selection Octree uses about 1MB, where one Selection Octree node uses 12 Bits for its state and 4 pointers for managing the Selection Octree hierarchy (the children of a node are stored in a singly-linked list). The Selection Octrees can be used for different selections at the same time, and points can be part of different Selection Octrees at the same time.

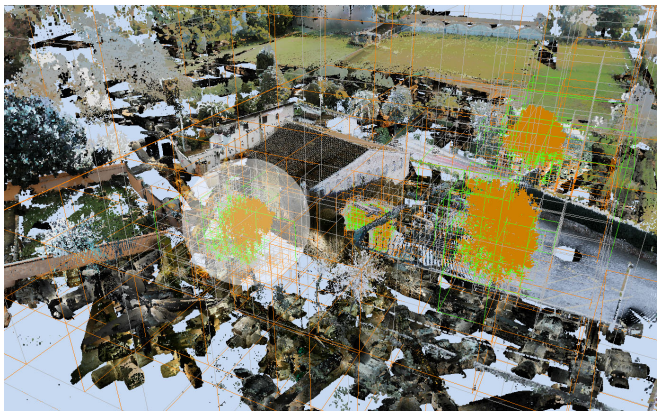

Figure 4: The nodes of the Selection Octree that encompass the selection are shown in green. All selected areas are managed by the same Selection Octree, the nodes connecting the areas shown in orange. Empty leaf nodes are shown in grey. The semi-transparent sphere is the selection sphere used to de-/select points.

4.1. Building the Selection Octree

The Selection Octree is built from the points that are selected by the user. For user interaction we provide a volumetric selection brush [4, 20] which follows the surface of the model by reading the Z-Buffer. If there is no valid entry in the Z-Buffer for a position, i.e., the Z-Buffer is at negative "infinity" for this position, than the z-value of the last valid position is used. The user can interact with the brush by moving the mouse. On pressing the left mouse button, all points that are loaded to the graphics card and are inside the selection brush are marked by a flag in the alpha channel of the color of the point (we do not need the alpha channel for rendering) by continuously intersecting the volume of the selection brush with the nodes of the MNO. On releasing the mouse button, all points (marked and not marked) from the nodes that were intersected by the volumetric brush since pressing the mouse button are copied into an array and inserted into the Selection Octree.

All these points are inserted at once, and the points are filtered down according to their position. Creation of new child nodes is continued until either only one point is left at a node, or if at one node only marked or unmarked points exist. Note that we do not store any points in the Selection Octree permanently, the points are only used to build the octree hierarchy. The points that are not marked have also to be inserted into the Selection Octree, as they are necessary to determine the exact border of the Selection Octree.

Building the Selection Octree is quite fast (see Table 5). The build up time is longest for very large selections (up to 2.5 seconds), while for smaller selections it is unnoticeable by the user. The Selection Octree can be updated in the same way as it is built, i.e., inserting new points into the already existing octree

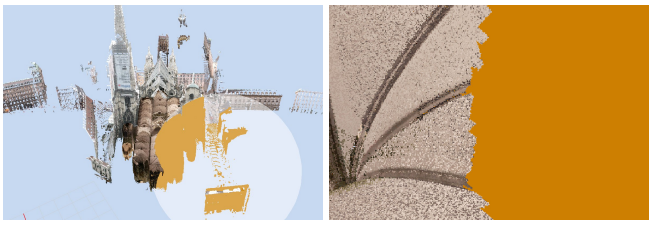

Figure 5: Selecting points in some distance to the model results in aliasing for the border of the selection when zooming into the model, as can be seen in the right image.

hierarchy. For further usage Selection Octrees can also be saved to disk and loaded again later.

4.2. Building the Deselection Octree

Building the Deselection Octree is done in a similar way as building the Selection Octree. The user again selects points with the selection brush, but now he can only mark points that have already been selected. On pressing the right mouse button all selected points within the brush are marked by a flag that they will be deselected. On releasing the mouse button again all points of all MNO nodes that were intersected by selection brush are inserted into the Deselection Octree. When the points are collected, they are copied from the graphics card into an array. After they have been copied (the copied points retain their alpha channel flags) the points on the graphics card reset all flags in the alpha channel, as they are now unselected points.

The Deselection Octree is then built to represent the volume described by the deselected points, i.e., still selected points and ordinary points are not included in the Deselection Octree. Afterwards it is substracted from the Selection Octree, which lasts only some milliseconds. The now reduced Selection Octree is optimized to merge nodes with the same properties to larger nodes. The Deselection Octree is then deleted from memory.

4.3. Properties of the Selection Octree

All points that are marked as selected can be rendered as invisible by projecting them to infinity inside the vertex shader (setting the point's w-coordinate to 0), enabling a preview of the model without the selected points.

The maximum resolution of the Selection Octree is bound by the size of the currently rendered octree nodes. If the user is selecting points at a coarse LOD, the resolution will be low, e.g., the bounds of the selection will exhibit aliasing artifacts. This can be seen when approaching the model and therefore rendering it at a finer LOD as shown in Figure 5. There does not seem to be an obvious solution to this problem, as an automatic refinement of the Selection Octree, without inserting new points, might not be in coherence with the points of the model. Another option would be to define a hull around the Selection Octree, and when loading new points from disk insert all points to the Selection Octree that are in the space between the outer border of the Selection Octree and the border of the hull around the Selection Octree, but we did not test this. On the other hand, the Selection Octree represents exactly the detail that was available to the user when he created the selection, so the result is not unexpected.

5. Point Size Heuristic

One of the most critical aspects when interacting with a huge point cloud is that the point sizes are chosen in a way that holes are avoided in closed surfaces. This is important to allow selection operations, because otherwise connected surfaces would not be visible anymore when viewed from a closer distance. On the other hand, we do not make any assumptions about sampling rates or the existence of surfaces in the point cloud, since outdoor scans often have wildly varying sampling rates in the model. We chose a compromise and present a heuristics that is based on the octree depth of points and a local density estimate. Figure 6 gives an example of the varying sampling densities when rendering a point-based model and the effect of the weighted point size.

For points in nodes where no children are rendered (leaf nodes or nodes whose children are too small to be rendered when projected to screen space), we simply choose the side length of the inner octree grid cell as the world space point size. The point size is then projected to screen space at a distance of 1 from the viewpoint, and so determines the point splat size for the points of an MNO node in screen space. If the distance of a point is other than 1 from the viewpoint, the point size is divided by the distance of the point in the vertex shader.

The problem is choosing the point size for points that are stored in the upper nodes of the MNO. The point sizes should fit the surrounding points which are stored in nodes at lower levels. If we use the grid cell size as we do for leaf nodes, points would become huge in upper hierarchy levels and hide all surrounding points. Note that we do not make any assumptions on the distance between neighboring points.

Our heuristic uses a *weighted point size* that reflects the number of points in the children of a rendered node as a local density estimate. Only nodes that are rendered contribute to the point size estimation. The point size is influenced by two weights, the number of points in an MNO node and the so-called level weight.

5.1. Virtual Depth

The first weight is the number of points in a node. The more points in a node, the higher its influence on the point size. This will ensure that inner nodes that have only few points stored in their children will render with a larger point size.

The point size is calculated for each node that is rendered in the current frame. The point size heuristic calculates a (potentially non-integer) *virtual depth* of the points in a node, so that they are rendered as if they were at another level in the hierarchy. The virtual depth vd_{node} for one node is calculated as

$$vd_{node} = \frac{\sum_i n_i * w(d_i) * d_i + n_{node} * w(d_{node}) * d_{node}}{\sum_i n_i * w(d_i) + n_{node} * w(d_{node})}$$

Figure 6: A view inside the basilica of the model of the Domitilla Catacomb. On the left side the point cloud is rendered with a fixed point size of one. The sparsely sampled areas are partly due to the different sampling densities, partly due to the LOD algorithm which prefers to load nodes in the center of the screen. On the right side the point cloud is rendered with the weighted point size. The level weight is calculated from the exact distribution of the points in the MNO.

where n_i is the number of points at node i, d_i is the depth of node i, and $w(d_i)$ is the level weight for node i. The index i goes over all rendered direct and indirect children of a node. Intuitively, this means if a node has many points in a child of depth d, then the point size of that node will be similar to the point size at level d in order to let the points in the inner node blend into the points of the "dense" children.

When only using this weight, the points of the upper levels in the hierarchy are still rendered too large compared to the points of the lower levels. This can be accounted for with the level weight.

5.2. *Level Weight*

The second weight is the number of points per level. This gives the lower levels more weight and pulls the point sizes towards these levels, as most points are held there. Figure 7 shows an example distribution of the points in the 20 levels (counted from 0 to 19) of the Domitilla Catacomb model (see Figure 9). The three graphs show different calculation methods for the distribution of the points. Other models we tested have similar distributions of the points.

The simple level weight just counts all points at each level. But since the MNO also stores points in the inner nodes of the octree, the points of the inner nodes contribute to the wrong hierarchy level, as they should be counted at the levels of the leaf nodes they would fall into.

We have done this with two different methods. First by estimating the number of points in the leaf nodes, and second by actually intersecting the points of the inner nodes with the bounding boxes of the leaf nodes.

In the first method we estimate the number of the points in the leaf nodes by assuming a uniform sampling distribution. Then we calculate how many points of an inner node would fall into a leaf node under this assumption, and add this number to the number of points in the leaf node. We do this for all leaf nodes. Note that an inner node can also be partially a leaf node, if not all children are occupied. In this case the bounding boxes of the empty child nodes act as leaf nodes. With this estimation

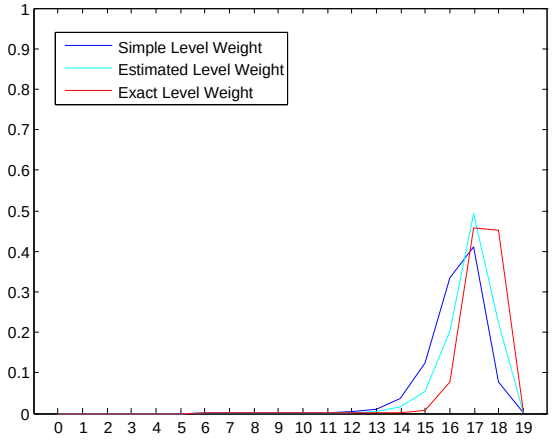

Figure 7: Comparison of the different level weight calculation methods for the model of the Domitilla Catacomb. With the exact calculation method some 90 percent of all points are in the 2nd and 3rd lowest levels combined.

the distribution of the points is already shifted about one level (see Figure 7, cyan graph).

In the second method we exactly calculate the points that are within the bounding boxes of the leaf nodes by simply intersecting the leaf nodes' bounding boxes with the points of their direct and indirect parent nodes and counting the total number of points that fall into each leaf node. As can be seen in Figure 7 (red graph) the resulting distribution of points per level is further shifted down in the hierarchy levels, resulting in a smaller point size for rendered points of the upper levels.

Figure 8 shows the blending of the points of the different hierarchy levels. Points of level 9 are rendered smaller than points of level 8. Combining all levels results in a homogeneous representation of the zoomed out area.

6. Results

To test the performance of the build up algorithm, we use four different point models (see Figure 9). The test computer

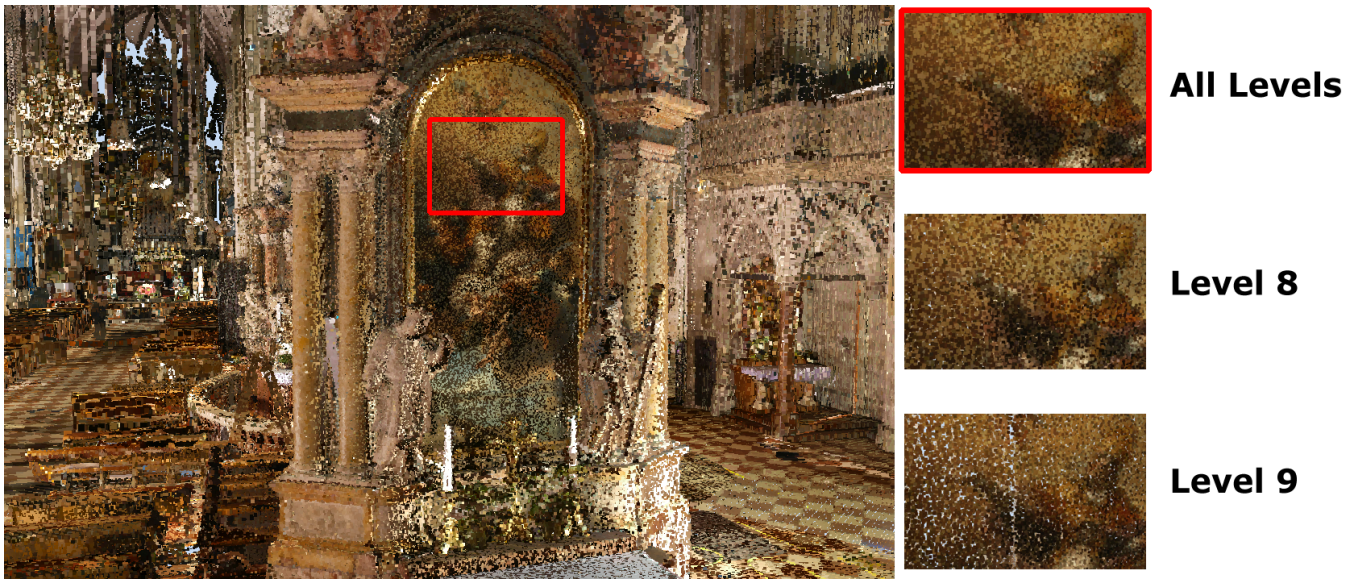

Figure 8: The points of the different levels are blended to make a closed surface of the point model. In the closeup points from hierarchy level 8 and 9 are shown. All levels combined result in the closed surface for the point model.

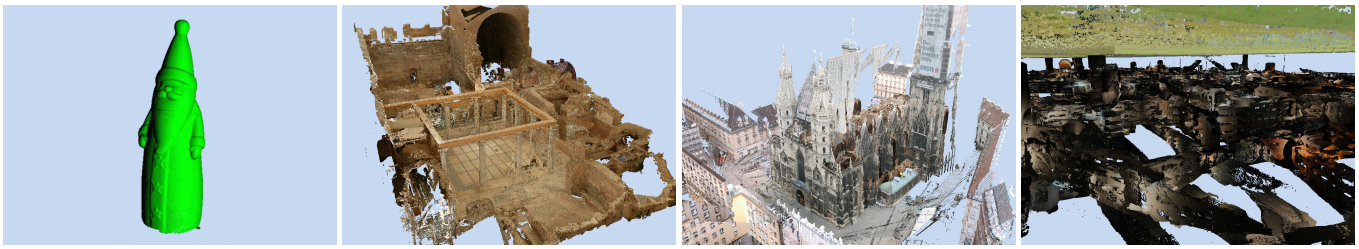

Figure 9: The scans used for benchmarking the build-up algorithm and the editing operations. Left: Santa Claus model, consisting of about 16M points. It was scanned with a scan-arm. Center left: Model of the “Hanghaus” in Ephesos, consisting of about 15M points. Center right: The Stephansdom in Vienna, consisting of 193 single scans and about 460M points. Right: A view of the subterranean Domitilla catacomb in Rome, hovering below the turf. This model consists of 1828 single scans and about 1.92 Billion points.

Model	Attributes	AttribsSize	Time	Build Time	Size on disk	Throughput
Santa Claus	p,c	16B	3m 25s	13.6s	255.0MB	18.7MB/s
Santa Claus	p,c,n	28B	4m 58s	13.9s	445.6MB	32.0MB/s
Ephesos	p,c	16B	34s	11.6s	235.9MB	20.3MB/s
Ephesos	p,c,n	28B	35s	12.1s	412.4MB	34.0MB/s

Table 1: The build-up times and sizes for the resulting models for the Santa Claus point cloud (16M points) and the Ephesos house (15M points) built with the MNO build up algorithm. The AttribsSize column shows the number of bytes that are used for all attributes of one point. The Time column shows the complete processing time, including reading the data files and converting them to a binary data stream. The Build Time column shows the build-up time alone. The models are built in-core. The size of the resulting models is larger than the added up size of the points alone due to padding at the disk sectors and the header file that holds the information about the octree hierarchy. The Attributes column shows which attributes were used for build up (p = position, c = color, n = normal).

Model	Attributes	Obj per leaf	Nodes mem	Time	Build Time	Size on disk
Santa Claus	p,c	131,072	512	11m 43s	1m 41s	778.6MB
Santa Claus	p,c,n	65,536	3000	15m 21s	1m 51s	1,321.3MB
Santa Claus	p,c,n	131,072	512	15m 07s	1m 44s	1,116.9MB
Santa Claus	p,c,n	500,000	128	14m 47s	1m 28s	989.6MB

Table 2: The build up times and sizes for the resulting models for the Santa Claus point cloud (16M points) built with the XGRT system. Again, the ”Time” column shows the complete processing time, including reading the data files. The ”Build Time” column shows the build up time alone. ”Obj per leaf” means the maximum number of points per leaf node and ”Nodes mem” means the maximum number of nodes in memory. The ”Attributes” column shows which attributes were used for build up (p = position, c = color, n = normal).

Model	Attributes	AttribsSize	Time	Build Time	Size on disk	Throughput
Stephansdom	p,c	16B	1h 45m 07s	1h 38m 31s	7,390.1MB	1.25MB/s
Stephansdom	p,c,n	28B	3h 02m 31s	2h 49m 50s	12,922.0MB	1.27MB/s
Stephansdom	p,c,n,r,i,s	44B	3h 58m 18s	3h 51m 44s	20,299.6MB	1.46MB/s
Domitilla	p,c	16B	3h 53m 21s	2h 22m 29s	30,812.4MB	3.60MB/s
Domitilla	p,c,n	28B	4h 28m 36s	3h 01m 31s	53,874.6MB	4.94MB/s
Domitilla	p,c,n,r,i,s	44B	5h 59m 51s	4h 27m 32s	84,622.8MB	5.27MB/s

Table 3: Here the AttribsSize column shows the number of bytes that are used for all attributes of one point. The Time column again shows the complete processing time, including reading the data files and converting them to a binary data stream. For the Domitilla model (1.92 Billion points) the point data files are the original files from the laser scanner which need long to process. The Build Time column shows the build-up time alone. Size on disk shows the size of the resulting model for out-of-core build up using the MNO build up algorithm. Throughput shows the number of bytes per second that are processed during the build up stage. The Stephansdom model (460M points) has a lower throughput then the Domitilla model, which is due to the large areas that the single scans cover in the Stephansdom model. Table 4 shows that this results in more LRU cache misses. Finally the Attributes column shows which attributes were used for build up (p = position, c = color, n = normal, r = point radius, i = point index, s = scanposition index).

Model	Attributes	LRU size	MNO nodes	Nodes to mem	Nodes to mem %
Stephansdom	p,c	1,287.7MB	54,474	228,495	412%
Stephansdom	p,c,n	1,288.0MB	54,474	281,623	507%
Stephansdom	p,c,n,r,i,s	1,288.0MB	54,474	315,267	568%
Domitilla	p,c	1,288.0MB	261,598	173,222	66%
Domitilla	p,c,n	1,288.0MB	261,598	245,959	94%
Domitilla	p,c,n,r,i,s	1,288.0MB	261,598	320,124	122%

Table 4: The LRU cache sizes in megabytes, the number of nodes in the final MNO, the number of nodes swapped back to memory during build up, and the percentage of the swapped back nodes with respect to the nodes of the final MNO. The single scans in the Stephansdom model contribute to larger areas of the model, which results in more cache misses in the build up LRU cache. Due to this more nodes have to be swapped out of and back to memory. The Attributes column shows which attributes were used for build up (p = position, c = color, n = normal, r = point radius, i = point index, s = scanposition index).

has an Intel Core2 Quad Q6600 CPU with 2.4 GHz, 4GB RAM, a RAID0+1 with 4 SATA hard disks running at 10.000 RPM, and a GeForce 8800GTX with 768MB VRAM.

We build up the models with different per-point attributes to evaluate the performance characteristics of the build-up algorithm. The possible attributes are position, color, normal, point radius, point index, and scan position index. In total these attributes use 44 bytes per point. The tests are performed with all or a subset of these attributes. When reading the point data from the source files and converting them to a binary stream, we can choose which attributes the points in the binary stream should have. If an attribute is not available in the source data, it is added for each point and filled with a default value. In the resulting binary stream all attributes that the user has chosen are available, and from this binary stream the MNO is then built. The attributes position, color, and normal are so called "meta-attributes" in our system, where we can choose which representation they should have on disk and in memory. When loading points from disk, during rendering the meta-attributes can be converted to a specific representation that is suitable for rendering, or they can be passed to the graphics card directly.

We use the Santa Claus model (16M points) and the Ephesos model (15M points) to test the in-core performance of our build-up algorithm, as they fit completely in the main memory of the computer. The models have position, color, and normal as attributes (using 28 bytes per point). During build up, the size of the LRU cache is at most 30% of the complete main memory. In Table 1, we show the results for the in-core build-up algorithm. The data throughput is much higher when normals are also used with the attributes, so the traversal of the octree structure is the main limiting factor when inserting points in-core. We can insert 1.21M points per second when using only position and color as attributes and 1.17M points per second when using a normal per point as well (these numbers are averaged over both models). We compare our build up algorithm with the algorithm presented by Wand et al. [11]. We use their XGRT point rendering system, which is provided as open source. The timings given for the build up in XGRT are from the console output of XGRT. Table 2 shows the parameters used for the data structure, the timings for the complete processing, and the timings for the in-core build up alone. The size of the resulting model on disk seems to be dependent on the maximum number of objects that are allowed in the leaf nodes, and it is 2 to 3 times larger than the size of the original scan data.

For testing the out-of-core performance we use the Stephansdom model (460M points), whose normals were estimated in a pre-process, and the Domitilla model (1.92 Billion points). Table 3 shows a comparison of the build-up times of these two models with different point attributes. The Stephansdom models show a similar build-up performance for the bytes per second that are processed (from 1.25MB to 1.46MB per second), but the Domitilla models on the other hand show a large discrepancy. It seems that the build up of the Stephansdom models is limited by the number of disk accesses, but the build up of the Domitilla models is limited by the access time to the files on disk. Another observation is that the Domitilla Catacomb models can be built with a higher build-up performance than the Stephansdom models. The reason for this seems to be that the single scans of the Domitilla model contribute to more limited areas of the model, due to the narrow hallways. This

# Points	SelOct Build Time
313,010	0.08s
609,897	0.18s
2,521,201	0.79s
12,048,415	2.18s
13,248,293	2.52s

Table 5: Times needed for building a Selection Octree.

# Points	Delete Time
8660	1.85s
2,316,123	1m 19.87s
16,622,231	2m 02.66s
627,325,824	11m 39.05s

Table 6: Times needed for deleting different areas from the Domitilla model.

results in less LRU cache misses and less disk accesses during build up (see Table 4).

We tested the build-up times of the Selection Octree (see Table 5). Inserting points can be done at a rate of 5.26M points per second. The Deselection Octree is built in the same time, the subtraction from the Selection Octree typically lasts 20 to 80 milliseconds.

Timings for deleting points from the Domitilla model are shown in Table 6. Deleting becomes more efficient for larger areas, because then nodes that are completely inside the selection can be deleted immediately.

7. Conclusion

We have presented a system for interactively editing huge point clouds which is efficient both in memory usage as well as interaction performance. Our system uses significantly less memory and offers higher build-up speed than previous work. It is built on the paradigm of representing selections as separate objects, so-called Selection Octrees. These can be built efficiently and allow both manipulations on as well as segmenting the point cloud. We also present an enhanced rendering optimized out-of-core data structure for storing huge point clouds that does not use additionally created points compared to the original point cloud, but still offers efficient operators for inserting and deleting points. The system is supported by a point-size heuristic that displays a closed surface on a point based model if the model is sufficiently densely sampled, and offers both better rendering quality and the use of depth-based brushes for creating selections.

8. Acknowledgments

This work was funded by the Austrian Research Promotion Agency (FFG) through the FIT-IT project “Terapoints”, and by the Austrian Science Fund (FWF) through the START project “The Domitilla-Catacomb in Rome. Archaeology, Architecture and Art History of a Late Roman Cemetery”. That project is possible by commission and with the help of the Pontificia Commissione di Archeologia Sacra/Roma.

References

[1] Riegl Laser Measurement Systems, http://www.riegl.com/, 2010.

[2] Zoller Fröhlich GmbH, http://www.zf-laser.com/, 2010.

[3] M. Gross, H. Pfister, Point-Based Graphics (The Morgan Kaufmann Series in Computer Graphics), Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007.

[4] T. Weyrich, M. Pauly, R. Keiser, S. Heinzle, S. Scandella, M. Gross, Post-processing of Scanned 3D Surface Data, in: M. Gross, H. Pfister, M. Alexa, S. Rusinkiewicz (Eds.), Symposium on Point-Based Graphics, Zürich, Switzerland, pp. 85–94.

[5] M. Zwicker, M. Pauly, O. Knoll, M. H. Gross, Pointshop 3D an Interactive System for Point-based Surface Editing, ACM Trans. Graph 21 (2002) 322–329.

[6] M. Levoy, The digital Michelangelo project, in: 3DIM99, pp. 2–11.

[7] M. Ginzton, K. Pulli, Scanalyze a system for aligning and merging range data, 2002.

[8] H. Xu, B. Chen, ActivePoints Acquisition, Processing and Navigation of Large Outdoor Environments, Technical Report, Department of Computer Science and Engineering University of Minnesota at Twin Cities, 2002.

[9] T. Boubekeur, C. Schlick, Interactive out-of-core texturing with point-sampled textures, in: M. Botsch, B. Chen, M. Pauly, M. Zwicker (Eds.), Symposium on Point-Based Graphics, Eurographics Association, Boston, Massachusetts, USA, 2006, pp. 67–73.

[10] C. Scheiblauer, N. Zimmermann, M. Wimmer, Interactive Domitilla Catacomb Exploration, in: K. Debattista, C. Perlingieri, D. Pitzalis, S. Spina (Eds.), VAST09: The 10th International Symposium on Virtual Reality, Archaeology and Intelligent Cultural Heritage, Eurographics Association, St. Julians, Malta, 2009, pp. 65–72.

[11] M. Wand, A. Berner, M. Bokeloh, A. Fleck, M. Hoffmann, P. Jenke, B. Maier, D. Staneker, A. Schilling, Interactive Editing of Large Point Clouds, in: Symposium on Point Based Graphics, Eurographics Association, Prague, Czech Republic, 2007, pp. 37–45.

[12] M. Wimmer, C. Scheiblauer, Instant Points, in: Proceedings Symposium on Point-Based Graphics 2006, Eurographics, Eurographics Association, 2006, pp. 129–136.

[13] C. Dachsbacher, C. Vogelgsang, M. Stamminger, Sequential point trees, in: Proceedings of ACM SIGGRAPH 2003, volume 22(3) of *ACM Transactions on Graphics*, ACM Press, 2003, pp. 657–662.

[14] R. Pajarola, M. Sainz, R. Lario, Xsplat: External memory multiresolution point visualization, in: Proceedings IASTED Invernational Conference on Visualization, Imaging and Image Processing, pp. 628–633.

[15] E. Gobbetti, F. Marton, Layered point clouds a simple and efficient multiresolution structure for distributing and rendering gigantic point-sampled models, Computers & Graphics 28 (2004) 815–826.

[16] A. Kalaiah, A. Varshney, Statistical geometry representation for efficient transmission and rendering, ACM Transactions on Graphics 24 (2005) 348–373.

[17] J. S. Vitter, External memory algorithms and data structures: dealing with massive data, ACM Computing Surveys 33 (2001) 209–271.

[18] V. Gandhi, J. M. Kang, S. Shekhar, Spatial Databases, Technical Report, Department of Computer Science, University of Minnesota, Minneapolis, USA, 2007.

[19] S. Bruckner, M. E. Gröller, Volumeshop: An interactive system for direct volume illustration, in: IEEE Visualization, IEEE Computer Society, 2005, p. 85.

[20] K. Bürger, J. Krüger, R. Westermann, Direct volume editing, IEEE Transactions on Visualization and Computer Graphics 14 (2008) 1388–1395.

[21] Google, http://code.google.com/p/google-sparsehash/, 2010.

[22] D. E. Knuth, The Art of Computer Programming, Volume 3: (2nd ed.) Sorting and Searching, Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1998.

[23] Microsoft, How NTFS Works, http://technet.microsoft.com/en-us/library/cc781134(v=WS.10).aspx, Microsoft TechNet Webpage (2003).