ISPRS Journal of Photogrammetry and Remote Sensing 76 (2013) 76–88

Contents lists available at SciVerse ScienceDirect

ISPRS Journal of Photogrammetry and Remote Sensing

journal homepage: www.elsevier.com/locate/isprsjprs

One billion points in the cloud – an octree for efficient processing of 3D laser scans

Jan Elseberg, Dorit Borrmann, Andreas Nüchter*

Jacobs University Bremen gGmbH, Automation Group, School of Engineering and Science, Campus Ring 1, 28759 Bremen, Germany

A R T I C L E I N F O

Article history:
Available online 4 December 2012

Keywords:
Octree
Tree data structure
Data compression
Frustum culling
Ray casting
RANSAC
Nearest neighbor search

A B S T R A C T

Automated 3-dimensional modeling pipelines include 3D scanning, registration, data abstraction, and visualization. All steps in such a pipeline require the processing of a massive amount of 3D data, due to the ability of current 3D scanners to sample environments with a high density. The increasing sampling rates make it easy to acquire Billions of spatial data points. This paper presents algorithms and data structures for handling these data. We propose an efficient octree to store and compress 3D data without loss of precision. We demonstrate its usage for an exchange file format, fast point cloud visualization, sped-up 3D scan matching, and shape detection algorithms. We evaluate our approach using typical terrestrial laser scans.

1. Introduction

Laser range scanning provides an efficient way to actively acquire accurate and dense 3D point clouds of object surfaces or environments. 3D point clouds provide a basis for rapid modeling in industrial automation, architecture, agriculture, construction or maintenance of tunnels and mines, facility management, and urban and regional planning. Modern terrestrial and kinematic laser scan systems acquire data at an astonishing rate. For example, the Faro Focus^{3D} delivers up to 976,000 3D points per second and the Velodyne HDL-64E yields 1.8 million range measurements per second. Kinematic laser scan systems often use multiple line scanners and also produce a huge amount of 3D data points. A common way to deal with the data is to process only a small subset of it. While this is an acceptable way of handling the data for some applications it calls into question why so many measurements were acquired in the first place.

In this paper we describe a spatial data structure called an octree. Using innovations from the computer graphics community, we develop an octree implementation with advantageous properties. First, we prefer a data structure that stores the raw point cloud over a highly approximative voxel representation. While the latter one is perfectly justifiable for some use cases, it is incompatible with tasks that require exact point measurements like data visualization and scan matching. Second, the octree ought to be fast, i.e., access, insert and delete operations must be in $O(\log n)$, where n is the number of stored points. Last and most important, the data structure has to be memory efficient. We present an easy to implement octree encoding that fulfills these requirements and is capable of storing 1 billion points in 8 GB of memory. It is also possible to employ disk caching with the data structure, i.e., larger data sets can be streamed from a mass storage device when processing it.

For achieving our goal to process 1 billion points in main memory on a standard computer, we have implemented an efficient data structure for 3D point clouds. In addition to the octree, this paper presents several algorithms exploiting the properties of our data structure. These algorithms include a novel and efficient RANSAC implementation for shape detection and a novel nearest neighbor search algorithm for ICP-based scan matching. Furthermore, we show that our memory efficient encoding is also universal enough to allow for other uses such as fast visualization. All presented algorithms are available under the GPL license in the *3DTK – The 3D Toolkit* and can be downloaded from http://threedtk.de. The toolkit contains a small show application that uses the frustum culling and processes 1 billion points while still enabling the user to navigte smoothly through the point cloud. In case more points have to be rendered than the graphics card is able to process in a given time, the point density is reduced dynamically, by sending only a fraction of points to the graphics card. Thus, the user is able to inspect large 3D scenes. Fig. 1 shows a scene with dynamic point reduction. The scene was recorded on the campus of the Jacobs University using a Riegl VZ-400 scanner. The data set contains 876,820,018 3D points and is easily processable with our octree implementation. The data set and windows executables are given at http://plum.eecs.jacobs-university.de/download/isprs2011/JacobsUni.zip. To view the 3D point cloud using the described frustum culling on a 64-Bit Windows system, please unzip it and change into the directory `win_x64`. Afterwards, call the program as `show -s 31 -e 80—loadOct../dat/`, which loads the provided 50 terrestrial 3D scans (starting with scan

* Corresponding author.
E-mail addresses: j.elseberg@jacobs-university.de (J. Elseberg), d.borrmann@jacobs-university.de (D. Borrmann), a.nuechter@jacobs-university.de (A. Nüchter).

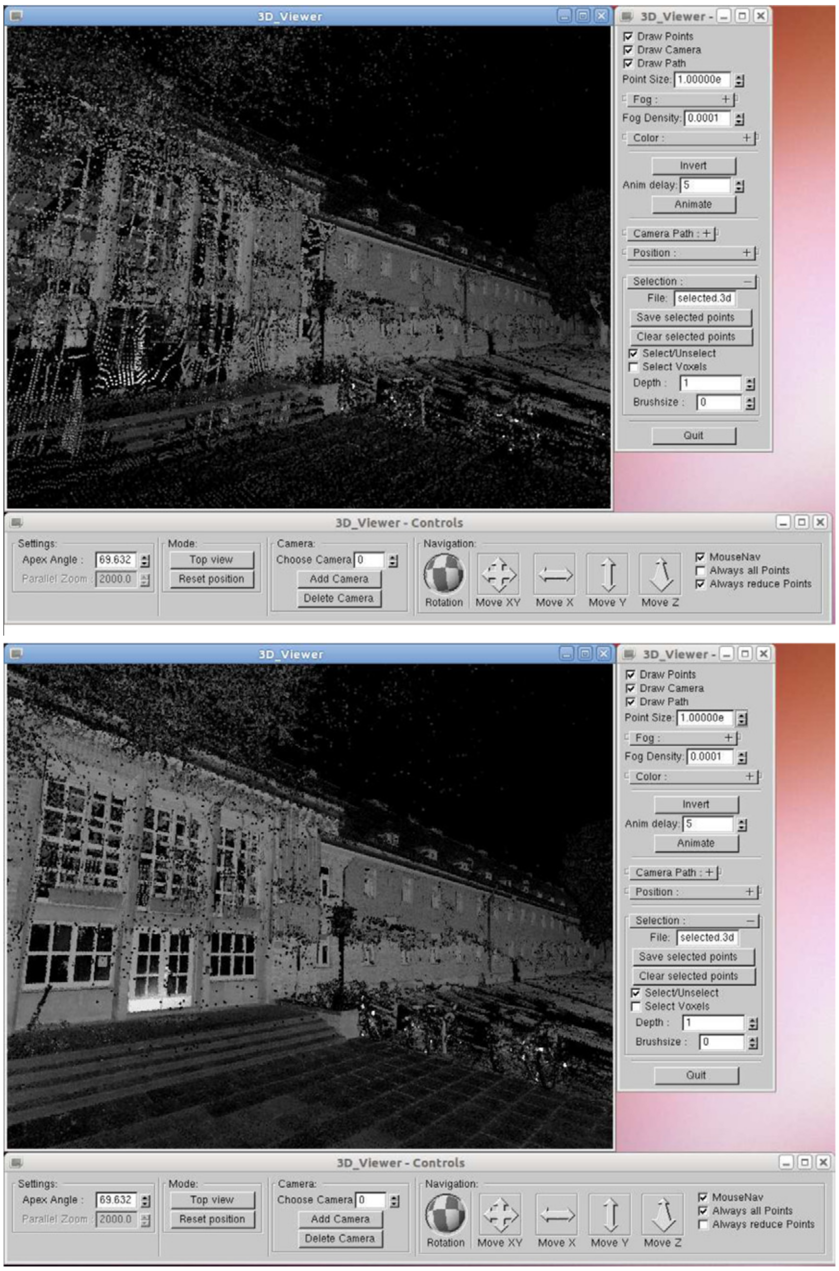

Fig. 1. Dynamic point reduction in *3DTK – The 3D Toolkit* (http://threedtk.de). Top: reduced point cloud to obtain a frate rate of 20 fps. Bottom: All 3D points.

number 31 ending with scan number 80) and shows them in a small OpenGL-based viewing program.

2. Octrees for storing 3D point clouds

An octree is a tree data structure that is used for indexing 3-dimensional data. It is the generalization of binary trees and quadtrees, which store one and 2-dimensional data respectively. Each node in an octree represents the volume formed by a rectangular cuboid, often, also in our implementation, simplified to an axis aligned cube. This is analogous to representing a line segment, or rectangle for binary and quadtrees. Consequently an octree node has up to eight children, each corresponding to one octant of the overlying cube/node. A node having no children usually implies that the corresponding volume can be uniformly represented, i.e., no further subdivision is necessary to disambiguate. This convention is not completely applicable when storing points, which are technically dimensionless, i.e., there is no volume associated with them. When storing a point cloud, we must therefore define a stopping rule for occupied volumes. We define both a maximal depth and a minimal number of points as a stopping criteria. If either the maximal depth is exceeded or the number of points is below the given limit leaf nodes, instead of inner nodes, are generated. Defining a maximal depth is equivalent to defining the smallest possible leaf size, also referred to as the voxel size. A list of points is stored in each occupied leaf. By applying two simple criteria we avoid building a perfect octree, i.e., an octree, where all leaves are at the same depth and all other nodes have exactly eight children. First and foremost the uniformity criteria above is applied to volumes without points, such that subdivision is not necessary in empty nodes. In fact, we only create child nodes for octree volumes that contain points. All nodes without children are interpreted as empty space. Second, we do not subdivide a volume further that contains only a single point. Laser scanners sample only the surface

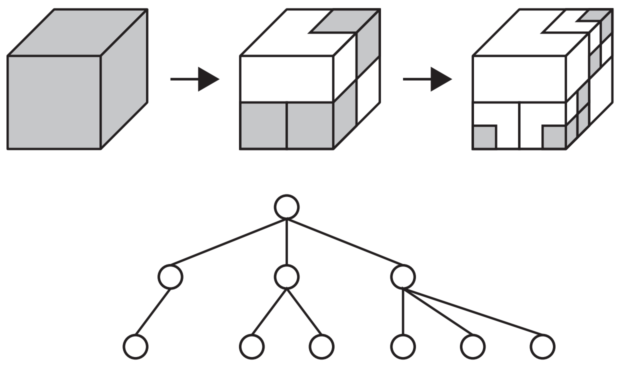

Fig. 2. Left: Spatial subdivisions of an octree up to level 3. Occupied leaf nodes are shaded grey. Right: The corresponding tree structure of the sparse datastructure.

of objects, and usually provide only a single distance measurement per angle pair. This leads to a 3-dimensional point cloud that is not fully volumetric. Consequently, most space is not occupied, and therefore most octree nodes will only have few children. The octree data structure is therefore ideally suited to store and retrieve 3D laser scanner data efficiently. A recursive refinement of an octree is illustrated in Fig. 2.

2.1. Memory efficient encoding of an octree

Implementations of an octree often do not prioritize memory efficiency. A relevant property of an octree is that uniform subvolumes are represented as single nodes that do not subdivide further. For applications, where the octree stores a pure voxelmap, this property compresses the data to a large degree. Of course, this is in comparison to a simple 3D grid stored linearly in memory. Thus arises the term sparse voxel octree in the computer graphics community, where an octree structure is used to efficiently access otherwise large amounts of data.

On the opposite side are the serialized pointer-free encodings. These have the highest potential for memory efficiency, since they do not need to store the actual octree structure. One such encoding is given by Girardeau-Montaut et al. (2005) and is implemented in the point cloud comparison software CloudCompare (Girardeau-Montaut, 2011). They employ the Morton order to store only the leaf level of an octree at 16 bytes per leaf. The Morton order or Z-order is an ordering of, in this case, 3-dimensional data (Morton, 1966). This approach has several drawbacks. Traversing the Octree is not possible in the classical sense. Instead binary search algorithms have to be applied when looking for a certain voxel.

Serialization is a very useful tool when storing the data for later use or when communicating over channels with limited bandwidth. Schnabel and Klein (2006) combine a serialized octree with arithmetic coding to provide superior point cloud compression. While this approach excels at compressing point clouds, it becomes infeasible to efficiently process the data for most applications, i.e. finding a point in this encoding is in $O(n)$ as the tree has to be traversed from the root in breadth-first order until the point has been found. A more detailed discussion of octree related work is given in Section 5.

We opt for a pointer based octree since it allows for several operations and applications which are not feasible with the above designs. Modification, i.e. adding or deleting points from a serialized octree involves shifting the entire region before or after the position, where the modification takes place. Wand et al. (2007) use a pointer based octree for interactive editing of large point cloud data. They also employ out-of-core storage to deal with data sets that do not fit into main memory. This technique, too, is out of the question for serialized octrees, since it cannot as easily be predicted which data chunk needs to be fetched. Even though our efficient octree design is capable of the above applications, this paper focusses on the algorithmic questions of visualization and processing.

Many implementations store redundant information in each octree node. In computer graphics, for example, neighbor pointers as well as a parent pointer are used to facilitate extremely fast ray tracing at the cost of additional memory. Another encoding, that redundantly stores the position and size in each node is given in Fig. 3, where `center` and `size` store the position and size of the node while `child` is an array of pointers to the eight children. This allows to stop subdivision for empty nodes having to divide the volume into equal subvolumes, thus potentially reducing the number of nodes required for storage. On a standard 64-bit architecture, each node requires 100 bytes of memory. Although the implementation is somewhat naive, it nicely illustrates the possible memory gains that can be achieved.

We create an efficient octree implementation that is free of redundancies and is nevertheless capable of fast access operations. Our implementation allows for access operations in $O(\log n)$. Add and delete operations are also in $O(\log n)$. In the worst case long blocks of memory will have to be allocated or deallocated in a leaf node.

Most information about inner nodes of an octree is computed when recursing through the structure. The depth of a node is calculated as the depth of its parent plus one. Due to the properties of the regular octant subdivisions the size of a node is a function of its depth. Similarly the position of a node is computed by displacing the position of the parent node by half of the cell size in the appropriate direction. The direction follows trivially from the size of the node and its position in the parent's list of children. Only the root node must therefore store some information about the octree as a whole, i.e., the position and size of the spanned volume. In the same manner parent pointers may be computed, or rather remembered, by pushing visited parents onto a stack. It is even possible to compute neighbor pointers by a fast indexing scheme. Section 4.2 explains this process. Unfortunately, as this operation requires backtracking along the parent stack it is necessarily less efficient than computing the other properties. For the sake of memory efficiency, we omit any information that is computable by traversing the tree. However, referring to the baseline implementation in Fig. 3 the removed redundancy accounts for only 24 of 100 bytes. 64 remain for child pointers and 12 bytes for the point storage. We downsized this by moving the information about whether or not a node exists from the node itself into its parent. We add a single byte, where each bit corresponds to one octant of the node. This allows us to remove the constraint to always store eight children, so that only those child nodes need to exist that contain valuable information in the first place. We can therefore remove 56 further bytes by storing only a single pointer to all children. Adding another byte, where each bit signals whether the

```
struct OcTree {
  float center[3];
  float size[3];
  OcTree *child[8];
  int nr_points;
  float **points;
};
```

Fig. 3. Definition of an octree with redundant information and eight pointers to child nodes. The size of this node is 100 bytes.

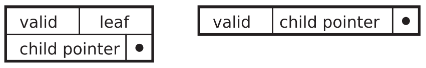

Fig. 4. The two proposed encodings of an octree node optimized for memory efficiency. The child pointer as the relative pointer is the largest part of an octree node, but varies in size to accomodate different systems. In our implementation for 64 bits systems, it is 48 bits. `valid` and `leaf` are 8 bits large. Left: The proposed encoding with separate bit fields for valid and leaf. An entire node is thus contained in only 8 bytes of memory. Right: Alternative solution resulting in a constant depth octree.

corresponding octant is a leaf allows the removal of the point information that is unnecessary in inner nodes. Two alternative encodings that result from these considerations are presented in Fig. 4.

Our encoding consists of three parts. The child pointer is the largest part of each node and is implemented as a relative pointer to the first child. All other valid children are arranged linearly in memory as shown in Fig. 4. The pointer can vary in size for different systems. For 64 bits architectures we have chosen 6 bytes. As this is sufficient to address a total of 256 terabyte, there is no need for an additional bit signaling for a far pointer as proposed by Samuli and Tero (2010). Using a far pointer flag would require more sophisticated memory management, but it would enable one to reduce the size of the child pointer to two or fewer bytes. There is a second, easier way to reduce the number of bytes required for the child pointer, if we are willing to sacrifice $O(\log n)$ add and delete operations. In this case, the octree can be stored in a linear array in breadth first order, with each child pointer simply indexing the array. However, further discussion will focus on using this 6 bytes pointer implementation.

`valid` and `leaf` are each a single byte large, 1 bit for each subvolume. `valid` bits signal whether the corresponding octant is present, while `leaf` bits signal whether the corresponding child is a leaf node. This encoding is somewhat redundant, as non-valid children cannot be leaf nodes. There are only $3^8 = 6561$ combinations possible. These could be compressed and represented with only 13 instead of 16 bits. Due to concerns about the runtime efficiency and the relatively minor reduction of the memory requirements, we decided against such a compression. It is possible to remove the leaf byte, by enforcing a constant depth of the octree (cf. Fig. 4, right). Similar to the other properties of a node, that are determined recursively, the depth of a node is always well defined. While the size of an octree node is reduced by enforcing a constant depth this procedure is certain to increase the number of nodes. Let n be the number of bytes used for a node implementation without a leaf byte, the percentaged increase in the number of nodes is limited by $\frac{1}{n}$ to achieve increased memory efficiency. For our implementation, this amounts to approximately 15% increase that is allowed at most. We found that point clouds acquired by laser scanners are so sparse that they require a smaller percentage only for shallow trees. Increasing the resolution and thereby the depth also increases the number of nodes significantly, which in turn increases the size in memory exponentially as demonstrated in Table 1. Even for the most compact dataset balance is obtained for a voxelsize of 10 cm. For resolutions above that, the difference in required memory space is so small as to be insignificant for most applications. Counterintuitively, then, the 8 bytes encoding is the more robust and more memory efficient choice for data that is as sparse or more sparse than laser scanner point clouds.

Our implementation stores points in the leaf nodes, thus they need to be represented differently from inner nodes. In Fig. 5, leaf nodes are pointers to arrays of points. The first entry is always the total number of points, then sequentially the information for each point, i.e., the coordinates and additional attributes such as reflectance. In that representation, leaf nodes would be n bytes larger than inner nodes, where n is the number of bytes used to encode the number of points. In our case it is more than sufficient to reserve $n = 4$ bytes for this purpose. Such a point list representation is then already more memory efficient than the usual `float**`, as it cuts down on a pointer.

2.2. Octree based compression of 3D point clouds

Our octree encoding drastically decreases the overhead for obtaining the data structure itself (cf. Table 1). As opposed to the reference implementation the memory for the point cloud exceeds now the overhead (cf. Fig. 6). Therefore, we seek to compress the point list as well. For a simple technical reason we like to store each point coordinate using only 2 bytes. Two bytes are exactly the resolution at which most laser scanners measure additional point attributes, such as reflectance, deviation. To store floating point coordinates in only 2 bytes without significant loss of precision, we use each bit of the 2 bytes coordinate as $\frac{s}{2^{16}}$ increments to the lower left front corner of the rectangular cuboid of the leaf node, where s is the side length of the cuboid. This is similar to color quantization as used for example by Gervauts and Purgathofer (1990).

Data of terrestrial laser scanners represented as 4 bytes floating point value has a precision of approx. 100 μm (100 μm) at the maximal distance of 500 m. At a smaller distance, e.g., at 1.5 m , the precision increases to 1 μm. To achieve the same 1 μm

Table 1
Memory requirements for the sparse Kurt3D point cloud using several octree implementations and different resolutions. The first column gives the size of the smallest leaf in the tree, i.e., half of its side length. The second and third columns give the number of nodes and leaf nodes for the 8 bytes per node implementation. The real size in memory as well as the average construction time follows in the next two columns. After those, the number of nodes for the 7 bytes implementation is given as well as the percentual increase and the real size in memory. Construction time has been omitted, as it is virtually equally to the 8 bytes implementation. The memory requirements and construction time for the implementation with 100 bytes per node are listed in the last two columns. (cf. Fig. 3).

	8 bytes				7 bytes			100 bytes	
Leaf size (cm)	# Nodes	# Leaves	Mem. size	Constr. time (ms)	# Nodes	Incr. %	Mem. size	Mem. size	Constr. time (ms)
876	1	8	104 B	4.0	1	0	103 B	954 B	1.1
438	9	26	384 B	4.6	9	0	375 B	3.71 kB	2.0
219	34	77	1.19 kB	5.2	35	2.9	1.16 kB	11.87 kB	2.6
109.5	103	202	3.24 kB	6.0	112	8.7	3.2 kB	33.28 kB	3.2
54.76	282	505	8.31 kB	6.7	314	11.3	8.25 kB	86.81 kB	3.7
27.38	698	1352	21.8 kB	7.6	819	17.3	21.95 kB	230.12 kB	4.2
13.69	1777	3688	58.47 kB	8.6	2171	22.1	59.45 kB	621.05 kB	4.8
6.846	4121	8840	139.04 kB	10.2	5859	42.1	147.09 kB	1.55 MB	5.8
3.423	9327	19,064	303.38 kB	11.9	14,699	57.5	331.66 kB	3.57 MB	6.4
1.711	17,219	34,697	554.11 kB	13.4	33,763	96	652.7 kB	7.25 MB	7.0
0.855	27,668	52,836	855.37 kB	14.3	68,460	147	1.11 MB	12.85 MB	7.9
0.427	37,351	68,253	1.11 MB	14.3	121,296	224	1.66 MB	20.09 MB	8.7

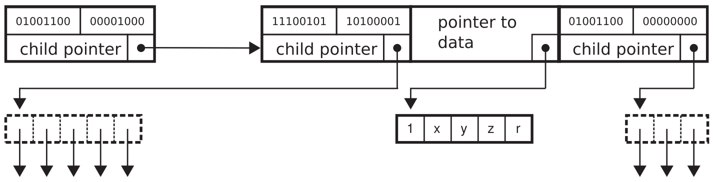

Fig. 5. An example of a simple octree as it is stored using the proposed encoding. The node in the upper left has three valid children, one of which is a leaf. Therefore, the child pointer only points to three nodes stored consecutively in memory. The leaf node in this example is a simple pointer to an array which stores both the number of points and the points with all their attributes.

precision the smallest volume in the octree must have a side length of 6.5 cm. Assuming a desired precision of 10 μm, which is still two orders of magnitude smaller than typical specified measurement precisions, the largest node is allowed to have a side length of 65 cm. At this voxel size the octree overhead is minimal even for large scans.

2.3. Efficient construction of an Octree

When constructing octrees from unorganized pointclouds care must be taken not to exceed the available memory while still processing the high number of points as fast as possible. We give an algorithm that has negligible memory requirements and a runtime of $O(n\log(n))$ in the number of points. First, the bounding box of the point cloud is computed and the root node is initialized. The construction of the tree is then accomplished by sorting the point-list similar to quicksort. At each node the given list of points is first reorded with respect to the center of the node in one dimension. The resulting two sublists are then both reordered with respect to the center in the next dimension. The same is done with the last dimension. This operation results in eight possibly empty lists of points, one for each possible child. As this is done recursively in quicksort fashion for every node the initial list is sorted into the Morton order. The construction of the tree that takes place

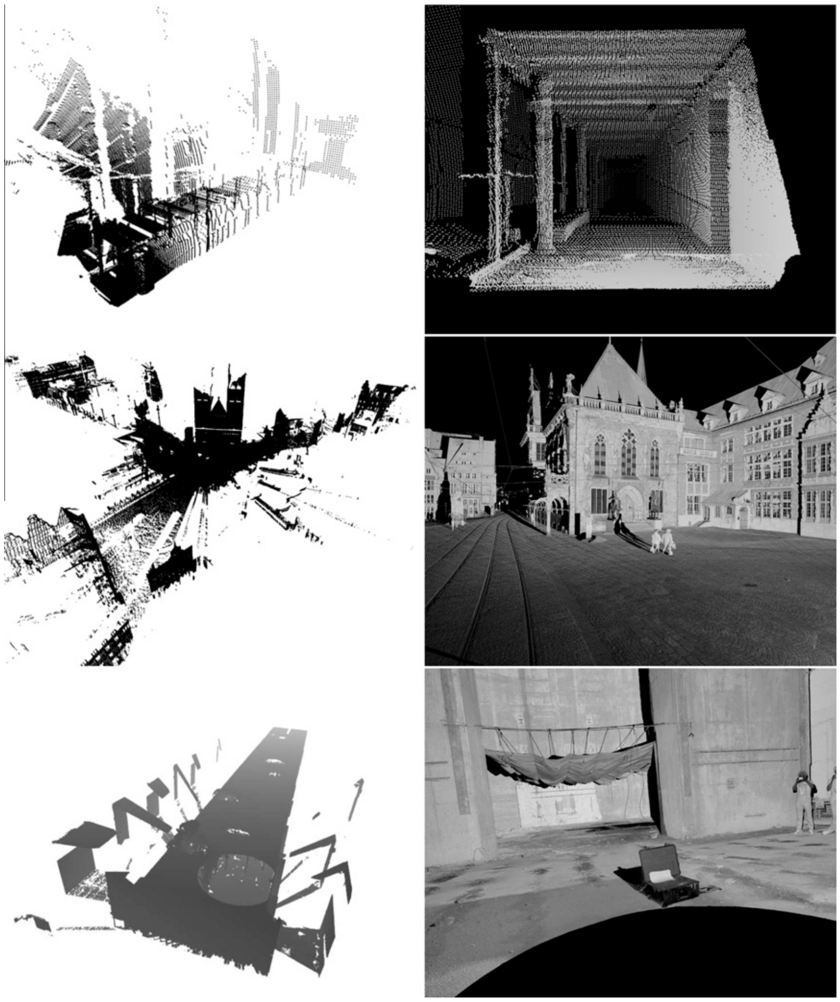

Fig. 6. Three point clouds are used for the following analysis. The left point cloud is a 3D scan that was acquired by the mobile robot Kurt3D using an actuated SICK LMS200 laser scanner in an office environment with 81,360 points (≈1.5 MB). Statistics for this data set is given in Table 1. The middle scan is a high resolution scan taken in the city center in Bremen using the Riegl VZ-400 3D scanner. The point cloud contains 15,896,875 points (≈303 MB). Refer to Table 2 for data on this point cloud. The right scan was also acquired by the RIEGL VZ-400, but in a large scale indoor environment, the bunker Valentin in Bremen-Farge. The point cloud is therefore very dense with 22,538,374 points (≈420 MB). Results are given in Table 2.

Table 2
Statistics for the Bremen city data set (cf. Table 1).

	8 bytes				7 bytes			100 bytes	
Leaf size (cm)	# Nodes	# Leaves	Mem. size	Constr. time (ms)	# Nodes	Incr. %	Mem. size	Mem. size	Constr. time (ms)
8560	6	12	192 B	1019.2	6	0	186 B	1.9 kB	557.1
4280	18	28	480 B	1321.2	18	0	462 B	4.8 kB	694.1
2140	46	86	1.4 kB	1535.7	46	0	1.3 kB	13.9 kB	939.2
1070	129	296	4.5 kB	1706.1	132	2.3	4.4 kB	45.3 kB	1165.7
535	408	800	12.8 kB	1909.7	428	4.9	12.5 kB	130.1 kB	1279.3
267	1166	2595	40.4 kB	2081.8	1228	5.3	39.7 kB	405.2 kB	1529.4
133	3616	7993	124.8 kB	2299.7	3823	5.7	122.6 kB	1.25 MB	1656.5
66.8	11,130	24,587	384.1 kB	2473.3	11,816	6.1	377.7 kB	3.85 MB	1895.5
33.4	33,965	75,999	1.18 MB	2687.5	36,403	7.1	1.16 MB	11.91 MB	2002.3
16.7	102,728	233,413	3.62 MB	2873.3	112,402	9.4	3.58 MB	36.65 MB	2146.5
8.35	302,573	687,529	10.67 MB	3134.9	345,815	14.2	10.67 MB	109.53 MB	2290.3
4.17	814,040	1,808,993	28.22 MB	3432.0	1,033,344	26.9	28.94 MB	301.28 MB	2471.9
2.08	1,927,234	4,166,979	65.42 MB	3721.0	2,842,337	47.4	69.9 MB	742.98 MB	2576.0
1.04	4,031,140	7,783,889	125.65 MB	3901.8	7,009,316	73.8	142.47 MB	1.568GB	2899.6
0.52	5,592,151	10,142,923	166.45 MB	4077.7	14,793,205	164.5	225.26 MB	2.643 GB	3199.9

Table 3
Statistics for the bunker Valentin data set (cf. Table 1).

	8 bytes				7 bytes			100 bytes	
Leaf size (cm)	# Nodes	# Leaves	Mem. size	Constr. time (ms)	# Nodes	Incr. %	Mem. size	Mem. size	Constr. time (ms)
5589	1	8	104 B	1330.0	1	0	103 B	954 B	565.1
2794	9	22	336 B	1631.9	9	0	327 B	3.28 kB	824.1
1397	30	59	948 B	1906.6	31	3.3	925 B	9.54 kB	1102.8
698	88	209	3.21 kB	2173.8	90	2.2	3.13 kB	31.69 kB	1367.7
349	294	755	11.41 kB	2403.9	299	1.7	11.15 kB	111.72 kB	1665.0
174	1038	2611	39.63 kB	2718.8	1054	1.5	38.71 kB	388.49 kB	1977.2
87.3	3602	8761	133.94 kB	2964.6	3665	1.7	130.78 kB	1.31 MB	2136.9
43.6	12,106	30,330	460.8 kB	3146.0	12,426	2.6	450.94 kB	4.53 MB	2434.0
21.8	41,098	101,743	1.54 MB	3409.5	42,756	4.0	1.52 MB	15.31 MB	2688.1
10.9	134,366	324,058	4.96 MB	3739.3	144,499	7.5	4.9 MB	49.667 MB	2767.0
5.45	412,174	969,512	14.93 MB	4079.1	468,557	13.6	14.91 MB	152.43 MB	3123.8
2.72	1,158,782	2,679,693	41.42 MB	4544.5	1,438,069	24.1	42.22 MB	436.48 MB	3227.9
1.36	2,977,454	6,617,215	103.22 MB	5057.0	4,117,762	38.2	108.23 MB	1.137 GB	3503.0
0.68	6,356,651	13,130,561	208.41 MB	5303.6	10,734,977	68.8	232.71 MB	2.529 GB	3895.8

simultaneously is therefore in $O(n\log(n))$. The necessary memory never exceeds the point list plus the octree structure.

2.4. Experiments and results

To demonstrate the effectiveness of the proposed octree encodings, we computed the required memory for the octree data structure (without the points) with different depths. This was done for three representative scans of differing density as given in Fig. 6. The data is given in Table 1–3 for different leaf sizes. It is important to note that the leaf size is only half of the side length of the leaf nodes, due to the way the octree volume is stored in our implementation. For all tests, the root volume and therefore all octree volumes axis aligned cubes. The size and position of the root is such that it represents the smallest cube possible to contain the entire data set. The time needed to construct the reference as well as the proposed octree has also been determined experimentally. For each leaf size the average construction time for the octrees out of 10 trials performed on an Intel(R) Xeon(R) E5520@2.27GHz is shown in Table 1–3.

The first data set has the smallest number of points, and is the least dense, i.e., it contains many points with large distances to their neighbors. Consequently, the performance of the constant depth octree very quickly deteriorates (cf. Table 1. Already, at a depth of five the memory benefit of the smaller node representation is negated. In the larger and denser datasets (cf. Tables 2 and 3), the octree reaches a depth of 10 before the constant depth requirement results in a larger memory footprint. Yet, even in the most dense dataset, several things speak against this constraint. At no resolution the memory benefit is particularly significant in the first place. The maximal memory saving in the bunker Valentin data set is only approx 20 kB. Compared to the size of the point cloud itself, which is about 700 MB this is neglegible. On the other hand, when the constant depth restriction results in a larger memory footprint, the penalty is several orders of magnitude larger than the benefit could ever be. The penalty also tends to increase polynomially with the tree depth. On level 15 the constant depth implementation requires almost 100 MB more memory. This is of course caused by the drastically increasing number of nodes required to pad the tree. A further consequence of this padding is the increased computations involved in iterating over the octree structure.

The time needed for the construction of either octree is near linear in the depth of the octree.. The reference octree is always faster by some fixed amount of time. This time difference is caused by the reallocation of the point data in the leaves, which happens in the proposed octree but not in the reference. In fact, if the construction of the reference also reallocates the points in a more compact fashion, it is slower than the proposed variant.

3. An open file format for exchanging 3D point clouds

Due to the great diversity of terrestrial 3D modeling applications several software products exists, which are suitable for special tasks. For airborne and kinematic laser scans there exists a common lidar data exchange format, the `.las` format (Samberg, 2007). "This binary file format is an alternative to proprietary systems or a generic ASCII file interchange system used by many companies. A problem with proprietary systems is that data cannot be easily taken from one system or process flow to another. In addition, processing performance is degraded because the reading and interpretation of ASCII elevation data can be very slow and the file size can be extremely large, even for small amounts of data."(American Society for Photogrammetry and Remote Sensing, 2011). This applies also for terrestrial 3D modeling pipelines, but there is currently no standard format available. Kern et al. (2009) defines a binary point cloud format that hence reduces the processing time.

The octree as given in Section 2 is well suited for storing large point cloud data, as it is a lossless compression, which reduces the size of a point cloud by a factor of roughly two and comes with a fast indexing. Table 4 presents the compression results for the three data sets. The serialization of the proposed octree results in an efficient method for storing point clouds.

Table 4
Compression results for the data sets. The compression ratio between the original binary data and the octree representation as well as the average error for representing the point of the data sets and their octree representation are given.

Data set	File size `.txt` MB	File size binary 64 (32 Bits) MB	File size compressed octree MB	Ratio %	Error µm
Kurt3D	1.907	1.862 (0.931)	0.472	50.73	4.165
Bremen city	477.0	424.4 (242.5)	121.6	50.14	5.102
Bunker Valentin	665.7	601.8 (343.9)	172.1	50.04	6.677

4. Efficient algorithms on octrees

4.1. Adaptive visualization using frustum culling

To exploit the octree structure for fast visualization we implemented frustum culling. The octree then works as a hierarchy of bounding volumes. Much like in the software `qsplat` (Rusinkiewicz and Levoy, 2000) it is used to quickly decide which points are visible. In addition, it also enables one to dynamically vary the level of detail to enforce a high framerate by rendering only a faction of points.

In computer graphics the frustum is the visible region of space. For a perspective projection, this region is a rectangular pyramid, constituted of six planes. Frustum culling is the process of distinguishing which parts of the scene are within and which parts are outside of the frustum, thus finding the visible parts of the scene. Non-visible elements do not need to be drawn, hence increasing the performance. Algorithm 1 describes how to efficiently implement the frustum culling using the octree data structure. The algorithm is initally called at the root of the octree, with the full viewing frustum.

Algorithm 1. display(set⟨plane⟩ frustum, octree node)

```
set⟨plane⟩ childfrustum;
for all $p \in$ frustum do
  if PlaneAABBCollision(p, this) = within then
    childfrustum.insert($p$)
  els if PlaneAABBCollision(p, this) = outside then
    return
  end if
end for
if isLeaf(node) then
  drawPoints(node)
else
  for all child $\in$ node.children do
    display(childfrustum, child)
  end for
end if
```

Algorithm 1 employs the function PlaneAABBCollision(), which determines the position of an octree volume with respect to a plane, i.e., whether the plane is within the volume or outside. The latter case has two subcases. Each plane of the frustum is oriented such that it is trivial to determine on which side a given point is. Consequently, if an octree volume is outside with respect to a plane, the node and all its children are not visible. If, on the other hand, the volume is entirely on the inside, culling with the respective plane is discontinued for all children.

To summarize, the given algorithm first determines which frustum planes are relevant for the children. Simultaneously, it checks wether the current octree volume is visible and terminates accordingly. The second step is to either display the points in the current leaf, or to recursively continue culling in the child nodes. In this fashion, one foregoes unneccessary collision checks both inside and outside of the viewing frustum. Furthermore, checks are also minimized for the intersection case, as only relevant frustum planes will be used. Fig. 7 illustrates this principle.

The main bottleneck in a software culling implementation is transmitting the point information to the graphics card when a large number of them is within the viewing frustum. Using the octree structure this problem can be mitigated in several ways. First, when an octree volume would appear as a single pixel it is sufficient to also send only a single vertex instead of all contained points without decreasing image quality. Second, we may dynamically adjust rendering quality to allow for navigation in the scene, similar to Richter and Döllner (2010) and Dachsbacher et al. (2003). As before, we send only single vertices in any octree volume that falls below a level-of-detail threshold (number of pixels on screen). The size of the rendered vertex is adjusted accordingly. With thresholds greater than 1, this trades resolution for speed. An example of levels of details are given in Fig. 8.

We have conducted experiments to find the optimal voxel resolution. Fig. 9 presents framerates for a point display employing the frustum culling implemented on the CPU. Several different voxel resolutions as well as the standard implementation with no software culling are plotted. To achieve comparable situations the point display has rendered a 3D-flight-through, where the camera moves on a path through the Bremen city data set. The camera started at a position, where the entire point cloud was visible, then proceeded to move through the point cloud close to the point of origin until it reached a position within the volume, where about 1% of the point cloud is visible. One observes a low framerate in the beginning, and an ever increasing speedup. The time for the rendering of each frame was measured by a clock with millisecond accuracy, so that the maximal framerate is 1000 frames per second.

The speedup gained by the frustum culling is small when large portions of the point cloud occupy the frustum. Reducing the number of points increases the framerate considerably more when using the octree culling as compared to the default point display. Note, that for scans that occupy an area this large it is more typical to only view a small fraction of the scene.

Comparing performances for various voxel sizes, we see that while large voxel sizes still result in some speedup, performance becomes more and more unsteady when a large number of points

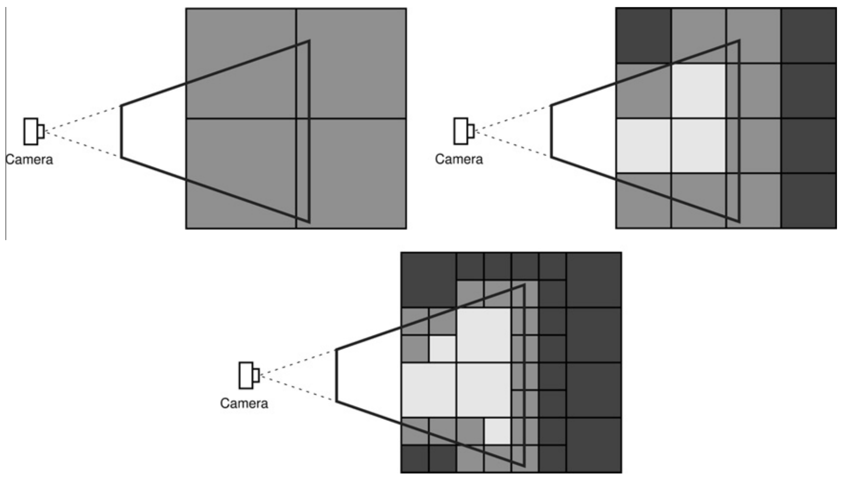

Fig. 7. The principle of frustum culling using octrees. Gray nodes are known to be partially within the frustum, so that culling must continue. Light gray Nodes are known to be entirely within the frustum, so that culling is discontinued. Dark gray nodes are entirely on the outside of the frustum.

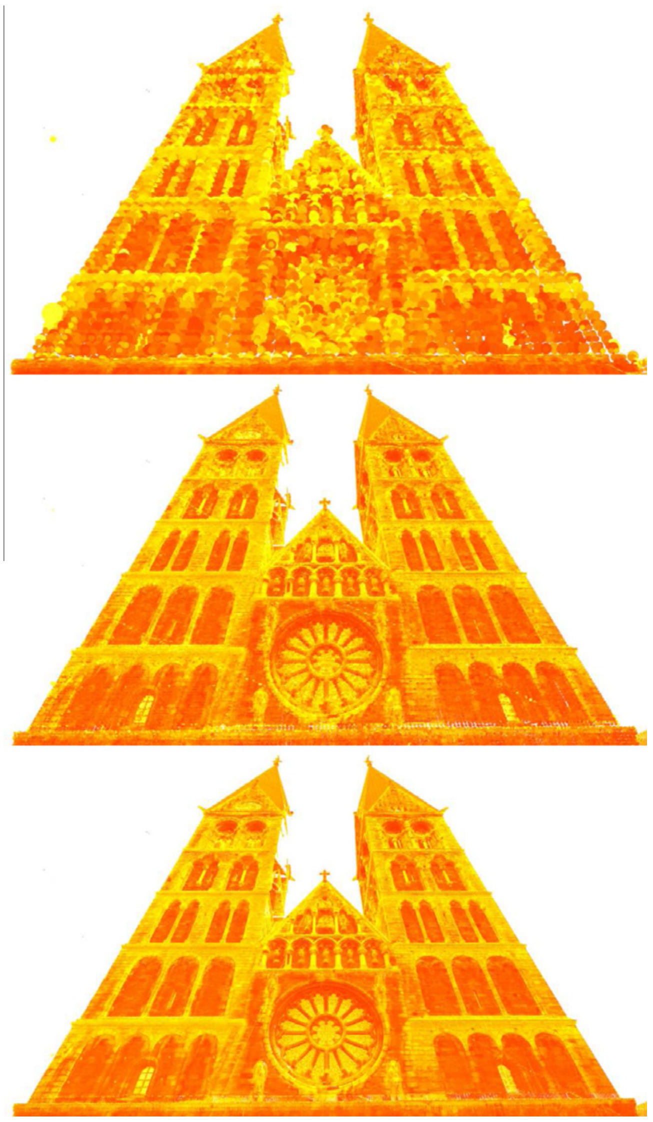

Fig. 8. Three levels of detail for the octree visualisation The point cloud is taken from the Bremen city data set.

is to be displayed. Framerates for large voxelsizes experience sudden jumps when voxels enter or leave the viewing frustum. This becomes apparent for a voxel size of 267 cm, but only somewhat noticable with the next smaller size of 133 cm. Generally, voxel sizes between 10 cm and 100 cm are optimal for these types of scans.

4.2. Ray casting

Ray casting is the method of finding an intersection of a ray with a surface. It is an important problem in computer graphics, where it is used to render images, typically of iso-surfaces (Roth, 1982; Knoll et al., 2006), and in robotics, where it is commonly used to sample potentional measurements (Thrun et al., 2000; Wurm et al., 2010). For rendering images, each pixel is followed from the viewpoint to the first object that is encountered. This is different from ray-tracing, where rays are reflected from objects so that multiple rays per image pixel are cast into the geometry. Ray-casting is thus a simplified ray-tracing technique.

Before efficient ray casting is possible in an octree without neighbor pointers, we detail how to perform fast indexed node access and neighbor traversal. Indexed node access is a lookup-query with (x,y,z) integer coordinates valid on the deepest level of the octree, i.e., an octree with depth d has integer coordinates $0–2^d - 1$ in each dimension. Naive lookup implementations perform *collision checks* with the octree planes. As this is an expensive task, we use integer coordinates for an efficient traversal of the octree with only a few bit operations. Similarly to Knoll et al. (2009) we assume the existence of a pre-computed array childBitDepth with childBitDepth$[d] = 1 \ll (\text{maxDepth} - d - 1)$. An efficient lookup is performed as in Algorithm 2. The key to this algorithm lies in the second line. Here the integer coordinates are mapped to the index of the child that contains the given coordinates. The algorithm also shows how parent pointers are simulated by a simple trace that is extended during the traversal of the tree. The function to find a neighbor node is merely an extended lookup. To find a neighbor of a given node, the node in the parent trace is selected that is the deepest that still contains the desired index. This can be efficiently computed by comparing the current node index with the desired index (cf. Algorithm 3). Then a lookup starting at that parent is started to locate the corresponding neighbor.

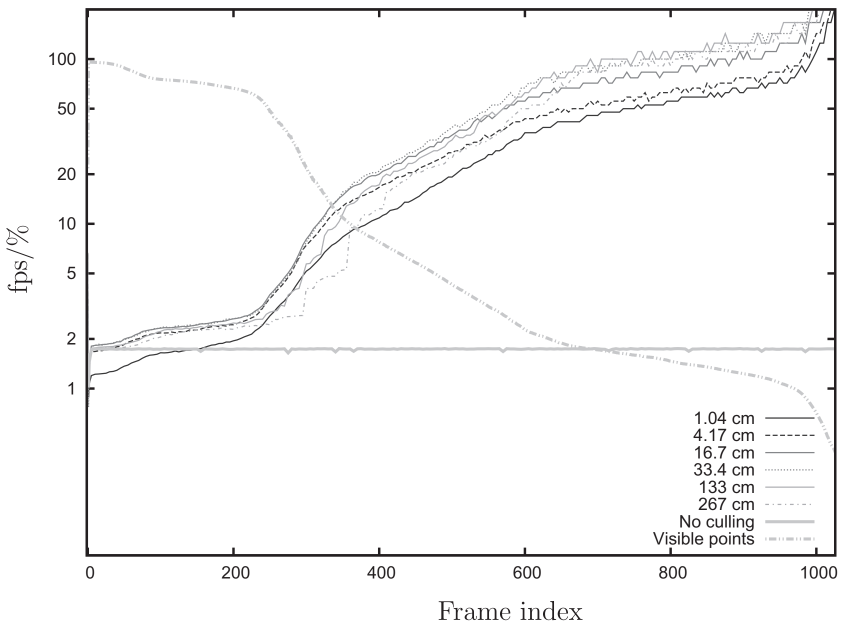

Fig. 9. Framerates of the octree culling with different voxel sizes over the frame index in the rendered video. The percentage of visible points in each frame is also plotted. Please note that the *y*-axis is on a logarithmic scale to be able to differentiate in the areas of both low and high framerates.

Algorithm 2. lookup(Vector3i index, octree node, octree *parentTrace, int depth)

```
loop
  int childBit = childBitDepth[depth]
  int childIndex = (index.x & childBit $\neq$0) $\ll$ 2 | (index.y &
   childBit $\neq$0) $\ll$ 1 | (index.z & childBit $\neq$0)
  octree *parentTrace[depth] = & node
  depth++
  node=node.children[childIndex]
  if isLeaf(node) then
    return
  end if
end loop
```

Algorithm 3. findNeighbor(Vector3i dindex, Vector3i cindex, octree *parentTrace)

```
int depth=mostSignificantBit((cindex.x $\wedge$ dindex.x) |
   (cindex.y $\wedge$ dindex.y) |
  (cindex.z $\wedge$ dindex.z));
lookup(dindex, parentTrace[depth], parentTrace, depth);
```

We eliminated the need for neighbor pointers for this algorithm. This comes at the cost of increased computational complexity. In the worst case we must backtrace completely up to the root and completely back down again. While these situations do not occur very often the traversal to a neighbor is still in $O(\log n)$. With the help of this function it is now feasible to implement ray casting, which is given in Algorithm 4. Here, Bresenham implements the 3-dimensional version of the well-known Bresenham line-algorithm, such that it consecutively returns 3-dimensional integer coordinates, that are traversed by the given ray.

Algorithm 4. castRay(Ray ray, octree root)

```
octree *parentTrace[maxDepth]
int depth = 0
octree node = lookup(ray.origin, root, parentTrace, depth)
Vector3i ci = Bresenham(ray);
Vector3i cci = ci;
loop
  while contains(node, ci) do
    cci=ci;
    ci=Bresenham(ray)
  end while
  node = findNeighbor(ci, cci, parentTrace)
  if isLeaf(node) then
    drawPoints(node)
    return
  end if
end loop
```

This naive ray casting can still be improved by collecting rays into packets as presented recently by Knoll et al. (2009). Using a technique called coherent octree traversal, access operations to the data structure is further minimized.

4.3. Nearest neighbor search for scan matching

Nearest neighbor search (NNS) is a part of many scan matching algorithms for establishing corresponding points. The most prominent example of this is the Iterative Closest Point (ICP) algorithm, which spends most of its processing time in the lookup of closest points (Besl and McKay, 1992). It is therefore of utmost importance for this application to reduce the computing time of this task. Naively implemented, finding a closest point requires iterating over the entire dataset, i.e., it is in $O(n)$, where n is the number of points in the point cloud. This expensive running time is avoided by employing metric data structures. The most popular data structure to speed up NNS in scan matching is the k-d tree (Friedman et al.,

Table 5
Average computing time for the ICP algorithm employing nearest neighbor search using *k*-d trees and octrees. The average was computed over 100 and 20 runs of ICP for the Kurt3D data set and the larger Riegl data sets, respectively. Default parameters were used for every test, with a maximal point-to-point distance of 25 cm and 50 ICP iterations. Random noise was added to the initial pose estimate for each run. The translational error was linearly distributed between −25 and 25 cm, whereas the rotational error was linearly distributed between −10° and 10°. Construction time for the trees is not included, all times are in milliseconds.

Data set	*k*-d Tree	Octree
Kurt3D	3043.099	2386.881
Bremen city	355848.476	314506.905
Bunker Valentin	837675.381	784241.905
Kurt3D reduced	757.514	625.683
Bremen city reduced	91735.667	74706.85
Bunker Valentin reduced	91153.0	74821.238

1977), where $k = 3$. As *k*-d trees are binary trees they allow an efficient implementation of NNS. Principally, octrees should allow for the same efficiency. In fact, due to the octree's regular subdivision it ought to be better suited for NNS than the popular *k*-d tree (Arya et al., 1998). The complication arises during the implementation of NNS in an octree. The key to an efficient traversal to the node containing the nearest neighbor for both tree variants is the order in which children are visited. The number of nodes that we need to visit is best reduced by the closest child first criteria (best bin first), i.e., the order of traversal is determined by the distance to the query point. This is trivial to do for the binary *k*-d tree, but requires some effort for an octree which may have one to eight child nodes.

For any octree node with eight children there is a total of 48 possible sequences in which to traverse the children. Every child corresponds to an octant of the entire coordinate space. The query point may fall into any of those eight octants. For each of those cases there are six possible traversals determined by the order of proximity of the query point to the three split planes. Therefore, NNS in an octree has to make proximity checks to three split planes, sort them and select the appropriate sequence of traversal for a closest child first search for every traversed node. Compared to this, the traversal order for a *k*-d tree order is instantly determined by a single proximity check, thereby avoiding unnecessary computations if nodes need not to be visited.

The regular subdivisions of an octree are still leveraged for an NNS that is in most cases faster or as fast as in a *k*-d tree. The largest benefit is that fast indexing as in Section 4.2 is possible in an octree. This allows us to directly traverse to the deepest octree node, which contains the bounding sphere of the query point, with a constant number of floating point operations. The full NNS with closest child first and backtracking is then performed on this node. The initial node lookup is considerably faster than the equivalent operation, essentially a point lookup that is already in the tree, is in a *k*-d tree. However, the speedup gained by this is clearly dependant on the maximal allowed distance to the query point. The smaller the maximal distance, the deeper the initial node lookup can traverse on average. The deeper said node is, the fewer computations are performed in the following NNS. To evaluate the performance of the NNS for varying maximal distances, we performed ICP scan matching employing the octree and a *k*-d tree NNS. Results are given in Table 5 and Fig. 10. Despite the previous argumentation, the octree based NNS does *not* suffer considerably more from larger maximal distances than the *k*d tree based NNS. We observe however, that there is an increase in the variance for the time required for the NNS using the octree. Conversely, the variance of the *k*-d tree based NNS is stable over all distances. This suggests that the octree is more vulnerable to the combination of large maximal distances and unfavorable starting pose estimates.

For ease of implementation and to further reduce the number of floating point operations, we restrict the number of traversals to eight instead of 48. Since the order of traversal only depends on the octant into which the query point falls, there is no need for proximity checks or sorting. Consequently, no floating point operations are required in our NNS implementation except in the leaves of the octree, where the list of stored points are checked against

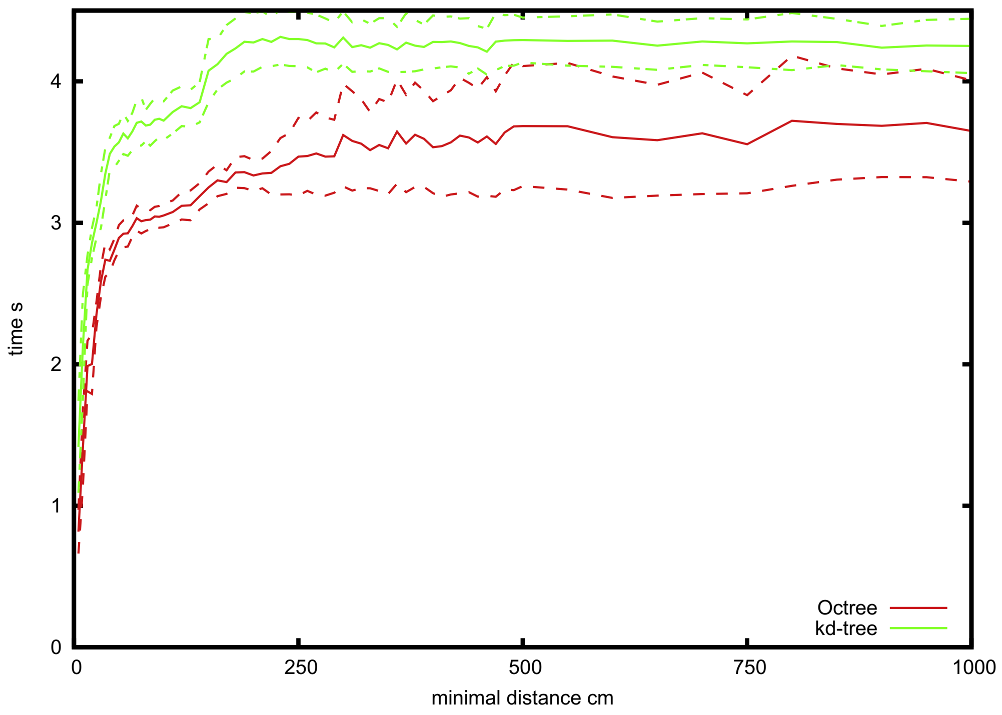

Fig. 10. Plot of the average and standard deviation of the computing time of ICP using *k*-d trees and octrees with respect to the maximal allowed matching distance. For each data point 100 runs of ICP, with 50 iterations each, were done with noise applied to the initial starting pose estimate as in Table 5.

the query point. The approach is summarized in Algorithm 5. We use the function FindClosestInLeaf(), which is a straightforward check of the points stored in the leaf. The initial lookup for finding the deepest octree node that contains the bounding box around the query point is a modified indexed lookup. For this purpose the two diametrically opposed corners of the bounding box are converted into integer coordinates. The tree is then traversed as in Algorithm 2 until both indices disagree on the child that is to be traversed next.

Algorithm 5. FindClosest

```
Input: query point $q$, maximal allowed distance $d$
  lookup deepest node $N$ containing bounding box of $q$
  convert $q$ to octree coordinate $i$
  return FindClosestInNode($N$, $q$, $i$, $d$)
```

Algorithm 6. FindClosestInNode

```
Input: query point $q$ and its coordinate $i$
Input: maximal allowed distance $d$ and the current node $N$
1:  compute child index $c$ from $i$
2:  for $j = 0$ to 8 do
3:    get next closest child $C = N$.children[sequence[$c$][$j$]];
4:    if $C$ is outside of bounding ball then
5:      return currently closest point $p$
6:    else
7:      if $C$ is a leaf then
8:        FindClosestInLeaf($C$, $q$, $d$)
9:        update currently closest point $p$
10:     else
11:       FindClosestInNode($C$, $q$, $i$, $d$)
12:       update currently closest point $p$
13:     end if
14:   end if
15: end for
16: return currently closest point $p$
```

4.4. RANSAC for efficient parameter estimation

The Random Sample Consensus (RANSAC) algorithm is an approach for estimating parameters of a model that best describes a set of sample points (Fischler and Bolles, 1981). While it is traditionally used for line and plane detection RANSAC can also be used for any other parameterized model. It is an iterative algorithm that repeatadly draws a small number of samples from the data to be modelled. From this subset of samples the parameters of the model are computed and the number of points in the entire data set that intersect with the model is calculated. As the process is repeated the model with the largest number of points is selected as the result.

The most expensive step of the algorithm is determining the number of points that agree with the candidate model. In an unorganized point cloud this requires iterating over all points. We employ the octree data structure and obtain a significant speedup. Similar to Section 4.2, where a frustum is checked against an octree, the candidate model is recursively intersected with the octree nodes to determine which nodes may contain points on the model. The process is exemplary depicted for the 2-dimensional case in Fig. 11. After a candidate line has been generated, intersection tests are performed on the children of nodes known to be at least partially on the line. This process is executed from top to bottom. Finally, the points in the leaves are counted. In this way a large number of points is automatically excluded from being checked against the model, thus leading to a speedup. A comparison of the computing time of a standard RANSAC for plane detection with octree-enabled version is given in Table 6. Even including the construction time of the octree we achieve significant speed-up. Examples of detected planes for the three data sets are given in Fig. 12.

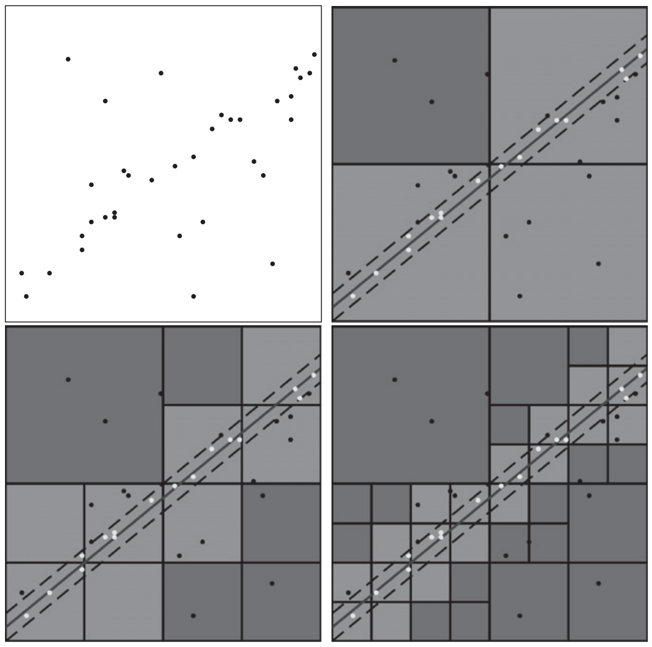
Fig. 11. Speed-up of RANSAC using a tree data structure. Top left: The initial sample set, where a line should be detected. Top right: A line has been generated from a sample set and is intersected with the octree. The dashed lines signify the maximal distance threshold of RANSAC. Dark gray Nodes are outside of the model, light gray nodes intersect the line. Bottom: The intersection test continues only in those children of the nodes that are known to intersect the model.

In addition octrees are able to speed up RANSAC by a smart sampling scheme. Schnabel et al. (2007) have shown that by carefully selecting samples that are in proximity to each other, the number of iterations required to detect a shape with a certain probability is reduced by several orders of magnitude. This is done by first selecting a sample in some random leaf l, and then selecting further samples only from children of a randomly selected parent node of l.

5. Related work

Since its introduction in the early 80's by Meagher (1982), the octree data structure has experienced widespread use among various fields that deal with large quantities of 3-dimensional data, especially computer graphics (Knoll et al., 2006; Knoll et al.,

Table 6
Average computing time of RANSAC for plane detection with 5000 iterations with as well as without the octree. Results were averaged over 100 runs. For each data set 5000 iterations of the RANSAC were done. The octree times include the construction of the octree (≈8 s for Bremen city and ≈10 s for the Bunker data set). All times are in milliseconds.

Data set	No octree	Octree	Speed-up
Kurt3D	1666.57	176.69	9.43
Bremen city	388551.55	11084.81	35.05
Bunker Valentin	539536.13	26399.15	20.43

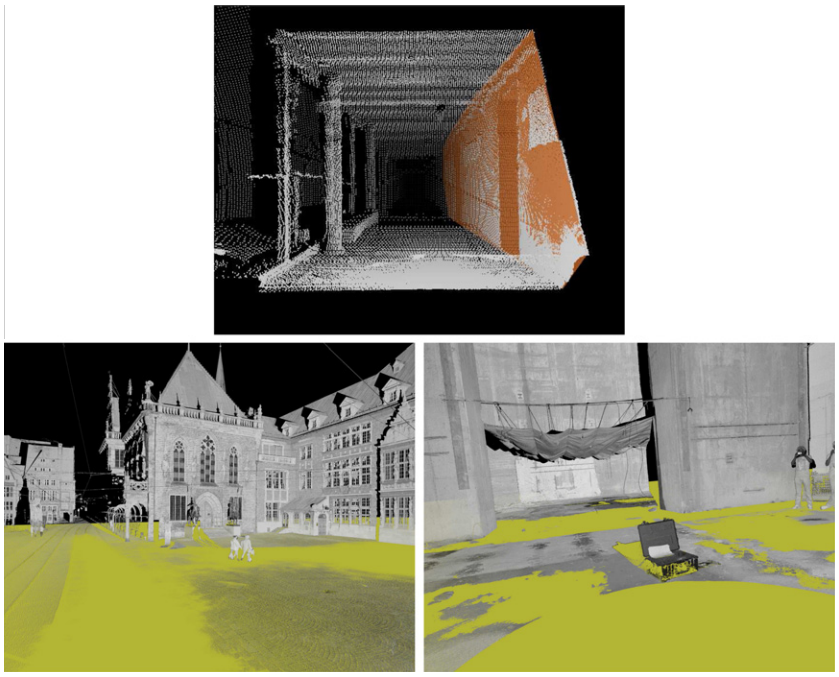

Fig. 12. Extracted planes using RANSAC for the three data sets presented in Fig. 6.

Table 7
Memory requirements for a single node of different libraries in bytes on 64 bits systems. An *x* notifies that the library is templated, and a number of bytes must be added to the total byte count. The number depends on the size of the type stored in the tree and can be expected to be several bytes.

Library	Node size	Remarks
`xgrt`	144 + *x*	
octomap	72 + *x*	
PCL	64 + *x*	
PCL low memory base	25 + *x*	(since ver 1.1.1. September 2011)
CloudCompare	16	size of leaf node
3DTK	8	

2009; Enrico et al., 2008; Samuli and Tero, 2010) but also theoretheical physics (Bielak et al., 2005) and, of course, robotics (Wurm et al., 2010).

In computer graphics and visualization the octree has recently had a resurgence under the term *sparse voxel octree*. Here, the octree is used for efficient ray tracing and casting. An octree simultaneously represents large datasets with a small memory footprint and provides ray casting in $O(\log n)$, which yields superior performance, since realtime visualization using ray tracing is usually in $O(n)$, where n is the number of objects in the scene. In this application octrees are representations of a voxelmap only, i.e., leaf nodes do not store other data, apart from the inherent properties needed for the visualization of a voxel such as color and normals. Samuli and Tero (2010) have presented an octree encoding for the GPU which is similar to ours. Their implementation stores countours in each node to improve visualization quality in addition to the usual color and normal information. Knoll et al. (2006), Knoll et al. (2009) have developed powerful ray tracing algorithms on octrees. They employ the fast indexing scheme as presented by Frisken and Perry (2002) and improve upon standard ray traversal with slice-based coherent octree traversal.

QSplat is a non-octree based visualization program for point clouds (Rusinkiewicz and Levoy, 2000). Points, their normals and colors are stored in a hierarchy of bounding spheres. Responsive display of massive amounts of data can be maintained by dynamically reducing the complexity of the object or scene. This is achieved by visualizing the hierarchy only to a certain depth. The bounding sphere data structure, although compact, is not well suited to applications other than visualization. Construction thereof is non-trivial and ambiguous. Due to its regularity, an octree is more compact and suited for other applications, too.

Several libraries exist which employ the octree data structure for storing and processing point clouds, like PCL (Rusu et al., 2011; Rusu and Cousins, 2011), octomap (Wurm et al., 2011), CloudCompare (Girardeau-Montaut, 2011) and `xgrt` (Wand et al., 2012). We give the node sizes for each library and compare them to our approach in Table 7.

Furthermore, octrees are used in the color quantization algorithm as designed by Gervauts and Purgathofer (1990) to minimize memory requirements. Each level in their octree represents 1 bit of the colors contained therein, beginning with the most significant bit in the first level. This is similar to our compression of points, where the more significant bits are implicitly stored in the octree data structure.

6. Conclusions and outlook

This paper presents a novel implementation of a basic data structure for 3D point clouds. The data structure, an octree, exceeds the purpose of storing data. The pipeline for obtaining 3-dimensional models is sped-up. To this end, we have presented an efficient exchange file format, fast point cloud visualization, effective 3D scan matching, and a clever plane detection algorithm.

In future work we will continue using our octree for efficient 3D point cloud processing, e.g., for globally consistent scan registration (Borrmann et al., 2008), for automatically deriving semantic information, for dynamic maps, i.e., maps that can handle changes of the scene, and for next-best-view planning.

Acknowledgments

We would like to thank Nikolaus Studnicka (RIEGL Laser Measurement Systems) for his idea to propose the octree as exchange file format. Furthermore, we would like to thank Prashant K.C. for acquiring many terrestrial laser scans on our campus.

References

Morton, 1966. Tech. Rep. Ottawa, Ontario, Canada, IBM Ltd.

Friedman, J.H., Bentley, J.L., Finkel, R.A., 1977. An algorithm for finding best matches in logarithmic expected time. ACM Transaction on Mathematical Software 3 (3), 209–226.

Fischler, M.A., Bolles, R.C., 1981. Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography. Communications of the ACM 24 (6), 381–395.

Meagher, Donald, 1982. Geometric modeling using octree encoding. Computer Graphics and Image Processing 19 (2), 129–147.

Roth, Scott D., 1982. Ray casting for modeling solids. Computer Graphics and Image Processing 18 (2), 109–144.

Gervauts, M., Purgathofer, W., 1990. A Simple Method for Color Quantization: Octree Quantization. In: Graphics Gems. Academic Press Professional Inc., San Diego, CA, USA, pp. 287–293.

Besl, P., McKay, N., 1992. A method for registration of 3-D shapes. IEEE Transactions on Pattern Analysis and Machine Intelligence (PAMI) 14 (2), 239–256.

Arya, S., Mount, D.M., Netanyahu, N.S., Silverman, R., Wu, A.Y., 1998. An optimal algorithms for approximate nearest neighbor searching in fixed dimensions. Journal of the ACM (JACM) 45 (6), 891–923.

Thrun, S., Fox, D., Burgard, W., 2000. A Real-time Algorithm for Mobile Robot Mapping with Application to Multi Robot and 3D Mapping. In: Proceedings of the IEEE International Conference on Robotics and Automation (ICRA '00), San Francisco, CA, USA, pp. 321–328.

Szymon Rusinkiewicz, Marc Levoy, 2000. QSplat: A multiresolution point rendering system for large meshes. In: Proceedings of the ACM SIGGRAPH. pp., 343–352.

Frisken, Sarah F., Perry, Ronald N., 2002. Simple and efficient traversal methods for quadtrees and octrees. Journal of Graphics Tools 7 (3), 1–11.

Dachsbacher, C., Vogelgsang, C., Stamminger, M., 2003. Sequential point trees. In: ACM SIGGRAPH 2003 Papers. SIGGRAPH '03, pp. 657–662.

Girardeau-Montaut, D., Roux, M., Marc, R., Thibault, G., 2005. Change detection on points cloud data acquired with a ground laser scanner. International Archives of Photogrammetry, Remote Sensing and Spatial Information Sciences 36 (Part 3/W19), 30–35.

Bielak, J., Ghattas, O., Kim, E.J., 2005. Parallel octree-based finite element method for large-scale earthquake ground motion simulation. Computer Modeling in Engineering and Sciences 10 (2), 99–112.

Knoll, A., Wald, I., Parker, S., Hansen, C., 2006. Interactive isosurface ray tracing of large octree volumes. 115–124.

Schnabel, R., Klein, R., 2006. Octree-based point-cloud compression. In: Symposium on Point-Based Graphics 2006, pp. 111–120.

Samberg, A., 2007. An Implementation of the ASPRS LAS Standard. International Archives of Photogrammetry, Remote Sensing and Spatial Information Sciences 36 (Part 3/W52), 363–372.

Schnabel, R., Wahl, R., Klein, R., 2007. Efficient RANSAC for point-cloud shape detection. Computer Graphics Forum 26 (2), 214–226.

Wand, M., Berner, A., Bokeloh, M., Fleck, A., Hoffmann, M., Jenke, P., Maier, B., Staneker, D., Schilling, A., 2007. In: Proceedings Symposium on Point-Based Graphics (PBG '07), pp. 37–46.

Enrico, Gobbetti, Fabio, Marton, Guiti, Iglesias, Antonio, Jose, 2008. A single-pass GPU ray casting framework for interactive out-of-core rendering of massive volumetric datasets. The Visual Computer: International Journal of Computer Graphics 24 (7), 797–806.

Borrmann, D., Elseberg, J., Lingemann, K., Nüchter, A., Hertzberg, J., 2008. Globally consistent 3D mapping with scan matching. Journal Robotics and Autonomous Systems (JRASs) 56 (2), 130–142.

Knoll, Aaron M., Wald, Ingo, Hansen, Charles D., 2009. Coherent multiresolution isosurface ray tracing. The Visual Computer: International Journal of Computer Graphics 25 (3), 209–225.

Fredie Kern, Michael Pospis, Olaf Prümm, 2009. In: Photogrammetrie Laserscanning Optische 3D-Messtechnik, Beiträge der Oldenburger 3D-Tage, pp. 20–30.

Laine, Samuli, Karras, Tero, 2010. Efficient sparse voxel octrees. In: Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games (I3D '10), pp. 55–63.

Wurm, K.M., Hornung, A., Bennewitz, M., Stachniss, C., Burgard, W., 2010. OctoMap: a probabilistic, flexible, and compact 3D map representation for robotic systems. In: Proceedings of the IEEE ICRA Workshop on Best Practice in 3D Perception and Modeling for Mobile Manipulation. Anchorage, AK, USA.

R. Richter, J. Döllner, 2010. Out-of-core real-time visualization of massive 3D point clouds. In: Proceedings of the 7th International Conference on Computer Graphics, Virtual Reality, Visualisation and Interaction in Africa, AFRIGRAPH '10, pp. 121–128.

Radu Bogdan Rusu, Steve Cousins, 2011. 3D is here: Point Cloud Library (PCL). In: Proceedings of the IEEE International Conference on Robotics and Automation (ICRA '11), ICRA Communications, pp. 1–4.

Girardeau-Montaut, D., 2011. Cloudcompare. <http://www.danielgm.net/cc> (Accessed 14.08.12).

American Society for Photogrammetry and Remote Sensing, 2011. Common Lidar Data Exchange Format. <http://www.asprs.org/Committee-General/LASer-LAS-File-Format-Exchange-A%ctivities.html> (Accessed 14.08.12).

Wurm, K.M., et al., 2011. Octomap. <http://octomap.sourceforge.net/> (Accessed 14.08.12).

Radu Bogdan Rusu, et al., 2011. Point Cloud Library. <http://pointclouds.org/> (Accessed 14.08.12).

M., Wand, Berner, A., Bokeloh, M., Fleck, A., Hoffmann, M., Jenke, P., Maier, B., Staneker, D., Parys, R., 2011. xgrt. <http://www.gris.uni-tuebingen.de/xgrt> (Accessed 14.08.12).