



Efficient Construction of Out-of-Core Octrees for Managing Large Point Sets

Jonathan Fischer¹, Paul Rosenthal² , and Lars Linsen³ 

¹Chemnitz Technical University, Germany ²University of Rostock, Germany ³University of Münster, Germany

Abstract

Among various space partitioning approaches for managing point sets out-of-core, octrees are commonly used for being simple and effective. An efficient and adaptive out-of-core octree construction method has been proposed by Kontkanen et al. [KTO11], generating the octree data in a single sweep over the points sorted in Morton order, for a given maximum point count m per octree leaf. Their method keeps $m + 1$ points in memory during the process, which may become an issue for large m . We present an extension to their algorithm that requires a minimum of two points to be held in memory in addition to a limited sequence of integers, thus adapting their method for use cases with large m . Moreover, we do not compute Morton codes explicitly but rather perform both the sorting and the octree generation directly on the point data, supporting coordinates of any finite precision.

CCS Concepts

• **Computing methodologies** → *Rendering; Point-based models;*

1. Introduction

Visualizing large point-based volumetric data sets imposes challenging data management tasks, especially for data with highly varying spatial point distribution, such as those produced by Smoothed Particle Hydrodynamics (SPH) simulations [LMD*11]. Due to their enormous sizes, such large SPH data sets demand out-of-core techniques and level-of-detail approaches for high-quality rendering at acceptable frame rates. During both preprocessing and rendering of these data, hierarchical space partitioning is employed for memory management and data processing tasks.

Among various hierarchical space partitioning schemes, octrees are popular for being simple and effective. Especially in the context of out-of-core techniques, they have the unique advantage of allowing a highly efficient construction: Given a maximum number m of points per octree leaf, an octree can be built during a single sweep over the point set if the points are sorted along a space-filling curve that traverses the octree to an infinite depth. The octree can be constructed immediately during the out-of-core point sorting process by integrating the construction as a simple filter right before outputting the sorted points.

The idea of building an octree for points along a space-filling curve has already been followed by Salmon and Warren [SW97] for use in N-body simulations. Later, Kontkanen et al. [KTO11] presented the procedure more thoroughly and extended it by a chunking scheme for saving more coherent octree data to disk to speed up its later traversal. For their use case of rendering large point clouds, they employed a rather small $m = 16$, resulting in a large octree which thus had to be out-of-core itself.

Our use case is memory management for processing large astrophysical SPH particle sets out-of-core. Since we require the particles of a rather compact volume region to be loaded at any time, a much larger m is appropriate, which may very well reach millions. Although constructing the octree in a single sweep over the particles is favorable, Kontkanen et al.'s algorithm bears the disadvantage of requiring $m + 1$ points in memory. While this is feasible also in our use case, we would like to employ a method with a memory footprint that does not depend on m , to assign more of the available memory to the sorting process that is run concurrently.

We therefore present an extension to their algorithm, which processes the sorted point stream in chunks of arbitrary size. Instead of keeping the points in yet unprocessed octree nodes in memory, we build a temporary sub-octree from them, which in this case is fully defined by its number of leaves at every level and the particle count of each leaf. When moving from one point chunk to the next, we thus only require one point and an integer sequence of size $\mathcal{O}(\log m)$ in memory, which keeps us equally well prepared for any points to come as saving all points would.

Moreover, we do not compute any Morton codes explicitly but instead employ a Morton order comparator function acting directly on the floating-point coordinates as proposed by Connor and Kumar [CK10]. Their method makes use of a routine xorMSB for computing the highest index of a differing bit of two floating-point numbers. Since we reuse this routine in our octree construction scheme, we introduce their comparator function in Section 2 before explaining our out-of-core octree construction algorithm in Section 3. We then conclude with a performance analysis in Section 4.

2. Computing Morton Order

Morton's Z -order curve is a space-filling curve with numerous applications concerning multidimensional data. For point coordinates with finite binary representation, as is the usual case for applications involving point data, its inverse function is well-defined and maps any point in multiple dimensions to its unique Z -value or *Morton code*, which can be computed by interleaving the binary digits of the point's coordinates. A set of points can therefore be sorted by their Z -values, in *Morton order*.

In many applications of Morton order, the point coordinates are rasterized into integer coordinates prior to computing the Z -values. This facilitates a uniform bit length for storing them but limits the coordinates' precision, which can be undesirable in cases of highly varying point densities. However, for sorting points by their Z -values it is not necessary to explicitly compute these. Chan [Cha06] showed that it suffices to find the pair of corresponding coordinates that differ in the most significant bit and compare these coordinates.

This approach has been extended to floating-point coordinates by Connor and Kumar [CK10]. We explain it in short here, assuming non-negative coordinates for simplicity, although it can be easily extended to signed ones. Given two points $\mathbf{x} = (x_1, x_2, x_3)$ and $\mathbf{y} = (y_1, y_2, y_3)$, a routine `xorMSB` is invoked on each coordinates pair (x_i, y_i) , which returns the greatest index of a differing bit of x_i and y_i , serving as a measure of the pair's importance. Then \mathbf{x} and \mathbf{y} are compared by their most important coordinates pair, giving precedence to higher index i in case of equal importance.

We reuse `xorMSB` in our octree construction method and therefore restate it here. Let \mathbb{F} denote the set of values of some floating-point data type, `MANT_LENGTH` the size of the mantissa used, and `INT_MIN` be smaller than the smallest index of a possibly set bit of any element of \mathbb{F} , i. e., smaller than the smallest representable exponent with the mantissa length subtracted.

Algorithm 1: `xorMSB(x, y)`

Data: $x, y \in \mathbb{F}$, both non-negative

Result: `INT_MIN` if $x = y$,

else the highest bit index where x and y differ

```

1 if  $x = y$  then return INT_MIN
2 if  $x = 0$  then return  $\lfloor \log_2 y \rfloor$ 
3 if  $y = 0$  then return  $\lfloor \log_2 x \rfloor$ 
4 if  $\lfloor \log_2 x \rfloor = \lfloor \log_2 y \rfloor$  then
5    $a \leftarrow \text{MANT\_LENGTH} - \lfloor \log_2 x \rfloor$ 
6   return  $\lfloor \log_2([2^a x] \oplus [2^a y]) \rfloor - a$ 
7 return  $\max(\lfloor \log_2 x \rfloor, \lfloor \log_2 y \rfloor)$ 

```

We provide pseudocode for `xorMSB` in Algorithm 1. Given $x, y \in \mathbb{F}$, we return the index of the highest significant bit of either x or y if the other highest significant bit index is not defined (Lines 2 and 3) or lower (Line 7). In case they are the same, we examine the mantissae in Line 6: We first shift the binary points of x and y by a bits, such that they have no fractional digit set. Then we convert each result to an integer type with at least `MANT_LENGTH` bits without losing any precision. Afterwards, we apply a bitwise XOR operation \oplus to them, take the index of the highest set bit of the result, and transfer it back to the order of binary indices of x and y .

3. Octree Construction

Kontkanen et al.'s Algorithm. The algorithm by Kontkanen et al. writes the octree nodes to a file in the order of a depth-first traversal following Morton order. Inner nodes are kept on a stack while processing their children, after which the inner nodes themselves are constructed.

Starting at the root node, the algorithm reads enough points to decide whether the *current node* needs to be split up, i. e., whether it contains more than m points. If this is the case, the node is pushed to the stack and its first child becomes the new current node. Otherwise, the current node is *finalized*, i. e., its node data are written to the file, and its next sibling becomes the current node. Whenever a finalized node is the last among its siblings, all points of its parent have been processed. Then, the parent node is popped from the stack and finalized itself. The procedure is followed until the root node is finalized.

To decide whether more than m points reside in the current node and to find its number of points if it is a leaf node, the algorithm keeps $m + 1$ points in a FIFO queue in memory. When finalizing a leaf node, its points are removed from the queue before refilling it with enough points to decide on the next node.

Our Extension. Our algorithm is an extension to Kontkanen et al.'s which considers the points in chunks of arbitrary size. If a chunk is not large enough to decide whether the current node contains more than m points, we build a virtual *temporary subtree* in memory, rooted in the current node. The subtree contains the point counts of the nodes that we may have to export in the future if their direct parents turn out to contain more than m points. Thereby, it needs to be split up to the level necessary to cope with any further points coming up in later chunks. Since we allow up to m exactly identical points in the stream, the temporary subtree must be deep enough to separate the last seen point \mathbf{p}_{last} from the last different point before. Despite possibly comprising many levels, the structure of the temporary subtree is rather simple in that, for every level, only the last node considered so far can be a non-leaf, as the ones coming before in Morton order are already known to not contain more than m points.

Figure 1 illustrates the temporary subtree in an example case. To encode it, we only need the point counts of the nodes *temporarily finalized* so far, i. e., the sibling nodes preceding the one containing \mathbf{p}_{last} in Morton order on every level. We structure this point count record as a double-ended queue \mathcal{S} of lists $\mathbf{s}_0, \mathbf{s}_1, \dots, \mathbf{s}_{\text{size}(\mathcal{S})-1}$, each of which contains the up to seven counts corresponding to one octree level. We write $\sum \mathbf{s}_i$ to denote the sum of the point counts in a single list \mathbf{s}_i , and $\sum \mathcal{S}$ to indicate the sum of all point counts in \mathcal{S} . Further, we use a variable n holding the number of points processed so far that do not belong to a finalized node yet.

If one wanted to keep our algorithm's memory footprint minimal, one could set it up to consider the points in chunks of size one, which would then require memory for two points, for \mathcal{S} , and for a small fixed number of bookkeeping variables used while processing the current point chunk. However, larger chunks greatly accelerate the process, as we discuss in Section 4.

We employ a variable B , which is always kept equal to the binary

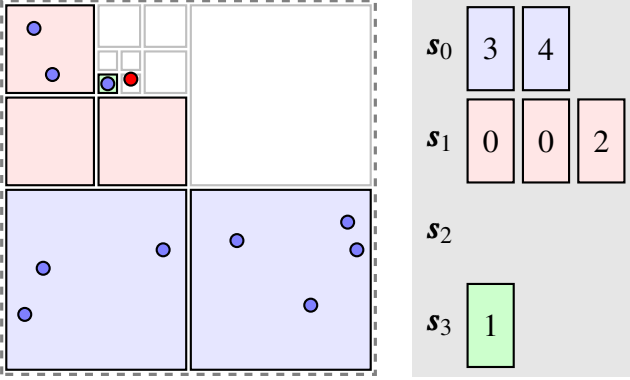


Figure 1: Example configuration of the temporary subtree in 2D. Having processed $11 \leq m$ points in the current node (dashed gray) so far, we cannot decide yet whether it has to be an inner node. Hence, we build up a temporary subtree rooted in the current node, up to the depth necessary to separate the last point \mathbf{p}_{last} (red) from its predecessor in Morton order. The point counts of the temporarily finalized nodes are recorded in a queue \mathcal{S} of lists $\mathbf{s}_0, \mathbf{s}_1 \dots$ depicted on the right-hand side, which defines the entire subtree structure.

logarithm (i.e., logarithm to base 2) of the current octree node's edge length. It indicates the node's octree level in absolute terms, irrespective of the root node size, which is why we call B its *absolute octree level* of the current node. Likewise, another absolute octree level is indicated by variable C , namely of a node considered for possibly later becoming the current node. While the current node is the octree node containing \mathbf{p}_{last} at level B , we refer to this node containing \mathbf{p}_{last} at level $C \leq B$ as the *current potential node*.

The greatest absolute octree level required for separating two points $\mathbf{x} = (x_1, x_2, x_3) \in \mathbb{F}^3$ and $\mathbf{y} = (y_1, y_2, y_3) \in \mathbb{F}^3$ is the binary logarithm of the edge length of the largest node containing \mathbf{x} but not \mathbf{y} . We compute it with the help of the xorMSB routine defined in Algorithm 1, using the function

$$\text{mbs}(\mathbf{x}, \mathbf{y}) = \max_{i=1}^3 (\text{xorMSB}(x_i, y_i)).$$

Given a point $\mathbf{x} = (x_1, x_2, x_3) \in \mathbb{F}^3$ and an absolute octree level $C \in \mathbb{Z}$, the octree node of size 2^C containing \mathbf{x} is fully fixed. To determine its index in $\{0, \dots, 7\}$ among its direct siblings in Morton order, we employ

$$\text{mbi}_C(\mathbf{x}) = \sum_{i=1}^3 2^{i-1} \left(\left\lfloor \frac{x_i}{2^C} \right\rfloor \bmod 2 \right).$$

We provide pseudo-code for our method in Algorithm 2. The steps referring to Kontkanen et al.'s method are highlighted with surrounding boxes. We have designed this code to also work in the case of multiple exactly equal points as long as there are at most m instances of each. However, during the following textual explanations, we assume distinct points for simplicity.

We consume the first point while initializing the current node, \mathcal{S} , B , n , and \mathbf{p}_{last} in Lines 1 to 5. Then we consider the remaining

Algorithm 2: Out-of-Core Octree Construction

- Data:**
- upper bound m of points per octree leaf
 - point sequence in Morton order, not containing more than m exactly equal instances
 - octree root node

```

1  set current node to root node
2  initialize  $\mathcal{S}$  to an empty list
3   $B \leftarrow \log_2(\text{current node edge length})$ 
4   $n \leftarrow 1$ 
5  initialize  $\mathbf{p}_{\text{last}}$  to the first point
6  foreach point chunk  $\mathbf{P} = (\mathbf{p}_0, \mathbf{p}_1, \dots, \mathbf{p}_{\text{size}(\mathbf{P})-1})$  do
7       $C \leftarrow B$ 
8      while  $\text{size}(\mathbf{P}) > 0$  do
9           $l \leftarrow \min(\text{size}(\mathbf{P}) - 1, m - n + \sum_{i=0}^{B-C-1} \sum \mathbf{s}_i)$ 
10         if  $\mathbf{p}_{\text{last}} = \mathbf{p}_{\text{size}(\mathbf{P})-1}$  then
11              $n \leftarrow n + \text{size}(\mathbf{P})$ 
12             remove all points from  $\mathbf{P}$ 
13             while  $n > m$  do
14                 execute lines 28 to 32
15         else if  $\mathbf{p}_{\text{last}} = \mathbf{p}_l$  or  $\text{mbs}(\mathbf{p}_{\text{last}}, \mathbf{p}_l) < C$  then
16              $C \leftarrow C - 1$ 
17             if  $\text{size}(\mathcal{S}) < B - C$  then
18                 append new list of  $\text{mbi}_C(\mathbf{p}_{\text{last}})$  zeros to  $\mathcal{S}$ 
19         else
20             shorten  $\mathcal{S}$  to size  $B - C$ 
21              $k \leftarrow$  index of first  $\mathbf{p}_k$  satisfying  $\text{mbs}(\mathbf{p}_{\text{last}}, \mathbf{p}_k) \geq C$ 
22              $C \leftarrow \text{mbs}(\mathbf{p}_{\text{last}}, \mathbf{p}_k)$ 
23              $z = \text{mbi}_C(\mathbf{p}_k) - \text{mbi}_C(\mathbf{p}_{\text{last}}) - 1$ 
24              $\mathbf{p}_{\text{last}} \leftarrow \mathbf{p}_k$ 
25              $n \leftarrow n + k + 1$ 
26             remove the first  $k + 1$  points from  $\mathbf{P}$ 
27             while  $n - 1 > m$  or  $(B > C$  and  $n > m)$  do
28                 push current node to stack
29                 finalize its children with point counts in  $\mathbf{s}_0$ 
30                  $n \leftarrow n - \sum \mathbf{s}_0$ 
31                  $B \leftarrow B - 1$ 
32                 remove the first list  $\mathbf{s}_0$  from  $\mathcal{S}$ 
33             if  $B > C$  then
34                 append point count  $n - \sum \mathcal{S} - 1$  to last list in  $\mathcal{S}$ 
35                 append  $z$  zeros to last list in  $\mathcal{S}$ 
36             else
37                 finalize current node (point count =  $n - 1$ )
38                 finalize  $C - B$  inner nodes from stack
39                 finalize  $z$  empty nodes
40                  $n \leftarrow 1$ 
41                  $B \leftarrow C$ 
42  finalize current node (point count =  $n$ ) and all of its ancestors

```

points in chunks of arbitrary size. When starting to process a new chunk \mathbf{P} , its points may allow us to finalize the current node. We therefore initialize the current potential node to the current node, i. e., C to B , in Line 7.

Conforming with Kontkanen et al.'s algorithm, we then try to access the m -th point after the current potential node's first one to investigate whether it has to be a leaf or an inner node. Since we have already seen $n - \sum_{i=0}^{B-C-1} \sum s_i$ points in this node, we subtract this amount from m to obtain the index of the point we seek. However, if \mathbf{P} is not large enough, we consider the last point in \mathbf{P} instead. In any case, we save its index to l in Line 9.

As soon as we reach an octree level C small enough to separate \mathbf{p}_{last} from \mathbf{p}_l , we find the number k of points in \mathbf{P} belonging to the current potential node (Line 21). Since $\text{mbs}(\mathbf{p}_{\text{last}}, \mathbf{p})$ grows with the point sequence, we can use a binary search for this step, which is worthwhile especially for large m . During this iteration, we consume all points until \mathbf{p}_k , which is the first one outside the current potential node and thus becomes the new \mathbf{p}_{last} . The absolute level of separation between \mathbf{p}_k and the point preceding it must be the one separating \mathbf{p}_k and \mathbf{p}_{last} . As this will be the level of the current potential node during the next iteration, we set C to it in Line 22. In order to later account for empty nodes in between the one containing \mathbf{p}_{last} and the one containing \mathbf{p}_k , we save their number as z , before updating \mathbf{p}_{last} , n , and \mathbf{P} .

With n increased, we handle provably inner nodes. This is necessary especially because if C was increased in Line 22, we want to unwind the temporary octree to the new level but have to make sure to use its data first. In fact, if C was increased in Line 22, we know that all ancestors of the current potential node must contain more than m points because else we would have found the new \mathbf{p}_{last} already at an earlier iteration with higher C . Hence, we test in Line 27 whether the current node is an inner one. Thereby, \mathbf{p}_{last} is already counted in n but belongs to the current node only if $B > C$. For each such identified inner node, we push it to the stack and finalize as many children of it as we have already temporarily finalized, thus consuming the first list in \mathcal{S} .

Afterwards, \mathcal{S} is nonempty only if we have not seen more than m points in the current node. In this case, C cannot have been increased in Line 22 as pointed out earlier, such that $B > C$ still holds. We thus temporarily finalize the current potential node by appending its point count $n - \sum \mathcal{S} - 1$ to the last list in \mathcal{S} in Line 34, followed by a 0 for each further empty sibling before \mathbf{p}_{last} .

If however $B \leq C$, the current potential node is the current node and C may have increased in Line 22. We therefore finalize the current node and its ancestors from the stack that do not contain \mathbf{p}_{last} . If any of those has still unfinalized children, we finalize them as empty nodes during this step. Afterwards, we finalize empty siblings of the node containing \mathbf{p}_{last} preceding it in Morton order.

When finishing a point chunk \mathbf{P} , our algorithm ensures that we have not found more than m points in the last chunk so far. This is because, after each increase of n , we finalize nodes until $n \leq m$. Therefore, after processing all point chunks, we can simply finalize the current node in Line 42 and all the inner nodes from the stack up to the root node, knowing that any further unfinalized nodes encountered during this step have to be empty leaves.

4. Performance Discussion and Conclusion

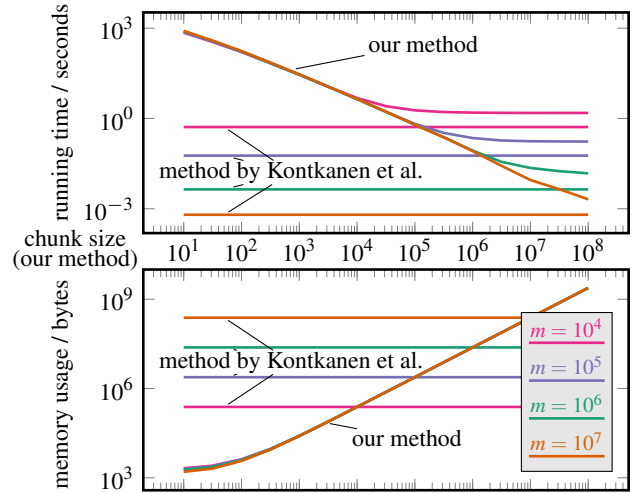


Figure 2: Logarithmic plot of time and memory required by our method and that of Kontkanen et al. for computing an octree containing 10^8 points, for various maximum point counts per leaf m . Computations were conducted on a Lenovo ThinkPad P15v Gen3 with 12th Gen Intel®Core™i7-12700H processor and 32GB RAM.

To compare the performance of our algorithm to that of Kontkanen et al., we implemented both in Python (cf. the source code on OSF [Fis24]). Since both algorithms read over the sorted points in one sweep and write out a stream of octree data to file, their workload for input/output is equal. We therefore focus on their in-core work. We have applied both methods on several sorted streams of 10^8 points residing in RAM and timed their work for $m = 10^4$, 10^5 , 10^6 , and 10^7 , employing various point chunk sized for our algorithm to examine its impact on performance, see Figure 2. For chunk sizes below m , the running time is roughly inversely proportional to the chunk sizes, since we have to build up the temporary octree at the end of every point chunk.

To estimate dynamic memory, we recorded the maximum length of the octree inner node stack and - for our method - of the point count record \mathcal{S} , during each octree construction run. Assuming 128 bytes for holding an inner octree node on the stack, 64 bytes for an element s_i of \mathcal{S} , and 24 bytes for a point in a considered chunk, we then computed the memory estimate as the sum of memory taken by objects of these three kinds. We found space for the points to greatly dominate the other components, causing the memory taken by our method to be roughly proportional to the chunk size.

Naturally, the method by Kontkanen et al. is the better choice if holding m points in memory is not an issue. However, in our use case, combining out-of-core point sorting and octree construction, we are better off assigning most of the available memory to i/o operations and accepting a higher CPU load, as long as the overall performance remains to be by bound by i/o. We have done so in an out-of-core SPH particle sorting stage to build an octree for memory management during a later construction of particle hierarchies for SPH data visualization [FLR15].

References

- [Cha06] CHAN T. M.: A minimalist's implementation of an approximate nearest neighbor algorithm in fixed dimension. Manuscript, 2006. [2](#)
- [CK10] CONNOR M., KUMAR P.: Fast construction of k-nearest neighbor graphs for point clouds. *IEEE transactions on visualization and computer graphics* 16, 4 (2010), 599–608. [1](#), [2](#)
- [Fis24] FISCHER J.: Efficient construction of out-of-core octrees for managing large point sets, Apr 2024. URL: osf.io/cu2g4. [4](#)
- [FLR15] FISCHER J., LINSEN L., ROSENTHAL P.: Error-minimizing sph particle merging for constructing multi-resolution hierarchies. In *Proceedings of the 10th International Smoothed Particle Hydrodynamics European Research Interest Community (SPHERIC) Workshop* (2015). [4](#)
- [KTO11] KONTKANEN J., TABELLION E., OVERBECK R. S.: Coherent out-of-core point-based global illumination. In *Computer Graphics Forum* (2011), vol. 30.4, Wiley Online Library, pp. 1353–1360. [1](#)
- [LMD*11] LINSEN L., MOLCHANOV V., DOBREV P., ROSSWOG S., ROSENTHAL P., LONG T. V.: Smoothviz: Visualization of smoothed particles hydrodynamics data. In *Hydrodynamics*, Schulz H. E., Simoes A. L. A., Lobosco R. J., (Eds.). IntechOpen, Rijeka, 2011, ch. 1. URL: <https://doi.org/10.5772/28338>, doi:10.5772/28338. [1](#)
- [SW97] SALMON J., WARREN M. S.: *Parallel, out-of-core methods for n-body simulation*. Tech. rep., Los Alamos National Lab.(LANL), Los Alamos, NM (United States), 1997. [1](#)