Pacific Graphics 2020
E. Eisemann, A. Jacobson, and F.-L Zhang
(Guest Editors)

*Volume 39 (2020), Number 7*

# Fast Out-of-Core Octree Generation for Massive Point Clouds

Markus Schütz, Stefan Ohrhallinger, Michael Wimmer

TU Wien, Institute of Visual Computing

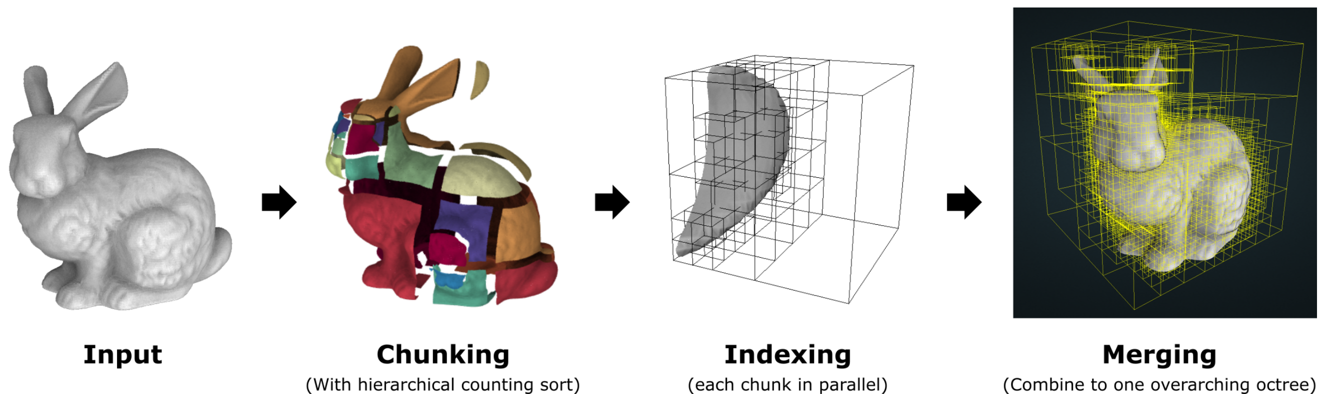



**Figure 1:** *Processing pipeline: An out-of-core hierarchical counting sort quickly generates chunks of suitable size which can then be indexed in parallel, and eventually merged into a single octree.*

**Abstract**

*We propose an efficient out-of-core octree generation method for arbitrarily large point clouds. It utilizes a hierarchical counting sort to quickly split the point cloud into small chunks, which are then processed in parallel. Levels of detail are generated by subsampling the full data set bottom up using one of multiple exchangeable sampling strategies. We introduce a fast hierarchical approximate blue-noise strategy and compare it to a uniform random sampling strategy. The throughput, including out-of-core access to disk, generating the octree, and writing the final result to disk, is about an order of magnitude faster than the state of the art, and reaches up to around 6 million points per second for the blue-noise approach and up to around 9 million points per second for the uniform random approach on modern SSDs.*

## 1. Introduction

Terrestrial laser scanning, photogrammetry, and aerial LIDAR scanning operations yield hundreds of millions to trillions of points nowadays. Due to the large storage and memory requirements, these kinds of data sets need to be processed in an out-of-core fashion, where only a small part of the data is loaded into RAM or GPU memory at any given time. For processing tasks, the point cloud data is often stored such that specific regions can be accessed quickly, e.g., in separate tiles or in hierarchical structures like kd-trees in which all points are stored in leaf nodes [PMOK14; OP]. For real-time rendering, level-of-detail (LOD) structures are required, where lower levels of detail contain representative subsets that give users the impression that they are looking at the full data set, even though only a fraction of it is being loaded and rendered. In this paper, we focus on the fast generation of LOD structures that are suitable for real-time rendering purposes.

The LOD structure that we target with our method is a variation of a layered point cloud [GM04] that uses an octree with an additive scheme, as used by Potree [POT] and Entwine [ENT]. These octree structures populate each node with a subset of the full data set, and the combination of all nodes yields the original data set without duplicates. Additive scheme means that during rendering, higher levels of detail contain additional points that are rendered

© 2020 The Author(s)
Computer Graphics Forum © 2020 The Eurographics Association and John Wiley & Sons Ltd. Published by John Wiley & Sons Ltd.

together with points in lower levels of detail. Alternatively, one could use a replacement scheme where higher levels of detail replace lower levels. The advantage of the replacement approach is that it would allow us to compute representative subsets with baked-in anti-aliasing, similar to mip maps, however we chose the additive approach at this time because it is faster to generate and render, and requires less memory. Figure 2 illustrates individual octree nodes and the subsets of points that are stored inside.

### 1.1. Problem Statement

Unstructured point cloud data exists in various forms that pose different problems and potential solutions to the generation of LOD structures. Large data sets are often a combination of individual scans, for example different flight lines in aerial LIDAR, or scan positions in terrestrial laser scanning. The former is often distributed in the form of non-overlapping tiles that contain all flight lines. The latter is regularly distributed as one file per scan position, which tend to significantly overlap with each other.

In our experience, the following properties of the input data have a significant impact on the LOD generation:

- Overlaps between input files: Non-overlapping files can be converted in parallel, and the results can be merged afterwards.
- Ordering of points: For out-of-core algorithms, a certain locality between points is beneficial since it reduces the need to swap data frequently from external storage. Ideally, each region is only loaded, processed and unloaded once.
- Density distribution: Relatively uniform point densities make it easier to distribute points into sufficiently small leaf nodes of a tree structure, which is advantageous for bottom-up approaches such as ours. Highly uneven distributions on the other hand (teapot in a stadium problem, e.g., a sparsely scanned country-wide data set with a single densely scanned building) may require recursive splitting steps until the points are broken down into sufficiently small chunks.

Our method works with all of the above-mentioned input data, and it does comparatively well with strongly overlapping input files such as individual scan positions of terrestrial laser scanning, as it reorganizes them into chunks anyway. Point clouds with non-uniform point densities benefit from our hierarchical counting sort, which splits a set of points by 9 octree levels with only two iterations over the points, which significantly reduces the amount of potentially required recursions. In fact, none of our test data sets with up to 116 billion points requires more than a single out-of-core counting sort pass.

Our contributions to the state of the art are:

- A hierarchical counting sort that is suitable to partition a point cloud by up to 9 octree levels by looping through all points twice. This counting sort is applied in an out-of-core fashion during the chunking phase in order to generate small point cloud files (~10M points), and is applied again in an in-core fashion during the indexing phase to create small leaf nodes (~10k points) that can then be subsampled bottom up.
- A simple and fast hierarchical approximate blue-noise subsampling algorithm that keeps sampling artifacts across borders of adjacent nodes subtle, even though distances are not enforced across borders.
- An octree generation method that utilizes the bandwidth of modern SSDs, rather than avoiding disk access at all costs.

## 2. Related Work

This section describes three categories of work related to our method: Counting sort, LOD structures for point clouds, and blue-noise sampling.

### 2.1. Counting Sort

Counting sort is an integer sorting algorithm that runs in linear time [Knu98; CLRS01], as opposed to its comparison sort based counterparts with a time complexity of $O(n \log n)$. It is applicable to data sets that are sorted by integer keys within a limited range. The range is limited because counting sort, as the name suggests, counts the amount of each occurring key and it uses an array of counters that is as large as the range of potential keys to do so. Applications of counting sort include point-in-cell simulations [Bow01], computing fixed-radius nearest neighbours for particle simulations [HCR2014], substeps of radix-sort routines, and in our case the block-wise sorting of points. A survey of Arkhipov et al. [AWLR17] discusses sorting algorithms on GPUs, including counting sort as well as radix sort based on counting sort.

We use counting sort to partition a point cloud into smaller chunks by up to 9 octree levels at once, i.e., we do a block-wise rather than a point-wise sorting. We also extend counting sort by a hierarchical component to merge small blocks into larger ones in order to avoid generating a massive amount of tiny chunks.

### 2.2. Point Cloud LOD Structures

QSplat was the first method that uses a hierarchical structure to display large point clouds in real time [RL00]. Dachsbacher et al. introduced sequential point trees (SPT), a similar but more GPU-friendly structure that sequentializes the hierarchy into an array [DVS03]. Points are essentially sorted by level of detail, and the amount of detail can be adjusted by rendering a smaller prefix (subset starting at 0) of the vertex buffer. Gobetti and Marton [GM04] proposed a GPU-friendly as well as view-dependent LOD structure called layered point clouds (LPC). This structure uses a three-dimensional binary tree in which each tree node stores selected samples of the full point cloud. Variations of layered

© 2020 The Author(s)
Computer Graphics Forum © 2020 The Eurographics Association and John Wiley & Sons Ltd.

point clouds with different tree structures and subsampling methods have since become the de-facto standard in state-of-the-art point cloud rendering engines. Further research explores different tree structures (octree, kd-tree with alternating axis, multi-way kd-tree), sampling methods (random, closest to center, Poisson-disk) and octree generation algorithms [WS06; WBB*08; GZPG10; SW11; EBN13; Sch16; Sch14; MVvM*15; Fra17; KJWX19].

PotreeConverter [POT] creates an octree structure that is largely identical to the modifiable nested octree structure [SW11; Sch14], but it uses a different sampling strategy that selects points with a certain minimum distance in each node. This ensures highly uniform subsamples within nodes, but the minimum distance is not enforced between neighboring nodes and parent or child nodes. Entwine creates a similar octree structure but features better parallel processing capabilities *"with parallelization in the cloud in mind"* [ENT; ENT2019]. Input files can be converted to an octree individually and in parallel, and then merged into a single complete Entwine Point Tile. Entwine uses a sampling strategy that selects the point that is closest to the center of a grid cell, which gives deterministic results and ensures some level of uniform coverage. Kang et al. propose an in-core approach where one picks a random point per cell within an inscribed sampling grid in order to efficiently generate an octree structure with a certain level of uniformity [KJWX19]. We use the same sampling strategy as one of two options, but our method extends to arbitrarily large point clouds and achieves a multiple times higher throughput even in cases where the point cloud would be small enough to be processed in-core.

We would like to mention the research of Martinez et al. [MVvM*15] as particularly relevant. They create an octree for 638 billion points by splitting the data set into 16 x 16 tiles and then executing PotreeConverter for each tile simultaneously on a server system with 2 x 8 cores and on a distributed supercomputer. Afterwards, the individually generated octrees are merged into a single one. The resulting runtime is 15 days, which corresponds to a throughput of around 0.49 million points per second. Our approach also splits the point cloud first and merges the results afterwards, but instead of splitting on a 2-dimensional $16^2$ grid, which corresponds to 4 quadtree levels, we split on a three-dimensional $512^3$ grid, which corresponds to 9 octree levels. In addition to that, our evaluation runs on a single-CPU system instead of a dual-CPU server system plus a supercomputer. We were only able to evaluate our approach for up to 100 billion points, however, due to lack of disk space. Similar in spirit to tiling the data first, Leimer and Scheiblauer [Lei13; Sch14] sort the data first in order to optimize the input for subsequent indexing operations.

Wand et al. [WBB*08], Scheiblauer [Sch14], and Kang et al. [KJWX19] report **in-core performances** of 0.3, 1.21, and 2.5 million points per second, respectively. Scheiblauer [Sch14], Richter and Döllner [RD10], Goswami et al. [GZPG10], Dieckmann and Klein [DK18], Martinez et al. [MVvM*15], and Discher et al. [DRD18] report **out-of-core performances** of around 0.49, 0.1, 0.6, 0.2, 0.49 and 1.35 million points per second, respectively.

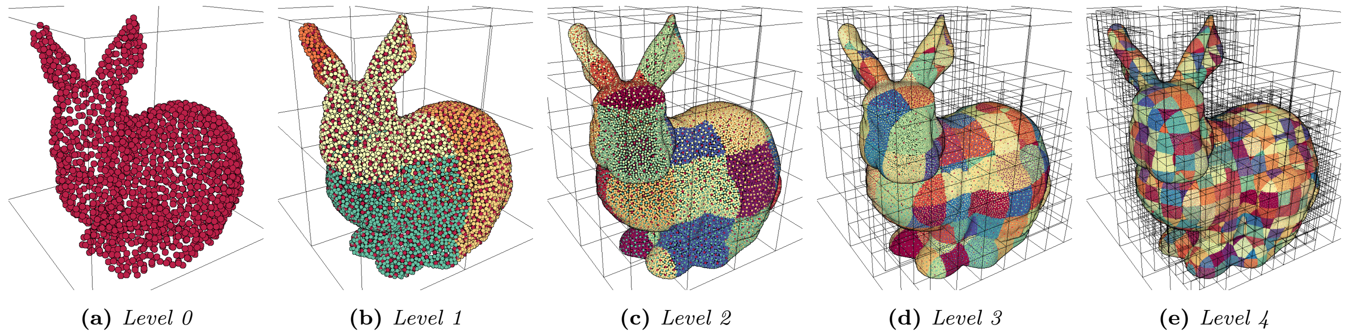
### 2.3. Blue-Noise Sampling

In the context of two or three-dimensional point samples, blue-noise sampling is characterized by point sets with a certain minimum distance between adjacent points, but also a lack of large gaps and regular sampling patterns. Point sets with blue noise characteristics are considered to be of high visual quality, and many papers explore strategies to generate samples with various applications and different levels of quality and performance. Cook proposes Poisson-disk sampling or jittering on a regular grid as two methods to generate such point sets [Coo86]. The majority of research on blue-noise sampling deals with the generation of samples at suitable locations, and we refer to Yan et al. [YGW*15] for an extensive survey of sampling methods. The following methods are closely related to our research because they either subsample a given set of points or produce hierarchical representations of three-dimensional point clouds by generating samples on a mesh. Yuksel describes sample elimination, i.e., subsampling, as a way to reduce a large number of points to a smaller set with blue-noise characteristics [Yuk15]. Dieckmann and Klein generate additive hierarchical Poisson-disk sets top-down by recursively trying to add points into octree nodes, and if they do not meet the minimum distance requirements, trying again in the next level of the octree [DK18]. This results in an octree structure similar to ours, but our approach differs in that it sorts the points first, does distance checks to the last few previously accepted samples, and also in a bottom-up approach, which allows us to operate in parallel right from the start. Brandt et al. [BJFadH19] compute a progressive point cloud with blue-noise characteristics from meshes on the GPU. Progressive here means that any prefix (subset from the beginning) still exhibits blue-noise characteristics, and the size of the prefix can be adjusted based on the distance to the object, similar to sequential point trees [DVS03].

## 3. Data Structure

This section describes the generated data structure and its representation on disk. We generate a layered point cloud in an octree, largely identical to the modifiable nested octree used by Scheiblauer [SW11] and Potree [Sch16]. Figure 2 illustrates the tree structure and the contents in its nodes. The root node of the octree contains a coarse subsample that represents the whole data set at a low level of detail. With each level, nodes contain increasingly higher resolution subsamples of their respective regions. One of the main issues of modifiable nested octrees is that previous work generates one file per octree node, e.g., Martinez et al. [MVvM*15], who use Potree for their work, end up with 38 million files for a point cloud of 638 billion points. Each individual file adds significant overhead to file system operations such as accessing, copying, deletion, uploading to a server, etc., thereby

© 2020 The Author(s)
Computer Graphics Forum © 2020 The Eurographics Association and John Wiley & Sons Ltd.



(a) *Level 0* (b) *Level 1* (c) *Level 2* (d) *Level 3* (e) *Level 4*

**Figure 2:** *Each octree node holds a subsample of the full point cloud. The root node contains a coarse low density subsample of the whole data set and with each level, the resolution is doubled. Points are colored by the node they belong to. Points in lower levels of detail are rendered together with points in higher levels of detail, as seen in (b) through the mixture of red points from the root and other colors from the respective nodes at level 1.*

increasing times for the respective operations from seconds and minutes to hours and days.

Our storage format differs in that we generate a single file for all points (octree.bin), a second file for the whole octree hierarchy (hierarchy.bin), and a third file with additional metadata (metadata.json). The octree.bin file contains all the points grouped by nodes in no particular order. Leaf nodes tend to be stored at the beginning of the file because we process the octree from the bottom up. Only the root node is guaranteed to be the last node inside the file. The hierarchy.bin file contains the full octree hierarchy, including location and size of each node inside the octree.bin file. Since the octree hierarchy itself can grow quite large, we group it into 4 levels, which allows us to quickly load only the parts of the hierarchy that are needed. Four levels amount to about 256 additional child nodes, assuming that an average of 4 direct child nodes exist for each node. When we load the hierarchy of the root node, we only get the first 4 levels. Once we reach nodes at the fourth level, we can load the next 4 levels for each node as required. The third file, metadata.json, contains information such as bounding box and point attributes that are required to load and decode the point data that is stored in octree.bin.

## 4. Method

Within this paper, we differentiate between *local* and *global* octrees. Local octrees are generated separately for each individual chunk of the point cloud, and the global octree is the result of eventually merging all local octrees into a single octree containing the entire point cloud.

Our method consists of the following steps (see Figure 1):

1. Chunking: Split point cloud into chunk files with up to around 10 million points.
2. Indexing: Build local octrees out of each chunk in parallel.
3. Merging: Combine all local octrees into a single global octree.

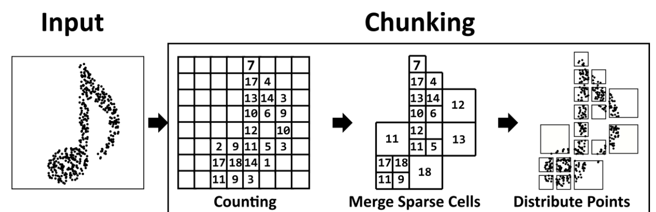


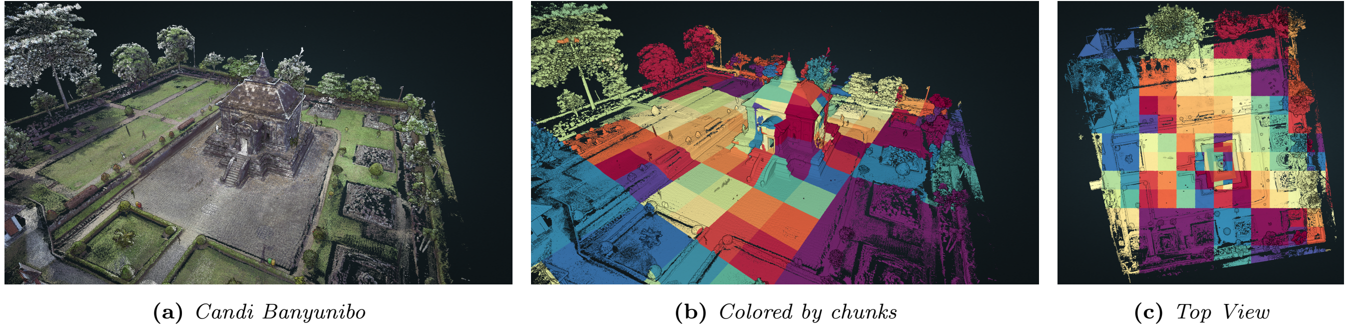

**Figure 3:** *Chunking uses a hierarchical counting sort in order to partition the points by multiple octree levels at once.* ***Counting:*** *Count number of points that fall into cells of a high-resolution grid.* ***Merge Sparse Cells:*** *Any 8 adjacent and octree-aligned cells with less than a certain number of combined points are merged into larger cells.* ***Distribute Points:*** *Now that we know the location, extent, and the number of points in each chunk, we loop through all points again and directly transfer each point to its respective chunk / file.*

### 4.1. Chunking

The chunking phase splits the point cloud into cubic chunks (e.g., files in out-of-core storage) that are small enough so that multiple chunks can be processed in parallel, but large enough to avoid a massive amount of tiny chunks. It is done using a hierarchical variation of counting sort, as shown in Figure 3. Counting sort is particularly useful for this task because it requires only two streaming passes: the first pass for counting points in a $2^{9^3} = 512^3$ grid (for an outer octree of 9 levels), and a second pass to distribute the points to the respective chunks. In more detail, the approach consists of the following steps:

**Counting** First, we divide the cubic bounding box of our point cloud into a 3-dimensional grid with $2^{depth}$ cells on each axis. Each cell represents a counter for all points inside it. A size to the power of two is required to align the counting grid with an octree. *Depth* specifies the depth of the octree. The depth and therefore the resolution of this grid

© 2020 The Author(s)
Computer Graphics Forum © 2020 The Eurographics Association and John Wiley & Sons Ltd.



**(a)** *Candi Banyunibo* **(b)** *Colored by chunks* **(c)** *Top View*

**Figure 4:** *(b+c) Each generated chunk (=file) rendered with a random color. The top view shows how more densely scanned regions are partitioned into smaller chunks, while sparsely sampled regions are merged into larger chunks before writing them to a file.*

defines the smallest possible extent of the chunks we are about to generate. The generated chunks should be small enough so that multiple chunks can fit in memory and be processed simultaneously by the indexing phase following later on. Larger point clouds will require a higher grid resolution so that the resulting chunks are sufficiently small. In our implementation, we use grid sizes of 128, 256 and 512 on each axis. A counter grid using 32 bit integers with a size of 128 requires $4 * 128^3 = 8$ MB memory, which constitutes a low memory footprint, is quickly allocated, and also quickly processed by the merging phase. A size of 512 requires 536 MB and already adds significant processing overhead. We suggest 128 for point clouds with less than about 100 million points, 512 for point clouds with more than 500 million points, and 256 in between. A grid size of 512 can accommodate point clouds with relatively uniform scan densities with hundreds of billions of points (e.g., aerial LIDAR, 116 billion points, see Figure 10) but also scans with strongly varying point densities (e.g., terrestrial laser scans) with a few billion points. The latter may lead to “teapot in a stadium" scenarios that result in chunks that are larger than desired because the counting grid cells are not small enough to split up the small "high-resolution teapot". In these cases, we suggest to recursively run the chunking process again on all chunks that are too large, e.g., larger than 500MB. Chunks should be small enough so that a total of at least $2 * numThreads * chunkSize$ RAM is available. However, we did not need to recursively split chunks for any of our test data sets benchmarked in Section 7.

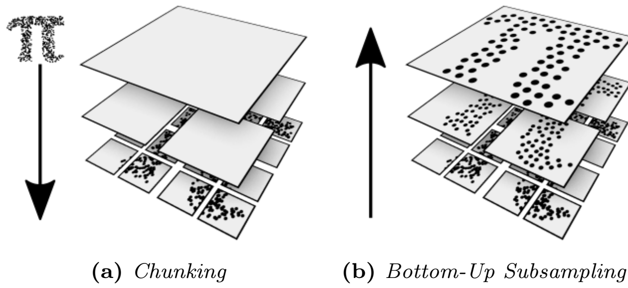
**Merging Sparse Cells** After we have counted the number of points in each cell, we recursively merge smaller cells that are sparse enough into bigger ones. Merging is implemented in a fashion similar to creating image pyramids or mip maps. We iterate over groups of 2x2x2 cells inside the counting grid (=highest level of the pyramid, where highest means most detailed), and if the sum is less than a threshold (e.g., 10 million points), we store the sum inside the next lower (less detailed) level of the pyramid. If the sum is higher than the threshold, we add the position and the level of all entries that are larger than zero to the list of chunks, and store $-1$ inside the next lower level of the pyramid, indicating that this region already contains finalized chunks and thereby marking it as unmergeable. Any 2x2x2 group of cells with at least one cell marked as unmergeable is treated as if the sum was larger than the threshold, i.e., all entries larger than zero are added to the list of chunks. This process is repeated recursively all the way up to level 0 of the pyramid.

**Create Chunk Lookup Table** After computing the list of chunks, we create a lookup table (LUT) with the same size as the counting grid and with pointers from the grid cells to the respective chunks. During the distribution phase, this LUT allows us to identify the target chunk for each point with a single lookup. It is populated by iterating through each chunk and then setting the pointer values of all covered cells to the respective chunk.

**Distributing** In the final step of the chunking phase, we iterate over all points again and project them to a cell as we did during the counting phase, but this time we access the cell of the LUT to retrieve a pointer to the target chunk and subsequently the file that this point will be written to.

The result of the chunking phase is a collection of files containing cubic chunks of points that do not overlap each other, align with a node in the global octree, and which can therefore be indexed simultaneously and trivially merged afterwards. The level and coordinate of each chunk inside the global octree is encoded in the filename. Each chunk starts with "r", followed by one number between 0 to 7 per level that indicates the index of the child node that we are traversing into. The index is a bit mask that represents the x,y and z coordinate of the child. The leftmost of the three bits stands for the x coordinate, the middle one for the y coordinate, and the rightmost one for the z coordinate. For example, index 5 = the sixth child = bitmask 0b101 = child coordinate x: 1, y: 0, z: 1. File "r063" represents a chunk at level 3 of the octree and we reach its location by first traversing from the root through child nodes 0, then 6, and then 3.

© 2020 The Author(s)
Computer Graphics Forum © 2020 The Eurographics Association and John Wiley & Sons Ltd.



**(a)** *Chunking* **(b)** *Bottom-Up Subsampling*

**Figure 5:** *Indexing: Create a multi-resolution octree for each generated chunk. (a) First, points in a chunk are partitioned into leaf nodes with a specified maximum number of points, using the same hierarchical counting sort procedure that was used to generate the chunk files. (b) Coarser levels of detail are then populated by recursively subsampling child nodes from the bottom up.*

### 4.2. Indexing

In this step, the previously generated chunks are loaded from disk and converted into local octrees in parallel using one thread per chunk. Points are first partitioned into leaf nodes and coarser levels of detail are populated by recursively extracting subsamples of higher levels of detail from the bottom up. The subsampling strategy is exchangeable and we describe two example implementations in further detail in section 5.

Building an octree out of a chunk is done in a bottom-up fashion, as shown in Figure 5. The points are first partitioned into leaf nodes with a certain maximum size using largely the same hierarchical counting sort approach as during the chunking step, but this time it is applied in-core. We suggest a maximum size of 10k points per leaf node to obtain a sufficiently fine grained LOD representation for real-time rendering purposes. Since there are multiple threads working on different chunks simultaneously, and because each thread reuses and resets the counting grid from one chunk to the next, we have to use lower resolution counting grids compared to sizes of up to 512 during chunking. In our implementation, we use counting grid sizes of 32 during the indexing phase, which corresponds to partitioning the points by 5 octree levels. This is often not enough to obtain leaf nodes with a maximum size of 10k points, so we recursively split all nodes that are still too large by another 5 octree levels until they are sufficiently small. Massive amounts of tiny nodes are also avoided the same way as during the chunking phase by merging leaf node candidates that have less than 10k points, combined. The hierarchical counting sort procedure produces a list of leaf nodes containing arrays of points. We then create the missing inner nodes between the local octree root and the computed leaf nodes to obtain an octree where only leaf nodes are populated, as shown in Figure 5a.

Coarser levels of detail are populated by recursively extracting samples out of finer levels of detail from the bottom up, as shown in Figure 5b. Bottom-up traversal is implemented as a post-order depth-first octree traversal. If a visited node is a leaf node, we ignore it. Otherwise, we apply one of the subsampling strategies described in section 5 with the points of all direct child nodes as the input. The accepted subsample is stored in the current node and the remaining points are transferred back to the child nodes. At this point, the child nodes are complete and no longer needed for further processing, so we flush them to a single output file (octree.bin). The only information we keep in memory is the octree hierarchy, including the location and size of a flushed node inside the output file, which is necessary because multiple simultaneously processed chunks flush nodes to a single output file in no particular order. During rendering, we will need the location and size of each node to load the right range of data from the file.

### 4.3. Merging

Each processed chunk represents a local octree at its respective location. Whenever a chunk has been fully processed, its root node is linked to the global octree. Once all chunks have been finished, the global octree consisting of all the chunk root nodes is subsampled bottom-up in the same way as the individual chunks during the indexing phase. After all nodes up to the root node have been written to octree.bin, we generate the hierarchy.bin and metadata.json files. The hierarchy is grouped into batches of 4 levels so that we can load parts of the hierarchy as needed. The first batch contains the first 4 levels of hierarchy for the root node. The next sets of batches contain additional 4 levels of hierarchy for all the nodes at the fourth level of the octree, nodes at the eigth level of the octree, etc. During rendering, this reduces the amount of hierarchy data that has to be loaded – from potentially hundreds of megabytes to a few hundred kilobytes initially and a few megabytes during ongoing traversal through the scene.

## 5. Subsampling

The subsampling strategy used during the indexing step in section 4.2 is, with some limitations, exchangeable. We implemented and evaluated two approaches, a fast random sampling strategy with a certain level of uniformity as described by Kang et al. [KJWX19], and an approximate Poisson-disk sampling strategy [Coo86; YGW*15] that enforces a minimum distance between points except between adjacent octree nodes. For each node, the input to the sampling method consists of points in its direct child nodes. The result is a subset of points that will be stored inside the current node, and the remaining points are transferred back to the respective child nodes. This process is repeated from the bottom up until all nodes up to the root node are populated with points. A limitation of this approach is that subsampling strategies do not have access to points in adjacent nodes, which can lead to noticeable sampling patterns along borders of two nodes. An advantage of bottom-up approaches is that the number of points to be subsampled quickly diminishes with each level. Martinez

© 2020 The Author(s)
Computer Graphics Forum © 2020 The Eurographics Association and John Wiley & Sons Ltd.

et al. [MVvM*15] found that a comparatively flat country-wide LIDAR scan of the Netherlands has a reduction factor of about 4, meaning that each coarser level of detail has only a fourth of the points left. Once we have subsampled the bottom-most level, we only need to do a quarter of the work for the next level, unlike top-down approaches where the majority of the points need to apply subsampling procedures at all levels of the hierarchy until they reach the bottom.

The random sampling approach suggested by Kang et al. [KJWX19] uses a uniform grid of $128^3$ cells, which leads to node sizes of the order of 10k points. These are not too small to cause a severe overhead of managing numerous tiny batches, but also not too large, so that frustum culling is able to discard a sufficient amount of points. Points are projected into the cells of the sampling grid, and for each cell, one random point is selected as a subsample for the current node. This approach is simple and fast, and the selection on a grid also ensures that the subsample adequately covers the full model with neither excessive clustering nor holes. We pick one random point per cell by first shuffling the input (using the standard C++ std::shuffle), and then accepting the first point in each cell of the sampling grid.

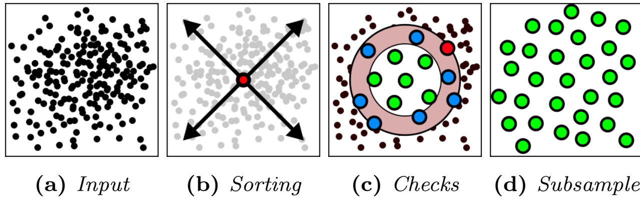
The Poisson-disk approach attempts to generate subsamples with visually pleasing blue-noise properties. It accepts points with a certain minimum distance to previously accepted points, and rejects them otherwise. This method is usually relatively slow due to the required distance checks, but the results are generally considered to be of higher quality. A naive approach to Poisson-disk sampling would iterate over a list of points and accept a new point if it is far enough away from all previously accepted points. Since most of the generated nodes contain around 10k to 50k points, this naive approach would lead to tens of thousands of distance checks per point on average. We reduce the computation time by sorting the points from inside out first, which implicitly gives us a spatial acceleration structure that restricts the search radius to a spherical shell with a thickness equal to the minimum distance. Sorting also packs the samples close together and eliminates the possibility of ridges that appear in Potree and Arena4D, as shown in Figure 7. These ridges appear when candidates are evaluated in an unfavourable order, e.g., lines of points with line 1 then 4, but it turns out line 3 was also far enough away from line 1 but it cannot be accepted anymore because we already accepted line 4, which is too close to line 3. Figure 6 illustrates the steps of our Poisson-Disk approach: First, the input samples are sorted inside-out by their distance to the center of the node. Then, for each input point we check the distances to previously accepted points in an outside-in order by looping through the list of accepted points, which is implicitly ordered inside-out, from the end. If the difference between $distance(center, candidate)$ and $distance(center, acceptedPoint)$ is larger than the minimum distance, we can safely accept the current candidate because all the other previously accepted points are even closer to the center and cannot be closer than minimum distance. Figure 6c shows that this corresponds to evaluating distance checks to previously accepted points within a ring (spherical shell in 3D). This approach works well in practice for two reasons: First, because the amount of input data – the points from all direct child nodes – is in the order of only 10k to 50k points. And second, because most point cloud data sets represent surfaces rather than volume data, so the amount of hit tests against previously accepted points inside the ring (spherical shell) is considerably lower than it would be with volume data sets. In case of volume data, we expect that our approach would need to be extended with an additional spatial acceleration structure that further restricts the search area.

Figure 7 shows the sampling patterns and artifacts of Potree, Entwine, Arena4D and our two strategies. Potree uses a form of Poisson-disk sampling within nodes, but the combination of nodes do not honor the minimum distance requirements. Furthermore, Potree evaluates points in the order in which they are stored on disk, and if the order is not favorable (previously accepted points block more suitable candidates that appear later in the list, e.g., data set CA13), sampling artifacts that manifest as ridges can appear ( [POT], p. 21). Arena4D produces similar sampling patterns and ridges, which leads us to believe that it is also affected by the order of the input. Entwine selects the points that are closest to the center of the sampling grid cells, which increases the average spacing between points and leads to a single deterministic subsampling result, independent of the order of the input. However, the resulting patterns have a noticeable regularity. The samples between cells appear farther apart than samples within a cell, which also produces noticeable but predictable and arguably less visually disturbing ridges. Our random sampling approach (after Kang et al.) shows results that are relatively similar to that of Potree in many cases. It does not suffer from ridges, but it can produce artifacts along slopes and smooth surfaces that manifest as denser lines that look similar to staircasing artifacts or contour lines. Our Poisson-disk approach produces high-quality results that honor the required minimum distances between overlapping nodes of different levels of detail. Although it does not enforce minimum distances between adjacent nodes, the distances end up sufficiently large in most cases due to Poisson-disk sampling from the inside out. By the time we evaluate candidates at the border, we already accepted points further inside, which reduces the chance that points closer to the border get accepted. It does not eliminate the possibility, however, so noticeable gaps or clusters at the border are possible, but less common and more subtle. Figure 11 illustrates the differences between random and uniform random selection, and it shows the impact of the sampling order on our Poisson-disk sampling approach.

## 6. Implementation Details

In this section, we describe implementation-specific details that are essential for our method and performance results

© 2020 The Author(s)
Computer Graphics Forum © 2020 The Eurographics Association and John Wiley & Sons Ltd.



**(a)** *Input* **(b)** *Sorting* **(c)** *Checks* **(d)** *Subsample*

**Figure 6:** *Our Poisson-disk sampling approach.* ***(b)*** *Sort points from inside out.* ***(c)*** *Loop through candidates from inside-out, run distance checks between candidate (red) to potential conflicts (blue) within ring as thick as the desired minimum distance.*

but do not fit into the more abstract description of our method in section 4. Our method was implemented and tested using C++.

**Parallel Counting Sort**: Counting is done in parallel using a grid of 32 bit atomic integers. During the chunking phase, multiple threads read and process different parts of the point cloud and increment the counters concurrently. Likewise during the distribution phase, we load points and write them to the chunk files in parallel.

**Sorting**: We use the parallel versions of standard C++ std::sort in our poisson-disk sampling method. Due to this, part of the indexing process is handled by multiple threads even though we only spawn one thread per chunk ourselves.

**Writing nodes**: Section 4.2 describes that all chunks are loaded and processed in parallel by multiple threads. Each thread first loads a chunk and eventually writes the results to disk, node by node. However, individually writing a large number of small nodes to disk is not efficient. Instead, we use a custom buffered writer object that collects and stores data from finished nodes in buffers of 16MB. If a newly finished node does not fit into the current buffer, the writer will flush the current buffer to disk in a dedicated thread, and simultaneously start building the next buffer. We also stop loading new chunks when the total backlog of the buffered writer exceeds 1GB, because sometimes loading and processing multiple chunks is faster than writing the results to a single file on disk.

## 7. Performance

We compare the performance of our octree generation method (random and Poisson-disk sampling) to the state-of-the-art software packages PotreeConverter, Entwine, and Arena4D. All methods generate some form of layered point cloud where each node is populated with subsamples of the original point cloud. At the start of each individual benchmark, we first empty the operating system cache of Microsoft Windows 10 using RamMap's "Empty Standby List" option. Otherwise, Windows would automatically keep previously accessed files in RAM for faster access, thereby distorting the results.

Rendering benchmarks are omitted because no change in rendering performance is expected. The generated level of detail structures are the same as the modifiable nested octree [Sch14] and Potree, with the only differences being the way they are stored on disk and the point sampling patterns due to different sampling strategies. The amount of points per octree node varies depending on the sampling strategy, but the variance is minor because all sampling strategies target the same point density.

Our test system consists of Windows 10, an AMD Ryzen 2700X (8 cores), 32GB RAM, a 1TB Samsung 970 PRO SSD and an 8TB WD8004FRYZ (7200RPM) HDD.

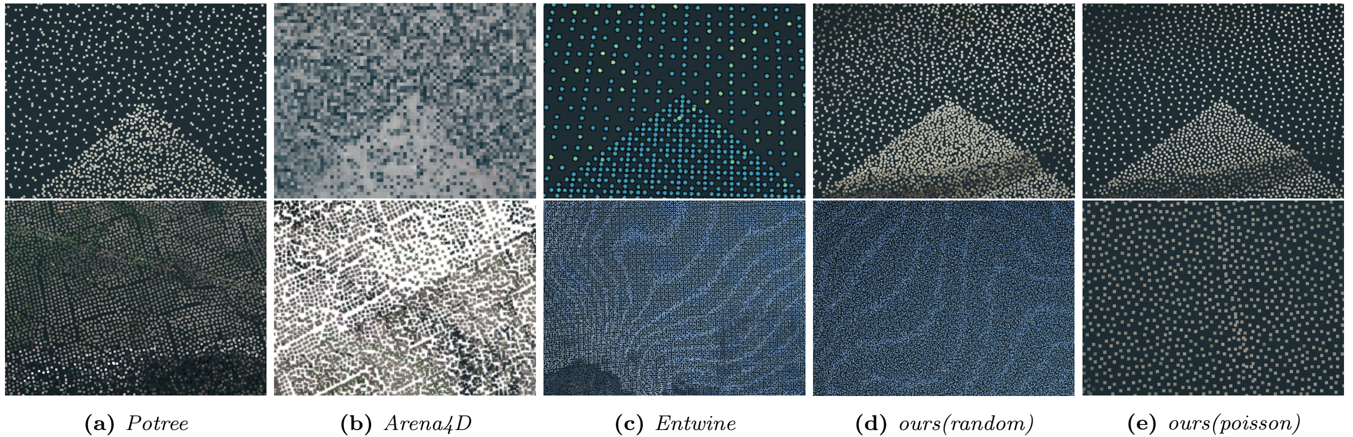
Table 2 lists benchmark results of our method for various data sets, evaluated on both, HDD and SSD. The final merging step is not listed because the majority of the time is spent on creating the chunks and then indexing the chunks in parallel but it is included in the total. Table 3 and Figure 9 compare the LOD generation times of our method to Potree, Entwine and Arena4D. The results show that on SSDs, our method is about an order of magnitude faster than these three packages. The difference is less extreme on HDDs, which indicates that our method is the only one that efficiently utilizes the higher bandwidth of SSDs. A notable observation are the results for the Eclepens data set, which show a significantly lower throughput on Potree and Entwine. This is because the points in this data set exhibit poor locality and as a result, the top-down approach of Potree frequently flushes but then reloads data because processing points at any stage requires distance checks to points from any previous stage. Our method does not suffer from this issue because it first groups points into chunks and once a chunk is processed, its points are not needed anymore at later stages. However, if the point cloud is sorted by morton order, Potree and Entwine become faster by a factor of 3 to 4, as opposed to Arena4D and our approach that do not benefit from sorted input.

### 7.1. Case Study: AHN3

Our largest test data set, AHN3, contains 116 billion points. It is a subset of an even larger scan of the whole netherlands [1,2] but we clipped it due to lack of disk space on our test system. Figure 8 shows different viewpoints of the data set. The input consists of 216 LAZ compressed point cloud files with a total of 531 GB. The outputs comprises a 3.2 TB file with uncompressed point data and 962 MB for the hierarchy data. The latter substantiates the importance of splitting the hierarchy into chunks that can be loaded on demand. A total of 22 778 chunks were created during the chunking phase. Figure 10 shows a histogram of the storage sizes of the chunks – all of them small enough to load and process 16 at a time in system memory, but only 24 or 0.1% of them are what we would consider too small with potentially negative impact due to overhead. Although the largest chunk (307MB) contains 10.9 million points (900k above the specified threshold)

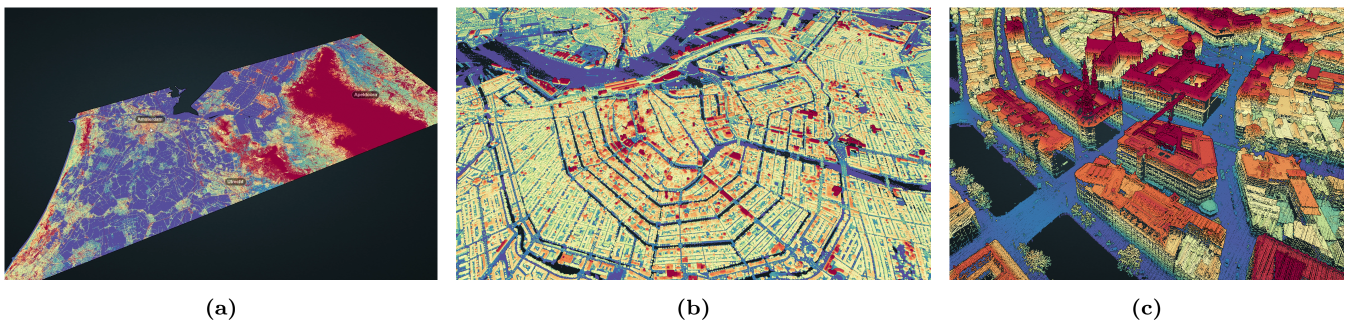
[1] https://downloads.pdok.nl/ahn3-downloadpage/
[2] https://www.ahn.nl/

© 2020 The Author(s)
Computer Graphics Forum © 2020 The Eurographics Association and John Wiley & Sons Ltd.



**(a)** *Potree* **(b)** *Arena4D* **(c)** *Entwine* **(d)** *ours(random)* **(e)** *ours(poisson)*

**Figure 7:** *Comparison of sampling patterns and artifacts. Top row: Sampling patterns at the border of two different levels of detail. These are often visible with low levels of detail settings or while waiting for higher levels of detail to be loaded. Bottom row: Most commonly encountered sampling artifacts. (a) Potree has noticeable clustering along borders of nodes, as well as ridges if the input is ordered unfavourably like in data set CA13. (b) Arena4D also shows ridges with some data sets. A detailed study and more faithful close-up screenshot was not possible because the software is closed and offers no option to render at very low levels of detail. (c) Entwine exhibits regular grid patterns. Staircasing artifacts are common along slopes or curved surfaces. (d) Our random approach looks similar to the results of Potree. It does not suffer from ridges or clustering at borders, but it shows a similar kind of staircasing artifacts on slopes and curved surfaces as Entwine. (e) Our Poisson-Disk approach shows uniform distances between points with no regularity. Points at borders of adjacent nodes can be too close or too far apart, but both cases are relatively rare and subtle due to the inside-out subsampling order.*



**(a)** **(b)** **(c)**

**Figure 8:** *AHN3 subset with 116 billion points (531GB compressed). (a) View of Amsterdam, Utrecht and Apeldoorn. (b) Downtown Amsterdam. (c) Closeup of Amsterdam.*

we refrain from recursively splitting it further because it is still small enough to be processed alongside other chunks.

It took a total of 35 hours to build the octree on an 8TB HDD drive – a throughput of 0.92 million points per second. Counting took 1h24m, chunking 8 hours, and indexing 25h32m. This was below our expectations, especially the indexing phase that started out with a rate of 4 million points per second during the first 10%, but ultimately dropped to 1 million points per second during the last 10%. From a purely algorithmic viewpoint this should not happen because all chunks are processed in parallel fully independently of each other, so we suspect either an implementation or hardware issue that will be investigated in the future. Compared to the state of the art, Martinez et al. [MVvM*15] in particular, our system still manages to achieve roughly double the throughput on a single CPU desktop system using a single HDD, instead of a combination of a dual-CPU server and a supercomputer cluster with roughly 200 dual-quad-core CPUs, of which an unspecified amount was used.

## 8. Problematic / Failure Cases

We identified following potential failure cases after our users evaluated the converter with their data sets.

One of our users used the converter to build an octree out of a synthetic data set of a cube made of $300^3$ points, which effectively represents a voxel data set stored as a point cloud. The Poisson-disk sampling strategy is currently not

© 2020 The Author(s)
Computer Graphics Forum © 2020 The Eurographics Association and John Wiley & Sons Ltd.

| Data Set | #Points | File Size | #Files |
|---|---|---|---|
| Eclepens | 68.7 M | 1.8 GB | 1 |
| Matterhorn | 274.9 M | 2 GB | 1 |
| Affandi | 2.7 B | 20 GB | 147 |
| CA13 | 17.7 B | 84 GB | 2336 |
| AHN3 | 116 B | 531 GB | 216 |

**Table 1:** *List of test data sets. Eclepens is stored in an uncompressed LAS file, all other data sets are in an compressed LAZ format. CA13 and AHN3 comprise of non-overlapping tiles, while Affandi consists of strongly overlapping single scan positions.*

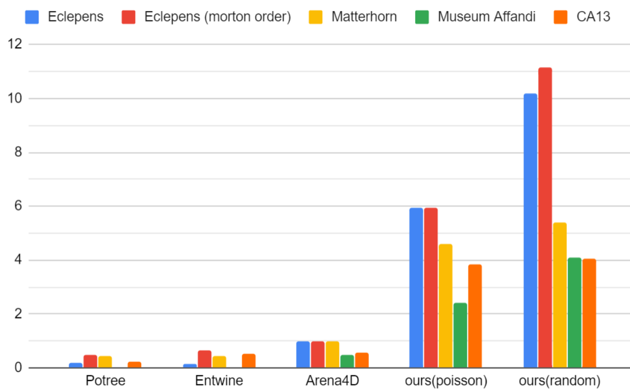



**Figure 9:** *Throughput of various approaches on an SSD in million points per second (higher is better).*

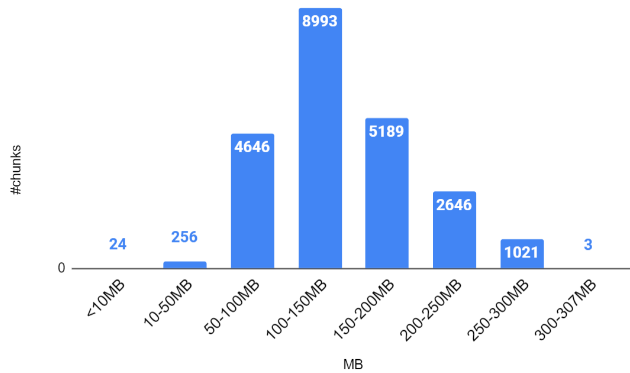



**Figure 10:** *Histogram showing the amount of generated chunks with certain storage sizes for the 116 billion points AHN3 data set and with a counting grid size of* $512^3$.

able to deal with such data sets, only the uniform random sampling strategy was able to successfully generate the LOD structure. Even so, the structure itself is not suitable for volumetric data sets. The resulting nodes contain about 2 million points and all of them store the point coordinates. Voxel structures would be more suitable because they do not need to explicitly store the coordinates for volumetric data sets, and because the efficient rendering of volume data sets using voxels is a well researched topic.

Another unexpectedly common failure case are data sets with a large amount of duplicates. The indexing step uses counting sort to partition points into leaf nodes that contain no more than $X$ points. However, if there are more than $X$ points at the exact same position, then our implementation kept recursing with no progress until the converter crashes. Different users had data sets with tens of thousands and up to 40 million duplicate points, but neither we nor our users could explain where they came from and if they served a purpose. We plan to address this issue by automatically removing duplicates if their number exceeds the maximum number of points per leaf node - something that only needs to be explicitly checked if the counting sort step inserted all points into a single node.

A third failure case was a point cloud with a bounding box that was a thousand times larger than it should have been. Nearly all of the points were within a region of about 400 meters, but the bounding box had an extent of 780 kilometers due to 6 outliers. This is problematic for two reasons: First, the initial chunking step will create just 2 chunks, one with the 6 outliers and another chunk with all the remaining points. It would therefore need to run again to split up the large chunk. The second issue is that the octree spans the whole bounding box. If the bounding box is 1024 times larger than the data it represents, then the octree will have $log_2 1024 = 10$ additional octree levels that serve no purpose but negatively affect rendering performance - Especially with features such as an addaptive point size shader that does an octree traversal for each point inside the vertex shader.

## 9. Conclusion and Future Work

We have shown that it is possible to generate LOD structures for large point clouds at rates of up to eleven million points per second in an out-of-core fashion with a random sampling method, or up to six million points per second with a high-quality approximate Poisson disk sampling method. This is achieved by splitting the data set into small chunks, generating octrees out of each chunk in parallel, and then merging the result into a single overarching octree.

In the future, we would like to attempt a massively parallel approach that generates the LOD structure on the GPU. An in-core version of a GPGPU LOD generation algorithm may be especially useful as a webbrowser-based application that allows users to drag & drop a point cloud into the browser window, which is then organized into an LOD structure as fast as the browser can load the data from disk. For

© 2020 The Author(s)
Computer Graphics Forum © 2020 The Eurographics Association and John Wiley & Sons Ltd.

| | Data Set | SSD count | SSD distribute | SSD index | SSD total | SSD Points / s | HDD count | HDD distribute | HDD index | HDD total | HDD Points / s |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Poisson-Disk | Eclepens | 0.6s | 1.6s | 9.2s | 11.6s | 5.9M | 18.7s | 1.8s | 10.7s | 31.2s | 2.2M |
| | Eclepens (morton) | 0.6s | 1.3s | 9.4s | 11.5s | 5.9M | 15.7s | 2.2s | 9.5s | 27.4s | 2.5M |
| | Matterhorn | 12.7s | 16.8s | 30.1s | 59.8s | 4.6M | 24.4s | 35.0s | 36.7s | 96.2s | 2.9M |
| | Museum Affandi | 2m 49s | 3m 57s | 11m 38s | 18m 25s | 2.4M | 2m 57s | 17m 14s | 18m 2s | 38m 18s | 1.2M |
| | CA13 | 14m 37s | 19m 20s | 42m 53s | 1h 17m | 3.8M | 14m 17s | 1h 30m | 1h 38m | 3h 24m | 1.4M |
| | AHN3 | - | - | - | - | - | 1h 24m | 7h 59m | 25h 32m | 34h 58m | 0.92M |
| Random | Eclepens | 0.9s | 1.8s | 4.0s | 6.7s | 10.2M | 19.4s | 2.1s | 5.6s | 27.1s | 2.5M |
| | Eclepens (morton) | 0.8s | 1.5s | 3.9s | 6.2s | 11.2M | 16.3s | 2.0s | 5.6s | 24.0s | 2.9M |
| | Matterhorn | 13.8s | 18.2s | 19.1s | 51.1s | 5.4M | 23.6s | 36.8s | 36.8s | 96.2s | 2.9M |
| | Museum Affandi | 3m 21s | 4m 1s | 3m 19s | 10m 54s | 4.1M | 3m 3s | 15m 59s | 17m 15s | 36m 22s | 1.2M |
| | CA13 | 15m | 20m 05s | 37m 52s | 1h 13m | 4.0M | 16m 24s | 1h 30m | 1h 40m | 3h 27m | 1.4M |

**Table 2:** *Time to generate an octree with our method and sampling strategies.*

| Data Set | SSD Potree | SSD Entwine | SSD Arena4D | SSD Poisson | SSD Random | HDD Potree | HDD Entwine | HDD Arena4D | HDD Poisson | HDD Random |
|---|---|---|---|---|---|---|---|---|---|---|
| Eclepens | 390.4 s | 444 s | 69.6 s | 11.6 s | 7 s | 515.3 s | 446 s | 89.2 s | 31.2 s | 27 s |
| Eclepens (morton) | 143.2 s | 108 s | 69.3 s | 11.6 s | 6.9 s | 145.6 s | 109 s | 81.3 s | 27.4 s | 23.9 s |
| Matterhorn | 640.1 s | 655 s | 284.8 s | 59.8 s | 57.7 s | 653.3 s | 709 s | 475 s | 96.2 s | 96.2 s |
| Museum Affandi | nomem | nomem | 1h 19m | 18m 25s | 10m 54s | nomem | nomem | 4h 11m | 38m 18s | 36m 22s |
| CA13 | 23h 28m | 9h 40m | 8h 27m | 1h 16m | 1h 13m | 2d 2h | 9h 11m | - | - | 3h 27m |

**Table 3:** *Comparing LOD creation times of Potree, Entwine and Arena4D to our method using Poisson-Disk or Random subsampling strategies. nomem indicates that the application ran out of memory (32GB).*

out-of-core approaches, an initial chunking step may still be necessary, but the indexing step could benefit greatly from GPGPU-based processing. In addition to faster LOD generation, we would also like to explore high-quality LOD generation, e.g., by computing and storing averaged attribute data in lower levels of detail. Most state-of-the-art methods, Wand et al. being a notable exception, pick a sample and promote it to lower levels of detail, which corresponds to downsizing images with nearest-neighbor interpolation. On the other hand, computing averages for lower levels of detail would correspond to mip mapping, which greatly reduces aliasing artifacts.

The full source code and windows binaries are available at https://github.com/potree/PotreeConverter/releases/tag/2.0. Videos are available at https://www.cg.tuwien.ac.at/research/publications/2020/SCHUETZ-2020-MPC/

## 10. Acknowledgements

The authors with to thank *Open Topography* and *PG&E* for providing the CA13 data set [CA13], *TU Wien, Institute of History of Art, Building Archaeology and Restoration* for the Museum Affandi and Candi Banyunibo data sets, *Pix4D* for the Eclepens and Matterhorn data sets, the Netherlands for the AHN3 data set, and the *Stanford University Computer Graphics Laboratory* for the Stanford Bunny model.

This research has been funded by the FWF projekt no. P32418, and by SITN, République et canton de Neuchâtel.

© 2020 The Author(s)
Computer Graphics Forum © 2020 The Eurographics Association and John Wiley & Sons Ltd.

## References

[AWLR17] Arkhipov, Dmitri I., Wu, Di, Li, Keqin, and Regan, Amelia C. *Sorting with GPUs: A Survey.* 2017. arXiv: 1709.02520 [cs.DC] 2.

[BJFadH19] Brandt, Sascha, Jähn, Claudius, Fischer, Matthias, and auf der Heide, Friedhelm Meyer. "Visibility-Aware Progressive Farthest Point Sampling on the GPU". *Computer Graphics Forum* 38.7 (2019), 413–424. doi: 10.1111/cgf.13848 3.

[Bow01] Bowers, Kevin. "Accelerating a Particle-in-Cell Simulation Using a Hybrid Counting Sort". *Journal of Computational Physics - J COMPUT PHYS* 173 (Nov. 2001), 393–411. doi: 10.1006/jcph.2001.6851 2.

[CA13] Pacific Gas & Electric Company. *PG&E Diablo Canyon Power Plant (DCPP): San Simeon and Cambria Faults, CA, Airborne Lidar survey.* Distributed by OpenTopography. 2013. doi: https://doi.org/10.5069/G9CN71V5 11.

[CLRS01] Cormen, Thomas H., Leiserson, Charles E., Rivest, Ronald L., and Stein, Clifford. *Introduction to Algorithms, Second Edition.* Section 8.2. The MIT Press, 2001. isbn: 0-262-03293-7 2.

[Coo86] Cook, Robert L. "Stochastic Sampling in Computer Graphics". *ACM Trans. Graph.* 5.1 (Jan. 1986), 51–72. issn: 0730-0301. doi: 10.1145/7529.8927 3, 6.

[DK18] Dieckmann, Alexander and Klein, Reinhard. "Hierarchical Additive Poisson Disk Sampling". *Vision, Modeling and Visualization.* The Eurographics Association, 2018. isbn: 978-3-03868-072-7. doi: 10.2312/vmv.20181256 3.

[DRD18] Discher, Sören, Richter, Rico, and Döllner, Jürgen. "A Scalable WebGL-based Approach for Visualizing Massive 3D Point Clouds using Semantics-Dependent Rendering Techniques". June 2018. doi: 10.1145/3208806.3208816 3.

[DVS03] Dachsbacher, Carsten, Vogelgsang, Christian, and Stamminger, Marc. "Sequential Point Trees". *ACM Trans. Graph.* 22.3 (July 2003), 657–662. issn: 0730-0301. doi: 10.1145/882262.882321 2, 3.

[EBN13] Elseberg, Jan, Borrmann, Dorit, and Nüchter, Andreas. "One billion points in the cloud – an octree for efficient processing of 3D laser scans". *ISPRS Journal of Photogrammetry and Remote Sensing* 76 (2013). Terrestrial 3D modelling, 76–88. issn: 0924-2716. doi: https://doi.org/10.1016/j.isprsjprs.2012.10.004 3.

[ENT] *Entwine*. Accessed 2020.06.29. url: https://entwine.io/ 1, 3.

[ENT2019] *Continental Scale Point Cloud Data Management and Exploitation with Entwine*. Accessed 2020.06.05. url: https://media.ccc.de/v/bucharest-129-continental-scale-point-cloud-data-management-and-exploitation-with-entwine 3.

[Fra17] Fraiss, Simon Maximilian. *Rendering Large Point Clouds in Unity*. Bachelor Thesis. Favoritenstrasse 9-11/E193-02, A-1040 Vienna, Austria, Sept. 2017. url: https://www.cg.tuwien.ac.at/research/publications/2017/FRAISS-2017-PCU/ 3.

[GM04] Gobbetti, Enrico and Marton, Fabio. "Layered point clouds: a simple and efficient multiresolution structure for distributing and rendering gigantic point-sampled models". *Computers & Graphics* 28.6 (2004), 815–826. issn: 0097-8493. doi: https://doi.org/10.1016/j.cag.2004.08.010 1, 2.

[GZPG10] Goswami, P., Zhang, Y., Pajarola, R., and Gobbetti, E. "High Quality Interactive Rendering of Massive Point Models Using Multi-way kd-Trees". *2010 18th Pacific Conference on Computer Graphics and Applications*. 2010, 93–100 3.

[HCR2014] Hoetzlein, Rama C. *FAST FIXED-RADIUS NEAREST NEIGHBORS: INTERACTIVE MILLION-PARTICLE FLUIDS*. GPU Technology Conference (GTC) 2014, Santa Clara, CA. 2014 2.

[KJWX19] Kang, L., Jiang, J., Wei, Y., and Xie, Y. "Efficient Randomized Hierarchy Construction for Interactive Visualization of Large Scale Point Clouds". *2019 IEEE Fourth International Conference on Data Science in Cyberspace (DSC)*. 2019, 593–597 3, 6, 7.

[Knu98] Knuth, Donald E. *The Art of Computer Programming*. Vol. 3. Section 5.2, Algorithm D. Addison-Wesley, 1998. isbn: 0-201-89685-0 2.

[Lei13] Leimer, Kurt. *External Sorting of Point Clouds*. Bachelor Thesis. Favoritenstrasse 9-11/E193-02, A-1040 Vienna, Austria, Sept. 2013. url: https://www.cg.tuwien.ac.at/research/publications/2013/leimer-2013-esopc/ 3.

[MVvM*15] Martinez-Rubi, Oscar, Verhoeven, Stefan, van Meersbergen, M., et al. "Taming the beast: Free and opensource massive point cloud web visualization". Capturing Reality Forum 2015, Salzburg, Austria. 2015 3, 7, 9.

[OP] *OPALS - Orientation and Processing of Airborne Laser Scanning data*. Accessed 2020.06.29. url: https://opals.geo.tuwien.ac.at/html/stable/index.html 1.

[PMOK14] Pfeifer, N., Mandlburger, G., Otepka, J., and Karel, W. "OPALS - A framework for Airborne Laser Scanning data analysis". *Computers, Environment and Urban Systems* 45 (2014), 125–136. issn: 0198-9715. doi: https://doi.org/10.1016/j.compenvurbsys.2013.11.002 1.

[POT] *Potree*. Accessed 2020.05.27. url: http://potree.org/ 1, 3, 7.

[RD10] Richter, Rico and Döllner, Jürgen. "Out-of-core real-time visualization of massive 3D point clouds". AFRIGRAPH 2010, South Africa. Jan. 2010, 121–128. doi: 10.1145/1811158.1811178 3.

[RL00] Rusinkiewicz, Szymon and Levoy, Marc. "QSplat: A Multiresolution Point Rendering System for Large Meshes". *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '00. USA: ACM Press/Addison-Wesley Publishing Co., 2000, 343–352. isbn: 1581132085. doi: 10.1145/344779.344940 2.

[Sch14] Scheiblauer, Claus. "Interactions with Gigantic Point Clouds". PhD thesis. Favoritenstrasse 9-11/E193-02, A-1040 Vienna, Austria: Institute of Computer Graphics and Algorithms, Vienna University of Technology, 2014. url: https://www.cg.tuwien.ac.at/research/publications/2014/scheiblauer-thesis/ 3, 8.

[Sch16] Schütz, Markus. "Potree: Rendering Large Point Clouds in Web Browsers". MA thesis. Favoritenstrasse 9-11/E193-02, A-1040 Vienna, Austria: Institute of Computer Graphics and Algorithms, Vienna University of Technology, Sept. 2016. url: https://www.cg.tuwien.ac.at/research/publications/2016/SCHUETZ-2016-POT/ 3.

[SW11] Scheiblauer, Claus and Wimmer, Michael. "Out-of-Core Selection and Editing of Huge Point Clouds". *Computers & Graphics* 35.2 (Apr. 2011), 342–351. issn: 0097-8493. url: https://www.cg.tuwien.ac.at/research/publications/2011/scheiblauer-2011-cag/ 3.

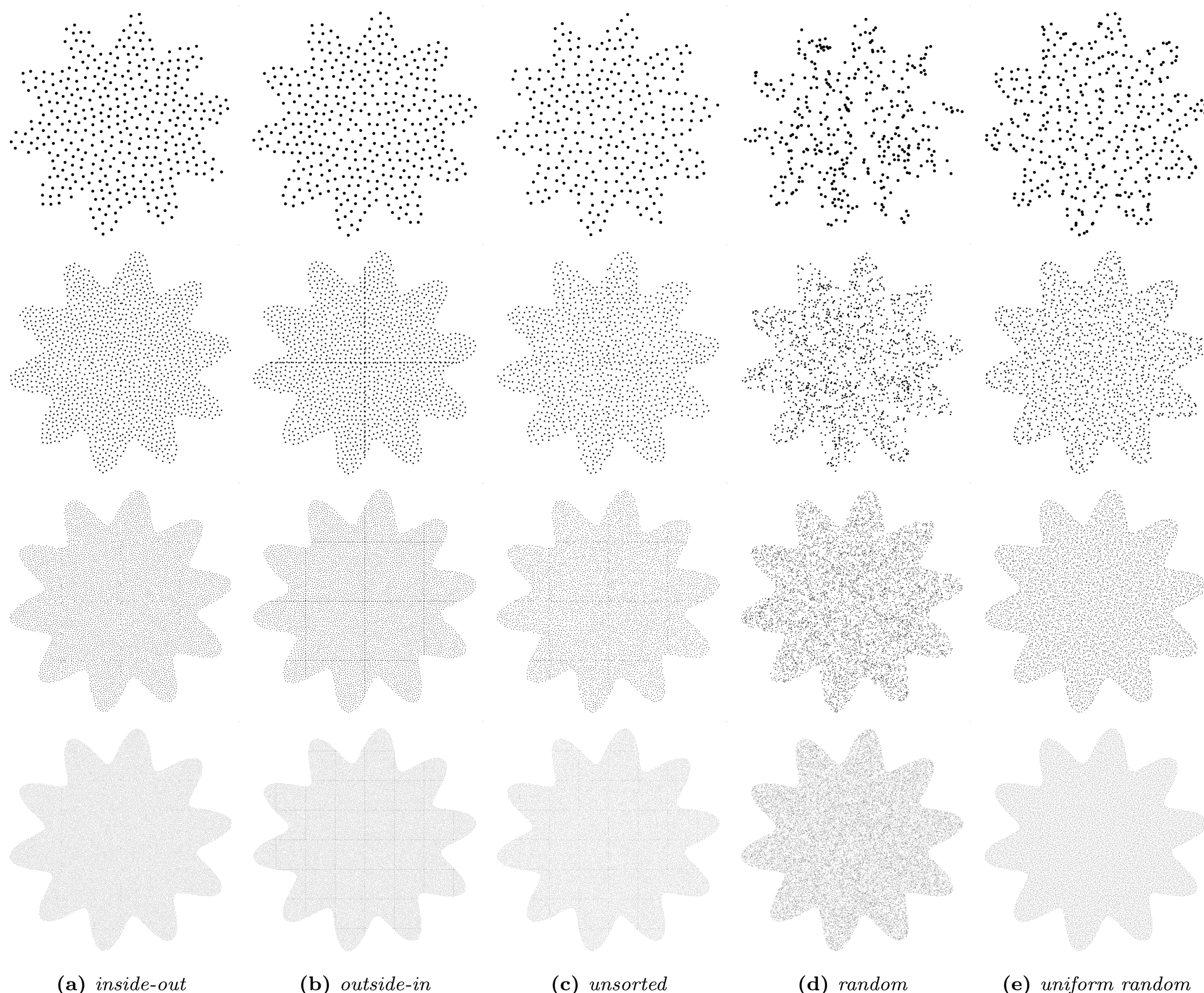
[WBB*08] Wand, Michael, Berner, Alexander, Bokeloh, Martin, et al. "Processing and interactive editing of huge point clouds from 3D scanners". *Computers & Graphics* 32.2 (2008), 204–220. issn: 0097-8493. doi: https://doi.org/10.1016/j.cag.2008.01.010 3.

[WS06] Wimmer, Michael and Scheiblauer, Claus. "Instant Points: Fast Rendering of Unprocessed Point Clouds". *Symposium on Point-Based Graphics*. The Eurographics Association, 2006. isbn: 3-905673-32-0. doi: 10.2312/SPBG/SPBG06/129-136 3.

[YGW*15] Yan, Dong-Ming, Guo, Jianwei, Wang, Bin, et al. "A Survey of Blue-Noise Sampling and Its Applications". *Journal of Computer Science and Technology* 30 (May 2015), 439–452. doi: 10.1007/s11390-015-1535-0 3, 6.

[Yuk15] Yuksel, Cem. "Sample Elimination for Generating Poisson Disk Sample Sets". *Comput. Graph. Forum* 34.2 (May 2015), 25–32. issn: 0167-7055. doi: 10.1111/cgf.12538 3.

© 2020 The Author(s)
Computer Graphics Forum © 2020 The Eurographics Association and John Wiley & Sons Ltd.



**(a)** *inside-out* **(b)** *outside-in* **(c)** *unsorted* **(d)** *random* **(e)** *uniform random*

**Figure 11:** *Comparison of hierarchical bottom-up subsampling strategies. Subsamples in the bottom row are organized in 8x8 tiles / nodes, which corresponds to a quadtree with a depth of 3 levels. They are all subsampled from the same high-density input point cloud. The rows above it are made up of 4x4, 2x2, and 1 tile at the top. Each row is a subsample of the one directly below of it. (a) and (e) are the strategies we implemented in our octree generator. (a,b,c) Our hierarchical Poisson-disk sampling strategy only enforces minimum distances within a node, but evaluating points from the inside out reduces the chance of clustering artifacts near borders, while outside-in evaluation performs even worse than unsorted. (d, e) Simple random subsampling leads to poor coverage with clusters in some regions and holes in others. A uniform random subsampling strategy that selects one point per grid cell improves the point distribution.*

© 2020 The Author(s)
Computer Graphics Forum © 2020 The Eurographics Association and John Wiley & Sons Ltd.