SimLOD: Simultaneous LOD Generation and Rendering for Point Clouds

MARKUS SCHÜTZ, LUKAS HERZBERGER, and MICHAEL WIMMER, TU Wien, Austria

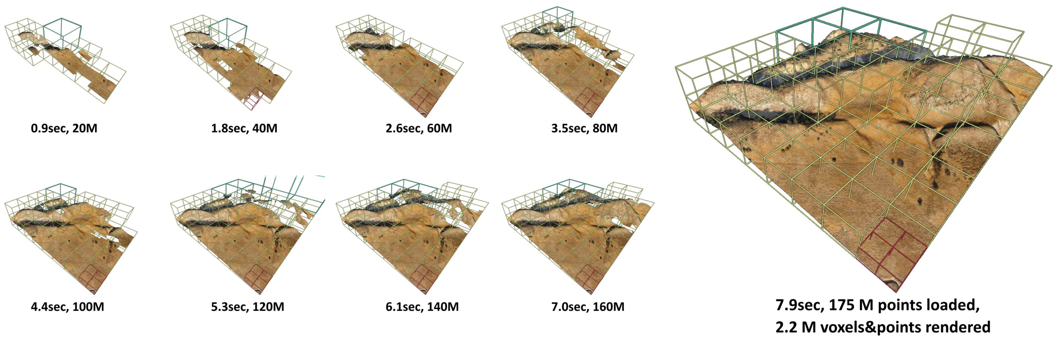


Fig. 1. Incrementally generating the LOD structure on-the-fly as the data set is being loaded. Depicted is a scenario where compressed data loads slowly, but our method scales to up to 580 million points per second.

About: We propose an incremental LOD generation approach for point clouds that allows us to simultaneously load points from disk, update an octree-based level-of-detail representation, and render the intermediate results in real time while additional points are still being loaded from disk. LOD construction and rendering are both implemented in CUDA and share the GPU's processing power, but each incremental update is lightweight enough to leave enough time to maintain real-time frame rates.

Background: LOD construction is typically implemented as a preprocessing step that requires users to wait before they are able to view the results in real time. This approach allows users to view results right away.

Results: Our approach is able to stream points from an SSD and update the octree on the GPU at rates of up to 580 million points per second (~9.3GB/s) on an RTX 4090 and a PCIe 5.0 SSD. Depending on the data set, our approach spends an average of about 1 to 2 ms to incrementally insert 1 million points into the octree, allowing us to insert several million points per frame into the LOD structure and render the intermediate results within the same frame.

Discussion/Limitations: We aim to provide near-instant, real-time visualization of large data sets without preprocessing. Out-of-core processing of arbitrarily large data sets and color-filtering for higher-quality LODs are subject to future work.

ACM Reference Format:
Markus Schütz, Lukas Herzberger, and Michael Wimmer. 2024. SimLOD: Simultaneous LOD Generation and Rendering for Point Clouds. *Proc. ACM Comput. Graph. Interact. Tech.* 7, 1 (May 2024), 20 pages. https://doi.org/10.1145/3651287

Authors' address: Markus Schütz, mschuetz@cg.tuwien.ac.at; Lukas Herzberger, lherzberger@cg.tuwien.ac.at; Michael Wimmer, wimmer@cg.tuwien.ac.at, TU Wien, Favoritenstr. 9-11 / E193-02, Vienna, Vienna, Austria, 1040.

ACM 2577-6193/2024/5-ART
https://doi.org/10.1145/3651287

1 INTRODUCTION

Point clouds are an alternative representation of 3D models, comprising colored points without connectivity, and are typically obtained by scanning the real world via means such as laser scanners or photogrammetry. Since they are colored on a per-point basis, large amounts of points are required to represent details that triangle meshes can cheaply simulate with textures. As such, point clouds are not an efficient representation for games, but they are nevertheless popular and ubiquitously available due to the need to scan real-world objects, buildings, and even whole countries.

Examples for massive point-cloud data sets include: The 3D Elevation Program (3DEP), which intends to scan the entire USA [48], and Entwine[16], which currently hosts 53.6 trillion points that were collected in various individual scan campaigns within the 3DEP program [49]. The Actueel Hoogtebestand Nederland (AHN) [2] program repeatedly scans the entire Netherlands, with the second campaign resulting in 640 billion points [3], and the fourth campaign being underway. Many other countries also run their own country-wide scanning programs to capture the current state of land and infrastructure. At a smaller scale, buildings are often scanned as part of construction, planning, and digital heritage. But even though these are smaller in extent, they still comprise hundreds of millions to several billion points due to the higher scan density of terrestrial LIDAR and photogrammetry.

One of the main issues when working with large point clouds is the computational effort that is required to process and render hundreds of millions to billions of points. Level-of-detail structures are an essential tool to quickly display visible parts of a scene up to a certain amount of detail, thus reducing load times and improving rendering performance on lower-end devices. However, generating these structures can also be a time-consuming process. Recent GPU-based methods [42] improved LOD compute times down to a second per billion points, but they still require users to wait until the entire data set has been loaded and processed before the resulting LOD structure can be rendered. Thus, if loading a billion points takes 60 seconds plus 1 second of processing, users still have to wait 61 seconds to inspect the results.

In this paper, we propose an incremental LOD generation approach that allows users to instantly look at data sets as they are streamed from disk, without the need to wait until LOD structures are generated in advance. This approach is currently in-core, i.e., data sets must fit into memory, but we expect that it will serve as a basis for future out-of-core implementations to support arbitrarily large data sets.

Our contributions to the state-of-the-art are as follows:

- An approach that instantly displays large amounts of points as they are loaded from fast SSDs, and simultaneously updates an LOD structure directly on the GPU to guarantee high real-time rendering performance.
- As a smaller, additional contribution, we demonstrate that dynamically growing arrays of points via unrolled linked lists (linked-lists of arrays) can be rendered efficiently in modern, compute-based rendering pipelines.

Specifically not a contribution is the development of a new LOD structure. We generate the same structure as Wand et al. [51] or Schütz et al. [42], which are also very similar to the widely used modifiable nested octrees [40]. We opted for constructing the former over the latter because the quantized voxels used for inner nodes compress better than full-precision points (down to 10 bits per colored voxel), which improves the transfer speed of lower LODs over the network. Furthermore, since that approach does not store original points in inner nodes (unlike nested octrees), we can compute more representative, color-filtered values for the inner nodes. However, both compression and color filtering are applied in post-processing before storing the results on disk and are not covered by this paper. This paper focuses on incrementally creating the LOD structure and its geometry as fast as possible for immediate display and picks a single color value from the first point that falls into a voxel cell.

2 RELATED WORK

2.1 LOD Structures for Point Clouds

Point-based and hybrid LOD representations were initially proposed as a means to efficiently render mesh models at lower resolutions [12, 13, 15, 39] and possibly switch to the original triangle model at close-up views. With the rising popularity of 3D scanners that produce point clouds as intermediate and/or final results, these algorithms also became useful to handle the enormous amounts of geometry that are generated by scanning the real world. Layered point clouds (LPC) [18] was the first GPU-friendly as well as view-dependent approach, which made it suitable for visualizing arbitrarily large data sets. LPCs organize points into a multi-resolution binary tree where each node represents a part of the point cloud at a certain level of detail, with the root node depicting the whole data set at a coarse resolution, and child nodes adding additional detail in their respective regions. Since then, further research has improved various aspects of LPCs, such as utilizing different tree structures [19, 35, 51, 53], improving LOD construction times [5, 26, 30, 32, 46] and higher-quality sampling strategies instead of selecting random subsets [42, 50].

In this paper, we focus on constructing a variation of LPCs proposed by Wand et al. [51], which utilizes an octree where each node creates a coarse representation of the point cloud with a resolution of 128^3 cells, and leaf nodes store the original, full-precision point data, as shown in Figure 3. Wand et al. suggest various primitives as coarse, representative samples (quantized points, Surfels, ...), but for this work we consider each cell of the 128^3 grid to be a voxel. A similar voxel-based LOD structure by Chajdas et al. [11] uses 256^3 voxel grids in inner nodes and original triangle data in leaf nodes. Modifiable nested octrees (MNOs) [40] are also similar to the approach by Wand et al. [51], but instead of storing all points in leaves and representative samples (Surfels, Voxels, ...) in inner nodes, MNOs fill empty grid cells with points from the original data set.

Since our goal is to display all points the instant they are loaded from disk to GPU memory, we need LOD construction approaches that are capable of efficiently inserting new points into the hierarchy, expanding it if necessary, and updating all affected levels of detail. This disqualifies recent bottom-up or hybrid bottom-up and top-down approaches [5, 32, 42, 46] that achieve a high construction performance, but which require preprocessing steps that iterate through all data before they actually start with the construction of the hierarchy. Wand et al. [51] as well as Scheiblauer and Wimmer [40], on the other hand, propose modifiable LOD structures with deletion and insertion methods, which make these inherently suitable to our goal since we can add a batch of points, draw the results, and then add another batch of points. Bormann et al. [4] were the first to specifically explore this concept for point clouds by utilizing MNOs, but flushing updated octree nodes to disk that an external rendering engine can then stream and display. They achieved a throughput of 1.8 million points per second, which is sufficient to construct an LOD structure as fast as a laser scanner generates point data. A downside of these CPU-based approaches is that they do not parallelize well, as threads need to avoid processing the same node or otherwise sync critical operations. In this paper, we propose a GPU-friendly approach that allows an arbitrary amount of threads to simultaneously insert points, which allows us to load points from the SSD and construct and render them on the GPU at rates of up to 580 million points per second, or up to 1.2 billion points per second if we exclusively count the duration of the LOD construction kernel.

While we focus on point clouds, there are some notable related works in other fields, especially voxel editing, that allow simultaneous LOD generation and rendering. In general, any LOD structure with insertion operations can be assumed to fit these criteria, as long as inserting a meaningful amount of geometry can be done in milliseconds. Keller et al. [27] introduce dynamic volume trees, a KD-tree-like structure that is managed on the GPU and which allows insertion/removal of voxel data, including the option for volumetric brushing. Careil et al. [9] as well as Molenaar and Eisemann [34] demonstrate voxel editing approaches that are backed by a compressed LOD structure. We believe that *Dreams* – a popular 3D scene painting and game development tool for the PS4 – also matches the criteria, as developers reported experiments with LOD structures, and described the current engine as a "cloud of clouds of point clouds" [17]. Zellmann et al. [56] also provide interesting related work in which they stream a frame of a massive animated scalar fields to the GPU, build an accelerating structure, and while one frame is being rendered, the next frame is already being loaded. The construction process requires each individual frame's data in full before it starts building the acceleration structure, however. Of interest is also GigaVoxels, which is based on similar concepts as layered point clouds or modifiable nested octrees, but targeted towards volumetric voxel data sets [14]. GigaVoxels uses an N^3 tree (e.g. octree in case of $N = 2$) where each node either comprises a brick of M^3 voxels or a constant value in case of a homogeneous volume.

2.2 Linked Lists

Linked lists are a well-known and simple structure whose constant insertion and deletion complexity, as well as the possibility to dynamically grow without relocation of existing data, make it useful as part of more complex data structures and algorithms (e.g., least-recently-used (LRU) Caches [55]). Regular linked lists are represented by nodes comprising one item and a pointer to the next node/item in the list. Unrolled linked lists [47] are a variation that store multiple items in each node and a pointer to the next set of items (i.e., they are linked lists of arrays). A similar data structure – std::hive – is currently being proposed for addition into the C++ standard [1]. std::hive also uses unrolled linked lists of blocks of data for efficient growth without reallocation (as opposed to std::vector), but further extends it with skipfields (for efficient skipping of deleted elements in a block) and growth factors (block sizes grow larger if many blocks are needed). On GPUs, linked lists can be used to realize order-independent transparency [54] by creating pixel-wise lists of fragments that can then be sorted and drawn front to back. Another use-case of pixel-wise lists of fragments is the efficient rendering of gaussian surfaces [7]. In this case, linked lists support front-to-back ray traversal of dynamically animated gaussians in the vicinity of the ray, which are therefore contributing candidates for the ray-isosurface intersection.

In this paper we use unrolled linked lists to efficiently append an unknown amount of points and voxels to octree nodes during LOD construction, and also efficiently render the coalesced sets of points in each linked list node.

3 DATA STRUCTURE

3.1 Octree

The LOD data structure we use is an octree-based [40] layered point cloud [18] with representative voxels in inner nodes and the original, full-precision point data in leaf nodes, which makes it essentially identical to the structures of Wand et al. [51] or Schütz et al. [42]. Leaf nodes store up to 50k points and inner nodes up to 128^3 (2M) voxels, but typically closer to 128^2 (16k) voxels due to the surfacic nature of point cloud data sets. The sparse nature of surface voxels is the reason why we store them in lists instead of grids – exactly the same as points.

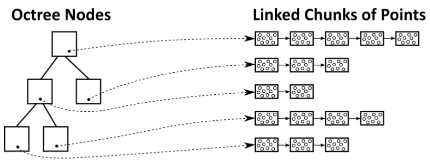
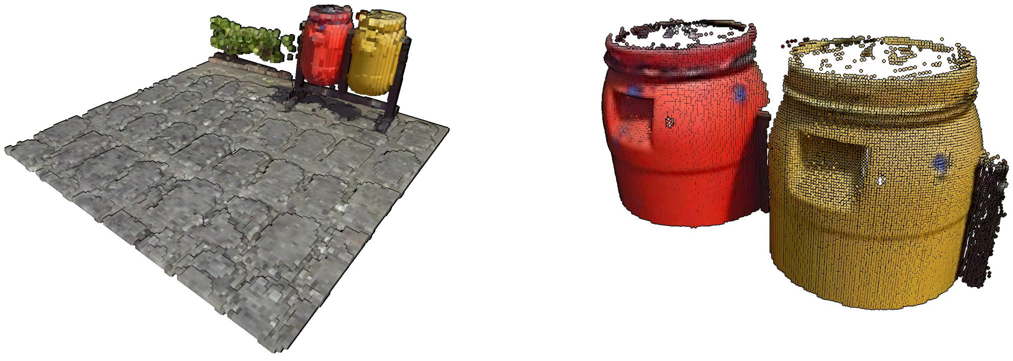


Fig. 2. Octree nodes store 3D data as linked chunks of points/voxels, which enables efficient growth as additional points are inserted over time.



(a) Inner node with Voxels. (b) Leaf node with points.

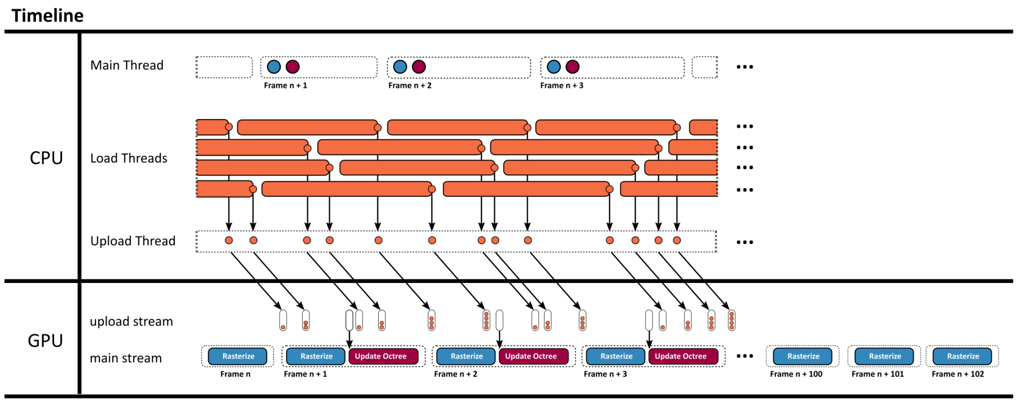
Fig. 3. (a) Close-up of a lower-resolution inner-node comprising 20 698 voxels that were sampled on a 128^3 grid. (b) A full-resolution leaf node comprising 22 858 points.

The difference to the structure of Schütz et al. [42] is
that we store points and voxels in linked lists of chunks of points, which allows us to add additional capacity by allocating and linking additional chunks, as shown in Figure 2. An additional difference to Wand et al. [51] is that they use hash maps for their 128^3 voxel sampling grids, whereas we use a $128^3 bit = 256kb$ occupancy grid per inner node to simplify massively parallel sampling on the GPU.

Despite the support for dynamic growth via linked lists, this structure still supports efficient rendering in compute-based pipelines, where each individual workgroup can process a coalesced set of points in a chunk in parallel, and then traverse to the next chunk as needed. In our implementation, each chunk stores up to $1,000$ points or voxels (Discussion in Section 6.5), with the latter being implemented as points where coordinates are quantized to the center of a voxel cell. This means that leaf nodes, which are limited to 50k points, hold up to 50 chunks in our implementation. The upper bound for inner nodes is about $\frac{128^3 voxels}{1000} \approx 2,100$ chunks, but in practice, it is closer to 16 chunks per inner node due to the aforementioned surfacic nature of point-cloud data sets.

3.2 Persistent Buffer

Since we require large amounts of memory allocations on the device from within the CUDA kernel throughout the LOD construction over hundreds of frames, we manage our own custom persistent buffer on device. To that end, we simply pre-allocate 90% of the available GPU memory and use a custom, CUDA-based memory allocator that manages allocations within this pre-allocated blob. Since we only add data but (with few exceptions) do not remove it, this allocator only needs to handle requests to more memory but not free previously allocated memory. Thus, we can simply use an atomic offset counter to keep note of the next available memory in the blob. An exception are chunks of points which are freed when a leaf node is turned into an inner node. These freed chunks are tracked via a chunk pool to make them available for re-allocation (Section 3.4). Note that sparse buffers via virtual memory management may be an alternative, as discussed in Section 7.



● A batch of one million points.
▬ Multiple load threads simultaneously load batches of points from the disk into pinned memory and queue them for upload.
⬚ A single upload thread asynchronously copies loaded points from pinned CPU memory into a queue in GPU memory.
●● Each frame, the main thread launches the rasterization and update kernels.
● The rasterization kernel renders the current state of the octree into an OpenGL framebuffer.
● The update kernel incrementally inserts all queued points into the octree.

Fig. 4. Timeline of our system over several frames.

3.3 Voxel Sampling Grid

Voxels are sampled by inscribing a 128^3 voxel grid into each inner node, using 1 bit per cell to indicate whether that cell is still empty or already occupied, amounting to 256kb of memory per inner node. This occupancy grid is only required during octree creation. The voxel coordinates and colors are stored in linearized form in linked chunks in order to allow fast rasterization of non-volumetric data, and therefore largely empty voxel grids. These linearized voxels are similar to points but with quantized coordinates. Grids are allocated from the persistent buffer whenever a leaf node is converted into an inner node during octree expansion (see Section 4.1).

3.4 Chunks and the Chunk Pool

We use chunks of points/voxels to dynamically increase the capacity of each node as needed, and a chunk pool where we return chunks that are freed after splitting a leaf node . Each chunk has a static capacity of N points/voxels ($1,000$ in our implementation), which makes it trivial to manage chunks as they all have the same size. Initially, the pool is empty and new chunks are allocated from the persistent buffer. When chunks are freed after splitting a leaf node, we store the pointers to these chunks inside the chunk pool. Future chunk allocations first attempt to acquire chunk pointers from the pool, and only allocate new chunks from the persistent buffer if there are none left in the pool.

4 INCREMENTAL LOD CONSTRUCTION

Our method loads batches of points from disk to GPU memory, updates the LOD structure in one CUDA kernel, and renders the updated results with another CUDA kernel. Figure 4 shows an overview of that pipeline. Both kernels utilize persistent threads [21, 28] using the cooperative group API [22] in order to merge numerous sub-passes into a single CUDA kernel. Points are loaded from disk to pinned CPU memory in batches of 1M points, utilizing multiple load threads. Whenever a batch is fully loaded, it is appended to a queue. A single uploader thread watches that queue and asynchronously copies any fully loaded batches to a queue in GPU memory. In each frame, the main thread launches the rasterize kernel that draws the entire scene, followed by an update kernel that incrementally inserts all batches of points into the octree that finished uploading to the GPU (while partially uploaded batches are handled in the subsequent frame).

In each frame, the GPU may receive several batches of 1M points each. The update kernel loops through the batches and inserts them into the octree as shown in Figure 5. First, the octree is expanded until the resulting leaf nodes will hold at most 50k points (without inserting them yet). It then traverses each point of the batch through the octree again to generate voxels for inner nodes. Afterwards, it allocates sufficient chunks for each node to store all points in leaf-, and voxels in inner nodes. In the last step, it inserts the points and voxels into the newly allocated chunks of memory. This process is repeated for each batch received in this frame.

The premise of this approach is that it is cheaper in massively parallel settings to traverse the octree multiple times for each point and only insert them once at the end, rather than traversing the tree once per point but with the need for complex synchronization mechanisms whenever a node needs splitting or additional chunks of memory need to be allocated.

4.1 Expanding the Octree

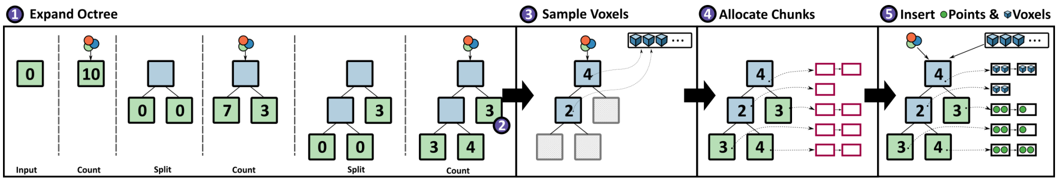
CPU-based top-down approaches [40, 51] typically traverse the hierarchy from root to leaf, update visited nodes along the way, and append points to leaf nodes. If a leaf node receives more than 50k points (see Section 3.1), it “spills" and is split into 8 child nodes. The points inside the spilling node are then redistributed to its newly generated child nodes. This approach works well on CPUs, where we can limit the insertion and expansion of a subtree to a single thread, but it raises issues in a massively parallel setting, where thousands of threads may want to insert points while we simultaneously need to split that node and redistribute the points it already contains.

To support massively parallel insertions of all points on the GPU, we propose an iterative approach that resembles a depth-first-iterative-deepening search [31]. Instead of attempting to fully expand the octree structure in a single step, we repeatedly expand it by one level until no more expansions are needed. This approach also decouples expansions of the hierarchy and insertions into a node’s list of points, which is now deferred to a separate pass. Since we already defer the insertion of points into nodes, we also defer the redistribution of points from spilled nodes. We maintain a spill buffer, which accumulates points of spilled nodes. Points in the spill buffer are subsequently treated exactly the same as points inside the batch that we are currently adding to the octree, i.e., the update kernel reinserts spilled points into the octree from scratch, along with the newly loaded batch of points.

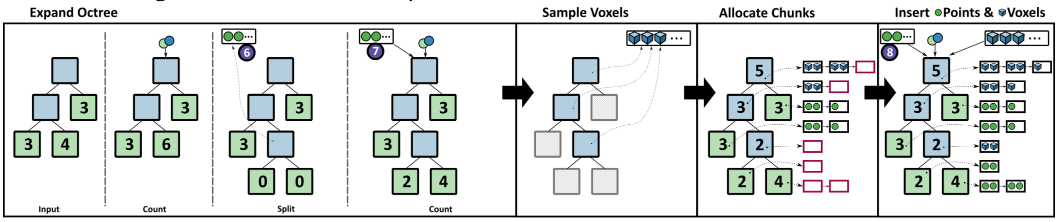
In detail, we repeat the following two sub-passes until no more nodes are spilled (see Figure 5):

- **Counting:** In each iteration, we traverse the octree for each point of the batch and all spilled points accumulated in previous iterations during the current update, and atomically increment the point counter of each hit leaf node. We do not count the same point twice in a leaf, however, so additional count passes will only affect newly generated leaves from the previous split pass.
- **Splitting:** All leaf nodes whose point counter exceeds a given threshold, e.g., 50k points, are split into 8 child nodes, each with a point counter of 0. The points it already contained are added to the list of spilled points. Note that the spilled points do not need to be associated with the nodes that they formerly belonged to – they are added to the octree from scratch. Furthermore, the chunks that stored the spilled points are released back to the chunk pool and may be acquired again later.

The expansion pass is finished when no more nodes are spilling.



(a) Adding 10 points to the octree. (1) Expanding the octree by repeatedly counting and splitting until leaf nodes hold at most T points (depicted: 5, in practice: 50k). (2) Leaves that were not split do not count points again. (3) The voxel sampling pass inserts all points again, creates voxels for empty cells in inner nodes, and stores new voxels (and the nodes they belong to) in a temporary *backlog* buffer. (4) Now that we know the number of new points and voxels, we allocate the necessary chunks (depicted size: 2, in practice: 1000) to store them. (5) All points are inserted again, traverse to the leaf, and are inserted into the chunks. Voxels from the *backlog* are inserted into the respective inner nodes.



(b) For illustrative purposes, we now add a batch of just two points which makes one of the nodes spill. (6) When splitting, we move all previously inserted points into a spill buffer. (7, 8) for the remainder of the current batch's insertion, points in the spill buffer and the batch get identical treatment.

Fig. 5. The CUDA kernel that incrementally updates the octree. (a) First, it inserts a batch with 10 points into the initially empty octree and (b) then adds another batch with two points that causes a split of a non-empty leaf node.

4.2 Voxel Sampling

Lower levels of detail are populated with voxel representations of the points that traversed these nodes. However, for efficiency reasons, we did not assign points to inner nodes during the expansion pass but defer this to a separate sampling pass. For this, we traverse each point through the octree again, and whenever a point visits an inner node, we project it into the inscribed 128^3 voxel sampling grid and check if the respective cell is empty or already occupied by a voxel. If the cell is empty, we create a voxel, increment the node's voxel counter, and set the corresponding bit in the sample grid to mark it as occupied. Note that in this way, the voxel gets the color of the first point that projects to it.

However, just like the points, we do not store voxels in the nodes right away because we do not know the amount of memory/chunks that each node requires until all voxels for the current incremental update are generated. Thus, voxels are first stored in a temporary backlog buffer with a large capacity. In theory, adding a batch of 1 million points may produce up to $(octreeLevels - 1)$ million voxels because each inner node's sampling grid has the potential to hold $128^3 = 2M$ voxels, and adding spatially close points may lead to several new octree levels until they are all separated into leaf nodes with at most 50k points. However, in practice, none of the test data sets of this paper produced more than 1M voxels per batch of 1M points, and of our numerous other data sets, the largest required backlog size was 2.4M voxels. Thus, we suggest using a backlog size of 10M points to be safe.

4.3 Allocating Chunks

After expansion and voxel sampling, we now know the exact amount of points and voxels that we need to store in leaf and inner nodes. Using this knowledge, we check whether the chunks of all affected nodes have sufficient free space to store the new points/voxels, or if we need to allocate new chunks of memory to raise the nodes' capacity by 1000 points or voxels per chunk. In total, we need $\lfloor \frac{counter + POINTS\_PER\_CHUNK - 1}{POINTS\_PER\_CHUNK} \rfloor$ linked chunks per node.

4.4 Storing Points and Voxels

To store points inside nodes, we traverse each point from the input batch and the spill buffer again through the octree to the respective leaf node and atomically update that node's *numPoints* variable. The atomic update returns the point index within the node, from which we can compute the index of the chunk and the index within the chunk where we store the point.

We then iterate through the voxels in the backlog buffer, which stores voxels and for each voxel a pointer to the inner node that it belongs to. Insertion is handled the same way as points – we atomically update each node's *numVoxels* variable, which returns an index from which we can compute the target chunk index and the position within that chunk.

5 RENDERING

Points and voxels are both drawn as pixel-sized splats by a CUDA kernel that utilizes atomic operations to retain the closest sample in each pixel [17, 20, 44]. Custom compute-based software-rasterization pipelines are particularly useful for our method because traditional vertex-shader-based pipelines are not capable of efficiently traversing linked lists. A CUDA kernel, however, has no issues looping through points in a chunk, and then traversing to the next chunk in the list. The recently introduced mesh and task shaders could theoretically also deal with linked lists of chunks of points, but they may benefit from smaller chunk sizes, and perhaps even finer-grained nodes (smaller sampling grids that lead to fewer voxels per node, and a lower maximum of points in leaf nodes).

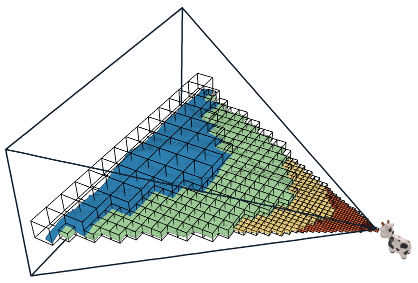


Fig. 6. LOD selection.

During rendering, we first assemble a list of visible nodes, comprising all nodes whose bounding box intersects the view frustum and which have a certain size on screen. Since inner nodes have a voxel resolution of 128^3, we need to draw their half-sized children if they grow larger than 128 pixels. Specifically, we draw nodes that fulfill all of the following conditions:

- They intersect the view frustum.
- Their parents are larger than 128 pixels.
- They are themselves smaller than 128 pixels, or they are a leaf node.

The last two conditions combined mean that we recursively evaluate whether a node is larger than 128 pixels, and if so we skip it and render all of its children instead. The pixel size of a node is computed by projecting all 8 vertices of the bounding box to the screen, and taking the maximum of the width and height. Figure 6 illustrates the resulting selection of rendered octree nodes within a frustum. Seemingly higher-LOD nodes are rendered towards the edge of the screen due to perspective distortions that make the screen-space bounding boxes bigger. For performance-sensitive applications, developers may instead want to do the opposite and reduce the LOD at the periphery and fill the resulting holes by increasing the point sizes.

To draw points or voxels, we launch one CUDA thread block per visible node whose threads loop through all samples of the node and jump to the next chunk when needed, as shown in listing 1. (Note: The assumption of launching one block per visible node is simplified since the amount of blocks is strongly limited when using the cooperative groups API. Instead, the available blocks would loop through the list of visible nodes until all of them are drawn. Please refer to *render.cu* in the provided source code for implementation details.)

```
auto grid = cooperative_groups::this_grid();
auto block = cooperative_groups::this_thread_block();

// All threads in a block grab the same node
int nodeIndex = grid.block_rank();
Node* node = visibleNodes[nodeIndex];
Chunk* chunk = node->points;
int chunkIndex = 0;

// Threads [0, blocksize) start to process points [0, blocksize),
// then advance by blocksize in each iteration.
// (Note: block.num_threads() == blocksize)
for(
    int pointIndex = block.thread_rank();
    pointIndex < node->numPoints;
    pointIndex += block.num_threads()
){

    // if thread is at end of chunk, advance to next
    int targetChunkIndex = pointIndex / POINTS_PER_CHUNK;
    if(chunkIndex < targetChunkIndex){
        chunk = chunk->next;
        chunkIndex++;
    }

    Point point = chunk->points[pointIndex % POINTS_PER_CHUNK];

    rasterize(point);
}
```

Listing 1. CUDA code showing threads of a block iterating through all points in a node, processing *block.num_threads* points at a time in parallel. Threads advance to the next chunk as needed.

6 EVALUATION

Our method was implemented in C++ and CUDA, and evaluated on the test data sets shown in Figure 7. The following systems were used for the evaluation:

OS	GPU	CPU	Disk
Windows 10	RTX 3060	Ryzen 7 2700X	Samsung 980 PRO, PCIe 3.0, 3.5GB/s
Windows 11	RTX 4090	Ryzen 9 7950X	Crucial T700, PCIe 5.0, 12.4GB/s

Special care was taken to ensure meaningful results for disk IO in our benchmarks:

- On Microsoft Windows, traditional C++ file IO operations such as `fread` or `ifstream` are automatically buffered by the operating system. This leads to two issues – First, it makes

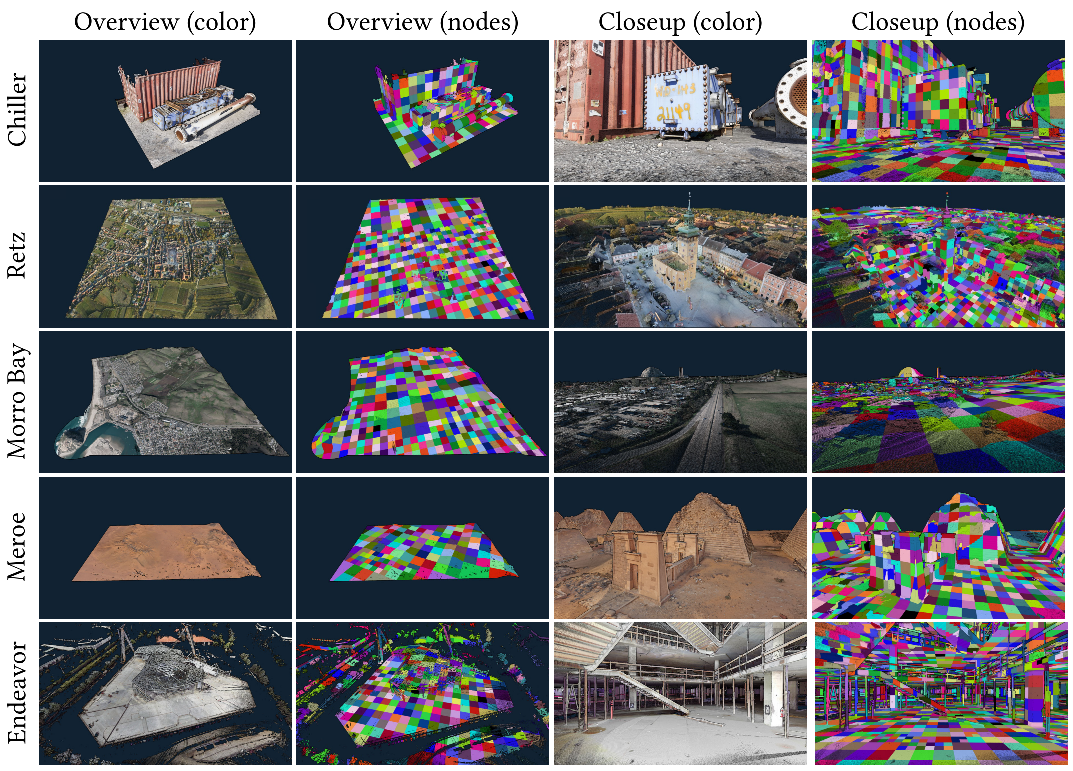


Fig. 7. Overview and close-ups of our test data sets. The second and fourth columns illustrate the rendered octree nodes.

the initial access to a file slower and significantly increases CPU usage, which decreases the overall performance of the application and caused stutters when streaming a file from SSD to GPU for the first time. Second, it makes future accesses to the same file faster because the OS now serves it from RAM instead of reading from disk.

- Since we are mostly interested in first-read performance, we implemented file access on Windows via the Windows API's `ReadFileEx` function together with the `FILE_FLAG_NO_BUFFERING` flag. It ensures that data is read from disk and also avoids caching it in the first place. As an added benefit, it also reduces CPU usage and resulting stutters.

We evaluated the following performance aspects, with respect to our goal of simultaneously updating the LOD structure and rendering the intermediate results:

(1) Throughput of the incremental LOD construction in isolation.
(2) Throughput of the incremental LOD construction while streaming points from disk and simultaneously rendering the intermediate results in real time.
(3) Average and maximum duration of all incremental updates.
(4) Performance of rendering nodes up to a certain level of detail.

6.1 Data Sets

We evaluated a total of five data sets shown in Figure 7, three file formats, and Morton-ordering vs. the original ordering by scan position. Chiller and Meroe are photogrammetry-based data sets, Morro Bay is captured via aerial LIDAR, Endeavor via terrestrial laser scans, and Retz via a combination of terrestrial (town center, high-density) and aerial LIDAR (surrounding, low-density).

The LAS and LAZ file formats are industry-standard point cloud formats. Both store XYZ, RGB, and several other attributes. Due to this, LAS requires either 26 or 34 bytes per point for our data sets. LAZ provides a good and lossless compression down to around 2-10 bytes/point, which is why most massive LIDAR data sets are distributed in that format. However, it is quite CPU-intensive and therefore slow to decode. SIM is a custom file format that stores points in the update kernel's expected format – XYZRGBA (3 x float + 4 x uint8, 16 bytes per point).

Endeavor is originally ordered by scan position and the timestamp of the points, but we also created a Morton-ordered variation to evaluate the impact of the order.

6.2 Construction Performance

Table 1 covers items 1-3 and shows the construction performance of our method on the test systems. The incremental LOD construction kernel itself achieves throughputs of up to 300M points per second on an RTX 3060, and up to 1.2 billion points per second on an RTX 4090. The whole system, including times to stream points from disk and render intermediate results, achieves up to 100 million points per second on an RTX 3060 and up to 580 million points per second on the RTX 4090. The durations of the incremental updates are indicators for the overall impact on fps (average) and occasional stutters (maximum). We implemented a time budget of 10ms per frame to reduce the maximum durations of the update kernel (RTX 3060: 45ms $\longrightarrow$ 16ms; RTX 4090 25ms $\longrightarrow$ 13ms). After the budget is exceeded, the kernel stops processing additional batches and resumes the next frame. This budget is especially relevant to deal with bursts where many loader threads finished loading batches in the same frame. Instead of inserting all batches in the same frame and causing a stutter, the budget redistributes the workload over the next few frames. Our method benefits from locality as shown by the Morton-ordered variant of the Endeavor data set, which increases the construction performance by a factor of x2.5 (497 MP/s $\longrightarrow$ 1221 MP/s).

6.3 Comparison to the state of the art

Table 2 compares our method to state-of-the-art applications with available source/binaries. *Entwine* and its successor *Untwine* are popular open-source applications that create octree-based MNOs. *Arena4D* is a proprietary point-cloud engine with a freely available LOD generation component that creates a KD-tree based MNO. *Potree* is an open source web-based viewer for point clouds that creates octree-based MNOs. The construction algorithm behind Potree is published by Schütz et al. [46]. SSKW23 [42] is, to our knowledge, the first GPU-based point-cloud indexing approach. As such, it achieves significant speedups over the other methods, but it is also currently only in-core. SSKW23 and our approach both create the LOD structure proposed by Wand et al. [51], which is very similar to the octree-based MNOs used by Entwine, Untwine and Potree.

Due to varying goals and circumstances, fair comparisons are difficult. For example, out-of-core methods also include the time to write the results back to disk, but this cannot be separately measured because reading, processing and writing of different parts of the data set happen simultaneously. Since all steps happen in parallel, writing may not increase the total runtime at all, or it may slightly increase it because it takes away resources from reading and processing. SSKW23 is particularly special because it only measures CUDA kernel compute times and completely ignores file IO. Our approach, on the other hand, targets stream-processing data on-the-fly as it is loaded from disk, and it is specifically optimized for fast loading and simultaneous processing. The full runtime of our approach can theoretically be slightly better than SSKW23 because we already processes the data simultaneously during loading, while SSKW23 only starts processing afterwards. However, because SSKW23's implementation is not optimized for loading, it is several times slower

Data Set	points (M)	format	size (GB)	Update avg (ms)	Update max (ms)	Duration updates (sec)	Duration total (sec)	Throughput updates (MP/s)	Throughput total (MP/s)	Throughput total (GB/s)
RTX 3060										
Chiller	73.6	LAS	1.9	1.2	13.3	0.2	1.3	297	54	1.4
		SIM	1.2	2.0	14.0	0.2	0.8	298	87	1.4
Retz	145.5	LAS	4.9	1.0	14.9	0.6	3.2	260	45	1.5
		SIM	2.3	2.8	14.3	0.5	1.6	272	91	1.4
Morro Bay	350.0	LAS	11.9	1.2	15.9	1.4	7.6	242	46	1.6
		SIM	5.6	4.1	16.1	1.4	3.5	247	100	1.6
RTX 4090										
Chiller	73.6	LAS	1.9	0.6	7.5	0.1	0.3	1,215	291	7.5
		SIM	1.2	0.6	8.0	0.1	0.2	1,217	439	7.2
Retz	145.5	LAS	4.9	0.4	8.0	0.1	0.7	1,145	221	7.4
		SIM	2.3	1.0	8.6	0.1	0.4	1,187	425	6.7
Morro Bay	350.0	LAS	11.9	0.6	9.2	0.4	1.5	979	234	8.0
		SIM	5.6	1.5	10.9	0.3	0.8	1,030	458	7.3
Meroe	684.4	LAS	23.3	0.7	10.4	0.8	2.8	882	241	8.2
		SIM	11.4	1.9	12.1	0.7	1.7	945	401	6.4
Endeavor	796.0	LAS	20.7	7.0	12.6	1.6	2.6	497	307	8.0
		LAZ	8.0	0.2	7.2	2.4	25.1	328	32	0.3
		SIM	12.7	9.1	12.9	1.6	2.3	497	341	5.4
Endeavor (z-order)		SIM	12.7	2.2	10.7	0.7	1.4	1,221	581	9.3

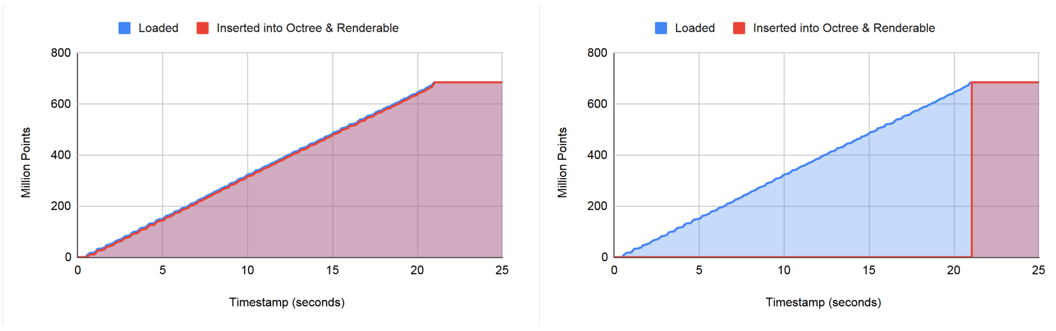
Table 1. LOD Construction Performance showing average and maximum durations of the update kernel, total duration of all updates or the whole system, and the throughput in million points per second (MP/s) or gigabytes per second (GB/s). Total duration includes the time to load points from disk, stream them to the GPU, and insert them into the octree. Update measures the duration of all incremental per-frame (may process multiple batches) updates in isolation. Throughput in GB/s refers to the file size, which depends on the number of points and the storage format (ranging from 10 (LAZ), 16 (SIM) to 26 or 34 (LAS) bytes per point).

		duration(seconds)					
		non-incremental					incremental
		out-of-core & CPU-based				in-core & GPU-based	
Data Set	Points	Entwine [16]	Untwine	Arena4D	Potree [46]	SSKW23 [42]	ours
Chiller	73.6 M	85	28	19.4	4.0	2.5 + 0.011	0.3
Retz	145.5 M	161	75	38.4	9.1	3.9 + 0.024	0.7
Morro Bay	350.0 M	523	134	77.3	27.3	8.7 + 0.046	1.5
Meroe	684.4 M	1230	190	185.8	39.4	nomem	2.8
Endeavor	796.0 M	8755	-	161.4	43.5	nomem	2.6
Endeavor^{zo}	796.0 M	996	853	223.7	42.5	nomem	1.4

Table 2. LOD construction performance comparison to other systems, evaluated on the RTX 4090 system. All methods include file IO, except for SSKW23 which only optimizes compute and not loading. Since it is also the only method where loading and processing are done one after the other instead of simultaneously, we report load and compute durations separately. With optimized IO, it should achieve the same as ours.
^{zo}: Sorted by z-order (Morton Code). *nomem*: Insufficient memory.

in that regard, thus making a full runtime comparison meaningless since they do not attempt to quickly load point clouds. For that reason, we report their load and compute times separately.

Nevertheless, Table 2 demonstrates that GPU-based processing speeds up the LOD construction by one to two orders of magnitude. On top of that, our incremental approach instantly provides



(a) Incremental LOD construction (Ours) (b) Non-Incremental LOD Construction (SSKW23)

Fig. 8. Incremental LOD generation allows us to immediately display data while it is being loaded. While the performance benchmarks show that our approach can handle large amounts of quickly loaded points, it is particularly interesting in scenarios where loading is slow. (a) shows a factual timeline recorded on a compressed Meroe data set, whose CPU-heavy decoding algorithm increases load times from 1.7 (*.SIM) or 2.8 (*.LAS) seconds to 21 (*.LAZ) seconds. (b) shows an exemplary timeline assuming the same load speed as ours but with SSKW23's near-instant non-incremental LOD construction (0.08s, extrapolated from Table 2 for Meroe). Although (b) is faster computation-wise, both finish at roughly the same time since (a) already processes during loading while (b) needs to wait until all data is loaded. For (b), however, users have to wait until the entire data is loaded and then processed before being able to see it.

visual feedback as data is being loaded, while all other approaches in the list are pre-processing approaches that require users to wait until the construction is finished. So while the times shown in Table 2 for our approach are for loading the whole data set, it already shows visual results as soon as the first batches of points are loaded, copied to GPU, and processed, as illustrated in Figure 8. The first points appear about 60ms after drag&dropping a point cloud file into our application, and from there we linearly progress until all points are loaded. Notably missing from the list is Bormann et al. [4], which also operates incrementally with immediate visual feedback, but we were not able to make their approach work on our system as it requires special trajectory data which we do not have, and which also does not exist for photogrammetry data sets. Compared to their own benchmarks on an unspecified 16-core system, our approach is up to 320 times faster (1.8MP/s $\longrightarrow$ 580MP/s), or up to 677 times faster if we only account for the construction kernel times without rendering and loading (1.8MP/s $\longrightarrow$ 1221MP/s).

If we only account for compute times without loading and rendering, our incremental approach is about 7.7 times slower than the non-incremental SSKW23 approach for the same first-come sampling method (Morro Bay: 7609 MP/s $\longrightarrow$ 979 MP/s). It is about 15.8 times (Morro Bay; with rendering: 14.8 MP/s $\longrightarrow$ 234 MP/s) to 66 times (Morro Bay; without rendering; 14.8 MP/s $\longrightarrow$ 979 MP/s) faster than the non-incremental, CPU-based Potree [46]. Due to the higher memory requirements of SSKW23, we are able to support larger data sets.

6.4 Rendering Performance

Regarding rendering performance, we show that linked lists of chunks of points/voxels are suitable for high-performance real-time rendering by rendering the constructed LOD structure at high resolutions (pixel-sized voxels). Table 3 shows that we are able to render up to 89.4 million pixel-sized points and voxels in 2.7 milliseconds, which leaves the majority of a frame's time for the construction kernel (or higher-quality shading). Table 4 shows that the size of chunks has negligible impact on rendering performance (provided they are larger than the workgroup size). In practice,

	overview					closeup				
	points	voxels	nodes	duration	samples/ms	points	voxels	nodes	duration	samples/ms
RTX 3060										
Chiller	2.6 M	8.4 M	441	3.3 ms	3.3 M	28.0 M	7.9 M	1678	7.4 ms	4.9 M
Retz	5.2 M	12.4 M	644	4.4 ms	4.0 M	18.9 M	7.5 M	1616	6.0 ms	4.4 M
Morro Bay	0.6 M	12.0 M	477	3.5 ms	3.6 M	16.3 M	13.7 M	1346	6.5 ms	4.6 M
RTX 4090										
Chiller	2.6 M	8.4 M	441	0.7 ms	15.7 M	28.0 M	7.9 M	1678	1.3 ms	27.6 M
Retz	5.2 M	12.4 M	644	0.8 ms	22.0 M	18.9 M	7.5 M	1616	1.1 ms	24.0 M
Morro Bay	0.6 M	12.0 M	477	0.8 ms	15.8 M	16.3 M	13.7 M	1346	1.1 ms	27.3 M
Meroe	1.9 M	2.0 M	190	0.5 ms	7.8 M	36.4 M	17.5 M	2500	1.9 ms	28.4 M
Endeavor	6.5 M	10.5 M	906	1.1 ms	15.5 M	72.7 M	16.7 M	4956	2.7 ms	33.1 M

Table 3. Rendering performance from overview and closeup viewpoints shown in Figure 7. Samples (Points+Voxels) are both rendered as pixel-sized splats.

performance-sensitive rendering engines (targeting browsers [10, 16, 41, 50], VR [45], or lower-end devices) will limit the number of points/voxels of the same or similar structures to a point budget in the single-digit millions, and then fill resulting gaps by increasing point sizes accordingly.

We implemented the atomic-based point rasterization by Schütz et al. [43], including the early-depth test. Compared to their brute-force approach that renders all points in each frame, our on-the-fly LOD approach reduces rendering times on the RTX 4090 by about 5 to 12 times, e.g. Morro Bay is rendered about 5 to 9 times faster (overview: 7.1ms $\longrightarrow$ 0.8ms; closeup: 6.3ms $\longrightarrow$ 1.1ms) and Endeavor is rendered about 5 to 12 times faster (overview: 13.7ms $\longrightarrow$ 1.1ms; closeup: 13.8ms $\longrightarrow$ 2.7ms). If necessary, the generated LOD structures would allow improving rendering performance further by lowering the detail to less than 1 point per pixel. In terms of throughput (rendered points/voxels per second), our method is several times slower (Morro Bay overview: 50MP/s $\longrightarrow$ 15.8MP/s; Endeavor overview: 58MP/s $\longrightarrow$ 15.5MP/s). This is likely because throughput dramatically rises with overdraw, because if thousands of points project to the same pixel, they share state and can collaboratively update the pixel. At this time, we did not implement the approach presented in Schütz et al.'s follow-up paper [44] that further improves rendering performance by compressing points and reducing memory bandwidth.

6.5 Chunk Sizes

Table 4 shows the impact of chunk sizes on LOD construction and rendering performance. Smaller chunk sizes reduce memory usage but also increase construction duration. The rendering duration, on the other hand, is unaffected by the range of tested chunk sizes. As long as chunks are a multiple or significantly larger than a warp of (32) threads, the warps are able to efficiently fetch a coalesced set of points from global memory. We opted for a chunk size of 1k for this paper because it makes our largest data set – *Endeavor* – fit on the GPU, and because the slightly better construction kernel performance of larger chunks did not significantly improve the total throughput of the system.

In the Meroe data set for a chunk size of 1 000 elements, we observed on overhead of about 2.3% (voxels) to 3% (points), i.e., 97% of the allocated capacity is utilized.

6.6 Memory Consumption

Table 5 shows amounts and memory usage of the data set and LOD structure. Voxels in lower levels of detail and the occupancy grid for voxel sampling increase memory usage by a factor of up to 1.6 in our implementation. Improved implementations could reduce the memory usage of voxels from 16 byte down to 6 byte per voxel (-62.5%) by storing voxel coordinates as 3x1 byte integers relative

Chunk Size	construct (ms)	Memory (GB)	rendering (ms)
500	933.9	17.1	1.9
1 000	734.9	17.2	1.9
2 000	654.5	17.6	1.9
5 000	618.1	18.9	1.9
10 000	611.0	21.0	1.9

Table 4. The Impact of points/voxels per chunk on total construction duration, memory usage for octree data, and rendering times. (Close-up viewpoint of the *Meroe* data set on an RTX 4090)

					LOD Memory Consumption				
Data Set	points	voxels	leaf nodes	inner nodes	points	voxels	occupancy grid	total	Increase
Morro Bay	350 M	128 M	18 676	5 086	5.8 GB	2.1 GB	1.3 GB	9.1 GB	x1.6
Meroe	684 M	209 M	39 648	9 613	11.3 GB	3.4 GB	2.5 GB	17.2 GB	x1.5
Endeavor	796 M	318 M	52 781	11 057	13.2 GB	5.2 GB	2.7 GB	21.1 GB	x1.6

Table 5. The change in memory requirements for the unstructured input point cloud (16 bytes per point) compared to the growable LOD structure including points, voxels (16 byte, each) and occupancy grids (256kb per inner node). Point and voxel counts are for the actual geometry without unused capacity in chunks, while memory consumption also includes unused capacity.

to octree nodes, and by storing colors as 3x1 byte and removing the unused fourth component. Memory usage for occupancy grids could be reduced by removing them on a least-recently-used basis. If needed again at a later time, a node's occupancy grid can be re-created from that node's list of voxels.

7 CONCLUSION, DISCUSSION AND POTENTIAL IMPROVEMENTS

In this paper, we have shown that GPU-based computing allows us to incrementally construct an LOD structure for point clouds close to the rate at which points can be loaded from an SSD, and immediately display the results to the user in real time. Thus, users are able to quickly inspect large data sets right away without the need to wait until LOD construction is finished. There are, however, several limitations and potential improvements that we would like to mention:

- **Out-of-Core**: This approach is currently in-core only and thus requires a GPU with sufficient memory to hold the constructed data structure. For arbitrarily large data sets and/or GPUs with less memory, out-of-core approaches are necessary that flush least-recently-modified-and-viewed nodes to disk. Once they are needed again, they will have to be reloaded – either because the node becomes visible after camera movement, or because newly loaded points are inserted into previously flushed nodes.
- **Compression**: In-between "keeping the node's growable data structure in memory" and "flushing the entire node to disk" is the potential to keep nodes in memory, but convert least-recently-used (and potentially finished) nodes into a more efficient structure. For example, voxel coordinates could be encoded relative to voxels in parent nodes, which requires about 2 bit per voxel, and color values of z-ordered voxels could be encoded with BC texture compression [33], which requires about 8 bit per color, for a total of 10 bit per voxel. Currently, our implementation uses 16 bytes (128 bit) per voxel.
- **Color-Filtering**: Our implementation currently does a first-come color sampling for voxels, which leads to aliasing artifacts similar to textured meshes without mipmapping, or in some cases even bias towards the first scan in a collection of multiple overlapping scans – Same as the first-come sampling version of Schütz et al. [42]. The implementation offers a rendering

mode that blends overlapping points [6, 43], which significantly improves quality, but a sparse amount of overlapping points at low LODs are not sufficient to reconstruct a perfect representation of all the missing points from higher LODs. Thus, proper color filtering approaches will need to be implemented to create representative averages at lower levels of detail. One potential option to do this is the hash map approach by Wand et al. [51].
- **Quality**: To improve quality, future work in fast and incremental LOD construction may benefit from fitting higher quality point primitives (Surfels, Gaussian Splats, ... [29, 38, 52, 57]) to represent lower levels of detail. Considering the throughput of SSDs (up to 580M Points/sec), efficient heuristics to quickly generate and update splats are required, and load balancing schemes that progressively refine the splats closer to the user's current viewpoint.
- **Sparse Buffers**: An alternative to the linked-list approach for growable arrays of points may be the use of virtual memory management (VMM) [37]. VMM allows allocating large amounts of virtual memory, and only allocates actual physical memory as needed (similar to OpenGL's `ARB_sparse_buffer` extension [25]). Thus, each node could allocate a massive virtual capacity for its points in advance, progressively back it with physical memory as the amount of points we add grows, and thereby make linked lists obsolete. We did not explore this option at this time because our entire update kernel – including allocation of new nodes, insertion of points and required allocations of additional memory, etc. – runs on the device, while VMM operations must be called from the host.
- **Order dependency**: For massive data sets, the usefulness of the method may depend on the order of input points. For the Netherlands data set, for example, we would have to analyze the input data to ensure that we focus on loading points that are relevant to the current camera perspective so that users don't have to wait a long time for their region of interest. Right now our approach is limited to in-core where this was less of a concern since the total load time – even for compressed LAZ files that limit load performance to about 25M points per second – is seconds to minutes, rather than hours.

The source code for this paper is available at https://github.com/m-schuetz/SimLOD. The repository also contains several subsets of the Morro Bay data set (which in turn is a subset of San Simeon [36]) in different file formats.

8 ACKNOWLEDGEMENTS

The authors wish to thank *Weiss AG* for the *Chiller* data set; *Riegl Laser Measurement Systems* for providing the data set of the town of *Retz*; *PG&E* and *Open Topography* for the *Morro Bay* data set (a subset of the San Simeon data set) [36]; *Iconem* for the *Northern necropolis - Meroë* data set [24]; *NVIDIA* for the *Endeavor* data set; *Keenan Crane* for the *Spot* model; Bunds et al. [8] for the San Andreas Fault data set; and the *TU Wien, Institute of History of Art, Building Archaeology and Restoration* for the Candi Banyunibo data set [23].

This research has been funded by WWTF project *ICT22-055 - Instant Visualization and Interaction for Large Point Clouds*, and FFG project *LargeClouds2BIM*.

REFERENCES

[1] 2023. P0447R21: Introduction of std::hive to the standard library. C++ Standards Comitee Proposal Paper. https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2023/p0447r21.html Accessed 2023.03.25.

[2] AHN [n. d.]. AHN. https://www.ahn.nl/kwaliteitsbeschrijving, Accessed 2023.06.01.

[3] AHN2 [n. d.]. AHN2. https://www.pdok.nl/introductie/-/article/actueel-hoogtebestand-nederland-ahn2-, Accessed 2021.03.27.

[4] Pascal Bormann, Tobias Dorra, Bastian Stahl, and Dieter W. Fellner. 2022. Real-time Indexing of Point Cloud Data During LiDAR Capture. In *Computer Graphics and Visual Computing (CGVC)*, Peter Vangorp and Martin J. Turner (Eds.). The Eurographics Association. https://doi.org/10.2312/cgvc.20221173

[5] Pascal Bormann and Michel Krämer. 2020. A System for Fast and Scalable Point Cloud Indexing Using Task Parallelism. In *Smart Tools and Apps for Graphics - Eurographics Italian Chapter Conference*, Silvia Biasotti, Ruggero Pintus, and Stefano Berretti (Eds.). The Eurographics Association.

[6] Mario Botsch, Alexander Hornung, Matthias Zwicker, and Leif Kobbelt. 2005. High-quality surface splatting on today's GPUs. In *Proceedings Eurographics/IEEE VGTC Symposium Point-Based Graphics, 2005.* 17–141.

[7] Stefan Bruckner. 2019. Dynamic Visibility-Driven Molecular Surfaces. *Computer Graphics Forum* (2019). https://doi.org/10.1111/cgf.13640

[8] M.P. Bunds, C. Scott, N.A. Toké, J. Saldivar, L. Woolstenhulme, J. Phillips, M. Keck, S. Smith, and M. Ranney. 2020. High Resolution Topography of the Central San Andreas Fault at Dry Lake Valley. Distributed by OpenTopography, Accessed 2023.09.29.

[9] Victor Careil, Markus Billeter, and Elmar Eisemann. 2020. Interactively modifying compressed sparse voxel representations. In *Computer Graphics Forum*, Vol. 39. Wiley Online Library, 111–119.

[10] Cesium [n. d.]. Cesium. https://github.com/CesiumGS/cesium/, Accessed 2021.03.19.

[11] Matthäus G. Chajdas, Matthias Reitinger, and Rüdiger Westermann. 2014. Scalable rendering for very large meshes. *Journal of WSCG* 22 (2014), 77–85.

[12] Liviu Coconu and Hans-Christian Hege. 2002. Hardware-Accelerated Point-Based Rendering of Complex Scenes. In *Eurographics Workshop on Rendering*, P. Debevec and S. Gibson (Eds.). The Eurographics Association. https://doi.org/10.2312/EGWR/EGWR02/043-052

[13] J.D. Cohen, D.G. Aliaga, and Weiqiang Zhang. 2001. Hybrid simplification: combining multi-resolution polygon and point rendering. In *Proceedings Visualization, 2001. VIS '01.* 37–539. https://doi.org/10.1109/VISUAL.2001.964491

[14] Cyril Crassin, Fabrice Neyret, Sylvain Lefebvre, and Elmar Eisemann. 2009. GigaVoxels: Ray-Guided Streaming for Efficient and Detailed Voxel Rendering. In *Proceedings of the 2009 Symposium on Interactive 3D Graphics and Games* (Boston, Massachusetts) *(I3D '09)*. Association for Computing Machinery, New York, NY, USA, 15–22.

[15] Carsten Dachsbacher, Christian Vogelgsang, and Marc Stamminger. 2003. Sequential Point Trees. *ACM Trans. Graph.* 22, 3 (2003), 657–662.

[16] Entwine [n. d.]. Entwine. https://entwine.io/, Accessed 2021.04.13.

[17] Alex Evans. 2015. Learning from failure: A Survey of Promising, Unconventional and Mostly Abandoned Renderers for 'Dreams PS4', a Geometrically Dense, Painterly UGC Game. In *ACM SIGGRAPH 2015 Courses, Advances in Real-Time Rendering in Games.* http://media.lolrus.mediamolecule.com/AlexEvans_SIGGRAPH-2015.pdf [Accessed 7-June-2022].

[18] Enrico Gobbetti and Fabio Marton. 2004. Layered Point Clouds: A Simple and Efficient Multiresolution Structure for Distributing and Rendering Gigantic Point-sampled Models. *Comput. Graph.* 28, 6 (2004), 815–826.

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

[20] Christian Günther, Thomas Kanzok, Lars Linsen, and Paul Rosenthal. 2013. A GPGPU-based Pipeline for Accelerated Rendering of Point Clouds. *J. WSCG* 21 (2013), 153–161.

[21] Kshitij Gupta, Jeff A. Stuart, and John D. Owens. 2012. A study of Persistent Threads style GPU programming for GPGPU workloads. In *2012 Innovative Parallel Computing (InPar).* 1–14. https://doi.org/10.1109/InPar.2012.6339596

[22] Mark Harris and Kyrylo Perelygin. 2017. Cooperative Groups: Flexible CUDA Thread Programming. https://developer.nvidia.com/blog/cooperative-groups/ Accessed 2023.06.05.

[23] U. Herbig, L. Stampfer, D. Grandits, I. Mayer, M. Pöchtrager, Ikaputra, and A. Setyastuti. 2019. DEVELOPING A MONITORING WORKFLOW FOR THE TEMPLES OF JAVA. *The International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences* XLII-2/W15 (2019), 555–562. https://www.int-arch-photogramm-remote-sens-spatial-inf-sci.net/XLII-2-W15/555/2019/

[24] Iconem. [n. d.]. Northern necropolis - Meroë. https://app.iconem.com/#/3d/project/public/6384d382-5e58-4454-b8e6-dec45b6e6078/scene/c038fb6f-c16b-421e-a2ac-f7292f1b1c64/ Accessed 2023.06.05.

[25] The Khronos Group Inc. 2014. ARB_sparse_buffer Extension. https://registry.khronos.org/OpenGL/extensions/ARB/ARB_sparse_buffer.txt/ Accessed 2023.06.01.

[26] Lai Kang, Jie Jiang, Yingmei Wei, and Yuxiang Xie. 2019. Efficient Randomized Hierarchy Construction for Interactive Visualization of Large Scale Point Clouds. In *2019 IEEE Fourth International Conference on Data Science in Cyberspace (DSC).* 593–597.

[27] Maik Keller, Nicolas Cuntz, and Andreas Kolb. 2009. Interactive Dynamic Volume Trees on the GPU. *VMV 2009 - Proceedings of the Vision, Modeling, and Visualization Workshop 2009*, 165–176.

[28] Michael Kenzel, Bernhard Kerbl, Dieter Schmalstieg, and Markus Steinberger. 2018. A High-Performance Software Graphics Pipeline Architecture for the GPU. 37, 4, Article 140 (jul 2018), 15 pages. https://doi.org/10.1145/3197517.3201374

[29] Bernhard Kerbl, Georgios Kopanas, Thomas Leimkühler, and George Drettakis. 2023. 3D Gaussian Splatting for Real-Time Radiance Field Rendering. *ACM Transactions on Graphics* 42, 4 (July 2023). https://repo-sam.inria.fr/fungraph/3d-

gaussian-splatting/
[30] Kevin Kocon and Pascal Bormann. 2021. Point cloud indexing using Big Data technologies. In *2021 IEEE International Conference on Big Data (Big Data)*. 109–119.
[31] Richard E Korf. 1985. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial intelligence* 27, 1 (1985), 97–109.
[32] Oscar Martinez-Rubi, Stefan Verhoeven, M. van Meersbergen, Markus Schütz, Peter van Oosterom, Romulo Goncalves, and T. P. M. Tijssen. 2015. Taming the beast: Free and open-source massive point cloud web visualization. Capturing Reality Forum 2015, Salzburg, Austria.
[33] Microsoft. 2022. BC7 Format. https://learn.microsoft.com/en-us/windows/win32/direct3d11/bc7-format#bc7-implementation/ Accessed 2023.06.09.
[34] Mathijs Molenaar and Elmar Eisemann. 2023. Editing Compressed High-resolution Voxel Scenes with Attributes. In *Eurographics 2023*. Eurographics, Wiley. https://doi.org/10.1111/cgf.14757
[35] Carlos J. Ogayar-Anguita, Alfonso López-Ruiz, Antonio J. Rueda-Ruiz, and Rafael J. Segura-Sánchez. 2023. Nested spatial data structures for optimal indexing of LiDAR data. *ISPRS Journal of Photogrammetry and Remote Sensing* 195 (2023), 287–297.
[36] Pacific Gas & Electric Company. 2013. PG&E Diablo Canyon Power Plant (DCPP): San Simeon and Cambria Faults, CA, Airborne Lidar survey. Distributed by OpenTopography.
[37] Cory Perry and Nikolay Sakharnykh. 2020. Introducing Low-Level GPU Virtual Memory Management. https://developer.nvidia.com/blog/introducing-low-level-gpu-virtual-memory-management/ Accessed 2023.06.01.
[38] Hanspeter Pfister, Matthias Zwicker, Jeroen van Baar, and Markus Gross. 2000. Surfels: Surface Elements As Rendering Primitives. In *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '00)*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 335–342.
[39] Szymon Rusinkiewicz and Marc Levoy. 2000. QSplat: A Multiresolution Point Rendering System for Large Meshes. In *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '00)*. ACM Press/Addison-Wesley Publishing Co., USA, 343–352.
[40] Claus Scheiblauer and Michael Wimmer. 2011. Out-of-Core Selection and Editing of Huge Point Clouds. *Computers & Graphics* 35, 2 (2011), 342–351.
[41] Markus Schütz. 2016. *Potree: Rendering Large Point Clouds in Web Browsers*. Master's thesis. Institute of Computer Graphics and Algorithms, TU Wien, Favoritenstrasse 9-11/E193-02, A-1040 Vienna, Austria. https://www.cg.tuwien.ac.at/research/publications/2016/SCHUETZ-2016-POT/
[42] Markus Schütz, Bernhard Kerbl, Philip Klaus, and Michael Wimmer. 2023. GPU-Accelerated LOD Generation for Point Clouds. https://www.cg.tuwien.ac.at/research/publications/2023/SCHUETZ-2023-LOD/
[43] Markus Schütz, Bernhard Kerbl, and Michael Wimmer. 2021. Rendering Point Clouds with Compute Shaders and Vertex Order Optimization. *Computer Graphics Forum* 40, 4 (2021), 115–126. https://www.cg.tuwien.ac.at/research/publications/2021/SCHUETZ-2021-PCC/
[44] Markus Schütz, Bernhard Kerbl, and Michael Wimmer. 2022. Software Rasterization of 2 Billion Points in Real Time. *Proceedings of the ACM on Computer Graphics and Interactive Techniques* 5, 3 (July 2022), 1–17.
[45] Markus Schütz, Katharina Krösl, and Michael Wimmer. 2019. Real-Time Continuous Level of Detail Rendering of Point Clouds. In *2019 IEEE Conference on Virtual Reality and 3D User Interfaces* (Osaka, Japan). IEEE, 103–110.
[46] Markus Schütz, Stefan Ohrhallinger, and Michael Wimmer. 2020. Fast Out-of-Core Octree Generation for Massive Point Clouds. *Computer Graphics Forum* 39, 7 (Nov. 2020), 1–13.
[47] Zhong Shao, John H Reppy, and Andrew W Appel. 1994. Unrolling lists. In *Proceedings of the 1994 ACM conference on LISP and functional programming*. 185–195.
[48] USGS:3DEP [n. d.]. 3D Elevation Program (3DEP). https://www.usgs.gov/core-science-systems/ngp/3dep, Accessed 2020.09.18.
[49] USGS:Entwine [n. d.]. USGS / Entwine. https://usgs.entwine.io, Accessed 2020.09.18.
[50] Peter van Oosterom, Simon van Oosterom, Haicheng Liu, Rod Thompson, Martijn Meijers, and Edward Verbree. 2022. Organizing and visualizing point clouds with continuous levels of detail. *ISPRS Journal of Photogrammetry and Remote Sensing* 194 (2022), 119–131.
[51] Michael Wand, Alexander Berner, Martin Bokeloh, Philipp Jenke, Arno Fleck, Mark Hoffmann, Benjamin Maier, Dirk Staneker, Andreas Schilling, and Hans-Peter Seidel. 2008. Processing and interactive editing of huge point clouds from 3D scanners. *Computers & Graphics* 32, 2 (2008), 204 – 220.
[52] Tim Weyrich, Simon Heinzle, Timo Aila, Daniel B. Fasnacht, Stephan Oetiker, Mario Botsch, Cyril Flaig, Simon Mall, Kaspar Rohrer, Norbert Felber, Hubert Kaeslin, and Markus Gross. 2007. A Hardware Architecture for Surface Splatting. *ACM Trans. Graph.* 26, 3 (jul 2007), 90–es. https://doi.org/10.1145/1276377.1276490
[53] Michael Wimmer and Claus Scheiblauer. 2006. Instant Points: Fast Rendering of Unprocessed Point Clouds. In *Symposium on Point-Based Graphics*, Mario Botsch, Baoquan Chen, Mark Pauly, and Matthias Zwicker (Eds.). The

Eurographics Association.

[54] Jason C. Yang, Justin Hensley, Holger Grün, and Nicolas Thibieroz. 2010. Real-Time Concurrent Linked List Construction on the GPU. *Computer Graphics Forum* 29, 4 (2010), 1297–1304. https://onlinelibrary.wiley.com/doi/abs/10.1111/j.1467-8659.2010.01725.x _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/j.1467-8659.2010.01725.x.

[55] Jinhyuk Yoon, Sang Lyul Min, and Yookun Cho. 2002. Buffer cache management: predicting the future from the past. In *Proceedings International Symposium on Parallel Architectures, Algorithms and Networks. I-SPAN'02.* 105–110. https://doi.org/10.1109/ISPAN.2002.1004268

[56] Stefan Zellmann, Ingo Wald, Alper Sahistan, Matthias Hellmann, and Will Usher. 2022. Design and Evaluation of a GPU Streaming Framework for Visualizing Time-Varying AMR Data. In *Eurographics Symposium on Parallel Graphics and Visualization*, Roxana Bujack, Julien Tierny, and Filip Sadlo (Eds.). The Eurographics Association. https://doi.org/10.2312/pgv.20221066

[57] Matthias Zwicker, Hanspeter Pfister, Jeroen van Baar, and Markus Gross. 2001. Surface Splatting. In *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '01)*. Association for Computing Machinery, New York, NY, USA, 371–378. https://doi.org/10.1145/383259.383300