DOI: 10.1111/cgf.14877
High-Performance Graphics 2023
J. Bikker and C. Gribble
(Guest Editors)

COMPUTER GRAPHICS *forum*
Volume 42 (2023), Number 8
e14877 *pp. 1 - 12*

GPU-Accelerated LOD Generation for Point Clouds

Markus Schütz[1], Bernhard Kerbl[3], Philip Klaus[2], Michael Wimmer[1]

[1]TU Wien, [2]Austrian Institute of Technology, [3]Inria, Université Côte d'Azur

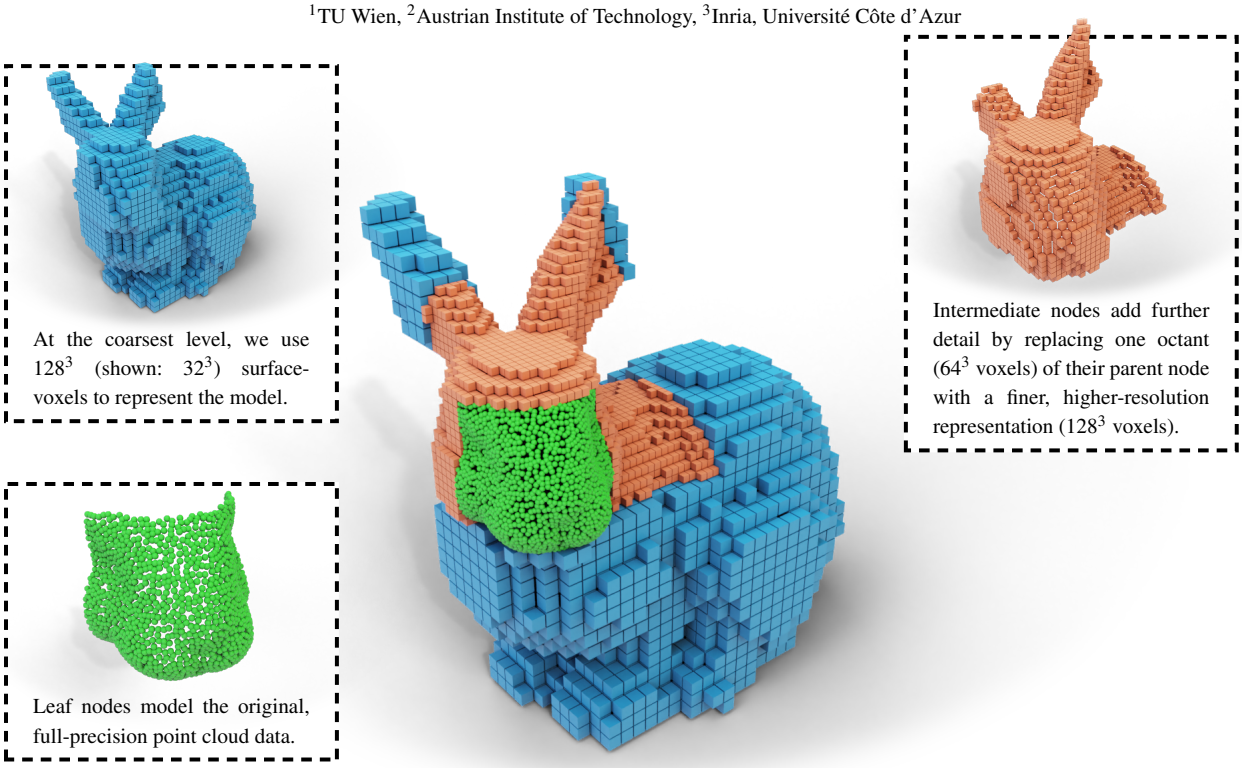


Figure 1: *The data structure is a hybrid voxel-point octree that utilizes color-filtered voxels for lower LODs, but displays the original point data at highest LOD. As most grid cells of each node are empty, surface-voxels are stored as vertices in a node's vertex buffer. During rendering, we aim to rasterize pixel-sized voxels to give viewers the impression that they are looking at full-precision point cloud data.*

Abstract

About: *We introduce a GPU-accelerated LOD construction process that creates a hybrid voxel-point-based variation of the widely used layered point cloud (LPC) structure for LOD rendering and streaming. The massive performance improvements provided by the GPU allow us to improve the quality of lower LODs via color filtering while still increasing construction speed compared to the non-filtered, CPU-based state of the art.*

Background: *LOD structures are required to render hundreds of millions to trillions of points, but constructing them takes time.*

Results: *LOD structures suitable for rendering and streaming are constructed at rates of about 1 billion points per second (with color filtering) to 4 billion points per second (sample-picking/random sampling, state of the art) on an RTX 3090 – an improvement of a factor of 80 to 400 times over the CPU-based state of the art (12 million points per second). Due to being in-core, model sizes are limited to about 500 million points per 24GB memory.*

Discussion: *Our method currently focuses on maximizing in-core construction speed on the GPU. Issues such as out-of-core construction of arbitrarily large data sets are not addressed, but we expect it to be suitable as a component of bottom-up out-of-core LOD construction schemes.*

CCS Concepts

• ***Computing methodologies*** → ***Rendering;***

1. Introduction

Point clouds – 3D models made of colored vertices – are widely used in the geospatial industry, archaeology, construction and other fields that operate on digital twins of real environments and objects. 3D scanning devices such as laser scanners digitize real surfaces in a point-wise fashion, resulting in hundreds of millions to trillions of points [AHN2; 3DEP; ENTW].

Processing and visualizing these massive amounts of points is an ongoing field of research, where advances in rendering hardware and algorithms compete with the ever growing size of the data sets. Compared to triangle meshes, point clouds tend to be significantly larger. While a few textured triangles are sufficient to give the illusion of a detailed and closed surface, thousands to millions of vertex-colored point samples are necessary to represent the same surface with a similar amount of detail and without holes between points upon closer inspection. Although point clouds can, and frequently are, converted to textured meshes, there are use cases and processes that favour the original point cloud data. For example, we may want to produce various different products out of a point cloud (digital elevation models (DEM), digital surface models (DSM), meshes), and we may want to recreate these from scratch with different settings (resolution, quality) or apply improved meshing algorithms in the future. We may also want to measure directly on the original, full-precision scan data, rather than derivatives with potentially reduced or made-up information. Furthermore, creating meshes is a time-consuming and still error-prone process, so operating directly on the source data may lead to faster and more accurate measurements.

The visualization of hundreds of millions of points requires level-of-detail structures that allow the rendering engine to load and display only the smallest subset of data with the biggest impact on visual quality. Important characteristics of LOD structures are a reduction in load times, memory usage, and improved rendering performance as only a fraction of the data is loaded and processed at any given time. LOD structures may also improve the visual quality via precomputed antialiasing (color filtering), similar to mip mapping and virtual texturing.

Layered point clouds (LPC) [GM04] and its variations are one of the most widely used LOD structures for point clouds. LPCs were originally introduced as a binary tree where each node contains a random subset of the point cloud. Lower-level nodes contain coarse subsets of large areas, while higher-level nodes contain increasingly more detailed subsets of smaller areas. During rendering, we draw nodes from coarse to fine until we reach the desired level of detail. More detailed nodes are discarded, as well as all the nodes outside the view frustum.

One of the main issues of LOD structures is that creating them is computationally expensive and takes time. The fastest LPC generation methods have a throughput of about 2.5 to 11 million points per second [KJWX19; BK20; KB21; SOW20], which makes the inspection of typical data sets with hundreds of millions of points tedious and time-intensive.

In this paper, we propose a GPU-accelerated LOD-generation process that creates a hybrid voxel-point-based variation of layered point clouds, using an octree. Our method constructs the octree up to two orders of magnitude faster than the current state of the art and generates color-filtered voxels in lower LODs to improve quality. We propose to use voxels instead of points in lower LODs because quantized integer voxel coordinates offer higher compression rates compared to full-precision point coordinates, and thus reduce transfer times when streaming lower LODs in web browsers. It should be noted, however, that our process is easily modified to support points in lower LODs, if desired.

Our contributions to the state of the art are:

- A hybrid voxel-point-based variation of Layered Point Clouds [GM04] and Modifiable Nested Octree [SW11]) that incorporates color-filtering and enables more efficient streaming of lower LODs due to higher compression rates of quantized voxel coordinates. It can simultaneously also be seen as a point-based variation of the hybrid voxel-triangle structure proposed by Chajdas et al. [CRW14].
- A CUDA-accelerated variation of a fast bottom-up LOD construction scheme for point clouds [SOW20] that is up to two orders of magnitudes faster than its CPU-based predecessor.

2. Related Work

Level-of-Detail methods allow us to render large 3D models that would otherwise be slow to render and/or too large to fit in memory [LRC*03]. LOD methods may offer multiple versions of a model with varying resolutions, spatially partition them so that we can load and render only subsets that are within the view-frustum (view-dependent LOD), and precompute anti-aliased geometry and textures to create higher-quality images with lower processing effort (e.g., mip mapping, virtual texturing). Of special note is Nanite [KSW21], which recently made it possible to efficiently stream and render massive, complex triangle meshes in a game engine. Nanite produces a hierarchy of clusters made up of 128 triangles, and switches between higher- and lower-LOD clusters based on screen-space error metrics. The engine also software-rasterizes small triangles as it was discovered that smaller triangles are rendered faster by directly drawing them to the framebuffer via atomic operations. This, in turn, made it viable for Nanite to aim for lower screen-space errors and draw detailed clusters comprising triangles that are few pixels large. In terms of rendering-performance and quality, Nanite far outperforms basic (not splat-based) point cloud data sets, including ones represented by LOD structures that we generate. We therefore do not challenge Nanite when it comes to games. Our approach is targeted towards scanned point data sets without a notion of a surface (no normals). These easily comprise hundreds of millions, up to hundreds of billions of points, where we offer orders of magnitude faster LOD construction performance to quickly inspect massive amounts of points.

2.1. Point-Based Level-of-Detail Structures

QSplat [RL00] allows displaying large meshes in real-time via point-based rendering of a bounding-sphere hierarchy. The bounding-sphere hierarchy is traversed until a sphere is encountered that is considered to be detailed enough (e.g. a few pixels large), and then rendered as splat with a matching size. Sequential point trees (SPT) [DVS03] is a similar but more GPU-friendly

structure that uses a sequentialized version of a bounding sphere hierarchy. The idea is to store the bounding spheres inside an array, sorted by their level of detail. During rendering, we invoke a draw call for the first X elements of the array, depending on the desired level of detail, and in the vertex shader discard the lower LODs while passing the higher, desired LODs to the rasterizer.

Layered Point Clouds (LPC) [GM04] was the first GPU-friendly as well as view-dependant LOD structure. Being view-dependant is particularly important as it allows us to adjust the level of detail based on camera position, view direction, and distance to the camera. Distant parts of the model are rendered at lower levels of detail, and regions outside the view frustum are culled entirely. LPCs achieve this by distributing points into a binary tree in which each node contains a random subsample of the original point cloud. Lower levels contain coarser subsamples of the whole data set, while higher levels contain increasingly more detailed subsamples of smaller regions.

Since then, variations of LPC proposed improvements to various aspects, such as using different tree structures [WS06; WBB*08; GZPG10; SW11; Sch14; RDD15; OLRS23], faster LOD construction processes [Sch14; MVvM*15; KJWX19; SOW20; BK20; KB21], sampling strategies that affect construction performance and visual quality [SKW19; KJWX19; SOW20; vvL*22], and editing operations such as efficient selection and deletion of points [WBB*08; SW11].

Peak LOD construction performances in terms of throughput were reported to be 2.5 [KJWX19], 2.6 [BK20], and 11.1 million [SOW20] points per second, with the former being an in-core procedure, and the latter two being out-of-core approaches.

This paper largely adapts the CPU-based LOD construction schemes of Schütz et al. [SOW20] into a GPU-based approach, and further extends it by color-filtered lower LODs. Color-filtered lower LODs were already suggested by Wand et al. [WBB*08] in 2008, but to our knowledge we are the first to attempt to construct layered point clouds with color filtering, while simultaneously improving construction times by two orders of magnitude despite the additional computational effort.

In contrast to geometric LOD streaming and rendering, recent work also explores the use of virtual texturing in order to load and utilize only a small subset of a potentially massive amount of texture data [SBMK20], potentially with splat-friendly compression [STS*21].

2.2. Voxelization and Voxel-Based Level-of-Detail Structures

In this paper we focus on surface voxels, i.e., models where each voxel is part of a 3D model's surface. In contrast, solid or volumetric voxel models also specify whether voxels are inside/outside of a 3D model (e.g., Minecraft and similar games), or the density or similar property of a volume at a given position (e.g., CT and MRI data).

Voxelization: Fang and Chen propose a voxelization approach that utilizes the hardware rasterizer to render slices of a 3D model. The pixels of each rendered slice then represent the voxels of the corresponding volume [FC00]. While efficient, the rasterizer may overlook parts of the surface that intersect pixels but not the sample point within a pixel. Schwarz and Seidel demonstrate a CUDA-based approach for the efficient and conservative or 6-separating surface voxelization of triangles on a 1024^3 sampling grid [SS10]. Conservative meaning that all voxels that are intersected by a triangle are identified, and 6-separating meaning that a smaller amount of intersected voxels that suffices for a grap-free voxelization is identified. (Note: graphics APIs nowadays also support conservative rasterization, potentially resolving limitations that [FC00] had back then).

Ray-based LOD: Sparse-Voxel-Octrees (SVO) are a ray-tracable hierarchical structure [LK11]. Each node represents a voxel with potentially 8 child voxels. During rendering, we traverse from coarser to finer voxels until we either encounter a leaf node, or a voxel is small enough to stop the traversal and display it. Since cube-shaped voxels are not an ideal representation for curved surfaces, the authors also propose contour-based voxel representations that align with a surface's orientation. Crassin and Green demonstrate voxelization algorithms for the GPU that can also generate such SVOs by first expanding the hierarchy in a top-down fashion, then populating the leaf nodes with voxels, and eventually inner nodes via mip mapping [CG12]. Kämpe et al. later proposed directed acyclic graphs (DAG) as a more efficent encoding for SVOs that reduces the required memory by 1-2 orders of magnitude [KSA13]. Beart et al. [BLD13] and Pätzold and Kolb [PK15] introduced efficient out-of-core SVO construction procedures (e.g. input: San Miguel, 10M triangles, 4096 voxel grid, **SVO**: 153s, **Baert**: 26s, **Pätzold**: 12s).

GigaVoxels [CNLE09] utilize an N^3 tree with M^3 bricks – Each node splits into N^3 child nodes (an octree in case of $N = 2$), and nodes comprise bricks of M^3 voxels, with M around 32. At runtime, nodes are traversed to find the voxel blocks with appropriate resolution which are then ray-marched. In addition, GigaVoxels can also filter colors from multiple brick levels to reduce aliasing artifacts via voxel-based mip mapping. Some similarities to the structure we generate are: we use an octree, hence $N = 2$. We use chunks/bricks of $M^3 = 128^3$. The main difference is that we intend to rasterize sparse surfacic data, so instead of volumetric bricks, we generate vertex buffers with voxel coordinates and colors.

Rasterization-based LOD: FarVoxels [GM05] uses a BSP-tree of view-dependent voxels in inner nodes and triangles in leaf nodes. View-dependent meaning that shading parameters for various view directions are computed during construction of the LOD structure and stored in each voxel. Chajdas et al. also use a hybrid voxel-triangle LOD structure [CRW14] but using an octree where each node represents a chunk of 256^3 voxels, and leaf nodes store original triangle data. During rendering, the chunks with the lowest yet sufficient resolution are rendered. Once the viewer zooms in, however, even the leaf node's voxel resolution may not be sufficient. In that case, the original triangle data contained in that leaf node will be rendered instead. The structure that we generate in this paper closely matches the structure of Chajdas et al., with two minor differences: First, instead of 256^3 cells in each node/chunk, we decided to use 128^3 for each node as it reduces the amount of memory required during construction, as well as the amount of voxels in each octree node. Second, since we use point clouds, leaf nodes

contain the original point instead of triangle data, and they do not contain any voxels.

2.3. High-Quality Rendering of Point Clouds

Point-based rendering, especially when combined with levels of detail, is subject to several types of aliasing issues that reduce quality. Splat-based methods such as QSplat [RL00], Surfels [PZvBG00], AutoSplats [PJW12] and Botsch et al. [BHZK05] improve quality by rendering points as disks that are aligned to a surface's orientation, and by blending overlapping disks via weighted averages. Hierarchical methods like QSplat [RL00] and Wand et al. [WBB*08] may also compute representative geometry at lower levels of detail by computing larger-scale splats or averaging normals. QSplat additionally applies color filtering by computing averages of collapsed points. On the other hand, layered point cloud methods – to which this paper belongs – typically focus on rendering and LOD construction speed rather than quality, and as such did not explore color filtering at lower LODs. With exceptions like Wand et al. [WBB*08], they also focus on non-oriented points without normals and therefore draw points with screen-facing rather than surface-oriented shapes. In this paper, we show that basic hierarchical color filtering via averaging (like QSplat does) can be implemented very efficiently on the GPU for layed point clouds.

2.4. Counting Sort

Counting sort is a linear-time $O(n)$ integer sorting algorithm that is capable of efficiently sorting large amounts of integers with just two iterations over the entire input data [Knu98; CLRS01]. In contrast, comparison-sort algorithms such as quicksort or merge sort have a complexity of $O(n \log n)$, and therefore the number of times each element of the input is accessed is not constant and increases logarithmically with the total length of the input array [Knu98]. The improved performance of counting sort comes with two limitations: First, it is limited to sorting integer keys. Second, the keys have a limited range because the algorithm maintains a counter for each potential key. In this paper, we use counting sort to partition points into leaf nodes (the buckets) using the target node's voxel grid coordinate as the integer key.

3. Data Structure

The generated data structure is a multi-resolution octree where leaf nodes contain the original points, and inner nodes are made up of voxels at coarse, level-dependent resolutions, as shown in Figure 1. Unlike fine-grained acceleration structures that have ray-tracing in mind (e.g. SVOs), our structure follows the rasterization-friendly layered-point-cloud scheme [GM04]. As such, each node comprises a larger batch of points or voxels, which can be efficiently rasterized in a single draw call. To be precise, leaf nodes store up to $T = 50$k points, and inner nodes contain surface voxels that match an inscribed 128^3 grid. Since we target surfacic 3D models, the number of occupied voxels in such a grid is usually closer to 128^2, so instead of storing the entire, sparsely populated grid, we store a vertex buffer comprising coordinates and colors of occupied voxels. An important difference between our structure and regular LPCs is

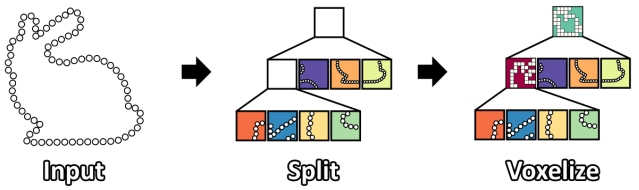


Figure 2: *LOD Construction Overview: First, points are split into leaf nodes of an octree with hierarchical counting sort. Then, inner nodes are populated bottom-up with voxelized representations of their children.*

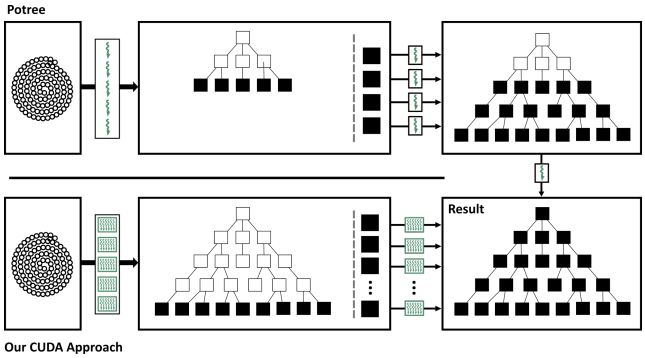


Figure 3: *Parallelism in Potree vs our CUDA-based approach. Potree uses all CPU threads to split the octree into chunks of 10M points. Chunks are then further split and converted into an octree, using one thread per chunk. A single thread then merges all chunks and populates the remaining coarsest LODs. Our CUDA-based approach directly splits into leaf nodes with about 50k points utilizing all threads in all workgroups, and it then populates inner nodes utilizing one workgroup per node.*

that LPCs distribute all points to all nodes of the entire tree structure, whereas our structure partitions all points into the leaf nodes. LPCs do not create additional or duplicate data – the original point cloud is recovered by merging points in all nodes of all levels. Our structure, on the other hand, creates new voxel data in inner nodes that is meant to faithfully represent the original model at specific resolutions. In that regard it is similar to the structure of Wand et al. [WBB*08], who store original point data in leaf nodes and surfels in inner nodes. It is also similar to the structure of Chajdas et al. [CRW14], who store triangles in leaf nodes and chunks of 256^3 voxels in inner nodes.

4. LOD Construction

Our CUDA-based LOD construction method creates the aforementioned LOD structure in two main steps: First, we partition all points of the input data set into octree leaf nodes with at most $T = 50k$ points. Second, we populate inner nodes with lower-resolution voxel models from bottom-up. By splitting into leaf-nodes with T points, we generate spatially constrained work packages that can be processed by numerous workgroups in parallel.

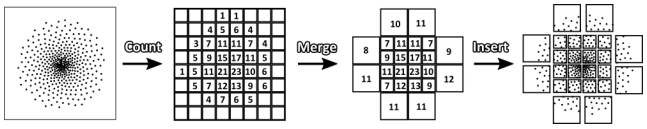


Figure 4: *We directly create an up to 8 levels deep octree using a hierarchical counting sort approach. Counting and merging gives us the full octree hierarchy with knowledge about the number of points in leaf nodes. We then allocate the right amount of memory and insert points into the respective leaves.*

4.1. Split Input into Leaf Nodes

To partition the input data into octree leaf nodes with at most T points, we adapt the hierarchical counting sort approach of Schütz et al. [SOW20]. Hierarchical counting sort allows us to partition a point cloud into an octree with *depth* levels by iterating over all points just twice – once to count and a second time to move the points to the sorted location in memory. For this CUDA approach, we suggest an initial octree depth of 8 (recursively extended, if needed).

4.1.1. Counting

At octree level 8, we have $(2^{level})^3 = (2^8)^3 = 256^3$ potential leaf nodes, so we create a corresponding counting grid with 256^3 cells. We then iterate through all points, project them into the counting grid and atomically increase the counter of the corresponding cell.

4.1.2. Merging

Although we want our leaf nodes to contain at most T points, we also do not want them to contain too few points, so in this step, we merge all groups of 2x2x2 cells that contain less than T points, as shown in Figure 4. For this purpose, we create a pyramid of counting grids with a size of $(2^{level})^3$ for each octree level. Whenever a 2x2x2 group of cells contains less than T points, we store the sum of points in the next coarser counting grid, and set all involved cells of the current level to 0. In addition to that, we also flag the cell at the coarser level as *unmergeable* if we were not able to merge it because the sum of points exceeded the threshold. Furthermore, if a 2x2x2 group of cells contains at least one cell that was previously marked as *unmergable*, then the whole group becomes unmergeable and is also flagged as such in the coarser level.

4.1.3. Creating Target Pointers

Each non-zero value in the counter pyramid represents an octree node. Positive integer values represent a leaf node and the amount of points they contain, while cells that were flagged as *unmergeable* represent inner nodes that will be populated with a currently unknown amount of voxels at a later time. We now create a list of all leaf and inner nodes, and in case of leaf-nodes, we allocate sufficient memory for the points. Additionally, we map the counter pyramid into a pointer pyramid, where each cell points to the respective leaf node (counter > 0), inner node (counter = *unmergeable*), or null if the cell is empty (counter = 0). The process of creating the target pointers corresponds to creating the prefix sums in regular counting sort. In both cases, the results are used to compute the sorted position of points in the final step.

4.1.4. Insert

The last step of the node-wise sorting procedure iterates over all points a second time, projects them again to cells of a 256^3 grid, but this time each point reads the node pointer from the recently created pointer pyramid. If we encounter a non-null pointer, we found the leaf node that this point belongs to, and add it. If we encounter a null pointer, then the cells at this octree level were merged into a lower level, i.e., we iterate upwards in our pointer-pyramid until we eventually encounter the leaf-node pointer.

4.1.5. Recursive Partitioning (If Necessary)

After these steps, all points are sorted into an octree with a depth of at most 8 levels, or fewer if small nodes were merged. However, 8 levels are not always sufficient, especially with massive amounts of points, or irregularly distributed points with dense clusters in some regions (e.g. teapot-in-a-stadium scenario). This issue is easily addressed by repeating the sort procedure for all leaf nodes that are still too large. Additional iterations can also adjust the sort-depth and counting-grid size to fit the computational effort.

In our case, we implemented one level of recursion (sufficient for all test data sets) in-between the initial counting and the merging step. After the initial counting procedure, some cell counters may exceed the given threshold T. For each such cell, we generate an additional counting grid pyramid with an octree depth of 4, comprising $(2^4)^3 = 16^3$ cells at the highest level and extending the octree from the initial up to 8 levels to up to 12 levels. For all cells that exceeded the threshold, we replace the counter in the main counting grid with a pointer to the newly created extended counting grid. We then iterate through all points again, project them to the original grid, and if a point hits a cell with a pointer, we follow the pointer and update the counters in the extended counting grid.

The merging, pointer creation and insertion steps are modified correspondingly. The merging step additionally merges the counters in the extended counting grids, and the pointer creation step also creates octree nodes and pointers for the extended grids. The insertion step still projects points to the main grid with depth 8, size 256^3, but additionally checks whether the targeted cell contains a regular node pointer or a pointer to an extended grid. If it's a pointer to a node, we proceed as usual and add the point to that node. If it's a pointer to an extended grid, we traverse into that grid and look there for the corresponding octree node pointer.

4.2. Voxel Sampling

After partitioning the input point cloud to the leaf nodes of an octree, we proceed to populate lower LODs from bottom up with voxelized representations of higher LODs. Voxels in each inner node are created by projecting points and voxels from its child nodes into that node's 128^3 voxel sampling grid. Afterwards, the voxels are serialized into an array and the sampling grid is no longer needed and discarded (or reused to process other nodes). We start the voxelization procedure by creating a list of the bottom-most empty inner nodes with exclusively non-empty child nodes, and then populate each node in this list with a coarser, voxelized representation of its child nodes, using one workgroup per node. This process of finding

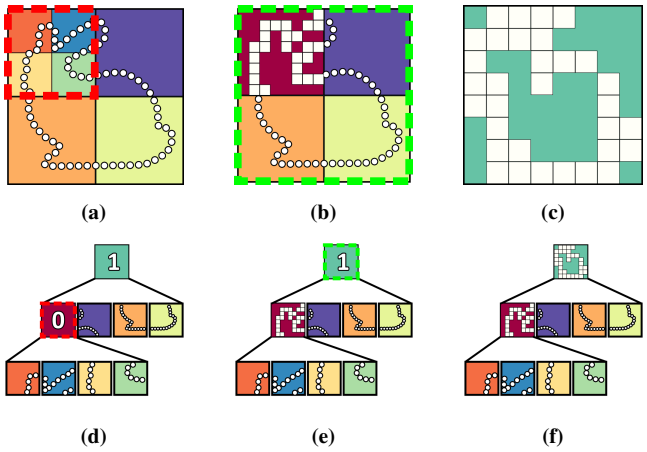


Figure 5: *Creating lower LODs: (a+d) The split pass partitioned all points into leaf nodes. Node* 0 *(dashed red) will be filled with a voxelized representation of all its children. (b+e) Node* 1 *(dashed green) is populated next with a coarser voxelized representation of voxels and points from its child nodes. (c) The voxelized root node. (f) Tree hierarchy showing leaf nodes with points and inner nodes with voxels.*

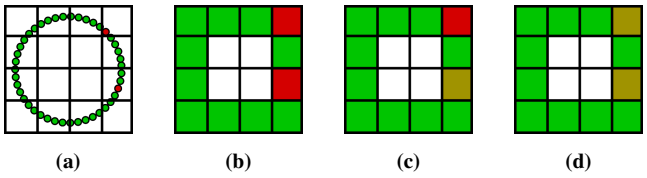


Figure 6: *(a) Samples and sampling grid. (b) Selecting a single sample per cell leads to a disproportional representation of outliers. (c) Averaging within a cell can improve color quality through proportional representation. Still susceptible to outliers from partial intersections. (d) Weighted averaging of samples within a distance, including those in adjacent voxels, further improves quality.*

the bottom-most empty nodes and populating them is repeated until we eventually process the root node and thereby finish the LOD construction.

Like splitting, the entire voxel-sampling kernel is implemented inside a single CUDA kernel. Unlike splitting, however, it makes heavy use of per-workgroup processing of work packages. Each workgroup fetches and processes one node at a time, until all nodes up to the root are populated with voxels. Reasons for matching nodes with workgroups are: We only need to allocate one sampling grid per workgroup; nodes frequently have less points or voxels than we have available GPU threads, so utilizing more threads is wasted effort; and we can take advantage of shared memory (L1 cache). Furthermore, each workgroup requires a considerable amount of global memory for the voxel sampling grid, so in order to minimize the amount of required memory while still utilizing all streaming multiprocessors, we launch exactly one workgroup per SM.

There are several possible sampling strategies that can be used to create a lower-resolution voxel representation out of the points and voxels in child nodes. They all have in common that after fetching a node to be voxelized, they iterate through the points and/or voxels of the 8 child nodes, project them to a sampling grid with a dimension of 128^3 cells, and finally extract the voxel samples from the grid and store them inside the node. They differ in the details of the sampling grid and the extraction process. We investigated the following sampling strategies, which provide trade-offs in construction performance and quality:

4.2.1. Sampling Strategy: First-Come-First-Serve

This fastest approach accepts the first sample (point or voxel) in each sample grid cell. The sample grid of the node we currently populate has a size of 128^3, but this approach processes each of its 8 child nodes sequentially, thus we only need to use and reuse a 64^3 grid for each child. The grid is implemented as a bitmask in shared memory, requiring $64^3 = 262$k bits, equal to 32 KB, which nicely fits into the limit of 48 KB static shared memory on CUDA devices. We also allocate a temporary list of accepted samples with a static capacity of C (e.g., 200k – more than we ever expect a node to accept) and a counter that tracks how many samples we accepted. This temporary, large list is used as an intermediate storage for a node's voxels, because we do not know the exact amount of required memory for each node in advance. The list is reset after each processed node by setting the counter to 0.

Each workgroup then fetches a node from a list of unprocessed nodes, iterates through that node's 8 children, first clearing the sample grid at the start of iteration, then projecting the child's samples (points or voxels) to the grid and atomically setting the corresponding cell's bit to 1 via *atomicOr*. The return value of *atomicOr* reports the previous value – if the cell's bit was previously 0 and is now 1, then the sample is the first processed sample to occupy that cell. In that case, we accept it and add it to a temporary list of accepted samples. If it is not the first (the corresponding bit was already 1), we ignore it. After iterating through all child nodes and their samples, we create an array of voxels as large as the amount of accepted samples to the currently processed node. The accepted samples are then voxelized by quantizing their coordinates to the 128^3 sampling grid, and storing them in the newly allocated array.

4.2.2. Sampling Strategy: Random

For each cell in our voxel sampling grid, we accept one random sample (point or voxel) from the child nodes. Choosing the random sample efficiently is done by computing a random number for each sample, encoding random number and sample index into a 32 bit integer, and then using atomicMax on the sample grid cell to retain the sample with the largest random value, as shown in Listing 1.

```
1 uint32_t encoded =
2     (randomNumber & 0xfff00000) |
3     (sampleIndex  & 0x000fffff);
4 atomicMax(&voxelGrid[voxelIndex], encoded);
```

Listing 1: *CUDA-pseudocode that selects one random sample and stores its ID inside a voxel cell.*

Since we now require 32 times as many bits per cell (for the 32 bit integer instead of 1 bit occupancy), the sampling grid does not

fit in shared memory anymore. Instead, we allocate one 128^3 voxel sampling grid per workgroup in global memory, which is reused for every node a workgroup processes. The amount of global memory required for each workgroup is $sizeof(uint32\_t) * 128^3 = 8\text{MB}$. To minimize the total amount of global memory reserved by all workgroups, we only launch one workgroup per streaming multiprocessor for a total of $SMs * sizeof(uint32\_t) * 128^3$ bytes, which is equal to 687MB on an RTX 3090 with 82 SMs. Similar to the first-come strategy, we track whenever an empty cell is filled with a sample, but since the accepted sample in a cell may randomly change, we create a list of occupied voxels instead of a list of accepted samples. After all points and voxels in the child nodes were projected to the sampling grid, we allocate a list of voxels with a size equal to the number of occupied voxel cells for the current node; then iterate through the list of occupied voxels; retrieve the indices of the accepted samples from the voxel grid; and finally store the respective sample in the node's list of voxels. Note that each sampling grid (128^3 cells) only needs to be fully cleared once at the beginning, but for reuse we only need to clear the voxels that were occupied (closer to 128^2 cells).

4.2.3. Sampling Strategy: Cell-wise Average

A single-sample color value (random, first-come, etc.) is a poor representation of all the higher-LOD samples that a voxel should represent, as shown in Figure 6b. This strategy generates a more representative color by computing the average value from all samples that are projected to a voxel cell, as shown in Figure 6c. For this strategy, each workgroup allocates a 128^3 sampling grid in global memory. Each cell stores the sum of red, green and blue values, as well as a counter of samples that contributed, thus each workgoup requires $4 * sizeof(uint32\_t) * 128^3$ bytes, i.e., 33MB. Since we spawn one workgroup per streaming multiprocessor, the kernel requires a total of $SMs * 33$ MB, about 2.7 GB on an RTX 3090 with 82 SMs. Like the random strategy, this strategy also maintains a list of occupied voxel cells and corresponding counter.

Each workgroup then fetches the next unprocessed node, iterates through all samples in that node's children and projects them to the sample grid. The RGB values of the sample are atomically added to the RGB values of the corresponding sample grid cell, and the counter of the sample grid cell is incremented by one. The first sample that incremented a cell's counter also adds that cell's voxel index to the list of occupied voxels. Afterwards, we allocate sufficient memory for an array of occupied voxels, then iterate through the list of occupied voxels, retrieve that voxel's RGB values and counter and compute the arithmetic average. The voxel coordinate and average color are then stored inside that node's list of voxels. As with the random sampling strategy, we only need to clear the occupied voxels in the sampling grid.

4.2.4. Sampling Strategy: Neighborhood Weighted Average

Computing the arithmetic mean of colors within a voxel is fast and improves quality, but it still suffers from aliasing issues, especially when a voxel captures a single sample that strongly differs from adjacent samples, as shown in Figure 6c. This strategy further improves quality by computing the weighted average of all samples within a certain distance to the center of a voxel, as shown in Figure 6d. The closer to the voxel center, the higher the contribution. In this paper, we use a simple linear weight function that cuts off at a distance of 1 (=width of a cell):

```
distance = length(samplePosition - voxelCenter)
weight   = clamp(1 - distance, 0, 1)
```

Coordinates are relative to the voxel sampling grid, i.e., $[0, 128)$. The given weight function evaluates to 1 if the sample position is equal to the target voxel position, or smaller than 1 otherwise. If a sample is farther than one cell's width away from the voxel center, the weight is cut to zero. Therefore, each sample may affect at most 2x2x2 surrounding voxels and we need to project each sample to 8 surrounding voxels, compute the corresponding weights, and add the weighted color values as well as the weight itself to the voxel grid using atomicAdd.

Note that adding colors and weights to all adjacent voxels would result in the generation of new voxels that are not backed by actual geometry samples, and therefore dilate the model. To avoid dilation, we first flag all voxels that contain an actual sample from a child node, and only add colors and weights to voxels that were flagged. Like the cell-wise average strategy, the first sample that flags a voxel adds that voxel's index to a temporary list of accepted voxels.

Next, we allocate sufficient memory for the currently processed node's voxel buffer based on the amount of accepted voxels. The voxel-indices in the temporary list of accepted voxels are used to extract the corresponding sum of colors and weights from the grid, the weighted is average computed via $\frac{sumOfColors}{sumOfWeights}$, and the result is stored in the node's voxel buffer. To clear the entire voxel sampling grid, it suffices to clear the accepted voxels as only the flagged, and therefore accepted voxels were modified.

The result of the LOD construction procedure is a voxelized LPC [GM04] (or voxel-point-based version of Chajdas et al. [CRW14]) where leaf nodes comprise up to T points, and inner nodes comprise lists of voxels that were generated by downsampling on a 128^3 sampling grid.

5. Rendering

The proposed voxelized LPC structure can be rendered in largely the same manner as other layered point-cloud approaches [GM04; SW11; SOW20] and high-performance point rasterization [SKW21; SKW22]. Each voxel is treated as a point, and we aim for a level of detail where voxels are roughly pixel-sized. We then traverse the octree, invoke draw calls for all nodes that are within the view-frustum and whose projected bounding box is larger than about 64 pixels (matching the voxel-grid size of 128^3, the goal to render pixel-sized voxels, and ensuring that child nodes are rendered if a node becomes larger than 128 pixels).

The main difference to other LPC approaches stems from the fact that LPCs are traditionally additive LOD approaches, meaning that higher levels do not replace lower levels – they are added and rendered together. Our voxelized LPC, on the other hand, is a replacing LOD structure where higher levels of detail completely replace lower levels, which has the advantage that lower-LOD voxels better represent the color values at their given level of detail and catch radius. As for rendering, we invoke one draw call for each

visible node and pass the respective node as a uniform. The vertex shader then checks whether the currently processed vertex/voxel is located in an octant with a visible child node. If it is, the voxel is discarded because the child node will replace it with finer voxels. If there is no visible child node, the voxel is drawn.

6. Evaluation

The proposed method was implemented in C++ and CUDA, and evaluated with the test data sets shown in Figure 7. The data sets were chosen to demonstrate that our method works on various types of data, including mostly 2.5D-like aerial LIDAR scans, but also on more complex objects with higher depth complexity (hidden surfaces) and data sets where some regions have a much higher point density (teapot-in-a-stadium scenario). Kernel run times were measured with *cuEventElapsedTime*. Only LOD construction times on the GPU were measured, I/O and host-device copies are not included. For each measure, we ran the construction process 5 times and report the median (to avoid highly volatile cold-start times).

The evaluation was conducted on two GPUs for our CUDA-based approach, and one CPU to compare with the CPU-based state of the art (Potree [SOW20]). The GPUs have similar specs and performance, but the *RTX A6000* has double as much memory which we require for the in-core LOD construction of the largest data set.

- NVIDIA RTX 3090 24GB
- NVIDIA RTX A6000 48GB
- AMD Ryzen 7 2700X (8 cores), 64GB RAM, Windows 10.

6.1. LOD Construction Performance

Table 1 compares LOD construction performance of our CUDA-based method to the CPU-based method of Potree. Potree offers a blue-noise sampling strategy that ensures a minimum distance between points at lower LODs, and a random sampling strategy that picks one random point per grid cell. The random sampling strategies of Potree and our CUDA approach are largely equivalent and therefore comparable. Since neither the blue-noise, nor the random sampling strategy of Potree compute color-filtered values, and because the blue-noise samples are not directly comparable to the voxelized samples of our CUDA approach, we always compare performances to Potree's fast random-sampling strategy. Furthermore, Potree is designed as an out-of-core application that simultaneously reads point data from disk, processes already loaded data, and writes result to disk. To increase the fairness of comparing processing times of our in-core implementation to Potree's out-of-core approach, we run Potree from a RAM-disk to ensure that disk I/O is not the limiting factor of its LOD construction process. We only had a mid-range CPU available for benchmarking Potree. Online benchmark repositories indicate that a high-end CPU with a similar price as our RTX 3090 – the *AMD Ryzen Threadripper 3960X* – is about three times faster as our *AMD Ryzen 7 2700X* [CPU].

The results in Table 1 indicate that GPGPU-based LOD construction approaches can be about **$\times$200 to $\times$300 times faster** compared to CPU-based approaches, with comparable quality (random). We also found that a first-come strategy further improves performance to about **$\times$400** with no apparent degradation of quality for our test data sets, but with potential quality reduction in scan-wise ordered terrestrial laser scans, as discussed in section 6.2. Finally, our GPU-based method was able to construct LOD structures with higher quality **$\times$80 (weighted, in neighborhood) to $\times$200 (within a single cell) faster** than the fast random sampling strategy of Potree.

6.2. Quality

Figure 8 illustrates differences in quality between different sampling strategies. *First-come* and *random* sampling suffer from strong aliasing artifacts that manifest as noisy images that sparkle during motion – similar to textured meshes without mip maps. In many data sets, both sampling strategies deliver similar results, but *first-come* spectacularly fails for terrestrial laser scans where multiple scans with different exposure overlap. In that case, the random strategy appears better as it provides a uniform mix of all scans, rather than clusters of varying scans. *Average* sampling within a single voxel provides significant improvements – it reduces noise, makes high-frequency features such as text more readable, and it removes sparkling artifacts during motion. However, it still suffers from numerous outliers due to sampling issues shown in Figure 6. *Weighted* average over adjacent cells removes these outliers and provides homogeneous, high-quality results similar to textured meshes with mip maps. Table 2 reports quantitative image error metrics for each method relative to high-quality ground truth renderings. Each metric is computed as the average of 9 close-ups and 9 zoomed-out views across 3 scenes (*Retz*, *Palmyra*, *Saint Roman*).

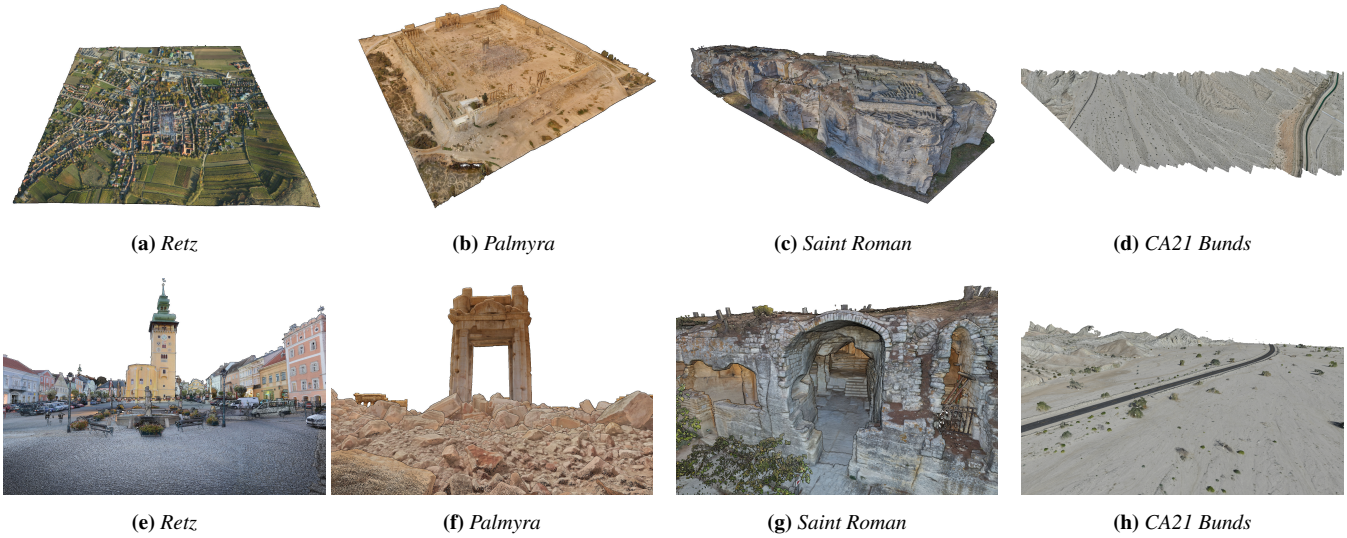
Table 2: *Obtained image error metrics (PSNR↑, SSIM↑, LPIPS↓) for close-up and zoomed-out views with different filtering methods. Screenshots with LOD rendered point sets are compared to the original data set, all rendered with high-quality shading that blends overlapping points [SKW21]. The results demonstrate that average and weighted samples are a significant better representation at lower LODs (zoomed-out).*

Method	Close-up Views			Zoomed-out Views		
	PSNR	SSIM	LPIPS	PSNR	SSIM	LPIPS
Random	32.60	0.957	0.035	23.65	0.756	0.188
Average	33.38	0.969	0.024	27.52	0.894	0.079
Weighted	**33.69**	**0.972**	**0.022**	**28.18**	**0.912**	**0.073**

6.3. Rendering

For rendering performance, we would like to refer to Schütz et al. [SKW21; SKW22] who implemented an optimized compute-shader based software rasterizer for both, brute-force rendered non-structured points, and for layered point clouds with the same configuration that we construct (octree with 128^3 sampling grid in inner nodes). They render a model comprising 529 million points in a virtual reality setting, which is then reduced to 21.4 million points in 2k octree nodes after evaluating the visibility for a given viewpoint. Drawing these 21.4 million points into two 6.8 megapixel framebuffers (one for each eye) takes a total of 8.3 ms.

We expect the same performance for the same amount of rendered samples (points+voxels) since the structure is essentially



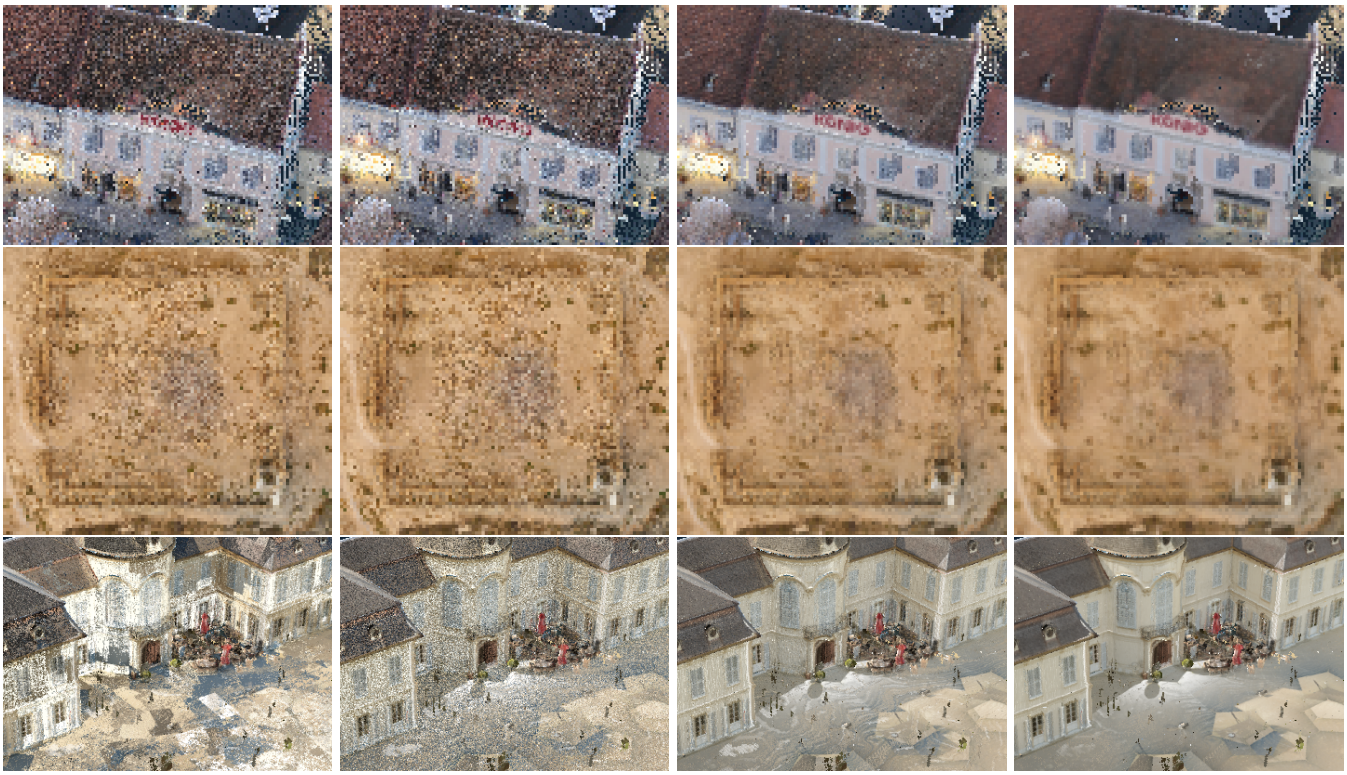
(a) *Retz* **(b)** *Palmyra* **(c)** *Saint Roman* **(d)** *CA21 Bunds*

(e) *Retz* **(f)** *Palmyra* **(g)** *Saint Roman* **(h)** *CA21 Bunds*

Figure 7: *Four data sets that challenge different aspects of the LOD construction process. Retz and Palmyra feature a teapot-in-a-stadium scenario (much higher point densities in regions of interest), Saint Roman comprises scans of outside as well as interior rooms, CA21 Bunds features a large amount of LIDAR data.*

				Ours				Potree		
	Data Set	points	GPU	split	voxelize	total	MP/s	total	MP/s	Speedup
random	Retz	145 M	RTX 3090	20.0	30.1	50.1	2 906	13 130	11	× 262
	Palmyra	258 M	RTX 3090	36.0	67.4	103.4	2 504	21 269	12	× 205
	Saint Roman	547 M	RTX 3090	75.3	137.7	213.0	2 569	43 570	13	× 204
	CA21_Bunds	976 M	RTX A6000	133.1	131.6	264.8	3 685	85 651	11	× 323
first-come	Retz	145 M	RTX 3090	20.8	12.8	33.7	4 319	13 130	11	× 389
	Palmyra	258 M	RTX 3090	37.1	19.0	56.1	4 619	21 269	12	× 379
	Saint Roman	547 M	RTX 3090	77.1	34.6	111.7	4 899	43 570	13	× 390
	CA21_Bunds	976 M	RTX A6000	134.5	60.9	195.4	4 994	85 651	11	× 438
average	Retz	145 M	RTX 3090	19.9	46.7	66.7	2 181	13 130	11	× 196
	Palmyra	258 M	RTX 3090	36.4	86.3	122.7	2 110	21 269	12	× 173
	Saint Roman	547 M	RTX 3090	75.6	173.4	248.9	2 198	43 570	13	× 175
	CA21_Bunds	976 M	RTX A6000	133.1	227.3	360.5	2 707	85 651	11	× 238
weighted	Retz	145 M	RTX 3090	20.7	123.8	144.5	1 006	13 130	11	× 90
	Palmyra	258 M	RTX 3090	37.3	231.6	268.9	962	21 269	12	× 79
	Saint Roman	547 M	RTX 3090	75.2	477.7	552.9	989	43 570	13	× 79
	CA21_Bunds	976 M	RTX A6000	133.8	638.5	772.3	1 263	85 651	11	× 111

Table 1: *LOD construction times of our CUDA approach compared to Potree [SOW20]. Four sampling strategies were implemented and evaluated in CUDA, and compared to Potree's fastest* random *sampling strategy. All timings in milliseconds, throughput in million points per second (MP/s).*



(a) *First-Come* **(b)** *Random* **(c)** *Average* **(d)** *Weighted*

Figure 8: *Quality of the four sampling strategies. Differences between first-come and random are negligible in our performance test data sets, but we added* Niederweiden *to the quality evaluation where it spectacularly fails. The average strategy significantly reduces noise and improves legibility of high-frequency features such as text, but still exhibits some noise/outliers due to sampling issues described in Figure 6. Weighted averages over voxel-boundaries addresses this issue and further improves quality.*

identical and voxels are treated as points. The difference is that the voxels we generate slightly increases the number of samples as they are additional redundant, representative data. In particular, Retz comprises 145M points and generates an additional 57M voxels (+39%), Palmyra comprises 259M points and generates an additional 95M voxels (+37%), and Saint Roman comprises 547 points and generates an additional 176M voxels (+32%). The views in the top row of Figure 7 with a minimum node size of 64 pixels and a framebuffer size of 2560x1140 pixel require: Retz(808 nodes, 5.2M points + 17.3M voxels), Palmyra(567 nodes, 3.8M points + 8.7M voxels) and Saint Roman(1.1k nodes, 6M points + 23.4M voxels), which is comparable to the benchmark of Schütz et al. [SKW22].

7. Conclusion and Discussion

We have shown that GPUs can accelerate the LOD construction of point clouds by two orders of magnitude compared to CPU-based approaches. There are, however, some limitations, potential issues or improvements, and implementation details that we would like to note and discuss.

- **Out-of-core** processing was not evaluated – the proposed approach is purely in-core, i.e., all data (input, temporary buffers, output), resides in GPU memory. However, we believe that this approach easily integrates in existing out-of-core and bottom-up LOD construction schemes [SOW20; BK20] which first partition point clouds into large chunks (e.g., 10 million points). Each chunk could then be processed on the GPU by the method proposed in this paper.
- **Weighted-average color filtering** computes weighted averages within a neighborhood, but only for points and voxels within a node – samples in close range but stored in another node are not considered. In practice, we found no perceivable issues with any of our test data sets, but sampling artifacts as shown in Figure 6c may occur, but rarely since they may occur between nodes instead of the 128^3 voxels inside the node.
- **Optimal color filtering** was not evaluated – we only showed that simple arithmetic averages within a cell or linear weighted samples within a range already significantly improve the quality. Better color filtering approaches are subject to future work and could include using more suitable weighting functions such as Gaussian or Lanczos.
- **View-dependent color filtering** may be an important topic for future work. Currently, surface samples that are visible from different directions collapse into a single voxel in lower levels of detail. For example, two sides of a wall may become a single voxel that holds only one color value for all view directions.

Future work and implementations might revisit view-dependent voxels or impostors [GM05; WWS01] or incorporate insights from (neural) radiance fields, which evaluate the most suitable color value of a surface for each view direction [MST*20; KPLD21], with spherical-harmonics-based approaches appearing particularly promising [STC*22; YLT*21; KKLD23].

- **Voxels** are used in lower LODs mainly because they compress better than full-precision point coordinates, which is beneficial for streaming over web browsers. Their disadvantage is that measurement operations (distance, height, area, ...) will have reduced precision in lower levels of detail. However, with some modifications, our method supports full-precision point samples in lower LODs since in our implementation, voxels and points use the same struct with floating point coordinates. For the first-come and random sampling methods, we can simply deactivate the quantization to populate lower LODs with accurate point coordinates. For the average and weighted average methods, we could also create a list of accepted samples in addition to occupied voxels, and then use the average color values from the occupied voxels, and the coordinate value from the accepted samples to populate the lower LOD nodes.
- **Memory usage** of the sampling grids of the average and weighted methods could be halved by utilizing fewer bits per channel. Instead of using four 32 bit integers, rgb and counters could be stored in a single 64 bit integer with a 18-18-18-10 bit pattern, which allows computing an average of up to 1024 samples per voxel. This would also reduce computational cost from four 32 bit or two 64 bit atomic-add down to a single 64 bit atomic-add per sample.
- **A hierarchical voxel encoding** scheme compresses voxel coordinates to about 2 bit on average for storage on disk. For each voxel in a node, an 8 bit mask determines whether the respective voxels in the child node exist. Since lower-LOD voxels split in 4 higher-LOD voxels on average, the number of bits for each higher-LOD voxel is about 2. Colors are BC-encoded and require an additional 8 bit per voxel, and all voxels are sorted in z-order to improve the coherence of colors in each BC-encoded block of voxels. This scheme also supports parallel, GPU-based decoding using one thread per parent voxel, where each thread may produce 0 to 8 child voxels.

The source code for this paper is available at `https://github.com/m-schuetz/CudaLOD`.

8. Acknowledgements

This research has been funded by FFG project LargeClouds2BIM.

The authors wish to thank *Iconem* for providing data sets *Palmyra* and *Saint Roman*; *Schloss Schönbrunn Kultur- und Betriebs GmbH, Schloss Niederweiden* and *Riegl Laser Measurement Systems* for providing the data set of Schloss Niederweiden; *Riegl Laser Measurement Systems* for providing the data set of the town of *Retz*; Bunds et al. and Open Topography for providing the data set CA21-Bunds [CA21Bunds]; and the *Stanford University Computer Graphics Laboratory* for the *Stanford Bunny*.

References

[3DEP] *3D Elevation Program (3DEP)*. `https://www.usgs.gov/core-science-systems/ngp/3dep`, Accessed 2020.09.18 2.

[AHN2] *AHN2*. `https://www.pdok.nl/introductie/-/article/actueel-hoogtebestand-nederland-ahn2-`, Accessed 2021.03.27 2.

[BHZK05] BOTSCH, MARIO, HORNUNG, ALEXANDER, ZWICKER, MATTHIAS, and KOBBELT, LEIF. "High-quality surface splatting on today's GPUs". *Proceedings Eurographics/IEEE VGTC Symposium Point-Based Graphics, 2005.* 2005, 17–141 4.

[BK20] BORMANN, PASCAL and KRÄMER, MICHEL. "A System for Fast and Scalable Point Cloud Indexing Using Task Parallelism". *Smart Tools and Apps for Graphics - Eurographics Italian Chapter Conference*. Ed. by BIASOTTI, SILVIA, PINTUS, RUGGERO, and BERRETTI, STEFANO. The Eurographics Association, 2020 2, 3, 10.

[BLD13] BAERT, JEROEN, LAGAE, ARES, and DUTRÉ, PHILIP. "Out-of-Core Construction of Sparse Voxel Octrees". *Proceedings of the 5th High-Performance Graphics Conference*. HPG '13. Anaheim, California: Association for Computing Machinery, 2013, 27–32. ISBN: 9781450321358 3.

[CA21Bunds] BUNDS, M.P., SCOTT, C., WHITNEY, B., and LEE, V.J.C. *High Resolution Topography of the Southern San Andreas Fault from Painted Canyon to Bombay Beach*. Accessed 2023.02.16. 2021. URL: `https://doi.org/10.5069/G94M92RG` 11.

[CG12] CRASSIN, CYRIL and GREEN, SIMON. "Octree-based sparse voxelization using the GPU hardware rasterizer". *OpenGL Insights* (2012), 303–318 3.

[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 4.

[CNLE09] CRASSIN, CYRIL, NEYRET, FABRICE, LEFEBVRE, SYLVAIN, and EISEMANN, ELMAR. "GigaVoxels: Ray-Guided Streaming for Efficient and Detailed Voxel Rendering". *Proceedings of the 2009 Symposium on Interactive 3D Graphics and Games*. I3D '09. Boston, Massachusetts: Association for Computing Machinery, 2009, 15–22. ISBN: 9781605584294 3.

[CPU] *CPU Benchmark - AMD Ryzen 2700X vs AMD Threadripper 3960X*. `https://www.cpubenchmark.net/compare/3238vs3617/AMD-Ryzen-7-2700X-vs-AMD-Ryzen-Threadripper-3960X`, Accessed 2023.02.27 8.

[CRW14] CHAJDAS, MATTHÄUS G., REITINGER, MATTHIAS, and WESTERMANN, RÜDIGER. "Scalable rendering for very large meshes". *Journal of WSCG* 22 (2014), 77–85 2–4, 7.

[DVS03] DACHSBACHER, CARSTEN, VOGELGSANG, CHRISTIAN, and STAMMINGER, MARC. "Sequential Point Trees". *ACM Trans. Graph.* 22.3 (2003), 657–662 2.

[ENTW] *USGS / Entwine*. `https://usgs.entwine.io`, Accessed 2020.09.18 2.

[FC00] FANG, SHIAOFEN and CHEN, HONGSHENG. "Hardware accelerated voxelization". *Computers & Graphics* 24.3 (2000), 433–442. ISSN: 0097-8493 3.

[GM04] GOBBETTI, ENRICO and MARTON, FABIO. "Layered Point Clouds: A Simple and Efficient Multiresolution Structure for Distributing and Rendering Gigantic Point-sampled Models". *Comput. Graph.* 28.6 (2004), 815–826 2–4, 7.

[GM05] GOBBETTI, ENRICO and MARTON, FABIO. "Far Voxels: A Multiresolution Framework for Interactive Rendering of Huge Complex 3D Models on Commodity Graphics Platforms". *ACM SIGGRAPH 2005 Papers*. SIGGRAPH '05. Los Angeles, California: Association for Computing Machinery, 2005, 878–885. ISBN: 9781450378253 3, 11.

[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.

[KB21] KOCON, KEVIN and BORMANN, PASCAL. "Point cloud indexing using Big Data technologies". *2021 IEEE International Conference on Big Data (Big Data)*. 2021, 109–119 2, 3.

[KJWX19] KANG, LAI, JIANG, JIE, WEI, YINGMEI, and XIE, YUXIANG. "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 2, 3.

[KKLD23] KERBL, BERNHARD, KOPANAS, GEORGIOS, LEIMKÜHLER, THOMAS, and DRETTAKIS, GEORGE. "3D Gaussian Splatting for Real-Time Radiance Field Rendering". *ACM Transactions on Graphics* 42.4 (July 2023) 11.

[Knu98] KNUTH, DONALD E. *The Art of Computer Programming*. Vol. 3. Section 5.2. Addison-Wesley, 1998. ISBN: 0-201-89685-0 4.

[KPLD21] KOPANAS, GEORGIOS, PHILIP, JULIEN, LEIMKÜHLER, THOMAS, and DRETTAKIS, GEORGE. "Point-Based Neural Rendering with Per-View Optimization". *Computer Graphics Forum (Proceedings of the Eurographics Symposium on Rendering)* 40.4 (June 2021) 11.

[KSA13] KÄMPE, VIKTOR, SINTORN, ERIK, and ASSARSSON, ULF. "High Resolution Sparse Voxel DAGs". *ACM Trans. Graph.* 32.4 (July 2013). ISSN: 0730-0301 3.

[KSW21] KARIS, BRIAN, STUBBE, RUNE, and WIHLIDAL, GRAHAM. "A Deep Dive into Nanite Virtualized Geometry". *ACM SIGGRAPH 2021 Courses, Advances in Real-Time Rendering in Games, Part 1*. 2021 2.

[LK11] LAINE, SAMULI and KARRAS, TERO. "Efficient Sparse Voxel Octrees". *IEEE Transactions on Visualization and Computer Graphics* 17.8 (2011), 1048–1059 3.

[LRC*03] LUEBKE, DAVID, REDDY, MARTIN, COHEN, JONATHAN D., et al. *Level of Detail for 3D Graphics*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2003. ISBN: 9780080510118 2.

[MST*20] MILDENHALL, BEN, SRINIVASAN, PRATUL P., TANCIK, MATTHEW, et al. "NeRF: Representing Scenes as Neural Radiance Fields for View Synthesis". *ECCV*. 2020 11.

[MVvM*15] MARTINEZ-RUBI, OSCAR, VERHOEVEN, STEFAN, van MEERSBERGEN, M., et al. "Taming the beast: Free and open-source massive point cloud web visualization". Capturing Reality Forum 2015, Salzburg, Austria. 2015 3.

[OLRS23] OGAYAR-ANGUITA, CARLOS J., LÓPEZ-RUIZ, ALFONSO, RUEDA-RUIZ, ANTONIO J., and SEGURA-SÁNCHEZ, RAFAEL J. "Nested spatial data structures for optimal indexing of LiDAR data". *ISPRS Journal of Photogrammetry and Remote Sensing* 195 (2023), 287–297. ISSN: 0924-2716 3.

[PJW12] PREINER, REINHOLD, JESCHKE, STEFAN, and WIMMER, MICHAEL. "Auto Splats: Dynamic Point Cloud Visualization on the GPU". *Proceedings of Eurographics Symposium on Parallel Graphics and Visualization*. Ed. by CHILDS, H. and KUHLEN, T. Eurographics Association 2012. Cagliari, 2012, 139–148 4.

[PK15] PÄTZOLD, MARTIN and KOLB, ANDREAS. "Grid-Free out-of-Core Voxelization to Sparse Voxel Octrees on GPU". *Proceedings of the 7th Conference on High-Performance Graphics*. HPG '15. Los Angeles, California: Association for Computing Machinery, 2015, 95–103. ISBN: 9781450337076 3.

[PZvBG00] PFISTER, HANSPETER, ZWICKER, MATTHIAS, van BAAR, JEROEN, and GROSS, MARKUS. "Surfels: Surface Elements As Rendering Primitives". *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '00. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 2000, 335–342 4.

[RDD15] RICHTER, RICO, DISCHER, SÖREN, and DÖLLNER, JÜRGEN. "Out-of-Core Visualization of Classified 3D Point Clouds". *3D Geoinformation Science: The Selected Papers of the 3D GeoInfo 2014*. Ed. by BREUNIG, MARTIN, AL-DOORI, MULHIM, BUTWILOWSKI, EDGAR, et al. Cham: Springer International Publishing, 2015, 227–242 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 2, 4.

[SBMK20] SCHMITZ, PATRIC, BLUT, TIMOTHY, MATTES, CHRISTIAN, and KOBBELT, LEIF. "High-Fidelity Point-Based Rendering of Large-Scale 3-D Scan Datasets". *IEEE Computer Graphics and Applications* 40.3 (2020), 19–31 3.

[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 3.

[SKW19] SCHÜTZ, MARKUS, KRÖSL, KATHARINA, and WIMMER, MICHAEL. "Real-Time Continuous Level of Detail Rendering of Point Clouds". *2019 IEEE Conference on Virtual Reality and 3D User Interfaces*. Osaka, Japan: IEEE, 2019, 103–110 3.

[SKW21] SCHÜTZ, MARKUS, KERBL, BERNHARD, and WIMMER, MICHAEL. "Rendering Point Clouds with Compute Shaders and Vertex Order Optimization". *Computer Graphics Forum* 40.4 (July 2021), 115–126. ISSN: 1467-8659 7, 8.

[SKW22] SCHÜTZ, MARKUS, KERBL, BERNHARD, and WIMMER, MICHAEL. "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. ISSN: 2577-6193 7, 8, 10.

[SOW20] SCHÜTZ, MARKUS, OHRHALLINGER, STEFAN, and WIMMER, MICHAEL. "Fast Out-of-Core Octree Generation for Massive Point Clouds". *Computer Graphics Forum* 39.7 (Nov. 2020), 1–13. ISSN: 1467-8659 2, 3, 5, 7–10.

[SS10] SCHWARZ, MICHAEL and SEIDEL, HANS-PETER. "Fast Parallel Surface and Solid Voxelization on GPUs". *ACM Trans. Graph.* 29.6 (Dec. 2010). ISSN: 0730-0301 3.

[STC*22] SARA FRIDOVICH-KEIL AND ALEX YU, TANCIK, MATTHEW, CHEN, QINHONG, et al. "Plenoxels: Radiance Fields without Neural Networks". *CVPR*. 2022 11.

[STS*21] SCHUSTER, KERSTEN, TRETTNER, PHILIP, SCHMITZ, PATRIC, et al. "Compression and Rendering of Textured Point Clouds via Sparse Coding". *High-Performance Graphics - Symposium Papers*. Ed. by BINDER, NIKOLAUS and RITSCHEL, TOBIAS. The Eurographics Association, 2021. ISBN: 978-3-03868-156-4 3.

[SW11] SCHEIBLAUER, CLAUS and WIMMER, MICHAEL. "Out-of-Core Selection and Editing of Huge Point Clouds". *Computers & Graphics* 35.2 (2011), 342–351 2, 3, 7.

[vvL*22] VAN OOSTEROM, PETER, VAN OOSTEROM, SIMON, LIU, HAICHENG, et al. "Organizing and visualizing point clouds with continuous levels of detail". *ISPRS Journal of Photogrammetry and Remote Sensing* 194 (2022), 119–131. ISSN: 0924-2716 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 3, 4.

[WS06] WIMMER, MICHAEL and SCHEIBLAUER, CLAUS. "Instant Points: Fast Rendering of Unprocessed Point Clouds". *Symposium on Point-Based Graphics*. Ed. by BOTSCH, MARIO, CHEN, BAOQUAN, PAULY, MARK, and ZWICKER, MATTHIAS. The Eurographics Association, 2006. ISBN: 3-905673-32-0 3.

[WWS01] WIMMER, MICHAEL, WONKA, PETER, and SILLION, FRANÇOIS. "Point-Based Impostors for Real-Time Visualization". *Rendering Techniques 2001 (Proceedings Eurographics Workshop on Rendering)*. Ed. by GORTLER, STEVEN J. and MYSZKOWSKI, KAROL. Eurographics. Springer-Verlag, June 2001, 163–176. ISBN: 3-211-83709-4 11.

[YLT*21] YU, ALEX, LI, RUILONG, TANCIK, MATTHEW, et al. "PlenOctrees for Real-time Rendering of Neural Radiance Fields". *ICCV*. 2021 11.