Software Rasterization of 2 Billion Points in Real Time

MARKUS SCHÜTZ, BERNHARD KERBL, and MICHAEL WIMMER, TU Wien, Austria

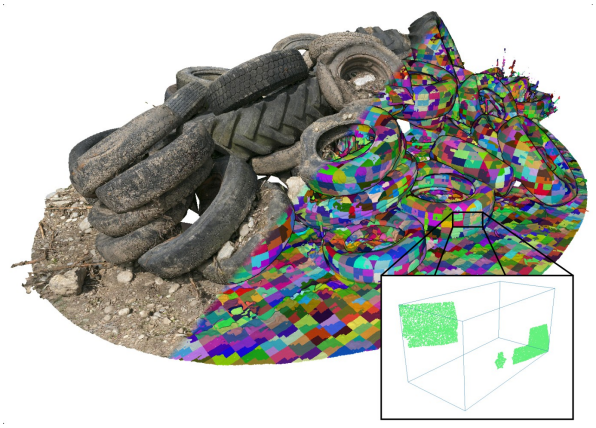


Fig. 1. A point cloud grouped into batches. Individual batches (inset) are rendered by a GPU workgroup using 128 threads, and each thread renders 80 points for a total of $128 \times 80 = 10\,240$ points per batch. Workgroups utilize batch bounding boxes for frustum culling and to determine the suitable coordinate precision: 10-bit fixed-precision integer coordinates (relative to the batch bounding box) provides sufficient precision for the majority of visible batches. Additional bits—enabling up to 30-bit coordinates—are loaded on demand.

The accelerated collection of detailed real-world 3D data in the form of ever-larger point clouds is sparking a demand for novel visualization techniques that are capable of rendering billions of point primitives in real-time. We propose a software rasterization pipeline for point clouds that is capable of rendering up to two billion points in real-time (60 FPS) on commodity hardware. Improvements over the state of the art are achieved by batching points, enabling a number of batch-level optimizations before rasterizing them within the same rendering pass. These optimizations include frustum culling, level-of-detail (LOD) rendering, and choosing the appropriate coordinate precision for a given batch of points directly within a compute workgroup. Adaptive coordinate precision, in conjunction with visibility buffers, reduces the required data for the majority of points to just four bytes, making our approach several times faster than the bandwidth-limited state of the art. Furthermore, support for LOD rendering makes our software rasterization approach suitable for rendering arbitrarily large point clouds, and to meet the elevated performance demands of virtual reality applications.

CCS Concepts: • **Computing methodologies** → **Rasterization**.

Additional Key Words and Phrases: point cloud rendering, rasterization, real-time rendering, virtual reality

Authors' address: Markus Schütz, mschuetz@cg.tuwien.ac.at; Bernhard Kerbl, kerbl@cg.tuwien.ac.at; Michael Wimmer, wimmer@cg.tuwien.ac.at, TU Wien, Austria.

2577-6193/2022/7-ART24
https://doi.org/10.1145/3543863

ACM Reference Format:
Markus Schütz, Bernhard Kerbl, and Michael Wimmer. 2022. Software Rasterization of 2 Billion Points in Real Time. *Proc. ACM Comput. Graph. Interact. Tech.* 5, 3, Article 24 (July 2022), 17 pages. https://doi.org/10.1145/3543863

1 INTRODUCTION

Modern hardware and software solutions are driving the collection of real-world geometry data—so-called digital twins—at a staggering pace: ubiquitous mobile cameras, RGB-D sensors and laser scanners can capture 3D point clouds of individual objects, or even entire buildings, within minutes. However, the resulting raw data sets—which may contain billions of points—quickly prove too demanding for viewing them at an interactive rate, even given the performance of modern-day graphics processing units (GPUs), with their streamlined hardware rasterization pipeline. Especially for virtual reality (VR) settings, the raised demands for visualization (stereoscopic rendering, high resolution and refresh rate) quickly becomes a limiting factor. Thus, costly preprocessing routines must be applied to the captured data sets prior to point cloud visualization, thereby inhibiting the user's abilities to preview, curate and edit captured data sets. In this paper, we introduce significant improvements to the software rasterization of points, allowing applications to draw large amounts of points in real time without the need for preprocessing routines.

With the introduction of hardware with dedicated triangle rasterization units, hand-crafting rasterization routines in software became largely obsolete. Custom-built rasterizers have nevertheless remained an ongoing topic of research and some eventually managed to beat hardware rasterization in specific scenarios [Liu et al. 2010]. But in general, dedicated hardware remains the fastest approach. Unreal Engine's Nanite is the first approach that promises far-reaching improvements for 3D games via hybrid software and hardware rasterization [Karis et al. 2021]. They found that rasterizing the fragments of pixel-sized triangles with atomic min-max operations is faster than pushing them through the hardware pipeline.

Compared to triangle meshes, point cloud models offer additional opportunities for efficient software rasterization, as the hardware rendering pipeline is largely dedicated to the rasterization of triangles rather than points. Point clouds are usually static and lack connectivity, therefore animation data, index buffers or vertex duplication are not required. The lack of a connected surface also makes UV maps and textures irrelevant, which is why colors are typically directly stored on a per-vertex basis. Point clouds acquired by laser scanners do not contain surface normals. However, using point primitives to represent geometric details of all frequencies necessitates ubiquitously high sampling density to obtain high-quality results. Rendering point clouds thus demands processing of vast, simplistic data sets with straightforward shading and a low number of touched fragments per point. By optimizing for these properties, we arrive at a tailor-made, high-performance solution for point cloud rendering. Our approach builds on [Schütz et al. 2021] to further optimize several aspects of software rasterization of points, leading to a significant increase in brute-force rendering performance. With these improvements, we aim to benefit fields that regularly work with data sets comprising billions of points, e.g., surveying, archaeology and architecture: they pave the way for instantly visualizing and interacting with captured data sets, ideally on-site. Our contributions to the state of the art of software rasterization of point clouds are:

- Assigning larger workloads to workgroups to enable efficient batch-level optimizations.
- Adaptive coordinate precision, coupled with visibility-buffering for 3× faster performance.
- Fine-grained frustum culling on batches of about 10 240 points, directly on the GPU.
- Support for state-of-the-art level-of-detail structures for point clouds.

In this paper, we will consider point clouds as 3D models made of colored vertices, where each vertex is projected to exactly one pixel. Although this is a fairly strict limitation, it allows us to devise algorithms that compete with graphics APIs that also only support one-pixel points, such as DirectX (`POINTLIST` primitive) and all backends that use it (WebGL, WebGPU, ANGLE, MS Windows games and applications, ...). We intend to use the evaluated performances of one-pixel points as a baseline for comparisons and leave support for larger point-sprites to future work.

2 RELATED WORK

2.1 Software Rasterization of Triangle Meshes

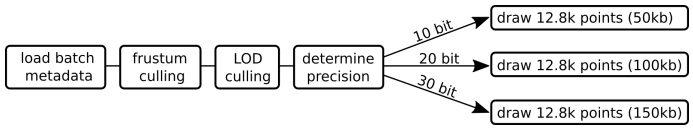
Early CPU-side solutions for triangle rasterization in software were largely made obsolete in the 2000s by GPUs and their high-performance rasterization components. The continuously advancing programmability of GPUs has given software rasterization its second wind: *Freepipe* demonstrated that for scenes containing many, small triangles, GPU software rasterization with one thread per triangle can outperform the hardware pipeline [Liu et al. 2010]. CudaRaster and Piko expanded on this idea, introducing optimizations for hierarchical triangle rasterization, achieving competitive performance with hardware rasterization even for larger triangles [Laine and Karras 2011; Patney et al. 2015]. Complete software implementations of OpenGL-style streaming pipelines, including sort-middle binning and hierarchical rasterization, have been presented for NVIDIA CUDA and OpenCL [Kenzel et al. 2018; Kim and Baek 2021]. A comprehensive analysis of previous software rasterization approaches and the challenges they tackle is found in [Frolov et al. 2020]. Most recently, software rasterization has received increased attention due to the launch of the Unreal Engine 5 and its virtual geometry feature, *Nanite* [Karis et al. 2021]. *Nanite* provides both a hardware and a software pipeline for rasterization geometry and selects the proper route for rendered geometry dynamically. In scenes with mostly pixel-sized triangles, their software pipeline reportedly achieves more than 3× speedup. Its striking success begs the question whether high-performance software rasterization has not been overlooked as a viable method for other 3D representations as well.

2.2 Software Raserization of Point Clouds

Günther et al. proposed a GPU-based approach that renders points up to an order of magnitude faster than native OpenGL point primitives [Günther et al. 2013]. When a point modifies a pixel, their busy-loop approach locks that pixel and updates depth and color buffers. Marrs et al. have used atomic min/max to reproject a depth buffer to different views [Marrs et al. 2018]. Since only depth values are needed, 32-bit atomic operations are sufficient. Schütz et al. render colored point clouds by encoding depth and color values of points into 64 bits, using 64-bit atomic min operations to find points with the lowest projected depth value for each pixel in an interleaved depth and color buffer [Schütz et al. 2021]. Our paper is based on their approach, making it several times faster while also adding support for frustum culling and LOD rendering. Recently, Rückert et al. have presented their approach for a differentiable novel-view synthesis renderer based on the same prior work, using their fast software rendering of one-pixel points to draw multiple resolutions of a point cloud, followed up by neural networks to fill holes and refine the results [Rückert et al. 2022].

2.3 Level-of-Detail for Point Clouds

Rusinkiewicz and Levoy introduced QSplat, a point-based level-of-detail data structure, as a means to interactively render large meshes [Rusinkiewicz and Levoy 2000]. They use a bounding-sphere hierarchy that is traversed until a sphere with a sufficiently small radius is encountered, which is then drawn on screen. Sequential Point Trees [Dachsbacher et al. 2003] are a more GPU-friendly



(a) Processing flowchart for a single point cloud rendering workgroup.

Fig. 2. Each workgroup renders one point cloud batch. If its projected bounding box is small, fewer coordinate bits are loaded per point, reducing memory bandwidth usage and boosting render performance accordingly.

approach that sequentializes a hierachical point-based representation of the model into a non-hierarchical list of points, sorted by their LOD: from a distance, only a small continuous subset representing a lower LOD needs to be rendered, without the need for costly traversal through a dense hierarchical structure. Layered point clouds [Gobbetti and Marton 2004] were one of the most impactful improvements to LOD rendering and are used to this day. The LPC constitutes a binary tree that splits the 3D space in half at each level of the hierachy. The key difference to the bounding-sphere hierarchy of QSPLATs is that each node itself is not a sphere, but a smaller point cloud comprising thousands of randomly selected points. The large amount of geometry in each node reduces the number of nodes required to store the full data set, leveraging the GPU's ecfficiency at rendering hundreds of batched primitives. Later work improved upon several aspects of layered point clouds, e.g., the tree-structure, LOD generation times, and density-based subsampling to properly support data sets with non-uniform density [Bormann and Krämer 2020; Elseberg et al. 2013; Goswami et al. 2010; Kang et al. 2019; Martinez-Rubi et al. 2015; Scheiblauer and Wimmer 2011; Wand et al. 2008]. Section 3.5 describes the support for layered point clouds with our approach. More specialized approaches have been proposed to extract several suitable LODs at once for multiple views using point-based rendering, as done, e.g., by Hollander et al. [2011].

2.4 Coordinate Quantization

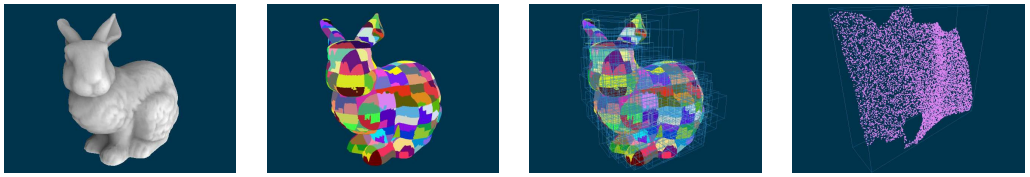
Quantization describes the conversion of a continuous signal to discrete samples. The uniform precision and control over the supported range, precision, and number of bits makes quantization a common method of coordinate compression schemes [Schuster et al. 2021], complemented by delta and entropy encoding [Deering 1995; Isenburg 2013] or hierarchical encoding [Botsch et al. 2002]. In this paper, we use quantization to encode floating point input coordinates as fixed-precision representations, such that their bits can be loaded adaptively (loading fewer bits if lower precision is sufficient, and fetching additional bits to refine the previously loaded low-precision coordinates on-demand).

3 METHOD

The core aspect of our rasterization method is the consideration and assignment of points as batches, with each workgroup rendering a single batch over several iterations. Utilizing larger batches (e.g., 10k points for work groups of 128 threads—80 points per thread) enables several optimizations that would otherwise be too inefficient to amortize corresponding additional checks. In addition, allowing for batches with varying amounts of points enables natural support for several widely used LOD structures, as discussed in Section 3.5. Fig. 2 provides an overview of the main steps for our approach, performed within each workgroup. In the following, we will first describe our basic rendering pipeline, which we will then gradually expand by additional features and optimizations that ultimately enable us to render point clouds several times faster than the state of the art.

(a) Unstructured Point Clouds (b) LOD Batches / Nodes

Fig. 3. (a) For unstructured, Morton-code-ordered data sets, we group 10 240 consecutive points into a batch. (b) For hierarchical (LOD) data, each node refers to a batch of variable size, potentially out of order.



(a) Point Cloud Data Set (b) Batches (c) Bounding Boxes (d) A single batch

Fig. 4. Morton-code ordered point cloud, grouped into batches of 10 240 consecutive points. The resulting locality suffices for effective frustum culling and bounding box-based vertex compression/quantization.

3.1 Data Structure

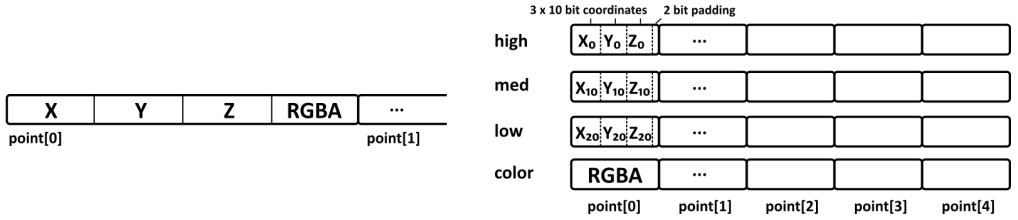
We first build and maintain a list of batches and a list of points. A batch references a number of consecutive points in the point list (see Fig. 3). Each batch stores the offset and number of points it contains, and their bounding box. Each entry in the point list holds four attributes: low, medium, and high precision parts of the coordinates, and color. Attributes are stored in a struct-of-arrays fashion so that we may only load components from memory we actually need during rendering.

For regular, unstructured point cloud data sets, we suggest to simply group about 10k consecutive points into batches and compute their bounding box while loading the points. To benefit from batch-based culling and quantization, points in a batch should exhibit strong locality. In practice, this was already the case for most data sets we encountered, particularly data sets generated via aerial LIDAR or photogrammetry. Otherwise, sorting by Morton code (z-order) provides an easy-to-implement and fast approach to introduce locality into data sets [Isenburg and Lindstrom 2005; Lawder and King 2000; Liu et al. 2020; Orenstein and Merrett 1984; Schütz et al. 2021]. For the remainder of this section, we assume that point clouds exhibit sufficient locality, either by default or through sorting. We discuss the impact of poor locality in Section 4. Fig. 1 and Fig. 4 show the results of grouping 10 240 consecutive points of a Morton-ordered point cloud into batches. Although the locality is not optimal, most batches are sufficiently compact with only a few outliers.

3.2 Basic Rendering Pipeline

The basic pipeline spawns a single compute workgroup for each batch to render its points. Each workgroup comprises 128 threads and each thread renders n points. In practice, we found n between 60 to 200 to perform equally well on an NVIDIA RTX 3090, and will therefore assume $n = 80$ points per thread ($128 \times 80 = 10\,240$ points per batch) throughout the paper. The rasterization process of each individual point is based on prior art [Schütz et al. 2021]: points are projected to pixel coordinates, depth and color values are encoded into a single 64-bit integer, and `atomicMin` is used after a cheap early-depth test to resolve visibility, as shown in Listing 1.

```
1 vec4 pos = worldViewProj * position;
2 int pixelID = toPixelID(pos);
3 int64_t depth = floatBitsToInt(pos.w);
```



(a) A commonly used array-of-structs memory layout for points using 16 bytes per point, with all point attributes stored side-by-side in a single buffer.

(b) Splitting coordinate bits into separate low-, medium- and high-precision buffers. Each buffer stores 10 bits per axis, encoded as 4 bytes per point.

Fig. 5. (a) Point cloud renderers typically load at least 16 bytes per point during rendering. (b) Splitting coordinate bits into three separate buffers allows loading just the required bits, depending on the viewpoint.

```
int64_t newPoint = (depth << 32) | pointIndex;
// Fetch previously written point
uint64_t oldPoint = framebuffer[pixelID];
// Early-depth test
if(newPoint < oldPoint)
    atomicMin(framebuffer[pixelID], newPoint);
```

Listing 1. Software point rasterization via compute shaders, including an early-depth test.

A fundamental improvement over [Schütz et al. 2021] is achieved with our approach due to issuing fewer, larger workloads to exploit extended thread persistence and reducing the scheduling overhead of the GPU. In addition, batching for unstructured point clouds enables efficient workgroup-wise frustum culling at a granularity of 10 240 points: At the start of each invocation, we first load the bounding box of the workgroup's batch and immediately abort if it lies outside the view frustum.

3.3 Adaptive Vertex Precision

The primary bottlenecks in [Schütz et al. 2021] is memory bandwidth usage. The authors reported a peak performance of 50 billion points per second, using 16 bytes per point, exploiting ≈85% of GPU memory bandwidth in their experiments. Hence, although per-point data is already compact, any significant improvement of rendering performance implies some form of attribute compression.

We propose an adaptive coordinate precision scheme that permits us to load only as many bits as necessary for a given viewpoint. We achieve this by splitting coordinates into three separate buffers, separating low, medium, and high precision bits of XYZ coordinates in an interleaved layout, as shown in Fig. 5. The low-precision part always needs to be loaded, while medium- and high-precision parts can be loaded on-demand to recover the remaining bits. The different precision levels are established via coordinate quantization, i.e., by converting coordinates to fixed-precision integers and splitting the so-quantized bits. To achieve a high coordinate precision with just a few bits, we quantize point coordinates to 30 bit fixed-precision integers relative to each batch's bounding box, indicating the point's position within the box. The X-coordinate, for example, is computed as

$$X_{30} = \min(\lfloor 2^{30} * \frac{x - boxMin.x}{boxSize.x} \rfloor, 2^{30} - 1) \tag{1}$$

These 30 bits are then split into three 10-bit components representing low-, medium- and high-precision parts. The 10 most significant bits (indices 20 to 29) encode the coordinate of a point

within a batch at a precision of $\frac{1}{2^{10}} = \frac{1}{1024}$ of its size. Considering that the majority of rendered batches in any given viewpoint are typically smaller than a couple of hundreds of pixels, and that 10 bits grant us 1024 different coordinate values, we find that 10 bits per axis are sufficient to render most points with sub-pixel coordinate precision. Due to buffer alignment recommendations, simplicity, and because the common minimum alignment on GPUs is 32 bits, we then pack the 10-bit x, y and z components into a single 32 bit integer with the remaining 2 bits used as padding, as shown in Listing 2. The result is a 32-bit integer buffer where each 4-byte element contains the 10 lowest precision bits of the three coordinate axes of a single point.

```
1 uint32_t X_low = (X_30 >> 20) & 1023;
2 uint32_t Y_low = (Y_30 >> 20) & 1023;
3 uint32_t Z_low = (Z_30 >> 20) & 1023;
4 uint32_t encoded = X_low | (Y_low << 10) | (Z_low << 20)
```

Listing 2. Encoding the most significant 10 bits of each axis into a single 32 bit integer.

Likewise, we generate two more buffers for the medium (bits 10 to 19) and high precision bits (bits 0 to 9). During rendering, each workgroup can now selectively load a single 4-byte integer containing just the low precision bits, two (medium precision) or 3 such integers (full precision, which allows for one 10^9 different coordinate values). We note that the usefulness of 30-bit precision is limited in practice, as we still convert integers to floating-point coordinates during rendering, resulting in a lossy conversion. This issue is not specific to our approach, however, as point clouds are typically already stored in integer coordinates on disk and the conversion to floats for rendering has always been an issue for point clouds with a large extent. Handling this issue is out of scope for this paper, but note that storing coordinates as 30-bit fixed-precision integers would allow us to convert them to highly accurate double-precision coordinates (e.g., for surveying applications), while traditional floating point storage causes an irrecoverable loss of precision.

The required point coordinate precision for rendering is determined from the projected size of the containing batch. If a batch projects to fewer than 500 pixels, we use 10-bit coordinates. The reason for choosing 500 pixels as the threshold, even though 10 bits can represent 1024 different coordinates values, is that quantization changes the distance between any two points by up to the size of a quantization grid cell, i.e., points that were one grid cell's size apart might now be twice as far apart. Using half the size of the quantization grid as the pixel threshold ensures that we do not introduce additional holes between rasterized points. Compared to [Schütz et al. 2021], this approach reduces the required memory bandwidth for most rendered points from 16 bytes down to just 8. However, we can further cut this in half by using a visibility-buffer (item-buffer) approach [Burns and Hunt 2013; López et al. 2021; Weghorst et al. 1984], i.e., we render point indices rather than colors into the framebuffer during the geometry pass, and transform the point indices to vertex colors in a post-processing step. Doing so reduces the amount of memory accesses to color values from the number of processed points to the number of actually visible points.

3.4 Optimizing Access Patterns

To identify performance bottlenecks of our approach, we performed a direct port of its GLSL shader code to NVIDIA CUDA. This enabled the use of the Nsight Compute tool to pinpoint suboptimal behavior in our routines. Dominant stall reasons revealed that performance is still governed by memory operations, with approximately 70% of the total kernel run time spent on them.

Since we choose the batch size to be a multiple of the work group size, threads persist and process multiple points in a loop before returning. Naïvely, the corresponding point data is fetched and rasterized to the framebuffer in each iteration. Due to the structure-of-arrays approach, all loading accesses target a linear range in memory, thus input data can be transferred by workgroups

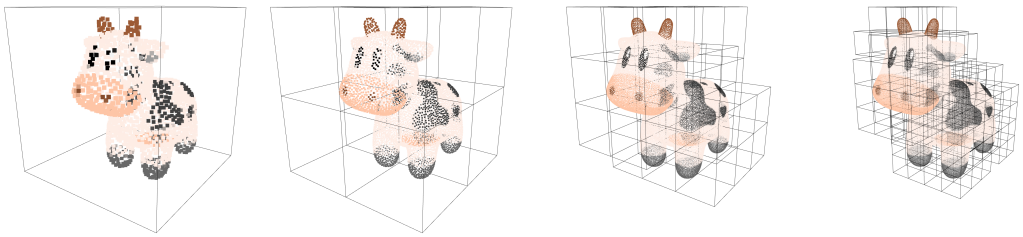
with perfect coalescence. However, loading each processed point individually at the start of its associated iteration still incurs a direct dependency of the steps in Listing 1 on global device memory latency. Due to the early-depth test in its body and the parameterizable iteration count, the compiler cannot trivially unroll the loop without creating secondary issues associated with complex program flow (e.g., frequent instruction cache misses). Hence, we perform manual pre-fetching of point data. Depending on the required precision, we can execute one to three vectorized 128-bit loads in each thread to yield enough data for four successive iterations. This policy significantly alleviates reported stalls due to memory latency and simultaneously reduces the number of memory instructions, without compromising on coalesced accesses. Overall, we found pre-fetching to cause $\approx$ 30% performance gain. A welcome side effect, though less impactful, is the reduction of updates to the framebuffer, since the early-depth test has a higher chance of failing: threads are now more likely to query (and find) information in the L1 cache for pixels they wrote to in a previous iteration.

With pre-fetching in place, the main remaining bottleneck is the code block for early-depth testing and framebuffer updates. Although these accesses are somewhat localized if points are spatially ordered, a residual irregularity remains in their access pattern. The coarse-grained memory transfer policy of the GPU consequently causes more than 2× the amount of actually accessed information to be transferred. However, this update pattern is inherent to the overall routine design, and our attempts to further improve on it without extensive revision caused diminishing returns.

3.5 Adding Support for Level-of-Detail Rendering

In this section, we will describe how support for some of the most popular LOD structures for point clouds—Layered Point Clouds (LPC) [Gobbetti and Marton 2004] and its variations—can be added to our point rasterization approach. LPCs are a hierarchical, spatial acceleration structure, where points with varying density are stored in the nodes of the tree, as shown in Fig. 6. Lower levels contain coarse, low-density subsets of the whole point cloud. With each level, the size of the nodes decreases, leading to an increase of the density of points as the number of points in each node stays roughly the same. The structures are often additive, i.e., higher LODs complement lower LODs instead of replacing them, but replacing schemes are also possible.

We implement and evaluate our support for LPC structures based on the Potree format, which constitutes a variation of LPC based on octrees and populates nodes with subsamples of the point cloud with a specific, level-dependant minimum distance. Each octree node comprises about 1k to 50k points, and Potree itself typically renders about 100 to 1000 nodes for any given viewpoint. We can easily adapt our data structure to support the Potree format by allowing varying numbers of points per batch, as shown in Fig. 3. The workgroup size of 128 threads remains the same, but each thread will now render $\lceil \frac{batch.numPoints}{128} \rceil$ points instead of exactly 80. Fig. 6 depicts octree nodes at several levels of detail. The Potree format is structured such that octree nodes whose projected bounding boxes are small can be entirely ignored, because the points stored in their parents will already provide sufficient detail. This means that the precision selection procedure described in Section 3.3 can now be used to entirely cull the node. We suggest to cull the nodes with the following conditions in mind: Each Potree node is cubic and stores a subsample of points with a resolution that approximately matches a grid with 128^3 cells, while our rasterizer maps each point to exactly 1 pixel. To avoid holes between rendered points, we therefore suggest to cull nodes that are smaller than 100 pixels on screen. However, it is also viable to cull larger octree nodes on lower-end GPUs to improve performance, and cover up the resulting holes in a post-processing pass, e.g., via depth-aware dilation or sophisticated hole-filling approaches [Grossman and Dally 1998; Kivi et al. 2022; Pintus et al. 2011; Rosenthal and Linsen 2008; Rückert et al. 2022].



(a) LOD 0 (b) LOD 1 (c) LOD 2 (d) LOD 3

Fig. 6. The Potree (layered point clouds) LOD structure is made up of cubic octree nodes, from coarse-to-fine subsamples of the original point cloud. We treat each octree node as one workgroup batch with a variable number of points; only octree nodes that encompass more than 100 pixels on screen are rendered.

3.6 Virtual Reality Rendering

Virtual reality usually implies stereo- or multi-view rendering, which greatly increases the performance requirements—even more so for point clouds, as they typically suffer from aliasing artifacts that detract from the VR experience. In addition to specific configurations of our pipeline (using LOD rendering for performance and large framebuffers for supersampling), we can exploit specific properties of VR rendering in our approach. First, scenes must be drawn at least twice—once for each eye. Instead of duplicating the entire rasterization pipeline by calling the compute shaders twice, we can modify our shaders to simply draw each point into both framebuffers. While this does not double the performance, it provides a significant improvement, as discussed in Section 4. Second, in a VR setup, details in the periphery do not provide the same fidelity as details in the center of the view. This is partially because most details are perceived in the gaze direction, i.e., mostly straight ahead in VR, but also because the rendered image will be distorted before it is shown inside the HMD to counter the lens distortion [Vlachos 2015]. We therefore suggest to render peripheral regions of the framebuffer with reduced geometric complexity by raising the threshold for LOD culling, e.g., culling nodes smaller than 300 pixels in the periphery, nodes smaller than 100 pixels in the center of the view, and 200 pixels in between. The resulting holes between points are then filled in post-processing, in our case via a simple depth-aware dilation shader that increases point sizes to up to 7x7 in the periphery and 3x3 in the center.

4 EVALUATION

The proposed method has been implemented in C++ and OpenGL, using compute shaders for parallel GPU execution. It was evaluated with the test data sets shown in Fig. 7. The smaller data sets, Eclepens and Morro Bay, exhibit low depth complexity, i.e., the number of hidden surfaces is typically small. Niederweiden and Banyunibo pose a bigger challenge due to the higher point density and an interior room that is either occluded when viewed from the outside, or it occludes everything else when viewed from the inside. The performance was computed through OpenGL timestamp queries at the start and end of each frame. All durations represent the average time of all frames over the course of one second. The evaluation was conducted on the following test systems:

- NVIDIA RTX 3090 24GB, AMD Ryzen 7 2700X (8 cores), 32GB RAM, running Windows 10.
- NVIDIA GTX 1060 3GB, AMD Ryzen 5 1600X (6 cores), 32GB RAM, running Windows 10.

Unless specified otherwise, all reported timings are from the RTX 3090 system. The GTX 1060 (3GB) was only capable of keeping the smallest test data set, Eclepens (68M points), in memory.

(a) Eclepens (b) Morro Bay (c) Banyunibo outs. (d) Banyunibo inside (e) Niederweiden

Fig. 7. Our test data sets used for evaluation. (a) A quarry captured with photogrammetry. (b) A coastal city, captured with aerial LIDAR. (c+d) A candi in indonesia, captured with both photogrammetry and a terrestrial laser scanner. (e) A manor captured with terrestrial laser scanning.

4.1 Rasterization Performance

Table 1 shows the results of rendering various data sets with our proposed basic approach (frustum culling and adaptive precision; Sections 3.2, 3.3), prefetch (basic + prefetch 4 points at a time; Section 3.4) and LOD (Section 3.5), compared with prior art [Schütz et al. 2021] and `GL_POINTS`. For unstructured point-cloud data, our approach with prefetching performs the fastest in all cases—up to 3× faster than previous work in overview scenarios, and 5× faster for close-up viewpoints that benefits from frustum culling.

Table 2 shows how many batches and points were rendered in a frame from the given viewpoint. Processed batches include all batches/nodes of the data set, since we spawn one workgroup per node. Rendered batches are those that pass the frustum and LOD culling tests. Frustum culling can reduce the amount of rendered batches to $< 50\%$ for unstructured point clouds (Banyunibo, inside), or down to several thousand out of hundreds of thousands in conjunction with LOD structures. The throughput is computed by taking the number of processed points in Table 2 and dividing it by the rendering time in Table 1. On the RTX 3090 system and with prefetching for unstructured data sets, we get throughputs of 69, 126.8, 144.7, 97.5 and 142.3 billion points per second for the five scenarios. Considering rendered points per 16 ($\approx \frac{1000}{60}$) milliseconds, we can alternatively express the throughput as renderable points at 60 FPS, which yields 1.1, 2.0, 2.3, 1.6 and 2.3 billion points per 16 milliseconds. A detailed breakdown of benefits due to prefetching is provided for Banyunibo (outside) in Fig. 8, based on the CUDA implementation of our approach.

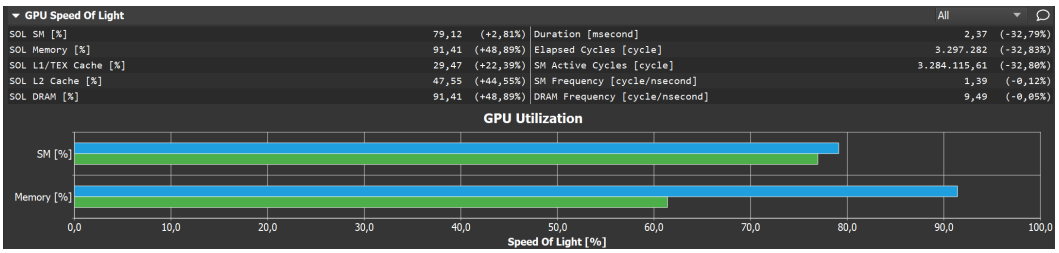


Fig. 8. Nsight Compute metrics for CUDA rasterization with prefetching disabled (green) and enabled (blue).

4.2 The Impact of Vertex Ordering

The disadvantage of our naïve approach for generating batches by grouping 10 240 consecutive points is that the resulting performances depend on the vertex order of the data set. Fig. 9 illustrates the vertex ordering of a terrestrial laser scanner that scans on two rotational axis, first

Table 1. Comparing rendering times (in milliseconds) with our new approach, GL_POINTS, and prior art by Schütz et al. [2021]. (pref.) uses the prefetch optimization in Section 3.4. (hqs) High-Quality Shading, as described in [Schütz et al. 2021]. All experiments conducted on an RTX 3090, except Eclepens* which was benchmarked on a GTX 1060. Framebuffer size: 2560×1140 (2.9MP). All point clouds sorted by Morton code.

				ours			ours (LOD)		prior art	
Data Set	points	size	GL	basic	pref.	hqs	basic	hqs	dedup	hqs
Eclepens	69M	1.1GB	34.9	1.1	1.0	2.8	0.7	1.4	1.9	7.6
Eclepens*	69M	1.1GB	72.2	6.5	5.1	16.7	2.1	5.3	9.5	26.2
Morro Bay	279M	4.4GB	231.7	2.7	2.2	9.6	0.8	1.9	6.0	33.5
Banyunibo (out)	529M	8.5GB	198.3	4.2	3.3	9.0	1.3	3.3	10.7	25.4
Banyunibo (in)	529M	8.5GB	67.9	2.6	2.2	6.1	2.1	4.6	11.1	24.4
Niederweiden	1000M	16GB	873.9	8.2	6.4	20.9	1.9	4.5	19.8	69.1

Table 2. Statistics for processed (proc.) and rasterized (rast.) batches and points during a frame. Processed batches: All fixed-size batches (unstructured) or variable-sized octree nodes (LOD). Rasterized batches: Batches that passed frustum and LOD culling. Processed points: Points that are loaded by the shader. Rasterized points: Points that pass point-wise frustum culling and early-depth, i.e., points attempting framebuffer updates.

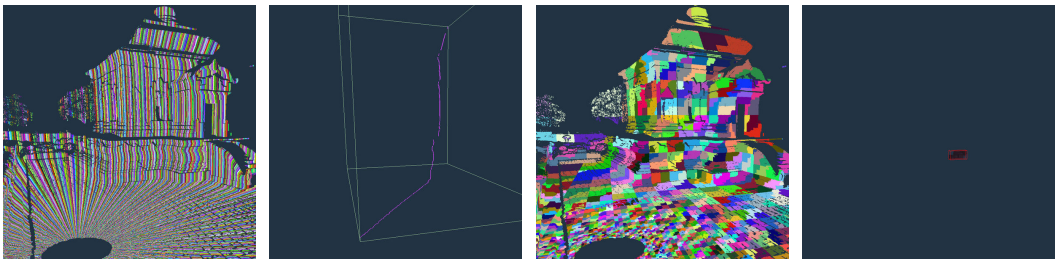
	unstructured				LOD			
	batches		points		batches		points	
Data Set	proc.	rast.	proc.	rast.	proc.	rast.	proc.	rast.
Eclepens	6.7k	6.7k	68.7M	13.0M	23.5k	422	3.9M	1.4M
Morro Bay	27.2k	27.2k	278.5M	12.2M	93.5k	577	5.6M	1.9M
Banyunibo (outside)	51.7k	46.6k	477.6M	13.7M	195.7k	3.3k	26.9M	4.4M
Banyunibo (inside)	51.7k	20.9k	214.5M	19.1M	195.7k	8.2k	67.1M	12.5M
Niederweiden	97.7k	88.9k	910.7M	51.6M	346.6k	2.1k	27.3M	6.4M

top-bottom (pitch) and then right-left (yaw). Consequently, 10 240 consecutive points usually form a 3-dimensional curve along the surface of the scanned object. The resulting batch has a large extent with mostly empty space. The next batch is formed by the next "scan-line", with an almost identical bounding box. Fig. 9 also demonstrates the vertex order and the resulting batches after the points are sorted by Morton order. The resulting batches are more compact with less empty space, and therefore more suitable to frustum culling and rendering with lower coordinate precision.

We evaluated the performance differences between scan-order and Morton-order on a subset of the Banyunibo data set comprising only of the laser scans. From an outside viewpoint, scan-order requires 5.5 ms to render a frame and Morton-order requires 3.9 ms, which is mostly attributed to the lower coordinate precision requirements of the compact batches. From an inside viewpoint, the scan-order requires 7.8 ms to render a frame and Morton-order requires 2.1 ms. In this case, the increased performance differences can further be attributed to frustum culling, which culls about 68% of all batches of the Morton-ordered data set, but only 35% of the scan-ordered data set.

4.3 Virtual Reality Performance

Virtual reality rendering requires significantly higher performance, since we need to render data sets with higher frame rates, twice per frame, and in high quality. We evaluated the VR performance of our approach on a Valve Index HMD (1440×1600 pixels per eye) with an RTX 3090 GPU. The targeted framerate is 90fps, thus each frame needs to be finished in 11.1 ms, or closer to about 9 ms to account for additional computations and overhead in the VR pipeline. The targeted resolution



(a) Batches, Scan Order (b) Batch, Scan Order (c) Batches, Morton Order (d) Batch, Morton Order

Fig. 9. Terrestrial laser scanners typically scan along two rotation axes. (a+b) Poor locality in scan order results in huge but mostly empty batches. (c+d) Morton order efficiently establishes locality and results in compact batches. Both batches in (b) and (d) contain the same number of points.

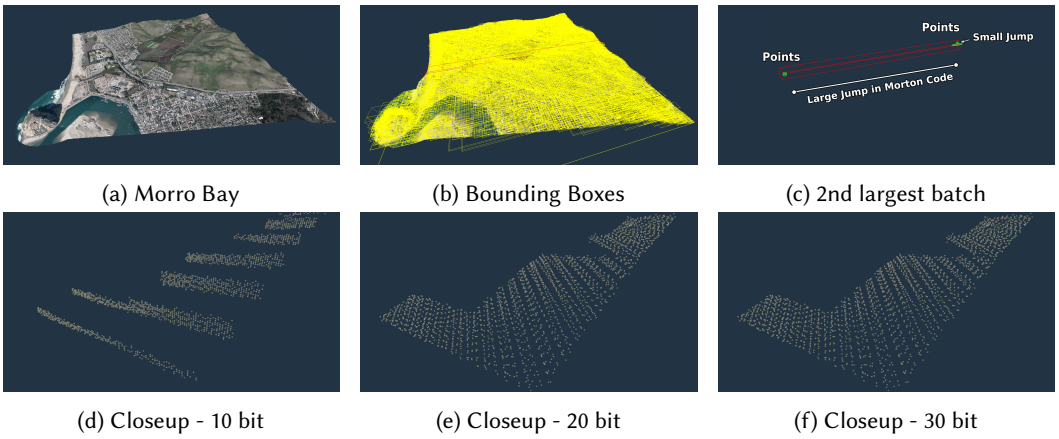
is 2478×2740 (6.8MP) per eye, mandated by the 150% resolution setting in SteamVR. The high resolution alleviates some aliasing issues, but high-quality shading [Schütz et al. 2021] is also required and used to avoid severe flickering artifacts.

We evaluated the VR performance with an outside-in view of the Candi Banyunibo data set, comprising 529 million points in 195k octree nodes. 21.4 million points in 2k nodes passed the frustum and LOD culling routines. After early-depth testing, 8 million points were drawn with `atomicMin`. The total frame time was 8.3 ms, which provides sufficient reserves for additional computations and overhead of the VR rendering pipeline. Rendering the depth buffer for the hqs shader took 1.7 ms for both eyes, or 1.3 ms when rendering just a single eye, which demonstrates the benefits of drawing each point to both render targets within a single compute shader invocation. Similarly, drawing the color buffer of the hqs approach took 2.5 ms for both eyes and 1.5 ms for a single eye. The resolve pass, which enlarges the more sparsely rendered points in the periphery and stores the colors into an OpenGL texture, takes about 2.7 ms per frame for VR rendering, which is mainly attributed to the large and dynamic dilation kernels of 3×3 (center) to 7×7 (periphery) pixels. Finally, clearing the relatively large depth and color buffers (2468×2740 per eye) takes about 0.6 ms per frame.

4.4 Adaptive Precision

4.4.1 Quality. Reducing the point coordinate precision improves performance, but it may also reduce the quality if the chosen precision is too low. Fig. 10 demonstrates this issue on the second-largest batch of the Morro Bay data set, which has an extent of [3668.6, 166.4, 141.7] m. Dividing the extents by 2^{10} yields the 10-bit-precision along each axis: [3.58, 0.16, 0.14] m. Fig. 10d shows a closeup of the insufficient precision along the x-axis that manifests as stripe patterns due to x values being truncated. Similarly, y- and z-axis also have slightly insufficient precision, resulting in a notable grid sampling pattern. Increasing the precision to 20 bits results in a coordinate precision of [3.50, 0.16, 0.14] mm for that batch, which is already more precise than the original data set. 30 bit precision is therefore unnecessary, even for the largest batches. However, the full version of the Morro Bay data set—(CA13) [Pacific Gas & Electric Company 2013]—has a much larger extent with 18 billion points in total, and may therefore result in large batches requiring 30 bits.

4.4.2 Performance. The isolated impact of adaptive precision was evaluated with the Banyunibo data set with frustum culling disabled and a viewpoint in which all batches are outside the view frustum. Due to this, all points are loaded and processed, but none are written to the framebuffer. We also limited the measurements to the geometry pass, instead of the whole frame. We then took



(a) Morro Bay (b) Bounding Boxes (c) 2nd largest batch

(d) Closeup - 10 bit (e) Closeup - 20 bit (f) Closeup - 30 bit

Fig. 10. Adaptive Precision. Top: Overview of the Morro Bay data set. Jumps in the Morton code can lead to excessively large batches. Bottom: Closeup of one cluster inside the second-largest batch. 10 bits are insufficient to display its points in a closeup viewpoint due to truncation, while 20 bits render it indistinguishable from the 30-bit encoding. Stripe patterns in (e, f) are due to line-wise scan pattern of the aerial LIDAR scanner.

three measures in which all batches were rendered with either 10, 20 or 30 bits for precision: [2.93, 5.23, 7.61] ms. These results indicate that 20- and 30-bit precision require about 1.75× and 2.6× as much processing time as 10-bit precision, respectively. Table 3 shows the nodes rendered with a precision of 10, 20 or 30 bits per coordinate axis. In Morton-ordered data sets, the vast majority of batches are compact enough to be rendered with 10 bit precision, even in closeup viewpoints.

5 DISCUSSION AND CONCLUSION

In this section, we will discuss several issues and limitations, as well as potential improvements that we have not evaluated, yet. For one, we believe that our approach yields a significant improvement for point cloud rendering, but it is not useful for games in its current state. Point clouds require a large amount of colored vertices to represent geometry without holes and sufficient color detail, while meshes can use a single textured triangle in place of thousands of points. However, massive amounts of point cloud data sets exist as a result of scanning the real world, and this paper provides tools to render them faster without the need to process the data into LOD structures or meshes. We hope that the presented approach might provide useful insights in future developments of hybrid hardware/software rasterization approaches for triangle meshes.

The visual quality in VR applications currently suffers from lack of proper color filtering. Although high-quality shading is applied and improves the results via blending, the issue is that the LOD structure removes most of the overlapping points, thus the blended result is not representative of the full point distribution. The results can be improved by applying color filtering to points in lower levels of detail during the construction of the LOD structure [Rusinkiewicz and Levoy 2000; Schütz et al. 2019; Wand et al. 2008]. Furthermore, implementing continuous LOD could improve the visual quality through a subtle transition in point density between LODs, eliminating popping artifacts as details are added and removed while navigating through the scene [Liu et al. 2020; Schütz et al. 2019].

The adaptive coordinate precision approach leads to significant performance improvements through a reduction in memory bandwidth usage, but it does not reduce storage requirements—coordinates still use up 12 bytes of GPU memory. At this point in time, we deliberately did not

Table 3. Precision of rendered nodes in several data sets using overview and closeup viewpoints. The more nodes are rendered with 10 bit precision, the better. Morro Bay performs well even without sorting. The original Banyunibo performs poorly due to large batches, which can be rectified by establishing Morton order.

			number of rendered nodes					
			overview			closeup		
Data Set	ordering	avg. batch size (m)	10 bit	20 bit	30 bit	10 bit	20 bit	30 bit
Eclepens	original	190.5	5 842	869	0	1 197	2 217	275
	morton	25.0	6 708	3	0	2 494	104	6
Morro Bay	original	164.2	27 198	4	0	14 340	943	12
	morton	47.6	27 197	5	0	14 407	102	2
Banyunibo	original	9.3	31 410	17 116	1 008	2 401	33 372	1 984
	morton	0.4	46 572	68	2	20 746	196	6

employ sophisticated compression approaches such as delta and entropy encoding [Deering 1995; Isenburg 2013] or hierarchical encoding [Botsch et al. 2002] due to their additional computational overhead and the inability to decode such coordinates individually in a straightforward manner: Delta and entropy encoding require to decode the points sequentially, which could work on a per-thread basis as each thread renders about 80 points sequentially. Generally, we expect that compression could work on a per-batch (10 240 points) basis, a per-thread (80 points) basis and/or a per-subgroup (32 or 64 cooperating threads) basis. Alternatively, fast (lossy) hardware encoding schemes may be used to encode detailed information more compactly [Evans 2015]. Future work could also build on learned approaches such as Schuster et al. who propose learned dictionaries to achieve high compression rates for textured splats that are quickly decoded directly in the fragment shader [Schuster et al. 2021].

There are several details in our proposed method and implementation that could still be improved in future iterations. For example, instead of assigning 10 bits to each coordinate axis and leaving 2 bits for padding, we could distribute the full available 32 bits according to the relative lengths of each axis of the bounding box, i.e., assigning more bits to the longest axis. This avoids huge variations in precision among the x, y and z coordinates, as shown in Fig. 10d, and thereby allows us to render larger batches with lower precision. Another potential improvement would be to split batches whose size is governed by jumps in the Morton code, as seen in Fig. 10c. For each batch, we could identify such jumps and, if they exceed certain thresholds, we could split them into two compact batches. The point buffer would remain unmodified: the existing entry in the batch buffer could be modified with the point index and number of points of one of the two resulting compact batches, and the other batch that references the remaining points can be appended as a new entry at the end of the batch buffer. A third potential improvement would be encoding values not just within the bounding box of a batch, but the bounding box for points in a warp or loop iteration, i.e., within the bounds of 32 or 128 consecutive Morton-ordered points.

We have shown that software rasterization, using standard OpenGL compute shaders, is capable of rendering up to 144.7 billion points per second (Section 4.1), which translates to 2.3 billion points at 60 frames per second (16 ms per frame). The data structure is simple and can be generated on-the-fly during loading for unstructured point clouds, although LOD structures may also be generated in a preprocessing step for further performance improvements. Peak performances were observed in Morton ordered data sets, but many other orderings (e.g., according to tiling and timestamps for aerial LIDAR scans) also generate substantial performance improvements, enabling us to exploit

the spatial locality between consecutive points in memory. Data sets without sufficient locality (e.g., terrestrial laser scans) can simply be sorted in Morton order in a low-overhead preprocessing step.

The source code to this paper is available at https://github.com/m-schuetz/compute_rasterizer.

ACKNOWLEDGMENTS

The authors thank *Schloss Schönbrunn Kultur- und Betriebs GmbH, Schloss Niederweiden* and *Riegl Laser Measurement Systems* for providing the data set of Schloss Niederweiden; the *TU Wien, Institute of History of Art, Building Archaeology and Restoration* for the Candi Banyunibo data set [Herbig et al. 2019]; *Open Topography* and *PG&E* for the Morro Bay (CA13) data set [Pacific Gas & Electric Company 2013]; *Pix4D* for the Eclepens quarry; Sketchfab user *nedo* for the *old tyres* (CC BY 4.0); Keenan Crane for the Spot model; and the *Stanford University Computer Graphics Laboratory* for the *Stanford Bunny* data set.

This research was funded by Österreichische Forschungsförderungsgesellschaft (FFG) project *LargeClouds2BIM (3851914)* and the Research Cluster "Smart Communities and Technologies (Smart CT)" at TU Wien.

REFERENCES

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. https://doi.org/10.2312/stag.20201250

Mario Botsch, Andreas Wiratanaya, and Leif Kobbelt. 2002. Efficient High Quality Rendering of Point Sampled Geometry. In *Proceedings of the 13th Eurographics Workshop on Rendering* (Pisa, Italy) *(EGRW '02)*. Eurographics Association, Goslar, DEU, 53–64.

Christopher A. Burns and Warren A. Hunt. 2013. The Visibility Buffer: A Cache-Friendly Approach to Deferred Shading. *Journal of Computer Graphics Techniques (JCGT)* 2, 2 (2013), 55–69. http://jcgt.org/published/0002/02/04/

Carsten Dachsbacher, Christian Vogelgsang, and Marc Stamminger. 2003. Sequential Point Trees. *ACM Trans. Graph.* 22, 3 (2003), 657–662. https://doi.org/10.1145/882262.882321

Michael Deering. 1995. Geometry compression. In *Proceedings of the 22nd annual conference on Computer graphics and interactive techniques*. 13–20.

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

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].

V. A. Frolov, V. A. Galaktionov, and B. H. Barladyan. 2020. Comparative study of high performance software rasterization techniques. *Mathematica Montisnigri* 47 (2020), 152–175. https://doi.org/10.20948/mathmontis-2020-47-13

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.

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.

Jeffrey P Grossman and William J Dally. 1998. Point sample rendering. In *Eurographics Workshop on Rendering Techniques*. Springer, 181–192.

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.

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://doi.org/10.5194/isprs-archives-XLII-2-W15-555-2019

Matthias Hollander, Tobias Ritschel, Elmar Eisemann, and Tamy Boubekeur. 2011. ManyLoDs: Parallel Many-View Level-of-Detail Selection for Real-Time Global Illumination. *Computer Graphics Forum* 30, 4 (2011), 1233–1240. https://doi.org/10.1111/j.1467-8659.2011.01982.x arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1111/j.1467-8659.2011.01982.x

Martin Isenburg. 2013. LASzip: lossless compression of LiDAR data. *Photogrammetric Engineering & Remote Sensing* 79 (2013). https://doi.org/10.14358/PERS.79.2.209

M. Isenburg and P. Lindstrom. 2005. Streaming meshes. In *VIS 05. IEEE Visualization, 2005.* 231–238.

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.

Brian Karis, Rune Stubbe, and Graham Wihlidal. 2021. A Deep Dive into Nanite Virtualized Geometry. In *ACM SIGGRAPH 2021 Courses, Advances in Real-Time Rendering in Games, Part 1*. https://advances.realtimerendering.com/s2021/index.html [Accessed 10-September-2021].

Michael Kenzel, Bernhard Kerbl, Dieter Schmalstieg, and Markus Steinberger. 2018. A High-performance Software Graphics Pipeline Architecture for the GPU. *ACM Trans. Graph.* 37, 4, Article 140 (2018), 15 pages.

Mingyu Kim and Nakhoon Baek. 2021. A 3D graphics rendering pipeline implementation based on the openCL massively parallel processing. *The Journal of Supercomputing* 77, 7 (2021), 7351–7367. https://doi.org/10.1007/s11227-020-03581-8

Petrus E.J. Kivi, Markku J. Mäkitalo, Jakub Zadnik, Julius Ikkala, Vinod Kumar Malamal Vadakital, and Pekka O. Jaaskelainen. 2022. Real-Time Rendering of Point Clouds with Photorealistic Effects: A Survey. *IEEE Access* 10 (26 Jan. 2022), 13151 – 13173. https://doi.org/10.1109/ACCESS.2022.3146768 Publisher Copyright: Author.

Samuli Laine and Tero Karras. 2011. High-Performance Software Rasterization on GPUs *(HPG '11)*. Association for Computing Machinery, New York, NY, USA, 79–88. https://doi.org/10.1145/2018323.2018337

J. K. Lawder and P. J. H. King. 2000. Using Space-Filling Curves for Multi-dimensional Indexing. In *Advances in Databases*, Brian Lings and Keith Jeffery (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 20–35.

Fang Liu, Meng-Cheng Huang, Xue-Hui Liu, and En-Hua Wu. 2010. FreePipe: A Programmable Parallel Rendering Architecture for Efficient Multi-Fragment Effects *(I3D '10)*. Association for Computing Machinery, New York, NY, USA, 75–82. https://doi.org/10.1145/1730804.1730817

Haicheng Liu, Peter van Oosterom, Martijn Meijers, Xuefeng Guan, Edward Verbree, and Mike Horhammer. 2020. HistSFC: Optimization for nD massive spatial points querying. *International Journal of Database Management Systems (IJDMS)* 12, 3 (2020), 7–28. https://doi.org/10.5121/ijdms.2020.12302

Alfonso López, Juan Manuel Jurado, Emilio José Padrón, Carlos Javier Ogayar, and Francisco Ramón Feito. 2021. Comparison of GPU-based Methods for Handling Point Cloud Occlusion. In *Spanish Computer Graphics Conference (CEIG)*, Lidia M. Ortega and Antonio Chica (Eds.). The Eurographics Association. https://doi.org/10.2312/ceig.20211364

Adam Marrs, Benjamin Watson, and Christopher Healey. 2018. View-warped Multi-view Soft Shadows for Local Area Lights. *Journal of Computer Graphics Techniques (JCGT)* 7, 3 (2018), 1–28.

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. https://doi.org/10.13140/RG.2.1.1731.4326/1 Capturing Reality Forum 2015, Salzburg, Austria.

J. A. Orenstein and T. H. Merrett. 1984. A Class of Data Structures for Associative Searching. In *Proceedings of the 3rd ACM SIGACT-SIGMOD Symposium on Principles of Database Systems* (Waterloo, Ontario, Canada) *(PODS '84)*. Association for Computing Machinery, New York, NY, USA, 181–190. https://doi.org/10.1145/588011.588037

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

Anjul Patney, Stanley Tzeng, Kerry A. Seitz, and John D. Owens. 2015. Piko: A Framework for Authoring Programmable Graphics Pipelines. *ACM Trans. Graph.* 34, 4, Article 147 (2015), 13 pages. https://doi.org/10.1145/2766973

Ruggero Pintus, Enrico Gobbetti, and Marco Agus. 2011. Real-Time Rendering of Massive Unstructured Raw Point Clouds Using Screen-Space Operators. In *Proceedings of the 12th International Conference on Virtual Reality, Archaeology and Cultural Heritage* (Prato, Italy) *(VAST'11)*. Eurographics Association, Goslar, DEU, 105–112.

Paul Rosenthal and Lars Linsen. 2008. Image-space point cloud rendering. In *Proceedings of Computer Graphics International*. 136–143.

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. https://doi.org/10.1145/344779.344940

Darius Rückert, Linus Franke, and Marc Stamminger. 2022. Adop: Approximate differentiable one-pixel point rendering. *To appear in ACM Transactions on Graphics* 41, 4 (jul 2022).

Claus Scheiblauer and Michael Wimmer. 2011. Out-of-Core Selection and Editing of Huge Point Clouds. *Computers & Graphics* 35, 2 (2011), 342–351.

Kersten Schuster, Philip Trettner, Patric Schmitz, Julian Schakib, and Leif Kobbelt. 2021. Compression and Rendering of Textured Point Clouds via Sparse Coding. In *High-Performance Graphics - Symposium Papers*, Nikolaus Binder and Tobias Ritschel (Eds.). The Eurographics Association. https://doi.org/10.2312/hpg.20211284

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://doi.org/10.1111/cgf.14345

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.

Alex Vlachos. 2015. Advanced VR Rendering. Game Developers Conference, industry talk. https://www.gdcvault.com/play/1021771/Advanced-VR Accessed 2018.11.20.

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. https://doi.org/10.1016/j.cag.2008.01.010

Hank Weghorst, Gary Hooper, and Donald P. Greenberg. 1984. Improved Computational Methods for Ray Tracing. *ACM Trans. Graph.* 3, 1 (1984), 52–69. https://doi.org/10.1145/357332.357335