Virtualized Point Cloud Rendering

José Antonio Collado, Alfonso López, Juan Manuel Jurado, and Juan Roberto Jiménez

***Abstract*—Remote sensing technologies, such as LiDAR, produce billions of points that commonly exceed the storage capacity of the GPU, restricting their processing and rendering. Level of detail (LoD) techniques have been widely investigated, but building the LoD structures is also time-consuming. This study proposes a GPU-driven culling system focused on determining the number of points visible in every frame. It can manipulate point clouds of any arbitrary size while maintaining a low memory footprint in both the CPU and GPU. Instead of organizing point clouds into hierarchical data structures, these are split into groups of points sorted using the Hilbert encoding. This alternative alleviates the occurrence of anomalous groups found in Morton curves. Instead of keeping the entire point cloud in the GPU, points are transferred on demand to ensure real-time capability. Accordingly, our solution can manipulate huge point clouds even in commodity hardware with low memory capacities. Moreover, hole filling is implemented to cover the gaps derived from insufficient density and our LoD system. Our proposal was evaluated with point clouds of up to 18 billion points, achieving an average of 80 frames per second (FPS) without perceptible quality loss. Relaxing memory constraints further enhances visual quality while maintaining an interactive frame rate. We assessed our method on real-world data, comparing it against three state-of-the-art methods, demonstrating its ability to handle significantly larger point clouds.**

***Index Terms*—GPU-driven, GPGPU, point cloud rendering, out-of-core rendering, dynamic rendering, virtual memory system, level of detail, acceleration structures, rasterization, visibility, point-based models.**

I. Introduction

Graphics processing units (GPUs) have undergone a significant transformation, transitioning to hardware architectures with substantial memory capacities able to accommodate large datasets. This evolution encompasses not only visualization tasks but also the realm of general-purpose programming on GPUs, such as in Machine Learning applications [1]. Despite these advancements, even the most cutting-edge GPUs encounter challenges regarding real-time rendering of large point clouds, often surpassing the capabilities of current hardware. Remote sensing technologies and software solutions produce billions of points that exceed the capacity of video memory and harden the real-time processing to provide an interactive frame rate. In addition, the hardware rendering pipeline is oriented to rasterizing triangles rather than points with vertex-wise information, including color. These points eventually comprise even dozens of attributes, as occurs in Light Detection and Ranging (LiDAR) scans that cover vast areas, producing huge point clouds of up to several terabytes. An example is the OpenTopography data catalog,[1] where high-resolution LiDAR, radar, and photogrammetry datasets are publicly released.

Although rendering huge point clouds is computationally demanding, the performance of real-time visualization can be partially enhanced with level of detail (LoD) structures [2], [3], [4]. In this manner, the number of rasterized points diminishes in certain parts of the scene without being noticeable to the naked human eye. However, organizing huge point clouds into LoD structures is also time-consuming. Usually, the process stalls until the entire dataset is loaded and organized. Besides this, rendering point clouds has numerous other challenges. High-quality results require very dense point clouds; otherwise, gaps are visible during rendering. Second, colors are obtained by sampling real-world landscapes at discrete intervals, which may cause the perception of noise. Finally, vast point cloud datasets require significant storage, even though most frames do not utilize the majority of their spatial and vertex-wise attributes.

In this paper, we propose a virtualized point cloud system where the footprint of huge point clouds is maintained low both in the Random Access Memory (RAM) and Video Random Access Memory (VRAM). Points are organized into spatially coherent groups that enable rapidly performing frustum culling, hence discarding not visible groups. The proposed sorting will be proven to generate tighter groups than in the widespread Morton encoding. Furthermore, a distance-based LoD is implemented per group to deal with huge volumes of data that could be simultaneously visible. Among the visible points, those not present in the GPU are transferred to it, taking advantage of the large transfer capacity of modern-day Peripheral Component Interconnect (PCI) buses. A similar approach is followed in the Central Processing Unit (CPU) to load point chunks from the disk and then submit them to the GPU. Finally, the rasterization is enhanced by interpolating colors in gaps. The entire pipeline

Received 24 July 2024; revised 31 March 2025; accepted 9 April 2025. Date of publication 21 April 2025; date of current version 5 September 2025. This work was supported in part by the research projects under Grant PID2022-137938OA-I00, Grant TED2021-132120B-I00, and Grant PID2021-126339OB-I00, and in part by ERDF funds "A way of doing Europe", as well as by the Spanish Ministry of Science, Innovation and Universities via a Juan de la Cierva postdoctoral grant to the second author under Grant JDC2023-051785-I. Recommended for acceptance by T. Ropinski. *(Corresponding author: José Antonio Collado.)*

José Antonio Collado, Juan Manuel Jurado, and Juan Roberto Jiménez are with the Department of Computer Science, University of Jaén, 23071 Jaén, Spain (e-mail: jcaraque@ujaen.es; jjurado@ujaen.es; rjimenez@ujaen.es).

Alfonso López is with the Department of Computer Science and Engineering Systems, University of Zaragoza, 50018 Zaragoza, Spain (e-mail: alfonso.lopezr@unizar.es).

The code is available on GitHub.

This article has supplementary downloadable material available at https://doi.org/10.1109/TVCG.2025.3562696, provided by the authors.

Digital Object Identifier 10.1109/TVCG.2025.3562696

[1] https://portal.opentopography.org/dataCatalog

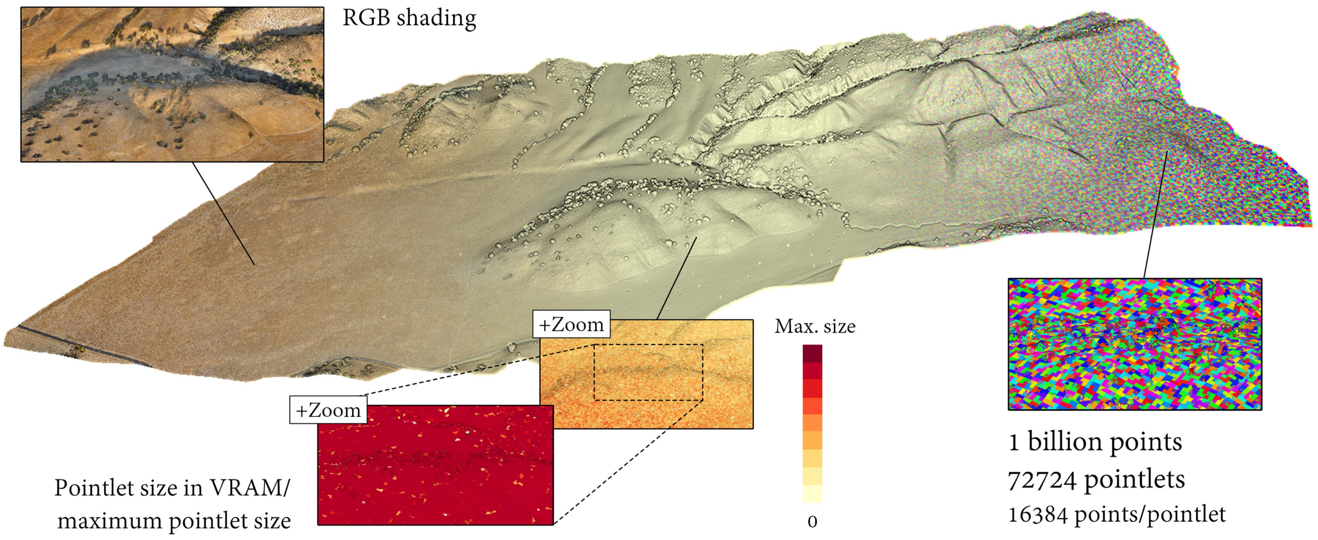

Fig. 1. Illustration of one billion points with their RGB colors (left side) and the clustering of points into *pointlets* (right side), i.e., groups of points that help in an efficient frustum culling and that are subsampled according to our LoD system. The centered area shows the percentage of points loaded in VRAM for every pointlet. As the camera gets closer, more points are transferred to VRAM.

enables the real-time rendering of dozens of billions of points while maintaining a minimal memory footprint, without any noticeable compromise on quality or user interaction.

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

- An asynchronous system designed for transferring and compacting data between disk, CPU, and GPU. Furthermore, we conducted experiments to determine the optimal parameters for our solution. We have also published the code on GitHub.
- Interactive visualization of several billions of points without delay nor significant visual cues from the LoD system.
- An alternative sorting method generating more compact groups of points, improving the frustum culling efficiency.
- A hierarchy-less structure that can be efficiently built and enables rapidly determining group-wise LoDs, as depicted in Fig. 1.

II. Related Work

A. Rasterization of Point Clouds

Many recent works have investigated the rendering of point clouds. Traditional rendering pipelines are not fully optimized for point-based rendering but rather for triangular meshes. Hence, the generation of huge point clouds of real-world scenarios requires the development of efficient methods for rendering these complex 3D models. It has long been proven that alternative GPGPU-based approaches offer better performance than their OpenGL counterpart, `GL_POINTS`. For instance, Open Computing Language (OpenCL) was first used to project points and their color using atomic operators together with an early depth test at the fragment level [5].

More recently, OpenGL's compute shaders were applied to determining visible points with the atomic minimum operator over 64-bit values encoding the distance to the view position and color (less significant bits) [6]. One of the main bottlenecks of previous work is that the whole point cloud is projected, while only a few points are visible in close-view projections. Later, Schütz et al. [7] grouped points into regular chunks of 10,240 points after sorting them with Morton encoding. The bottleneck of memory bandwidth usage was tackled by minimizing the reads: the points were quantized and encoded in 32 bits in three different buffers (low, medium, and high resolution) so that the highest resolution requires three memory accesses. Furthermore, several values were prefetched at every iteration to unroll the loop (one thread processes several points). Still, huge point clouds of several billions of points do not fit in the VRAM. Goel et al. [8] alleviated this drawback with data compression and real-time decompression using Huffman codes, rasterizing up to 6 B points.

In comparison, our work offers real-time visualization with only a small part of the point cloud in the VRAM. To this end, we adapted one of the most recent point cloud rasterizers [9]. In our work, the maximum number of allocated points is not strictly limited by the VRAM. Instead, it is bounded by the software according to the hardware capabilities and user requirements. In the worst-case scenario, the visible points do not fit in VRAM or RAM, hence increasing the data transferring between disk, CPU, and GPU.

B. Rasterization of Triangles Meshes

Akin to point cloud rasterization, recent innovations have been proposed for triangle mesh rendering. Traditional pipelines are optimized for this representation, but they underperform with small, pixel-sized triangles. To tackle this, *mesh shaders* [10], [11] were proposed to give developers more control over which triangles are going to be rasterized. Following these innovations, *Unreal Engine* released *Nanite* [12], combining software and hardware rasterization to render a huge number of triangles while maintaining an interactive frame rate. This is achieved through spatial clusters of triangles known as *meshlets* where different LoDs are pre-computed, with higher LoDs having fewer triangles while preserving the meshlet boundaries. Therefore, their connectivity is not broken regardless of the LoD required in each meshlet. Additionally, LoDs are selected according to the screen space size of the projected triangles. Having all these meshlets and LoDs in VRAM is memory costly, therefore they can be loaded on demand.

Following this approach, we split the point clouds into groups of points that are read and transferred to the GPU when they are visible from a viewpoint. From now on, we will refer to these groups of points as **pointlets** to emphasize they are composed of points instead of triangles as in triangle meshes. Moreover, there will be more primitives in pointlets than meshlets to equalize their occupied screen space in point clouds of several billions of points. We have previously used this spatially coherent division to accelerate point selections in large point clouds [13].

C. *Hole Filling and Colour*

The rendering of point clouds is particularly intricate since the RGB colors of low-density point clouds are harder to interpret. It is possible to exploit the data exchange between threads within a warp, i.e., a group of 32 GPU threads, to blend colors from overlapping samples and reduce the image noise [9]. Similarly, the color of the downsampled blocks of the point cloud can be estimated by weighting the contribution of several points. Regarding LiDAR point clouds, color has been previously enhanced by weighting RGB information with the observed intensity [14]. Further investigations in point cloud rendering are focused on photorealism to replicate transparency, shadows, and other effects [15].

Another feature that helps to interpret point cloud renderings is to highlight significant depth changes, such as in the edges of buildings. In this manner, geometry can be interpreted even from a single view. Without calculating the normals, it is possible to outline geometry changes with the depth buffer by leveraging directional light, ambient occlusion, and line shading [16]. A simpler, yet effective approach is eye-dome lighting (EDL), where the outlining factor is rapidly estimated from the maximum depth difference between a point and its neighbors [17]. It can even be modulated with a strength factor, thus making geometrical features more evident. Otherwise, bilateral filtering has been applied to enhance the values near the edges [14].

Another source of problems is holes due to low point density [18]. It can be partially solved by removing background points according to the solid angle w.r.t. their neighbors, and then using anisotropic filling [16]. However, the latter approach requires a high point density. During hole-filling, we can also find background pixels that should not be changed (they are part of the default background). These can be detected with convolutions [15]. Otherwise, pixels are filled and the image is smoothed to avoid aliasing. Another approach is to represent points as circular splats [14], [19] or square splats [20], both with adaptive size according to the camera distance. Finally, another promising field is Neural Radiance Fields (NeRFs) and Gaussian splatting. More specifically, the adaptive size of Gaussian splats [21], the removal of noise and the spawning of new points in incomplete regions [22] are of special interest. Yet, note that these approaches rely on Novel View Synthesis and therefore 1) require training and 2) necessitate the image dataset besides a starting point cloud.

Given that our work is oriented to dense point clouds with up to several billions of points, gaps mainly result from the LoD system, thus uniformly dispersed. Accordingly, we implement an efficient hole-filling algorithm that is extremely simple rather than visually appealing. Moreover, we propose an occlusion check that unlike Pintus et al. [16] does not require the surface orientation.

D. *Level of Detail of Point Clouds*

Several studies have endeavored to refine the point grouping process with two primary objectives: first, to rapidly build the LoD structure and second, to seamlessly integrate this optimization into the user experience, ensuring that the underlying LoD remains imperceptible.

The hierarchical structures for organizing point clouds are frequently named Layered Point Clouds (LPCs). They are binary trees whose deeper nodes offer higher LoDs and point density. These are particularly suitable for point cloud visualization since they are view-dependent. Therefore, the selected LoD changes according to the view position, and the LPC nodes can be discarded during rendering if they occupy less than a number of pixels on the screen [7]. A few variations [18] from this baseline approach are based on octrees [23], [24], [25], kd-trees [16], [26], voxelizations [14], grids [27] and ray-based structures such as sparse-voxel octrees (SVO) [24]. Other LPCs are Hierarchically Layered Tiles (HLT), which organize points into a hierarchy (e.g., quadtree), and then each node comprises multi-layer tiles, from higher to lower LoD. Moreover, a similar approach has been followed for voxelizations [14]. Notice that mid-level nodes in these structures have a subsample of points, which may cause aliasing since these could significantly differ from points in higher LoDs. To solve this, colors can be averaged in intermediate nodes [6]. Wavelet decomposition has also been applied over ordered points to create a hierarchical LoD over the basis of a kd-tree organized using Principal Component Analysis (PCA) [26]. Another path yet to be explored is the nesting of several data structures to better organize the points [28].

Hierarchy-less structures have also been treated to reduce aliasing and provide a continuous LoD (cLoD). Schütz et al. [6] stored different subsampled point clouds (levels) in a flat array, each denser than the previous one, which were then accessed to iteratively build an image. Depending on the distance, the number of rendered levels varies. A similar concept was used by Dachsbacher et al. [29]. Moreover, cLoD can be adapted to fit the distribution ratio of distance-based LoD (dLoD) [30].

Another option is to mix both approaches by flattening hierarchical data structures to ease the data access [29]. In their work, the octree was flattened to sort the nodes from higher to lower error (measuring how well a parent disk approximates the children's disks). Finally, it is also possible to project a set of random points over new frames whose baseline is the re-projection of previously visible points. In this manner, the new frames are generated in several steps using the points visible in previous frames [31] and no LoD levels must be calculated before rendering.

Unlike the revised works, we use an unstructured, Hilbert-code ordered point cloud whose underlying data structure is a simple array. It is more trivial to manage and does not require additional metadata for the data structure. For instance, Sim-LOD [2] improved their predecessor work [7] in terms of LoD; however, both the octree metadata and the point clouds are stored on the GPU.

E. Spatial Organization of Point Clouds

Points can be organized into spatially coherent groups with similar outcomes during the rasterization, hence improving data access patterns. A frequent sorting method is Z-curves (Morton codes) to facilitate data locality [30], and can be further enhanced by shuffling points to also distribute the workload throughout the available graphics processing clusters (GPCs). However, splitting points sorted with Z-curves into regular chunks also leads to spatially large groups that cannot be omitted during frustum culling checks [7]. Otherwise, Hilbert encoding has also been studied to sort points [32], [33]. It has been primarily applied to improving database access and storage since it is supported by several well-known database frameworks. Instead of organizing points into Z-shapes, Hilbert codes have a maze-like shape [34] that resembles the octree generation [35]. Hence, it mitigates the large jumps reported in Morton codes.

F. Management of Point Cloud Data

The efficient management of huge data volumes has been previously investigated, especially focused on spatial queries on databases [32], [36]. A few studies have vaguely referred to the asynchronous data transfer between the storage system, CPU and GPU [37]. While the term massive point cloud management is frequently observed, these approaches are rather focused on rapidly building LoD-based structures and reducing the wait time for users [2], or comprising data to minimize the storage footprint [4], [8]. Deibe et al. [4] dynamically computed the tile layers in the HLTs in a Web browser and points that were not already present in memory were transferred from the server side. Then, the optimal number of tiles and points in higher LoDs was evaluated. A similar on-demand data transfer was implemented by having duplicate rendering and depth buffers, while CPU threads received requests to load points from the disk [3]. A frame rate below 20 was reported for point clouds of up to 65.5 M points. Another approach focuses on computing a set of textured meshes that accurately depict the original point cloud [38], reporting a lower memory footprint when compared to the entire point cloud. Huge point clouds can also be iteratively displayed, even with hierarchical data structures such as octrees, which implies updating them progressively [2]. These are, nevertheless, changed in the GPU by simply transferring points from old to new nodes. However, the most common assumption in point cloud studies is that they fit in the GPU, therefore not requiring any virtualization strategy [7], [31].

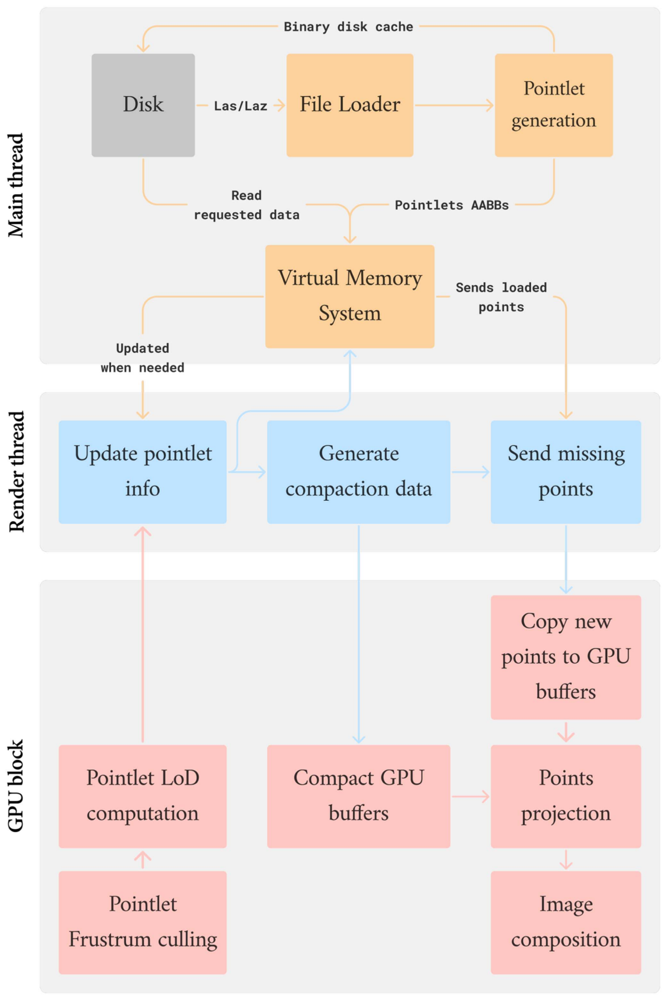

Fig. 2. Overview of the proposed method. The main thread oversees a group of worker threads, which are responsible for loading the requested points, handling real-time accesses to small segments of each pointlet. Furthermore, a dedicated render thread manages the communication between the GPU and the main thread.

III. Methodology

Our methodology revolves around subdividing point datasets into pointlets, from which the LoD system and data requests are organized. In the following, we describe how points are structured (Sections III-A and III-B), and then, we explain the virtual memory system that enables a low footprint in both GPU and CPU (Section III-C). Finally, the rendering is enhanced with hole filling in Section III-D. The overview of our method is depicted in Fig. 2. The Virtual Memory System (VMS) is the core of our work since it collects the point data. The render thread checks whether the points requested by the GPU visibility checks are available in the VMS. These points are either already loaded or must be asynchronously read from the disk. On the other hand, the GPU block manages the rendering stage, including visibility checks.

A. Pointlets

The quality of a pointlet is assessed based on its dimensions. For instance, a pointlet with a smaller bounding box will enable

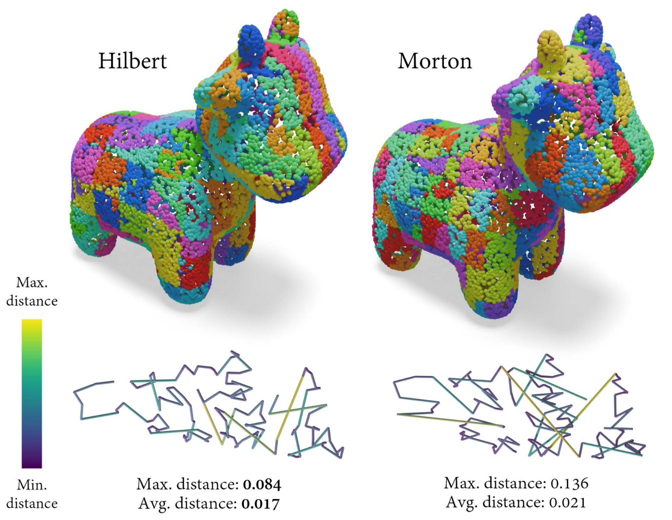

Fig. 3. The points of the same model sorted using Hilbert and Morton encoding, and split into uniform pointlets. A random pointlet is picked to illustrate how Hilbert tends to show less spatial jumps, including the average and maximum distance within that pointlet.

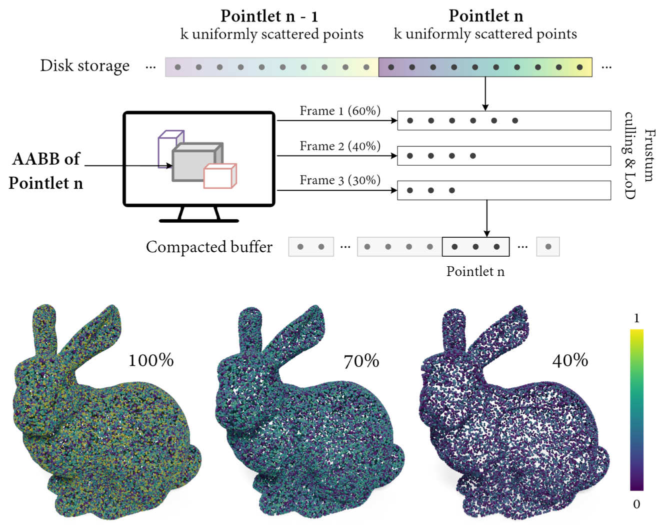

Fig. 4. Memory layout when rendering point clouds. In the disk, the sorted points are organized into pointlets, with the points within each pointlet shuffled. A certain percentage of points from each pointlet is then selected based on our Level of Detail (LoD) system for a given frame. These selected points are transferred to the GPU and compacted into a point buffer. In the figure below, each pointlet of the bunny model is rendered using the same percentage of points. This does not occur in our renderer unless every pointlet occupy a similar portion of the screen space, such as in distant views.

more accurate frustum culling and LoD selection. Hence, the pointlet quality plays a crucial role in this work, as this factor will drive the performance of the rest of the stages. State-of-the-art methodologies frequently employ Morton codes to organize point clouds spatially [9]. The pointlets are then built by selecting as many contiguous points as the pointlet size. Despite being computationally efficient and producing spatially coherent results, sorting points using the Morton encoding also leads to spatial disruptions derived from large jumps as illustrated in Fig. 3. A more optimal approach involves finding the k-nearest neighbors (KNN), frequently solved in the literature by organizing points into a k-D tree. However, building spatial data structures is typically more time-consuming than the previous encoding and sorting approach.

Consequently, we have sorted our point clouds using another space-filling curve, known as the Hilbert curve, which has good locality properties for spatial applications [39], [40], as illustrated in Fig. 3. Additionally, Hilbert encoding can be computed from 30-bit Morton codes [41] and therefore can be smoothly incorporated into the state-of-the-art work. This approach has a significantly lower response time than building spatial structures while achieving better spatial clustering. Finally, the point cloud is split into fixed-size pointlets once sorted. Their size, k, is calculated as follows, depending on the number of points, n:

$$k = 2^{\log_2 \frac{n}{m}}; \tag{1}$$

where m is the number of pointlets that leverage the number of tasks and their workload. In our implementation, m equals 100,000, though it can be adjusted depending on the target hardware.

Upon generation of the pointlets, their axis-aligned bounding boxes (AABBs) are calculated and stored in a GPU buffer that will not be modified again throughout the process life. During rendering, AABBs enable estimating the number of pixels the projected pointlets occupy on the screen, hence determining their LoD.

B. Level of Detail

The first rendering stage is to discard pointlets early based on the frustum culling of their AABBs. Then, AABBs are projected into the image space to estimate the covered pixels, akin to other state-of-the-art methods [7]. From here, this coverage helps determine the importance of each pointlet, l, as illustrated in Fig. 4. l is calculated as follows:

$$l_i = \gamma_i \cdot \left\| \frac{v}{2} \cdot p_{\mathtt{center}_i} - \frac{v}{2} \cdot p_{\mathtt{max}_i} \right\| \tag{2}$$

where γ_i is used to cope with pointlets with anomalous dimensions, and it is calculated as shown in (3). v refers to the viewport size, given as a tuple (width, height), and p are points projected as shown in (3).

$$\gamma_i = \frac{\left(\frac{\sum_j^n \| a_{\mathtt{center}_j} - a_{\mathtt{max}_j} \|}{n} \right)^2}{\| a_{\mathtt{center}_i} - a_{\mathtt{max}_i} \|^2}$$
$$p_{\mathtt{center}_i} = \frac{(P \cdot a_{\mathtt{center}_i})_{xy}}{(P \cdot a_{\mathtt{center}_i})_w}; \; p_{\mathtt{max}_i} = \frac{(P \cdot a_{\mathtt{max}_i})_{xy}}{(P \cdot a_{\mathtt{max}_i})_w} \tag{3}$$

with P being the camera projection matrix, n being the number of pointlets and a referring to AABB points, either its center or maximum point.

However, (3) does not provide the number of points rendered for every pointlet. Initially, we multiplied the result of (3) by a ϱ factor, with values in range $[30, 60]$ providing good results. The main drawback of this approach is that ϱ is not an intuitive factor. Furthermore, it leads to low occupancy of the point buffer for far and closeup views. Instead, we normalized the importance of each pointlet by dividing it by the overall sum of all the weights ($\sum_i^n l_i$), and multiplied it by a static point bucket size, $\mathcal{B}$. It is much more intuitive since it resembles the maximum number of points that can be rendered per frame. Instead of rendering as few points as possible, it leads to using the whole point bucket size as long as there are sufficient points. By default, it is set to 25M points, though we have experimented with a bucket size of up to 90M points.

In this manner, we smoothly integrate a distance-based LoD by estimating the projected AABB length in pixel units. It is also comparable to continuous LoD as our point cloud structure is not hierarchical and pointlets are expected to cover fewer pixels as they get further from the camera.

C. *Virtual Memory System*

The virtual memory system represents the core of this work. Its main purpose is to ensure the seamless loading of points upon request while effectively removing unnecessary ones. A brief insight into our CPU and GPU memory layout is provided to understand this section better.

First, two buffers are allocated in VRAM to store 1) positions and 2) any other kind of attribute, such as RGB colors. The length of these buffers must be sufficient to allocate the points selected by the LoD system, whose number is bounded by $\mathcal{B}$. However, our approach requires reusing the information from the previous frame in the following one. Therefore, we duplicated both buffers to maintain the information of two consecutive frames. Previous work [7] reported that, from the overall number of points, only a few were finally rasterized; for instance, from 1 B points, 51 M points were rasterized at most. Also, note that the maximum size of Shader Storage Buffer Objects (SSBOs) in OpenGL is significantly restrictive (2 GB), regardless of the VRAM capacity. Accordingly, a point buffer of 2 GB allows us to maintain up to 178,956,970 points (2 GB / 12 bytes). Although limited in size, this number of points is far from the number of potentially visible points since our LoD system will significantly decimate them. Moreover, we can even control the maximum number of loaded points using the bucket size, $\mathcal{B}$. In addition, another buffer is necessary for storing the pointlets' AABBs.

We also utilize four additional buffers allocated in RAM, though they are directly accessible from the GPU via Direct Memory Access (DMA). These buffers are implemented using OpenGL's mapped memory. The first two buffers transfer new points to the GPU, with a size determined during compilation according to a user-defined constant. The third one is utilized to transfer the results of the LoD step from the GPU to the CPU. The last one is employed for transferring compaction information, which will be further explained in Section III-C1.

On the CPU side, dynamic memory management is far more straightforward. Pointlets are represented by objects with four fields: the number of required points, the number of already loaded points and two buffers with the positions and attributes of the loaded points. Hence, the data in the CPU is managed in a pointlet-wise manner that avoids transferring data to a global buffer and compacting information. Each pointlet buffer is resized as required, according to the memory limitations and the information demanded in a frame. Additionally, another vector stores the LoD results of the last frame.

Regarding data types, points are represented by three simple floating point values (12 bytes). Other representations split point

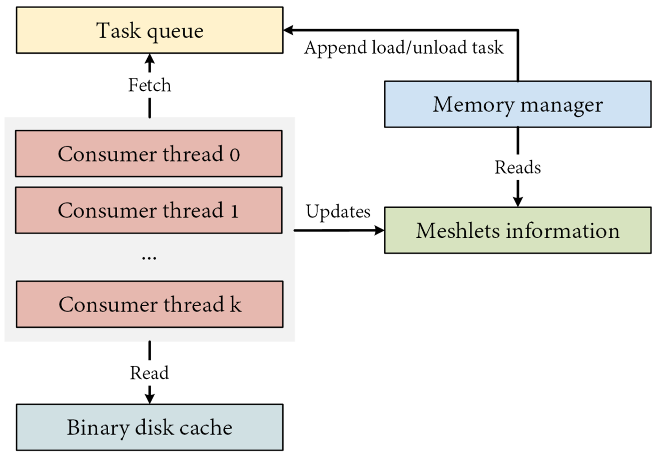

Fig. 5. Management of system-side memory. The memory manager instantiates Load/Unload tasks based on current pointlet information. Subsequently, task consumers retrieve these tasks from the parallel queue to solve them.

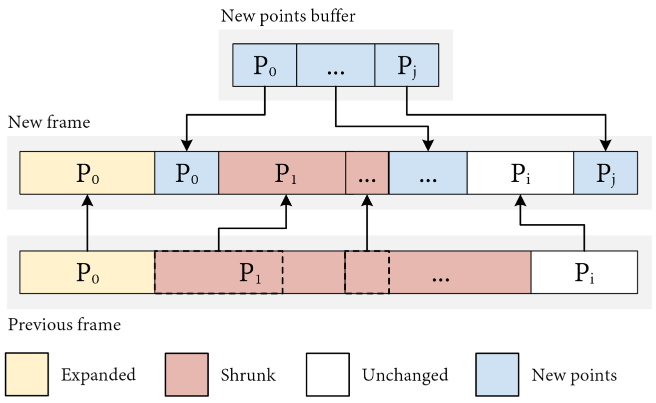

Fig. 6. Compaction procedure, including the transfer of new points.

coordinates and encode them at different resolutions to minimize data reads for lower LoDs [7]. However, VRAM bandwidth is not our main bottleneck and we opted for a simpler data representation. On the other hand, RGBA colors are encoded as 4-byte values (`uint32`). This representation allows us to render many other attributes that can be encoded in 4-byte floating-point data.

Following is explained the data transfer procedure step by step:

1) GPU Buffer Compaction: We rebuild the point and attribute buffers on a per-frame basis. From now on, we will refer to rebuilding the point buffer, but remember that points and attributes are stored in different buffers, as explained earlier in this section. This stage employs the results of the LoD system to annotate the discrepancies between the points to be rendered in the following frame and those rendered in the previous frame, which are already loaded into the GPU. As a result, we compute the *compaction info* for each pointlet, which specifies the source range from which points are to be copied and the destination range where they will be copied, ensuring sufficient space for new points. Subsequently, this information is transferred to the GPU, where a shader copies each point and attribute to other buffers, as shown in Fig. 6.

2) Transferring Missing Points to the GPU: Transferring points to the GPU involves several steps. First, we determine which new points are necessary using the compaction information. There are three potential outcomes from here: 1) the requested points are already loaded in RAM, 2) they are yet unloaded or 3) some points are loaded while others are not. In the first scenario, which is also the simplest, the points are written into the new points buffer, along with their corresponding position. In the second case, the number of points required for a pointlet is modified since the points cannot be transferred yet to the GPU. Hence, the virtual memory system loads them as soon as possible; meanwhile, fewer points than requested are rasterized. We also adjust the LoD of the current frame to reflect the number of points loaded in the CPU. Finally, the hybrid scenario consists of transferring points available in RAM to the GPU, while requesting the remaining points. In that case, the LoD buffer is adjusted to include only the points already available, excluding those that need to be fetched from disk.

3) Managing System Memory: To prevent the system from being idle, point requests are constantly arriving at our task queue. Given the assumption that the whole point cloud cannot be maintained in RAM, it is necessary to determine which points are simultaneously available. To accomplish this, as illustrated in Fig. 5, the virtual memory system monitors the number of points required in every pointlet. If the number of required points does not match the number of loaded points, the system enqueues a task aimed at loading/unloading points. However, several criteria must be met when appending these tasks. First, a load task is enqueued whether the number of required points is higher than the number of loaded ones, regardless of the system state. On the other hand, an unloading task is enqueued if the number of required points is lower than the number of loaded points and the system is approaching the maximum RAM capacity. Hence, loaded points are not erased as long as there is sufficient memory. Note that, the queue size is established at compile time, limiting the maximum number of pending tasks. In this manner, the latency for solving individual tasks is kept low, and they can be solved in a few frames at most.

From the perspective of a consumer thread, whose number is variable, tasks are first processed by comparing their information with the current pointlet information. This approach ensures that the required task does not collide with the most recent state. Each pointlet has its own *mutex* variable to prevent other consumer threads from modifying the same pointlet. Hence, if another thread changes a pointlet, the thread that failed to access that pointlet concludes since it is already being updated based on its current state. Depending on the free RAM, task consumers can load up to 25% more points than demanded, thus anticipating the following requests.

D. Software Rasterization

The point cloud rasterization refers to the projection of points into the viewport. As our primary objective is not improving the visualization, but the performance, we implemented the baseline projection procedure. The points are mapped to the viewport pixels and an atomic `min` operator is employed to identify the point with the minimum depth, together with its color. From here, it is trivial to adopt other color-enhancing techniques, such as High-Quality shading (HQS) in Schütz et al. [9].

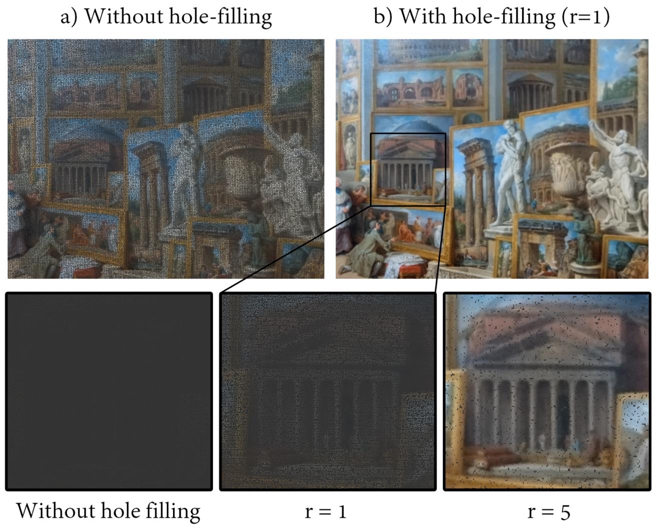

Fig. 7. Comparison of original point cloud and gaps filled using a radius of 1. The closeup view of the painting illustrates the need for larger radius to fill larger gaps..

However, the designed LoD system subsamples the point cloud, hence showing a higher number of gaps. Therefore, a hole-filling method must be implemented to mask these gaps while still subsampling the pointlets to enable 1) reducing the VRAM and RAM footprint and 2) improving the frame ratio.

1) Hole Filling: Unlike mesh shading, the rendering of point clouds may present gaps due to insufficient density in certain areas. This drawback is pushed further by LoD systems that reduce the number of rasterized points. Therefore, filling these gaps prevents users from perceiving the rendered image as noisy or incomplete. Instead of enhancing the color in nonempty pixels as in HQS, we estimated the color in empty pixels. A pixel is filled with the mean color of the surrounding pixels if there is at least one nonempty pixel in the $n \times n$ neighborhood. However, empty pixels are not filled when dealing with holes larger than $n \times n$ pixels, as shown in the zoomed-in views in Fig. 7. Additionally, no boundary detection is performed, thus filling the edges of the point cloud. Notice that a boundary is defined as the frontier between the point cloud and the background. Therefore, filled edges only appear from far views (where they are notably less visible). However, this drawback comes at the expense of a significantly fast hole-filling algorithm.

2) Visibility: Large point clouds are typically denser and do not display background points unless observed from closeup views. Despite being seldom observed, background points may harden the interpretation of the point cloud, as observed in Fig. 9. Previous work has addressed this limitation [16], but it relies on data unavailable in our late shading stages, such as positions, or data unavailable in our point clouds, e.g., normals. Specifically, only colors and depth values are available when composing the final image after projection.

Given the depth values of two points, we can calculate an angle β enclosed between the vectors illustrated in Fig. 8 as

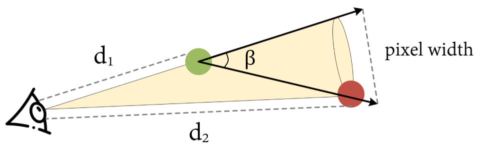

Fig. 8. Occlusion checks are conducted by calculating β, the angle between two rays cast from the main point, colored in green.

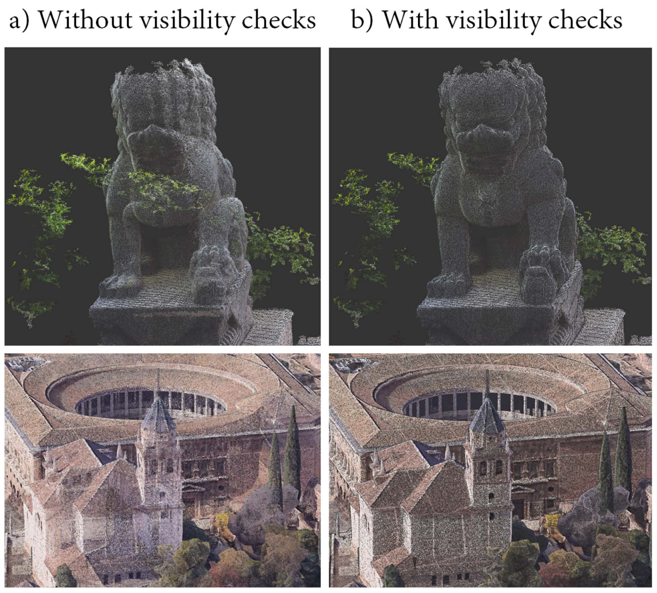

Fig. 9. Point clouds rendered by a) deactivating and b) activating occlusion checks.

proposed in (4):

$$\begin{aligned} pixel\ depth &= 2 \cdot \max(d_1, d_2) * \tan \frac{FOV_x}{v_{width}} \\ \beta &= \left| \arctan \frac{pixel\ width}{d_1 - d_2} \right| \end{aligned} \tag{4}$$

with $pixel\ depth$ being the horizontal displacement and $d_1 - d_2$ being the vertical displacement. Filtering by $\beta > 0.1$ proved to be an effective threshold for the datasets used in this work. FOV_x and v_{width} are the camera's horizontal field of view (FOV) and viewport width, respectively. Similar to hole-filling, this stage requires a radius to check the visibility of points w.r.t. their surroundings.

IV. Results

We conducted several experiments to showcase the capacity of loading huge point clouds and the performance of our work despite transferring significant volumes of data on demand. Comparisons are performed against the state-of-the-art study: Schütz et al. [7], SimLOD [2] and Goel et al. [8]. The first and last implement a non-hierarchical LoD system, whereas SimLOD builds an octree. However, all store the whole point clouds and data structures in VRAM. To provide a fair comparison, HQS shaders were disabled during the frame time recording.

Experiments were performed on a PC with Intel I9 12900 K 3.2 GHz, 64 GB RAM, RTX 4070 GPU with 12 GB VRAM (Ada Lovelace architecture), a 2 TB Samsung 990 PRO PCIe 4 NVME (MZ-V9P2T0BW) SSD and Windows 11 OS. Our method was implemented in C++20 using C++ multithreading utilities for CPU parallel processing and OpenGL 4.6 [42] for rendering and GPGPU (general-purpose computing on GPU). The datasets were either obtained from OpenTopography or collected by us. We used five point clouds (see Fig. 10): 1) Alhambra (100 million points), 2) solar plant (500 million points), 3) San Andreas fault (1 billion points) [43], 4) Dangermond (8 billion points) [44] and 5) San Simeon and Cambria faults (18 billion points) [45].

TABLE I
Results of Evaluated Pointlet Metrics, Together With the Response Time for Organizing the Point Cloud Into Pointlets

Approach	Metric	Alhambra	Solar plant
Morton	Maximum extent (m)	14.198	197.419
	Mean extent (m)	0.300	2.305
	Standard deviation (m)	0.226	2.307
	Mean squareness (m)	0.085	0.899
	Response time (s)	**1,974**	**8,553**
Hilbert	Maximum extent (m)	**13.124**	177.566
	Mean extent (m)	0.253	1.971
	Standard deviation (m)	**0.154**	**1.721**
	Mean squareness (m)	0.074	0.798
	Response time (s)	3,473	13,170
K-d tree	Maximum extent (m)	19.361	**170.785**
	Mean extent (m)	**0.246**	**1.917**
	Standard deviation (m)	0.218	2.268
	Mean squareness (m)	**0.054**	**0.645**
	Response time (s)	862,929	43,053,817

The best results are highlighted in bold.

A. Pointlet Metrics

The Hilbert curves were adopted to reduce large space jumps reported in previous work. This improvement is observed during rendering and in spatial metrics. We evaluated the bounding boxes of pointlets to extract their maximum and mean extent, standard deviation and mean squareness (deviation from a perfect square). Table I presents the results obtained over the first two point clouds, using Morton and Hilbert encoding and a K-d tree. In addition, we incorporated the response time for organizing the point cloud into pointlets.

By observing Table I, it is clear that Hilbert curves leverage response time and required pointlet properties. K-d trees have excellent results, at the expense of being significantly more time-consuming. Though spatial queries were not handled in parallel in K-d trees, the response time is still far from the results obtained with Morton and Hilbert encoding: ~11.95 hours for 500 M points, in contrast to ~8.5 seconds for Morton and ~13 seconds for Hilbert curves. Note that, we built Hilbert codes from Morton codes, and therefore, it is impossible to improve the response time.

B. Performance

The principal objective of this study is to maintain a low memory footprint in CPU and GPU while providing a stable

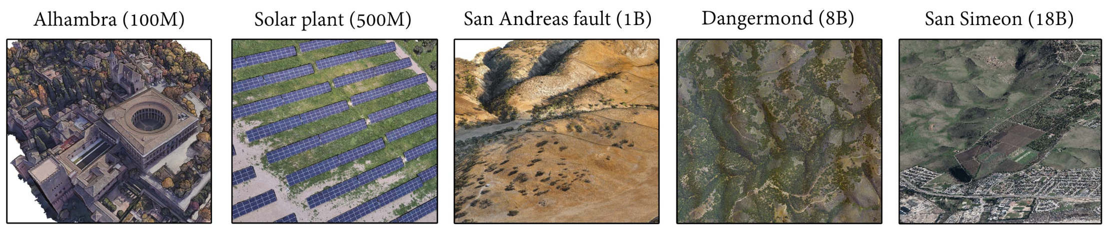

Fig. 10. Point clouds over which the experiments were conducted, ranging in size from a few hundred million to 18 billion points.

TABLE II
COMPARISON OF THE PROPOSED METHOD USING $\mathcal{B} = 25\,\text{M}$ AGAINST THE STATE-OF-THE-ART WORKS [2], [7], [8]

Data Set	#points	Method	Avg. FPS	Min. FPS	Max. FPS	1% min. FPS	0.1% min. FPS
Alhambra	100M	Schütz et al., 2022	**341.6**	**209.9**	**355.4**	**292.5**	**209.9**
		SimLOD	250.6	127.1	457.7	150.5	135.3
		Goel et al., 2024	161.9	130.2	242	141.4	134.5
		Ours	110.9	35.4	173	40.1	35.7
Solar plant	500M	Schütz et al., 2022	58.8	49.6	71.3	51.9	49.6
		SimLOD	NA	NA	NA	NA	NA
		Goel et al., 2024	37	10.1	2,209.2*	10.6	10.1
		Ours	**111.6**	**50.8**	**160**	**56.2**	**51.3**
San Andreas	1B	Schütz et al., 2022	17.6	14.5	18.3	15.3	14.5
		SimLOD	NA	NA	NA	NA	NA
		Goel et al., 2024	82.6	**71.3**	108.3	**72.8**	**71.4**
		Ours	**110.5**	43.5	**195.5**	50.4	43.5
Dangermond	8B	Schütz et al., 2022	NA	NA	NA	NA	NA
		SimLOD	NA	NA	NA	NA	NA
		Goel et al., 2024	NA	NA	NA	NA	NA
		Ours	**101.8**	**32.3**	**165.68**	**40.6**	**32.3**
San Simeon	18B	Schütz et al., 2022	NA	NA	NA	NA	NA
		SimLOD	NA	NA	NA	NA	NA
		Goel et al., 2024	NA	NA	NA	NA	NA
		Ours	**73.1**	**30.8**	**140.3**	**32.9**	**30.8**

The best results are highlighted in bold. NA (not applicable) indicates that the point cloud could not be fully loaded. The asterisk indicates that the program malfunctioned and failed to load the point cloud, resulting in incorrect statistics.

rendering performance. We evaluated this by recording the frame ratio obtained when visualizing point clouds ranging from a few hundred million to up to several billions of points. Since our work depends on the viewpoint due to the frustum culling stage, reporting only the mean frame ratio does not reflect well the nature of our method. To this end, we conducted several experiments.

First, we collected the frame times while orbiting around the point cloud. While following this path, we changed the zoom level by increasing and decreasing the vertical field of view. We established the bucket size, $\mathcal{B}$, to 25 M points. Table II compares the result of our methods against the state-of-the-art, and Fig. 11 illustrates the average FPS. Note that some rows of Table II display 'NA' because the compared approaches could not fully load all the point clouds.

Schütz et al. achieved the best performance when rendering the smallest point cloud since it was fully loaded into VRAM. Similarly, SimLOD obtained good results, despite having a slight overhead from the LoD system. However, these results did not extend to larger point clouds. SimLOD could not fully load them, and Schütz et al.'s approach struggled when projecting several hundred million points. Additionally, the method by Goel et al., which shares the same baseline renderer as Schütz et al. exhibited similar behavior, with slightly worse performance due to real-time decompression. However, this does not hold for the San Andreas point cloud, where Goel et al. achieved better results. The key advantage of real-time decompression is its ability to reduce GPU memory reads, making it more beneficial than direct memory access when handling larger point clouds. On the other hand, the largest two point clouds could not be

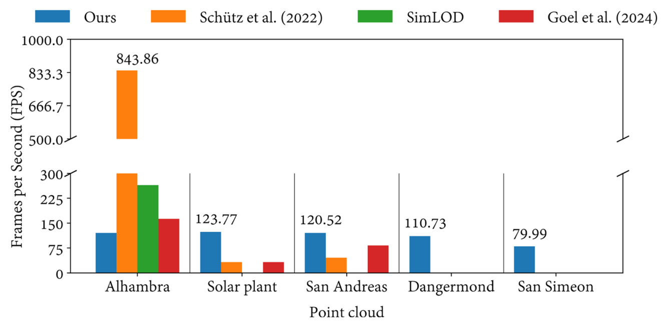

Fig. 11. Average number of FPS per point cloud and approach. The text above the bar highlights the highest FPS average per point cloud.

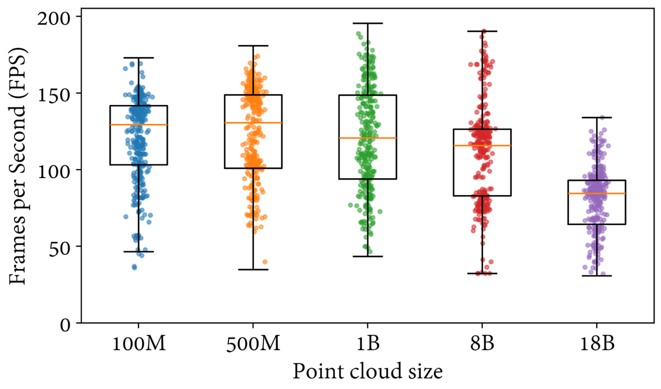

Fig. 12. Box plot of FPS recorded by orbiting the camera and zooming in and out while maintaining the focus at the center of the point cloud.

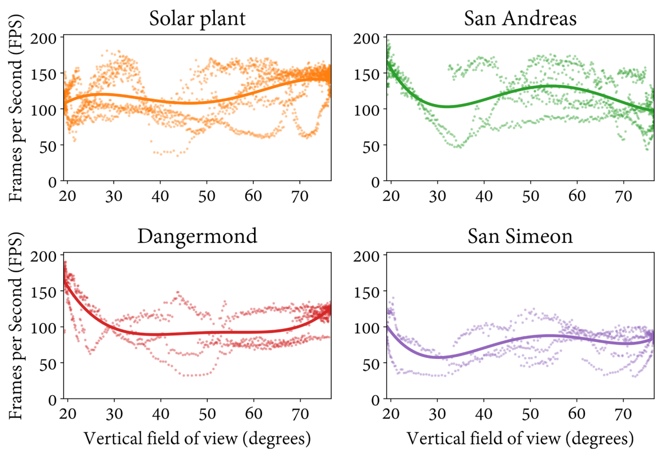

Fig. 13. Relation between recorded FPS and the camera's vertical FOV. The lines reflect the fourth-degree polynomial fit to the recorded values, represented with dots.

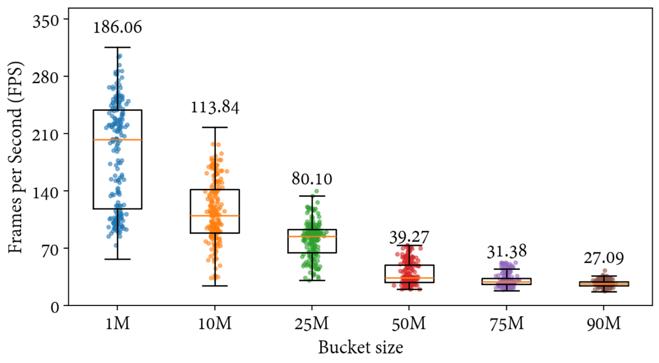

Fig. 14. FPS collected while varying the bucket size for rendering a point cloud of 18 B points.

processed by any of the compared approaches. In contrast, our method offers a more stable and scalable solution, regardless of point cloud size.

Fig. 12 illustrates the interval of recorded FPS using our approach with $\mathcal{B} = 25\,\mathrm{M}$, whereas Fig. 13 tries to correlate the camera's FOV and the FPS rate. Both figures illustrate the FPS collected by zooming in and out while the camera orbits around the point cloud. As expected, there is no clear correlation since the number of rendered points varies significantly across different point clouds, even with the same FOV. Moreover, even within a single point cloud, performance fluctuates under the same FOV due to factors such as previously observed points, variations in geometry within the frustum and non-uniform point density. Consequently, the recorded FPS values primarily depend on the number of points to be compacted and the camera path leading to a frame. The main takeaway from Fig. 13 is that performance cannot be reliably predicted while the camera moves and highly depends on the specific point cloud being rendered. However, Fig. 12 also indicates that, on average, performance remains relatively consistent, regardless of the point cloud size.

In summary, correlating the recorded FPS values with all the involved parameters, and illustrating it, is not straightforward. At most, we can conclude from Fig. 13 that lower FOVs result in slightly lower performance. This is expected, as they require rendering fewer pointlets at their maximum LoD, many of which may not yet be loaded. In contrast, far views render a larger number of pointlets with fewer points, either already loaded or available within a few frames. Additionally, far views involve fewer visibility changes, reducing compaction overhead.

Another relevant experiment is to record the performance with different bucket sizes. Previously, we used $\mathcal{B} = 25\,\mathrm{M}$, but larger sizes offer fewer visual cues of the LoD system (it is further evaluated in Section IV-C). Therefore, Fig. 14 shows the average FPS and the interval of observed values for a point cloud with 18 B points when changing $\mathcal{B}$. As depicted, larger bucket sizes worsen the performance but maintain an interactive frame rate. Note that we conducted this experiment based on the conclusions of Schütz et al. [7]. For a point cloud of 1 B points, the maximum number of simultaneously rendered points was 51M points. Also, we considered that larger point clouds typically grow in spatial size due to more extensive surveys. Therefore, the conclusions drawn from [7] are valid for point clouds larger than 1 B points. Furthermore, the bounds of 51M points can be even lowered with the help of the LoD system and hole-filling.

Fig. 15 provides further details into the compaction delay by illustrating the average response time across five rendering stages for two point clouds, containing 1 B and 18 B points, while changing the bucket size. As observed, the time spent on compaction increases linearly with the bucket size, although it remains similar to the response time of the projection phase. Therefore, the combined overhead of compacting and projecting visible points nearly doubles the frame time of the first compute-shader-based renderer described by Schütz et al. [9]. Besides this, there is no linear growth in time as the point cloud size increases. It rather depends on the camera path and the point cloud features.

For further insight into real-time performance, we encourage the readers to watch the video published as Additional data. It shows the user interaction in our application while visualizing 18 B points with $\mathcal{B} = 25\,\mathrm{M}$. To this end, we transition from closeup to far views, showing no visual cues of real-time data transfers nor subsampling from the LOD system.

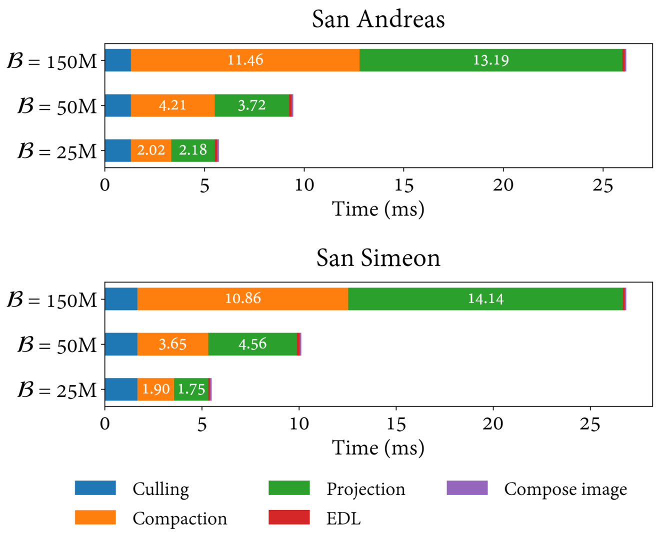

Fig. 15. Breakdown of the average response time across the five rendering stages.

C. Visual Comparison

This paper comprises data transfers between disk, CPU and GPU, but also diminishes the number of simultaneously rendered points with LoD techniques. The experiment illustrated in Fig. 16 checks the visual differences between ground-truth images (without LoD and hole-filling) and two other images: 1) with LoD and no hole-filling and 2) with LoD and hole-filling. We tested these scenarios over sparser point clouds that better show the limitations of both components (Alhambra, 100 M points, and a subsampled version of San Andreas fault, 187 M). Larger point clouds grow in spatial size and density, thus showing fewer gaps when enabling the LoD. Also, disabling LoD is impossible for larger point clouds since the maximum size of SSBOs is 2 GB, i.e., 187 M points. We used a bucket size, $\mathcal{B}$, of 187 M to render the ground truth, and $\mathcal{B} \leftarrow 50\,\text{M}$ for testing the LoD system. Images were rendered at a resolution of 1920×1080, and two views were proposed for each point cloud (closeup and far view). The distance was computed as the sum of errors in red, green and blue channels, and was normalized according to the maximum error. The Peak Signal To Noise Ratio (PSNR) is illustrated in the third column to quantify the error. Remember that the objective is to maximize the PSNR.

The differences in the third column come from representing pointlets with fewer points, hence not matching the ground-truth color. In far views, colors slightly vary while gaps are hardly visible since individual pointlets represent a tiny portion of the viewport. On the other hand, closer views render fewer pointlets that are not as undersampled as in far views, but also show gaps due to point cloud sparsity. The differences in the fifth column come from undersampled pointlets and filled gaps. In both views, the rendered gaps are mainly due to the point cloud sparsity and not due to the LoD system. Therefore, gaps are even visible in the ground-truth image.

Two main conclusions can be drawn from Fig. 16. First, closeup views are barely affected by the LoD, and second, the error from the LoD is significantly more apparent in far views. However, it is only noticeable due to the slight color variations and not due to gaps. Sparser point clouds benefit from the hole-filling with small radii, though it can be increased at the expense of worsening the performance.

Another relevant factor in the LoD system is $\mathcal{B}$, the maximum number of points rendered in a frame. Fig. 17 illustrates the PSNR obtained when decreasing β for closeup and far views of a solar plant. As expected, the visual differences are far more visible with a lower bucket size. Although we picked a default bucket size of 25M to leverage performance and rendering fidelity, visual differences are partially mitigated with hole-filling.

V. Conclusion and Future Work

We have introduced a system capable of rendering huge point clouds on computers with limited memory capabilities. Our system dynamically loads points as required, while those no longer needed are discarded, ensuring a minimal impact on performance. We have compared our work with three state-of-the-art methods, remarking how our method keeps a real-time interaction no matter the point cloud size. The proposal is based on an architecture that employs pointlets as a fundamental unit for determining the required points before being stored in the memory system. We have proposed the usage of Hilbert curves as an alternative to Morton order in the computation of the pointlets. To our knowledge, the Hilbert curve has not been used before in point cloud rendering. Using this approach, the clustering of points is optimized, thus enabling a more effective frustum culling and a non-hierarchical LoD system that uniformly subsamples the pointlets. Additionally, we have proposed visual enhancements to remove occluded points and fill gaps derived from sparse point clouds and LoD subsampling.

In this study, the experiments have been conducted on a desktop computer; however, the proposed architecture has the potential to be extendable to rendering large point clouds on mobile devices. While these devices are currently far from desktop GPUs in computing and bandwidth capabilities, high-end mobile GPUs are expected to make this approach increasingly viable. On the other hand, low-end mobile devices may encounter challenges since computing and bandwidth significantly lag behind their high-end counterparts. The viability of running our approach on those devices depends on imposing significant constraints on both point buckets and memory buffers. Moreover, the method's implementation may require using the Vulkan API, making it incompatible with devices lacking Vulkan support.

Other pending tasks that could improve this work are data quantization, as explained in previous work [7], to reduce the number of data reads in GPU threads, and data compression [8]. In addition, one of the main bottlenecks of our approach is the large number of concurrent threads dispatched during the compaction stage whether many pointlets are visible and any of them have a substantial number of visible points. In this case, a significant portion of these threads stays idle, so calculating this number would have a meaningful impact. Finally, reducing the number of operations performed in the CPU would also lead to lower data exchanges and higher parallelism.

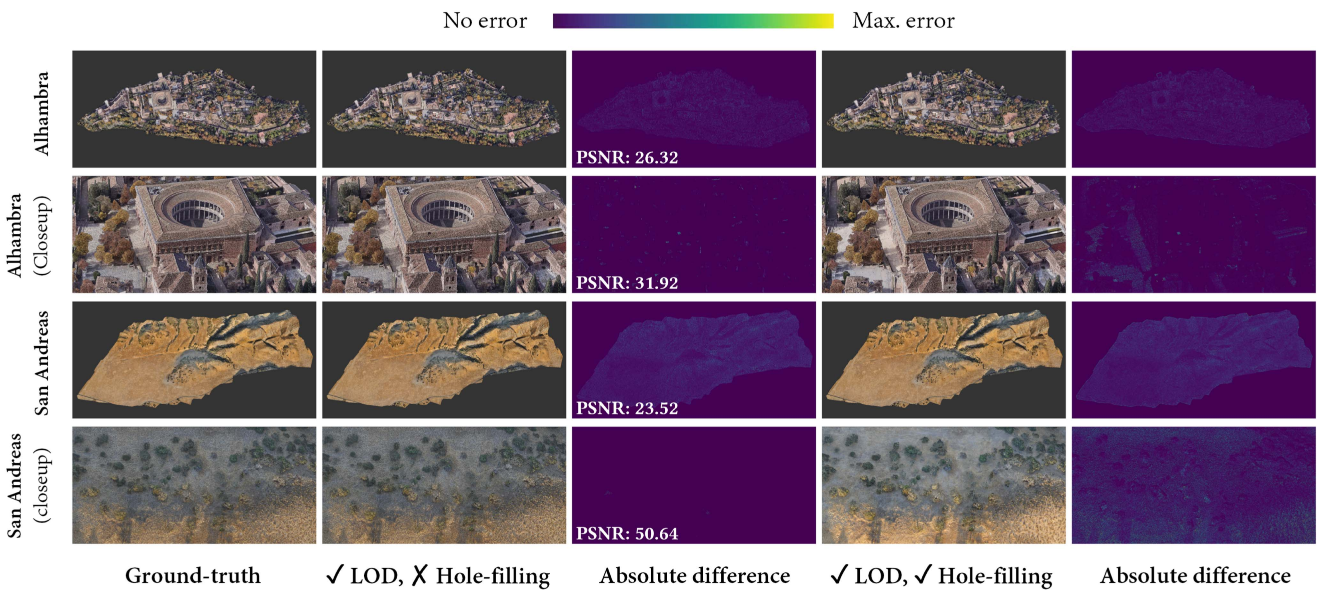

Fig. 16. Visual differences between the ground-truth image and images rendered using LoD and no hole-filling (second column) and enabling both components (fourth column). The last column does not display the PSNR as it also enhances the ground-truth image by filling gaps.

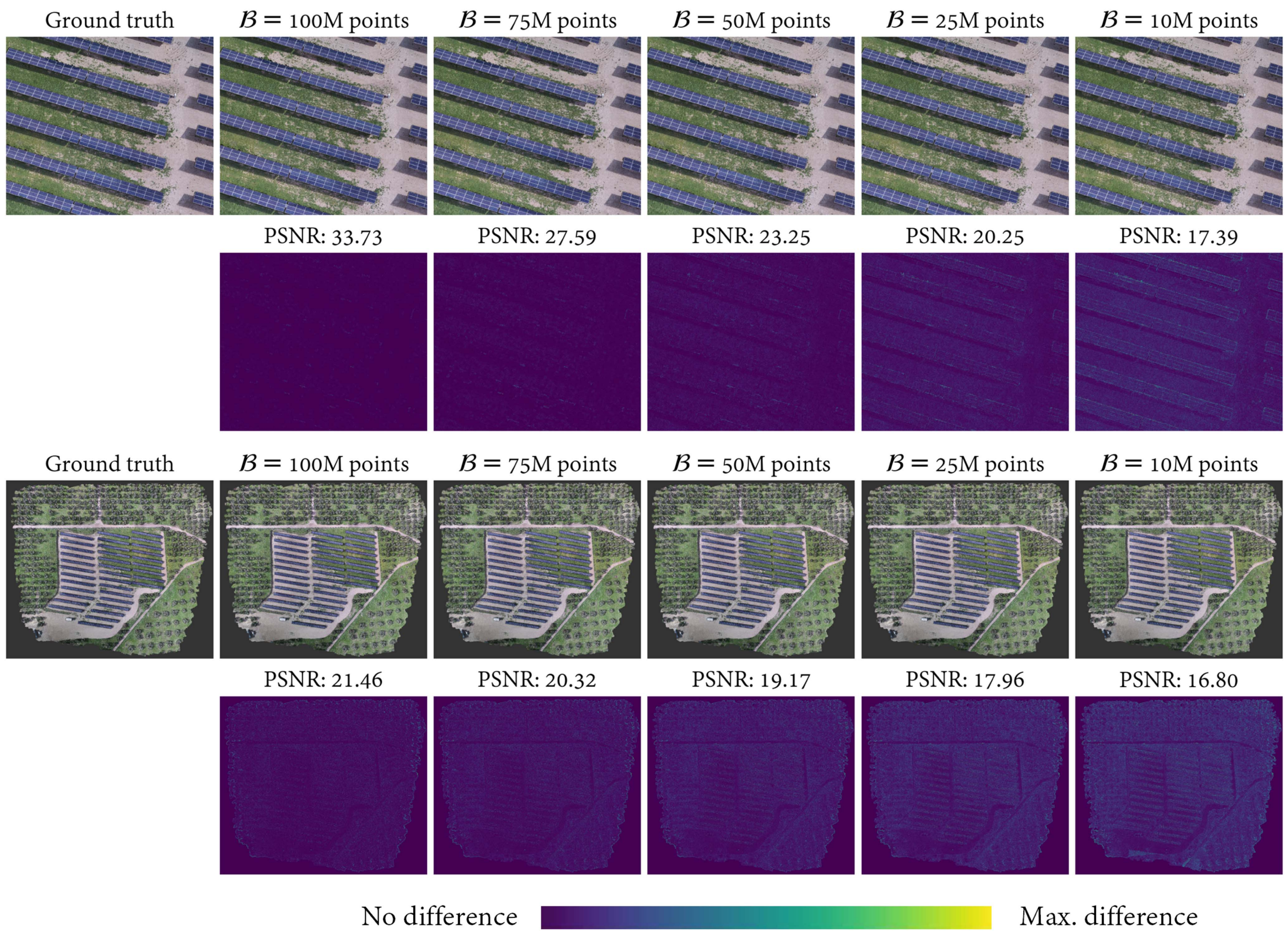
Ground truth
$\mathcal{B}$ = 100M points
$\mathcal{B}$ = 75M points
$\mathcal{B}$ = 50M points
$\mathcal{B}$ = 25M points
$\mathcal{B}$ = 10M points
PSNR: 33.73
PSNR: 27.59
PSNR: 23.25
PSNR: 20.25
PSNR: 17.39
Ground truth
$\mathcal{B}$ = 100M points
$\mathcal{B}$ = 75M points
$\mathcal{B}$ = 50M points
$\mathcal{B}$ = 25M points
$\mathcal{B}$ = 10M points
PSNR: 21.46
PSNR: 20.32
PSNR: 19.17
PSNR: 17.96
PSNR: 16.80
No difference
Max. difference

Fig. 17. Visual differences between ground-truth images and other images obtained by changing the bucket size. The error is calculated by summing the error of red, green and blue channels.

REFERENCES

[1] L. Tuggener, P. Sager, Y. Taoudi-Benchekroun, B. F. Grewe, and T. Stadelmann, "So you want your private LLM at home?: A survey and benchmark of methods for efficient GPTs," in *Proc. 11th IEEE Swiss Conf. Data Sci.*, ZHAW Zürcher Hochschule für Angewandte Wissenschaften, 2024, pp. 205–212. [Online]. Available: https://digitalcollection.zhaw.ch/handle/11475/30279

[2] M. Schütz, L. Herzberger, and M. Wimmer, "SimLOD: Simultaneous LOD generation and rendering," Oct. 2023, *arXiv:2310.03567*. [Online]. Available: http://arxiv.org/abs/2310.03567

[3] T. Kanzok, L. Linsen, and P. Rosenthal, "An interactive visualization system for huge architectural laser scans," in *Proc. 10th Int. Conf. Comput. Graph. Theory Appl.*, 2015, pp. 265–273.

[4] D. Deibe, M. Amor, and R. Doallo, "Supporting multi-resolution out-of-core rendering of massive LiDAR point clouds through non-redundant data structures," *Int. J. Geographical Inf. Sci.*, vol. 33, no. 3, pp. 593–617, Mar. 2019, doi: 10.1080/13658816.2018.1549734.

[5] C. Günther, T. Kanzok, L. Linsen, and P. Rosenthal, "A GPGPU-based pipeline for accelerated rendering of point clouds," *J. WSCG*, vol. 21, pp. 153–162, Jun. 2013.

[6] M. Schütz, K. Krösl, and M. Wimmer, "Real-time continuous level of detail rendering of point clouds," in *Proc. 2019 IEEE Conf. Virtual Reality 3D User Interfaces*, 2019, pp. 103–110. [Online]. Available: https://ieeexplore.ieee.org/document/8798284

[7] M. Schütz, B. Kerbl, and M. Wimmer, "Software rasterization of 2 billion points in real time," in *Proc. ACM Comput. Graph. Interactive Techn.*, vol. 5, no. 3, pp. 24:1–24:17, Jul. 2022, doi: 10.1145/3543863.

[8] R. Goel, M. Schütz, P. J. Narayanan, and B. Kerbl, "Real-time decompression and rasterization of massive point clouds," in *Proc. ACM Comput. Graph. Interact. Techn.*, vol. 7, no. 3, pp. 48:1–48:15, 2024, doi: 10.1145/3675373.

[9] M. Schütz, B. Kerbl, and M. Wimmer, "Rendering point clouds with compute shaders and vertex order optimization," *Comput. Graph. Forum*, vol. 40, no. 4, pp. 115–126, 2021, doi: 10.1111/cgf.14345.

[10] M. Kraemer, "GDC 2019: Asteroids mesh shading with DX12," Mar. 2019. [Online]. Available: https://developer.nvidia.com/video/GDC-19/MESH_SHADING

[11] C. Kubisch, "Introduction to turing mesh shaders," Sep. 2018. [Online]. Available: https://developer.nvidia.com/blog/introduction-turing-mesh-shaders/

[12] B. Karis, "A deep dive into nanite virtualized geometry," *SIGGRAPH Adv. Real-Time Rendering*, vol. 21, Oct. 2021. [Online]. Available: https://www.youtube.com/watch?v=eviSykqSUUw

[13] L. M. Ortega, J. C. Fernández, J. A. Collado, and J. F. R. Feito, *An Efficient Point Selection Process Over a Meshlet-Structured Point Cloud*. Eindhoven, Netherlands: The Eurographics Association, 2024, doi: 10.2312/ceig.20241144.

[14] M. Comino, C. Andújar, A. Chica, and P. Brunet, "Error-aware construction and rendering of multi-scan panoramas from massive point clouds," *Comput. Vis. Image Understanding*, vol. 157, pp. 43–54, Apr. 2017. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1077314216301461

[15] P. Dobrev, P. Rosenthal, and L. Linsen, "An image-space approach to interactive point cloud rendering including shadows and transparency," *Comput. Graph. Geometry*, no. 3, pp. 2–25, 2010. [Online]. Available: https://drive.google.com/file/d/1ub2SO6QKPCRplxmyiJeTBJWZMwh5FeMw/view

[16] R. Pintus, E. Gobbetti, and M. Agus, *Real-Time Rendering of Massive Unstructured Raw Point Clouds Using Screen-Space Operators*. Eindhoven, Netherlands: The Eurographics Association, 2011, doi: 10.2312/VAST.VAST11.105-112.

[17] A. Ribes and C. Boucheny, *Eye-Dome Lighting: A Non-Photorealistic Shading Technique*. Clifton Park, NY, USA: Kitware, Apr. 2011.

[18] O. Wegen, W. Scheibel, M. Trapp, R. Richter, and J. Dollner, "A survey on non-photorealistic rendering approaches for point cloud visualization," *IEEE Trans. Vis. Comput. Graph.*, to be published, doi: 10.1109/TVCG.2024.3402610. [Online]. Available: https://ieeexplore.ieee.org/document/10541081

[19] M. Botsch, A. Hornung, M. Zwicker, and L. Kobbelt, "High-quality surface splatting on today's GPUs," in *Proc. Eurographics/IEEE VGTC Symp. Point-Based Graph.*, 2005, pp. 17–141. [Online]. Available: https://ieeexplore.ieee.org/document/1500313

[20] J. Kordež, M. Marolt, and C. Bohak, "Real-time interpolated rendering of terrain point cloud data," *Sensors*, vol. 23, no. 1, pp. 72–88, Jan. 2023. [Online]. Available: https://www.mdpi.com/1424-8220/23/1/72

[21] B. Kerbl, G. Kopanas, T. Leimkühler, and G. Drettakis, "3D Gaussian splatting for real-time radiance field rendering," *ACM Trans. Graph.*, vol. 42, no. 4, Jul. 2023, Art. no. 139. [Online]. Available: https://repo-sam.inria.fr/fungraph/3d-gaussian-splatting/

[22] L. Franke, D. Rückert, L. Fink, M. Innmann, and M. Stamminger, "VET: Visual error tomography for point cloud completion and high-quality neural rendering," in *Proc. SIGGRAPH Asia Conf. Papers*, New York, NY, USA, 2023, pp. 1–12, doi: 10.1145/3610548.3618212.

[23] P. Y. Timokhin and M. V. Mikhaylyuk, "A method to order point clouds for visualization on the ray tracing pipeline," *Program. Comput. Softw.*, vol. 50, no. 3, pp. 264–272, Jun. 2024, doi: 10.1134/S0361768824700075.

[24] M. Schütz, B. Kerbl, P. Klaus, and M. Wimmer, "GPU-Accelerated LOD generation for point clouds," *Comput. Graph. Forum*, vol. 42, no. 8, 2023, Art. no. e14877, doi: 10.1111/cgf.14877.

[25] C. Scheiblauer and M. Wimmer, "Analysis of interactive editing operations for out-of-core point-cloud hierarchies," in *Proc. WSCG Full Paper*, 2013, pp. 123–132. [Online]. Available: https://repositum.tuwien.at/handle/20.500.12708/54731

[26] J. Martens and J. Blankenbach, "A decomposition scheme for continuous level of detail, streaming and lossy compression of unordered point clouds," *Graphical Models*, vol. 130, Dec. 2023, Art. no. 101208. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1524070323000383

[27] M. Wand et al., "Processing and interactive editing of huge point clouds from 3D scanners," *Comput. Graph.*, vol. 32, no. 2, pp. 204–220, 2008.

[28] C. J. Ogayar-Anguita, A. López-Ruiz, A. J. Rueda-Ruiz, and R. J. Segura-Sánchez, "Nested spatial data structures for optimal indexing of LiDAR data," *ISPRS J. Photogrammetry Remote Sens.*, vol. 195, pp. 287–297, Jan. 2023. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0924271622003112

[29] C. Dachsbacher, C. Vogelgsang, and M. Stamminger, "Sequential point trees," *ACM Trans. Graph.*, vol. 22, no. 3, pp. 657–662, Jul. 2003, doi: 10.1145/882262.882321.

[30] P. van Oosterom, S. van Oosterom, H. Liu, R. Thompson, M. Meijers, and E. Verbree, "Organizing and visualizing point clouds with continuous levels of detail," *ISPRS J. Photogrammetry Remote Sens.*, vol. 194, pp. 119–131, Dec. 2022. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0924271622002647

[31] J. Otepka, G. Mandlburger, M. Schütz, N. Pfeifer, and M. Wimmer, "Efficient loading and visualization of massive feature-rich point clouds without hierarchical acceleration structures," *Int. Arch. Photogrammetry Remote Sens. Spatial Inf. Sci.*, vol. 43, pp. 293–300, 2020. [Online]. Available: https://isprs-archives.copernicus.org/articles/XLIII-B2-2020/293/2020/

[32] P. van Oosterom et al., "Massive point cloud data management: Design, implementation and execution of a point cloud benchmark," *Comput. Graph.*, vol. 49, pp. 92–125, Jun. 2015. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0097849315000084

[33] W. Chen, X. Zhu, G. Chen, and B. Yu, "Efficient point cloud analysis using hilbert curve," in *Proc. Eur. Conf. Comput. Vis.*, Cham, Switzerland: Springer Nature, 2022, pp. 730–747.

[34] Y. Imamura, T. Shinohara, K. Hirata, and T. Kuboyama, "Fast hilbert sort algorithm without using Hilbert indices," in *Proc. Int. Conf. Similarity Search Appl.*, Cham, Switzerland: Springer International Publishing, 2016, pp. 259–267.

[35] M. Bader, *Space-Filling Curves: An Introduction With Applications in Scientific Computing*. Berlin, Germany: Springer, Aug. 2016.

[36] J. A. Béjar-Martos, A. J. Rueda-Ruiz, C. J. Ogayar-Anguita, R. J. Segura-Sánchez, and A. López-Ruiz, "Strategies for the storage of large LiDAR datasets–A performance comparison," *Remote Sens.*, vol. 14, no. 11, pp. 36–50, Jan. 2022. [Online]. Available: https://www.mdpi.com/2072-4292/14/11/2623

[37] O. Mures, A. Jaspe-Villanueva, E. Padrón, and J. Rabuñal, "Virtual reality and point-based rendering in architecture and heritage," in *Virtual and Augmented Reality: Concepts, Methodologies, Tools, and Applications*, Hershey, PA, USA: IGI Global, Apr. 2016, pp. 549–565.

[38] M. Arikan, R. Preiner, C. Scheiblauer, S. Jeschke, and M. Wimmer, "Large-scale point-cloud visualization through localized textured surface reconstruction," *IEEE Trans. Vis. Comput. Graph.*, vol. 20, no. 9, pp. 1280–1292, Sep. 2014. [Online]. Available: https://ieeexplore.ieee.org/document/6774475

[39] H. Haverkort, "An inventory of three-dimensional Hilbert space-filling curves," Oct. 2016, *arXiv:1109.2323*. [Online]. Available: http://arxiv.org/abs/1109.2323
[40] J. K. Lawder and P. J. H. King, "Using space-filling curves for multi-dimensional indexing," in *Proc. Brit. Nat. Conf. Databases*, Berlin, Heidelberg: Springer, 2000, pp. 20–35.
[41] rawrunprotected, "LUT-based 3D Hilbert curves," May 2020. [Online]. Available: https://threadlocalmutex.com/
[42] G. Leese, J. Kessenich, D. Baldwin, and R. Rost, "The OpenGL shading language, version 4.60.8," Aug. 2023. [Online]. Available: https://registry.khronos.org/OpenGL/specs/gl/GLSLangSpec.4.60.html
[43] M. Bunds et al., "High resolution topography of the central San Andreas Fault, California," 2020. [Online]. Available: https://opentopography.org/meta/OT.082020.32610.1
[44] OpenTopography, "LiDAR survey of dangermond preserve, CA," 2022. [Online]. Available: https://opentopography.org/meta/OT.042022.32611.1
[45] Pacific Gas and Electric Company (PG&E), "PG&E diablo canyon power plant (DCPP): San Simeon and Cambria Faults, CA," 2013. [Online]. Available: https://opentopography.org/meta/OT.032013.26910.2

José Antonio Collado received the MSc degree in computer science from the University of Jaén in 2024. He is currently working toward the PhD degree with the University of Jaén. Currently, he is a member of the Graphics and Geomatics research group. His current work is focused on high-performance real-time visualization of huge 3D data volumes, such as point clouds. His main research interests are graphics engines, rendering techniques, real-time rendering and GPGPU.

Alfonso López received the BSc and MSc degrees in computer science from the University of Jaén, in 2019 and 2020, respectively, and the PhD degree in computer science from the University of Jaén, in 2023. He was granted a Juan de la Cierva Postdoctoral Fellowship, in 2024 to conduct his research with the University of Zaragoza. His interests span GPGPU, parallel computing, real-time rendering, photogrammetry, image processing and geometric algorithms.

Juan Manuel Jurado received the MS degree in computer science, and the award of the best master's thesis from the University of Jaén, in 2017, the MS degree in high-performance computing from the University of A Coruña, in 2020, and the PhD degree in computer science from the University of Jaén. He is an associate professor with the Department of Computer Science, University of Jaén, Spain. His research interests include modeling, processing, and rendering 3D real-world scenarios, with a focus on multi-sensorial data fusion and observation of time-varying phenomena. His recent work has focused on developing efficient geometric representations of natural and urban environments, encoding remote-sensing data, and simulating their physical behavior.

Juan Roberto Jiménez received the master's degree in computer science from the University of Granada, in 1998. He is an associate professor with the Department of Computer Science, University of Jaén. He is a member of the Graphics and Geomatic research group and the Center for Advanced Studies in Information and Communication Technologies of the University of Jaén. In 1999, he was granted a scholarship from the Ministry of Education and Science to develop a doctoral thesis with the University of Girona. In 2000, he was hired by the University of Jaén in the Computer Science Department. In 2010, he read his doctoral thesis with the European mention with the Polytechnic University of Catalonia under the supervision of Xavier Pueyo (University of Girona) and Karol Myszkowski (Max Planck Institute for Informatics – Germany). His research focuses on computer graphics techniques, GPU programming, simulation and virtual and augmented reality. He has more than 30 international publications in journals, chapters of books and conferences. He is currently in charge of a Spanish national research project. Currently, he is a member of the board of the Spanish Section of the Eurographics Scientific Society.