TU WIEN Informatics

Interactive Exploration of Point Clouds

DISSERTATION

submitted in partial fulfillment of the requirements for the degree of

Doktor der Technischen Wissenschaften

by

Dipl.-Ing. Markus Schütz
Registration Number 00825723

to the Faculty of Informatics

at the TU Wien

Advisor: Univ.Prof. Dipl.-Ing. Dipl.-Ing. Dr.techn. Michael Wimmer

The dissertation has been reviewed by:

Mario Botsch	Carsten Dachsbacher

Vienna, 8th March, 2021

Markus Schütz

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Erklärung zur Verfassung der Arbeit

Dipl.-Ing. Markus Schütz

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 8. März 2021

Markus Schütz

Acknowledgements

I wish to thank everyone that contributed to this thesis one way or another, through discussions, funding, support and knowledge. First and foremost to my supervisor Michael Wimmer, who made it possible for me to research this subject at TU Wien, and whose mentorship and help allowed me to publish the results in prestigious conferences and journals. I would also like to thank Katharina Krösl for fruitful discussions about virtual reality, Bernhard Steiner for his discussions about real-time rendering, and Johannes Unterguggenberger for suffering my frequent visits to his office. Thanks to Gottfried Mandlburger and Johannes Otepka, who initiated the work on one of the papers by approaching me with a specific problem that I had not thought of, but which presented a common issue/condition in their field.

Thanks to everyone who contributed data sets that allowed us to work on, test and illustrate the methods presented in our research. In particular, the *Ludwig Boltzmann Institute for Archaeological Prospection and Virtual Archaeology (LBI ArchPro)* for providing the Heidentor data set; *Riegl Laser Measurement Systems* for providing data sets of Vienna and Retz; *PG&E and Open Topography* for providing a point cloud of the San Simeon coastal area (CA13); the *TU Wien, Institute of History of Art, Building Archaeology and Restoration* for providing data sets of historical buildings such as Kirchenburg Arbegen, Candi Sari, Candi Banyunibo, and Museum Affandi; *Pix4D* for providing data sets of a quarry (Eclepens) and the Matterhorn; *the Netherlands* for providing data sets of their entire country (AHN2, AHN3), *NVIDIA* for providing photogrammetry and laser scans of the construction site of their new headquarters (Endeavor data set), the *Stanford University Computer Graphics Laboratory* for providing the popular Stanford Bunny data set, and to *Keenan Crane* for providing several 3D models for research, specifically the Spot data set.

Thanks to the funding by different agencies and companies that made it possible to invest the time into research and development of the methods in this thesis. In particular, the GCD doctoral college program by TU Wien[1] that funded the majority of the PhD, the FWF Austrian Science Fund through project "Superhumans"(ID: FWF P32418-N31), and FFG through project "LargeClouds2BIM" (ID: FFG 880846). Additional funding for the fast octree generation was provided by SITN (République et canton de Neuchâtel) and Geodelta (Delft).

[1] http://gcd.tuwien.ac.at/

Kurzfassung

Laser Scanning, Photogrammetrie und andere 3D-Aufnahmeverfahren erzeugen Datensätze, die aus Millionen, Milliarden, bis hin zu Billionen von Punkten bestehen. Moderne high-end GPUs sind zwar in der Lage Datensätze im zweistelligen Millionenbereich in Echtzeit darzustellen, für hunderte Millionen Punkte oder mehr sind allerdings hierarchische Beschleunigungsstrukturen notwendig. In dieser Dissertation stellen wir neu entwickelte Methoden vor, die das Ziel haben große Punktwolkendatensätze in besserer Qualität und in Echzeit darzustellen.

Zwei dieser Methoden behandeln die Geschwindigkeit der Generierung von Strukturen zum Darstellen in unterschiedlichen Detailgraden (Level of Detail, LOD). Existierende Methoden können LOD-Strukturen mit einer Geschwindigkeit von bis zu etwa einer Million Punkte pro Sekunde erstellen, was bei hunderten Millionen Punkten zu Wartezeiten von einigen Minuten führt. Unsere Ansätze um diese Wartezeiten zu verkürzen sind einerseits eine Methode, mit der LOD-Stukturen schneller generiert werden können, andererseits eine Methode, mit der jede Punktwolke, die in den GPU Speicher passt, auch ohne LOD-Struktur in Echtzeit dargestellt werden kann. Ersteres generiert LOD-Strukturen mit einer Geschwindigkeit von bis zu 10 Millionen Punkte pro Sekunde, und letzteres kann Punktwolken mit Geschwindigkeiten von 37 Millionen bis 100 Millionen Punkten pro Sekunde von der Festplatte laden und direkt in Echtzeit darstellen noch während weitere Punkte geladen werden.

Unser dritter Beitrag zum State-of-the-Art dient der Verbesserung der Bildqualität durch die Verwendung von kontinuierlichen statt diskreten Übergängen zwischen Detailstufen. Diskrete LOD-Methoden leiden an auffälligen Treppenartefakten an den Übergängen von einem LOD zum nächsten. Die stufenartige Reduktion der Punktdichte ist unschön zu betrachten und führt während der Navigation zu störenden Popping-Artefakten wenn Punkte blockweise ein- und ausgeblendet werden. Unsere kontinuierliche LOD-Methode sorgt für einen flüssigen Übergang von einem LOD zum nächsten und vermeidet Popping-Artefakte durch punktweises statt blockweises ein- und ausblenden der Daten.

Abstract

Laser scanning, photogrammetry and other 3D scanning approaches generate data sets comprising millions to trillions of points. Modern GPUs can easily render a few million and up to tens of millions of points in real time, but data sets with hundreds of millions of points and more require acceleration structures to be rendered in real time. In this thesis, we present three contributions to the state of the art with the goal of improving the performance as well as the quality of real-time rendered point clouds.

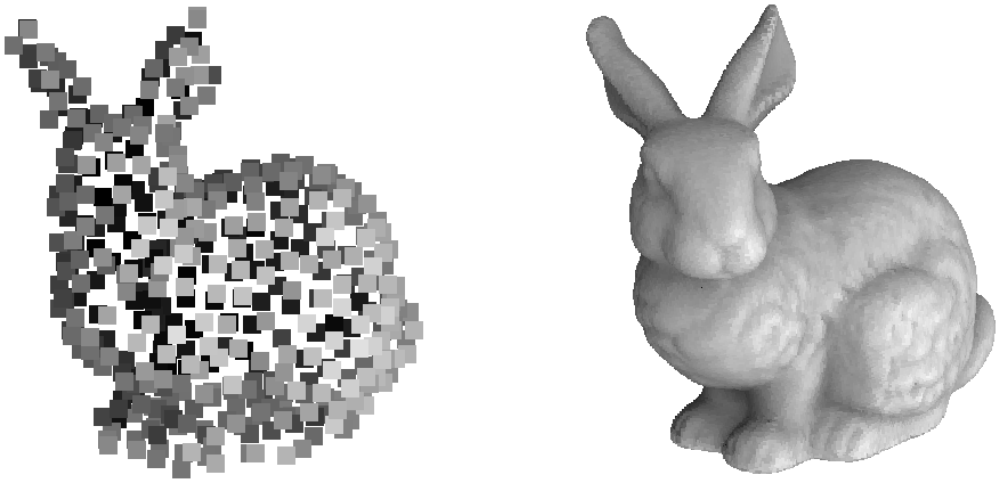
Two of our contributions address the performance of LOD structure generation. State-of-the-art approaches achieve a throughput of up to around 1 million points per second, which requires users to wait minutes even for smaller data sets with a few hundred million points. Our proposed solutions are: A bottom-up LOD generation approach that creates LOD structures up to an order of magnitude faster than previous work, and a progressive rendering approach that is capable of rendering any point cloud that fits in GPU memory in real time, without the need to generate LOD structures at all. The former achieves a throughput of up to 10 million points per second, and the latter is capable of loading point clouds at rates of up to 37 million points per second from an industry-standard point-cloud format (LAS), and up to 100 million points per second if the file format matches the vertex buffer format. Since it does not need LOD structures, the progressive rendering approach can render already loaded points right away while additional points are still being loaded.

Our third contribution improves the quality of LOD-based point-cloud rendering by introducing a continuous level-of-detail approach that produces gradual transitions in point density, rather than the characteristic and noticeable blocks from discrete LOD structures. It is mainly targeted towards VR applications, where discrete levels of detail are especially noticeable and disturbing, in a large part due to the popping of chunks of points during motion.

Contents

CHAPTER 1

Introduction



(a) Hundreds of points. (b) Millions of points.

Figure 1.1: Point clouds are 3D models made of unconnected points. The impression of a closed surface model can be achieved through a sufficiently large point density.

Points were already proposed as a rendering primitive by Levoy and Whitted [LW85] back in 1985, but one of the main reasons that point-cloud rendering became popular is the 3D scanning of real-world locations and objects. Time-of-flight laser scanners, for example, observe the real world by measuring distances from the scanner to pointed locations, and then using the known orientation of the laser to transform the distance values into 3D point coordinates. Photogrammetry, on the other hand, uses multiple overlapping images to construct a set of points that fits the images when viewed from the respective viewpoints.

Although not as commonly encountered as mesh-based models, point clouds make up a massive amount of data. The Netherlands periodically scan the entire country, and their second operation, labeled AHN2[1], resulted in 640 billion points – roughly 1.6 TB compressed data [MRVvM+15]. Similarly, the United States are currently scanning the entire nation under the 3D Elevation Program (3DEP)[2] [USGa]. As of today, 21 trillion points are available for viewing in web browsers[3] [3DE, USGb].

Use cases of point clouds include cultural heritage [SAMGA+20, HSG+19, SZW09, SBV+13, PVM+20], disaster management and response [CCD+21, GFP+18], surveying and mapping [BCFB19, OPH+10, SMSV10, HHF+14, SRS+19], monitoring [GGS08, KLZ+18], robotics [SPF+17, OKWM12], bathymetry [DMM+15, HR08], constructing mesh models out of scan data [ASF+13, EGO+20], and more. All of the listed use cases have in common that they use 3D scanners (laser scanners, photogrammetry, etc.) to create 3D point cloud models of the real world, but the motives to create these models vary. Point clouds for cultural heritage projects typically involve scanning old buildings and structures for the purpose of documenting, studying, and sharing interactive 3D models with an interested audience. For disaster management and response, 3D models are used to aid in preventing, preparing for, or dealing with emergency situations. Surveying involves measuring real-world objects and land dimensions for purposes such as construction planning and maintaining cadastres. Mapping is the task of creating 2D or 3D maps. In robotics, point clouds may be captured to enable the robot to analyze its surroundings and to safely navigate the area. This may also involve doing simultaneous localization and mapping (SLAM), which is the task of mapping an area and at the same time keeping track of your own location within the newly mapped area. Bathymetry refers to the mapping of underwater terrain and requires special scanning techniques, since 3D scanning techniques for land are typically not directly applicable in water due to its different optical properties. Lastly, (mesh-)reconstruction refers to the generation of 3D mesh models out of an array of unconnected points. Mesh models are often advantageous because they offer higher visual quality, are faster to render, and have no holes between adjacent surface samples. Downsides are that reconstructed meshes constitute made-up data that may not necessarily be correct (e.g., holes that exist in reality may have been closed), they may have lost real high-frequency features that are

[1] https://www.pdok.nl/downloads/-/article/actueel-hoogtebestand-nederland-ahn2-

[2] https://www.usgs.gov/core-science-systems/ngp/3dep

[3] https://usgs.entwine.io/

misinterpreted as outliers or noise (e.g., wires, grass, leaves), or simplified into low-poly blobs (e.g., a tree canopy comprising of individual leaves gets reinterpreted as a spherical blob), and that reconstruction can take a long time. Whether the original point cloud data or reconstructed meshes are preferred varies from use case to use case. In this thesis, we focus on real-time rendering techniques for point cloud data sets in order to provide methods and tools to aid those who need to work with large point-based models.

1.1 Motivation

The already enormous, yet still constantly increasing, storage requirements of point-cloud data sets, coupled with the need to visualize and analyze them, makes point-cloud processing and rendering an ongoing and relevant research topic. Some important topics are noise removal, reconstruction (generating meshes), computing digital elevation models (DEM)[4] or digital surface models (DSM)[5], the rendering of arbitrarily large point clouds, and the high-quality rendering of these data sets. Our thesis focuses on the latter two topics, the rendering of arbitrarily large point clouds and the high-quality rendering of point clouds. Being able to navigate through data sets like AHN2 (640 billion) and 3DEP (21 trillion) in real time and in 3D provides insights that could not be obtained from static renderings or derivatives like 2D maps. Real-time rendering allows users to inspect and analyze (e.g., quality, noise, measurements, elevation profiles, filtering) the captured data with instant feedback.

A secondary motivation of our thesis is to develop methods that also work on mid-range hardware as well as computationally demanding platforms such as virtual reality devices. The difficulty of virtual reality is that it has significantly higher requirements on frame rate and quality, ultimately resulting in an around 3 to 5 times higher computational effort. Most desktop monitors have a frame rate of 60 hz, whereas VR devices range from 72 hz (Oculus Quest), 90 hz (Oculus Rift CV1, HTC Vive) to 120 hz (Valve Index). On top of that, VR devices require to render the scene twice, once for the left and once for the right eye. The need for higher rendering quality further increases the computational effort because aliasing and other rendering artifacts are even more noticeable in VR compared to viewing the same result on a desktop monitor. Rendering in VR and rendering on mid-range hardware are related goals in that both require methods that provide renderings with acceptable quality, limited by a reduced amount of computational power.

Another goal is to deliver real-time rendered point clouds on web-based platforms, since web browsers are the easiest and most effective way to share data sets with a large audience. Our previous work, Potree [6], is already capable of and used to stream and render data sets with hundreds of billions of points in real time in web browsers, but it suffers from various limitations. For example, it is based on WebGL, which does not

[4] 3D model of the terrain, excluding plants, bridges, buildings
[5] 3D model of the terrain and all objects on top of it
[6] `http://potree.org/`

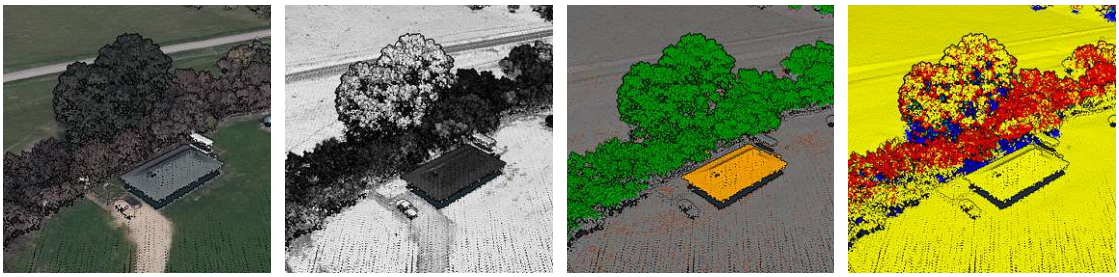
support compute shaders and other features of modern graphics APIs. Due to this, this thesis indirectly addresses only one aspect of web-based rendering that does not require compute shaders: The fast generation of LOD structures. However, with WebGPU, the upcoming successor to WebGL, other parts of this thesis can also be implemented in web browsers in the future.

1.2 Background

In this chapter, we describe some properties of point-cloud data sets, how point clouds are rendered in real time, and differences between point clouds and other types of 3D models.

1.2.1 Point-Cloud Data

A point cloud is a set of points with no connectivity and no textures (generally speaking – there are exceptions). Each point consists of a coordinate value and a set of attributes such as color, intensity, classification, return number, normals, etc. Some point clouds may have no attributes besides position, others might contain more than 50 attributes, roughly equivalent to 100 to 200 bytes per point. Figure 1.2 shows various potential attributes.



(a) RGB (b) Intensity (c) Classification (d) Return Number

Figure 1.2: Various attributes of a point cloud. Figure taken from [Sch16].

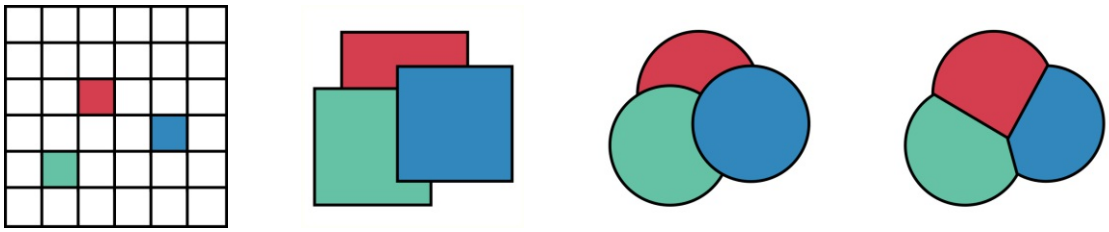
Some potentially available point attributes are:

- **RGB**: Available by default in most photogrammetry data sets since photogrammetry reconstructs geometry from images. Laser scans require extra effort, e.g., by additionally capturing photos with a camera mounted on the scanner, or by projecting otherwise available image data like satellite maps.
- **Intensity**: The strength of the reflected laser signal. The intensity correlates with the reflectance of a material but is also affected by additional factors such as distance between scanner and point, atmospheric conditions, etc.

- **Reflectance**: The reflectance of the surface material. Can be computed from intensity [HP07].
- **Classification**: Describes the type of object that this point belongs to, e.g., vegetation, water, building, ground, noise, etc.
- **Gps-Time**: The moment in time at which this point was acquired.
- **Number of Returns and Return Number**: Specifies how many points a single laser beam returned, and the return number of the current point. Relevant for aerial LIDAR, which can penetrate through tree canopies all the way to the ground of a forest. Some part of the laser signal might be reflected from one leaf (1st return), another part from the next leaf (2nd return), and the last part makes it through the gaps between leaves all the way to the ground (last return). Generally, if you can see the sky from the ground of a forest, a focused laser beam also has a chance to observe the ground out of a plane.
- **Normals**: While normals are a regular part of triangle models, they are not common in point clouds since points are not surface elements per se. Point clouds obtained from scans do not contain normals by default since laser scanners only capture the location of a surface but not its orientation. If normals are required, they have to be computed afterwards based on the neighbourhood of points.

1.2.2 Point-Cloud Rendering

Graphics APIs such as OpenGL, WebGL, DirectX, Vulkan and WebGPU natively offer points as a rendering primitive – named GL_POINTS, gl.POINTS, D3DPT_POINTLIST, VK_PRIMITIVE_TOPOLOGY_POINT_LIST, and "point-list", respectively. All of them support rendering points with a size of a single pixel, while OpenGL and WebGL also support rendering points as quads whose size is specified via the shader variable gl_PointSize. DirectX and WebGPU, on the other hand, only support point sizes of one pixel. If larger point sizes are required, developers need to implement their own ways of increasing the size, e.g., by transforming points to quads via geometry shaders or instancing a quad at each point position. In addition to pixels or quads, points may also be rendered as spheres, camera-facing cones or paraboloids, or oriented disks - each with their own advantages and disadvantages. Figure 1.3 illustrates some potential rendering primitives for points. Rendering points as single pixels is fast but usually tends to result in holes in the surface. Other primitives may be able to fully cover the holes but if they are too large, the surface samples will occlude each other, resulting in a negative impact on quality and performance. High-quality rendering methods, such as surface splats [ZPVBG01, PJW12, BHZK05], do not suffer from self-occlusion of overlapping points, in fact, they require a certain amount of overlaps and then blend fragments together. However, this has a performance impact because of the required overdraw and additional render passes.



(a) Pixels (b) Quads (c) Circles (d) Screen-aligned Paraboloids

Figure 1.3: Various shapes that can be used to display points.

Alternatively to the standard rendering pipeline, GPGPU approaches (OpenCL or OpenGL compute shaders) have also shown the potential to render large point clouds up to 10 times faster, e.g., through busy loops that write points to pixels once they acquire a lock [GKLR13], or compute shaders that encode depth and color values into a 64bit integer, and then store the closest point in an output buffer using atomicMin [SW19]. However, these GPGPU approaches only explore rendering performance with a size of a single pixel per point. The rasterizer of the standard rendering pipeline likely still scales better to larger point sizes, but exhaustive benchmarks are subject to future research.

For further point-based rendering techniques, we would like to refer to the survey by Kobbelt and Botsch [KB04] and the fourth edition of *"Real-Time Rendering"* [AMHH19].

Eye-Dome-Lighting (EDL)

Most point clouds usually do not have normals, which are required for illumination models. Without illumination, renderings may appear flat and it becomes hard or impossible to understand the geometry of a model. Eye-Dome-Lighting (EDL) [Bou09] is a form of illumination model that computes shading from the depth map without the need for normal vectors, which makes it especially useful for point-cloud rendering. It essentially operates like an edge detection filter, i.e., it analyzes the differences in depth of the neighborhood of each pixel in order to compute the shading. Small differences in depth indicate a surface that points directly towards the camera, medium differences indicate a surface that points slightly off the camera, and very large differences indicate silhouettes. The differences in depth are then mapped to a shading value that gives the scene the look of being illuminated by a light that is attached to the camera, and also the characteristic edges along silhouettes. Figure 1.4 illustrates the EDL render passes, and Figure 1.5 shows an example of the results.

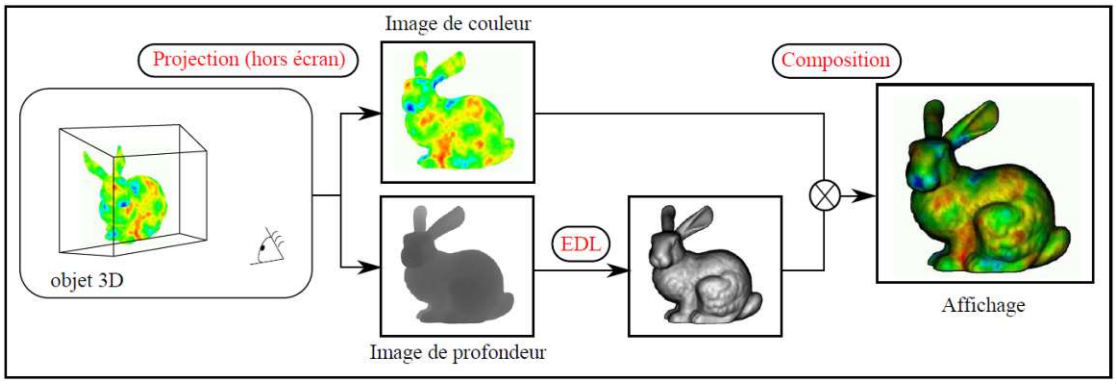
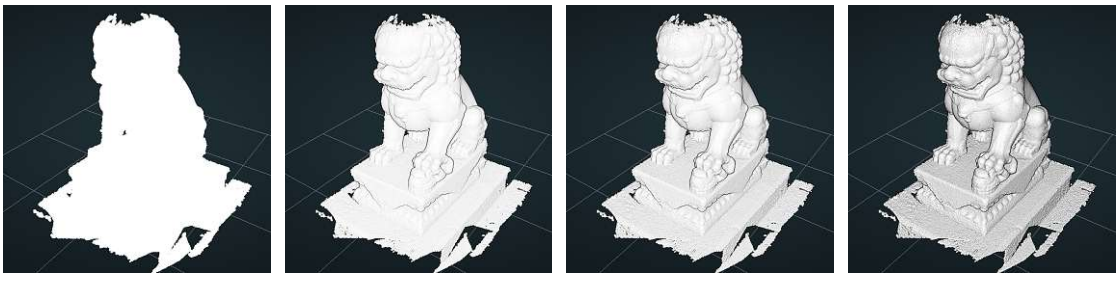


Figure 1.4: EDL render passes. First, depth map and colors are rendered. Then, shading is computed from the depth map and finally, shading and colors are combined. Figure taken from [Bou09].

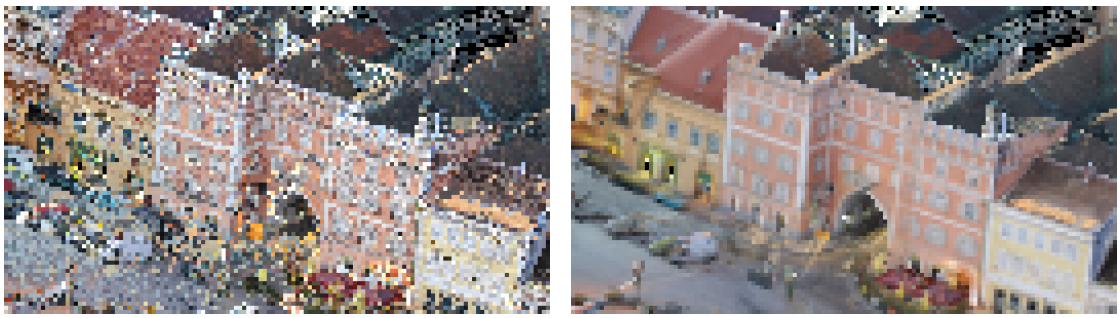


(a) Strength 0 (b) Strength 0.2 (c) Strength 0.5 (d) Strength 1

Figure 1.5: Eye-Dome-Lighting of a point cloud with neither normals nor colors. Figure taken from [Sch16]

High-Quality Point-Cloud Rendering

Basic point-cloud rendering approaches suffer from self-occluding surface samples and, as a consequence, aliasing artifacts, as shown in Figure 1.6. Adjacent points that belong to the same front-most surface occlude each other even though all of them should contribute to the image. When zoomed out, hundreds of points may be projected to a single pixel, but only the closest point will be displayed. As a result, rendered images have a grainy/noisy look to them and the whole image sparkles/flickers during motion.



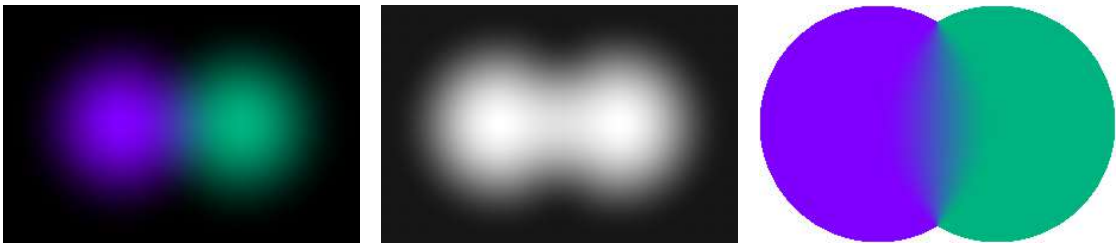
(a) Aliasing artifacts (b) High-quality splatting

Figure 1.6: (a) Standard point-cloud rendering displaying only the nearest fragment. (b) High-quality splatting, which blends overlapping fragments together [BHZK05]. Figure taken from [SW19].

These types of aliasing artifacts are strongly related to rendering textured meshes without mip mapping and the solutions follow similar ideas. Mip mapping addresses the problem of when the bounding box of a pixel projects to a large area of a texture by precomputing a blend of these large portions of the texture. The advantages are twofold - fewer aliasing artifacts and faster performance due to improved memory access. Unfortunately, mip mapping is not directly applicable to point-cloud rendering since points are rendered with vertex colors instead of textures.

Elliptical weighted average (EWA) splatting [ZPvG02] and its variations address the same problem as mip mapping does, but the execution is very different. The goal is still the same – all surface samples (texels, points) projected to a pixel should contribute to the result – but EWA filtering approaches achieve this by recomputing a blend of overlapping surface samples (points) in each frame. A GPU-friendly implementation of EWA filtering [BHZK05] implements three passes: A visibility pass that computes a shifted depth map, an attribute pass that computes a weighted sum of attributes and a sum of weights, and a normalization pass that divides the weighted sum by the sum of weights. The shifted depth map ensures that only fragments within a certain range from the closest fragment pass the depth test. The attribute pass uses additive blending and a floating-point target buffer to sum up the weighted attribute values and the weights themselves. Figure 1.7 shows the result of the attribute and normalization passes. This high-quality splatting method results in anti-aliasing for small points by computing

average color values of points in a pixel, and it also results in smooth transitions between overlapping points that are large.



(a) Weighted attributes (b) Sum of weights (c) Normalization

Figure 1.7: Figure taken from [Sch16]

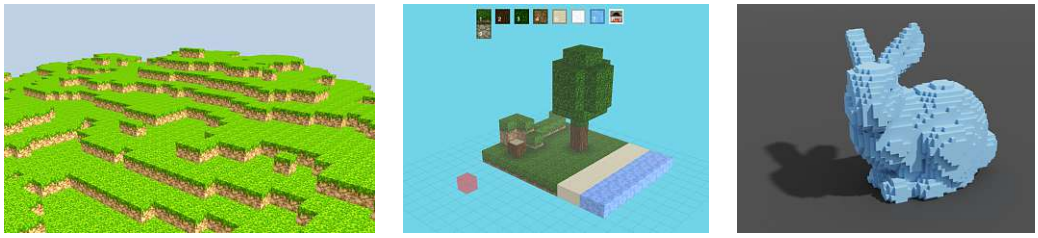
Another option for anti-aliasing that more closely resembles the hierarchical nature of mip mapping is to build a hierarchical structure and store points with average values in lower levels of detail [WBB+07]. However, the majority of state-of-the-art point-cloud renderers use level-of-detail structures that pick samples from the original data set and promote them to lower levels of detail, rather than computing averages. We suspect that the main reasons for this are performance, ease of implementation, and memory requirements compared to approaches that use averages. However, we also expect a shift to LOD structures that use precomputed averages in the future, as the quality of the renderings is becoming more important.

1.2.3 Differences to Other Rendering Structures

This section discusses some similarities and differences to related rendering structures and methods, specifically voxels, molecule rendering, and particle systems. For example, atoms in molecule data sets could also be interpreted as point clouds or particle systems, but there are some specific requirements on visualization methods to make these data sets intelligible.

Point Clouds vs. Voxels

Voxels are a three-dimensional analogue to pixels and are mainly used to represent volume data. Basic voxel data sets consist of a three-dimensional grid with a certain resolution, e.g., 512^3, where each grid cell represents the state and properties of a voxel. In the game Minecraft, voxel cells can be empty or depict various materials such as grass, stone, water, etc. In its most basic form, a voxel describes whether a specific cell represents the inside or the outside of a 3D model. In Figure 1.8c, all cells that represent the inside of the Stanford bunny are visualized by a box.



(a) Three.js Voxel Demo (b) Voxel Painter (c) Stanford Bunny

Figure 1.8: Basic voxel models.

A major difference between point clouds and voxels is that the former explicitly specify the coordinates of vertices, whereas the latter implicitly specify the coordinates of samples through their position in a grid array. Voxels are therefore usually better suited to volumetric data sets where a large part of the bounding box is occupied by samples because in that case, it becomes cheaper to infer the coordinate of a sample by its location in the grid array. The disadvantage, however, is that inferring the coordinate from a grid array reduces the coordinate precision to the resolution of the grid, which gives voxels their characteristic box look. Point clouds, on the other hand, are more suitable to data sets that occupy a small part of a given bounding volume. In this case, it is cheaper to explicitly store the coordinate of each point, which also increases the coordinate precision of each sample to the respective numeric primitive (int32, float, double, ...) or some form of quantized or compressed coordinate representation [BWK02, DD04].

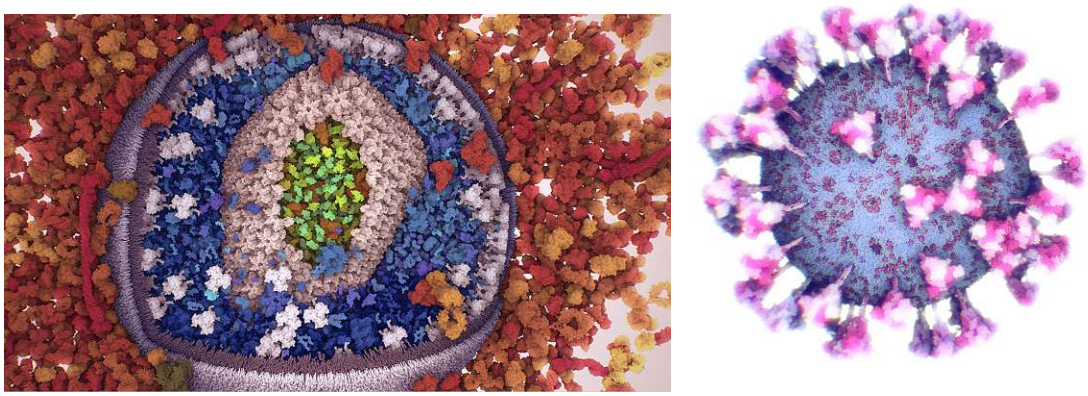
Voxel data sets can be rendered with various approaches, including rendering each voxel as a box, creating a mesh model out of a voxel data set (surface reconstruction), or directly rendering the voxel volume through raycasting. Marching cubes [LC87], for example, is an algorithm that creates a partially smoothed surface out of a voxel grid. Its original purpose was to create surface meshes out of medical data that consist of scalar voxel fields, where voxels don't just represent a binary inside/outside state but rather the density of a material, e.g., bones and muscles. The user can specify a certain density value and the marching cubes algorithm will proceed to build a surface model between voxels of lower and higher density. This way, users can choose to build a surface that encapsulates the whole human, i.e., any voxel denser than air is considered *inside*, or only the bones, i.e., any voxel as dense as bone is considered inside, but muscles, air, etc. are not.

Direct volume rendering, as the name implies, renders the voxel volume directly without producing derivatives such as surface meshes. This is mainly done by casting rays from pixels into the voxel volume, and then evaluating voxel values along the ray [KW03, GBKG04]. A significant advantage of volume ray casting over surface reconstruction is that it allows volumetric translucency, which gives users additional hints about the nature of a volume. Bones, for example, could be rendered as opaque solid surfaces, but muscles and other soft tissue can be rendered as translucent volumes in front of the bone. The mapping from the density of a voxel to a color and opacity value is done via

transfer functions [LKG+16]. While the ray traverses through the voxel volume, it looks up the density at certain intervals along the ray, maps this density to a color and opacity value using the transfer function, and then adds the contribution of the sample to the final pixel color. Traversal stops as soon as a fully opaque sample is encountered, or if the contribution of additional samples falls below a certain threshold because of the accumulated opacity of previous translucent samples (e.g., dense fog or looking through multiple sunglasses).

Point Clouds vs. Molecular Rendering

Molecular rendering relates to point clouds in that molecule data sets may consist of billions of atoms. Some important differences are that molecules are typically more voluminous, that atoms are connected, and that specialized visualization techniques are required to present the data in an intelligible way. On the one hand, the voluminous nature makes rendering more difficult because each view contains a large number of occluded points that reduce performance even though they are invisible. On the other hand, the specialized visualization techniques usually involve significant reductions of the level of detail by merging clusters of atoms into abstract shapes, which significantly reduces the amount of primitives that need to be rendered [MAPV15].



(a) (b)

Figure 1.9: (a) Molecular model of an HIV virion. Figure taken from slides of [MMS+16]. (b) Molecular model of the Coronavirus SARS-CoV-2. Figure courtesy of Nanographics [7].

Point Clouds vs. Particle Systems

Particle systems are typically computer-animated sets of small particular matter, with each particle being represented by one or a small number of shapes such as billboards, spheres, or complex geometry like mesh models. An animated set of points created entirely

[7] https://nanographics.at/

on the computer would typically be referred to as a particle system, whereas 3D scans and animations captured by 3D scanners at different moments in time would be rather referred to as a point cloud. Classic examples for computer-generated particle systems are smoke, fire, explosions, dust, snow, etc. Particle system animations usually involve an emitter, animation rules, particle shapes and materials, and a lifetime. Emitters create particles from a single point or spread over a surface or volume. Animation rules describe the behaviour of the particle, i.e., where they will travel to, whether all of them travel in the same direction or randomized within a certain range, if they are affected by forces like gravity, and also changes in shape and material. Particle shapes might not always represent the shape of the particles they are supposed to simulate. For example, a dust cloud may consist of millions of particles, but simulating such a large number of points is not feasible for games. Instead, particles are simulated in groups, i.e., millions of actual particles are represented by thousands of simulated particles, e.g., through textured planes where each textured plane represents thousands of particles with similar appearance and behaviour.

Figure 1.10: Particle systems simulating movement and collisions of snow and water. Figure taken from Kolb et al. [KLRS04].

1.3 Rendering Large Point Clouds

Efficiently rendering large data sets requires level-of-detail structures that reduce the number of triangles, points, voxels, etc. that are rendered. Three of the main problems that are addressed by LOD structures are:

- **Memory:** Reducing the memory requirements by only loading and rendering a small amount of the full data set. Additional data is loaded on demand, and data that is no longer needed is removed from memory. This is commonly referred to as out-of-core rendering.
- **Peformance:** Reducing the performance requirements by processing and rendering a small amount of the full data set. This is possible because out of a hundred million triangles, at most two million can be visible in an image with a resolution of 1920x1080. The remaining 98 million triangles waste rendering time without contributing to the final result.

- **Quality:** Reducing aliasing artifacts that happen when a large amount of data is projected to a single pixel. Choosing a single sample (triangle, point, texel, ...) per pixel leads to a loss of information that manifests as noisy/grainy renderings and also leads to sparkling during motion because the selected samples change from frame to frame, e.g., a white texel might be selected in one frame and a black texel in the next.

A wide range of LOD algorithms for mesh and terrain rendering is described in *"Level of Detail for 3D Graphics"*[LRC+03]. The book differentiates between discrete, continuous, view-independent, and fidelity vs. budget-based simplification of 3D models. According to the book, a discrete LOD approach *"creates multiple versions of every object, each at a different level of detail, during an offline preprocess. At run-time the appropriate level of detail, or LOD, is chosen to represent the object"*. Because the viewpoint is not accounted for in the LOD generation, the details are typically reduced uniformly across the whole object. This definition of discrete LOD does not apply to view-dependent LOD methods, which render parts of a model with a higher (but discrete) LOD close to the camera, and a lower LOD at the distance. In this thesis, we will use a slightly adapted definition of discrete LODs that also differentiates between discrete and view-dependent vs. continuous and view-dependent levels of detail:

- **discrete vs. continuous:** Describes whether levels of detail switch in sudden steps or if there is a smooth transition between one level of detail to the next, independent of whether this happens for the whole object or parts of it.
- **view-independent vs. view-dependent:** View-independent approaches take the distance between the viewer and the whole object (i.e., its bounding-box center), but not the view frustum, camera orientation, or distance to individual parts of the object into account. This works well for outside-in views of objects. If users move within the bounding box of the object, view-dependent approaches are better suited because they typically operate in a finer-grained fashion that allows rendering different parts of the model at different levels of detail, or not at all if outside the view frustum.
- **fidelity-based vs. budget-based:** Fidelity-based methods try to match certain quality requirements, e.g., a density of one triangle per pixel. Budget-based methods render as much detail as possible under certain performance constraints, e.g., rendering at most one million points per frame. Throughout this thesis, the term *point budget* is frequently used to refer to the number of points that should be drawn in a frame.

The first level-of-detail approaches for point clouds were proposed in 2000 with Surfels [PZvBG00] and QSplat [RL00, RL01]. The former was mainly targeted at high-quality rendering using point-based rendering, whereas the latter introduced an approach to render large triangle models as a hierarchically organized point cloud. QSplat first

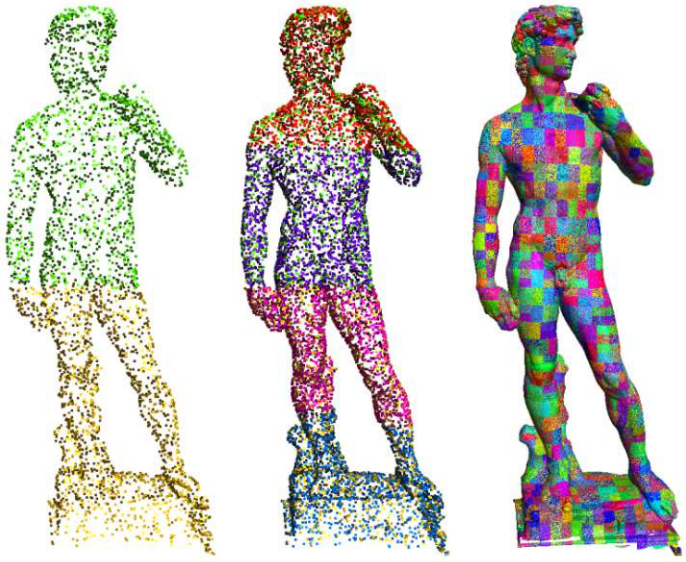
generates a bounding-sphere hierarchy based on the input mesh. During runtime, the hierarchy is traversed, and nodes are drawn as points if they are leaf nodes or if the bounding sphere is so small that further traversal does not pay off. At the time, QSplat was targeted towards systems without dedicated graphics hardware, i.e., it was implemented as a CPU-based renderer and as such, the structure is not suitable to massively parallel GPU rendering. Dachsbacher et al. [DVS03] introduced a more GPU-friendly structure, the Sequential Point Tree, which resembles a flattened version of the QSplat structure, stored in a single array sorted by lowest to highest level of detail. During runtime, a certain prefix (subset from the beginning) of the array is rendered, depending on the distance to the model. Sequential Point Trees belong to the class of view-independent LOD methods, meaning they do not take the camera orientation into account. These kinds of LOD methods are useful for outside-in views, i.e., the user observes the object from around but not from the inside. 3D models that are meant to be navigated from the inside require view-dependent LOD structures, which also cull away data based on the view frustum.

Gobbetti and Marton [GM04b, GM04a] introduced Layered Point Clouds in 2004 as a GPU-friendly and view-dependent LOD structure for large point clouds that serves as the basis for most of today's state-of-the-art LOD structures for point clouds.

1.3.1 The Layered Point-Cloud LOD Structure

Layered point clouds and their variations are a common LOD structure in state-of-the art point-cloud renderers, e.g., Potree [Pot], Entwine [Enta, ENTb], Arena4D [Are]. This section provides a detailed description of layered point clouds using octrees, which we use in our own implementations.

Layered Point Clouds (LPC) were introduced by Gobetti and Marton [GM04b] as a multiresolution structure for point clouds that groups points into chunks with M evenly distributed points each. The root node contains a representative subsample of the whole point cloud comprising M evenly distributed points. The space is then split in two along the longest axis and the remaining points are distributed to the respective children. Subsampling and splitting are repeated recursively until a child node contains less than M points, in which case the node is considered a leaf node. Figure 1.11 shows the amount of detail achieved by rendering the first level, the first two levels, and by rendering all chunks that are needed such that the resulting resolution is about 1 pixel.



(a) level 0, clusters: 2, splats: 4k

(b) level 0+1, clusters: 6, splats: 12k

(c) pixel tol.: 1, clusters: 1720, splats: 3435k

Figure 1.11: Original layered point cloud structure using a binary tree that splits along the longest axis. Points colored node-wise. Figure taken from [GM04a].

The LPC structure consists of two major components: The hierarchy that that describes a partitioning of the three-dimensional space into nodes, and chunks of points associated with each node. Although the hierarchy was originally defined to be a binary tree that splits along the longest axis, we will also consider variations such as octrees, kd-trees, multi-way kd-trees, etc. to be layered point-cloud structures within this thesis, since the basic concepts are largely the same.

Our research presented in this thesis is based on layered point clouds with an octree hierarchy, and point samples within nodes are selected by density criteria. Wimmer and Scheiblauer [WS06] originally formulated a nested octree structure where points were organized in an outer octree, and the points inside the octree nodes were themselves also organized into an inner octree. For the purpose of modifying operations on large point clouds, they simplified the nested octree structure into a modifiable nested octree (MNO) structure, where the concept of an inner octree was discarded and the points were instead stored in a simple array [SW11, Sch14]. The MNO structure is essentially an LPC using an octree hierarchy, but with one additional difference over the original LPC structure: the subsampling strategy. During the MNO indexing process, each node is assigned a sparse grid with 128^3 cells. The points of the input point clouds are then added to the root node and whenever a free cell is encountered, the point will be added to the root node. If a cell is already occupied, the point will be forwarded to the respective child node where the subsampling is recursively repeated. To avoid generating an excessive

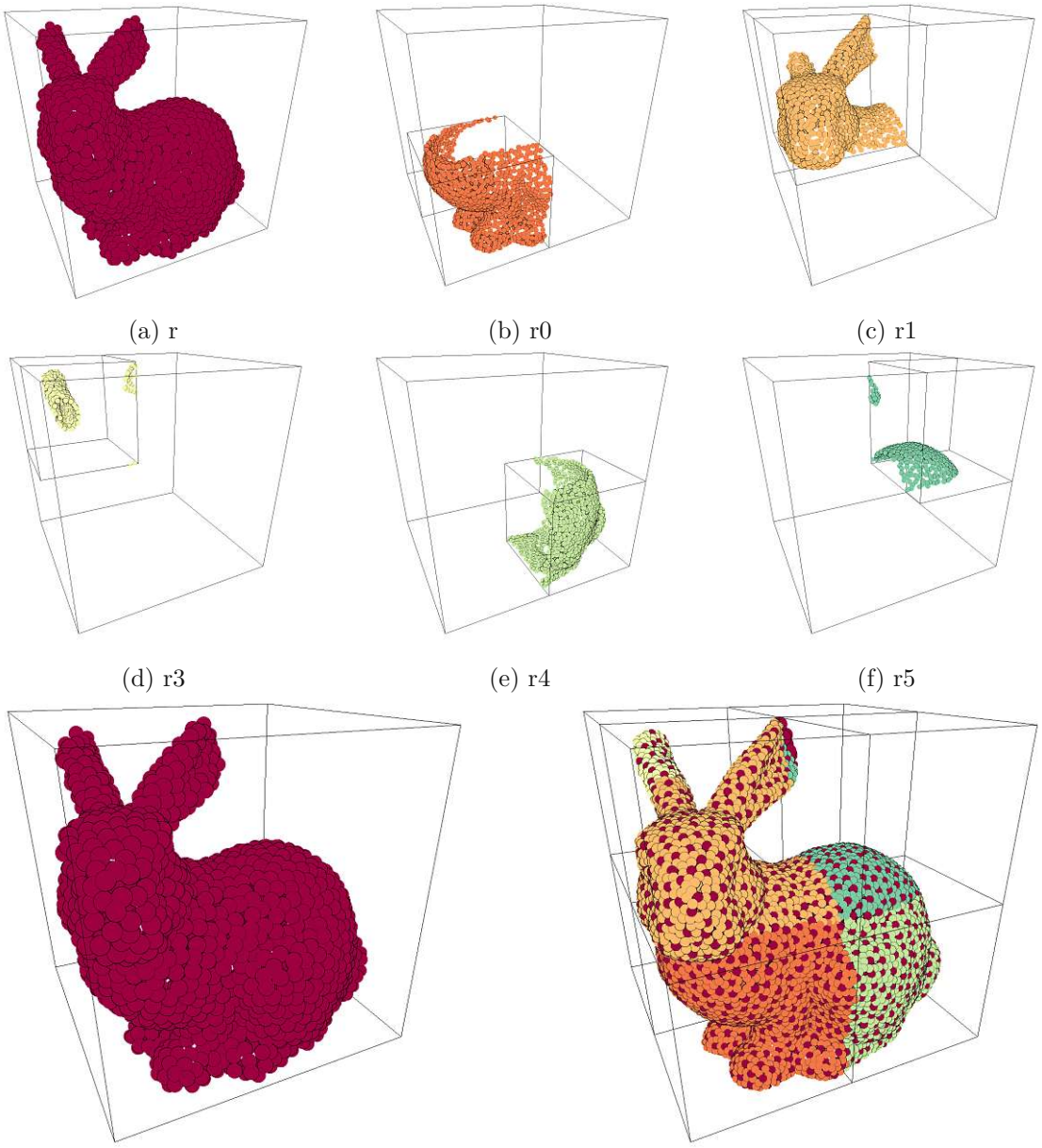
amount of nodes with a large tree depth, child nodes will initially start out as leaf nodes that act as a simple storage array. Up to the first X (e.g. 1000) points, points are simply inserted into the leaf node. If the threshold is exceeded, the hierarchy is expanded and the leaf node is transformed into an inner node that now executes the sparse grid sampling strategy whenever a point is added to it. Right after transforming the leaf to an inner node, all previously stored points are re-added, but this time the sampling strategy is executed. Figure 1.12 shows an example of a layered point cloud organized as an octree. Throughout the remainder of the thesis, *octree* will refer to this structure, unless specified otherwise.

Further work based on layered point clouds includes Goswami et al., who reduce the branching factor by using a multi-way kd-tree structure [GZPG10]. Tredinnick and Ponto et al. [PTC17, TBP16] developed a progressive point cloud rendering technique based on an octree structure. In each frame, a subset of the visible octree nodes is rendered, and previously rendered details are preserved through reprojection [WDP99]. Over the course of multiple frames, the result converges to the final image once all visible nodes were rendered at least once. Discher et al. [DMS+18] developed an efficient virtual reality point cloud renderer using a kd-tree structure, coupled with hidden-mesh optimization and a node-wise drawing order from front to back. Kang et al. [KJWX19] propose an efficient in-core octree generation algorithm that achieves a throughput of up to 2.5 million points per second through a sampling strategy that selects a random point per sample grid cell. To the best of our knowledge, Martinez et al. [MRVvM+15] were the first to built an octree of as many as 640 billion points[8] using their Massive-PotreeConverter[9]. They first partition the full point cloud into 16x16 tiles, then generate an octree out of each tile in parallel, and finally merge all tiles into a single overarching octree.

Wand et al. also proposed an octree structure specifically targeted at editing operations [WBB+07]. In this structure, all points are distributed into leaf nodes, rather than just the leftovers that weren't accepted by inner nodes. The inner nodes then store representative points that can be either samples that were chosen by checking occupancy of a sampling grid, similar to the modifiable nested octree structure, or they can be computed as a an average from all points in a region to reduce aliasing artifacts. In addition to the point samples, Wand et al. also compute the number of points that are represented by a point in lower levels of detail, which is used to track insertion and deletion operations.

[8] AHN2 data set, country wide scan of the Netherlands

[9] https://github.com/NLeSC/Massive-PotreeConverter

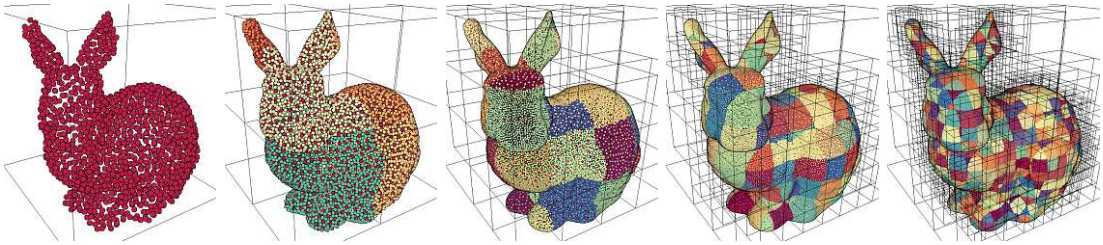


(a) r (b) r0 (c) r1

(d) r3 (e) r4 (f) r5

(g) level 0 (h) level 1

Figure 1.12: Variation of layered point clouds using an octree. Each node stores a subsample of the full point cloud with a certain density or spacing between points. In this figure, the spacing between points is equal to a 1/32nd of the width of the node. (a) The root node has the id *r*. (b-f) Descendants are identified by appending the child node index to the node id. E.g., *r0* is the first child of the root node. (g) Level 0 consists only of the root node itself. (h) Each further level renders nodes at that level and also all ancestors up to the root. There are no duplicates - the full data set is recovered by combining all points in all nodes.



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

Figure 1.13: Rendering nodes up to level 4. Figure taken from [SMOW20].

1.4 Problem Statement

This thesis, "Interactive Exploration of Point Clouds", aims to improve the state of the art with respect to point-cloud rendering quality and performance. During the course of this thesis, we have identified the following two problems as particularly relevant:

- Generating LOD structures for large point clouds is a slow and time-consuming process. The fastest available LOD generators achieve throughputs of up to around 1 million points per second – around 20MB/second assuming 20 bytes per point. Modern SSDs offer read and write performances that are 100 times as large as that, however. We address this problem in two fundamentally different ways: Through algorithms that improve LOD generation performance by up to an order of magnitude, and by a progressive rendering method that can render large point clouds without the need to generate LOD structures at all.
- The widely used discrete LOD structures organize points into LOD chunks with different point densities. The problem is that the transition from one LOD to the next exhibits very noticeable sudden drops in point densities, which are especially disturbing in virtual reality. We developed a continuous LOD rendering method that results in smooth transitions from high levels of detail to lower levels of detail, thereby improving the perceived visual quality.

1.5 Contributions to the State of the Art

The key contributions of this thesis to the state of the art are:

1.5.1 Octree Structure Generation

Summary: A fast LOD generation method that organizes points into an octree up to an order of magnitude faster than the state of the art. The key contribution of this method is using a hierarchical counting sort to efficiently break the full point cloud down into small chunks that can then be processed in parallel, while at the same time

avoiding generating a massive amount of tiny chunks. In addition, we also contribute an approximate blue-noise sampling strategy suitable for the bottom-up generation of the octree.

Individual Contributions: The first author, Markus Schütz, devised and implemented the method. Stefan Ohrhallinger and Michael Wimmer contributed ideas and discussions to sampling strategies and implementation details, and were involved in writing the paper.

Publication: Markus Schütz, Stefan Ohrhallinger, Michael Wimmer. *"Fast Out-of-Core Octree Generation for Massive Point Clouds"*. To be published in a special issue of Computer Graphics Forum, and to be presented at the Pacific Graphics 2021 conference [SOW20]. The paper is available for download at our institute page: `https://www.cg.tuwien.ac.at/research/publications/2020/SCHUETZ-2020-MPC/`.

1.5.2 Progressive Rendering

Summary: A progressive point-cloud rendering method that is capable of rendering unstructured point-cloud data without the need to generate LOD structures in advance. The key contributions are the high-quality progressive point cloud rendering by rendering random subsets of the point cloud in each frame (details are preserved through reprojection) and efficiently rendering random subsets by shuffling the points on-the-fly during loading using a unique random number generator suitable for massively parallel shuffling on the GPU.

Individual Contributions: The first author, Markus Schütz, devised and implemented the method – initially as a poster contribution to SIGGRAPH 2018, which was later extended to a full paper at EUROGRAPHICS 2020. Gottfried Mandlburger and Johannes Otepka (Department of geodesy) initiated the effort to extend the poster to a paper by approaching us with a problem that the poster did not address – handling point clouds with a large amount of attributes per point. They were subsequently involved in discussing strategies to solve this problem, as well as providing respective data sets and writing the paper. Michael Wimmer was involved in discussions about algorithmic details and also made sure to push towards supporting as many points as the GPU memory could fit (evaluated with up to 1 billion points), rather than stopping at point clouds that fit into the maximum size of a single buffer (2GB, 134 million points). In addition to the listed contributors, Stefan Ohrhallinger also helped formulating a proof for the random number generation formula used in the paper.

Publication: Markus Schütz, Gottfried Mandlburger, Johannes Otepka, Michael Wimmer. *"Progressive Real-Time Rendering of One Billion Points"*. Published in Computer Graphics Forum, presented at Eurographics 2020 [SMOW20]. The paper is available for download at our institute page: `https://www.cg.tuwien.ac.at/research/publications/2020/schuetz-2020-PPC/`.

1.5.3 Continuous LOD

Summary: A continuous level-of-detail approach that replaces the sudden drops of point densities known from discrete LOD structures by a gradual reduction of the point density. This is achieved by taking a standard discrete LOD structure, randomizing the discrete values of each point, and then subsampling points on a frame-by-frame basis based on the randomized LOD value of the points and the desired continuous LOD value as computed from the current viewpoint.

Individual Contributions: The first author, Markus Schütz, devised and implemented the method. Katharina Krösl and Michael Wimmer participated in high-level discussions about the method and goals. Katharina Krösl also contributed in organizing and evaluating the user study, and Michael Wimmer was involved in writing the paper.

Publication: Markus Schütz, Katharina Krösl, Michael Wimmer. *"Real-Time Continuous Level of Detail Rendering of Point Clouds"*. Published in 2019 IEEE Conference on Virtual Reality and 3D User Interfaces (VR), presented at IEEE VR 2019, Osaka [SKW19]. The paper is available for download at our institute page: `https://www.cg.tuwien.ac.at/research/publications/2019/schuetz-2019-CLOD/`.

1.5.4 Compute Shader Point Rendering

Summary: An approach to render point clouds with custom compute shaders instead of using the traditional rendering pipeline. This is done by encoding color and depth into a single 64-bit integer, and then using atomicMin to retain the points with the smallest depth values.

Individual Contributions: The first author, Markus Schütz, devised and implemented the paper. Michael Wimmer participated in discussions about the method and was involved in writing the 2-page abstract as well as the poster.

Publication: Markus Schütz, Michael Wimmer. *"Rendering Point Clouds with Compute Shaders"*. Poster publication at SIGGRAPH Asia 2019. [SW19]. The poster is available for download at our institute page: `https://www.cg.tuwien.ac.at/research/publications/2019/SCHUETZ-2019-PCC/`.

Although our approach to *Progressive Rendering* is meant to be a replacing alternative to *Octree Structure Generation*, Treddinick and Ponto et al. [TBP16, PTC17] have previously shown that progressive rendering and LOD rendering are not mutually exclusive and can be combined to further improve the performance of LOD-based rendering approaches. *Continuous LOD* is based on the discrete octree structure produced by *Octree Structure Generation*, but our current implementation only retains the hierarchy levels and discards the remaining information, e.g., the grouping into nodes. The compute shader based point-cloud rendering method is an early experiment that has shown potential to be up to 10x faster than the traditional rendering pipeline in some cases, but initial tests with LOD-based structures were not successful, yet. Detailed contributions are listed in the respective chapters and further discussions are found in the conclusion.

CHAPTER 2

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

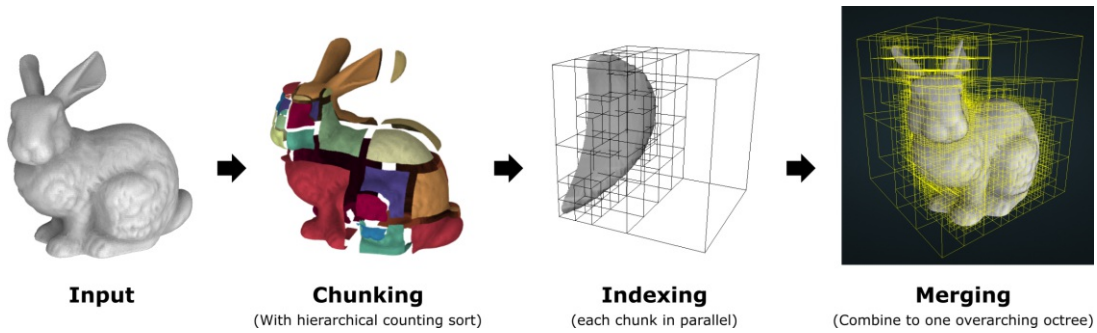


Figure 2.1: Our LOD generation approach first partitions the input into small chunks. Each chunk is then converted into an octree in parallel, and eventually merged into a single overarching octree. Figure taken from [SOW20].

The contents of this chapter were initially published as a paper *"Fast Out-of-Core Octree Generation for Massive Point Clouds"* in Computer Graphics Forum, and will be presented at the Pacific Graphics 2021 conference [SOW20].

2.1 Introduction

Terrestrial laser scanning, photogrammetry, and aerial LIDAR scanning operations yield hundreds of millions to trillions of points nowadays. Due to the large storage and memory requirements, these kinds of data sets need to be processed in an out-of-core fashion, where only a small part of the data is loaded into RAM or GPU memory at any given time. For processing tasks, the point-cloud data is often stored such that specific regions can be accessed quickly, e.g., in separate tiles or in hierarchical structures like kd-trees in which all points are stored in leaf nodes [PMOK14, OPA]. For real-time rendering, level-of-detail (LOD) structures are required (as discussed in Section 1.3), where lower levels of detail contain representative subsets that give users the impression that they are looking at the full data set, even though only a fraction of it is being loaded and rendered. In this chapter, we focus on the fast generation of LOD structures that are suitable for real-time rendering purposes.

The LOD structure that we target with our method is a variation of a layered point cloud [GM04b] that uses an octree with an additive scheme, as used by Potree [Pot] and Entwine [Enta]. These octree structures populate each node with a subset of the full data set, and the combination of all nodes yields the original data set without duplicates. Additive scheme means that during rendering, higher levels of detail contain additional points that are rendered together with points in lower levels of detail. Alternatively, one could use a replacement scheme where higher levels of detail replace lower levels. The advantage of the replacement approach is that it would allow us to compute representative subsets with baked-in anti-aliasing, similar to mip maps, however we chose the additive approach at this time because it is faster to generate and render, and requires less memory. Figure 2.2 illustrates individual octree nodes and the subsets of points that are stored inside.

Our proposed LOD generation method attempts to take advantage of SSDs by adding an IO heavy preprocessing step (chunking) that rearranges points on disk into small chunks that are better suited for parallel processing in the subsequent LOD generation (indexing) step. The generated chunks are small enough so that one chunk per CPU core can be loaded and processed. The resulting LOD structure is a layered point cloud octree, largely identical to the modifiable nested octree [Sch14] and Potree structure [Pot], but using different subsampling strategies and writing the results to a total of three instead of thousands to millions of files. The implemented subsampling strategies are an approximate poisson-disk strategy, and a uniform random sampling strategy.

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

- A hierarchical counting sort that is suitable to partition a point cloud by up to 9 octree levels by looping through all points twice. This counting sort is applied in an out-of-core fashion during the chunking phase in order to generate small point cloud files (∼10M points), and is applied again in an in-core fashion during the indexing

phase to create small leaf nodes (∼10k points) that can then be subsampled bottom up.

- A simple and fast hierarchical approximate blue-noise subsampling algorithm that keeps sampling artifacts across borders of adjacent nodes subtle, even though distances are not enforced across borders.
- An octree generation method that utilizes the bandwidth of modern SSDs, rather than avoiding disk access at all costs.

2.2 Related Work

This section describes three categories of work related to our method: Counting sort, LOD structures for point clouds, and blue-noise sampling. The previous art on LOD structures is mostly covered in Chapter 1 but we will discuss some particularly relevant parts.

2.2.1 Counting Sort

Counting sort is an integer sorting algorithm that runs in linear time [Knu98, CLRS01], as opposed to its comparison sort based counterparts with a time complexity of $O(n \log n)$. It is applicable to data sets that are sorted by integer keys within a limited range. The range is limited because counting sort, as the name suggests, counts the amount of each occurring key and it uses an array of counters that is as large as the range of potential keys to do so. Applications of counting sort include point-in-cell simulations [Bow01], computing fixed-radius nearest neighbours for particle simulations [Hoe14], substeps of radix-sort routines, and in our case the block-wise sorting of points. A survey of Arkhipov et al. [AWLR17] discusses sorting algorithms on GPUs, including counting sort as well as radix sort based on counting sort.

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

2.2.2 Point Cloud LOD Structures

In addition to the LOD approaches mentioned in Section 1.3, we would like to mention the research of Martinez et al. [MRVvM+15] as particularly relevant. They create an octree for 638 billion points by splitting the data set into 16 x 16 tiles and then executing PotreeConverter for each tile simultaneously on a server system with 2 x 8 cores and on a distributed supercomputer. Afterwards, the individually generated octrees are merged into a single one. The resulting runtime is 15 days, which corresponds to a throughput of around 0.49 million points per second. Our approach also splits the point cloud first and merges the results afterwards, but instead of splitting on a 2-dimensional 16^2 grid,

which corresponds to 4 quadtree levels, we split on a three-dimensional 512^3 grid, which corresponds to 9 octree levels. In addition to that, our evaluation runs on a single-CPU system instead of a dual-CPU server system plus a supercomputer. We were only able to evaluate our approach for up to 100 billion points, however, due to lack of disk space. Similar in spirit to tiling the data first, Leimer and Scheiblauer [Lei13, Sch14] sort the data first in order to optimize the input for subsequent indexing operations.

Simultaneously to us, Bormann and Krämer worked on similar octree generation improvements that also aim to better exploit CPU cores and SSDs [BK20]. They also rearrange points in a preprocessing step in order to organize them into small batches that can be indexed in parallel, but they do this by sorting points in morton order whereas we use a hierarchical counting sort approach.

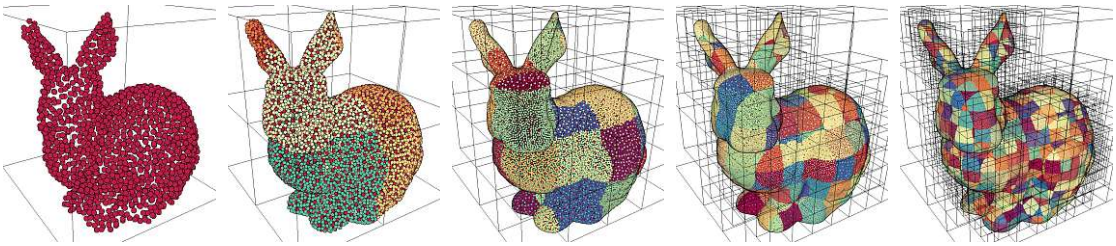
We would also like to list some performance numbers from prior work, specifically the throughout of the LOD generator in points per second. Wand et al. [WBB+08], Scheiblauer [Sch14], and Kang et al. [KJWX19] report **in-core performances** of about 0.3, 1.21, and 2.5 million points per second, respectively. Scheiblauer [Sch14], Richter and Döllner [RD10], Goswami et al. [GZPG10], Dieckmann and Klein [DK18], Martinez et al. [MRVvM+15], and Discher et al. [DRD18] report **out-of-core performances** of around 0.49, 0.1, 0.6, 0.2, 0.49 and 1.35 million points per second, respectively.

2.2.3 Blue-Noise Sampling

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

blue-noise characteristics, and the size of the prefix can be adjusted based on the distance to the object, similar to sequential point trees [DVS03].

2.3 Data Structure



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

Figure 2.2: Each octree node holds a subsample of the full point cloud. The root node contains a coarse low density subsample of the whole data set and with each level, the resolution is doubled. Points are colored by the node they belong to. Points in lower levels of detail are rendered together with points in higher levels of detail, as seen in (b) through the mixture of red points from the root and other colors from the respective nodes at level 1. Figure taken from [SOW20].

This section describes the generated data structure and its representation on disk. We generate a layered point cloud in an octree, largely identical to the modifiable nested octree used by Scheiblauer [SW11] and Potree [Sch16]. Figure 2.2 illustrates the tree structure and the contents in its nodes. The root node of the octree contains a coarse subsample that represents the whole data set at a low level of detail. With each level, nodes contain increasingly higher resolution subsamples of their respective regions. One of the main issues of modifiable nested octrees is that previous work generates one file per octree node, e.g., Martinez et al. [MRVvM$^+$15], who use Potree for their work, end up with 38 million files for a point cloud of 638 billion points. Each individual file adds significant overhead to file system operations such as accessing, copying, deletion, uploading to a server, etc., thereby increasing times for the respective operations from seconds and minutes to hours and days.

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

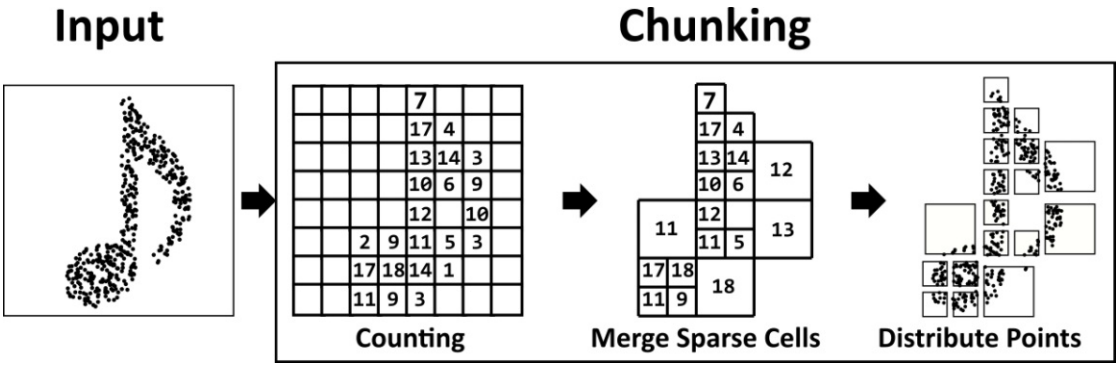


Figure 2.3: Chunking uses a hierarchical counting sort in order to partition the points by multiple octree levels at once. **Counting:** Count number of points that fall into cells of a high-resolution grid. **Merge Sparse Cells:** Any 8 adjacent and octree-aligned cells with less than a certain number of combined points are merged into larger cells. **Distribute Points:** Now that we know the location, extent, and the number of points in each chunk, we loop through all points again and directly transfer each point to its respective chunk / file.

load the hierarchy of the root node, we only get the first 4 levels. Once we reach nodes at the fourth level, we can load the next 4 levels for each node as required. The third file, metadata.json, contains information such as bounding box and point attributes that are required to load and decode the point data that is stored in octree.bin.

2.4 Method

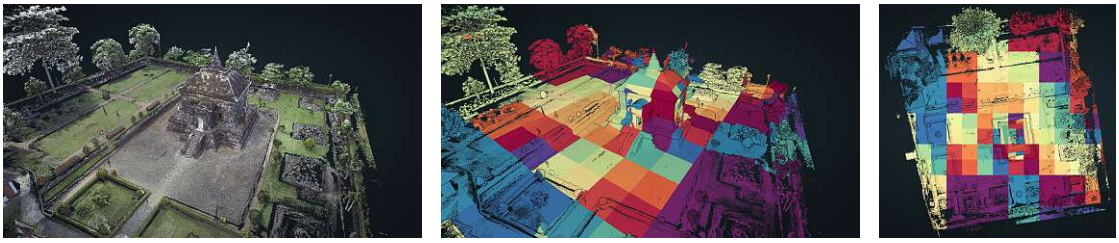
In this method, we differentiate between *local* and *global* octrees. Local octrees are generated separately for each individual chunk of the point cloud, and the global octree is the result of eventually merging all local octrees into a single octree containing the entire point cloud.

Our method consists of the following steps (see Figure 2.1):

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

2.4.1 Chunking

The chunking phase splits the point cloud into cubic chunks (e.g., files in out-of-core storage) that are small enough so that multiple chunks can be processed in parallel, but



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

Figure 2.4: (b+c) Each generated chunk (=file) rendered with a random color. The top view shows how more densely scanned regions are partitioned into smaller chunks, while sparsely sampled regions are merged into larger chunks before writing them to a file.

large enough to avoid a massive amount of tiny chunks. It is done using a hierarchical variation of counting sort, as shown in Figure 2.3. Counting sort is particularly useful for this task because it requires only two streaming passes: the first pass for counting points in a $2^{9^3} = 512^3$ grid (for an outer octree of 9 levels), and a second pass to distribute the points to the respective chunks. In more detail, the approach consists of the following steps:

Counting First, we divide the cubic bounding box of our point cloud into a 3-dimensional grid with 2^{depth} cells on each axis. Each cell represents a counter for all points inside it. A size to the power of two is required to align the counting grid with an octree. *Depth* specifies the depth of the octree. The depth and therefore the resolution of this grid defines the smallest possible extent of the chunks we are about to generate. The generated chunks should be small enough so that multiple chunks can fit in memory and be processed simultaneously by the indexing phase following later on. Larger point clouds will require a higher grid resolution so that the resulting chunks are sufficiently small. In our implementation, we use grid sizes of 128, 256 and 512 on each axis. A counter grid using 32 bit integers with a size of 128 requires $4 * 128^3 = 8$ MB memory, which constitutes a low memory footprint, is quickly allocated, and also quickly processed by the merging phase. A size of 512 requires 536 MB and already adds significant processing overhead. We suggest 128 for point clouds with less than about 100 million points, 512 for point clouds with more than 500 million points, and 256 in between. A grid size of 512 can accommodate point clouds with relatively uniform scan densities with hundreds of billions of points (e.g., aerial LIDAR, 116 billion points, see Figure 2.11) but also scans with strongly varying point densities (e.g., terrestrial laser scans) with a few billion points. The latter may lead to "teapot in a stadium" scenarios that result in chunks that are larger than desired because the counting grid cells are not small enough to split up the small "high-resolution teapot". In these cases, we suggest to recursively run the chunking process again on all chunks that are too large, e.g., larger than 500MB. Chunks should be small enough so that a total of at least $2 * numThreads * chunkSize$ RAM is available. However, we did not need to recursively split chunks for any of our test data sets benchmarked in Section 2.7.

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

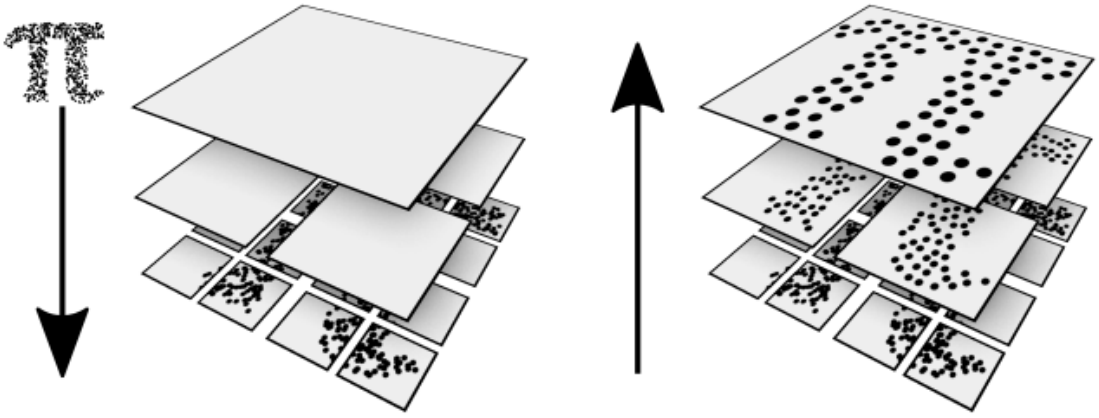
Create Chunk Lookup Table After computing the list of chunks, we create a lookup table (LUT) with the same size as the counting grid and with pointers from the grid cells to the respective chunks. During the distribution phase, this LUT allows us to identify the target chunk for each point with a single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

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

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

2.4.2 Indexing

In this step, the previously generated chunks are loaded from disk and converted into local octrees in parallel using one thread per chunk. Points are first partitioned into leaf nodes and coarser levels of detail are populated by recursively extracting subsamples of higher levels of detail from the bottom up. The subsampling strategy is exchangeable and we describe two example implementations in further detail in section 2.5.



(a) Chunking (b) Bottom-Up Subsampling

Figure 2.5: Indexing: Create a multi-resolution octree for each generated chunk. (a) First, points in a chunk are partitioned into leaf nodes with a specified maximum number of points, using the same hierarchical counting sort procedure that was used to generate the chunk files. (b) Coarser levels of detail are then populated by recursively subsampling child nodes from the bottom up.

Building an octree out of a chunk is done in a bottom-up fashion, as shown in Figure 2.5. The points are first partitioned into leaf nodes with a certain maximum size using largely the same hierarchical counting sort approach as during the chunking step, but this time it is applied in-core. We suggest a maximum size of 10k points per leaf node to obtain a sufficiently fine grained LOD representation for real-time rendering purposes. Since there are multiple threads working on different chunks simultaneously, and because each thread reuses and resets the counting grid from one chunk to the next, we have to use lower resolution counting grids compared to sizes of up to 512 during chunking. In our implementation, we use counting grid sizes of 32 during the indexing phase, which corresponds to partitioning the points by 5 octree levels. This is often not enough to obtain leaf nodes with a maximum size of 10k points, so we recursively split all nodes that are still too large by another 5 octree levels until they are sufficiently small. Massive amounts of tiny nodes are also avoided the same way as during the chunking phase by merging leaf node candidates that have less than 10k points, combined. The hierarchical counting sort procedure produces a list of leaf nodes containing arrays of points. We then create the missing inner nodes between the local octree root and the computed leaf nodes to obtain an octree where only leaf nodes are populated, as shown in Figure 2.5a.

Coarser levels of detail are populated by recursively extracting samples out of finer levels of detail from the bottom up, as shown in Figure 2.5b. Bottom-up traversal is implemented as a post-order depth-first octree traversal. If a visited node is a leaf node, we ignore it. Otherwise, we apply one of the subsampling strategies described in section 2.5 with the points of all direct child nodes as the input. The accepted subsample

is stored in the current node and the remaining points are transferred back to the child nodes. At this point, the child nodes are complete and no longer needed for further processing, so we flush them to a single output file (octree.bin). The only information we keep in memory is the octree hierarchy, including the location and size of a flushed node inside the output file, which is necessary because multiple simultaneously processed chunks flush nodes to a single output file in no particular order. During rendering, we will need the location and size of each node to load the right range of data from the file.

2.4.3 Merging

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

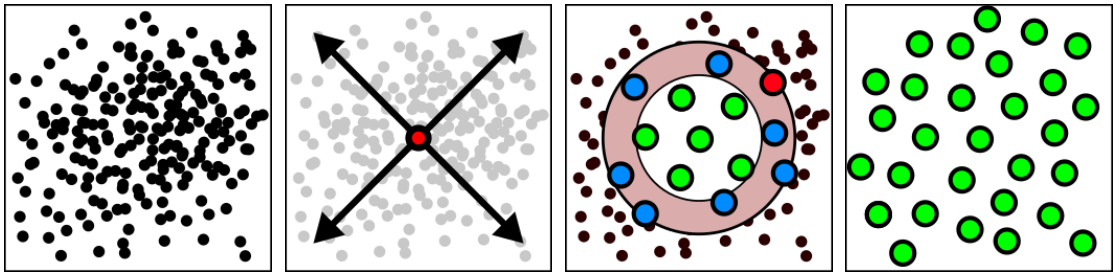
2.5 Subsampling

The subsampling strategy used during the indexing step in section 2.4.2 is, with some limitations, exchangeable. We implemented and evaluated two approaches, a fast random sampling strategy with a certain level of uniformity as described by Kang et al. [KJWX19], and an approximate Poisson-disk sampling strategy [Coo86, YGW+15] that enforces a minimum distance between points except between adjacent octree nodes. For each node, the input to the sampling method consists of points in its direct child nodes. The result is a subset of points that will be stored inside the current node, and the remaining points are transferred back to the respective child nodes. This process is repeated from the bottom up until all nodes up to the root node are populated with points. A limitation of this approach is that subsampling strategies do not have access to points in adjacent nodes, which can lead to noticeable sampling patterns along borders of two nodes. An advantage of bottom-up approaches is that the number of points to be subsampled quickly diminishes with each level. Martinez et al. [MRVvM+15] found that a comparatively flat country-wide LIDAR scan of the Netherlands has a reduction factor of about 4, meaning that each coarser level of detail has only a fourth of the points left. Once we have subsampled the bottom-most level, we only need to do a quarter of the work for the next level, unlike top-down approaches where the majority of the points need to apply subsampling procedures at all levels of the hierarchy until they reach the bottom.

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

The Poisson-disk approach attempts to generate subsamples with visually pleasing blue-noise properties. It accepts points with a certain minimum distance to previously accepted points, and rejects them otherwise. This method is usually relatively slow due to the required distance checks, but the results are generally considered to be of higher quality. A naive approach to Poisson-disk sampling would iterate over a list of points and accept a new point if it is far enough away from all previously accepted points. Since most of the generated nodes contain around 10k to 50k points, this naive approach would lead to tens of thousands of distance checks per point on average. We reduce the computation time by sorting the points from inside out first, which implicitly gives us a spatial acceleration structure that restricts the search radius to a spherical shell with a thickness equal to the minimum distance. Sorting also packs the samples close together and eliminates the possibility of ridges that appear in Potree and Arena4D, as shown in Figure 2.7. These ridges appear when candidates are evaluated in an unfavourable order, e.g., lines of points with line 1 then 4, but it turns out line 3 was also far enough away from line 1 but it cannot be accepted anymore because we already accepted line 4, which is too close to line 3. Figure 2.6 illustrates the steps of our Poisson-Disk approach: First, the input samples are sorted inside-out by their distance to the center of the node. Then, for each input point we check the distances to previously accepted points in an outside-in order by looping through the list of accepted points, which is implicitly ordered inside-out, from the end. If the difference between $distance(center, candidate)$ and $distance(center, acceptedPoint)$ is larger than the minimum distance, we can safely accept the current candidate because all the other previously accepted points are even closer to the center and cannot be closer than minimum distance. Figure 2.6c shows that this corresponds to evaluating distance checks to previously accepted points within a ring (spherical shell in 3D). This approach works well in practice for two reasons: First, because the amount of input data – the points from all direct child nodes – is in the order of only 10k to 50k points. And second, because most point cloud data sets represent surfaces rather than volume data, so the amount of hit tests against previously accepted points inside the ring (spherical shell) is considerably lower than it would be with volume data sets. In case of volume data, we expect that our approach would need to be extended with an additional spatial acceleration structure that further restricts the search area.

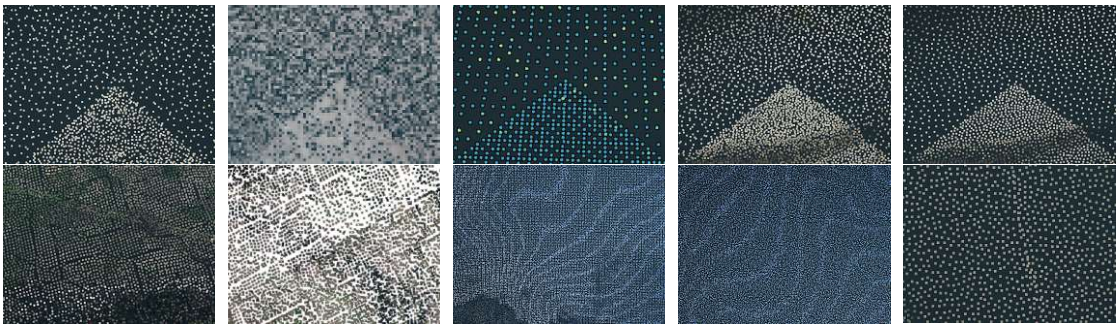
Figure 2.7 shows the sampling patterns and artifacts of Potree, Entwine, Arena4D and



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

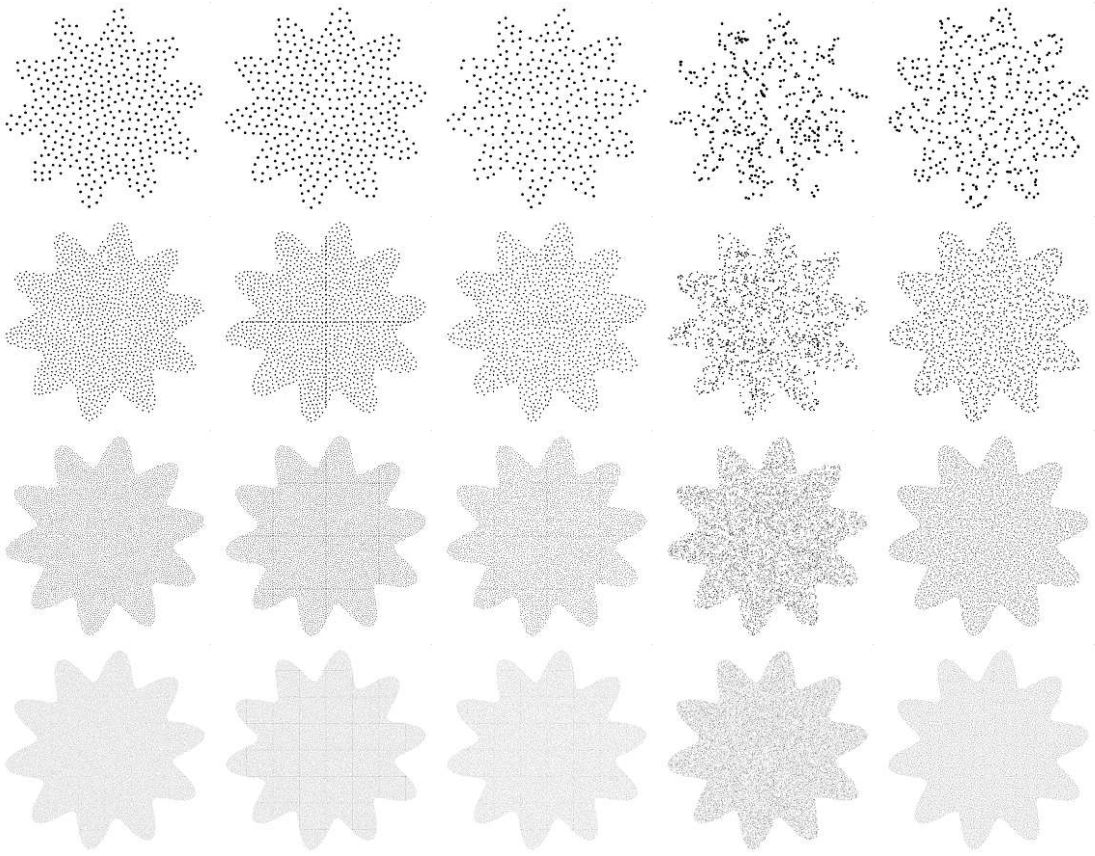
Figure 2.6: Our Poisson-disk sampling approach. **(b)** Sort points from inside out. **(c)** Loop through candidates from inside-out, run distance checks between candidate (red) to potential conflicts (blue) within ring as thick as the desired minimum distance.

our two strategies. Potree uses a form of Poisson-disk sampling within nodes, but the combination of nodes do not honor the minimum distance requirements. Furthermore, Potree evaluates points in the order in which they are stored on disk, and if the order is not favorable (previously accepted points block more suitable candidates that appear later in the list, e.g., data set CA13), sampling artifacts that manifest as ridges can appear ([Pot], p. 21). Arena4D produces similar sampling patterns and ridges, which leads us to believe that it is also affected by the order of the input. Entwine selects the points that are closest to the center of the sampling grid cells, which increases the average spacing between points and leads to a single deterministic subsampling result, independent of the order of the input. However, the resulting patterns have a noticeable regularity. The samples between cells appear farther apart than samples within a cell, which also produces noticeable but predictable and arguably less visually disturbing ridges. Our random sampling approach (after Kang et al.) shows results that are relatively similar to that of Potree in many cases. It does not suffer from ridges, but it can produce artifacts along slopes and smooth surfaces that manifest as denser lines that look similar to staircasing artifacts or contour lines. Our Poisson-disk approach produces high-quality results that honor the required minimum distances between overlapping nodes of different levels of detail. Although it does not enforce minimum distances between adjacent nodes, the distances end up sufficiently large in most cases due to Poisson-disk sampling from the inside out. By the time we evaluate candidates at the border, we already accepted points further inside, which reduces the chance that points closer to the border get accepted. It does not eliminate the possibility, however, so noticeable gaps or clusters at the border are possible, but less common and more subtle. Figure 2.8 illustrates the differences between random and uniform random selection, and it shows the impact of the sampling order on our Poisson-disk sampling approach.



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

Figure 2.7: Comparison of sampling patterns and artifacts. Top row: Sampling patterns at the border of two different levels of detail. These are often visible with low levels of detail settings or while waiting for higher levels of detail to be loaded. Bottom row: Most commonly encountered sampling artifacts. (a) Potree has noticeable clustering along borders of nodes, as well as ridges if the input is ordered unfavourably like in data set CA13. (b) Arena4D also shows ridges with some data sets. A detailed study and more faithful close-up screenshot was not possible because the software is closed and offers no option to render at very low levels of detail. (c) Entwine exhibits regular grid patterns. Staircasing artifacts are common along slopes or curved surfaces. (d) Our random approach looks similar to the results of Potree. It does not suffer from ridges or clustering at borders, but it shows a similar kind of staircasing artifacts on slopes and curved surfaces as Entwine. (e) Our Poisson-Disk approach shows uniform distances between points with no regularity. Points at borders of adjacent nodes can be too close or too far apart, but both cases are relatively rare and subtle due to the inside-out subsampling order.



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

Figure 2.8: Comparison of hierarchical bottom-up subsampling strategies. Subsamples in the bottom row are organized in 8x8 tiles / nodes, which corresponds to a quadtree with a depth of 3 levels. They are all subsampled from the same high-density input point cloud. The rows above it are made up of 4x4, 2x2, and 1 tile at the top. Each row is a subsample of the one directly below of it. (a) and (e) are the strategies we implemented in our octree generator. (a,b,c) Our hierarchical Poisson-disk sampling strategy only enforces minimum distances within a node, but evaluating points from the inside out reduces the chance of clustering artifacts near borders, while outside-in evaluation performs even worse than unsorted. (d, e) Simple random subsampling leads to poor coverage with clusters in some regions and holes in others. A uniform random subsampling strategy that selects one point per grid cell improves the point distribution.

2.6 Implementation Details

In this section, we describe implementation-specific details that are essential for our method and performance results but do not fit into the more abstract description of our

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

Table 2.1: List of test data sets. Eclepens is stored in an uncompressed LAS file, all other data sets are in an compressed LAZ format. CA13 and AHN3 comprise of non-overlapping tiles, while Affandi consists of strongly overlapping single scan positions.

method in section 2.4. Our method was implemented and tested using C++.

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

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

Writing nodes: Section 2.4.2 describes that all chunks are loaded and processed in parallel by multiple threads. Each thread first loads a chunk and eventually writes the results to disk, node by node. However, individually writing a large number of small nodes to disk is not efficient. Instead, we use a custom buffered writer object that collects and stores data from finished nodes in buffers of 16MB. If a newly finished node does not fit into the current buffer, the writer will flush the current buffer to disk in a dedicated thread, and simultaneously start building the next buffer. We also stop loading new chunks when the total backlog of the buffered writer exceeds 1GB, because sometimes loading and processing multiple chunks is faster than writing the results to a single file on disk.

2.7 Performance

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

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

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

Table 2.2 lists benchmark results of our method for various data sets, evaluated on both, HDD and SSD. The final merging step is not listed because the majority of the time is spent on creating the chunks and then indexing the chunks in parallel but it is included in the total. Table 2.3 and Figure 2.10 compare the LOD generation times of our method to Potree, Entwine and Arena4D. The results show that on SSDs, our method is about an order of magnitude faster than these three packages. The difference is less extreme on HDDs, which indicates that our method is the only one that efficiently utilizes the higher bandwidth of SSDs. A notable observation are the results for the Eclepens data set, which show a significantly lower throughput on Potree and Entwine. This is because the points in this data set exhibit poor locality and as a result, the top-down approach of Potree frequently flushes but then reloads data because processing points at any stage requires distance checks to points from any previous stage. Our method does not suffer from this issue because it first groups points into chunks and once a chunk is processed, its points are not needed anymore at later stages. However, if the point cloud is sorted by morton order, Potree and Entwine become faster by a factor of 3 to 4, as opposed to Arena4D and our approach that do not benefit from sorted input.

2.7.1 Case Study: AHN3

Our largest test data set, AHN3, contains 116 billion points. It is a subset of an even larger scan of the whole netherlands [1,2] but we clipped it due to lack of disk space on our test system. Figure 2.9 shows different viewpoints of the data set. The input consists of 216 LAZ compressed point cloud files with a total of 531 GB. The outputs comprises a 3.2 TB file with uncompressed point data and 962 MB for the hierarchy data. The latter substantiates the importance of splitting the hierarchy into chunks that can be loaded on demand. A total of 22 778 chunks were created during the chunking phase. Figure 2.11 shows a histogram of the storage sizes of the chunks – all of them small enough to load and process 16 at a time in system memory, but only 24 or 0.1% of them are what we would consider too small with potentially negative impact due to overhead. Although the largest chunk (307MB) contains 10.9 million points (900k above the specified threshold) we refrain from recursively splitting it further because it is still small enough to be processed alongside other chunks.

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

2.8 Problematic / Failure Cases

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

One of our users used the converter to build an octree out of a synthetic data set of a cube made of 300^3 points, which effectively represents a voxel data set stored as a point cloud. The Poisson-disk sampling strategy is currently not able to deal with such data sets, only the uniform random sampling strategy was able to successfully generate the LOD structure. Even so, the structure itself is not suitable for volumetric data sets. The resulting nodes contain about 2 million points and all of them store the point coordinates. Voxel structures would be more suitable because they do not need to explicitly store the

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

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

Table 2.2: Time to generate an octree with our method and sampling strategies.

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

Table 2.3: Comparing LOD creation times of Potree, Entwine and Arena4D to our method using Poisson-Disk or Random subsampling strategies. *nomem* indicates that the application ran out of memory (32GB).

(a) (b) (c)

Figure 2.9: AHN3 subset with 116 billion points (531GB compressed). (a) View of Amsterdam, Utrecht and Apeldoorn. (b) Downtown Amsterdam. (c) Closeup of Amsterdam.

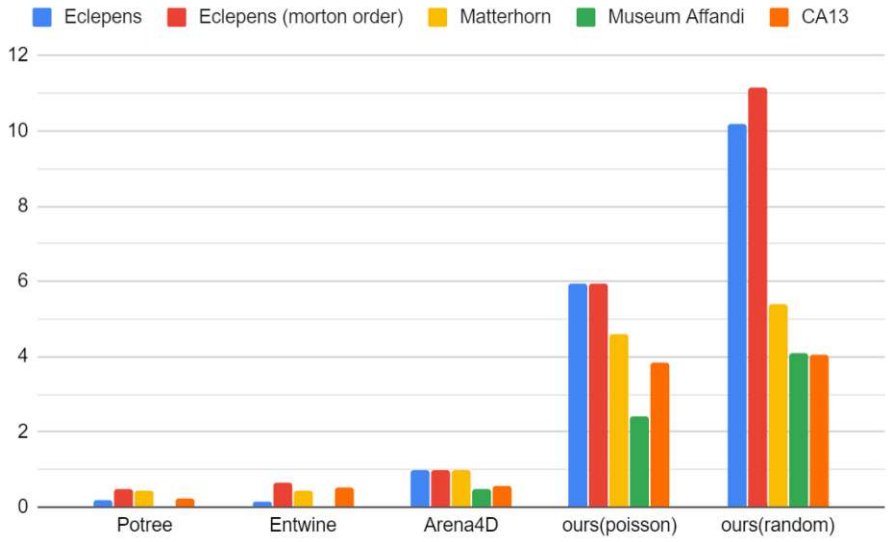


Figure 2.10: Throughput of various approaches on an SSD in million points per second (higher is better).

coordinates for volumetric data sets, and because the efficient rendering of volume data sets using voxels is a well researched topic.

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

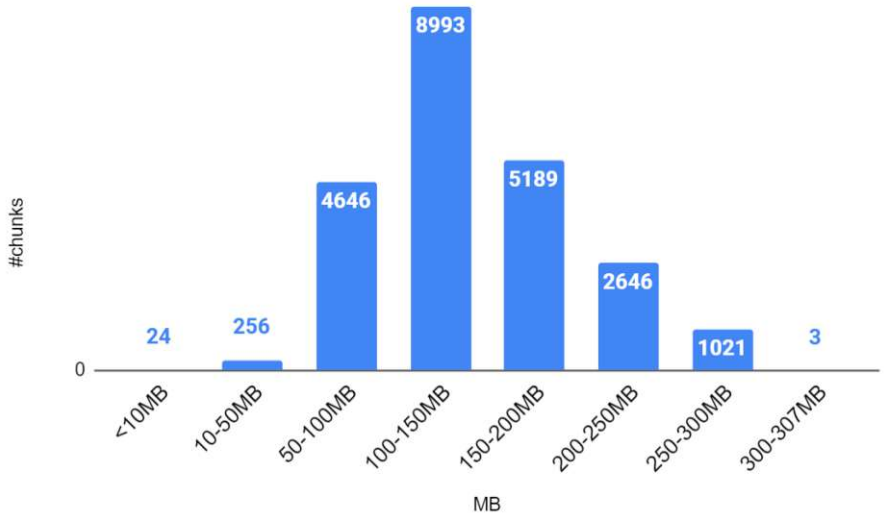


Figure 2.11: Histogram showing the amount of generated chunks with certain storage sizes for the 116 billion points AHN3 data set and with a counting grid size of 512^3.

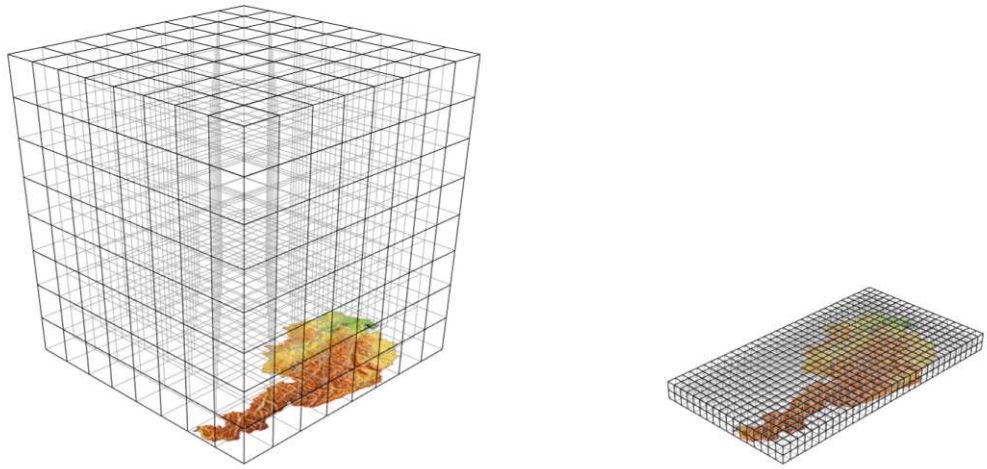
A third failure case was a point cloud with a bounding box that was a thousand times larger than it should have been. Nearly all of the points were within a region of about 400 meters, but the bounding box had an extent of 780 kilometers due to 6 outliers. This is problematic for two reasons: First,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

2.9 Conclusion

In this chapter, we have shown an LOD generation method for point clouds that is capable of generating an octree up to ten times faster than the state of the art. This is achieved by a chunking pass that efficiently divides the input into small batches that can be processed in parallel on the one hand, and by applying efficient subsampling strategies that select suitable points from higher detail nodes and promote them to lower detail nodes on the other hand.

We believe that hierarchical counting sort is a beneficial contribution to the state of the art as it allows us to efficiently partition a point cloud by multiple octree levels with just two passes over all points. A potential improvement includes developing ways to

efficiently split even more levels at once while avoiding an exponential growth of 8^{levels} of memory usage from the counting grid. For example, large-scale aerial LIDAR data is relatively flat so using a cubic grid wastes memory. Instead, the counting grid size could be adapted to the bounding box of the input, as shown in Figure 2.12. The resulting grid cells would still be cubic and align with an octree, we would just not allocate memory for cells that are outside of the bounding box. Taking Austria as an example: The extent of a point cloud of all of Austria would be roughly 575km x 300km x 3.8km. With our implemented approach that uses a cubic counting grid to split by 9 octree levels, our counting grid would consist of 512 x 512 x 512 cells taking a total of 512MB of memory. Adapting the counting grid to the bounding box would result in 512 x 268 x 4 cells that require just 2MB of RAM. The reduced memory footprint subsequently allows us to increase the octree depth, e.g., to 11 levels using 2048 x 1067 x 14 cells (116MB).



(a) Implemented cubic counting grid (b) Recommended fitted counting grid

Figure 2.12: We implemented a cubic counting grid, but in hindsight we would recommend adjusting the counting grid size to the bounding box of the point cloud. The smaller memory footprint of a fitted grid would allow finer counting grid resolutions. Map of Austria courtesy of GinkgoMaps [Gin].

CHAPTER 3

Progressive Rendering of Unstructured Point Clouds

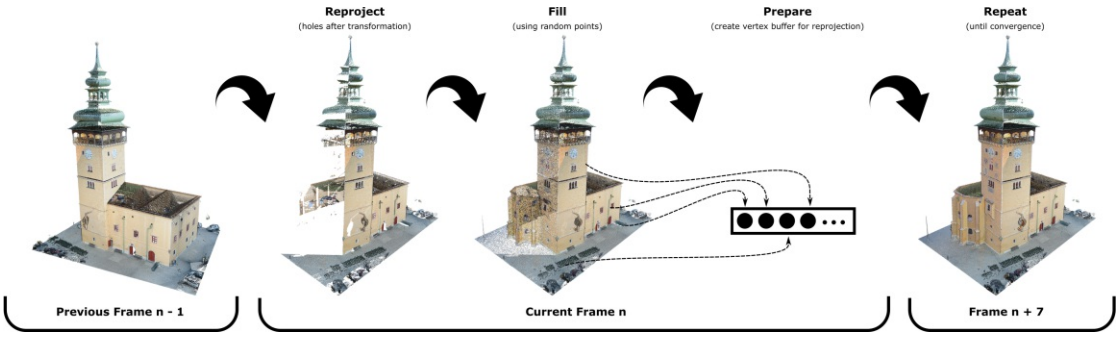


Figure 3.1: The progressive rendering of point clouds via reprojection and filling allows us to maintain real-time frame rates by distributing the rendering of large point clouds over multiple frames, without the need to generate acceleration structures in advance. Filling holes with randomized subsets of the full data set leads to higher quality convergence patterns, and the duration to convergence can be adjusted by the amount of random points that are rendered in the fill pass. *Retz* point cloud courtesy of *Riegl*.

The contents of this chapter were initially published as a paper *"Progressive Real-Time Rendering of One Billion Points"* in Computer Graphics Forum, and presented at EUROGRAPHICS 2020 [SMOW20].

3.1 Introduction

Rendering arbitrarily large point clouds requires hierarchical acceleration structures, but the generation of these structures is slow and time-consuming. Chapter 2 already introduced a method to improve the generation of such structures from up to around 1M points per second to up to around 10 million points per second, but this still requires users to wait tens of seconds or minutes before being able to look at and navigate through hundreds of millions of points. In this section, we introduce a method that allows rendering any point cloud that fits in GPU memory in real time, without the need to generate hierarchical acceleration structures at all. Our implementation is capable of loading unstructured point clouds at rates of up to 100 million points per second, and to render already loaded points in real time while remaining points are still being loaded. This enables users to instantly start looking at data sets with hundreds of millions of points, with 100M points being loaded after the first second, and the remaining data being loaded and simultaneously rendered in real time. In practice, loading from the widely used LAS format caps at 37M points per second in our current implementation, but we believe that there are opportunities for improvements, especially with upcoming direct load technologies (disk to GPU, without detour through CPU and system memory).

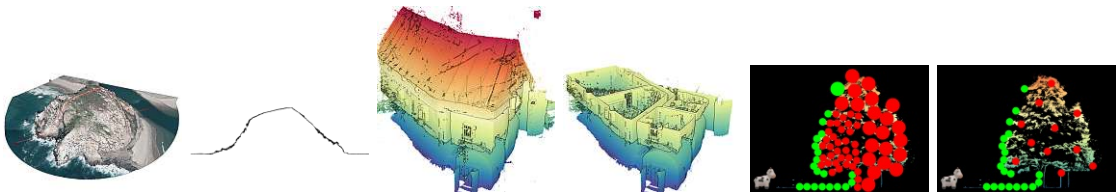
Our progressive rendering approach also addresses the issue of handling point-cloud data with a large number of attributes. Most point clouds contain at least an XYZ coordinate and either a color value or a scalar value with various meanings. This basic format consumes at least 16 bytes per point. However, some use cases require a large amount of additional per-point attributes. Possible attributes include intensity, reflectance, classification, return number, scan angle, GPS-time, echo ratio, beam direction, surface normals, etc., which can increase the storage requirements to more than 100 bytes per point. Storing all these attributes negatively affects load-, processing- and rendering times, even if only a small amount of attributes is actually needed. We realign the data from the commonly used array-of-structs layout into the struct-of-arrays layout, which allows us to only stream the currently needed attribute data to the GPU, thereby increasing the amount of points we can store in GPU memory, as well as reducing the time to stream the data from CPU to GPU memory. Section 3.3.5 describes this issue in detail.

The basic idea of progressive rendering for point clouds is that rather than rendering all points in a single frame, we are going to distribute the rendering over multiple frames. In each frame, part of the full data set is drawn and previously rendered details are preserved by reprojecting the previous frame to the current one. This method ensures real-time frame rates at all times while at the same time converging to the full result over the course of multiple frames.

The targeted use case of this method are workflows in which users want to look at moderately large (hundreds of millions of points) point clouds without the need to wait minutes until LOD structures are generated. An additional requirement is that point clouds with a relatively large amount of attributes, e.g., 50 attributes with around 100

bytes per point, are supported as well. Applications that we believe might benefit from our method include OPALS [OPA] and LAStools [LAS] because they process and produce point clouds as intermediate processing results, or CloudCompare [Clo] because it is a viewer that supports a large variety of non-hierarchical point cloud formats.

A beneficial side effect of progressive rendering is that it is advantageous for data sets with high depth-complexity. Depth-complexity denotes the amount of occluded surface layers in a given viewpoint, as shown in Figure 3.2. Even though only the front-most layer is displayed, point cloud renderers also need to process and render the hidden layers because it's not known in advance whether the points will be visible or not. When rendering with LOD structures, a high depth complexity requires a correspondingly higher point budget to achieve the same level of detail. For example, a point budget of 1 million is sufficient to render 1 million points with no depth complexity, but if there are 9 occluded layers (e.g. additional rooms behind a wall) then the point budget has to be increased to 10 million to obtain the same amount of detail. Progressive rendering, on the other hand, will converge to a full detail image without the need to increase the point budget. To compare with the example before, we can keep rendering just 1 million points per frame, but if there are 10 million points, it will take 10 frames rather than 1 frame until the image converges.



(a) Low Complexity (b) High Complexity (c) LOD (d) Progressive

Figure 3.2: Depth-complexity denotes the amount of occluded surface layers in a given viewpoint.

Our main contributions are as follows:

- We introduce a progressive method that renders point clouds that fit in memory in real time without hierarchical structures, tested with up to one billion points.
- In each frame, our progressive method fills holes by rendering a random subset of the point cloud, which results in a relatively uniform and pleasant convergence to the full image.
- We show how to create these random points incrementally and in parallel on the GPU using a prime number based pseudo-random number generator that generates unique integer values in a given range.
- Our method allows real-time rendering of already loaded data, while remaining data is still being loaded from disk.

- It achieves disk to GPU transfer rates of up to 37M points/s or 1GB/s for the widely used LAS point cloud file format, and up to 100M points/s or 1.6GB/s for a simple binary file that matches the GPU vertex buffer format.

- It is tailored to and supports point clouds with large amounts of attributes – tested with up to 50 attributes and 107 bytes per point.

While our method allows users to render any point cloud that fits in GPU memory in real time without preprocessing, it does not allow users to render point clouds that are larger than that. Low end devices with little GPU memory will have to use out-of-core structures, instead. However, many point cloud viewers currently show raw unstructured data that require about the same amount of GPU memory as our progressive method. Regarding performance, our method adds a certain overhead of up to as many points as there are pixels, but any point cloud that is considerably larger than this overhead will perform better with our method, even on low-end devices.

3.2 Related Work

Previous work related to our method includes shuffling algorithms, methods that rely on random sampling of 3D data, progressive – or incremental – rendering algorithms in various domains, especially those using reprojection, and also hierarchical rendering algorithms for large point clouds. While we do not use hierarchical structures, or any other spatial acceleration structure, we consider these to be related because they are, to the best of our knowledge, the only other option to render large point clouds at rates higher than 60 frames per second.

Randomization and random sampling algorithms are widely used to subsample large amounts of data quickly without producing potentially distracting regular sampling patterns. Stamminger and Drettakis [SD01], and Deussen et al. [DCSD02] create point based representations of meshes by randomly sampling the surface. Rendering subsets or prefixes of these random samples amounts to a de facto continuous level of detail representation. Similarly, Wand et al. render large scenes by rendering a sufficient amount of random point samples rather than the full triangle data [WFP+01]. The data used in our method is similar to these methods in that we also create and use point sets with no particular order. Differences are that our data sets contain only unique data, i.e. no multiple instances of the same object, so we can not optimize for this case, and that we render it in a way so that any point cloud that fits in GPU is rendered with full detail. Oosterom describes the use of random numbers as a basis for continuous level of detail [vO19]. Van der Maaden [Jip19] and Schütz et al. [SKW19] randomly subsample precomputed discrete levels of detail hierarchies of points at runtime. A subsample with continuously decreasing density is then selected from this hierarchy by chosing points based on the distance to the viewer, the local density or spacing of a point, and a random number.

Temporal coherence denotes the similarity of a scene or rendered image over time. Badt [BJ88] already suggested to take advantage of temporal coherence in ray tracing by reprojecting the pixels in the previous frame to the current one. This saves a considerable amount of work because most of the surfaces that are visible in the current frame were already visible in the previous frame. Walter et al. [WDP99] proposed a point-based structure for reprojection, the Render Cache, which is used to reproject previously rendered data to the new frame, and to keep track of possibly outdated regions of the image that need to be updated. The goal of the Render Cache is to maintain interactive frame rates during motion or when editing a scene in ray or path tracers, but to make sure the frame eventually converges if there is no further change to the scene or camera. Jevans [Jev92] exploits temporal coherence in object space by tracking objects that move so that only animated parts of the scene need to be retraced. In a state-of-the-art report on temporal-coherence methods from 2011, Scherzer et al. [SYM+11] discuss a wide range of algorithms that exploit coherence, with a special focus on reprojection algorithms

Tredinnick et al. [TBP16] and Ponto et al. [PTC17] proposed a progressive rendering method for point clouds that is related to ours. The previous frame is reprojected to the current one and holes are filled by rendering additional points. Their work focuses on hierarchical methods, however, and in each frame they render a different set of octree nodes within the view frustum. Even though only part of the data is rendered in each frame, the image converges to the full amount of detail after a few frames. Our method differs in that we focus on progressively rendering unstructured data for which no hierarchical structure was generated in advance, which allows us to look at unstructured data up to two orders of magnitudes faster than methods that require hierarchical structures. Another similar technique by Futterlieb et al. [FTB16] renders cached results during movement and accumulates details when the camera does not move.

3.3 Progressive Rendering

In the context of our method, progressive rendering means that we distribute the task of rendering the full point cloud over multiple frames, instead of doing all the work in a single frame. The goal is to maintain real-time frame rates and keep the application responsive at all times. The basic idea to achieve this goal is to reproject the previous frame, since most of the previously visible points are likely to be visible again in the current frame, and then fill holes that appear due to disocclusions with randomly selected additional points to obtain a high-quality convergence behavior. Over the course of multiple frames, the result converges to an image of the full model. The number of randomly selected points to fill holes is referred to as the *point budget*, similarly to hierarchical methods where it refers to the number of points that are selected form the hierarchy and rendered in a frame. In our progressive method, the budget can be adjusted to favor performance (low budget) over faster convergence to the full image (high budget). The number of points that are reprojected are not included in the budget because the reprojection has a fixed cost that can not be adjusted.

In this section, we will describe the necessary data structures, how we load points, an efficient way to incrementally shuffle points during loading, the actual rendering pipeline, how we adaptively select the point budget to minimize the duration to convergence while maintaining real-time frame rates, and how to switch between different point attributes.

3.3.1 Data Structure

Our method employs two data structures in order to stream new attribute data to the GPU, and to quickly render a certain amount of random points in each frame.

On the CPU side, point attributes are stored in a struct-of-arrays fashion, i.e., one array stores exactly one attribute: $[RRR][GGG][BBB]$. This allows us to stream specific attributes from CPU to GPU with minimal usage of memory bandwidth, since accessing a value of an attribute array will load a whole cache line of subsequent values into the CPU cache [Dre07]. An interleaved array, on the other hand, would result in loading various different attributes of a point into the CPU cache, which is not useful if only one attribute of a point is needed. This is important because we keep attributes in their original form (e.g doubles or 64-bit integers) in CPU memory and only transform an attribute to a GPU friendly format (e.g. floats) when we switch to it. The struct-of-arrays memory layout reduces the required memory bandwidth during transformation from up to the full size of the point cloud to $attributeSize \cdot numPoints$ bytes. The reason we keep attributes in their original form is that many of them are stored in a format that is not directly useful for rendering, but all of the data they contain may be important. RGB data, for example, requires 2 bytes per channel in our test data sets, and preemptively reducing it to 1 byte each for rendering purposes results in loss of data that may be needed at a later time.

On the GPU side, we use a shuffled interleaved vertex buffer with 16 bytes per point as our rendering data structure, which is created by inserting points at pseudo-random locations. Due to the maximum buffer size of 2^{31} bytes on modern GPUs, the shuffled Vertex Buffer Object (VBO) may actually consist of multiple buffers – one for every $2^{31}/16 \approx 134$ million points. Shuffling is done because it reduces the problem of rendering a batch of N random points to rendering N consecutive points. Each point contains 12 bytes for XYZ coordinates, and another 4 bytes for attribute data. The attribute data may contain a single 4 byte float, or four unsigned bytes. The former is used to visualize single scalar attribute values, and the latter is used to visualize vectors of attributes, such as colors and normals. It is up to the vertex shader to interpret the data as needed. Newly loaded batches of points or new batches of attributes are not directly uploaded to the shuffled VBO. Instead, they are uploaded to a separate *Distribute* buffer that holds a single batch of 500k points. A compute shader then inserts the points or attributes to the respective shuffled location in the VBO. The *Distribute* buffer receives 16 byte XYZRGBA during the initial loading from disk, but only 4 bytes per point, i.e., just the attribute data, when switching to a new attribute. Finally, the *Reproject* buffer contains all the points that are visible at the end of a frame. In addition to position and attribute

data, it also stores the index of that point inside the shuffled VBO, which is needed during reprojection to write the point indices along with point colors to the framebuffer.

3.3.2 Loading

One of the objectives of our method is that intermediate results are shown in real time while remaining data is being loaded. In order to achieve this, files are loaded and transformed to GPU-ready buffers in parallel, and the task of the main thread is simplified to sending batches that are ready to the GPU. Figure 3.3 illustrates this process in a time line. The load thread is dedicated to reading binary data in batches of 500k points from disk. Three additional parser threads transform the binary batches and separate the interleaved point data into one array per attribute, which are then appended to the struct-of-arrays structure in main memory. During the start of the next frame, the main thread sends the XYZRGBA attribute of all batches that were fully loaded and parsed in the previous frame to the GPU. The composite XYZRGBA array is a special case that gets assembled by the parser threads after all other attributes are stored in separate arrays, because this is the initial data that we send to the GPU.

Note that the full-sized vertex buffer is allocated at the beginning and that vertices will be default-initialized until all points are loaded. These points would be rendered as black points at location $(0, 0, 0)$. We suggest to discard default-initialized points during rendering until the data set is fully loaded, especially since drawing millions of points at the same pixel is more expensive in terms of performance than drawing the same number of points distributed across the screen.

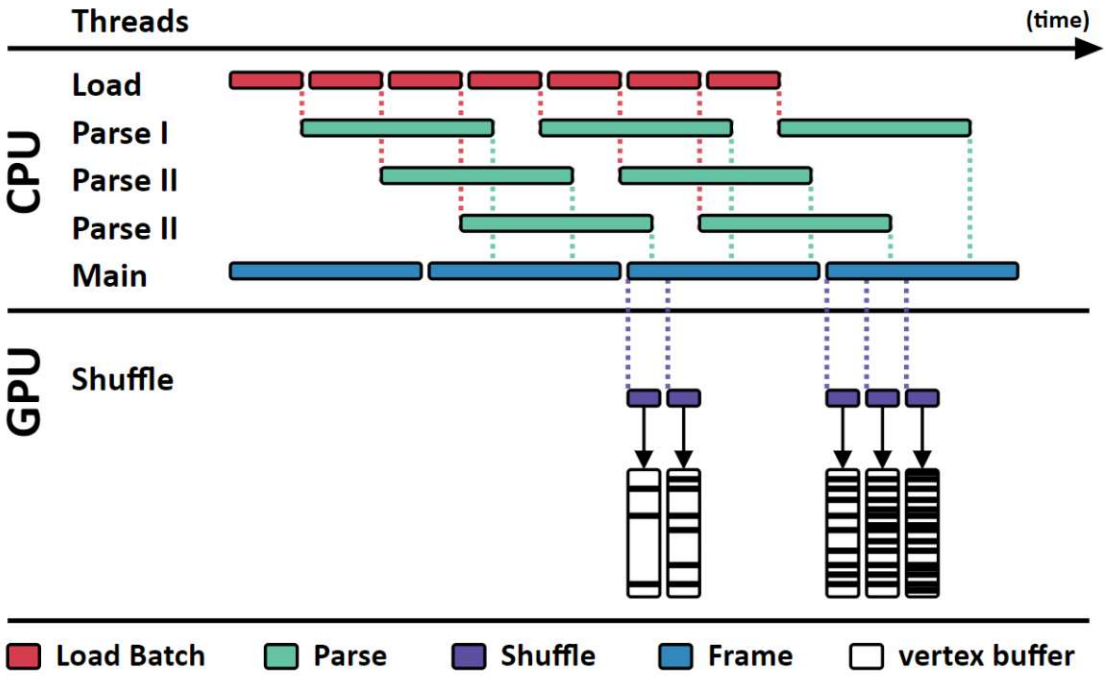


Figure 3.3: The load pipeline. One thread is dedicated to reading batches of binary data from disk. Three threads are used to transform the loaded binary batches into the structure-of-arrays memory layout. At the start of each frame, the main thread uploads fully parsed batches to the GPU and executes a compute shader that moves each point to its shuffled location in the vertex buffer.

3.3.3 Incremental Parallel Shuffling

Rendering randomly selected points improves the perceived visual quality during convergence to the final image, compared to rendering points in their original and potentially sorted order. Points are shuffled during loading so that we can efficiently render N random points by rendering a subset of N consecutive points from the vertex buffer. Since we want to display the points with our progressive method while additional points are still being loaded from disk, we need to use a shuffling method that is capable of incrementally shuffling points as they become available. We use the approach described by Preshing to compute a permutation of a sequence of numbers $[0, ..., P-1]$, where P is a prime that is congruent to 3 (mod 4) [Pre12]. This approach maps each number in the sequence to another number of the same set without collisions, i.e. there will be no duplicates. In our case, we assume the input to be the index of the point in the original array of points, and the output to be the position of the point inside the shuffled array. This allows us to directly copy the points to their position inside the shuffled array with a compute shader without the need to synchronize between threads. The permutation function is given by:

$$\text{permute}(i) = \begin{cases} i^2 \bmod P, & \text{if } i \leq \frac{P}{2} \\ P - i^2 \bmod P, & \text{if } i < P \\ i, & \text{otherwise} \end{cases} \tag{3.1}$$

The last case covers point clouds where the number of points N does not equal a suitable prime. In this case, we find the next smaller prime $P \leq N$, shuffle all the points in that range, and leave the remaining points unshuffled. The number of unshuffled points is negligible because the gap between consecutive primes is small. The largest gap between two consecutive primes $P \equiv 3 \,(\text{mod}\, 4)$ for up to 500 million points is 532, between primes 184 007 671 and 184 008 203. This means that for up to 500 million points, at most 532 may not be shuffled. Trying to shuffle them as well results in extra work with insignificant improvement. Alternatively, one could find the next larger prime, shuffle the entire data set, and leave a negligible amount of vertex buffer elements empty.

A disadvantage of the prime number-based method is the relatively low quality of the permutation after only one pass, which manifests as noticeable patterns, as shown in Figure 3.4. Since Equation 3.1 is bijective – each element of the input set $[0, 1, 2, ..., P-1]$ maps to exactly one distinct element of the same set – we can simply apply it multiple times and still obtain the same number of unique target indices. Our final shuffling function is therefore given as:

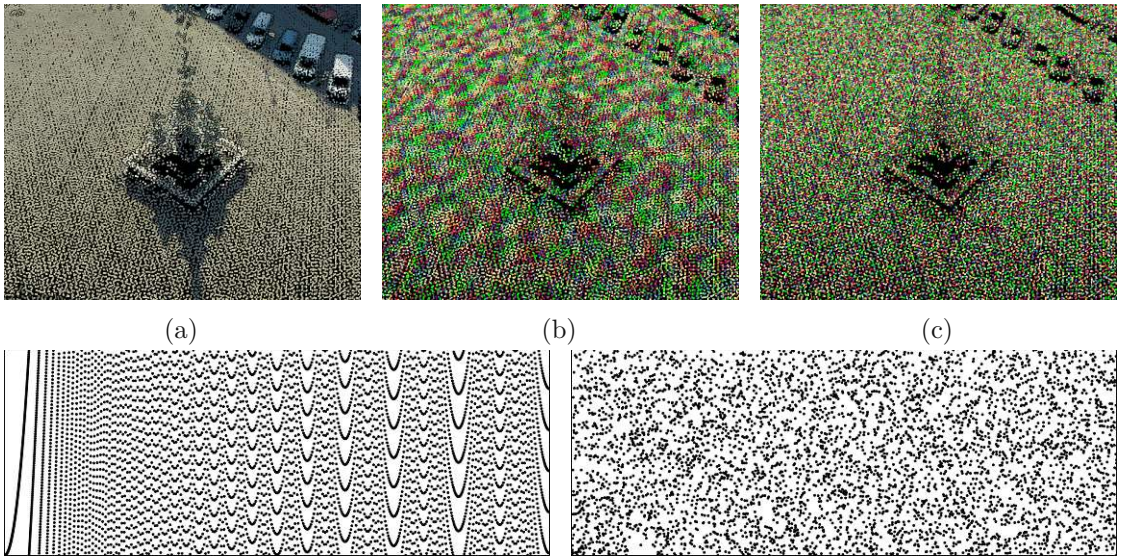
$$\text{targetIndex}(i) = \text{permute}(\text{permute}(i)) \tag{3.2}$$

Applying permute twice results in a randomness that is not necessarily of high quality, but sufficiently random for our progressive rendering method. With "not high-quality" we mean that there are certain patterns, and some random numbers may be predictable from previous random numbers. For example, Equation 3.1 is monotonically increasing for the first $\sqrt[2]{P}$ numbers and Equation 3.2 is monotonically increasing for the first $\sqrt[4]{P}$ numbers. The former is immediately obvious in Figure 3.4 (d), and the same patterns are visible repeatedly throughout the function graph. The latter is not noticeable in Figure 3.4 (e). As long as patterns aren't immediately obvious visually, we consider a random number generator sufficiently random for our method.

The big advantage of the prime number-based method over other methods like the Fisher-Yates shuffle is that it can be applied to each input index i individually, without depending on the state from previous calculations and with no collisions. It is therefore inherently parallelizable and can be implemented in a compute shader on the GPU without synchronization between threads.

3.3.4 Rendering Pipeline

The progressive rendering method reprojects the previous frame to the current frame, and then fills in missing data by rendering a certain number of random points. Over



(a) (b) (c)

(d) $f(i) = permute(i)$ (e) $f(i) = permute(permute(i))$

Figure 3.4: (a) Points colored by RGB. (b+c) Points colored by indices. (b+d) One pass of Equation 3.1 produces noticeable patterns in the mapping of input indices to target indices – some patches of points preserve locality after the shuffle. (c+e) Applying Equation 3.1 twice results in a sufficiently random permutation of input to target indices.

the course of multiple frames, the result will converge to the same image that we would get by rendering all points at once, not accounting for render order and z-fighting issues. This method is realized in three render passes:

1. **Reproject**: Render all the points that were visible in the previous frame, reprojected to the current frame.
2. **Fill**: Render a batch of random points to fill holes. This is done efficiently by rendering subsets of a shuffled vertex buffer.
3. **Prepare**: Create a new vertex buffer from all points that are visible in the rendered image. This vertex buffer will be used in pass one of the next frame.

The idea of reprojection is that most of the data that was visible in the previous frame will also be visible in the current frame, so it may make sense to reuse it. However, during movements, previously occluded parts of the surface and parts that were outside of the frustum may become visible, but since this data is missing from the previous frame, holes and empty regions will appear, as shown in Figure 3.1. The *Fill* pass attempts to fill missing areas by adding random points. We chose to fill using randomly selected points because it results in a relatively uniform convergence to the final result over the whole image with no apparent pattern, and because it looks relatively pleasant compared to

filling with sorted chunks of points. If points are in some way sorted or structured, it will result in unpleasant flickering artifacts during motion because in each frame, parts of the image will fully converge, while other parts will see no progress at all until later frames.

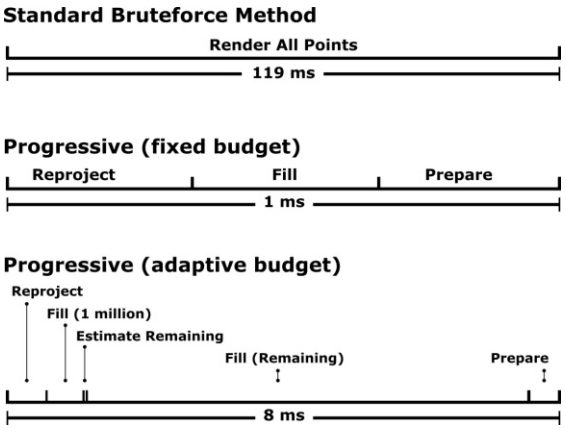
Depending on hardware capabilities, we render S random points with $S \in [1M, 30M]$ each frame during the *Fill* pass. The selection of random points is achieved by rendering subsets of the shuffled vertex buffer. In the first frame, points from $[0, S)$ are rendered, then in the next frame $[S, 2 \cdot S)$ and so on. Once we have looped through all the points in the vertex buffer, we repeat again from the beginning. Without camera motion, the image converges to the full result after looping through all points once. During motion, the whole buffer must be repeatedly looped through in order to keep filling new holes.

The third and final pass – *Prepare* – creates the *Reproject* vertex buffer out of all currently visible points, which is then used in the *Reproject* pass of the next frame. Note that instead of reprojecting the points directly from the framebuffer, which would lead to inaccuracies, we identify the original point projected to a pixel and reproject it from its original coordinates. This requires that the *Reproject* and *Fill* passes both write point indices into an additional index-color attachment on the framebuffer. A compute shader iterates over all pixels, reads the point indices from the index-color attachment, and copies the respective points from the shuffled vertex buffer into the *Reproject* vertex buffer. In addition to XYZ and the 4-byte attribute value, all points in the *Reproject* buffer also store the shuffle point index, which is needed by the *Reproject* pass to write the correct index of a point inside the shuffled vertex buffer into the index-color attachment.

In our implementation, multisample anti-aliasing (MSAA) effectively acts as supersampling. The *Prepare* pass makes no distinction between pixels or MSAA samples, and may therefore produce one point per sample for up to $msaa\_samples \cdot pixels$ points. The next *Reproject* pass then renders all of them. A 1920x1080 framebuffer without MSAA may therefore reproject up to about 2 million points. With 4xMSAA it will be four times as much, about 8 million points.

Adaptive Fill Budget

A basic implementation of the *Fill* pass renders a fixed amount of random points to fill gaps, e.g., 1 million points. This is not optimal because the *Reproject* and *Prepare* passes use up render time to preserve detail, but only the *Fill* pass drives progress towards convergence. In order to improve convergence times, we need to maximize the number of points rendered during the *Fill* pass while maintaining real-time frame rates. The following figure shows the ratio of time spent on passes of a standard brute-force approach, a basic fixed fill-budget approach, and an adaptive fill-budget approach with improved ratio of basic overhead to progress.



The timings were evaluated with a data set comprising 302 million points. The brute-force approach significantly exceeds the limit of 16.6ms. The fixed fill budget approach with a budget of 1M points is well below the limit but has little progress per frame, and only 33% of the time is spent on the *Fill* pass that drives progress. The adaptive fill budget renders a fixed amount of points first, and then an estimated additional amount that can be rendered in the remaining time based on the time it took to render the fixed amount. It spends 90% of the rendering time on filling holes and progressing towards convergence.

Since render times vary greatly depending on viewpoints, we cannot reliably use past frames to estimate the adaptive fill budget for the current frame. Instead, we measure render timings of the current frame directly on the GPU and then estimate the number of additional points we can render. If 60fps are desired, the frame must be fully rendered within $\frac{1}{60}s = 16.6$ms. To estimate an adaptive fill budget, we measure the time since the beginning of the frame, and the time it took to render the first 1 million points. From this, we compute the number of rendered points per millisecond, which we then multiply by the milliseconds we have left to finish the frame. Due to the margin of error of this estimate and time consumed by additional render passes and GPU tasks, we suggest to assume the available time to be well below 16.6ms, e.g., around 10ms.

An advantage of the adaptive fill budget is that it implicitly accounts for points that are outside the view frustum. While rendering the first 1 million points, points that are outside the view frustum take less time to render and the adaptive budget will be correspondingly larger. For the subsequently rendered remaining points, due to the randomization, roughly the same percentage will be outside the view frustum. The first step essentially provides a representative percentage of the amount of points that will be outside the view frustum during the second step. Due to this, the adaptive budget can vary from 20 million points per frame in viewpoints where all points are within the view frustum, up to 100 million points for close-up viewpoints within the point cloud, when a large portion of points is outside the view frustum (Measured on an RTX 2080 Ti).

Implementations of an adaptive budget have to take care to avoid CPU-GPU sync-points. In OpenGL, we suggest to use timer queries that write timestamps directly into GPU

buffers, and use compute shaders to estimate the remaining budget directly on the GPU. The compute shader then writes the estimated number of points we can render in the remaining time into another buffer that is used as an argument to an indirect draw call.

Convergence

Progressive point-cloud rendering methods distribute the rendering process over multiple frames until the result converges to the final image. We can calculate the number of frames until convergence, but the actual time to convergence has to be measured. We can also differentiate convergence behaviour that progresses in localized batches, or uniformly over the whole model, as shown in Figure 3.5. The advantage of convergence in localized batches is that the GPU often (but not always, see Table 3.4) renders vertices faster if they are sorted by locality. The disadvantage, however, is that it results in severe flickering artifacts since some regions converge immediately, and others don't converge at all until later frames. Rendering randomly selected points may be slower, but it results in a uniform and pleasant convergence over the whole model.

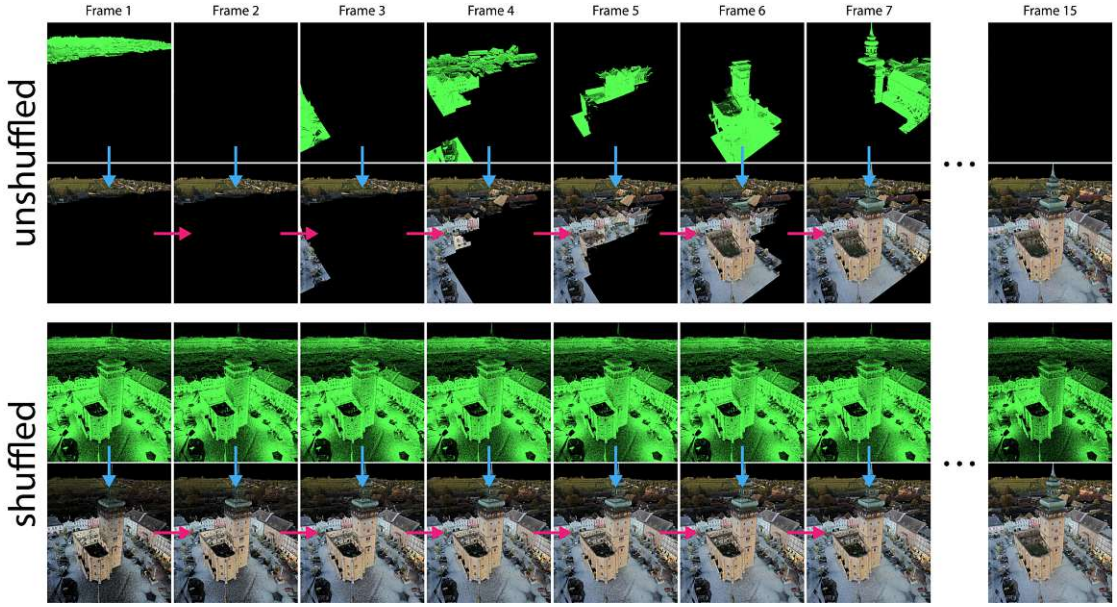
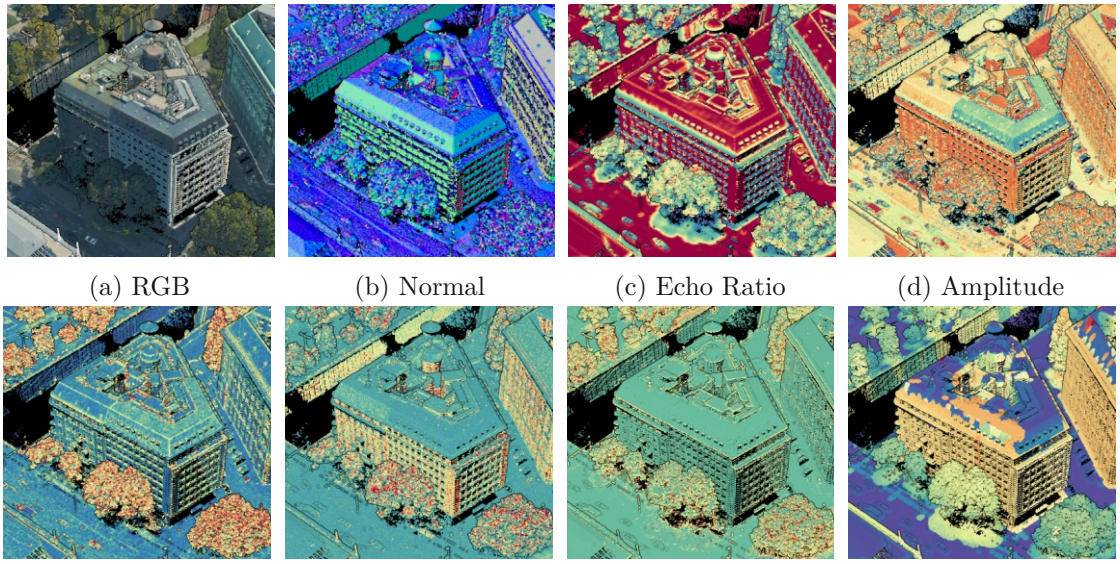


Figure 3.5: Convergence behavior of unshuffled and shuffled point clouds. First and third rows: Varying subsets of 10 million points that the *Fill* pass will render in that frame. Second and fourth rows: The image that is displayed to the user after reprojecting the previous frame to the current one, and filling missing data with the selected subsets. The unshuffled version renders localized batches, and sometimes no data at all if the subset is completely outside the view frustum. The shuffled version quickly covers the entire screen. Both versions converge to the same image after 15 frames.

When motion stops, the framebuffer converges after all points were rendered at least once



(a) RGB (b) Normal (c) Echo Ratio (d) Amplitude

(e) Sphericity (f) Deviation (g) Return Nr. (h) vRange

Figure 3.6: Various attributes of a point cloud. Each attribute increases the bytes per point, which in turn increases memory requirements. We keep only one attribute at a time in GPU memory to maximize the number of points we can store. New attributes are streamed to GPU and replace old ones on demand. *Vienna* point cloud courtesy of *Riegl*.

during the *Fill* pass, i.e., after $\frac{\#points}{budget}$ frames. With a fixed fill budget of 1 million points, it will take 100 frames to finish rendering 100 million points. The actual time until convergence isn't easily predictable, since it varies between GPUs, number of points in view-frustum, number of overlapping points, etc. The rate of time spent on the *Reproject*, *Fill* and *Prepare* passes also affects time to convergence, since only the *Fill* pass keeps progressing further whereas the other two passes consume time without driving progress. If all three passes consume the same amount of time, we end up with one third of the available performance spent on progressing to convergence. However, if 0.19 ms are spent on reprojection, another 0.34 ms on preparing a new vertex buffer, and 3.34 ms on filling holes, then we are utilizing $3.34/(0.19 + 3.34 + 0.34) \approx 86\%$ of the render time towards progressing to convergence while still maintaining real-time frame rates (timings taken from Table 3.3).

3.3.5 Streaming Point Attributes

Our data sets consist of point clouds with hundreds of millions of points, and up to 50 different attributes for up to 107 bytes per point. Figure 3.6 shows various attributes that a point cloud can contain. Assuming 10GB of GPU memory and 107 bytes per point, we could store at most $\frac{10*1024^3}{107} = 100M$ points on the GPU. In most cases, we only need the coordinates plus one to 4 attributes at any time, so the remaining attributes consume memory unnecessarily. In addition to that, our rendering pipeline is also strongly affected

by memory bandwidth because vertex buffers are recomputed each frame. The more bytes a vertex has, the slower it will be to generate a new vertex buffer. Due to this, we only keep 16 bytes per point in GPU memory, comprising of $3 \cdot 4 = 12$ bytes for the XYZ coordinates and another 4 bytes encoding one to four attributes. This allows us to store up to $\frac{10*1024^3}{16} = 671M$ points in 10GB of GPU memory, although the actual amount will be lower since other parts of the application, as well as other applications and the OS, also require some GPU memory.

In order to be able to visualize all available attributes, we keep them in main memory and stream them to the GPU once a user asks to see another attribute. At the start of each frame, the main thread sends multiple batches of attributes to the GPU. A compute shader distributes the vertex attribute data to the respective vertices with the same pseudo-random target index computation that is used during the initial loading step, thereby overriding the previous vertex attribute data. Streaming a new attribute from CPU to GPU happens at rates of 300 to 800 million points per second on an RTX 2080 Ti, depending on whether the attribute is a single scalar or a vector of up to 4 values. For the *Vienna* data set with 124 million points, switching attributes takes 0.155 to 0.356 seconds. While a new attribute is streamed, the application remains interactive, but the amount of attribute batches that are uploaded to the GPU in each frame increases frame times up to 200ms. However, implementations may place a limit on how many batches they will upload in each frame in order to maintain real-time frame rates. Similarly as during the initial upload, points keep their original value until they are overwritten by new data over the course of multiple frames. In the case of the initial loading of point data, points that are not yet loaded are rendered but discarded inside the vertex shader. In the case of attribute streaming, points that are not yet overwritten are rendered with their previous attribute value. This can turn the not yet overwritten attribute values into visual noise because they will also be rendered with the shader parameters of the new attribute. We did not attempt to fix this issue at this time because transitions from one attribute to another are relatively fast.

Since attributes may not fully fit inside the 4 bytes that are available on GPU-side, we let users specify suitable transformations from the original attribute range to a range of $[0, 255]$ per channel in case of vectors, or $[0, 1]$ in case of scalars. This allows users to inspect large attributes like GPS-Time (double) in full precision by re-transforming from the full-precision data with a different range, if needed.

3.4 Evaluation

In this section, we provide background information on the widely used LAS point-cloud file format that we use, followed by an introduction to our test data sets, and conclude with an evaluation of load and rendering performances.

(a) Heidentor (b) Retz (c) Arbegen (d) Vienna (e) Morro Bay

Figure 3.7: Data sets used in our evaluation. The same viewpoints were used in the rendering benchmarks in Table 3.3.

3.4.1 LAS File Format

Two of the most widely used binary point-cloud file formats are LAS [ASP19] and its compressed counterpart, LAZ [Ise13]. We use the LAS format due to its compatibility with a wide range of applications, and because the simple but strict file format makes it easy to develop custom file loaders. The LAS format stores points in an interleaved fashion, i.e., each point is stored one after the other. Points consist of a set of attributes, and various predefined point data record formats (enumerated by 0, 1, ...) describe which combination of attributes are stored. Some attributes, such as XYZ, intensity, return number and classification are present in all available formats, whether they are needed or not. Others are only present in specific formats, such as RGB in formats 2 and 3, or GPS-time in 1 and 3. Since version 1.4, the spec also allows a standardized definition of custom extra attributes in addition to the fixed set of attributes. Our point-cloud application makes heavy use of these extra attributes, and some of our data sets use 50 attributes that require up to 107 bytes per point. The large amount of information for each individual point leads to challenges such as increased memory requirements and memory bandwidth usage.

3.4.2 Data Sets

Screenshots and descriptions of our test data sets are provided in Figure 3.7 and Table 3.1. Vienna and Morro Bay were captured with airborne laser scanners that provide a relatively uniform but low density over a large area. Retz was scanned with a combination of airborne and terrestrial laser scanning. The former provides a low-density model of the town and the surrounding area, and the latter augments it with higher detail at the center of the town.

3.4.3 Performance

This section lists and discusses performance results for loading unstructured data from LAS files, how to achieve 100M points per second by loading from an optimal file format, performance of the rendering pipeline, and a comparison to hierarchical structures.

The performances were evaluated on following test systems:

Data Set	Size	#points	#attributes	bpp	pt/m^2	km^2	Acquisition Meth.
Heidentor	0.7 GB	25.8 M	12	26	-	-	Photogrammetry
Retz	4.9 GB	145.5 M	13	34	56.43	2.58	ALS + TLS
Arbegen	6.7 GB	258.0 M	12	26	-	-	TLS
Vienna	29.6 GB	276.7 M	50	107	57.52	4.81	ALS
Morro Bay	13.8 GB	407.0 M	13	34	22.06	18.45	ALS

Table 3.1: Key parameters of used data sets. ALS: Airborne Laser Scanning. TLS: Terrestrial Laser Scanning. bpp: bytes per point.

- **2080**: A desktop system with an AMD Ryzen 2700X CPU, an NVIDIA RTX 2080 Ti with 11GB GPU memory, a 1TB Samsung 970 PRO SSD, and 32GB RAM.
- **1660**: A notebook system with an Intel i7-9750H CPU, an NVIDIA GTX 1660 Ti Max-Q with 6GB GPU memory, a 256GB SSD, and 16GB RAM.
- **940**: A notebook system with an Intel Core i7-7500U CPU, an NVIDIA 940MX with 2GB GPU memory, a 1TB SATA HDD, and 16GB RAM.
- **Titan**: A desktop system with an NVIDIA RTX Titan with 24GB GPU memory, and 24GB RAM.

We confirmed the claim that our method works for up to one billion points within an expected margin of performance during temporary access to a system with an NVIDIA RTX Titan with 24GB GPU memory. The data set is fully loaded in around 25 seconds, and an average adaptive fill budget of around 20 million points per frame is rendered in real time.

Loading LAS Files

In this section, we discuss the time it takes to *Load* LAS files from disk, and the time it takes to fully parse and *Upload* the data to the GPU, as shown in Table 3.2. Loading from disk and uploading to the GPU happen in parallel, as shown in Figure 3.3. The timings for the latter are therefore the total time of doing both. The *Upload* column shows that parsing and uploading finishes tenths of a second after the last batch of binary data was loaded from disk. For these benchmarks, we deactivated OS-level file caching under Microsoft Windows by calling CreateFile with the FILE_FLAG_NO_BUFFERING flag on the LAS file, before loading it with the standard C FILE API using fopen.

Loading 100 Million Points Per Second

The evaluated LAS load performance is limited in bandwidth for multiple reasons: First, all LAS formats store various attributes that may not actually be needed. Second, the interleaved (Array-of-Structures) layout needs to be transformed to a Structure-of-Arrays layout during loading to allow for efficient switching between attributes. And third,

Model	Points	File Size	bpp	Hierarchical			Progressive			
				Converter	Duration	Points/s	Format	Load	Upload	Points/s
Heidentor	25.8 M	0.7 GB	26	Potree	36 s	0.72 M	LAS	0.56 s	0.70 s	37.18 M
				Entwine	40 s	0.60 M	VBO	0.30 s	0.41 s	63.48 M
				Arena4D	43 s	0.65 M				
Vienna	276.7 M	29.6 GB	107	Potree	611 s	0.45 M	LAS	32.75 s	32.92 s	8.40 M
				Entwine	301 s	0.92 M	VBO	3.36 s	3.36 s	82.34 M
				Arena4D	306 s	0.90 M				
Morro Bay	407.0 M	13.8 GB	34	Potree	776 s	0.52 M	LAS	14.31 s	14.36 s	28.35 M
				Entwine	564 s	0.72 M	VBO	4.05 s	4.05 s	100.45 M
				Arena4D	391 s	1.04 M				

Table 3.2: Load Performance. Time needed to create a hierarchical LOD structure from LAS files in advance, compared to the time needed to directly load and render the non-hierarchical data with our progressive method. *Points/s*: Number of points per second processed. *LAS*: Load, parse and upload LAS files. *VBO*: Load and upload files in the same format as the vertex buffer. All benchmarks are done on system *2080* and its NVMe SSD.

attributes are encoded in a way that is not directly useful for rendering, e.g., RGB values are stored in 2 bytes each, and coordinates are stored as 32-bit fixed-precision integers in order to maximize the precision of 32-bit values while avoiding the additional disk-space cost of 64-bit data types. During loading, the coordinates are transformed to a floating-point type, usually double values for accurate processing, or origin-centered floats for rendering. An ideal file format with respect to low loading times would need few bytes per point and require little to no transformation of the attribute values.

We are able to achieve load performances of up to 100 million points per second by limiting ourselves to XYZ and RGBA values, storing points in the same format on disk as we use in the GPU vertex buffers, and directly transferring buffers from disk to GPU without any modifications. Coordinates are stored as single-precision floating-point values and colors as unsigned bytes. Each point requires 16 bytes. The resulting transfer rate from disk to GPU is 1.6GB/s. These numbers also include the times for shuffling the transferred data. The achievable read performance of the utilized SSD is 2.5 GB/s according to the UserBenchmark test suite. This puts the disk-to-GPU performance of our implementation (1.6GB/s) at 64% of the theoretically achievable disk-to-RAM performance (2.5GB/s).

Comparison to Hierarchical Structures

One of our claims is that our method allows users to quickly look at large point-cloud data that would otherwise require a relatively slow generation of hierarchical structures. In order to put this claim into perspective, we show how long it takes for state-of-the-art converters Potree [Sch16], Entwine [Enta], and Arena4D [Are] to generate these acceleration structures out of our test data in Table 3.2. The results show that our progressive rendering method can fully load/prepare and display large data sets up to about 100 to 200 times faster than hierarchical methods if the file format matches the vertex buffer format, or about 30 to 50 times faster from LAS files. For example, it takes Potree 776 seconds and Arena4D 391 seconds to build a hierarchical structure out of a LAS file, whereas our method can fully load the same data in 14.36 seconds from a LAS file, or 4 seconds from a vertex buffer formatted file. Not accounted for in these comparison is the time that hierarchical structures then need to load and render the data, or the time that our method needs to converge to the full result. Considering that our method already renders the data while remaining data is loaded, the result will already be close to convergence by the time all the data is loaded, with only the last or last few batches partially missing for another couple of frames.

We acknowledge that this is not an entirely fair comparison because these converters spend time reading data, writing it back to disk and potentially reading it again, whereas our progressive method does not have to write data back to disk. However, the data structures required to generate hierarchical acceleration structures also need additional memory during conversion, so the converters would not be able to hold as many points in memory as our progressive method during the conversion without flushing data back to disk.

Rendering

Table 3.3 shows the performance of our progressive approach, including the time it takes to render a single frame and the time it takes until the image converges. We also compare these timings to a brute-force approach where all the points are rendered in each frame. All measurements are computed as the median over the past 10 frames. Framebuffer resolutions are 1920x1080, roughly 2 Megapixels. Our renderer is implemented in OpenGL and we use the GL_POINTS drawing primitive with a point size of 1 in our benchmarks. For the brute-force approach, shuffling is deactivated for two reasons: First, because brute-force approaches usually render the data in their original order. And second, because in the majority of cases we tested, rendering shuffled data was slower than rendering the same points in their original order that exhibited a certain amount of locality between consecutively stored points. A notable exception is the *Morro Bay* data set, where the shuffled vertex buffer renders faster from the chosen viewpoint in Figure 3.7, with almost all 407 million points located inside the view frustum. Once the user zooms in, the situation reverses and rendering the shuffled data becomes slower again. Table 3.4 illustrates these differences in performance. This difference in rendering times of shuffled and unshuffled buffers contributes to the fact that the convergence times for our progressive method are usually significantly higher than the rendering times of the brute-force approach, except for *Morro Bay* where the duration to convergence can be lower than the time needed to render the unshuffled data set with the brute-force approach.

The 1 billion points claim was evaluated on an RTX Titan with 24GB memory. 17.3GB of GPU memory was in use after the data set was fully loaded. The used data set is a variation of the *Morro Bay* data set that covers a larger area.

The *#reprojected* column shows the number of points that are visible at the end of the frame, and which are subsequently rendered in the first pass of the next frame. The Heidentor exhibits a small number of reprojected points because it covers a small portion of the screen. For the Vienna data set, the number of reprojected points is close to the number of pixels because the data set covers the whole screen in the given viewpoint. The difference between the number of reprojected points and the number of pixels is caused by scan shadows in some regions, and insufficient point density in others. The benchmark of the Morro Bay data set on the 2080 system illustrates that the number of reprojected points is directly proportional to the number of MSAA samples.

A notable observation is made in Table 3.3 regarding the *Vienna* data set on system *1660*. If all 277 million points are rendered, the *Prepare* pass takes almost seven times as long as when only the first 230 million points are rendered. Similarly, the brute-force method requires more than double the time if all 277 million points are rendered, instead of only the first 230 million points. This is because at some point, the GPU will allocate shared system memory instead of dedicated GPU memory to our buffers. Average rendering times for the *Fill* pass do not change much because we render small subsets at a time and because most of the data is still rendered from GPU memory – only a subset that did not

Model	Brute-force	Progressive	Shuffled?
Heidentor	9.69 ms	3.56 ms	yes
	6.87 ms	2.62 ms	no
Vienna	143.27 ms	5.68 ms	yes
	84.84 ms	4.43 ms	no
Morro Bay	150.97 ms	3.99 ms	yes
	295.92 ms	7.59 ms	no

Table 3.4: Performance difference of rendering shuffled and unshuffled vertex buffers. Progressive method rendered with 1x MSAA and a budget of 10M points per frame.

fit is rendered from system memory. The *Prepare* pass, on the other hand, slows down drastically because it now has to also read randomly shuffled point data from shared system memory as well. The total rendering time still remains below 16.6ms, which means even data sets that do not fit in GPU memory can be rendered in real time, but the time to convergence increases by a multiple.

Regarding depth complexity, we tested various viewpoints inside and outside the *Arbegen* point cloud. This data set consists of selected scan positions of the interior and parts of the exterior of a house with multiple rooms, an attic, a cellar and a hallway to the cellar. With the octree system of *Potree*, the point budget has to be increased to values of around 20 to 25 million points per frame to obtain a resolution of 1 point per 2x2 pixels, or as high as 55 million points to obtain a resolution of 1 point per pixel. The reason for this is because points that are hidden behind floors, ceilings, walls and noise have to be rendered due to the lack of occlusion culling. As a result, occluded points consume the majority of the render time, without contributing to the image. In the worst case, 55 million points were rendered but at most 2 million points were visible at a time on a 2 megapixel screen. Our progressive method, on the other hand, converges to the full image with a point budget as low as 1 million points per frame, plus the amount of points that are reprojected.

Virtual Reality

Our progressive method is able to maintain 2×90 frames per second in different viewpoints required by the HTC VIVE, at a resolution of 1448×1608 per eye on an RTX 2080 Ti with 4xMSAA and a fixed fill budget of 3 million points per frame after the data was fully loaded. For the *Vienna* data set, the frame rate targets are also achieved during loading if the fixed fill budget is lowered to one million points. In VR, the entire progressive rendering pipeline is executed twice, once for each eye. Since the frame rate is locked at 90fps and the fill-budget at 3 million points, the image converges at a rate of 180 million points per second. The number of points of the model affects

Model	Points	System	MSAA	Brute-force	Progressive					
					Budget	Reproject	Fill	Prepare	Total	#reprojected
Heidentor	25.8 M	2080	no	6.87	1 M	0.10	0.33	0.15	0.71	392k
		2080	no	6.87	10 M	0.10	3.19	0.15	3.56	392k
		1660	no	10.10	1 M	0.20	0.96	0.29	1.68	392k
		940	no	44.02	1 M	1.27	8.39	2.67	12.72	392k
Vienna	276.7 M	2080	no	84.84	1 M	0.35	0.45	0.62	1.54	1.9M
	276.7 M	2080	no	84.84	10 M	0.35	4.49	0.72	5.68	1.9M
	276.7 M	1660	no	233.30	1 M	0.72	1.72	9.04	11.71	1.9M
Vienna	230.0 M	1660	no	107.81	1 M	0.72	1.84	1.35	4.31	-
Morro Bay	407.0 M	2080	no	295.92	1 M	0.19	0.34	0.31	0.94	913k
		2080	no	295.92	10 M	0.19	3.34	0.34	3.99	913k
		2080	4x	-	10 M	1.09	8.20	0.92	10.35	3.6M
		2080	16x	-	10 M	4.01	19.39	3.09	26.60	14.6M
Morro Bay	1,000 M	Titan	no	-	1 M	0.18	0.19	0.38	0.84	-
		Titan	no	-	10 M	0.23	1.43	0.34	2.19	-

Table 3.3: Rendering performance. (Brute-force) Time to render all points in a single frame. (Progressive) Time spent on the passes and the total time of a progressively rendered frame. (Budget) Number of points rendered in the *Fill* pass. All timings in milliseconds. Resolution: 1920x1080.

time to convergence, but it does not affect rendering performance because we render at most $numReprojected + fillbudget$ points per eye in a frame. Since the pose of the head-mounted display always changes from frame to frame – even if it sits on a table, due to tracking noise – the result will closely approach but never truly reach convergence. With 4xMSAA and a resolution of 1448x1608, the maximum number of points that may be reprojected per eye is $4 \cdot 1448 \cdot 1608 = 9.3M$.

3.5 Limitations, Discussion and Future Work

In this section, we list and discuss some of the limitations of our current approach.

- Our method is currently in-core only. The complete data set must fit into CPU memory, and the position data and chosen attribute must fit into GPU memory. The GPU needs to store 16 bytes per point, but the CPU needs to store all the attributes to enable fast switching of attributes. However, implementations can also choose to keep attribute data in separate files on disk to quickly stream them to the GPU without the need to hold them in RAM.

- Our progressive method is developed for data without spatial acceleration structures. However, not having acceleration structures increases the duration to convergence since we cannot use frustum culling or culling by LOD to reduce the amount of points to the most viable candidates. Future work may explore the possibility of creating simple acceleration structures during loading, or afterwards in parallel to improve performance and quality during runtime.

- We currently do not offer a cost-effective method for quality improvements. MSAA works, but effectively acts as costly supersampling. The *Prepare* pass automatically treats each MSAA sample in the rendered image as if it was a separate pixel.

- Regions with higher point density will progress quicker than regions with low density, because the random subsets rendered by the *Fill* pass will also have a higher density in these regions.

- If points occupy more than one pixel or MSAA sample, then the *Prepare* pass will add them multiple times into the dynamically generated vertex buffer for reprojection. We tested an alternative implementation of the *Prepare* pass that operates on 2x2 frame buffer samples (= 4 pixels with no MSAA, 1 pixel with 4xMSAA) and only adds those points that are unique within that 2x2 sample grid. This form of approximate prevention of duplicates increased performance up to 10% for point sizes of 2x2 pixels and 4xMSAA. No performance improvements and sometimes performance losses of up to 5% were observed with smaller point sizes or without MSAA, because the duplicate prevention cost roughly as much time in the *Prepare* pass as it saves in the *Reproject* pass.

As part of future work, we would like to investigate suitable anti-aliasing strategies that are targeted specifically at this progressive approach.

3.6 Conclusion

In this chapter, we have shown a method that can render any point cloud that fits in GPU memory in real time without the need to generate acceleration structures in advance. This is achieved by distributing the task of rendering a large data set over the course of multiple frames by reprojecting the previous frame to preserve already rendered details, and then adding a random subset of points to drive progress until convergence. We see our progressive point cloud rendering method as an advantageous alternative to brute-force rendering and to LOD methods in cases where users want to quickly inspect a few hundred million points. Our LOD generator presented in Chapter 2 has a throughput of up to 10M points / second and requires tens of seconds up to minutes to build an LOD structure out of hundreds of millions of points. Users have to wait until the octree generation is finished before looking at the results. Our progressive rendering method, on the other hand, can load data sets at rates of up to 37M points second (LAS) or 100M points / second (VBO formatted) and simultaneously render the already loaded parts in real-time. However, data sets that do not fit into the available GPU memory still require LOD structures for out-of-core rendering.

We believe that it has the potential to replace the brute-force rendering approach (all points in each frame) used in point cloud renderers that do not support LOD rendering, but also some LOD approaches as long as the point cloud fits into memory.

Progressive rendering allows us to efficiently render point clouds with high depth complexity, which hierarchical structures alone do not handle well. We believe that progressive rendering methods, such as ours for non-hierarchical data and Tredinnick and Ponto et al. [TBP16, PTC17] for hierarchical data, will prove to be essential to render increasingly complex scan data in real time.

Code and videos are available at `https://github.com/m-schuetz/skye` and `https://www.cg.tuwien.ac.at/research/publications/2020/schuetz-2020-PPC/`.

CHAPTER 4

Real-Time Continuous Level-of-Detail Rendering of Point Clouds

Figure 4.1: Continuous level of detail attempts to achieve gradual transitions from high levels of detail to lower levels of detail depending on distance to the viewer and distance to the center of the screen. Our targeted use case are VR applications where classic discrete LOD structures are especially noticeable due to sudden drops in point density and popping artifacts during motion.

The contents of this chapter were initially published as a conference paper *"Real-Time Continuous Level of Detail Rendering of Point Clouds"*, and presented at IEEE VR 2019 in Osaka [SKW19].

4.1 Introduction

Level-of-detail methods can be categorized as either discrete or continuous. Both approaches reduce the number of points with increasing distance from the viewer, but with discrete methods the density drops in sudden steps, whereas continuous methods exhibit a smooth and gradual reduction of the point density. Discrete LOD (DLOD) structures are mainly used nowadays because they are easy and fast to implement, generate and render, but they suffer from noticeable rendering artifacts. Two of the main issues are the sudden change from one LOD to the next, especially with low point budgets or during loading, and popping artifacts during motion as individual chunks are added and removed from the scene. In this chapter, we introduce a continuous LOD (CLOD) method that results in a gradual reduction of the point density from one LOD to the next, as opposed to DLOD methods that exhibit step-wise transitions. The results are in contrast to our LOD generation method presented in Chapter 2, which generates discrete LOD structures in a preprocessing step. However, our current implementation of CLOD is actually based on pregenerated discrete LOD levels, and transforms the step-wise transitions to gradual transitions at runtime through sample elimination.

Virtual Reality introduces additional issues in point-cloud rendering, such as drastically increased performance and quality requirements. The HTC Vive and Oculus Rift head-mounted displays (HMDs) both require a frame rate of 90 frames per second (FPS) per eye, for a total of 180 FPS. Render target sizes are also relatively large, with display resolutions of 1080×1200 for the HTC Vive and the Oculus Rift, and a recommended resolution that is about 1.4 times higher in each direction to account for distortion and aliasing [Vla15]. Furthermore, anti-aliasing becomes mandatory because aliasing and other rendering artifacts are much more noticeable in VR. Therefore, the level of detail of a point cloud has to be reduced considerably in order to meet these high performance requirements. Unfortunately, this augments noticeable popping artifacts prevalent in discrete LOD approaches as larger chunks of points are blended in and out during motion.

The distortion effect of the lenses inside HMDs also results in wasteful rendering at the outer regions of the image. The lenses create a pincushion distortion that has to be countered by a barrel distortion before displaying the rendered image. By default, this barrel distortion is applied to the rendered image and strongly compresses outer regions. This results in an effectively reduced resolution for outer regions and thus needlessly rendered pixels [Vla15]. NVIDIA's multi-res shading and lens-matched shading address this issue by rendering outer regions at lower resolutions [Kra18], but this only reduces shading cost, not geometry cost as is relevant for point clouds. We also take advantage of the reduced pixel resolution in the periphery by continuously reducing the point density from the center to the edge of the display.

Figure 4.1 shows an example of a model that is particularly difficult to render using state-of-the art discrete level-of-detail (DLOD) mechanisms in VR even though it is only 86 million points: It has high depth complexity from most viewpoints due to multiple floors and additional structures like fences and wires. The commonly used octrees and

kd-trees produce chunks that do not align well with arbitrarily oriented walls, pillars or stairs. Also, the sizes of the chunks are limited in granularity, which makes frustum culling, but also focusing on details towards the center, less efficient. Chapter 3 already introduced progressive rendering as a method that is particularly well-suited to models with high depth-complexity because the complexity does not affect the time it takes to render a frame – only the time it takes to converge to the full result. Our CLOD method also provides two, albeit not quite as significant, advantages over DLOD methods when it comes to models with high depth complexity. The first is that CLOD rendering has a higher visual quality than DLOD rendering even at lower levels of detail, especially during motion, since it eliminates popping artifacts. The second is a result of our CLOD implementation that constitutes a limitation on the one hand, but an advantage with respect to depth-complexity on the other: All points are stored in a single unstructured vertex buffer and in each frame, we only need to invoke one draw call, no matter how many points (or theoretically visible nodes) are drawn.

Our proposed continuous LOD method addresses the challenges of real-time point cloud rendering in VR through the following contributions:

1. Our method eliminates chunk-wise popping artifacts prevalent in state-of-the art DLOD methods, and evaluates in a point-wise rather than chunk-wise fashion which points to render.

2. The change of detail as users move through the scene is much less noticeable due to a subtle point-wise fading approach.

3. Our method exhibits a smooth transition in density as the distance to the viewer increases, as opposed to sudden jumps in density prevalent in DLOD methods.

4. The point density is decreased away from the center of the image, in order to account for the reduced resolution after barrel distortion in VR. As a result, significantly fewer points have to be rendered in the periphery.

4.2 Related Work

Continuous LOD rendering refers to selecting and rendering subsets of the hierarchy and points such that the transition between different levels of details is smooth, i.e., the point density changes gradually rather than suddenly. QSplat [RL00] and Sequential point trees [DVS03], two of the oldest LOD methods for point clouds, already supported continuous LOD rendering. Both approaches are fidelity-based and allow rendering all points up to a certain screen-space point density. However, QSplat is targeted to CPU-based rendering and not suited for the GPU, and sequential point trees are view-dependent, meaning they are not suited to inside-out viewpoints, i.e., data sets in which users navigate within the bounding box of the object. Layered point clouds [GM04b] are the most widely used choice for streaming and rendering massive point clouds, but

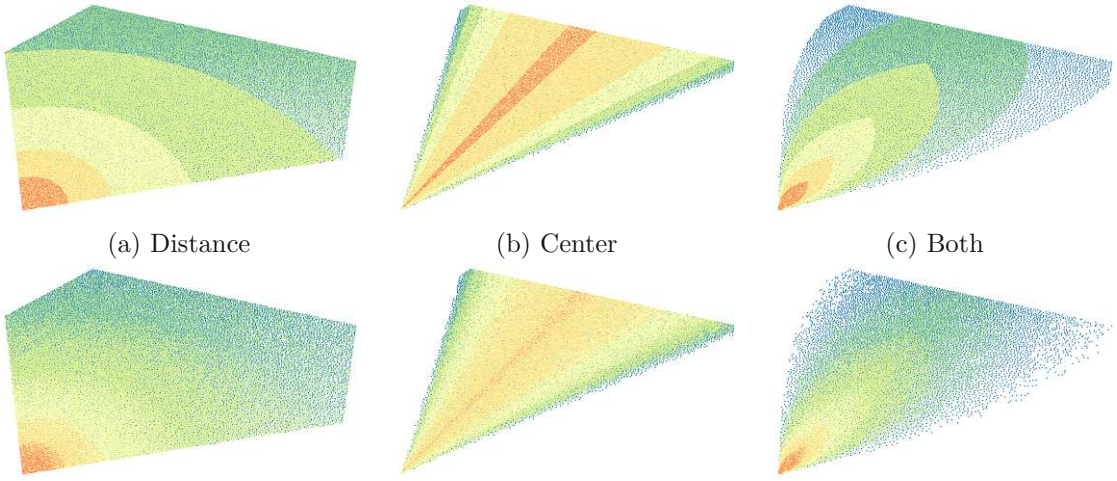
their structure stores points in discrete levels of detail. On our case (see Chapter 2), the spacing between points is halved with each level, and the transition from one level to the next results in a sudden drop of the point density. Other methods based on layered point clouds suffer from the same issue.

Progressive sample generators such as Recursive Wang Tiles [KCODL06], Progressive Multi-Jittered Sample Sequences [CKK18] or farthest point sampling [ELPZ97] constitute a form of view-independent continuous LOD. These approaches add points such that with each additional sample, the overall density of the model remains relatively uniform even though it increases. Cura et al. [CPP16] suggest intra-level ordering as an option to achieve continuous LOD for point clouds, similar to progressive sample generators. Points are first grouped into discrete LODs and within each LOD, points are further ordered randomly or based on morton, hilbert or halbert sequences. Due to the ordering, progressively loading additional points of a single discrete LOD level results in a continuously higher level of detail.

Simultaneously to our research, TU Delft developed view-dependent continuous LOD point-cloud rendering approaches that also take the perspective into account. Van der Maaden [Jip19] suggests to evaluate the visibility point-wise rather than node-wise, which we also do in our approach. The master thesis further suggests three strategies to filter points with continuous LOD properties: Random removal, filtering bands, and point radius density. The random removal approach *"works on the basis of generating a random value for each point, and either keeping or discarding the point according to a threshold"*. This is essentially the same approach we also implement in our research. The filtering bands approach selects points by density bands that are finer grained than the hierarchy levels of discrete hierarchical acceleration structures. Instead of reducing the density by the factor $\frac{1}{2}$ at 2 times the distance, an arbitrary amount of filtering bands may reduce the density by $\frac{1}{x}$ at x times the distance. The jump from one band to another is still discrete, but each jump is smaller and a large amount of bands may produce results that are indistinguishable from a continuous transition. The point radius density approach uses spheres that are tightly packed and grow with the distance to the camera, and then selects one point per sphere to achieve a continuous reduction in point density. Liu et al. [LOM+20] also recently published a method that randomizes the LOD value of each point to allow selecting subsets of points based on desired continuous LOD criteria. Major differences to our approach are that their approach is mainly targeted towards querying continuous LOD point sets from a database, whereas our approach is targeted towards downsampling a full-detail model to a continuous detail model directly on the GPU in order to obtain a continuous LOD model for the current viewpoint as quickly as possible.

4.3 Continuous Level of Detail

The basic idea behind our continuous level-of-detail method is to repeatedly create a reduced low-resolution version of the full point cloud at runtime, based on the current



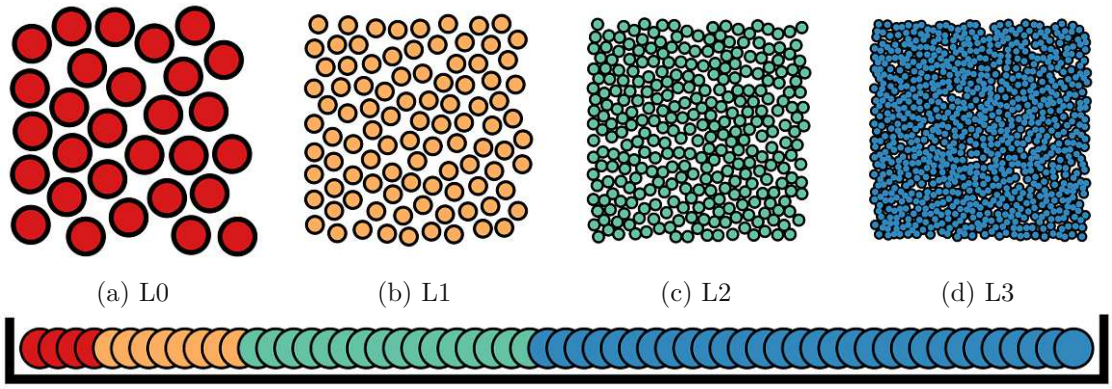
(a) Distance (b) Center (c) Both

(d) Distance (e) Center (f) Both

Figure 4.2: Point density reduction by distance to camera, distance to the center of the screen, and both. Top-row: Filtering based on the level of a point inside the hierarchy, with sudden drops in density. Bottom-row: Continuous results with dither-like patterns after adding a random value between 0 and 1 to the level of each point. Colors represent the level attribute of a point.

camera orientation and position, and with a gradual reduction in density as the distance to the camera increases. Our current implementation skips common optimizations such as frustum culling or hierarchical traversal. Instead, it iterates through all points of the data set to identify the points that should be rendered and stores the relevant points in a new vertex buffer. This reduced model is created over the course of a few frames – fast enough to give the impression that the model is always up-to-date, yet slow enough so as not to take too much performance away from the actual rendering process. We are able to create an updated version of the reduced model every 5 to 6 frames on a GTX 1080 by allocating around 1.1 milliseconds to the reduction step for point clouds up to 104 million points.

The main aspect that makes our method continuous is a runtime randomization of state-of-the-art discrete structures. These discrete structures organize points in level 0, 1, and so on. Adding random numbers between zero to one to the level of each point in such a discrete hierarchy then allows us to filter on a continuous scale rather than integer intervals. At a distance of 9.3 meters, we may want to display points up to level 2.3, for example. The results exhibit a continuous smooth transition in density with dither-like patterns, as shown in Figure 4.2.



(a) L0 (b) L1 (c) L2 (d) L3

(e) All points stored in a single vertex buffer, preferably but not necessarily ordered.

Figure 4.3: Our CLOD structure is a series of subsamples of the original point cloud that is then flattened into a single array.

4.3.1 Data Structure

When designing a chunk-based LOD system for points, one has to decide whether data from higher (more detailed) levels should be added to the data from lower levels (corresponding to memory optimized sequential point trees [WS06]), or replace it (corresponding to the original sequential point trees [DVS03], and similar to the data structure by Wand et al. [WBB+07]). The advantage of the additive approach is that it does not require additional memory and we do not have to take care of removing lower levels before displaying a higher level. The advantage of replacing lower levels is that each level can store and display representative points for the current level without data from another level inbetween.

We chose to primarily implement an additive scheme for performance reasons.However, instead of storing the points in a hierarchy as in the original schemes, our CLOD data structure is a single flat array, with the hierarchy level stored as a point attribute, and the structure is evaluated in a point-wise fashion on the GPU rather than a chunk-wise fashion on the CPU.

Points are subsampled by enforcing a level-dependent spacing between points, given by

$$spacing_{level} = \frac{rootSpacing}{2^{level}} \tag{4.1}$$

For example, level 0 contains points with a spacing of 1 meter, level 1 points with a spacing of 0.5 meters, and so on. Each point is assigned to exactly one level, and merging all levels results in the original point cloud.

To reduce aliasing at lower levels, we borrow the idea of averaging from the replacement scheme: points that are assigned to lower-level nodes keep their original position but have

their original color value replaced by an average color over the area they represent. This is not an entirely accurate solution since points with averages over different radii will be intermixed due to the additive LOD scheme, but it is significantly better for the visual quality than aliasing effects from keeping the original color values. Overblurring occurs but remains a minor issue because in any given view, points with the correct averaging radii from high LODs far outnumber points with larger averaging radii from lower LODs.

The order of the points inside the array is not important because our in-core method repeatedly iterates over all points to produce a downsampled version at runtime. However, we still recommend to order them from lowest level to highest level because the method can be applied to any subset of the data, and ordering points from lowest level to highest level allows us to display a coarse version of the whole model while increasingly higher levels of detail are still being loaded.

4.3.2 Build-Up

The build-up step for our CLOD structure uses the publicly available PotreeConverter [Pot, Sch16] to organize points into an octree, and custom scripts to average colors and flatten the hierarchy into an array.

The PotreeConverter creates an octree out of a point cloud that can be used to stream and render only relevant chunks of points up to a certain level of detail. It also selects points based on the spacing between points as given by Equation 4.1, which we need for our CLOD simplification algorithm .The result exhibits two issues, however, that we addressed with additional custom scripts.

The first issue is aliasing as discussed above, as points in lower-resolution octree levels store the color from a single input point, rather than the average over the area they represent. We address this issue by computing the arithmetic mean of the color of a point in level_n, and the colors of all points at level_{n+1} within the distance defined by *spacing* at level n.

The second issue is the large amount of relatively sparsely populated nodes that are generated, and then stored in separate files. Each node consists of around 100 to 10,000 points. Handling a large amount of small files results in I/O overhead that is unnecessary since our method does not deal with individual tiles, but rather all points as a single large blob. We therefore concatenate all the octree nodes into a single file. The only hierarchical information that is kept is the octree level of a point, which is stored as a byte inside the alpha component of the point's color.

The result of the build-up step is an array of points where each point contains position, an average color over the area it represents, and its level within the hierarchy. Each point requires 16 bytes: 12 bytes for position, 3 bytes for color, and 1 byte for the hierarchy level.

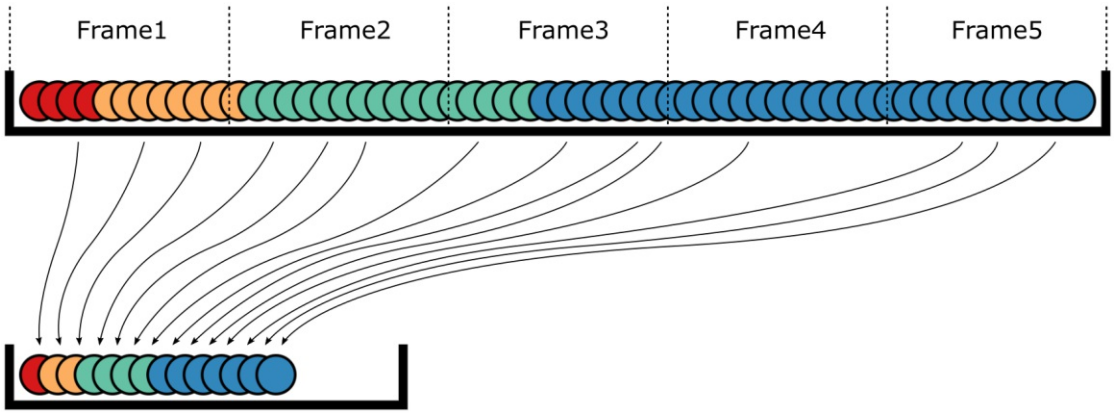


Figure 4.4: Over the course of a few frames, the compute shader iterates through all points of the full point cloud and copies the subset that will be rendered into a separate vertex buffer.

4.3.3 Rendering

The rendering process consists of two steps:

1. Reduce: Repeatedly recreating a reduced vertex buffer over the course of a few frames, based on view-frustum and a CLOD-factor.
2. Draw: Rendering the most recent reduced vertex buffer.

Reduce

The reduce step creates a new vertex buffer by iterating through all points in the full point cloud and copying those that match the desired level of detail in the target buffer, as shown in Figure 4.4.

We want to obtain a vertex buffer with a specific target spacing between points, depending on the distance to the camera. As a baseline, we would like a point spacing of 1 millimeter at a distance of 1 meter. The baseline is adjustable by a CLOD-factor and multiplied by the distance. This definition for the baseline makes it independent of the field of view and the resolution of the target devices, so that users will get to see the same points for the same viewpoint, except for additional points in the periphery with higher fields of view.

For desktop use, this desired minimum spacing (millimeters) between points at any given distance to the viewer (meters) is computed as

$$targetSpacingDesktop = \frac{distance \cdot CLOD}{1000} \tag{4.2}$$

In order to account for lens distortions and the resulting reduction in resolution in outer regions of the image, we compute the desired spacing in VR as

$$targetSpacingVR = \frac{targetSpacingDesktop}{max(1 - a \cdot dc, minDensity)} \tag{4.3}$$

The denominator reduces the density in the periphery, which leads to a significantly reduced workload for the vertex shader. *dc* specifies the distance to the center in *normalized device coordinate* space, without depth. *dc* is zero at the center and one at the border of the ellipse inscribed within the screen. *a* is used to adjust how fast the density decreases and *minDensity* specifies a lower limit on the reduction in percent. We suggest values of 0.5 for *a* and 0.3 for *minDensity*.

We also tried to reduce the density from the center based on a Gaussian function but found no significant improvement. We therefore settled with the simpler Equation 4.3. Figure 4.2 shows the result of the reduction for Equation 4.2 and Equation 4.3.

In VR, the reduce step is executed once per frame for a single HMD-centered frustum that covers both eyes, and the same reduced model is then rendered for both eyes. During quick motions, it will be noticeable that the currently rendered model is missing points outside of this view frustum because it takes 5 to 6 frames to produce an updated model. To alleviate this issue, the reduce shader clips against an extended frustum by projecting a point to normalized device coordinates, and then clipping the x and y axes against a range of $[-2, 2]$ instead of $[-1, 1]$. We suggest to set *minDensity* to around 0.3 to capture additional points in the extended frustum in a low but sufficient density to account for quick head movements.

The reduce operation is implemented as a compute shader that iterates over all points and stores the points that pass our CLOD requirements in a new buffer, as shown in Figure 4.4. The requirements are evaluated by first clipping against the extended frustum, and then comparing the spacing of the currently processed point to the target spacing at that point's location. If the spacing of that point is smaller than the targeted spacing, the point will be discarded because it represents a higher level of detail than necessary. The spacing of a point is obtained from its level in the hierarchy by applying Equation 4.1. The spacing can alternatively be interpreted as the amount of space that this point occupies for itself. If the targeted spacing is large, only points that occupy a respectively large space should be visible.

At this stage, all the points are still classified in discrete integer levels, which continues to produce sharp drops in density. In order to produce smooth transitions, we add a random factor between 0 to 1 to the level of the point. This pseudo-random factor is different for each point, but remains the same for each point over time.

The randomization of the level of a point also randomizes the spacing, the claimed minimum distance to another point at this hierarchy level. Since we add to the level, the reported spacing is reduced. Some points of the same hierarchy level are now more likely

to fail the targeted spacing requirements than others, which leads to a smooth falloff in density.

Draw

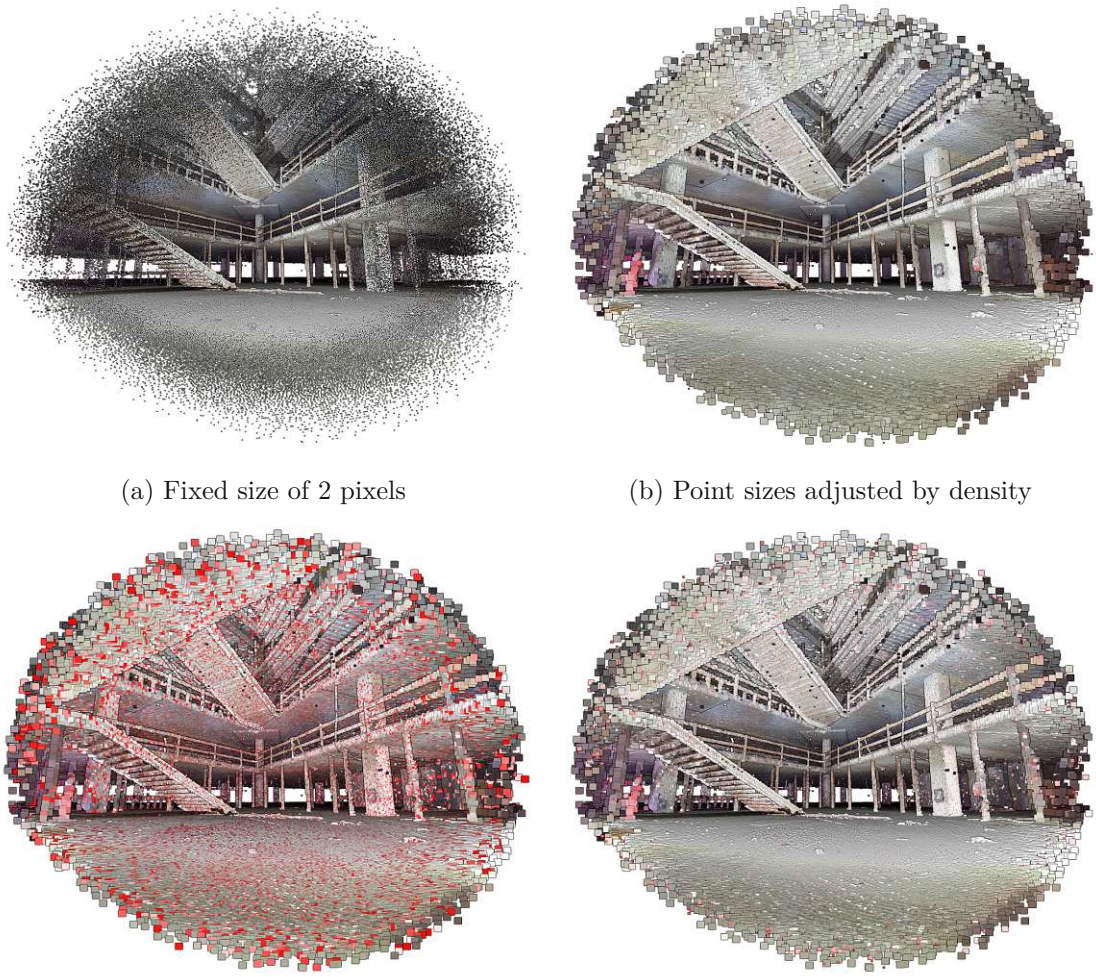
The draw step renders the most recent fully reduced vertex buffer, as created by the reduce step. Since the vertex buffers are created directly on the GPU, we use the *gl.drawArraysIndirect* command to render the data without the need to send the vertex buffer, or the number of points it contains, back to the CPU.

Point sizes are set to *targetSpacing* (millimeters) in order to fill the holes that appear due to the reduce step, as shown in Figure 4.5b. Points with a lower spacing are discarded in the reduce step, and resizing the remaining points makes up for the reduced density. However, this only deals with holes that would be caused by our CLOD method. It does not deal with holes due to insufficient or varying sample density in the original point cloud. We suggest to additionally specify a minimum point size in millimeters, based on the sample density of the 3D scanner, to avoid huge gaps between points upon closer inspection by the user.

At this stage, moving through the scene still exhibits irritating popping artifacts of individual points if points accepted by the reduce shader immediately appear with their full world-space size of *targetSpacing*. Using *targetSpacing* as the world-space point size results in pixel sizes that are independent of the distance to the viewer. As users get closer to a point, the *targetSpacing* at this point's location, and therefore that point's world-space size, shrinks, but its resulting pixel size remains the same. This means that depending on the CLOD factor, newly accepted points pop in at a certain pixel size (e.g., 5 pixels). To improve from point-wise popping to point-wise fading, we propose an additional blend range within which points grow to their full size, as shown in Figure 4.6. At the moment when a point first becomes visible, that is as soon as $targetSpacing = pointSpacing$, its size is set to 0. Once we get closer, the point grows until it reaches its full size of $targetSpacing$ when $targetSpacing = blendRangeFactor \cdot pointSpacing$. Within the blend range, point sizes are linearly interpolated between $[0, targetSpacing]$. We suggest a value of 0.8 for the $blendRangeFactor$ so that users have to move another 20% closer to the point until it reaches its full size. Note that this point-wise fading method depends on $targetSpacing$ but not on time. As a result, points fade in or out depending on the distance to the user as well as distance to the center of the screen. Note also that the blend range is defined as a fraction of the *pointSpacing*, and is therefore larger at lower levels of detail. It may take 10 meters until points of lower LODs grow to their full size and only 1 meter for points at higher LODs, but it is always the same fraction of distance from the appearance of a point to full growth.

4.4 Results

Figure 4.7b shows that our CLOD approach achieves a more uniform distribution of points in screen-space, as opposed to DLOD methods like in Figure 4.7a. The desired



(a) Fixed size of 2 pixels

(b) Point sizes adjusted by density

(c) Identifying points in blend-range (red)

(d) Reducing size of points in blend-range

Figure 4.5: Result of the reduction step and application of a blend-threshold to fade-in additional detail.

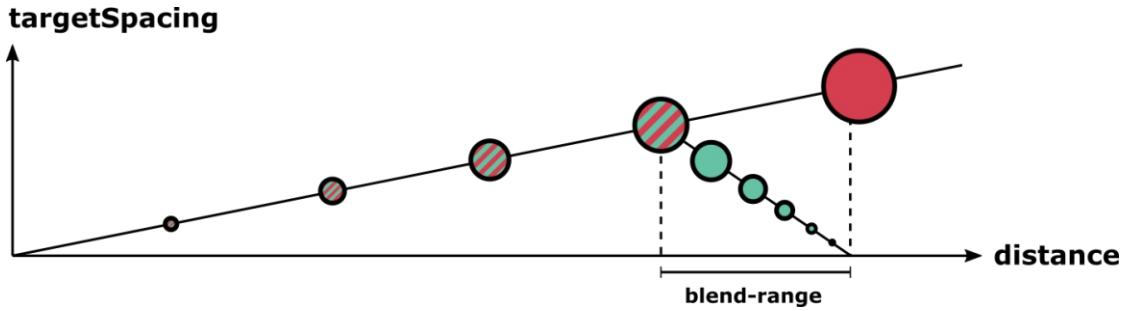


Figure 4.6: Point sizes are set to targetSpacing in order to fill holes from filtering by targetSpacing. To avoid points popping in at full size the moment they become visible (red), we let them grow to full size within a certain blend-range (green).

(a) DLOD (b) CLOD (c) CLOD VR

Figure 4.7: Distribution of 150k points with DLOD and CLOD.

density at any location can be specified in the reduce shader by any mapping of a 3D coordinate to a target spacing. Equation 4.2 and Equation 4.3 are the two we propose for desktop and VR rendering, respectively, but they can be replaced or combined with other mathematical equations.

4.4.1 Implementation

Our CLOD renderer is implemented in JavaScript based on Google's V8 engine [V8], with custom bindings to OpenGL 4.5 and OpenVR [Ope]. While fast for a scripting engine, tree-traversal of hundreds to thousands of octree nodes, as done by Potree [Sch16], is an expensive operation in JavaScript that, combined with also relatively expensive OpenGL draw calls, can cost a couple of milliseconds of CPU time per frame. A considerable advantage of the proposed CLOD implementation in scripting engines is that it eliminates tree-traversal of the point cloud and reduces draw calls to a single call to *glDispatchCompute* and *glDrawArrayIndirect*, at the cost of reserving 1 millisecond of GPU time per frame for the reduce shader.

Source code samples for this work are available at `https://github.com/m-schuetz/ieeevr_2019_clod`.

4.5 User Study

The improved acceptance of our method was evaluated in a user study with 23 participants. We chose to evaluate our results by a user study instead of comparing error metrics to a ground truth because popping artifacts, perceived quality, and potential issues with density reductions in the periphery are largely a perception issue.

We invited staff of a visual computing research center, staff of an archaeology research center, and students and staff of the visual computing group of a university to participate in our user study. All three institutions work with point clouds to a certain extent, so many of our participants have already captured, processed, or visualized point clouds before. Our participants were between 25 to 59 years old, 78% were male and 22% female. All participants are regular users of computers. Two participants have never experienced VR, 5 and 7 rarely and sometimes experience VR, and 9 participants regularly or often experience VR. 7 participants work sometimes with point clouds, and another 7 regularly or often. 6 users have never experienced point clouds in VR before, 12 users rarely, 2 sometimes and 3 regularly work with point clouds in VR.

Users were confronted with two point clouds, Matterhorn and Endeavor, in four scenarios inside the VR environment. They could freely switch between three LOD methods, labeled A, B, and C, with the touch pad on the controller. Method A is an octree-based level-of-detail system, implemented after Potree [Sch16], which adjusts point sizes to the level of detail. Method B is based on the same octree approach, but points have a fixed pixel size. Method C is our approach that renders with continuous level of detail.

The four scenarios are:

1. Matterhorn: 104M points; high level of detail.
2. Endeavor Small-Points: 86M points; low level of detail; small point sizes, insufficient to fill holes
3. Endeavor Large-Points: 86M points; medium level of detail; point sizes for method A and B adapted to cover holes
4. Endeavor Illuminated: 86M points; medium level of detail; no colors, each point is white, but illuminated by Eye-Dome-Lighting [Bou09]. We decided to evaluate this case because the original colored data exhibits strong noise and flickering artifacts since it is a combination of individual scans with different light exposures.

Tests were executed on a VIVE (6 participants) and a VIVE Pro (17 participants), depending on availability. Render-target resolution was set to about 1512 times 1680 pixels for both in the SteamVR settings menu. Level of detail was set conservatively to maintain 90 fps in the worst case on all of our test systems, which were equipped with either a GTX 1070 or GTX 1080. The point budget for the octree methods, and the CLOD factor for our method was set to render no more than 3 million points at a time,

so that participants were presented with the same images (apart from HMD resolution) regardless of the used system.

The workflow for each participant was as follows:

1. Hand participant an informed user consent form to sign.
2. Ask questions about the demographics. We asked the user's age, gender, their occupation, and how often they work with computers, virtual reality, point clouds, and point clouds in virtual reality.
3. Show users the four scenarios, one after another. During each scenario, users were asked to rate their overall impression, how noticeable changes in the level of detail are, and how irritating these changes are on a scale of 1 to 10 for each method. We filled out their answers in Google forms so that they could remain inside the VR environment and take another look before giving their answers.
4. After the four scenarios, users were asked to choose their favourite method, and the second best.

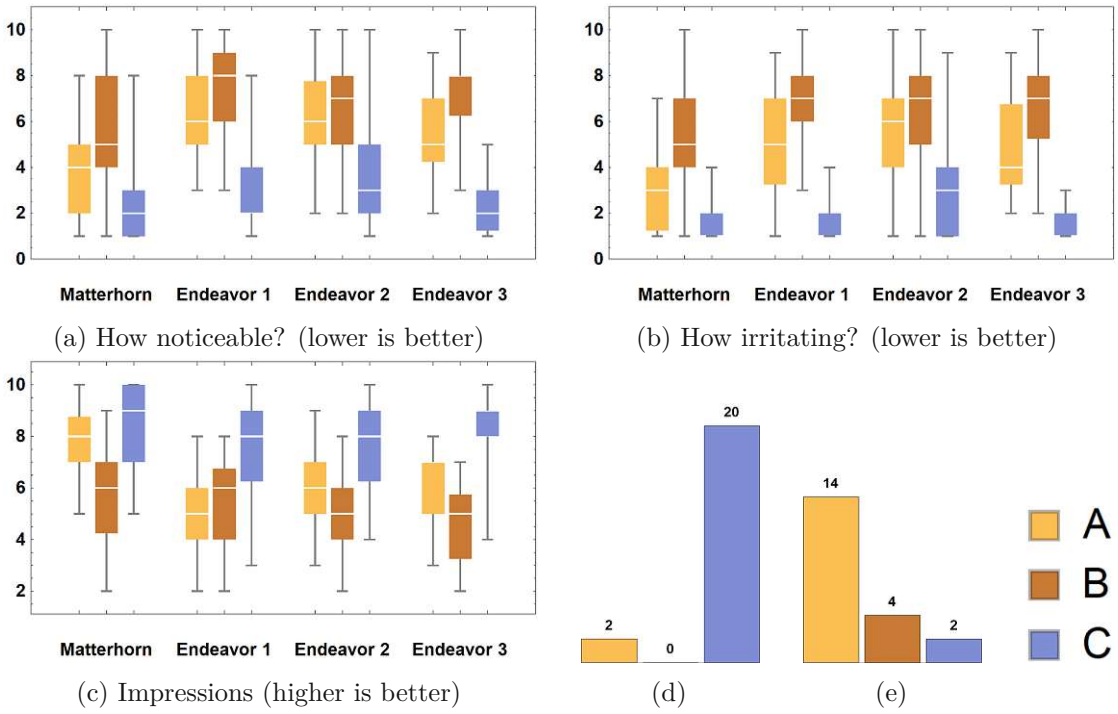
Our hypotheses were:

1. $H1_A$ and $H1_B$: Method C gives a better impression of the point cloud than methods A and B.
2. $H2_A$ and $H2_B$: Method C has less noticeable changes in the level of detail than methods A and B.
3. $H3_A$ and $H3_B$: Method C has less irritating changes in the level of detail than methods A and B.

We tested our hypotheses using the Mann-Whitney U test as it is non-parametric and does not assume a normal distribution, and compared the p-values with a significance level of $\alpha = 0.005$. 22 out of 24 combinations of scenarios and hypotheses have shown a significant improvement of our method ($p < 0.0037$). The non-significant cases are $H1_A$ in the Matterhorn scenario ($p = 0.152$; Figure 4.8c), and $H2_A$ in the Matterhorn scenario ($p = 0.012$; Figure 4.8a), i.e. method C does not leave a statistically significantly better impression and LOD is not significantly less noticeable compared to method A in the Matterhorn scenario. This is not unexpected since the Matterhorn data set has a low geometric complexity and mostly low-frequency color variations that do not pose a challenge for any of the evaluated methods. However, our method C shows significant improvements over A and B in all scenarios of the Endeavor data set, because it has a high geometric complexity that state-of-the art DLOD methods do not handle well.

Apart from that, 20 out of 23 participants rated our method the preferred one. One user did not name a preferred method, and three users did not name a runner-up.

We would also like to discuss one notable result. Figure 4.8a and Figure 4.8b show that our method was more noticeable and irritating in scenario 3, compared to its results in the other three scenarios. We believe that this is largely caused by the inhomogeneous and noisy colors of the points in this data set, combined with a higher level of detail and point sizes as opposed to scenario 2. The point cloud is a mix of laser scans from many different positions, each colored by photographs at the respective location. Because of the varying light exposure at different positions, some scans ended up darker than others, even in overlapping regions. As a consequence, overlapping regions are represented by bright to white points from one scan, and dark to black points from another scan. When a user navigates towards such a region, black points would suddenly appear in a previously mostly grey region. This is less of an issue at lower levels of detail where colors are averaged over the area they represent.



(a) How noticeable? (lower is better)

(b) How irritating? (lower is better)

(c) Impressions (higher is better)

(d)

(e)

Figure 4.8: Box plots of the users' ratings for methods A, B, and our method, C. (d) Overall preferred method. (e) Runner-up.

4.6 Conclusion

In this chapter, we presented a method to create and render point clouds with a continuous level of detail that is fast enough for VR applications. The continuous LOD results in subtle and less irritating changes of detail as users move through the scene, and therefore improves the VR experience over discrete LOD methods. This is achieved through three main contributions: first, a compute shader that iterates over the point cloud and selects

points according to a target spacing; second, a randomization of point LODs to avoid harsh changes between levels of detail, and third, a transition period where the size of new points is increased smoothly. A user study has confirmed that users prefer our method to state-of-the-art point-rendering methods.

We were able to implement our CLOD method without hierarchical traversal while matching the detail and performance of an octree-based DLOD method. The downside is that this currently limits our method to point clouds that fit in GPU memory, and also increases the number of frames it takes to compute a new down-sampled version of the point cloud with the number of input points. However, we believe that CLOD can be implemented in a similar hierarchical and out-of-core fashion as state-of-the-art DLOD methods by applying the reduction step to the nodes of the DLOD structure and discard points the same way we do now. In the future, we would like to evaluate optimal ways to realize this in order to achieve continuous level of detail for arbitrarily large point clouds.

Z-fighting issues are currently augmented by repeatedly recreating a new vertex buffer. Normally, repeatedly rendering two overlapping triangles at the same depth would return the same result as long as they are rendered in the same order and if they are projected with the same matrices, i.e., if there is no motion. Since the order of points inside the down-sampled model is unpredictable and changes with each iteration, so does the order in which they are drawn by the GPU, which results in constant z-fighting, even if there is no motion. We currently don't address this issue other than by carefully selecting the near clip plane. One potential solution to z-fighting are high-quality splatting methods that blend overlapping fragments together instead of picking a single one [BHZK05, ZPvG02, ZPVBG01, PZvBG00]. Slight inaccuracies in the depth buffer won't matter since we will compute the average within a certain depth range in any case. Another solution is to ensure that points at a specific level of detail have a certain minimum distance to each other, which reduces the chance that two or more points end up having the same depth value. This was not the case during our evaluation because the LOD generation method we used at the time did not enforce minimum distances across different levels of detail. The LOD generation method presented in Chapter 2 largely solves this issue due to the approximate poisson-disk sampling, but some points at the border between octree nodes might still end up being too close together.

Our current continuous LOD approach is based on point cloud data with coordinate and color values, but without normals, size, etc. Future work might explore the benefit of combining continuous LOD with more elaborate point-based rendering methods that take advantage of additional geometric information, such as surface-splatting [BHZK05, ZPvG02, ZPVBG01, PZvBG00]. A hierarchy of surface splats could comprise of oriented ellipses with radii that are adjusted to fill the gaps between the points at each level of detail, which could either improve our current blend-range approach or make it completely obsolete.

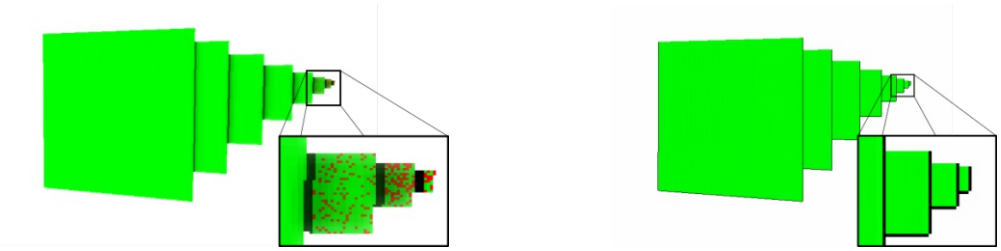
We currently reduce the point density in the periphery mainly to account for the distortion that is applied to the rendered image before it is displayed in an HMD. The same principle could likely be used for foveated rendering in order to adjust the point density to the

viewing direction of the user's eyes. A potential issue could be that our method requires multiple frames to compute a new reduced vertex buffer, which may or may not be too slow for quick eye movements.

Rendering performance of our method is currently harder to predict in advance compared to DLOD methods, because it is fidelity-based rather than budget-based. The same quality settings may result in 1 million points in one viewpoint, and 3 million points in another. However, adapting the level of detail to the performance of previous frames may be less noticeable than with DLOD methods.

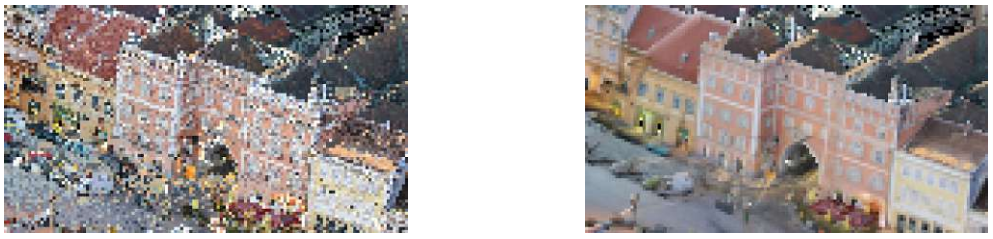
CHAPTER 5

Rendering Point Clouds with Compute Shaders



(a) Regular OpenGL z-fighting (b) 40 bit integer depth buffer

Figure 5.1: A 40 bit depth-buffer component reduces z-fighting.



(a) Regular Rasterization (b) High-Quality Shading

Figure 5.2: Two compute shader approaches: One displaying the closest point, the other blending overlapping points together.

The contents of this chapter were initially published in the poster *"Rendering Point Clouds with Compute Shaders"* at SIGGRAPH Asia 2019 [SW19].

In this chapter, we briefly introduce our method to render point clouds with compute shaders instead of the traditional rendering pipeline.

Traditionally, point clouds in OpenGL are rendered with the glDrawArrays(GL_POINT, ...) command, which passes point primitives through the OpenGL rendering pipeline. While many parts are programmable nowadays, others remain fixed. An appealing quality of GPGPU is that it gives developers the possibility to write their own rendering pipeline [KKSS18]. Günther et al. [GKLR13] proposed an OpenCL-based point-cloud renderer back in 2013, but were limited to 32-bit atomic operations at the time. Instead of using atomicMin, they implemented a busy loop with an early-out optimization to achieve major performance improvements over OpenGL.

In many cases, especially triangle rendering, the regular rendering pipeline remains faster than GPGPU rasterizers. However, GPGPU allows implementing features that may not be possible in the regular pipeline, such as improved depth buffers. The classic OpenGL projection matrices map nearby depth values over most of the available depth-buffer range, while leaving only little precision to distant parts of the scene. To make things worse, vertex transformations and resulting depth values are processed with floating-point numbers, which have a higher precision close to zero. A well-known trick to improve the precision is to reverse the depth buffer and map the near-clip plane to 1, and the far-clip plane to 0, so that distant depth values are sampled at a higher precision. Further information can be found in NVIDIA's "Depth Precision Visualized" article [Ree15]. Nonetheless, depth precision is inevitably lost during the vertex transformation, and the subsequent storage of the result in a single-precision floating-point vector. Our approach improves depth precision by computing the depth with double precision, and storing the result in a 40-bit integer buffer.

5.1 Method

We developed two approaches to draw point clouds with compute shaders instead of the classic rendering pipeline. The first method uses atomicMin to write the closest point into a custom framebuffer. The second method implements a blending function used for *high-quality surface splatting* [BHZK05]. Surface splatting typically refers to rendering points as oriented disks or ellipses, but in our case we are only rendering one pixel per point. We will therefore refer to it as *high-quality shading*, rather than splatting.

5.1.1 Rasterization via AtomicMin

This approach encodes color and depth into a single 64-bit integer and uses atomicMin to write the closest fragment into a shader storage buffer that acts as our framebuffer. RGB values are stored in the least significant, and the depth in the most significant bits. Due to this, atomicMin primarily takes the depth into account when it writes the value into the framebuffer, except when two fragments have exactly the same depth. In the latter case, the fragment with the smaller color value is picked.

Our approach gives developers control over a 40-bit integer depth value with a uniform or customizable and easily predictable precision over the whole range. 40 bit are sufficient to represent 1 trillion different values. Assuming millimeter precision, we end up with 1 trillion mm = 1 million km, which means we can represent the depth value of any object on earth and as far away as the moon in millimeter precision. To obtain millimeter precision in a scene that is represented in meters, we compute the depth in double precision, multiply it by 1 000, and store the integer part in an int64_t type value. It is also possible to split the full range of depth into sub-ranges with different precision, if higher precision near the camera is required without sacrificing view distance. A progression with, for example, half the precision at double the distance may be a reasonable choice, but functions such as log and pow do not work on double values at this time. Instead, developers can manually map depth ranges to different precisions, e.g., [0m, 10m] to nanometers, [10m, 10km] to micrometers and $[10km, 10\,000km]$ to millimeters. Each of these ranges occupy at most 10 billion integer values for a total of 30 billion out of 1 trillion available values.

In the render pass, depth and colors are encoded into a single 64-bit integer. The depth value is shifted 24 bits to the left, reducing its available range to 40 bits, and the color value is stored in the rightmost 24 bits. AtomicMin is then used to write this 64 bit integer into the SSBO. The atomic min operation stores new fragments only if the encoded depth value is smaller than previously written fragments.

```
1 int64_t val64 = (u64Depth << 24) | int64_t(vertex.colors);
2 atomicMin(ssFramebuffer[pixelID], val64);
```

In the resolve pass, a different compute shader that runs on each pixel reads the values from our custom framebuffer and stores the color values in an actual OpenGL texture. The shader also clears our framebuffer at the end by setting each value to 0xfffffffff000000. The first five bytes are the depth component which are reset to the maximum value, and the last three bytes are the RGB component which act as the background color. If set to zero, the background will be black.

```
1 uint64_t val64 = ssFramebuffer[pixelID];
2 uint ucol = uint(val64 & 0x00FFFFFFUL);
3 vec4 color = 255.0 * unpackUnorm4x8(ucol);
4 uvec4 icolor = uvec4(color);
5 imageStore(texture, pixelCoords, icolor);
6
7 ssFramebuffer[pixelID] = 0xfffffffff000000UL;
```

5.1.2 High-Quality Shading

The second approach is a simplified implementation of *High-Quality Surface Splatting on Today's GPUs* [BHZK05] that blends together single-pixel-points, rather than surface splats. It achieves anti-aliasing by computing an average of the closest fragments within a pixel. Many of the points in a pixel are samples of the same front-most surface and

Table 5.1: Rendering times for Heidentor (26M points), Retz (145M points on 2080 TI, 100M on 1060 GTX) and Morro Bay (117M points).

Model	GPU	AtomicMin	HQ-Shading	GL_POINT
Heidentor	2080 TI	1.64 ms	3.37 ms	5.71 ms
	1060 GTX	4.88 ms	11.75 ms	13.60 ms
Retz	2080 TI	6.41 ms	12.95 ms	34.04 ms
	1060 GTX	14.89 ms	31.78 ms	58.82 ms
Morro Bay	2080 TI	5.83 ms	15.48 ms	60.26 ms

therefore all of them should contribute to the pixel. In basic rendering approaches, however, only the closest fragment is drawn.

Our compute-based version works as follows. The first pass creates a depth-buffer using the atomicMin approach from the previous section. The second pass sums up the RGB values of all fragments whose linear depth values are at most 1% larger than the corresponding depth buffer value. Using a percentage makes this method work at arbitrary distances. Each fragment that passes the depth-test also increments the fragment counter for that pixel. In order to reduce the amount of atomicAdd operations in the second pass, we combine the red and green channels into a single 64 bit integer, and the blue channel and the counter value into another 64 bit integer.

```
1 int64_t rg = (r << 32) | g;
2 int64_t ba = (b << 32) | a;
3
4 atomicAdd(ssRG[pixelID], rg);
5 atomicAdd(ssBA[pixelID], ba);
```

In the third pass, the final color value of a pixel is computed by dividing the sum of fragment colors by the number of fragments. The result is an image where each pixel contains the average of overlapping points within a certain depth range, rather than only the closest point.

```
1 uint64_t rg = ssRG[pixelID];
2 uint64_t ba = ssBA[pixelID];
3
4 uint a = uint(ba & 0xFFFFFFFFUL);
5 uint r = uint((rg >> 32) / a);
6 uint g = uint((rg & 0xFFFFFFFFUL) / a);
7 uint b = uint((ba >> 32) / a);
8
9 uvec4 icolor = uvec4(r, g, b, a);
10 imageStore(texture, pixelCoords, icolor);
11
12 ssFramebuffer[pixelID] = 0UL;
13 ssRG[pixelID] = 0UL;
14 ssBA[pixelID] = 0UL;
15 ssDepthbuffer[pixelID] = 0xfffffffff000000UL;
```

5.2 Performance

Table 5.1 compares rendering times of our two compute based methods against the traditional GL_POINT method. Retz on a 2080 TI renders 5.3 times faster, and Morro bay renders 10 times faster with atomicMin than GL_POINTs. The high-quality shading method renders 2.6 times and 3.9 times faster for the respective data sets. We would like to note that the results vary greatly depending on the order of points and the selected viewpoint. Shuffling points reduces the efficiency of our compute based method. More detailed benchmarking will be part of future work.

5.3 Conclusions

We have shown that in the context of point clouds, compute shaders are not only a viable, but possibly advantageous alternative to the traditional OpenGL rendering pipeline, with speed-ups of up to 10 times. However, at this time all work was done and evaluated on point sizes of one pixel. Initial tests have shown that our current compute shader implementation scales roughly linearly with the number of pixels per point, whereas the OpenGL rasterizer scales better than that. Our approach is therefore ideal for point sizes of 1 pixel, but less suited for sizes larger than 2 pixels.

This computer shaded based approach is an experiment that successfully renders tens of millions of points with a size of 1 pixel faster than the OpenGL points primitive, but exhaustive benchmarks and evaluations are still subject to future work. An initial attempt to use this approach to render layered point clouds did not lead to performance improvements but rather a loss of performance by up to around 50%. We suspect that one reason for the loss of performance might be that points inside the octree nodes had little to no locality compared to the full point cloud data sets in their original order. In the future, we would like to work on an exhaustive evaluation of the GPGPU vs. the traditional approach in order to exactly determine the cases that lead to better or worse performance.

With WebGPU on the horizon, we would also like to evaluate the feasibility of our approach in web browsers. A likely issue is that WebGPU only supports 32 bit data types, but we believe that 32 bit atomics are sufficient for the high-quality approach. Differences to our OpenGL-based implementation are that the depth buffer would have a precision of 32 instead of 40 bit, and instead of two atomicAdd on a red+green and a blue+count buffer, we would need to do four atomicAdd on separate red, green, blue and count buffers.

CHAPTER 6

Conclusion

6.1 Summary

In this thesis, we have shown methods to improve the performance of generating LOD structures for point clouds, rendering large point clouds without the need for LOD structures, and a continuous LOD approach that improves visual quality by computing gradual transitions between LODs instead of discrete LOD methods with sudden drops in point density.

The fast generation of an LOD structure for point clouds (Chapter 2) shows that octrees can be generated up to ten times faster than the state-of-the-art, especially with a random sampling strategy that processes up to 11 million points per second. A high-quality sampling strategy that generates blue-noise samples is slower but still manages to achieve a throughput of up to around 6 million points per second. These performance improvements were realized by adding a preprocessing pass that first analyzes the data and then reorganizes it into a structure that is suitable to parallel processing. Specifically, a hierarchical counting sort was used to first count the number of points on a grid, then deduce suitable chunks from the counters, and finally iterate over all points again to transfer them to the respective chunks.

However, even our fast LOD generation method still takes a minute to prepare an octree for 270 million points or a couple of minutes for a billion points. In order to allow users to immediately inspect and navigate through any point cloud that fits in GPU memory - up to 1 billion with 24GB - we also developed a progressive point cloud rendering method that does not require LOD structures at all, as shown in Chapter 3. Our progressive method renders a random selection of points in each frame and preserves previously rendered points through reprojection. As a result, it converges to the full detail image over the course of a few frames, usually a fraction of a second. Even state-of-the-art LOD structures do not converge to the full result since they render from low to high levels of

detail but stop once a certain point budget is reached. Another advantage of progressive rendering is that it deals well with models with high depth complexity. While LOD methods have to increase the point budget to counter an increase in depth complexity - thereby sacrificing performance - our progressive method can always maintain real-time framerates while still converging to the full-detail image. The main impact of models with high depth complexity on progressive rendering is that the time to convergence increases. The disadvantages are that our current implementation of progressive rendering offers no option for anti-aliasing other than MSAA, and that it is an in-core method that can only render as many points as can fit in memory.

Our continuous LOD approach subsamples a point cloud at runtime to achieve a gradual transition in point density rather than the sudden drops in density known from discrete LOD structures. The current implementation does so by flattening a discrete LOD structure into an array (similar to an unordered version of Dachsbacher et al. [DVS03]) but retaining the hierarchy level as an additional attribute in each point. At runtime, a compute shader iterates over all points, randomizes the LOD value to turn the discrete integer LOD values into continuous floating-point LOD values, and selects a subset with continuous transition based on LOD value and viewpoint. This current implementation is in-core and does not support frustum culling because it iterates over the whole flattened and unsorted vertex buffer. As such, it is not a production-ready approach but rather an experiment that shows that continuous LOD for point clouds can be done, and that it has significant advantages such as the elimination of node-wise popping artifacts prevalent with discrete LOD structures. A user study has shown that users prefer our CLOD method over DLOD methods while rendering roughly the same amount of points.

6.2 Combining LOD Generation, Progressive Rendering and CLOD

We initially developed the fast LOD generation as a method to generate layered point clouds faster, and progressive rendering and continuous LOD as mutually exclusive alternatives to rendering with layered point clouds. In retrospect, however, we found that the methods are not entirely orthogonal and that they can benefit from each other. In this section, we would like to briefly discuss potential combinations of seemingly mutually exclusive methods.

Discrete LOD generation & CLOD:
Our CLOD structure uses layered point clouds and then removes information, namely the partitioning into nodes. This was a decision to make our implementation simpler and there is no reason why our CLOD approach should not work with unmodified layered point clouds. We believe that continuous LOD could also be obtained by rendering discrete chunks of points, and then discarding points in the vertex shader until the transition appears smooth. An actual implementation might not be that trivial, however, since there are additional considerations that need to be addressed. For example, what to do

while additional nodes are still being loaded and what if distant regions are temporarily rendered at higher LOD than close regions due to the order in which nodes were loaded?

LOD generation & Progressive Rendering:
Similarly, progressive rendering can benefit from layered point clouds, and Treddinick and Ponto et al, [TBP16, PTC17] have, in fact, done so before. Their method also reprojects the previous frame to preserve detail, but rather than adding more detail through random points from the whole point cloud, they add additional detail by rendering different selections of the currently visible octree nodes. Due to this, they achieve a significantly faster convergence because data sets with billions of points only need to progressively render subsets with tens of millions of points to reach convergence. Note that convergence means something different in a hierarchical context: Our in-core method converges when all points of the full data set are rendered, whereas their hierarchical out-of-core method converges when all nodes up to a certain point budget or level of detail are rendered. This is perfectly reasonable because with massive point clouds, we would not want to load all details, only details up to a user defined resolution. An open issue that their work has not addressed yet is the quality of the progression. Since their work renders different nodes in each frame, it can happen that popping/stuttering artifacts may appear during camera motion because some disocclusions are filled completely right away, while others are filled in future frames. Our in-core method addresses this exact same problem through shuffling of the vertex buffer. A potential solution for hierarchical structures might be to shuffle points in each node, then render 10% of each node over 10 frames. We expect to implement and experiment with hierarchical progressive rendering in the future since we believe that it will become an essential approach to rendering large point clouds because it allows us to maintain real-time frame rates at all times while still converging to high detail-images.

Progressive Rendering & CLOD:
We are not entirely certain about the feasibility of this combination but believe it has merit. Hierarchical progressive rendering would allow us to converge to high details that make CLOD obsolete. However, CLOD may potentially improve the quality of the intermediate results by ensuring that no discrete LOD blocks are visible while additional data is still being loaded. Considering that this intermediate state may persist for seconds or tens of seconds on slower internet connections, CLOD may significantly improve the visual quality in the meantime.

6.3 Outlook and Future Work

Load and SSD performances; direct load to GPU

We expect major improvements in load performance from local disks over the next few years as SSDs get faster, and fast SSDs get more popular. The PS5 was recently announced having an SSD with a read bandwidth of 5.5GB/s and the XBox Series X

with a read bandwidth of 2.4GB/s [1]. On PCs, NVIDIA is pushing towards direct IO from disk to GPU without a roundtrip through CPU and system memory [2]. On modern PCIe 4 SSDs, this may allow developers to load point-cloud data from disk directly to GPU memory at rates of around 5GB/s. This amounts to around 300 million uncompressed points comprising position and color attributes, or around 200 million points in the widely used LAS format with several additional point attributes. These kinds of performances would significantly boost the usefulness of our progressive rendering method considering that the loaded data could be unpacked and shuffled directly on the GPU. We currently achieve up to 37 million points per second loading the LAS format, but with direct storage and processing all the data directly on the GPU, we are certain that loading files at 100 to 200 million points per second - and simultaneously rendering them in real time - is feasible within the next few years.

A potential limitation are compressed data formats that require loading points in order. The LAZ format, for example, has partial random access but only with a granularity of usually 50,000 points, i.e., we can only spawn one thread per 50,000 points. A granularity of that order lends itself well to decompression using tens of threads on the CPU, but less so with thousands of threads on the GPU. With direct load, it might be necessary to use other compression algorithms that are better suited to massive parallelism on the GPU, or experiment with smaller chunk sizes.

Combining High-Quality Rendering, Progressive Rendering, and Hierarchical Rendering

Although previous work and our contributions address many problems and shortcomings of real-time point-cloud rendering, there is still a gap between academic solutions and practical implementations. There are various point-cloud renderers that are already capable of rendering hundreds of billions of points in real-time, but none that are able to do so in high quality comparable to textured meshes, and especially not on mid-range hardware. In my opinion, the state of the art already provides most solutions to achieve this goal, but some additional research and engineering effort may be required to connect the individual parts. In the future, we would like to attempt to achieve high-quality rendering of arbitrarily large point clouds through a combination of an LOD structure that computes averaged color values in lower LODs (like Wand et al. [WBB+07]), coupled with fast LOD generation to make this structure viable for hundreds of billions of points (like our fast LOD generation approach, see Chapter 2), and using progressive rendering with LOD structures to achieve high rendering performance even for models with a high depth-complexity (like Tredinnick and Ponto et al [TBP16, PTC17] for hierarchical structures, coupled with our randomized progressive rendering for pleasant convergence patterns, see Chapter 3).

[1] https://www.theverge.com/2020/3/18/21185141/ps5-playstation-5-xbox-series-x-comparison-specs-features-release-date

[2] https://on-demand.gputechconf.com/supercomputing/2019/pdf/sc1922-gpudirect-storage-transfer-data-directly-to-gpu-memory-alleviating-io-bottlenecks.pdf

Simultaneous LOD Generation and Rendering

We are also committed to work on a progressive LOD generation scheme that allows users to immediately start looking at progressively refined LOD models of large point clouds while the LOD generation is still in progress. The progressive rendering without LOD structures may be used to provide a very fast overview of hundreds of billions of points while an octree is built in parallel depending on the user's current viewpoint. As the user navigates through the scene, the LOD generation will shift its focus towards the new viewpoint in order to provide higher details as quickly as possible, while always maintaining real-time frame rates. We believe that this would be a massive improvement in workflows that involve looking at and working with large point clouds, since users will not have to wait minutes or hours until LOD structures are fully generated in advance.

WebGPU: Bringing it All to Web Browsers

Potree, one of our oldest and still ongoing projects, is a browser-based viewer for large point clouds. Development started in 2012 and is based on Javascript and WebGL. Unfortunately, WebGL has many limitations that prevent us from implementing the methods that were developed during this thesis in web browsers, most importantly compute shaders. Since around 2017, major browser developers started development on a successor to WebGL, called WebGPU. WebGPU is still in active development but Firefox, Chrome and Safari already offer experimental support.

We intent to use WebGPU as a chance to completely rewrite Potree from scratch with a modern graphics API. Our expectations are significantly faster rendering performances, especially if thousands of nodes are visible at once, higher quality, and the potential to implement and experiment with new ways to render point clouds that would not have been possible with WebGL due to the lack of compute shaders.

Bibliography

[3DE] USGS 3DEP LiDAR Point Clouds. `https://registry.opendata.aws/usgs-lidar/`, Accessed 2020.09.18.

[AMHH19] Tomas Akenine-Möller, Eric Haines, and Naty Hoffman. *Real-Time Rendering, Fourth Edition*. A. K. Peters, Ltd. / CRC Press, 2019.

[Are] Arena4d. `http://veesus.com/`, Accessed 2019.10.02.

[ASF+13] Murat Arikan, Michael Schwärzler, Simon Flöry, Michael Wimmer, and Stefan Maierhofer. O-snap: Optimization-based snapping for modeling architecture. *ACM Transactions on Graphics*, 32(1):6, 2013.

[ASP19] ASPRS. *LAS Specification 1.4 - R14*. The American Society for Photogrammetry & Remote Sensing (ASPRS), 3 2019. Rev. 14.

[AWLR17] Dmitri I. Arkhipov, Di Wu, Keqin Li, and Amelia C. Regan. Sorting with GPUs: A Survey, 2017.

[BCFB19] Vincenzo Barrile, Gabriele Candela, Antonino Fotia, and Ernesto Bernardo. Uav survey of bridges and viaduct: Workflow and application. In Sanjay Misra, Osvaldo Gervasi, Beniamino Murgante, Elena Stankova, Vladimir Korkhov, Carmelo Torre, Ana Maria A.C. Rocha, David Taniar, Bernady O. Apduhan, and Eufemia Tarantino, editors, *Computational Science and Its Applications – ICCSA 2019*, pages 269–284, Cham, 2019. Springer International Publishing.

[BHZK05] M. Botsch, A. Hornung, M. Zwicker, and L. Kobbelt. High-quality surface splatting on today's GPUs. In *Proceedings Eurographics/IEEE VGTC Symposium Point-Based Graphics, 2005.*, pages 17–141, 2005.

[BJ88] Sig Badt Jr. Two algorithms for taking advantage of temporal coherence in ray tracing. *The Visual Computer*, 4:123–132, 05 1988.

[BJFadH19] Sascha Brandt, Claudius Jähn, Matthias Fischer, and Friedhelm Meyer auf der Heide. Visibility-aware progressive farthest point sampling on the gpu. *Computer Graphics Forum*, 38(7):413–424, 2019.

[BK20] Pascal Bormann and Michel Krämer. A System for Fast and Scalable Point Cloud Indexing Using Task Parallelism. In Silvia Biasotti, Ruggero Pintus, and Stefano Berretti, editors, *Smart Tools and Apps for Graphics - Eurographics Italian Chapter Conference*. The Eurographics Association, 2020.

[Bou09] Christian Boucheny. *Visualisation scientifique de grands volumes de données: Pour une approche perceptive*. PhD thesis, Joseph Fourier University, 2009.

[Bow01] Kevin Bowers. Accelerating a particle-in-cell simulation using a hybrid counting sort. *Journal of Computational Physics*, 173:393–411, 11 2001.

[BWK02] Mario Botsch, Andreas Wiratanaya, and Leif Kobbelt. Efficient High Quality Rendering of Point Sampled Geometry. In P. Debevec and S. Gibson, editors, *Eurographics Workshop on Rendering*. The Eurographics Association, 2002.

[CCD+21] Pierfranco Costabile, Carmelina Costanzo, Gianluca De Lorenzo, Rosa De Santis, Nadia Penna, and Francesco Macchione. Terrestrial and airborne laser scanning and 2-d modelling for 3-d flood hazard maps in urban areas: new opportunities and perspectives. *Environmental Modelling & Software*, 135:104889, 2021.

[CKK18] Per Christensen, Andrew Kensler, and Charlie Kilpatrick. Progressive multi-jittered sample sequences. *Computer Graphics Forum*, 37(4):21–33, 2018.

[Clo] CloudCompare - 3D point cloud and mesh processing software. `https://www.danielgm.net/cc`, Accessed 2019.07.01.

[CLRS01] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Second Edition*. The MIT Press, 2001. Section 8.2.

[Coo86] Robert L. Cook. Stochastic sampling in computer graphics. *ACM Trans. Graph.*, 5(1):51–72, January 1986.

[CPP16] Rémi Cura, Julien Perret, and Nicolas Paparoditis. Implicit lod using points ordering for processing and visualisation in point cloud servers. *arXiv preprint arXiv:1602.06920*, 2016.

[DCSD02] O. Deussen, C. Colditz, M. Stamminger, and G. Drettakis. Interactive visualization of complex plant ecosystems. In *IEEE Visualization, 2002. VIS 2002.*, pages 219–226, Oct 2002.

[DD04] F. Duguet and G. Drettakis. Flexible point-based rendering on mobile devices. *IEEE Computer Graphics and Applications*, 24(4):57–63, 2004.

[DK18] Alexander Dieckmann and Reinhard Klein. Hierarchical Additive Poisson Disk Sampling. In *Vision, Modeling and Visualization*. The Eurographics Association, 2018.

[DMM+15] Michael Doneus, Igor Miholjek, Gottfried Mandlburger, Nives Doneus, Geert Verhoeven, Christian Briese, and Michael Pregesbauer. Airborne laser bathymetry for documentation of submerged archaeological sites in shallow water. In *ISPRS - International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, pages 99–107, 2015.

[DMS+18] Sören Discher, Leon Masopust, Sebastian Schulz, Rico Richter, and Jürgen Döllner. A point-based and image-based multi-pass rendering technique for visualizing massive 3d point clouds in vr environments. In *2018 Journal of WSCG*, volume 26, 06 2018.

[DRD18] Sören Discher, Rico Richter, and Jürgen Döllner. A scalable webgl-based approach for visualizing massive 3d point clouds using semantics-dependent rendering techniques. In *Proceedings of the 23rd International ACM Conference on 3D Web Technology*, Web3D '18, New York, NY, USA, 2018. Association for Computing Machinery.

[Dre07] Ulrich Drepper. What every programmer should know about memory, 2007.

[DVS03] Carsten Dachsbacher, Christian Vogelgsang, and Marc Stamminger. Sequential point trees. *ACM Transactions on Graphics*, 22:657, 07 2003.

[EGO+20] Philipp Erler, Paul Guerrero, Stefan Ohrhallinger, Michael Wimmer, and Niloy Mitra. Points2surf: Learning implicit surfaces from point clouds. In Andrea Vedaldi, Horst Bischof, Thomas Brox, and Jan-Michael Frahm, editors, *Computer Vision – ECCV 2020*, volume 12350 of *Lecture Notes in Computer Science*, pages 108–124, Cham, October 2020. Springer International Publishing.

[ELPZ97] Yuval Eldar, Michael Lindenbaum, Moshe Porat, and Yehoshua Zeevi. The farthest point strategy for progressive image sampling. *IEEE transactions on image processing : a publication of the IEEE Signal Processing Society*, 6:1305–15, 02 1997.

[Enta] Entwine. `https://entwine.io/`, Accessed 2020.06.29.

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

[FTB16] Jörg Futterlieb, Christian Teutsch, and Dirk Berndt. Smooth visualization of large point clouds. *IADIS International Journal on Computer Science and Information Systems*, 11(2):146–158, 2016.

[GBKG04] Sören Grimm, Stefan Bruckner, Armin Kanitsar, and Meister Eduard Gröller. Memory efficient acceleration structures and techniques for cpu-based volume raycasting of large data. In C. Silva D. Silver, T. Ertl, editor, *Proceedings IEEE/SIGGRAPH Symposium on Volume Visualization and Graphics*, pages 1–8, October 2004.

[GFP+18] S. Gagliolo, R. Fagandini, D. Passoni, B. Federici, I. Ferrando, D. Pagliari, L. Pinto, and D. Sguerso. Parameter optimization for creating reliable photogrammetric models in emergency scenarios. *Applied Geomatics*, 10:501–514, 2018.

[GGS08] Diego González-Aguilera, Javier Gómez-Lahoz, and José Sánchez. A new approach for structural monitoring of large dams with a three-dimensional laser scanner. *Sensors*, 8(9):5866–5883, 2008.

[Gin] Ginkgo Maps, Landkarten Österreich, CC-BY-3.0. `http://ginkgomaps.com/landkarten_oesterreich.html`, Accessed 2020.06.05.

[GKLR13] Christian Günther, Thomas Kanzok, Lars Linsen, and Paul Rosenthal. A gpgpu-based pipeline for accelerated rendering of point clouds. *J. WSCG*, 21:153–161, 2013.

[GM04a] Enrico Gobbetti and Fabio Marton. Layered Point Clouds. In Markus Gross, Hanspeter Pfister, Marc Alexa, and Szymon Rusinkiewicz, editors, *SPBG'04 Symposium on Point - Based Graphics 2004*. The Eurographics Association, 2004.

[GM04b] Enrico Gobbetti and Fabio Marton. Layered point clouds: A simple and efficient multiresolution structure for distributing and rendering gigantic point-sampled models. *Computers and Graphics*, 28(6):815–826, December 2004.

[GZPG10] P. Goswami, Y. Zhang, R. Pajarola, and E. Gobbetti. High quality interactive rendering of massive point models using multi-way kd-trees. In *2010 18th Pacific Conference on Computer Graphics and Applications*, pages 93–100, Sept 2010.

[HHF+14] M. Hämmerle, B. Höfle, J. Fuchs, A. Schröder-Ritzrau, N. Vollweiler, and N. Frank. Comparison of kinect and terrestrial lidar capturing natural karst cave 3-d objects. *IEEE Geoscience and Remote Sensing Letters*, 11(11):1896–1900, 2014.

[Hoe14] Rama C. Hoetzlein. Fast fixed-radius nearest neighbors: Interactive million-particle fluids, 2014. GPU Technology Conference (GTC) 2014, Santa Clara, CA.

[HP07] Bernhard Höfle and Norbert Pfeifer. Correction of laser scanning intensity data: Data and model-driven approaches. *ISPRS Journal of Photogrammetry and Remote Sensing*, 62(6):415 – 433, 2007.

[HR08] Robert C. Hilldale and David Raff. Assessing the ability of airborne lidar to map river bathymetry. *Earth Surface Processes and Landforms*, 33(5):773–783, 2008.

[HSG+19] U. Herbig, L. Stampfer, D. Grandits, I. Mayer, M. Pöchtrager, Ikaputra, and A. Setyastuti. Developing a monitoring workflow for the temples of java. *ISPRS - International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, 4215:555–562, 2019.

[Ise13] Martin Isenburg. Laszip: lossless compression of lidar data. *Photogrammetric Engineering & Remote Sensing*, 79, 2013.

[Jev92] David A. Jevans. Object space temporal coherence for ray tracing. In *Proceedings of the Conference on Graphics Interface '92*, pages 176–183, San Francisco, CA, USA, 1992. Morgan Kaufmann Publishers Inc.

[Jip19] Jippe van der Maaden. Vario-scale visualization of the AHN2 point cloud. Master's thesis, Delft University of Technology, Postbus 5, 2600 AA Delft, Netherlands, 2019.

[KB04] Leif Kobbelt and Mario Botsch. A survey of point-based techniques in computer graphics. *Computers & Graphics*, 28(6):801 – 814, 2004.

[KCODL06] Johannes Kopf, Daniel Cohen-Or, Oliver Deussen, and Dani Lischinski. Recursive wang tiles for real-time blue noise. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2006)*, 25(3):509–518, 2006.

[KJWX19] L. Kang, J. Jiang, Y. Wei, and Y. Xie. Efficient randomized hierarchy construction for interactive visualization of large scale point clouds. In *2019 IEEE Fourth International Conference on Data Science in Cyberspace (DSC)*, pages 593–597, 2019.

[KKSS18] Michael Kenzel, Bernhard Kerbl, Dieter Schmalstieg, and Markus Steinberger. A high-performance software graphics pipeline architecture for the gpu. *ACM Trans. Graph.*, 37(4):140:1–140:15, July 2018.

[KLRS04] A. Kolb, L. Latta, and C. Rezk-Salama. Hardware-based simulation and collision detection for large particle systems. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*,

HWWS '04, page 123–131, New York, NY, USA, 2004. Association for Computing Machinery.

[KLZ+18] Moritz Kirsch, Sandra Lorenz, Robert Zimmermann, Laura Tusa, Robert Möckel, Philip Hödl, René Booysen, Mahdi Khodadadzadeh, and Richard Gloaguen. Integration of terrestrial and drone-borne hyperspectral and photogrammetric sensing methods for exploration mapping and mining monitoring. *Remote Sensing*, 10(9):1366, 2018.

[Knu98] Donald E. Knuth. *The Art of Computer Programming*, volume 3. Addison-Wesley, 1998. Section 5.2, Algorithm D.

[Kra18] Manuel Kraemer. Accelerating Your VR Games with VRWorks. NVIDIA's GPU Technology Conference (GTC), 2018.

[KW03] J. Kruger and R. Westermann. Acceleration techniques for gpu-based volume rendering. In *IEEE Visualization, 2003. VIS 2003.*, pages 287–292, 2003.

[LAS] Lastools. `https://rapidlasso.com/lastools`, Accessed 2020.01.22.

[LC87] William E. Lorensen and Harvey E. Cline. Marching cubes: A high resolution 3d surface construction algorithm. *SIGGRAPH Comput. Graph.*, 21(4):163–169, August 1987.

[Lei13] Kurt Leimer. External sorting of point clouds, September 2013. Bachelor Thesis.

[LKG+16] P. Ljung, J. Krüger, Meister Eduard Gröller, Markus Hadwiger, C. Hansen, and Anders Ynnerman. State of the art in transfer functions for direct volume rendering. *Computer Graphics Forum (2016)*, 35(3):669–691, 2016.

[LOM+20] Haicheng Liu, Peter Oosterom, Martijn Meijers, Xuefeng Guan, Edward Verbree, and Mike Horhammer. Histsfc: Optimization for nd massive spatial points querying, 06 2020.

[LRC+03] David Luebke, Martin Reddy, Jonathan D. Cohen, Amitabh Varshney, Benjamin Watson, and Robert Huebner. *Level of Detail for 3D Graphics*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2003.

[LW85] Marc Levoy and Turner Whitted. The use of points as a display primitive. Technical report, University of North Carolina at Chapel Hill, 1985.

[MAPV15] Mathieu Le Muzic, Ludovic Autin, Julius Parulek, and Ivan Viola. cellVIEW: a Tool for Illustrative and Multi-Scale Rendering of Large Biomolecular Datasets. In Katja Bühler, Lars Linsen, and Nigel W. John,

editors, *Eurographics Workshop on Visual Computing for Biology and Medicine*, pages 61–70. EG Digital Library, The Eurographics Association, September 2015.

[MMS+16] Mathieu Le Muzic, Peter Mindek, Johannes Sorger, Ludovic Autin, David Goodsell, and Ivan Viola. Visibility equalizer: Cutaway visualization of mesoscopic biological models. *Computer Graphics Forum*, 35(3), 2016.

[MRVvM+15] Oscar Martinez-Rubi, Stefan Verhoeven, M. van Meersbergen, Markus Schütz, Peter van Oosterom, Romulo Goncalves, and T. P. M. Tijssen. Taming the beast: Free and open-source massive point cloud web visualization. 11 2015. Capturing Reality Forum 2015, Salzburg, Austria.

[OKWM12] Ayrton Oliver, Steven Kang, Burkhard C. Wünsche, and Bruce MacDonald. Using the kinect as a navigation sensor for mobile robotics. In *Proceedings of the 27th Conference on Image and Vision Computing New Zealand*, pages 509–514, 2012.

[OPA] Opals - orientation and processing of airborne laser scanning data. `https://opals.geo.tuwien.ac.at/html/stable/index.html`, Accessed 2019.06.26.

[Ope] Openvr. `https://github.com/ValveSoftware/openvr`. Accessed 2018.11.28.

[OPH+10] Thomas Ortner, Gerhard Paar, Gerd Hesina, Robert Tobler, and Bernhard Nauschnegg. Towards true underground infrastructure surface documentation. *Proceedings of CORP 2010*, pages 783–792, 2010.

[PJW12] Reinhold Preiner, Stefan Jeschke, and Michael Wimmer. Auto splats: Dynamic point cloud visualization on the gpu. In H. Childs and T. Kuhlen, editors, *Proceedings of Eurographics Symposium on Parallel Graphics and Visualization*, pages 139–148. Eurographics Association 2012, May 2012.

[PMOK14] N. Pfeifer, G. Mandlburger, J. Otepka, and W. Karel. Opals - a framework for airborne laser scanning data analysis. *Computers, Environment and Urban Systems*, 45:125 – 136, 2014.

[Pot] Potree. `http://potree.org`, Accessed 2020.05.27.

[Pre12] Jeff Preshing. How to generate a sequence of unique random integers, 2012. `https://preshing.com/20121224/how-to-generate-a-sequence-of-unique-random-integers`, Accessed 2019.09.16.

[PTC17] Kevin Ponto, Ross Tredinnick, and Gail Casper. Simulating the experience of home environments. In *2017 International Conference on Virtual Rehabilitation (ICVR)*, pages 1–9, June 2017.

[PVM+20] Florent Poux, Quentin Valembois, Christian Mattes, Leif Kobbelt, and Roland Billen. Initial user-centered design of a virtual reality heritage system: Applications for digital tourism. *Remote Sensing*, 12(16):2583, Aug 2020.

[PZvBG00] Hanspeter Pfister, Matthias Zwicker, Jeroen van Baar, and Markus Gross. Surfels: Surface elements as rendering primitives. In *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '00, pages 335–342, New York, NY, USA, 2000. ACM Press/Addison-Wesley Publishing Co.

[RD10] Rico Richter and Jürgen Döllner. Out-of-core real-time visualization of massive 3d point clouds. In *Proceedings of the 7th International Conference on Computer Graphics, Virtual Reality, Visualisation and Interaction in Africa*, AFRIGRAPH '10, page 121–128, New York, NY, USA, 2010. Association for Computing Machinery.

[Ree15] Nathan Reed. Depth precision visualized, July 2015. `https://developer.nvidia.com/content/depth-precision-visualized`, Accessed 2021.03.05.

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

[RL01] Szymon Rusinkiewicz and Marc Levoy. Streaming QSplat: A Viewer for Networked Visualization of Large, Dense Models. In *Proceedings of the 2001 Symposium on Interactive 3D Graphics*, I3D '01, page 63–68, New York, NY, USA, 2001. Association for Computing Machinery.

[SAMGA+20] Luis Javier Sánchez-Aparicio, Maria-Giovanna Masciotta, Joaquín García-Alvarez, Luís F. Ramos, Daniel V. Oliveira, José Antonio Martín-Jiménez, Diego González-Aguilera, and Paula Monteiro. Web-gis approach to preventive conservation of heritage buildings. *Automation in Construction*, 118:103304, 2020.

[SBV+13] Nikolaus Studnicka, Christian Briese, Geert Verhoeven, Gerald Zach, and Camillo Ressl. The roman heidentor as study object to compare mobile laser scanning data and multi-view image reconstruction. In *10th International Conference on Archaeological Prospection, Proceedings*, pages 25–28, 2013.

[Sch14] Claus Scheiblauer. *Interactions with Gigantic Point Clouds*. PhD thesis, Institute of Computer Graphics and Algorithms, Vienna University of Technology, Favoritenstrasse 9-11/E193-02, A-1040 Vienna, Austria, 2014.

[Sch16] Markus Schütz. Potree: Rendering large point clouds in web browsers. Master's thesis, Institute of Computer Graphics and Algorithms, Vienna University of Technology, Favoritenstrasse 9-11/E193-02, A-1040 Vienna, Austria, September 2016.

[SD01] Marc Stamminger and George Drettakis. Interactive sampling and rendering for complex and procedural geometry. In *Proceedings of the 12th Eurographics Conference on Rendering*, EGWR'01, page 151–162, Goslar, DEU, 2001. Eurographics Association.

[SKW19] Markus Schütz, Katharina Krösl, and Michael Wimmer. Real-time continuous level of detail rendering of point clouds. In *2019 IEEE Conference on Virtual Reality and 3D User Interfaces*, pages 103–110. IEEE, March 2019.

[SMOW20] Markus Schütz, Gottfried Mandlburger, Johannes Otepka, and Michael Wimmer. Progressive real-time rendering of one billion points without hierarchical acceleration structures. *Computer Graphics Forum*, 39(2):1–1, May 2020.

[SMSV10] Betty Sovilla, Jim N. McElwaine, Mark Schaer, and Julien Vallet. Variation of deposition depth with slope angle in snow avalanches: Measurements from vallée de la sionne. *Journal of Geophysical Research: Earth Surface*, 115(F2), 2010.

[SOW20] Markus Schütz, Stefan Ohrhallinger, and Michael Wimmer. Fast out-of-core octree generation for massive point clouds. *Computer Graphics Forum*, 39(7), 2020.

[SPF+17] M. Sturari, M. Paolanti, E. Frontoni, A. Mancini, and P. Zingaretti. Robotic platform for deep change detection for rail safety and security. In *2017 European Conference on Mobile Robots (ECMR)*, pages 1–6, 2017.

[SRS+19] Harald Steinlechner, Bernhard Rainer, Michael Schwärzler, Georg Haaser, Attila Szabo, Stefan Maierhofer, and Michael Wimmer. Adaptive point-cloud segmentation for assisted interactions. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, page 14, 2019.

[SW11] Claus Scheiblauer and Michael Wimmer. Out-of-core selection and editing of huge point clouds. *Computers & Graphics*, 35(2):342–351, April 2011.

[SW19] Markus Schütz and Michael Wimmer. Rendering point clouds with compute shaders, November 2019. Poster presented at SIGGRAPH Asia (2019-11).

[SYM+11] Daniel Scherzer, Lei Yang, Oliver Mattausch, Diego Nehab, Pedro V. Sander, Michael Wimmer, and Elmar Eisemann. A Survey on Temporal Coherence Methods in Real-Time Rendering. In N. John and B. Wyvill,

editors, *Eurographics 2011 - State of the Art Reports*. The Eurographics Association, 2011.

[SZW09] Claus Scheiblauer, Norbert Zimmermann, and Michael Wimmer. Interactive domitilla catacomb exploration. In *10th VAST International Symposium on Virtual Reality, Archaeology and Cultural Heritage (VAST09)*, pages 65–72. Eurographics Association, September 2009.

[TBP16] R. Tredinnick, M. Broecker, and K. Ponto. Progressive feedback point cloud rendering for virtual reality display. In *2016 IEEE Virtual Reality (VR)*, pages 301–302, March 2016.

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

[USGb] USGS / Entwine. `https://usgs.entwine.io`, Accessed 2020.09.18.

[V8] V8 javascript engine. `https://v8.dev/`. Accessed 2018.11.28.

[Vla15] Alex Vlachos. Advanced vr rendering. Game Developers Conference, industry talk, March 2015. `https://www.gdcvault.com/play/1021771/Advanced-VR`, Accessed 2018.11.20.

[vO19] Peter van Oosterom. From discrete to continuous levels of detail for managing nd-pointclouds, June 2019. `https://www.gsw2019.org/speaker/peter-van-oosterom`, Accessed 2020.01.31.

[WBB+07] Michael Wand, Alexander Berner, Martin Bokeloh, Arno Fleck, Mark Hoffmann, Philipp Jenke, Benjamin Maier, Dirk Staneker, and Andreas Schilling. Interactive editing of large point clouds. In *SPBG*, 2007.

[WBB+08] Michael Wand, Alexander Berner, Martin Bokeloh, Philipp Jenke, Arno Fleck, Mark Hoffmann, Benjamin Maier, Dirk Staneker, Andreas Schilling, and Hans-Peter Seidel. Processing and interactive editing of huge point clouds from 3d scanners. *Computers & Graphics*, 32(2):204 – 220, 2008.

[WDP99] Bruce Walter, George Drettakis, and Steven Parker. Interactive rendering using the render cache. In Dani Lischinski and Greg Ward Larson, editors, *Rendering Techniques' 99*, pages 19–30, Vienna, 1999. Springer Vienna.

[WFP+01] Michael Wand, Matthias Fischer, Ingmar Peter, Friedhelm Meyer auf der Heide, and Wolfgang Straßer. The randomized z-buffer algorithm: Interactive rendering of highly complex scenes. In *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '01, page 361–370, New York, NY, USA, 2001. Association for Computing Machinery.

[WS06] Michael Wimmer and Claus Scheiblauer. Instant points: Fast rendering of unprocessed point clouds. In *Proceedings Symposium on Point-Based Graphics 2006*, pages 129–136. Eurographics, Eurographics Association, July 2006.

[YGW+15] Dong-Ming Yan, Jianwei Guo, Bin Wang, Xiaopeng Zhang, and Peter Wonka. A survey of blue-noise sampling and its applications. *Journal of Computer Science and Technology*, 30:439–452, 05 2015.

[Yuk15] Cem Yuksel. Sample elimination for generating poisson disk sample sets. *Computer Graphics Forum*, 34(2):25–32, May 2015.

[ZPVBG01] Matthias Zwicker, Hanspeter Pfister, Jeroen Van Baar, and Markus Gross. Surface splatting. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 371–378, 2001.

[ZPvG02] M. Zwicker, H. Pfister, J. van Baar, and M. Gross. Ewa splatting. *IEEE Transactions on Visualization and Computer Graphics*, 8(3):223–238, 2002.