Received December 3, 2021, accepted January 7, 2022, date of publication January 26, 2022, date of current version February 4, 2022.

Digital Object Identifier 10.1109/ACCESS.2022.3146768

Real-Time Rendering of Point Clouds With Photorealistic Effects: A Survey

PETRUS E. J. KIVI[1], MARKKU J. MÄKITALO[1], JAKUB ŽÁDNÍK[1], JULIUS IKKALA[1], VINOD KUMAR MALAMAL VADAKITAL[2], AND PEKKA O. JÄÄSKELÄINEN[1]
[1]Unit of Computing Sciences, Tampere University, 33014 Tampere, Finland
[2]Nokia Technologies, 33100 Tampere, Finland

Corresponding author: Markku J. Mäkitalo (markku.makitalo@tuni.fi)

This work was supported in part by the Electronic Components and Systems for European Leadership (ECSEL) Joint Undertaking (JU) through the European Union's Horizon 2020 Research and Innovation Program and The Netherlands, Czech Republic, Finland, Spain, and Italy, under Grant 783162 (FitOptiVis); in part by the European Union's Horizon 2020 Research and Innovation Program under Grant 871738 (CPSoSaware); and in part by the Academy of Finland under Grant 325530.

ABSTRACT Readily available RGB-D cameras in smart phones and improving 3D scanning technologies have made it possible to produce detailed point cloud and point-based models of real world objects even in real time. Rendering such models in high quality and at satisfactory frame rates is needed for realistic *extended reality* (XR) applications. This publication reviews real-time photorealistic point cloud rendering methods which directly ray trace or rasterize point cloud models, with an emphasis on ray tracing and real-time performance. We found that real-time direct point cloud ray tracing research has been focused on static non-animated content, and thus, open research possibilities include adapting modern dedicated ray tracing hardware for increased performance for animated and live captured scenes, and adding path tracing techniques to increase photorealistic effects in the rendering result. A categorization and discussion on the capabilities of state-of-the-art photorealistic point cloud rendering methods is presented by surveying both real-time and offline methods, which are assumed to become real-time capable with the advances in near-future hardware. Challenges and future trends are derived by comparing different rasterization and ray tracing methods as well as acceleration structures for point clouds in terms of produced rendering effects and speed.

INDEX TERMS Survey, point clouds, photorealistic rendering, ray tracing, real-time rendering, point-based models, acceleration, rasterization.

I. INTRODUCTION

Point cloud and point-based visualization have been studied for more than three decades. Levoy and Whitted suggested using points as a geometric primitive for rendering instead of polygonal meshes or parametric surfaces in 1985 [1]. The argument was that the number of geometric primitives in rendered scenes would keep increasing such that their projected sizes would decrease to sub-pixel areas in screen space, thus, justifying the use of point primitives instead. Since then, research on visualization of point cloud and point-based models has produced techniques like splatting [2], [3] and direct ray tracing of point cloud and point-based models [4], [5].

The challenge in point cloud visualization has been the lack of a continuous surface representation. This poses both benefits and disadvantages. On the one hand, point clouds are more flexible than connected meshes and can be used to store and visualize huge data sets with up to 100 million points in real time [6]. On the other hand, generating correct

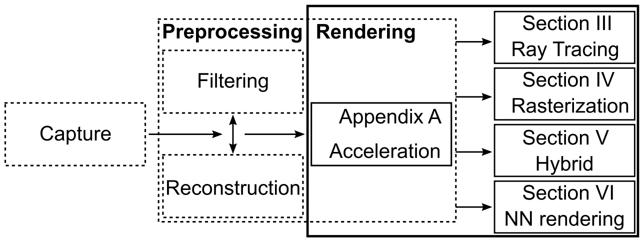

FIGURE 1. An overview schema of the different pipeline steps from point cloud capture to rendering. Within solid lines are the subjects covered and in dashed lines are the subjects omitted in this survey. The arrows represent the point cloud data flow. Generally, acceleration can be considered either preprocessing or rendering, but in our case, we only consider methods performant enough to reconstruct acceleration structures at rendering time.

The associate editor coordinating the review of this manuscript and approving it for publication was Gerardo Di Martino.

surface attributes between point cloud points in a rasterization pipeline needs special handling compared to the interpolation and intersection of values between vertices in a triangle or other polygonal-based model. This is also true when finding an intersection point between a viewing ray and the implied surface representation of a point cloud.

Direct point cloud rendering challenges can be circumvented by reconstructing the whole point cloud into a renderable surface representation such as a triangulated mesh. The capturing and reconstruction of real-world scenes especially with *RGB-depth* (RGB-D) cameras – reviewed in various publications [7]–[9] – has seen a lot of research after the seminal publication introducing KinectFusion [10]. Many of the fusion-based systems reconstruct the scenes directly into mesh representations with an intermediate scene description based on *signed distance fields* (SDF) at capture time [10]–[13]. However, full global reconstruction is computationally expensive especially on large models, and it can be wasteful if only parts of the scene are viewed or models become obsolete, for example in animated or streamed point cloud scenarios. Furthermore, recent advances in point cloud compression by the MPEG standardization group [14] have enhanced the possibilities of transferring raw point clouds efficiently, for example from point cloud capture sites to end users. This opens the possibility for doing the direct rendering effort at the user's end. Thus, the motivation behind this survey is the availability of methods specifically for direct real-time photorealistic rendering for surface point clouds.

A key aspect of photorealism is the detail level of the model which, in the case of point clouds, means the number of points used to represent a specific size object in the real world. Available models in repositories such as The Stanford 3D Scanning Repository [15] have a typical point count in the order of 10^6 and single detailed human-like models, such as Lucy, have up to 10^7 points. The previously mentioned KinectFusion and its successors work with up to 512^3 grid structures corresponding to almost 135 million potential data points, but most of the data entries are empty space, making it hard to illustrate their effective point resolution. *Level of detail* (LOD) systems and culling techniques can further reduce the number of points considered at rendering time, easing computationally expensive rendering methods like ray tracing. However, lowering detail and omitting points not inside a view frustum need to be used carefully in conjunction with ray tracing, because *global illumination* (GI) effects might be negatively impacted. A comprehensive review of LOD and culling techniques is out of the scope of this survey, but several notable examples are included.

For applications such as holoportation [16] and other telepresence systems [17], [18], the end-to-end latency from scene capture to rendered image frame in client scene is a motivating factor. Multiple human models may be transferred, leading to even more complex mesh or point cloud models with a lot of data points. These systems have shown real-time capabilities on large device clusters of up to 8 high-end PCs and multiple simultaneous capturing devices. Furthermore, these applications have focused on delivering a coherent scene model from the capturing site to the client rendering site. However, producing photorealistic lighting effects on the transferred models has not been considered. High-quality mesh and other surface generation methods for point clouds have been thoroughly surveyed [19]–[27]. Thus, the subject of surface reconstruction for point clouds is out of scope.

A. DEFINING THE RESEARCH PROBLEM

In the context of this survey, live-captured point clouds refer to streams of point clouds that may not have any coherence between consecutive frames. Contrary to this, synthetic skeletal animated models can reuse lower-level parts of an acceleration structure by applying the respective affine transforms directly to subsets of the data structure. However, such assumptions cannot be made for an unpredictable stream of points in a live scenario, which leads to the reconstruction of the acceleration structure for each frame.

A frame rate of over 75 FPS and an HD resolution of 1080p or more for interactive content in XR is needed for a comfortable viewing experience on *head mounted displays* [28]. These requirements are satisfied with modern virtual reality headsets with frame refresh rates of 80 to 144 Hz and up to 2K resolution per eye [29], [30]. Furthermore, as soft shadows and reflections tend to be ubiquitous in all rendering applications, a stricter demand for photorealistic rendering is applied. All of this should preferably be done in an end-to-end fashion, meaning that the rendering pipeline starts after the captured point cloud and ends in the photorealistically rendered frame. Thus, we derive the following research question for our survey:

What is the state-of-the-art technique for photorealistic end-to-end direct point cloud rendering for a high-quality human-sized model (10^7 points), in 75 FPS, and a resolution of 1080p on consumer hardware?

B. INCLUSION CRITERIA

There is no objective method found in the literature to classify applications as real-time or interactive. For this survey, however, we defined a lower bound for *real-time* frame rate for the surveyed methods as at least 10 FPS, *interactive* frame rate as at least 1 FPS, and *offline* to refer to the rest. These definitions ensure the inclusion of methods measured on older hardware in the survey.

Point cloud ray tracing methods, including offline methods, were exhaustively surveyed and they were deemed to be real time or interactive if they achieved these results with at least a medium resolution of 512×512 and a small point cloud in the order of 10^3 points. Computationally demanding effects, such as caustics, were not required of the methods because they require pre-calculations even in mesh-based ray tracing. Similar requirements were demanded from rasterization papers with the exception of a point cloud size in the order of 10^4 points (to accommodate the needed resources for photorealistic effects implemented with rasterization) and at least some photorealistic effects supported, such as reflections or refractions. We estimated that these requirements, considerably smaller compared to the research question, correspond to the gain in performance when extrapolating from older to modern hardware.

C. CONTRIBUTIONS

This survey provides the following novel contributions to the existing body of point cloud rendering literature:

- An exhaustive survey on ray and path tracing methods for surface point cloud rendering (Section III).
- A review of real-time and interactive rasterization, hybrid (combining rasterization and ray tracing), and point-based neural rendering methods that exhibit photorealistic rendering effects (Sections IV, V, and VI).

Additionally, we provide an overview of acceleration methods designed and applicable for real-time and interactive photorealistic point cloud rendering that could be utilized in an animated or live-captured point cloud scene (Appendix A). Furthermore, the computational capabilities of older surveyed methods are analyzed on and extrapolated to modern hardware in Appendix B.

D. STRUCTURE OF THE SURVEY

The rest of the survey is structured in the following way. Section II covers surveys and reviews related to real-time photorealistic point cloud rendering with justifications on what this survey provides compared to previous work on the subject. The main contribution of this survey, namely, reviewing real-time, interactive, and near interactive methods for photorealistic point cloud rendering is presented in Sections III, IV, V, and VI. A discussion on current capabilities and trends in real-time photorealistic point cloud rendering as well as on future research possibilities is given in Section VII together with answers to the research question. We conclude the publication in Section VIII.

Additionally, Appendix A surveys acceleration methods designed or easily applicable for point cloud rendering, and Appendix B extrapolates the computational performance of older surveyed methods to modern hardware.

II. RELATED SURVEYS AND REVIEWS

In this section, surveys and reviews that relate to point cloud rendering and acceleration structures for point clouds are covered. The goal is to briefly summarize what the previous survey and review publications have covered and what this survey contributes to the existing survey literature. Furthermore, parallel surveys related to the point cloud processing pipeline from capture to pre-processing (depicted inside dashed lines in Figure 1) are presented with justifications why the subject matter in those surveys is not covered in this survey. Finally, the contributions of this survey compared to the existing survey literature are reiterated.

Related surveys and reviews include general point cloud visualization mostly focused on rasterization-based methods and techniques [31]. Specifically close to this survey is the publication comparing different rasterization-based methods to meshed model rendering for real time point cloud visualization [32]. Our contribution, compared to [32], is the review of photorealistic methods of point cloud rendering and the review of modern methods, because the publication in [32] is from 2004. Additionally, a book on point-based graphics is available [33]. Participating media uses an underlying particle- or point-based schema, but generally they approximate the effect of particles in the air as a density texture instead of a point cloud (survey available in [34]). RGB-D image registration techniques are surveyed in [7] and *simultaneous localization and mapping* (SLAM) methods are reviewed in [35], both of which are out of scope in this survey. The subject of capturing and scanning real world objects has been extensively surveyed with a focus on RGB-D images in [8] and VR-centered capturing in [9]. The capturing and scanning of point clouds is also out of scope in this survey.

Methods concerning *smoothed-particle hydrodynamics* (SPH) fluid simulation and rendering have been reviewed in [36]. SPH fluid rendering is closely related to surface point cloud rendering as SPH methods consider the fluid as single particles inside a fluid volume interacting with each other. Compared to surface point clouds, SPH fluid particles have attributes from the simulation, such as mass and velocity, rarely present in generic point clouds. Thus, by-product information like particle radii can be used in the rendering of particles. Although this survey considers specific simulation techniques in SPH as being out of scope, the rendering of SPH is reviewed. In [36], an overview of the SPH rendering methods is given, whereas our survey concentrates on methods with applicability to point cloud rendering and provides a more in-depth analysis of the rendering features and computational performance of these methods.

Augmented (AR) and *mixed reality* (MR) rendering systems strive to visualize real and virtual elements seamlessly together. The immersiveness is enhanced by producing realistic interactions between real and virtual objects, for example correct occlusions and lighting effects between objects. Generally, point cloud rendering also tries to visualize objects in different locations and lighting than their original environment. However, the virtual objects planted inside AR and MR environments are almost exclusively in a mesh representation and, as such, the rendering of the objects is done by mesh-based methods. Thus, the review of AR and MR methods in general have been excluded from this survey, and we refer the reader to two state-of-the-art surveys on the subject [37], [38].

III. RAY TRACING POINT CLOUDS

In the literature, three main methods for direct point cloud ray tracing have been established. These methods are illustrated in Figure 2. Cone and cylinder/beam tracing are methods which circumvent the problem of individual points having no explicit surface to intersect, by expanding rays into a volume object able to capture points inside volume segments of the ray. Implicit and isosurface approaches use a metric, such as Euclidian distance, to evaluate the proximity of a surface implied by points near a sampling point along an intersecting ray. Finally, the *moving least squares* (MLS)–based method produces a local polynomial fit, such as a second-order curved surface, by projecting a sampling point onto a weighted local reference plane, based on which the polynomial fit is done [39]. This polynomial surface can be iteratively or non-iteratively constructed and intersected by a viewing ray. The main difference between general implicit surfaces and MLS-based surfaces is that implicit surfaces are locally linear, whereas MLS surfaces typically have a higher-order local approximation.

The previous book on point-based graphics from 2007 [33] has already covered many of the earlier ray tracing methods, which we briefly summarize first. Beam tracing point cloud was one the first methods to produce Whitted style reflective and refractive rendering [4] for point clouds. A predefined beam radius based on point density was used and points, accelerated by an octree data structure and falling inside a beam section within a leaf node, were weighted based on orthogonal distance to beam center. Gaussian weighting ensured smoothly blended attribute values. In [40], the idea was expanded with cone tracing in order to support advanced effects such as soft shadows. They used a dual resolution model of a scene with high-resolution triangles and lower-resolution point primitives in an octree data structure. A more reconstruction-oriented approach used actual rays instead of beams or cones to intersect a point cloud as locally reconstructed MLS surfaces [39]. Points were stored in a *bounding sphere hierarchy* (BSH), and the MLS surfaces were reconstructed using the centers of the leaf node spheres as a reference point, providing a cached reconstruction for multiple instances of intersection tests. Previous methods supported only static point clouds, which was addressed in [41] for animated and deforming scenes. Surfels (points with radii and surface normal attributes) were lazily updated based on underlying simulation nodes, i.e., only surfels visible to primary and secondary rays were updated at rendering time. Finally, the first interactive ray tracing method for static point clouds represented as surfels was introduced in [5]. A k-d tree was first traversed to a leaf node where implicit surface values were evaluated at constant intervals. The interval where a sign change occurred was further sub-divided until a threshold value was reached and an intersection was registered at an interpolated value. A commonality between these methods was a CPU implementation without utilization of the GPU.

In this section, we review the various ray tracing methods not covered in other literature that are capable of producing photorealism in a scene by rendering point clouds in real time. The focus is on surveying purely ray-tracing-based methods exhibiting one or more photorealistic effects such as reflections, hard or soft shadows, refractions, and GI.

A. OFFLINE METHODS

The methods introduced in this section do not achieve interactive frame rates of larger than 1 FPS with any of their reported point cloud sizes or screen pixel amounts. These include the already covered methods in the point-based graphics book [4], namely [39]–[41]. However, most of the methods are from the early 2000s and they might be implementable in real time on modern hardware. We discuss the possible application of these methods further in Sections III-D and VII.

In [42], a splat-based ray tracing scheme was utilized. A raw point cloud, possibly accompanied with surface normals, was used to generate intersectable surface splats. Splats were created iteratively by expanding a linear plane fit onto the point cloud until an error bound was violated. A normal gradient field on each splat was solved as a linear system of two unknown major gradient directions given the point normals and locations within the neighborhood of the splat. If normals were not present in the raw point cloud, then a least squares plane fit on the nearest neighbors of a given point was done as a preprocess and the direction of the plane was used as the surface normal of the given point. The splat generation took 4 seconds for the Fuel model of 35 thousand points transformed into 28 thousand splats and 85 seconds for the Buddha model of 544 thousand points transformed into 384 thousand splats. The actual Whitted style ray tracing accelerated by a dynamic splat octree with hard shadows, reflections, and refractions took 84 seconds for 28 thousand splats and 408 seconds for 384 thousand splats. In both cases a resolution of 1200×1200 and 1 sample per pixel (spp) with two ray bounces was used. A 3.06 GHz Intel Xeon CPU was used for all processing.

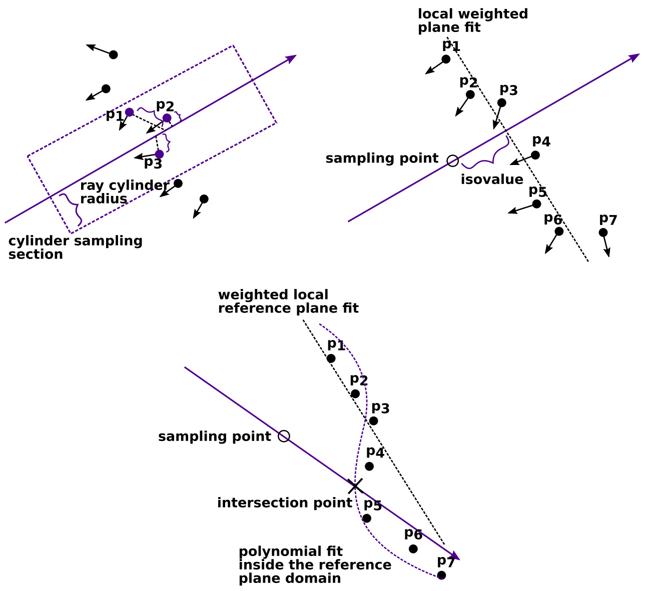

FIGURE 2. The three major methods for finding an intersections between a ray and a point cloud. Top left: Point density–based cone and cylinder/beam tracing. Top right: Implicit surface and isosurface evaluation; isovalue is the distance between the sampling point along the ray and the surface implied by the points. Bottom: *Moving least squares* (MLS)–based local polynomial surface reconstruction; the intersection point is found by first projecting the sampling point onto a reference plane, and then fitting a local polynomial based on that.

B. INTERACTIVE METHODS

In [43], a LOD system accelerated with a k-d tree was generated to ray trace splat primitives acquired from a point cloud. Later, their compression-based method was able to achieve ray tracing at interactive frame rates of 0.5–2 FPS with up to 28 million points [44]. The used compression method generated a dictionary of MLS patches of points with orientation and location placed in an instanced k-d tree for accelerated ray-patch intersection. Once a k-d tree leaf node with a few patches was traversed each patch was intersected by iteratively refining MLS surface intersections until a sufficient error tolerance was achieved and all patch intersection were interpolated based on patch center distance. The performance with huge point cloud models was due to the instanced patch-based compression being able to fit the compressed point cloud in GPU memory reducing CPU-GPU memory transfers at rendering time. The number of spp and supported rendering effects were not reported. However, based on the rendered images only correct direct lighting without stochastic effects were presented and thus 1 spp is implied. Several hours of preprocessing time was reported for the compression encoding and patch-based k-d tree construction.

C. REAL-TIME METHODS

As discussed before, the method in [5] was the first CPU-based method that achieved interactive frame rates on plausibly sized point clouds. However, the first parallel real-time GPU-based point cloud ray tracer was published in [45]. As they did not use any acceleration structure for the secondary rays, they had to limit the ray depth of reflection ray traversal to achieve reflections and refractions at interactive frame rates. Contrary to the implicit surface approach, a local MLS surface reconstruction was used to construct an intersectable surface along the primary rays, which was a more suitable algorithm for parallel execution. For secondary rays, a coarse intersection point was generated with a single pass of the MLS algorithm. A follow up publication [46] improved the method by introducing a grid-based acceleration structure and refraction rays.

A more traditional approach of using splats as the intersectable primitive was done in [47] with a performant splat octree method executed on the GPU. Up to 4 spp for primary rays and 10 bounces for GI effects were used. However, the octree was built in a preprocessing step and the execution time was not discussed. The authors in [48] used the same method of textured splats stored in an octree and used normal shading for primary lighting and tracing rays to generate shadows. The splat octree differed from the implicit and MLS surface methods in that the intersection querying was done on the splats themselves and not by ray marching implicit values or reconstructing local surfaces. However, the challenge of producing the input splats from a raw point cloud was not discussed.

The same authors later improved their splat octree methods by transforming the splats into an implicit representation. This enabled real-time performance with reflections, refractions, and shadows [49]. Exactly like in [5], isosurface values of the implicit function were stored in the leaf node corners and interpolated when the ray intersections were queried. Despite real-time performance claims, constructing the octree and the implicit surface values in the corner points was a preprocess step and took tens of seconds, making the method applicable in real time only for static point clouds. Precomputed caustic effects were later added into a similar octree ray tracing structure in [50]. A caustic sample map was stored in the octree leaf node corners the same way as the implicit function values and the luminance values were interpolated at the intersection point to produce realistic caustics. However, dynamic point clouds and lighting were not supported in real time. Finally, the method in [51] uses splat primitives, with a k-d tree to accelerate the intersection testing iteratively, producing an intersection point and an average surface normal from neighboring points at interactive frame rates.

D. SUMMARY AND DISCUSSION

Real-time direct surface point cloud ray tracing methods mostly require oriented points with radii (splats) as input for their rendering pipeline, implying a splat generation algorithm for surface point clouds. The authors in [44] generate their MLS-based surface from raw point clouds but require heavy preprocessing and compression taking up to several hours. Preprocessing is utilized by all methods ranging from 0.17 FPS (5.9 seconds) of point-set surface construction on a $32^3 = 32768$ voxel grid [45] to approximately 20 to 180 seconds for isosurface value octree construction [48], [50]. Consequently, the methods were only capable of ray tracing static scenes in real time with the reported hardware.

We identify the state-of-the-art methods in real-time photorealistic point cloud rendering in terms of rendering speed and the amount of supported photorealistic effects and summarize all of the methods in Table 1. In order to make the results as comparable as possible we have to account for the three major factors affecting the rendering time: the number of points/particles in the scene, the screen resolution, and the number of spp. All methods use 1 spp except [52] and [47], which use 2 spp for particle inter-reflections and 4 spp for supersampling, respectively. This makes most of the methods' spp counts fairly compatible when assessing performance. We have categorized methods achieving real time ($\geq$ 10FPS) or interactive real time ($\geq$ 1FPS), mutually exclusively, and reported the closest maximum number of points still renderable in real-time or interactive frame rates. Further details on scalability based on resolution and point count as well as the used rendering hardware are presented in Table 5.

Surface point cloud ray tracing methods have challenges achieving higher real-time frame rates. Generally, the methods have a medium resolution of approximately 512×512 and work on point clouds ranging from a magnitude of 10^3 to 10^6 points. The most performant method in this category is presented in [47], which achieves 55 FPS with a resolution of 512×512 and 10^6 points. Furthermore, all the methods in this category have omitted the time needed for preprocessing the point cloud, including surface normal and point radius generation.

IV. RASTERIZATION

In this section, rasterization-based methods for real-time photorealistic point cloud rendering from the 2010s are surveyed. With the increasing computational power of GPUs and the introduction of more programmable processing units for GPGPU computing, sophisticated real time rendering algorithms for point clouds could be developed further in the new decade. Nevertheless, the requirement for photorealism is relaxed compared to ray tracing methods as rasterization does not inherently support for example shadows and reflections, and separate shading passes have to be executed to generate these effects. The goal is to review methods that at least support some sort of realistic shading model. Additionally, the early general point cloud rasterization methods from the 2000s are briefly reviewed for context.

1) EARLY POINT CLOUD SPLATTING IN THE 2000s

Seminal methods for the popular splatting method (Figure 3) were developed in both [2] and [53]. Even though they were not focused on photorealistic effects as such, they paved the way for other splat-based methods. Consequently in [54], radiance (direct lighting and shadows) and irradiance (indirect lighting) were stored into a splat cache on the CPU/RAM and an octree splat cache on the GPU, respectively. It was designed for splatting the radiance and irradiance on a triangle mesh model, but it was extendable to point cloud models if a method for point cloud splat generation was available. For proper realistic GI, splat radii, normals, normal gradients/interpolation, and material properties (BRDF definitions) were needed. The irradiance cache was rasterized on a 60×60 unit hemisphere covering a single splat. Instead of targeting photorealistic effects as such, fast splat rendering was used in [55]. The method assumed a readily available elliptical splat model with surface normals and ellipse axis radii, but the generation time of such a model from a raw point cloud was not discussed. Compared to [54], *elliptical weighted average* (EWA) filtering between the splats in object space and approximate screen space anti-aliasing for edge aliasing was added for rendering quality enhancement. However, the method only supported traditional shading and shadow maps and extendability for other photorealistic rendering effects was not discussed. The method achieved a rendering speed of up to 17.5 million splats per second. A similar splatting method implemented on the GPU added transparent, reflective, and refractive effects in [56] with *deferred blending*, which already achieved interactive frame rates with up to $2 \cdot 10^6$ points.

In order to reach real-time frame rates, a LOD system was introduced for rendering point clouds of up to 10^7 number of points [57]. Instead of accurate elliptical splats in the previous method, they used screen space splatting with pixel sized splats and a nested octree structure implemented as a parallel memory optimized sequential point tree, which sequentialized a subset of vertices in the octree sorted by importance into a buffer. Nevertheless, its applicability to dynamic scenes was hindered by the construction times of the sequential point tree [58], which took from minutes up to hours on the CPU with point cloud sizes of 10^7 to 10^8. Furthermore, only crude primary color shading was used due to the lack of point normals. These problems were alleviated in [59] with a fully real-time splat-based rendering method. The method used a variation of the PSS definition called the *algebraic point set surface* (APSS), which fit spheres with varying curvature instead of polynomials onto the surface defined by the circular splats. To further enhance the rendering quality, the splats were upsampled before projection by placing a constant amount of sub-splats on regular intervals inside a square plane fit around the super-splat. Several acceleration structures were utilized for efficient local reconstruction, and their optimal redundant pyramid data structure yielded a throughput of 100 thousand points in about 2 milliseconds. Compared to using splats directly, APSS provided a smoother reconstruction than using for example the previously popular EWA filtering for splats.

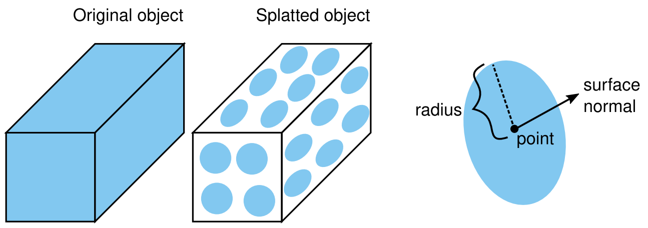

FIGURE 3. **Splats are a prominently used point description especially in point cloud rasterization. Points need a radius (dashed line) and a surface normal (arrows) in order to be depicted as splats.**

2) MODERN POINT CLOUD RASTERIZATION PIPELINES IN THE 2010s

Adopting the previous ideas of using splat-based rendering for point data, a multi-layered splatting method with spheres for particle-based liquid rendering was introduced in [60]. Effectively, using spheres provided a similarly smoothed effect for liquid surfaces as the APSS approach, but needed additional depth map smoothing with an iterative curvature flow filter. The authors rendered several layers of the liquid in order to produce separated foam and normal perturbed refractions. In [61], the authors took the idea of multi-layer depth map rendering further by using up to four layers for transparency with hard and soft shadows at real-time frame rates. However, they returned to splat primitives and, thus, needed screen space hole and occlusion filling as well as edge-aware smoothing and anti-aliasing for consistent results. Another application area for the splat primitive was the detailed rendering of facial scans [62]. The authors adaptively distributed the splats depending on detail level and adjusted the splat sizes accordingly with screen space hole filling applied

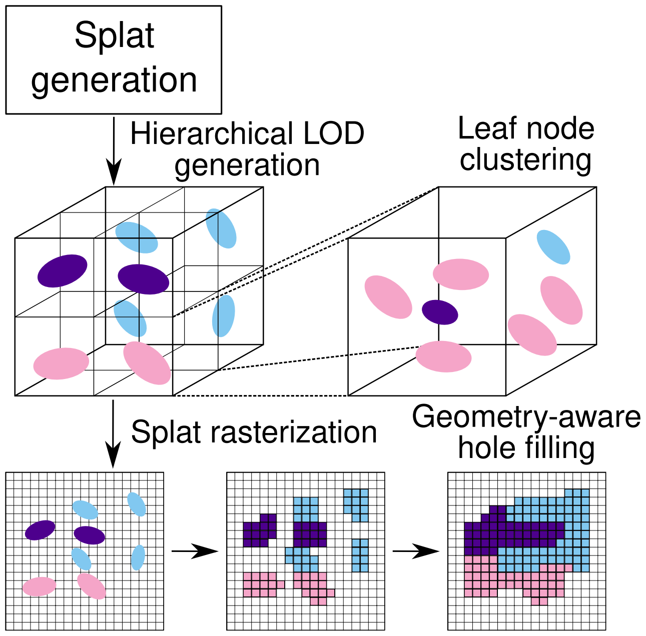

FIGURE 4. A typical point cloud rasterization pipeline needs a splat-based representation of the scene for sufficient surface coverage in rasterization. For large point cloud scenes, hierarchical data structures that cluster aggregate attributes from leaf node splats to parent nodes are utilized for efficient *level of detail* (LOD) rendering. Geometry-aware hole filling takes care of rendering artifacts in screen space.

accordingly. Self-shadowing with hard shadows on the face and up to six layers of sub-surface scattering was achieved in real time partly due to the reduced rendering effort with the adaptive splat distribution, but the preprocessing work entailed static models.

In [63], the authors demonstrated that a GPGPU-based point cloud rasterization pipeline can outperform a fixed-function pipeline optimized for triangle meshes. Specifically, they implemented an OpenCL pipeline tailored for point primitives, instead of relying on the more traditional OpenGL 4 pipeline. With a real laser scanning dataset having 138 million points, their implementation achieved 56 FPS on an AMD Radeon HD 7970 GPU, as opposed to their OpenGL baseline achieving only 5 FPS, both with a 1024×1024 resolution. However, the efficiency of their method relies on having to perform a large number of z-tests. Hence, the benefits of incorporating it into existing systems, where a LOD mechanism usually reduces the amount of z-tests, are not immediately evident. Moreover, their implementation utilized only the scanned geometry and color information, and did not focus on shading or producing any photorealistic effects.

With increasing point and splat counts in more intricate models, a LOD structure based on *multi-way k-d trees* (MWKD) was introduced [64] (depicted in Figure 4). The multi-way nature of the data structure was due to several feature discrimination strategies such as normal deviation clustering, entropy-based point reduction, and modified k-clustering. These were used for splat count reduction with a representative MWKD node and hashing as well as compression reduced the resulting model size further. Additionally, hierarchical interpolation between LOD levels increased smoothness and temporal reusage decreased costly out-of-core fetches. This approach was applied in an archaeological setting with features such as relighting and shadows in scenes with a torch tool and laser pointers with intersection testing [65].

The computational challenges of preprocessing for dynamic scenes were tackled in the AutoSplats method, which did not assume point radii or surface normals [3]. The authors generated object space oriented splats with a screen space k-NN search algorithm that iteratively grew or shrunk the considered area around points. The surface normals of points were estimated with PCA on k-NN points, after which a weighted average of the points was used for a plane fit. Basic shading was incorporated but other realistic rendering effects such as shadows were not present.

A. SUMMARY AND DISCUSSION

Surface point cloud rasterization methods are typically reliant on pregenerated splats, i.e., both pregenerated surface normals and point radii. The most performant one out of those showing one or more photorealistic rendering effects is the implementation in [65], which utilizes a combination of [55] and [64]: it achieves up to 100 FPS with a resolution of 1920×1080 and $14 \cdot 10^6$ points with $2 \cdot 10^6$ visible points. However, as discussed previously, the rasterization-based methods, with the exception of [61], only provide hard shadows and Phong shading, which is considerably less photorealistic than the effects produced by the ray tracing methods.

V. HYBRID RENDERING METHODS

Methods combining both rasterization and ray tracing for their rendering pipeline are reviewed in this section. Specifically, methods from the subject areas of point-based indirect illumination and SPH fluid rendering are able to produce photorealistic effects for point-based models, with a hybrid approach incorporating the benefits of both rasterization-based and ray-tracing-based techniques.

A. POINT-BASED GLOBAL ILLUMINATION

A precursor to the *point-based global illumination* (PBGI) method produced meshless radiosity and GI effects [66]. An intersectable model was sub-sampled with splats and stored into a multi-level R-tree with fine-to-coarse basis function approximations. Several ray traced bounces, including support for three bounces of glossy reflections or multi-light irradiance, were used to generate GI effects with irradiance gathering – the general concept behind PBGI is illustrated in Figure 5. The authors later increased the rendering time performance by constructing the R-tree and ray tracing single-bounce GI in a preprocessing step [67]. The original method [66] was also re-implemented on the GPU with a parallel implementation of the R-tree basis function hierarchy evaluation and ray tracing [68]. They achieved an order of magnitude faster rendering time for meshless radiosity GI, but due to the limitations of the GPU implementation, only the indirect lighting was splatted and ray traced, whereas the direct lighting was rasterized.

The original technical report on approximate PBGI introduced GI effects with environment maps and spherical harmonics projected onto surfels (surface splats) [69]. Surfels were stored in an octree for accelerated environment mapping and spherical harmonics. The method was reported to be extendable to any surface representation that can be transformed into surfels with the necessary attributes for photorealistic rendering effects. Because the spherical harmonics approach was an approximation for GI, only smooth effects such as glossy reflections, blurry refractions, and soft shadows were supported. Actual ray tracing was suggested as an extension to the rendering pipeline for sharp reflections, refractions, and shadows. Preprocessing took up to minutes, of which the generation of surfels took 18 seconds. An enhanced version significantly sped up the generation of surfels and the octree structure from mesh models [70]. Furthermore, the extendability of the method to pure point cloud models with rasterized primary visibility was explicitly discussed, but this approach assumed readily available surface normals and point radii from the raw point cloud.

In [71], the PBGI indirect lighting method was further improved in terms of image quality by introducing light voxels for volume data points, which resembled probe-based lighting. They added inward gathering and outward scattering of indirect light for realistic light interactions inside a volume point cloud and between the rest of the scene. Metric results showed a 2.1–4.3 percent deviation from a reference rendered with a Monte Carlo ray tracer, compared to 5.6–14.4 percent with the method in [69].

The computational overhead of octree construction for the surfels on the CPU was tackled in [72] with an out-of-core point sample octree construction algorithm based on efficient Morton code sorting. The octree provided a data structure utilized by several CPU cores in parallel for LOD ray tracing, producing GI effects like single-bounce diffuse inter-reflections, *ambient occlusion*, and high-dynamic-range environment map lighting. A similar PBGI implementation was presented in a short paper [73], which was focused on reducing the memory footprint of the point cloud size accompanying PBGI methods. They achieved up to two orders of magnitude smaller point clouds with a multi-resolution implementation of the octree structure. Instead of using explicit ray tracing for the indirect lighting, an intricate progressive rasterization with hemispherical microbuffers was used for radiosity gathering on surfels [74]. The idea was that reduced microbuffer sizes of 32 × 32 were sufficient to capture indirect lighting and glossy-to-glossy reflections with additional glossy environment lobes. Furthermore, surfels were clustered via k-means such that random samples could be used as representatives of a clusters contribution for performance gains. The microbuffer approach was further advanced with LOD-like k-clustering of near and far surfels

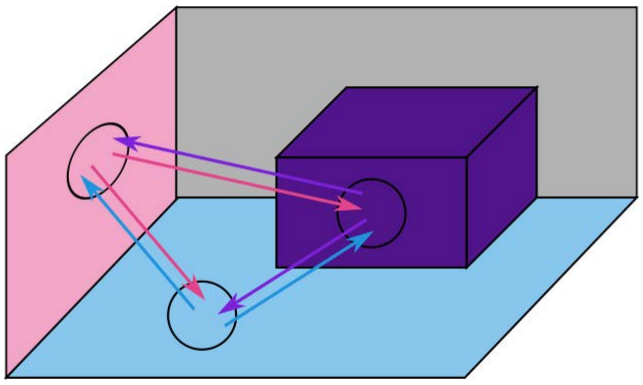
FIGURE 5. ***Point-based global illumination*** **uses a splat or surfel (ellipses with black outlines) representation of the scene to both gather and scatter secondary lighting information from the scene. The gathering of irradiance can be achieved by either microbuffer rasterization or actual ray tracing.**

based on attribute mean values [75]. Faraway surfels were approximated with the cluster mean values, whereas the near surfels were further refined by their exact attribute values. The method was offline, but it was able to reduce frame times by a factor of 2 to 3 to the original PBGI method [69].

A return to explicit ray tracing for PBGI in [76] featured a GPU-based implementation with accurate surfel occlusion evaluations. The authors used a fixed budget of 500 thousand surfels generated in a preprocessing step and inserted into a tree structure in a few seconds. The novelty of the method was a reduction step, which traversed the PBGI tree up and down to find the optimal level in the hierarchy in terms of rendering speed and GI quality. Ideas from photon mapping, PBGI, and radiosity caching were combined in an approximate GI ray tracing pipeline [77]. In addition to a traditional radiance- and irradiance-based PBGI solution (shown in Figure 5), the authors proposed a photon-mapping-like stage, where they ray traced light rays from emissive objects to pre-generated radiosity particle locations weighted by realistically correct BRDF and normal-based coefficients. To accelerate scattering and gathering of light into the radiance particles, the authors generated geometry lists between the most contributing particles in a preprocessing step.

For applications with purely static scenes, precomputed PBGI was a more efficient approach. Volumetric shadows for point cloud represented participating media were presented in [78]. They generated a height field occlusion map in 2D light space stored in a quad-tree for acceleration. At rendering time, the view rays were transformed into light space, and shadows were cast in the participating media where the sampled view ray locations were below the occlusion height field values. Similarly in [79], real-time GI re-lighting in real-world scanned cave scenes was achieved by storing a point cloud and its direct illumination into a sparse voxel octree in an offline preprocessing step. In [80], the result of radiosity gathering and scattering inside a homogeneous volume was stored into a precomputed PBGI octree, which was used at rendering time for accelerated light gathering. The system supported multiple scattering events and light bounces inside the volumes. A 60-fold decrease in computation time and a 6-fold decrease in memory consumption compared to a reference solution was reported. Later, a GPU version of the method was implemented, which achieved real-time frame rates with up to 50 million particles [81]. Furthermore, [82] extended the original offline method in single-scattering cases by transforming the angle-dependent light transport values from RGB-angle space into the frequency domain with a covariance matrix eigenvector representation. They adopted the idea from [83], where surface and participating media events, such as free space transport and reflections, altered the covariance space eigenvectors with an appropriate matrix transformation.

An incremental improvement to the PBGI method ray traversal was proposed in [84]. A BSH tree traversal was implemented on 16-vector wide SIMD operations, which achieved a 2 to 3-fold decrease in rendering time compared to a non-parallel implementation. The authors elaborated their methods further in [85], by introducing a hybrid parallel BVH tree traversal scheme using a packet and single thread SIMD depending on the BSH-tree detail level. They examined microbuffer sizes of 32×32 and 128×128 for radiosity gathering in point samples, and also used an adaptive resolution method to refine glossy reflection areas. Recently in [86], the surfel-based global illumination scheme was further enhanced with an adaptive ray tracing heuristic and dynamic surfel spawning for constant screen space surfel occupation. Local variance and frequency of surfel visibility were used together with global ray count limit to adaptively send more rays in more frequently used and higher-variance surfels. Also, advanced techniques such as importance sampling with ray guiding, local surfel irradiance sharing, and stochastic light cuts and reservoir sampling for multiple-light sampling were used to produce fast converging and robust irradiance results even in varying complex scenes and lighting conditions.

B. INDIRECT LIGHTING WITH POINT-BASED METHODS

Using a point-based method solely for indirect lighting has been explored for both shadow generation with *imperfect shadow maps* (ISM) and reflection generation with *reflective shadow maps* (RSM). These methods use a reduced representation of the scene and splat it into second bounce screen space microbuffers for sufficiently accurate effects, which is depicted in Figure 6.

Real-time indirect lighting with *virtual point lights* (VPL) and ISMs was presented in [87]. As in standard VPL methods, the scene was importance sampled via cube maps from the view of the scene lighting, to generate points gathering virtual light contributions. Holes in the ISMs were filled with a push-pull filtering kernel, and finally the individual VPL contributions were gathered and blurred with G-buffer awareness. The authors later improved their method in terms of quality by substituting the VPLs with a full splat-based

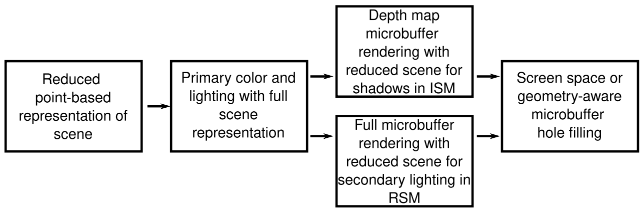
Reduced point-based representation of scene
Primary color and lighting with full scene representation
Depth map microbuffer rendering with reduced scene for shadows in ISM
Full microbuffer rendering with reduced scene for secondary lighting in RSM
Screen space or geometry-aware microbuffer hole filling

FIGURE 6. Both *imperfect shadow map* (ISM) and *reflective shadow map* (RSM) techniques use a reduced point-based representation, such as points or splats, of the scene to render microbuffers at light or pixel locations. ISMs use it for efficient depth-based shadow mapping, whereas RSMs achieve secondary lighting effects for plausible GI by gathering full lighting information into the buffer. Hole filling in the microbuffers is needed due to sparse point coverage.

surface representation stored in a BSH binary tree in [88]. The splat-based scene representation was used for both indirect and direct lighting, and locating a suitable-resolution tree cut for LOD was based on solid angle coverage. The novel contribution was that a preconstructed BSH was updated in real time even in dynamic scenes.

Several publications took the ideas of ISM, RSM, and instant radiosity with VPLs, and applied them in various real-time rendering applications. In [89], the authors combined differential rendering and instant radiosity, in order to produce multi-bounce GI effects between real and virtual objects in real time. Their contribution to the point-based approach of ISMs was the use of surface normal aligned quad splats and parabolic projection to achieve depth-based splat sizes instead of the original inverse depth splat sizes. In a later publication [90], the problem of unsupported double occlusions and incorrect color bleeding between real and virtual objects was resolved with rasterizing two depth maps: one for real objects and one for virtual objects. Furthermore, the authors achieved interactive scene reconstruction by integrating the KinectFusion [10] method to their rendering pipeline.

In [91], the ISM technique was extended to imperfect volumes by rasterizing stochastic point samples based on triangle sizes into a voxel grid and generating cached radiance samples per voxel via ISMs, and finally rendering the voxels with ray marching in screen space. Similarly, [92] presented a voxelized shadow mapping technique by synthesizing VPL shadow maps into a 3D grid, which they also extended to ISMs in a later publication [93]. Instead of using VPLs directly, *virtual area lights* were generated in [94] by clustering VPLs with importance-sampling-based warping and, thus, generating fewer ISMs with similar quality to VPLs in real time. The remaining challenge of distributing VPL samples in RSM and sampling the scene with points in ISM was tackled in [95]. They generated a bi-directional RSM to place VPLs in locations contributing the most to the final lighting of the scene. Furthermore, they improved the point sample distribution in ISM by placing stochastic samples based on triangle solid angle contribution and distance in view space. Finally, [96] used GPU-accelerated tessellation shaders with point generation mode to produce point samples

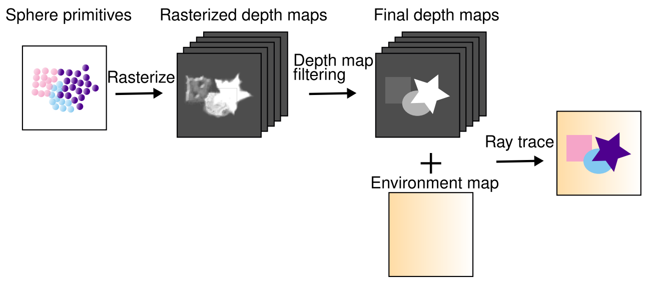

FIGURE 7. The ubiquitous particle-based fluid rendering pipeline for refractive and reflective effects consist of the following steps: one or multiple depth map generation with sphere splatting (rasterization), depth map smoothing or reconstruction with screen space filter kernels, and ray casting or rasterizing depth maps and environment map with single or multiple reflective or refractive fluid surface events. Optionally, fluid thickness-based color attenuation and alpha-transparency can be added.

for fast ISM rasterization based on triangle areas. Furthermore, the adaptive VPL placement was based on Metropolis-Hastings sampling.

C. REAL-TIME PARTICLE- AND FLUID-BASED METHODS

The ubiquitous screen space meshless rendering pipeline for realistic real-time visualization of particle-based simulations, utilized by almost all methods surveyed in this section, was established in [97]. An overview of the method is depicted in Figure 7. They utilized a sphere rasterization method inspired by splatting [2], [53] for depth map generation from point-based models and combined it with curvature-flow-based depth map smoothing as well as fluid thickness extraction to produce composited fluid effects like Fresnel coefficient weighted environment map reflections and refractions in addition to thickness-based transparency and light color attenuation.

In [98], existing SPH fluid simulation and rendering methods were improved by implementing fully GPU-based voxelized SDF rendering for particles. A surface particle subset was extracted from the fluid by comparing the distances of particles to their respective neighborhood mass centers. The surface particles and their proximities were rasterized into a 3D voxelized SDF texture with preset resolution and the resulting SDF was ray cast from the camera and composited into a final transparent image of the fluid and the surrounding scene. In [99], actual refractive effects were produced with a hybrid approach with combined rasterized sphere sprites and bilateral smoothing which generated smoothed fluid front and back face depth maps. Refraction rays were cast from the front face pixel depth locations, and an iterative secant technique refined the ray exit point in the back facing depth map. Later, the authors improved the method in terms of image quality and computational performance by separating splash and surface particles and generating four layers of front and back depth maps in total [100]. Four-layered refractions with realistically blended specular reflections were achieved at a considerably small resolution in real time. Furthermore, a light weight light attenuation model based on the Beer-Lambert equations was added with negligible computation times to increase photorealism with differently absorbing wavelengths.

A similar technique was also applied in a deferred shading pipeline [101] but only with single-layer depth maps. The particles were quantized into a fixed resolution grid in order to fit more points into memory and speed up rendering. The authors generated an approximate refraction effect by distorting the background transparent through the fluid based on fluid surface normals. A simple ray casting technique for adaptive rendering of transparent fluids was also published in [102]. The method aimed at accelerating rendering of fluid simulations by using a perspective grid acceleration structure (consisting of pyramids with cut-off tops) and representative rays for each grid cell for adaptive ray casting. Thus, ray casting was only used for adapting the sample rate but not to produce photorealistic effects other than aggregate transparency.

Further optimizations for GPU utilization were presented in several later publications. A GPU hashing scheme was used for accelerated neighborhood queries and real-time rendering of dynamic particle-based fluid simulations in [103]. The method used familiar screen space splatting with sphere primitives but, instead of depth maps, generated a ray marchable isosurface. Only the front-most surface of the fluid volume was rendered, and an alpha transparency technique extracted an approximate fluid thickness to produce fluid depth and refractions. Multiple GPUs were utilized in a distributed renderer in [104]. Instead of using curvature flow, the authors used a more lightweight bilateral filter for depth map smoothing. A full GPU implementation of a view aligned voxelized sphere particle structure showed that inter-particle refractions and reflections with up to seven fluid layers was possible at interactive frame rates. A view-space aligned voxelized sphere particle method fully implemented on the GPU was published in [52]. With photorealistic inter-particle refractions and reflections between up to seven fluid layers, they achieved interactive frame rates. The voxels were traversed to find up to seven levels of continuous fluid surfaces based on neighboring voxel cells, and finally the extracted and linked surface cells were smoothed with an iterative curvature flow kernel to produce continuous and smooth intersectable surfaces. Finally, a modern implementation with the Nvidia OptiX ray tracing framework [105] and a CUDA-based GPU rendering system was presented in [106].

Overall, particle-based ray tracing methods are a viable substitute for surface point cloud rendering methods. They typically trump more sophisticated local reconstruction and splatting rendering methods in speed, resolution, photorealistic effects, and the number of rendered points. However, due to the depth map rasterization approach, the method is naturally limited only to effects produced by particles inside the view frustum, which makes it similar to screen space ray tracing in its extent. The basic rendering pipeline for particle-based homogeneous fluid rendering popularized in [97] and utilized in practically all the surveyed particle-based methods is depicted in Figure 7. It consists of 3 steps: sphere splatting into one or more screen space depth maps, smoothing and filtering the generated depth maps with sphere flattening and edge detail preserving filtering kernels, and rasterizing or ray tracing the front-most depth map and generating photorealistic effects with layered depth map refractions, environment map reflections and fluid color attenuation.

The focus in the particle-based rendering literature has been to generate real-time rendering results with photorealistic fluid effects from SPH-based simulation, implying rendering support for dynamic scenes and, thus, real-time acceleration construction for ray tracing or direct depth map rasterization methods. The SPH simulations usually yield per-particle densities, masses, and motion vectors, which are available for rendering time effects such as foam generation and velocity deformation on particles. However, these attributes can be omitted in the surveyed methods and they are still applicable to raw point/particle clouds with only coordinate information and other optional attributes for traditional rendering pipelines (for example color and material properties).

All surveyed particle-based methods exhibit real-time frame rates with single-layer depth map [97], [103], [104], [106] as well as double-layer front- and back-face depth maps [99]. Additionally, foam and fluid separated particle depth maps [101] including up to four-layer depth maps [100] and up to seven-layer depth maps with particle inter-reflections [52] are also supported in real-time or interactive frame rates. All methods generate translucency, thickness-based light attenuation, Phong shading, environment map specular reflections and approximated or exact refractions, and Fresnel equation composited final color.

D. SUMMARY AND DISCUSSION

For point-based global illumination, [81] achieved real-time frame rates with up to 50 million particles: 30 FPS when moving the viewpoint, or 25 FPS when editing the light position or the material parameters. As another highlight, [86] demonstrated a promising future direction for producing high-quality results, by combining hardware-accelerated ray tracing with various advanced techniques for the purpose of real-time gaming. Their method converged in about 2–7 seconds depending on the amount of lights in the scene, at a 4K output resolution on a PlayStation 5.

As for point-based methods targeting solely indirect illumination, in [89], multi-bounce GI effects for mixed reality were computed at 22–24 FPS on an Nvidia GeForce GTX 285 GPU. They used 256 VPLs, with the scene being represented by 1024 points per VPL and the output resolution being 1024×768 pixels. In [96], GPU tessellation was used for ISM splatting. With 1024 VPLs in total, and 128 VPLs evaluated at each shaded pixel, they achieved 30–54 FPS at a 512×512 output resolution on an Nvidia GeForce GTX 470 GPU.

Virtual area lights were used in [94], yielding indirect illumination results at 74 FPS with fewer artifacts than with a VPL-based approach, for a 1280×720 output resolution on an AMD Radeon HD 6850 GPU. However, reducing the number of indirect light sources degrades the quality of the indirect shadow information; their solution for improving the shadow sampling takes an additional 26 ms.

Particle-based volume methods work on larger frame resolutions compared to most ray tracing methods. Most of them achieve relatively high frame rates even with a resolution of 1280×720. The fastest method achieved a frame rate of 97 FPS with a resolution of 1024×1024 and $16 \cdot 10^3$ points [98]. However, the number of used points is comparably low, given the fact that the particle-based methods have a lower number of actual surface points. Thus, a more comparable method with a larger point cloud size is the method in [100], which achieved 48 FPS with a resolution of 1024×768 and $4 \cdot 10^6$ points.

Out of the particle-based fluid volume rendering methods, the one in [52] exhibited the most photorealistic effects at low interactive frame rates with up to $7 \cdot 10^6$ volume particles. Supported effects included 2 spp reflections for particle inter-reflection-based GI, multi-layered (up to 7) refraction, and extendability to other ray tracing effects such as shadows and path tracing. In terms of rendering performance, the method in [101] achieves real-time performance on particle-based fluid volumes with up to $5 \cdot 10^8$ particle points. They used a binning strategy with up to $2 \cdot 10^3$ brick bins for accelerated rendering with single-layer approximate refractions and specular reflections. As discussed earlier, the number of points/particles is not directly comparable to surface point cloud rendering methods. We identify that the relationship between the particle-based volume rendering and surface point cloud rendering in this regard, and in terms of method applicability and scalability in each domain, would be a fruitful subject of future research.

VI. POINT-BASED NEURAL RENDERING

We briefly review the latest state-of-the-art neural network solutions for point cloud and point-based model rendering. Only a handful of current solutions can produce plausibly photorealistic point cloud renderings in real time, and all of them require scenes to have static geometry. Furthermore, many of the neural networks have to be re-trained on target specific point cloud inputs and they need to have ground truth images for training, which are not readily available in applications such as holoportation or teleconferencing solutions. However, as the area of point-based neural network rendering is emerging, more general and computationally effective solutions independent of domain-specific training might be possible in the near future.

A neural network solution for upsampling and filling a splatted point cloud was published in [107]. Instead of using a rendering network to learn a mapping from a point cloud to final image, it used a *GAN*-based network to learn a post-processing step from an incomplete and low-frequency splatted rendering to a high quality final frame. The splatting method worked in real time by using a k-d tree to fit a suitable point-wise normal and splat radius based on k-NN in 12 sectors around the splat with similar surface normals. However, the actual upsampling and filling GAN network ran at interactive frame rates only for low-resolution images. Thus, the authors used the network to render or pre-render high quality key locations and viewpoints in a scene, and used the lower-quality splatted results for interactive 3D scene navigation.

One of the first neural network–based methods attempting full scene capture (capturing or deducing scene geometry, material properties and lighting information, from multiple unstructured image sets in a single system) was proposed in [108]. They used a point cloud, a depth map, and a segmentation image constructed from multiple images of a single landmark scene as an input to a two-stage network. The first stage learned a descriptor vector set from the input point cloud, depth map, and segmentation to further encode various aspects of the scene. The descriptor set together with the inputs were further fed to the second stage (rendering network), which mapped the inputs to a final rendered image from a novel viewpoint with new lighting conditions. Furthermore, the semantic segmentation was used to mark non-relevant foreground and non-stationary objects in the view of the rendered landmark so that the network could both learn to remove and re-instantiate these elements in novel views. The inference time during rendering and generation was reported to be 330 ms at a resolution of 512×512 on an Nvidia Titan V GPU. Training time was not reported.

In [109], a two-part neural network, based on convolutional U-Net architecture, was used to render a point cloud input in novel viewpoints. The first part consisted of an 8 dimensional point descriptor vector set and camera parameters, which was rasterized into multiple-resolutions and given to the second, neural rendering network part to learn the mapping of the descriptors to an RGB image. Both the point-based neural rendering network and the descriptor vector set were trained in a two-step fashion. First, the rendering network and the descriptor set were pre-trained on a general input sequence, after which the descriptor set was reset and retrained with a local input sequence (closely resembling the validation set) with the pre-trained rendering network. The authors reported better image quality compared to similar neural rendering methods based on mesh primitives. However, their method was also limited to scenes with static models and lighting, making it non-transferable to scenarios like dynamic point cloud streaming and re-lighting.

To alleviate the problems of rasterizing the descriptor vectors directly onto the image plane, a voxelization-based method was suggested in [110]. Instead of using a multiresolution approach on the same image plane with nearest depth pixel selection, the authors suggested rasterizing the descriptor vectors onto different depth planes on the view frustum called frustum voxels. Additionally, they blended all the fragments on single pixels based on a distance metric to the pixel center on both the image plane and perpendicular depth weighting the closer fragments more. This improved quality on sparse point cloud inputs and decreased background bleeding due to holes in the rasterized point clouds. Nevertheless, the rendering times were not disclosed. Furthermore, the method was similarly designed for static models and lighting.

The method in [111] used a radiance field (i.e., light field) representation of the scene to render an image from novel viewpoints via a volume rendering technique. The radiance field, produced from images of multiple viewpoints, was given as an input to two parallel fine and coarse fully connected neural networks, which gave a color and density output to a volume renderer for final composition. Entries from the radiance field were encoded in a higher-dimensional space in order to preserve fine geometrical details. The final rendering time was measured in tens of seconds.

In [112], the authors introduced a temporally consistent neural renderer with a differentiable point-based pipeline. Multi-view stereo [113] was used for the initial reconstruction of the 3D geometry, followed by further per-view optimization based on bi-directional EWA splatting, probabilistic per-view depth testing, and effective camera selection. Their main application was also novel-view synthesis, although the method is suitable for multi-view harmonization and image stylization as well. Their free-viewpoint rendering ran at 4.5 FPS on an Nvidia RTX 6000 GPU, once the network had been optimized for the scene; training took several hours.

Neural rendering with a sphere-based geometry representation was proposed in the *Pulsar* method [114]. A fully differentiable rendering pipeline simultaneously forward renders and backward propagates scene representation refinement, projection operation, and neural shading. The sphere representation included radii, transparencies, and feature vectors describing local geometry and lighting, which were all learned during training time and applied at rendering time.

A three-stage differentiable point renderer, called *ADOP*, was able to beat OpenGL point primitive rendering in both image quality and rendering speed [115]. The system consisted of a differentiable rasterized (inspired by the efficient compute shader point rendering in [116]) and tonemapper with a neural renderer in between, which took the output of the rasterizer as input and produced an HDR image for the tonemapper. The system refined and updated the input point cloud and camera model as well as both the neural renderer weights and tonemapper HDR parameters. In order to increase both speed and image quality compared to native OpenGL point primitive rendering, a stochastic point discard heuristic with discarded pixel geometry utilization for spatial gradients was employed.

A. SUMMARY AND DISCUSSION

Various methods for point-based neural network rendering have been proposed. Both image-based post processing of splatted point clouds [107] and model-to-rendering direct mappings [108]–[112] have emerged. Specific solutions tried to tackle the problem of background bleeding in sparse point clouds with a multi-resolution image-space solution [109] and an object space view frustum voxelization [110]. In [111], a custom encoding of spatial and photometric data stored information in a radiance field (light field), and the authors in [108] were able to relight scenes based on non-organized images of landmark locations and remove transient objects from the foreground with segmentation. However, only a few of these methods were able to perform in real time and all of them rely on external algorithms to produce inputs like point clouds, depth maps, segmentation, or splatting, which were not included in final rendering time.

Nevertheless, we highlight two prominent novel viewpoint point cloud renderers achieving 23.7 ms with 10^6 spheres at a 1000×1000 resolution [114] and 5.7 ms rendering time with 10^7 points in full HD [115], which are highly usable neural point cloud renderers in scenes with static lighting and geometry. The former of the methods can learn the sphere representation from multi-view images, whereas the latter method only needs coarsely triangulated textured point clouds from RGB images with rough camera parameter estimates as input. A summary of the real-time performance of the methods is in Table 5.

VII. DISCUSSION

In this section, we discuss the state of the art in real-time photorealistic point cloud rendering based on the surveyed methods, focusing on the capabilities of producing photorealistic rendering effects in real time in Section VII-A. The posed research question is answered in Section VII-B. Additionally, we discuss the current capabilities of dynamic acceleration structure construction and updating for point clouds. This section is concluded with a discussion on the possibilities of future research in Section VII-C.

A. STATE OF THE ART IN PHOTOREALISTIC POINT CLOUD RENDERING

As shown in Table 5, over 20 point cloud rendering methods achieve interactive or real-time frame rates. Particle-based volume visualization methods have seen the most active research in recent years, whereas publications on surface point cloud ray tracing have been prominent in the late 2000s. Surface point cloud rasterization methods exhibiting photorealistic effects have not been researched as actively, but the computational efficiency of these methods compared to ray tracing makes them a viable option in photorealistic visualization.

Particle-based volume methods consider point clouds that represent an object within a volume with possible interactions, as discussed in Section V. This sets it apart from surface point clouds in two ways.

On the one hand, the number of points in a particle system depicting the surface of the volume is much smaller than the total number of points. As photorealistic effects, such as reflections and refractions, are mostly concerned with the interaction of light at surface boundaries, the visualization effort is largely concentrated on the surface points. Thus, the points within a volume can be ignored if the surface is opaque or the volume is homogeneous in color and material. The latter statement is true for all of the surveyed particle-based methods as they work with homogeneous liquids. It should be noted, however, that more accurate photorealistic refractions would benefit from taking into account the effect of varying volume densities if such information was available.

Particle-based systems almost always have an underlying simulation governing the interaction between the particles and their movement based on the particle characteristics. Usually, these simulations produce extra attributes for the particles, which eases the visualization part. These attributes include for example the particle radius, which is used in most of the hybrid approach methods where particles are first projected into screen space as spheres with the accompanying radius, and then ray traced in screen space.

The screen space nature of the particle-based volume surveyed methods means that all the photorealistic effects are inherently limited to the parts of the scene currently in the view frustum and the pre-rendered environment map. This means that the surface point cloud ray tracing methods provide more realism at least in theory because, for example, their reflections and refractions can include interactions outside the view space. The screen space representation is both a benefit in terms of rendering speed and a hindrance in less realistic effects in real-time particle-based visualization.

1) SUPPORTED PHOTOREALISM

The supported photorealistic effects of real-time point cloud rendering methods are summarized in Table 1. It should be noted that all the methods concentrate on some photorealistic aspects, but none of them report real-time capabilities with a path tracing approach where actual GI effects would be present. As such, it is still an open question whether point clouds can be path traced in real time or whether GI effects, in general, can be produced for point clouds in real time.

In general, the effects produced by the different methods can be categorized in the following way. All surface point cloud ray tracing methods, except [47], use a Whitted style ray tracing approach, which means that they support hard shadows and sharp specular reflections. Furthermore, single-layer refractions are supported in [46], [48]–[50], whereas [47] provides soft shadows from multiple light sources and sharp specular reflections with multiple bounces.

Real-time surface point cloud rasterization methods provide only hard shadows [55], [62], [65] or soft shadows [61] with Phong shading as their photorealistic effects. This means that even though rasterization methods are computationally more efficient compared to ray tracing, the supported photorealistic effects for point cloud rendering are very limited.

All of the particle-based volume methods are concerned with refractions through liquids and reflections on the surface of liquids. The reflections exhibited in all methods, except [102], are sharp specular reflections from the accompanying pre-rendered environment map or between objects in the scene. The dividing factor between these

TABLE 1. Rendering feature table for all real-time and interactive methods that show one or more photorealistic rendering effects.

Method	Real time ($\geq$ 10 FPS)	Interactive ($\geq$ 1 FPS)	Real time / Interactive # points	Real/Inter. $\geq 1000^2$ resolution	Unorganized input points	Dynamic points	Surface points	Offline preproc.	Acceleration	Input normals	Input radius	Ray tracing	Shadows	Reflections	Refractions	spp
Surface point cloud ray tracing methods																
[5] 2005	✓		$1 \cdot 10^6$				✓	✓	✓	✓	✓	✓	×	×		1
[45] 2006		✓	$1 \cdot 10^5$				✓	✓	✓	✓	✓	✓	×	×		1
[46] 2007		✓	$5 \cdot 10^4$				✓	✓	✓			✓	×	×	✓	1
[49] 2010	✓		$1 \cdot 10^7$				✓	✓	✓	✓	✓	✓	×	×	✓	1
[47] 2010		✓	$4 \cdot 10^6$	✓			✓	✓	✓	✓	✓	✓	✓✓✓	×××		4
[48] 2010	✓		$1 \cdot 10^6$				✓	✓	✓	✓	✓	✓	×	×	✓	1
[50] 2011	✓		$1 \cdot 10^7$				✓	✓	✓	✓	✓	✓	×	×	✓	1
Surface point cloud rasterization methods																
[55] 2005	✓		$7 \cdot 10^5$		✓	✓	✓			✓	✓		×			
[56] 2007	✓	✓	$1 \cdot 10^6 / 2 \cdot 10^6$		✓	✓	✓			✓	✓			×	×	
[61] 2010	✓		$4 \cdot 10^5$	✓	✓	✓	✓			✓			✓			
[62] 2012	✓		$2 \cdot 10^6$		✓	✓	✓			✓			×			
[65] 2018	✓		$1 \cdot 10^7$ / $2 - 8 \cdot 10^6$ visible	✓	✓		✓						×			
Particle-based volume methods																
[97] 2009	✓		$6 \cdot 10^4$	✓	✓	✓								×	×	1
[98] 2010	✓		$8 \cdot 10^5$	✓	✓	✓			✓			××			××	1
[99] 2014	✓		$1 \cdot 10^5$	✓	✓	✓					✓	××		×	✓✓	1
[101] 2014	✓		$5 \cdot 10^8$ / $2 \cdot 10^3$ brick bins	✓	✓	✓			✓			××		×	×	1
[100] 2016	✓		$1 \cdot 10^6$	✓	✓	✓					✓	✓		×	✓✓✓	1
[102] 2016	✓		$1 \cdot 10^6$	✓	✓	✓			✓			××				1
[103] 2017	✓		$2 \cdot 10^6$	✓	✓	✓			✓			××		×	×	1
[104] 2017	✓		$1 \cdot 10^6$	✓	✓	✓								×	×	1
[106] 2018	✓ (static)	✓ (dynamic)	$2 \cdot 10^6$	✓	✓	✓			✓			✓		×	✓	1
[52] 2018		✓	$7 \cdot 10^6$	✓	✓	✓			✓			✓		×	✓✓✓	2

TABLE 2. Marker legend for Table 1.

Marker	Ray tracing	Shadows	Reflections	Refractions
×	Primary rays	Single-light hard shadows	Single-bounce specular	Single-layer env. map sampling
××	Secondary rays			Two-layer env. map sampling
×××		Multiple-light hard shadows	Multi-bounce specular	Multi-layer env. map sampling
✓	All visibility	Single-light soft shadows	Single-bounce glossy	Single-layer ray traced
✓✓				Two-layer ray traced
✓✓✓		Multiple-light soft shadows		Multi-layer ray traced

methods, however, is the approach to refractions. [97], [98], [101], [103], [104] produce refractions only by sampling the environment map, whereas [52], [99], [100], [106] also refract light from other objects in the scene including intra-refractions. Furthermore, the number of interaction layers used for refraction varies between the methods with [98], [99] having dual-layer refractions and [52], [100] having up to four-layer refractions.

Photorealism in point cloud rendering is also affected by the detail level of the point cloud model. In the point cloud capturing literature, a definite point resolution has not been established, and it varies depending on the application ranging from smaller density of points for urban and outdoor environments to high-resolution details for indoor and human capturing. For the use case of most of the methods surveyed in this publication, namely human-sized objects, some estimates can be made on typical point cloud resolutions. For example, the KinectFusion and its successors typically use a grid structure of up to 512^3 entries or over 10^8 potential data points in an indoor environment. However, most of these grid entries are empty, making the evaluation of the total number of relevant points difficult. Models in The Stanford

3D Scanning Repository [15] give a rough estimate of the model detail needed to represent scanned objects. The typical size is around 10^6, whereas more detailed models, such as the human-like Lucy model, can have up to 10^7 points. On the other hand, urban and man-made structures typically exhibit less geometric detail, and include areas like floors and ceilings, which can be expressed with less data. Hybrid rendering primitive approaches, like using meshes for linear static areas of the scene and point clouds for more detailed areas, might yield better results in these cases.

B. ANSWERING THE RESEARCH QUESTION

What is the state-of-the-art technique for photorealistic end-to-end direct point cloud rendering for a high-quality human-sized model (10^7 points), in 75 FPS, and a resolution of 1080p on consumer hardware?

As discussed in Section III-D, none of the surveyed direct point cloud rendering methods are able to produce all of the photorealistic effects with their respective hardware with the established requirements. Therefore, no direct point cloud rendering methods have shown photorealistic rendering at 1080p and 75 FPS for at least 10^7 points. The method closest to these demands is presented in [47], capable of rendering soft shadows from multiple light sources and sharp reflections with multiple bounces for 10^6 points and a resolution of 512×512 at 55 FPS. Moreover, as discussed in Appendix B, projecting the measurement provided on an Nvidia RTX 275 to a modern RTX 2080 Ti could mean a ray tracing performance of 75 FPS to 130 FPS with requirements posed in the research question. Adding the needed acceleration structure, namely the octree structure, presents a negligible computational overhead to the end-to-end pipeline if 10^6 points are used or even up to 10^7 on modern GPUs. However, as splats are assumed for the input of the ray tracing method, it is difficult to estimate how much preprocessing effort is needed to generate the splats. The methods in [52] and [89] achieve real-time performance but lack the photorealistic effects required.

Point-based neural rendering has showed promising results for point cloud rendering from multi-view image inputs. The *Pulsar* system [114] and the *ADOP* method [115] achieve 42 FPS with 10^6 spheres and a 1000×1000 resolution and 175 FPS with 10^7 points and 1080p resolution, respectively. However, the systems are primarily used for novel view rendering and are thus limited to static geometry and lighting.

Furthermore, we highlight the method in [117] as the most efficient BVH generation method. As discussed in Appendix A, the authors provided a state-of-the-art acceleration structure construction method for both octrees and BVHs with up to $1.77 \cdot 10^6$ points in real time. Specifically, the octree construction took 0.88 ms, meaning an almost negligible performance loss to an end-to-end pipeline. Extrapolating this result to accommodate the desired 10^7 point count already yields a significant cost of at least 8 ms. As discussed in Appendix B, however, an optimistic lower bound estimate for the performance of the construction on an Nvidia RTX 2080 Ti was established at 0.24 ms for BVHs and 0.62 ms for octrees and 10^7 points. However, an acceleration structure method for volumetric meshes showed a more conservative construction time of 50 ms and a reference OptiX acceleration structure construction time of 200 ms. This means that the method may meet the posed requirements on modern hardware, but an actual implementation is needed to verify the capabilities.

C. FUTURE RESEARCH

The latest research on real-time point cloud ray tracing methods was published almost a decade ago. There is research to be done especially on extending ray tracing with path tracing for point clouds. Based on this survey, combining both rasterization and ray tracing techniques into a hybrid approach similar to the particle-based volume rendering could be highly beneficial in terms of computational efficiency. As particle-based approaches have worked on volumes, novel approaches applying the same techniques on surface point clouds would be interesting. Nevertheless, as discussed before, these methods would inherently work in screen space and suffer from similar problems as other methods such as *screen space ray tracing*.

Further research could be done to establish the sufficient detail needed in captured scenes and models. Researching the trade-offs between model detail-level and network bandwidth as well as the impact on photorealism and rendering speed may provide an interesting aspect. Establishing the level of photorealism with regards to point cloud size that can be captured, transferred, and rendered in real time with current hardware is a possibly fruitful avenue for research. Also, modern point-based neural rendering methods have built-in capabilities for point cloud and scene reconstruction due to learnable geometry features. Utilizing the scene understanding aspect of deep learning methods may provide an insight to sufficient level-of-detail of point clouds and geometry for photorealistic rendering in varying scene and lighting settings.

With the recent arrival of GPUs with dedicated ray tracing hardware, such as the Nvidia RTX series with Turing [118] and Ampere [119] architectures, harnessing the support for custom intersection functions for point cloud intersection could yield interesting results. Combining this with acceleration structures specifically designed for point clouds and comparing them to existing supported acceleration structures could also be useful. Based on our evaluation in Appendix B, meeting the requirements posed in the research question is feasible. The publication dates of the reviewed methods span two decades, and consequently, many generations of hardware architectures have since been released. Implementing and testing the highlighted methods for point cloud rendering on identical modern hardware would provide a fair comparison between them and definitively answer our posed research question.

VIII. CONCLUSION

In this survey, we reviewed real-time photorealistic rendering methods for point cloud visualization. Specifically, ray tracing methods for point cloud rendering were exhaustively surveyed and real-time rasterization and hybrid methods for realistic rendering were reviewed for comparison.

We found that direct photorealistic rendering is possible at 130 FPS and HD resolution [47] when estimating the performance on modern hardware. For the desired point clouds in the order of 10^7 points of size, an acceleration structure could be constructed in negligible time on modern GPUs [117]. Furthermore, point-based neural rendering can achieve novel viewpoint rendering for point clouds in static scenes with 10^7 points and 1080p resolution at 175 FPS.

Based on our findings, we highlighted that photorealistic rendering of live captured point cloud content has open research problems, such as utilizing path tracing for point clouds directly. Extending the state-of-the-art methods with path tracing to verify the performance of the methods on the current dedicated ray tracing hardware is left as future work.

To conclude, the performance numbers achieved with the state-of-the-art methods do not yet satisfy the requirement of photorealistic point cloud rendering when considering a sufficient end-to-end latency of at least 75 FPS, a minimum high-quality screen resolution of 1080p, and an adequately detailed point cloud in the order of 10^7 points. Direct point cloud ray tracing is an order of magnitude behind in terms of point cloud size and resolution. However, based on our estimations, the posed requirements may be achievable on modern desktop-scale GPUs with optimistic assumptions, let alone on mobile scale GPUs where the AR use case is more typical.

APPENDIX A
ACCELERATION DATA STRUCTURES FOR POINT CLOUD DATA

In this section, we briefly review acceleration structures designed and applicable to point cloud rendering, specifically, acceleration for point cloud ray tracing. This serves the purpose of estimating the performance of dynamic acceleration construction in an end-to-end system with temporally incoherent point cloud input. For a more exhaustive review on the acceleration of triangle-based animated ray tracing, we refer to [120]. Based on our findings, the most prominent methods for accelerating point cloud rendering specifically are *k-d trees* and *octrees*. Both k-d trees and octrees are depicted in Figure 8. Apart from these, we review also general spatial data structures and K-nearest neighbors.

One of the first parallelized octree data structure construction methods taking advantage of the GPU was presented in [58], which achieved approximately a ten-fold speed-up compared to the seminal QSplat system in [2]. The rendering, LOD selection and fine-grained culling of the hierarchy were done on the GPU, freeing the CPU for other tasks, such as coarse initial culling. The method provided efficient LOD queries and selection with a sorted node structure

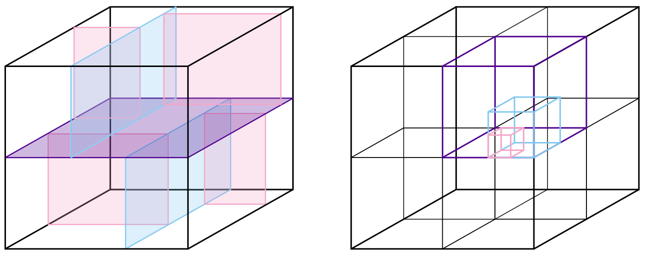
FIGURE 8. The most prominent acceleration structures designed or applicable for point cloud rendering, namely the k-d tree (left) and the octree (right), are illustrated.

based on rendering distance and a hole-free result with slight overlapping between hereditary nodes. Similarly, a layered point cloud structure with a hierarchical LOD system for the GPU was introduced in [121]. The multiresolution structure precomputed and binned point cloud points into nodes in world space hierarchically, and instead of rendering distance, used coarse to fine sorting based on sampling densities. The system also supported culling techniques and compression for further computational efficiency. In [122], a k-d tree acceleration structure was used for k-means clustering and quantizing, which could also be applied to LOD support. However, instead of point clouds their method was utilized for *spherical harmonics* (SH) coefficients storage and they utilized clustered averaged SH coefficients for the LOD support in a *point-based global illumination* (PBGI) rendering system with a focus on quality, not real-time speed.

K-d trees were also used directly for ray tracing acceleration. A real-time k-d tree construction method on the GPU was presented in [123] that used triangles or points with color without any extra attributes as input. The method used a greedy *surface area heuristic* (SAH) scheme, which outperformed a thorough SAH-based CPU implementation in smaller scenes. Furthermore, ray tracing and traversal speeds indicated better quality in practice for all types of scenes compared to the exhaustive CPU implementation. In [124], k-d tree structures were utilized to support for volume ray tracing. They extended the OpenVDB software [124] with a novel contribution of offering fast dynamic updating and ray traversal operations for a GPU implementation of an accelerated volume ray tracing framework.

An extensive system that supported several different acceleration structures, such as Octrees and BVHs, for ray tracing were built from a common binary radix tree representation in [117]. The tree construction started with a Morton code generation and sorting step, which placed 3D pointwise data on a space ordered curve in lexicographical Morton code order. Finally, a data structure for general k-NN querying on point-based data implemented a *random ball cover* (RBC) algorithm using a subset of representative points to bin a set of 3D points [125]. Contrary to the exhaustive search of iterating and sorting points per representative point bin, their GPU implementation iterated the 3D points and placed them to their respective representative bins, which sacrificed

TABLE 3. Comparison of processing unit counts of the different GPUs used in the surveyed methods, with details retrieved from [128].

GPU	Used in	Processing units	Increase to RTX 2080 Ti
Nvidia RTX 2080 Ti, 2018	N/A	4352	1×
ATI Radeon 9700, 2002	[58]	N/A (fixed pipeline)	N/A
Nvidia 6800 GT/Ultra, 2004	[45], [55]	N/A (fixed pipeline)	N/A
Nvidia 7900 GT, 2006	[46]	N/A (fixed pipeline)	N/A
GeForce 8800 GT, 2007	[97]	112	39×
GeForce 8800 GTS/ULTRA, 2007	[97], [123]	128	34×
Nvidia Quadro FX 3800, 2009	[62]	192	23×
Nvidia GTX 2XX, 2009	[47]–[50], [60], [61], [98]	240	18×
Nvidia GTX 460, 2010	[125]	336	13×
Nvidia GTX 480, 2010	[117]	480	9×
Nvidia GTX 960, 2015	[106]	1024	4.3×
Nvidia GTX 970 2014	[52], [103]	1664	2.6×
Nvidia GTX Titan, 2013	[52], [99]–[102]	2688	1.6×
Nvidia Titan X, 2015	[65]	3584	1.2×
Nvidia Quadro M6000, 2015	[129]	3072	1.4×

completeness for a efficient parallel algorithm. Even though the authors did not mention ray tracing as an application, k-NN querying is essentially a part of all the covered intersection testing algorithms for point cloud data.

A. SUMMARY AND DISCUSSION

K-d tree solutions in [123] for k-NN search and PCA-based surface normals as well as in [122] for the storage of SH coefficients of a surfel-based scene have shown their effectiveness on point data. The former method achieved a performance of 21 ms for static and 310 ms for dynamic deforming scenes with 171 thousand points, yielding a frame rate of 48 FPS and 3.2 FPS, respectively. The latter method was quality-oriented and not specifically suitable for real-time acceleration. Octrees were utilized for LOD-based rasterization acceleration in [58], but only the final rendering speed of 50 to 80 million splats per second was disclosed without mentioning the construction or update time of the data structure.

The most prominent method for acceleration was presented in [117]. Even though the method was primarily oriented for triangle-based scenes, the fact of using triangle centroids to organize the data structures made the method inherently point-based. For scenes with up to 1.77 million points, 0.34 ms for BVH construction and 0.88 ms for octree construction were reported. By extrapolating with the asymptotic behavior of acceleration structure construction $n \log(n)$, the results for 10^7 points yields $>$ 3 ms for BVH construction and $>$ 8 ms for octree construction. Furthermore, with a modern Nvidia RTX 2080 Ti GPU, the construction time decreases to a negligible 0.24 ms and 0.62 ms, respectively, in the optimistic case of the construction algorithm fully parallelizing.

The covered methods are mostly software-based methods for constructing the acceleration structures. Dedicated hardware solutions for further accelerating the construction have been proposed [126], [127]. Utilizing hardware-accelerated BVH and other data structure construction and updating for point cloud ray and path tracing could yield even more performance and support for larger point clouds.

APPENDIX B COMPUTATIONAL PERFORMANCE ANALYSIS

To equally juxtapose the methods implemented on various GPUs, we compare the increase in processing units/cores from the original GPU in the publications to a state-of-the-art desktop GPU, namely an Nvidia RTX 2080 Ti. This allows

TABLE 4. Point-based acceleration structures achieving real-time construction time. We crudely estimated the frame rate projected onto a modern consumer GPU (Nvidia RTX 2080 Ti, listed in Table 3) with the specifications posed in the research question (RQ), meaning 10^7 points. Note that the acceleration construction times scale at $n\log(n)$ w.r.t. the number of points n.

Method	Number of elements	Acceleration type	Construction time (ms)	Hardware	Construction time (ms) with RQ requirements
[58] 2003	(splats) $50 \cdot 10^6$	K-d tree	1000	ATI Radeon 9700 GPU	N/A (fixed pipeline)
[123] 2008	$27 \cdot 10^3$	K-d tree	17	Nvidia GeForce 8800 ULTRA GPU	293
	$252 \cdot 10^3$		93		139
[125] 2011	512	k-NN	0.33	Nvidia GeForce GTX 460 GPU	1261
	8192		1.59		262
[117] 2012	$174 \cdot 10^3$	Octree	0.46	Nvidia GeForce GTX 480 GPU	3.9
		BVH	0.05		0.43
	$1.77 \cdot 10^6$	Octree	0.88		0.62
		BVH	0.34		0.24
[129] 2016	(voxels) $8.6 \cdot 10^9$	K-d tree	560	Nvidia Quadro M6000 GPU	0.32

us to get a crude estimate of the capabilities of the different methods on modern hardware assuming that the algorithms would fully parallelize and utilize all available cores, which is an optimistic assumption. Also, for older hardware with more fixed graphics pipelines the comparison is hard as they do not have processing units comparable to modern GPUs. However, this method was chosen to give the reader a general idea of how the methods would compare on equal terms. Furthermore, the task of actually implementing all of the methods on modern hardware is very time-consuming and out of scope in this survey.

The different GPU platforms and their respective processing unit or core counts are summarized in Table 3. For the target platform of RTX 2080 Ti, we omitted the calculation of RT cores into the total count of cores as it is not clear how they could be compared to other cores on older hardware. As an example calculation, we evaluate the performance of the direct point cloud ray tracing method in [47]. This evaluation assumes that the presented method is fully parallelizable and that memory transfer and storage do not present a bottleneck to the pipeline, which is of course unrealistically optimistic. Consequently, this approximation should be treated as an estimate of the optimistic potential of the method. We approximate the number of processing elements in the Nvidia GTX 275 used in [47] with the numbers for Nvidia GTX 280 in [130], which has a total of 240 streaming processors, i.e., processing units. Nvidia RTX 2080 Ti has 4352 CUDA "cores", comparable to processing units, and additionally 68 RT cores which could further accelerate ray tracing. If we simply use the CUDA core number for the estimate, the increase is $18\times$. Assuming a linear increase in the workload when increasing the resolution from 512×512 to 1080p, almost an 8-fold increase in computation is gained. This would yield a total increase of $2.3\times$ in compute power and increase the reported 55 FPS in [47] to 130 FPS on a an RTX 2080 Ti. The frame rate projections of the rest of the methods in Table 5 are done in a similar way.

The calculations made for the projected frame rates of the methods in Table 5 do not consider the number of rendering primitives used in ray tracing. Acceleration construction and updating is the most affected by the number of rendering primitives. Similar to before, we evaluate an upper bound estimate for the acceleration structure method in [117] highlighted in Appendix A-A. The original method was implemented on the Nvidia GTX 480, which means $9\times$ more cores on the Nvidia RTX 2080 Ti excluding the RT cores (see Table 3). Assuming an $\mathcal{O}(n\log(n))$ increase in computation time with respect to the n number of primitives when constructing acceleration structures, the 0.34 ms BVH and 0.88 ms octree construction time for $1.77 \cdot 10^6$ points corresponds to 3.9 ms and 10 ms for $1.77 \cdot 10^7$ points, respectively. As also shown in Table 4 before, in the utopic scenario of a fully parallelizable construction algorithm without considering the memory bottleneck, the RTX 2080 Ti could perform the construction in 0.24 ms for the BVH and in 0.62 ms for the octree construction. However, [131] reported construction times for their acceleration structure for volumetric tetrahedral meshes together with reference times measured for the default OptiX acceleration structure on an Nvidia RTX 2080 Ti. With up to $1.2 \cdot 10^7$ tetrahedrals, their acceleration structure was constructed in slightly over 50 ms and the OptiX reference was constructed in less than 200 ms, or 20 FPS and 5 FPS, respectively. Based on their results, the required 75 FPS can be achieved with approximately $0.7 \cdot 10^6$ dynamically changing primitives if the acceleration structure needs to be constructed from scratch for each frame.

TABLE 5. Point-based ray tracing, rasterization, hybrid methods, and point-based neural rendering achieving real-time frame rates and some photorealistic rendering effects in scenes with various amounts of rendering primitives and at different resolutions. We crudely estimated the frame rate, where applicable, projected onto modern consumer GPUs (Nvidia RTX 2080 Ti) with the specifications posed in the research question (RQ), meaning a resolution of 1080p and 10^7 points.

Real-time method	Number of elements	Frame resolution (pix)	Frame rate (FPS)	Rendering hardware	Frame rate (FPS) with RQ requirements
Surface point cloud ray tracing methods	**Splats**		**Preprocessing time omitted**		
[5] 2005	$2 \cdot 10^3$	640 × 480	30	2.4 GHz AMD	N/A (CPU)
	$125 \cdot 10^3$		22	dual-Opteron CPU	
	$24 \cdot 10^6$		2		
[45] 2006	$36 \cdot 10^3$	512 × 512	5.30	Nvidia	N/A
	$48 \cdot 10^3$		4.00	6800 GT GPU	(fixed pipeline)
	$109 \cdot 10^6$		1.17		
[46] 2007	$36 \cdot 10^3$	640 × 480	1.33	Nvidia	N/A
	$48 \cdot 10^3$		2.16	7900 GT GPU	(fixed pipeline)
[49] 2010	$1.3 \cdot 10^6$	512 × 512	17	Nvidia 896 MB	39
	$3.6 \cdot 10^6$		11	GTX 275 GPU	25
	$14.7 \cdot 10^6$		8		18
[47] 2010	$1 \cdot 10^6$	512 × 512	55	Nvidia	130
	$3.8 \cdot 10^6$		4	GTX 275 GPU	9.1
[48] 2010 & [50] 2011	$1 \cdot 10^6$	512 × 512	13.8	Nvidia 896 MB	31
	$1.3 \cdot 10^6$		12.5	GTX 275 GPU	28
[51] 2011	(points) $883 \cdot 10^3$	400 × 500		2.66 GHz Intel CPU	N/A (CPU)
	→ (splats) $81 \cdot 10^3$		1.6		
	(points) $5 \cdot 10^6$				
	→ (splats) $197 \cdot 10^3$		1.4		
Surface point cloud rasterization methods	**Splats**				
[55] 2005	$137 \cdot 10^3$	512 × 512	97	Nvidia	N/A
	$1.1 \cdot 10^6$		14.8	6800 Ultra GPU	(fixed pipeline)
	$4.0 \cdot 10^6$		4.4		
[61] 2010	$128 \cdot 10^3$	1024 × 1024	22	Nvidia	200
	$535 \cdot 10^3$		7.5	GTX 260 GPU	68
	$883 \cdot 10^3$		5		46
[62] 2012	$33 \cdot 10^3$	(approx.) 512 × 512	168	Nvidia Quadro	490
	$471 \cdot 10^3$		76	FX 3800 GPU	220
	$1.0 \cdot 10^6$		28		81
[65] 2018	$14 \cdot 10^6$			Nvidia 12 GB	
	→ (visible) $2 - 8 \cdot 10^6$	1920 × 1080	25–100	Titan X GPU	30–120
Particle-based volume methods (hybrid)	**Dynamic particles**		**Preprocessing time included**		
[97] 2009	$64 \cdot 10^3$	1024 × 768		Nvidia 512 MB	
		(depth map res.) 256 × 192	57	8800 GTS GPU	740
		(depth map res.) 512 × 384	51		660
		(depth map res.) 1024 × 768	20		260
[98] 2010	$16 \cdot 10^3$	(approx.) 1024 × 1024	53	Nvidia 512 MB	1000
	$255 \cdot 10^3$		10	8800 GT GPU	200
	$16 \cdot 10^3$		97	Nvidia 1 GB	880
	$255 \cdot 10^3$		18	GTX 280 GPU	160
[60] 2010	$20 - 64 \cdot 10^3$	1280 × 720	71–83	Nvidia	570–660
	(with foam) $20 - 64 \cdot 10^3$		59–67	GTX 280 GPU	470–540
[99] 2014	$100 \cdot 10^3$	640 × 480	79	Nvidia	19
		1024 × 768	33	GTX Titan GPU	20
[101] 2014	$500 \cdot 10^6$			Nvidia 6 GB	
	→ (bricks) 1919	1280 × 720	15	GTX Titan GPU	9.1
	$180 \cdot 10^6$				
	→ (bricks) 3175		11		6.7
[100] 2016	$40 \cdot 10^3$	640 × 480	76	Nvidia	18
	$125 \cdot 10^3$		222	GTX Titan GPU	79
	$4 \cdot 10^6$	1024 × 768	48		29
[102] 2016	$500 \cdot 10^3$	1024 × 1024	18	Nvidia 6 GB	15
	$2.5 \cdot 10^6$		4.1	GTX Titan GPU	3.3
	$10 \cdot 10^6$		2.1		1.7
[103] 2017	$780 \cdot 10^3$	512 × 512	106	Nvidia	35
	$2 \cdot 10^6$	1024 × 1024	19	GTX 970 GPU	25
	$14 \cdot 10^6$	1920 × 1920	0.42		1.9
[104] 2017	$52 \cdot 10^3$	(not disclosed)	120	4 × server node:	N/A
	$102 \cdot 10^3$		101	2 × Nvidia	(server multi-device)
	$231 \cdot 10^3$		80	GTX 480 GPU	
[106] 2018	(static particles) $750 \cdot 10^3$	1024 × 1024	23	Nvidia 4 GB	50
	$1 \cdot 10^6$	1440 × 1440	4	GTX 960 GPU	17
	$2 \cdot 10^6$	1920 × 1920	2		15
[52] 2018	$6.7 \cdot 10^6$			Nvidia	
	→ (froxels) $34 \cdot 10^9$	1920 × 1080	2.0	GTX 970 GPU	5.2
	$7.1 \cdot 10^6$				
	→ (froxels) $34 \cdot 10^9$		1.6		4.1
	$6.7 \cdot 10^6$			Nvidia	
	→ (froxels) $34 \cdot 10^9$		1.1	GTX Titan GPU	1.8
	$7.1 \cdot 10^6$				
	→ (froxels) $34 \cdot 10^9$		0.68		1.0
Point-based neural rendering methods	**Points**		**Inference time (ms)**		
[107] 2018	(pixels as input) N/A	1920 × 1080	1356	Nvidia Titan X GPU	N/A
[108] 2019	(pixel buffers as input) N/A	512 × 512	330	Nvidia Titan V	N/A
[109] 2019	(splats) $\sim 10^6$	1920 × 1080	62	GeForce RTX 2080 Ti	N/A
[114] 2021	(sphere representations) $\sim 10^6$	1000 × 1000	23.7	GeForce RTX 2080	N/A
[115] 2021	10^7	1920 × 1080	5.7	GeForce RTX 2080 Ti	N/A

REFERENCES

[1] M. Levoy and T. Whitted. (1985). *The Use Points as a Display Primitive*. Citeseer. [Online]. Available: https://graphics.stanford.edu/papers/points/

[2] S. Rusinkiewicz and M. Levoy, "QSplat: A multiresolution point rendering system for large meshes," in *Proc. 27th Annu. Conf. Comput. Graph. Interact. Techn. (SIGGRAPH)*, 2000, pp. 343–352.

[3] R. Preiner, S. Jeschke, and M. Wimmer, "Auto splats: Dynamic point cloud visualization on the GPU," in *Proc. Eurographics Symp. Parallel Graph. Visualizat.*, H. Childs, T. Kuhlen, and F. Marton, Eds. The Eurographics Association, 2012, pp. 139–148.

[4] G. Schaufler and H. Jensen, "Ray tracing point sampled geometry," in *Proc. 11th Eurogr. Workshop Rendering*, 2000, pp. 319–328.

[5] I. Wald and H.-P. Seidel, "Interactive ray tracing of point-based models," in *Proc. Eurographics/IEEE VGTC Symp. Point-Based Graph.*, Jun. 2005, pp. 9–16.

[6] M. Schutz, K. Krosl, and M. Wimmer, "Real-time continuous level of detail rendering of point clouds," in *Proc. IEEE Conf. Virtual Reality 3D User Interfaces (VR)*, Mar. 2019, pp. 103–110.

[7] P. Stotko, "State of the art in real-time registration of RGB-D images," in *Central European Seminar on Computer Graphics for Students*. Smolenice, Slovakia: Tim Golla, 2016, pp. 155–170.

[8] E. E. Hitomi, J. V. Silva, and G. C. Ruppert, "3D scanning using RGBD imaging devices: A survey," in *Developments in Medical Image Processing and Computational Vision*. Cham, Switzerland: Springer, 2015, pp. 379–395.

[9] C. Richardt, J. Tompkin, and G. Wetzstein, "Capture, reconstruction, and representation of the visual real world for virtual reality," in *Real VR Immersive Digital Reality*. Cham, Switzerland: Springer, 2020, pp. 3–32.

[10] R. A. Newcombe, S. Izadi, O. Hilliges, D. Molyneaux, D. Kim, A. J. Davison, P. Kohi, J. Shotton, S. Hodges, and A. Fitzgibbon, "KinectFusion: Real-time dense surface mapping and tracking," in *Proc. 10th IEEE Int. Symp. Mixed Augmented Reality*, Oct. 2011, pp. 127–136.

[11] M. Dou, S. Khamis, Y. Degtyarev, P. Davidson, S. R. Fanello, A. Kowdle, S. O. Escolano, C. Rhemann, D. Kim, J. Taylor, P. Kohli, V. Tankovich, and S. Izadi, "Fusion4D: Real-time performance capture of challenging scenes," *ACM Trans. Graph.*, vol. 35, no. 4, pp. 1–13, Jul. 2016.

[12] T. Whelan, M. Kaess, M. Fallon, H. Johannsson, J. Leonard, and J. McDonald, "Kintinuous: Spatially extended KinectFusion," in *Proc. AAAI*, 2012, pp. 1–10.

[13] R. A. Newcombe, D. Fox, and S. M. Seitz, "DynamicFusion: Reconstruction and tracking of non-rigid scenes in real-time," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2015, pp. 343–352.

[14] S. Schwarz, M. Preda, V. Baroncini, M. Budagavi, P. Cesar, P. A. Chou, R. A. Cohen, M. Krivokuca, S. Lasserre, Z. Li, and J. Llach, "Emerging MPEG standards for point cloud compression," *IEEE J. Emerg. Sel. Topics Circuits Syst.*, vol. 9, no. 1, pp. 133–148, Mar. 2019.

[15] C. G. L. Stanford. (2014). *The Stanford 3D Scanning Repository*. [Online]. Available: https://graphics.stanford.edu/data/3Dscanrep/

[16] S. Orts-Escolano, C. Rhemann, S. Fanello, W. Chang, A. Kowdle, Y. Degtyarev, D. Kim, P. L. Davidson, S. Khamis, M. Dou, and V. Tankovich, "Holoportation: Virtual 3D teleportation in real-time," in *Proc. 29th Annu. Symp. User Interface Softw. Technol.*, Oct. 2016, pp. 741–754.

[17] P. Stotko, S. Krumpen, M. Weinmann, and R. Klein, "Efficient 3D reconstruction and streaming for group-scale multi-client live telepresence," in *Proc. IEEE Int. Symp. Mixed Augmented Reality (ISMAR)*, Oct. 2019, pp. 19–25.

[18] P. Stotko, S. Krumpen, M. B. Hullin, M. Weinmann, and R. Klein, "SLAMCast: Large-scale, real-time 3D reconstruction and streaming for immersive multi-client live telepresence," *IEEE Trans. Vis. Comput. Graphics*, vol. 25, no. 5, pp. 2102–2112, May 2019.

[19] M. Berger, A. Tagliasacchi, L. M. Seversky, P. Alliez, J. A. Levine, A. Sharf, and C. T. and Silva, "State of the art in surface reconstruction from point clouds," in *Eurographics*, S. Lefebvre and M. Spagnuolo, Eds. Geneva, Switzerland: The Eurographics Association, 2014, pp. 1–27.

[20] M. Berger, A. Tagliasacchi, L. M. Seversky, P. Alliez, G. Guennebaud, J. A. Levine, A. Sharf, and C. T. Silva, "A survey of surface reconstruction from point clouds," *Comput. Graph. Forum*, vol. 36, no. 1, pp. 301–329, 2017.

[21] K. Khanna and N. Rajpal, "Survey of curve and surface reconstruction algorithms from a set of unorganized points," in *Proc. 3rd Int. Conf. Soft Comput. Problem Solving*. New Delhi, India: Springer, 2014, pp. 451–458.

[22] S. P. Lim and H. Haron, "Surface reconstruction techniques: A review," *Artif. Intell. Rev.*, vol. 42, no. 1, pp. 59–78, Jun. 2014.

[23] P. Musialski, P. Wonka, D. G. Aliaga, M. Wimmer, L. van Gool, and W. Purgathofer, "A survey of urban reconstruction," *Comput. Graph. Forum*, vol. 32, no. 6, pp. 146–177, Sep. 2013.

[24] G. Pintore, C. Mura, F. Ganovelli, L. Fuentes-Perez, R. Pajarola, and E. Gobbetti, "State-of-the-art in automatic 3D reconstruction of structured indoor environments," *Comput. Graph. Forum*, vol. 39, no. 2, pp. 667–699, May 2020.

[25] H. Harms, J. Beck, J. Ziegler, and C. Stiller, "Accuracy analysis of surface normal reconstruction in stereo vision," in *Proc. IEEE Intell. Vehicles Symp.*, Jun. 2014, pp. 730–736.

[26] K. Jordan and P. Mordohai, "A quantitative evaluation of surface normal estimation in point clouds," in *Proc. IEEE/RSJ Int. Conf. Intell. Robots Syst.*, Sep. 2014, pp. 4220–4226.

[27] A. Khatamian and H. R. Arabnia, "Survey on 3D surface reconstruction," *J. Inf. Process. Syst.*, vol. 12, no. 3, pp. 338–357, 2016.

[28] P. Kourtesis, S. Collina, L. A. A. Doumas, and S. E. MacPherson, "Technological competence is a pre-condition for effective implementation of virtual reality head mounted displays in human neuroscience: A technological review and meta-analysis," *Frontiers Hum. Neurosci.*, vol. 13, p. 342, Oct. 2019.

[29] A. Mehrfard, J. Fotouhi, G. Taylor, T. Forster, N. Navab, and B. Fuerst, "A comparative analysis of virtual reality head-mounted display systems," 2019, *arXiv:1912.02913*.

[30] V. Angelov, E. Petkov, G. Shipkovenski, and T. Kalushkov, "Modern virtual reality headsets," in *Proc. Int. Congr. Hum.-Computer Interact., Optim. Robotic Appl. (HORA)*, Jun. 2020, pp. 1–5.

[31] L. Kobbelt and M. Botsch, "A survey of point-based techniques in computer graphics," *Comput. Graph.*, vol. 28, no. 6, pp. 801–814, Dec. 2004.

[32] M. Sainz and R. Pajarola, "Point-based rendering techniques," *Comput. Graph.*, vol. 28, no. 6, pp. 869–879, Dec. 2004.

[33] M. Gross and H. Pfister, *Point-Based Graphics*. San Francisco, CA, USA: Morgan Kaufmann, 2007.

[34] E. Cerezo, F. Pérez, X. Pueyo, F. J. Seron, and F. X. Sillion, "A survey on participating media rendering techniques," *Vis. Comput.*, vol. 21, no. 5, pp. 303–328, Jun. 2005.

[35] B. Huang, J. Zhao, and J. Liu, "A survey of simultaneous localization and mapping," 2019, *arXiv:1909.05214*.

[36] M. Ihmsen, J. Orthmann, B. Solenthaler, A. Kolb, and M. Teschner, "SPH fluids in computer graphics," in *Eurographics*, S. Lefebvre and M. Spagnuolo, Eds. Geneva, Switzerland: The Eurographics Association, 2014.

[37] A. Alhakamy and M. Tuceryan, "Real-time illumination and visual coherence for photorealistic Augmented/Mixed reality," *ACM Comput. Surv.*, vol. 53, no. 3, pp. 1–34, May 2021.

[38] J. Kronander, F. Banterle, A. Gardner, E. Miandji, and J. Unger, "Photorealistic rendering of mixed reality scenes," *Comput. Graph. Forum*, vol. 34, no. 2, pp. 643–665, May 2015.

[39] A. Adamson and M. Alexa, "Ray tracing point set surfaces," in *Proc. Shape Model. Int.*, 2003, pp. 272–279.

[40] M. Wand and W. Straßer, "Multi-resolution point-sample raytracing," in *Proc. Graph. Interface Conf.*, 2003, pp. 139–148.

[41] B. Adams, R. Keiser, M. Pauly, L. J. Guibas, M. Gross, and P. Dutré, "Efficient raytracing of deforming point-sampled surfaces," *Comput. Graph. Forum*, vol. 24, no. 3, pp. 677–684, Sep. 2005.

[42] L. Linsen, K. Müller, and P. Rosenthal, "Splat-based ray tracing of point clouds," *J. WSCG*, vol. 15, nos. 1–3, pp. 51–58, 2007.

[43] E. Hubo, T. Mertens, T. Haber, and P. Bekaert, "The quantized kd-tree: Efficient ray tracing of compressed point clouds," in *Proc. IEEE Symp. Interact. Ray Tracing*, Sep. 2006, pp. 105–113.

[44] E. Hubo, T. Mertens, T. Haber, and P. Bekaert, "Self-similarity-based compression of point clouds, with application to ray tracing," in *Proc. Eurographics Symp. Point-Based Graph.*, 2007, pp. 129–137.

[45] E. Tejada, J. Gois, L. Nonato, A. Castelo, and T. Ertl, "Hardware-accelerated extraction and rendering of point set surfaces," in *Proc. 8th Joint Eurograph./IEEE VGTC Conf. Vis. (EUROVIS)*, May 2006, pp. 21–28.

[46] E. Tejada, T. Schafhitzel, and T. Ertl, "Hardware-accelerated point-based rendering of surfaces and volumes, in *Proc. 15th Int. Conf. Central Eur. Comput. Graph., Vis. Comput. Vision*, 2007, pp. 41–48.

[47] S. Kashyap, R. Goradia, P. Chaudhuri, and S. Chandran, "Real time ray tracing of point-based models," in *Proc. ACM SIGGRAPH Symp. Interact. 3D Graph. Games (I3D)*, 2010, p. 1.

[48] R. Goradia, S. Kashyap, P. Chaudhuri, and S. Chandran, "GPU-based ray tracing of splats," in *Proc. 18th Pacific Conf. Comput. Graph. Appl.*, Sep. 2010, pp. 101–108.

[49] S. Kashyap, R. Goradia, P. Chaudhuri, and S. Chandran, "Implicit surface octrees for ray tracing point models," in *Proc. 7th Indian Conf. Comput. Vis., Graph. Image Process. (ICVGIP)*, 2010, pp. 227–234.

[50] R. Goradia, M. S. S. Kashyap, P. Chaudhuri, and S. Chandran, "Tracing specular light paths in point-based scenes," *Vis. Comput.*, vol. 27, no. 12, pp. 1083–1097, Dec. 2011.

[51] P. Cai, D. Kong, S. Wang, and B. Yin, "A height-difference-based ray tracing of point models," in *Proc. 10th Int. Conf. Virtual Reality Continuum Appl. Ind. (VRCAI)*, 2011, pp. 91–98.

[52] T. Zirr and C. Dachsbacher, "Memory-efficient on-the-fly voxelization and rendering of particle data," *IEEE Trans. Vis. Comput. Graphics*, vol. 24, no. 2, pp. 1155–1166, Feb. 2018.

[53] M. Zwicker, H. Pfister, J. V. Baar, and M. Gross, "Surface splatting," in *Proc. 28th Annu. Conf. Comput. Graph. Interact. Techn.*, 2001, pp. 371–378.

[54] P. Gautron, J. Krivanek, K. Bouatouch, and S. Pattanaik, "Radiance cache splatting: A GPU-friendly global illumination algorithm," in *Proc. Eurographics Symp. Rendering*, 2005, pp. 55–64.

[55] M. Botsch, A. Hornung, M. Zwicker, and L. Kobbelt, "High-quality surface splatting on today's GPUs," in *Proc. Eurogr./IEEE VGTC Symp. Point-Based Graph.*, Jun. 2005, pp. 17–141.

[56] Y. Zhang and R. Pajarola, "Deferred blending: Image composition for single-pass point rendering," *Comput. Graph.*, vol. 31, no. 2, pp. 175–189, Apr. 2007.

[57] M. Wimmer and C. Scheiblauer, "Instant points: Fast rendering of unprocessed point clouds," in *Proc. Symp. Point-Based Graph.*, M. Botsch, B. Chen, M. Pauly, and M. Zwicker, Eds. Geneva, Switzerland: The Eurographics Association, 2006, pp. 129–136.

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

[59] G. Guennebaud, M. Germann, and M. Gross, "Dynamic sampling and rendering of algebraic point set surfaces," *Comput. Graph. Forum*, vol. 27, no. 2, pp. 653–662, 2008.

[60] F. Bagar, D. Scherzer, and M. Wimmer, "A layered particle-based fluid model for real-time rendering of water," *Comput. Graph. Forum*, vol. 29, no. 4, pp. 1383–1389, Aug. 2010.

[61] P. Dobrev, P. Rosenthal, and L. Linsen, "An image-space approach to interactive point cloud rendering including shadows and transparency," *Comput. Graph. Geometry*, vol. 12, no. 3, pp. 2–25, 2010.

[62] H.-J. Kim, A. Cengiz Öztireli, M. Gross, and S.-M. Choi, "Adaptive surface splatting for facial rendering," *Comput. Animation Virtual Worlds*, vol. 23, nos. 3–4, pp. 363–373, May 2012.

[63] C. Günther, T. Kanzok, L. Linsen, and P. Rosenthal, "A GPGPU-based pipeline for accelerated rendering of point clouds," *J. WSCG*, vol. 21, no. 2, pp. 153–162, 2013.

[64] P. Goswami, F. Erol, R. Mukhi, R. Pajarola, and E. Gobbetti, "An efficient multi-resolution framework for high quality interactive rendering of massive point clouds using multi-way kd-trees," *Vis. Comput.*, vol. 29, no. 1, pp. 69–83, Jan. 2012.

[65] A. Kulik, A. Kunert, S. Beck, C.-F. Matthes, A. Schollmeyer, A. Kreskowski, B. Fröhlich, S. Cobb, and M. D'Cruz, "Virtual valcamonica: Collaborative exploration of prehistoric petroglyphs and their surrounding environment in multi-user virtual reality," *Presence: Teleoperators Virtual Environ.*, vol. 26, no. 3, pp. 297–321, May 2018.

[66] J. Lehtinen, M. Zwicker, J. Kontkanen, E. Turquin, F. X. Sillion, and T. Aila, "Meshless finite elements for hierarchical global illumination," Publications Telecommun. Softw. Multimedia, Helsinki Univ. Technol., Espoo, Finland, Tech. Rep. TML-B7, 2007.

[67] J. Lehtinen, M. Zwicker, E. Turquin, J. Kontkanen, F. Durand, F. X. Sillion, and T. Aila, "A meshless hierarchical representation for light transport," *ACM Trans. Graph.*, vol. 27, no. 3, pp. 1–9, Aug. 2008.

[68] M. Zollhöfer and M. Stamminger, "Meshless hierarchical radiosity on the GPU," in *Vision, Modeling, and Visualization*. Geneva, Switzerland: The Eurographics Association, 2011, pp. 331–338.

[69] P. Christensen, "Point-based approximate color bleeding," *Pixar Tech. Notes*, vol. 2, no. 5, pp. 1–9, 2008.

[70] P. Christensen, "Point-based global illumination for movie production," in *Proc. ACM SIGGRAPH Courses*, 2010, pp. 1–19.

[71] C. Gibson and Z. Wood, "An approach to point based approximate color bleeding with volumes," in *Advances in Visual Computing*. Berlin, Germany: Springer, 2011, pp. 441–450.

[72] J. Kontkanen, E. Tabellion, and R. S. Overbeck, "Coherent out-of-core point-based global illumination," *Comput. Graph. Forum*, vol. 30, no. 4, pp. 1353–1360, Jun. 2011.

[73] B. Wang, Y. Xu, and X. Meng, "Progressive point-based global illumination," in *Proc. Workshop Digit. Media Digit. Content Manage.*, May 2011, pp. 181–184.

[74] D. Maletz and R. Wang, "Importance point projection for gpu-based final gathering," in *Proc. 22nd Eurogr. Conf. Rendering*. Goslar, DEU: Eurographics Association, 2011, pp. 1327–1336.

[75] B. Wang, J. Huang, B. Buchholz, X. Meng, and T. Boubekeur, "Factorized point based global illumination," *Comput. Graph. Forum*, vol. 32, no. 4, pp. 117–123, Jul. 2013.

[76] T. Harada, "Micro-buffer rasterization reduction method for environment lighting using point-based rendering," in *Proc. Graph. Interface*, 2014, pp. 95–102.

[77] B. Chang, S. Park, and I. Ihm, "Diffuse global illumination in particle spaces," *Multimedia Tools Appl.*, vol. 74, no. 13, pp. 4987–5006, Jul. 2015.

[78] J. Hanika, P. Hillman, M. Hill, and L. Fascione, "Camera space volumetric shadows," in *Proc. Digit. Prod. Symp.*, New York, NY, USA, 2012, pp. 7–14.

[79] N. Schertler, M. Salm, J. Staib, and S. Gumhold, "Visualization of scanned cave data with global illumination," in *Proc. Workshop Visualisation Environ. Sci. (EnvirVis)*, K. Rink, A. Middel, and D. Zeckzer, Eds. Geneva, Switzerland: The Eurographics Association, 2016.

[80] B. Wang, J.-D. Gascuel, and N. Holzschuch, "Point-based light transport for participating media with refractive boundaries," in *Proc. Eurographics Symp. Rendering Experim. Ideas Implementations*, E. Eisemann and E. Fiume, Eds. Geneva, Switzerland: The Eurographics Association, 2016.

[81] B. Wang and N. Holzschuch, "Point-based rendering for homogeneous participating media with refractive boundaries," *IEEE Trans. Vis. Comput. Graphics*, vol. 24, no. 10, pp. 2743–2757, Oct. 2018.

[82] Y. Liang, B. Wang, L. Wang, and N. Holzschuch, "Fast computation of single scattering in participating media with refractive boundaries using frequency analysis," *IEEE Trans. Vis. Comput. Graphics*, vol. 26, no. 10, pp. 2961–2969, Oct. 2019.

[83] L. Belcour, C. Soler, K. Subr, N. Holzschuch, and F. Durand, "5D covariance tracing for efficient defocus and motion blur," *ACM Trans. Graph.*, vol. 32, no. 3, pp. 1–18, Jun. 2013.

[84] X. Xu, P. Wang, B. Wang, L. Wang, C. Tu, X. Meng, and T. Boubekeur, "Efficient point based global illumination on Intel MIC architecture," in *Eurographics*, L. G. Magalhaes and R. Mantiuk, Eds. Geneva, Switzerland: The Eurographics Association, 2016.

[85] X. Xu, B. Wang, L. Wang, Y. Xu, and T. Boubekeur, "Vectorized point based global illumination on Intel MIC architecture," *Comput. Graph.*, vol. 70, pp. 206–213, Feb. 2018.

[86] H. Halén and K. Hayward, "Global illumination based on surfels," in *Proc. ACM SIGGRAPH Symp. Interactive 3D Graph. Games*, 2021, pp. 1399–1405. [Online]. Available: https://www.ea.com/seed/news/siggraph21-global-illumination-surfels

[87] T. Ritschel, T. Grosch, M. H. Kim, H.-P. Seidel, C. Dachsbacher, and J. Kautz, "Imperfect shadow maps for efficient computation of indirect illumination," *ACM Trans. Graph.*, vol. 27, no. 5, pp. 1–8, Dec. 2008.

[88] T. Ritschel, T. Engelhardt, T. Grosch, H.-P. Seidel, J. Kautz, and C. Dachsbacher, "Micro-rendering for scalable, parallel final gathering," in Proc. *ACM SIGGRAPH Asia Papers*. New York, NY, USA, 2009, pp. 1–8.

[89] M. Knecht, C. Traxler, O. Mattausch, W. Purgathofer, and M. Wimmer, "Differential instant radiosity for mixed reality," in *Proc. IEEE Int. Symp. Mixed Augmented Reality*, Oct. 2010, pp. 99–107.

[90] M. Knecht, C. Traxler, O. Mattausch, and M. Wimmer, "Reciprocal shading for mixed reality," *Comput. Graph.*, vol. 36, no. 7, pp. 846–856, Nov. 2012.

[91] P. Mavridis and G. Papaioannou, "Global illumination using imperfect volumes," in *Proc. Int. Conf. Comput. Graph. Theory Appl. (VISIGRAPP)*, 2011, pp. 160–165.

[92] C. Wyman, "Voxelized shadow volumes," in *Proc. ACM SIGGRAPH Symp. High Perform. Graph.*, 2011, pp. 33–40.

[93] C. Wyman and Z. Dai, "Imperfect voxelized shadow volumes," in *Proc. ACM SIGGRAPH Talks (SIGGRAPH)*, New York, NY, USA, 2013, pp. 45–52.

[94] C. Weinzierl-Heigl, "Efficient VAL-based real-time global illumination," in *Proc. 17th Central Eur. Seminar Comput. Graph.*, 2013, pp. 1–8.

[95] T. Ritschel, E. Eisemann, I. Ha, J. D. K. Kim, and H.-P. Seidel, ''Making imperfect shadow maps view-adaptive: High-quality global illumination in large dynamic scenes,'' *Comput. Graph. Forum*, vol. 30, no. 8, pp. 2258–2269, Dec. 2011.
[96] T. Barák, J. Bittner, and V. Havran, ''Temporally coherent adaptive sampling for imperfect shadow maps,'' *Comput. Graph. Forum*, vol. 32, no. 4, pp. 87–96, Jul. 2013.
[97] W. J. van der Laan, S. Green, and M. Sainz, ''Screen space fluid rendering with curvature flow,'' in *Proc. Symp. Interact. 3D Graph. Games*, New York, NY, USA, 2009, pp. 91–98.
[98] P. Goswami, P. Schlegel, B. Solenthaler, and R. Pajarola, ''Interactive SPH simulation and rendering on the GPU,'' in *Proc. ACM SIGGRAPH/Eurographics Symp. Comput. Animation*, 2010, pp. 55–64.
[99] T. Imai, Y. Kanamori, Y. Fukui, and J. Mitani, ''Real-time screen-space liquid rendering with two-sided refractions,'' in *Proc. NICOGRAPH Int.*, 2014, pp. 71–76.
[100] T. Imai, Y. Kanamori, and J. Mitani, ''Real-time screen-space liquid rendering with complex refractions: Real-time rendering of liquids with up to four refractions,'' *Comput. Animation Virtual Worlds*, vol. 27, pp. 425–434, May 2016.
[101] F. Reichl, M. G. Chajdas, J. Schneider, and R. Westermann, ''Interactive rendering of giga-particle fluid simulations,'' in *Proc. Eurographics/ACM SIGGRAPH Symp. High Perform. Graph.*, I. Wald and J. Ragan-Kelley, Eds. Geneva, Switzerland: The Eurographics Association, 2014, pp. 105–116.
[102] H. Hochstetter, J. Orthmann, and A. Kolb, ''Adaptive sampling for on-the-fly ray casting of particle-based fluids,'' in *Proc. High Perform. Graph.* Goslar, DEU: Eurographics Association, 2016, pp. 129–138.
[103] X. Xiao, S. Zhang, and X. Yang, ''Real-time high-quality surface rendering for large scale particle-based fluids,'' in *Proc. 21st ACM SIGGRAPH Symp. Interact. 3D Graph. Games*, Feb. 2017, pp. 1–8.
[104] F. Q. Zhang, Z. W. Wang, J. Chang, J. Zhang, and F. Tian, ''A fast framework construction and visualization method for particle-based fluid,'' *EUPASIP J. Image Video Process.*, vol. 2017, no. 1, p. 79, 2017.
[105] S. G. Parker, J. Bigler, A. Dietrich, H. Friedrich, J. Hoberock, D. Luebke, D. McAllister, M. McGuire, K. Morley, A. Robison, and M. Stich, ''OptiX: A general purpose ray tracing engine,'' *ACM Trans. Graph.*, vol. 29, no. 4, 2010.
[106] C. J. dos Santos Brito, A. L. B. V. e Silva, J. M. Teixeira, and V. Teichrieb, ''Ray tracer based rendering solution for large scale fluid rendering,'' *Comput. Graph.*, vol. 77, pp. 65–79, Dec. 2018.
[107] G. Bui, T. Le, B. Morago, and Y. Duan, ''Point-based rendering enhancement via deep learning,'' *Vis. Comput.*, vol. 34, nos. 6–8, pp. 829–841, Jun. 2018.
[108] M. Meshry, D. B. Goldman, S. Khamis, H. Hoppe, R. Pandey, N. Snavely, and R. Martin-Brualla, ''Neural rerendering in the wild,'' in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2019, pp. 6878–6887.
[109] K.-A. Aliev, A. Sevastopolsky, M. Kolos, D. Ulyanov, and V. Lempitsky, ''Neural point-based graphics,'' 2019, *arXiv:1906.08240*.
[110] P. Dai, Y. Zhang, Z. Li, S. Liu, and B. Zeng, ''Neural point cloud rendering via multi-plane projection,'' in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2020, pp. 7830–7839.
[111] B. Mildenhall, P. P. Srinivasan, M. Tancik, J. T. Barron, R. Ramamoorthi, and R. Ng, ''Nerf: Representing scenes as neural radiance fields for view synthesis,'' in *Proc. Eur. Conf. Comput. Vis.* Cham, Switzerland: Springer, 2020, pp. 405–421.
[112] G. Kopanas, J. Philip, T. Leimkühler, and G. Drettakis, ''Point-based neural rendering with per-view optimization,'' in *Proc. Comput. Graph. Forum Eurographics Symp. Rendering*, vol. 40, no. 4, Jun. 2021, pp. 29–43. [Online]. Available: https://www-sop.inria.fr/reves/Basilic/2021/KPLD21
[113] M. Goesele, N. Snavely, B. Curless, H. Hoppe, and S. M. Seitz, ''Multi-view stereo for community photo collections,'' in *Proc. IEEE 11th Int. Conf. Comput. Vis.*, Oct. 2007, pp. 1–8.
[114] C. Lassner and M. Zollhofer, ''Pulsar: Efficient sphere-based neural rendering,'' in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2021, pp. 1440–1449.
[115] D. Rückert, L. Franke, and M. Stamminger, ''ADOP: Approximate differentiable one-pixel point rendering,'' 2021, *arXiv:2110.06635*.
[116] M. Schütz, B. Kerbl, and M. Wimmer, ''Rendering point clouds with compute shaders and vertex order optimization,'' *Comput. Graph. Forum*, vol. 40, no. 4, pp. 115–126, Jul. 2021.
[117] T. Karras, ''Maximizing parallelism in the construction of BVHs, octrees, and k-d trees,'' in *Eurographics / ACM SIGGRAPH Symp. High Perform. Graph.* Goslar, DEU: Eurographics Association, 2012, pp. 33–37.
[118] Nvidia. (2018). *Nvidia Turing GPU Architecture*. [Online]. Available: https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf
[119] Nvidia. (2021). *Nvidia Ampere GA102 GPU Architecture*. [Online]. Available: https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf
[120] Y. Deng, Y. Ni, Z. Li, S. Mu, and W. Zhang, ''Toward real-time ray tracing: A survey on hardware acceleration and microarchitecture techniques,'' *ACM Comput. Surv.*, vol. 50, no. 4, pp. 1–41, Jul. 2017.
[121] E. Gobbetti and F. Marton, ''Layered point clouds: A simple and efficient multiresolution structure for distributing and rendering gigantic point-sampled models,'' *Comput. Graph.*, vol. 28, no. 6, pp. 815–826, Dec. 2004.
[122] B. Buchholz and T. Boubekeur, ''Quantized point-based global illumination,'' *Comput. Graph. Forum*, vol. 31, no. 4, pp. 1399–1405, Jun. 2012.
[123] K. Zhou, Q. Hou, R. Wang, and B. Guo, ''Real-time KD-tree construction on graphics hardware,'' *ACM Trans. Graph.*, vol. 27, no. 5, pp. 1–11, Dec. 2008.
[124] K. Museth, ''VDB: high-resolution sparse volumes with dynamic topology,'' *ACM Trans. Graph.*, vol. 32, no. 3, pp. 1–22, Jun. 2013.
[125] D. Neumann, F. Lugauer, S. Bauer, J. Wasza, and J. Hornegger, ''Real-time RGB-D mapping and 3-D modeling on the GPU using the random ball cover data structure,'' in *Proc. IEEE Int. Conf. Comput. Vis. Workshops (ICCV Workshops)*, Nov. 2011, pp. 1161–1167.
[126] T. Viitanen, M. Koskela, P. Jääskeläinen, H. Kultala, and J. Takala, ''MergeTree: A fast hardware HLBVH constructor for animated ray tracing,'' *ACM Trans. Graph.*, vol. 36, no. 5, pp. 1–14, Oct. 2017.
[127] D. Lin, E. Vasiou, C. Yuksel, D. Kopta, and E. Brunvand, ''Hardware-accelerated dual-split trees,'' *Proc. ACM Comput. Graph. Interact. Tech.*, vol. 3, no. 2, pp. 20:1–20:21, 2020.
[128] Techpowerup. *GPU Specs Database*. Accessed: Dec. 1, 2021. [Online]. Available: https://www.techpowerup.com/gpu-specs
[129] R. K. Hoetzlein, ''GVDB: Raytracing sparse voxel database structures on the GPU,'' in *Eurographics/ ACM SIGGRAPH Symp. High Perform. Graph.*, U. Assarsson and W. Hunt, Eds. Geneva, Switzerland: The Eurographics Association, 2016, pp. 109–117.
[130] Nvidia. (2008). *Nvidia GeForce GTX 200 GPU Architectural Overview*. [Online]. Available: https://www.nvidia.com/docs/IO/55506/GeForce_GTX_200_GPU_Technical_Brief.pdf
[131] D. Ströter, J. S. Mueller-Roemer, A. Stork, and D. W. Fellner, ''OLBVH: Octree linear bounding volume hierarchy for volumetric meshes,'' *Vis. Comput.*, vol. 36, nos. 10–12, pp. 2327–2340, Oct. 2020.

PETRUS E. J. KIVI received the master's degree in mathematics from Tampere University, in 2020. He is currently a Researcher at Tampere University. His research interests include stereoscopic rendering and point cloud rendering.

MARKKU J. MÄKITALO received the Ph.D. degree in signal processing from the Tampere University of Technology (TUT), in 2013. He is currently a Postdoctoral Researcher at Tampere University. His research interests include photorealistic real-time multi-view rendering, low-latency image and video compression, and immersive rendering technologies, such as light field rendering.

JAKUB ŽÁDNÍK received the master's degree in engineering from the University of Brno, in 2017. He is currently pursuing the Ph.D. degree with Tampere University. His research interests include low-latency image and video compression, and parallel computing.

JULIUS IKKALA received the master's degree in information technology from Tampere University, in 2021. He is currently a Researcher at Tampere University. His research interests include photorealistic real-time rendering and ray tracing.

VINOD KUMAR MALAMAL VADAKITAL received the B.E. degree in computer science and engineering from Bangalore University, Bengaluru, India, in 1998, and the M.S. degree in information technology and the Ph.D. degree in signal processing from the Tampere University of Technology, Tampere, Finland, in 2005 and 2012, respectively. He is currently a Bell Labs Distinguished Member of Technical Staff and a Principal Researcher with the Media Technology Research Team, Nokia Technologies, Tampere. His research interests include the domains of video signal processing, computer vision, and XR technologies. He has previously been an Editor of the *High Efficiency Image File Format* (HEIF) version 1 specification and is also an Editor of the *MPEG Immersive Video Specification*.

PEKKA O. JÄÄSKELÄINEN is currently an Associate Professor at Tampere University. He has researched heterogeneous platform customization and programming since the early 2000s. He is also an active contributor to various heterogeneous parallel platform related open source projects. Related to his work on implementing challenging real-time applications, he is interested in next generation distributed architectures for photorealistic real-time rendering.